ACSL By Example

Towards a Formally Verified Standard Library

Version 18.0.0
for
Frama-C 18 (Argon)
December 2018

Jens Gerlach
Tim Sikatzki

Former Authors

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. You may use the GitHub issue tracker[4] to report suggestions or errors. Alternatively, you may email them 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] `https://github.com/fraunhoferfokus/acsl-by-example/issues`

Changes

This release is intended for Frama-C 18 (*Argon*), issued in November 2018.[5]

For changes in previous versions we refer to Appendix B on Page 213.

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.

[5] See `https://frama-c.com/download.html`

Contents

List of logic definitions

List of Figures

List of Tables

1. 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 [1]) and the Frama-C/WP plug-in [2] of Frama-C [3] (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[6] 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, Loïc Correnson, Zaynah Dargaye, Florent Kirchner, Virgile Prevosto, and Armand Puccetti from CEA LIST[7] and Pascal Cuoq from TrustInSoft[8].

We also like to express our gratitude to Claude Marché (LRI/INRIA)[9] and Yannick Moy (AdaCore)[10] 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.

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

1.1. Frama-C

The Framework for Modular Analyses of C, Frama-C [3], 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 [4], a laboratory of applied research on software-intensive technologies, and INRIA Saclay [5], the French National Institute for Research in Computer Science and Control in collaboration with LRI [6], the Laboratory for Computer Science at Université Paris-Sud.

ACSL (ANSI/ISO-C Specification Language) [1] 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 [2] 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 [7] 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 [8] first. The principles presented in this paper are also documented in the ACSL reference document [9].

1.2. 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 2 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 [10].

We have grouped various standard algorithms in chapters as follows:

- non-mutating algorithms (Chapter 3)
- maximum/minimum algorithms (Chapter 4)
- binary search algorithms (Chapter 5)
- mutating algorithms (Chapter 6)
- numeric algorithms (Chapter 7)
- heap algorithms (Chapter 8)
- sorting algorithms (Chapter 9)

The order of these chapters reflects their increasing complexity.

In Chapter 10, we present several well-known classical implementations of sorting algorithms, all of which share the same contract.

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.

1.3. 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.

1.3.1. Array and ranges

The C Standard describes an array as a "contiguously allocated nonempty set of objects" [11, §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.

1.3.2. 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`.

2. The Hoare calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [12], 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 [9, §2.2] in full detail. For the purposes of this tutorial, the notions shown in Table 2.1 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 2.1.: 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.[11] 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 2.2 to 2.12 in order to reason about programs. We will comment on them in the following sections.

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

2.1. 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 2.2, 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 2.2: 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 2.3 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 2.3: 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 2.3, 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 2.4.

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

Listing 2.4: Simpler, but *faulty* assignment rule

Listings 2.5–2.7 show some example instances of this faulty rule.

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

Listing 2.5: An example instance of the faulty rule from Listing 2.4

While Listing 2.5 happens to be ok, Listing 2.6 and 2.7 lead to postconditions that are obviously nonsensical formulas.

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

Listing 2.6: An example instance of the faulty rule from Listing 2.4

The reason is that in the assignment in Listing 2.6 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Listing 2.7 just destroys the property from its precondition.

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

Listing 2.7: An example instance of the faulty rule from Listing 2.4

Note that the correct example Listing 2.5 can as well be obtained as an instance of the correct rule from Listing 2.2, 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`.

2.2. The sequence rule

The sequence rule, shown in Listing 2.8, 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;
```

$\leadsto$

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

Listing 2.8: The sequence rule

2.3. The implication rule

The implication rule, shown in Listing 2.9, 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 2.9 by the right one.

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

$\leadsto$

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

Listing 2.9: The implication rule

2.4. The choice rule

The choice rule, depicted in Listing 2.10, 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;
```

$\leadsto$

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

Listing 2.10: The choice rule

Listing 2.11 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 2.11: 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.

2.5. The loop rule

The loop rule, shown in Listing 2.12, 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 2.12: 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 2.13 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 2.11 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 2.13: An abstract ring buffer loop

To reuse the proof from Listing 2.11, we had to drop the conjunct `!done`, since we didn't consider it in Listing 2.11. 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 2.11, 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 2.11. 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 Section 6.4 on Page 81.

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`[12] 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 2.11 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.

[12] 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`.

2.6. 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 [13] 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`.

3. Non-mutating algorithms

In this chapter, we consider *non-mutating* algorithms of the C++ Standard Library [14, §25.2]. 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` (Section 3.1 on Page 28) provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a given range. In Section 3.2, on Page 30, a predicate is introduced in order to simplify the specification.
- `find_first_of` (Section 3.3, on Page 32) 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` (Section 3.4 on Page 34) can be used to find equal neighbors in an array.
- `equal` and `mismatch` (Section 3.5 on Page 36) are useful for comparing two ranges element-by-element and identifying where they differ.
- `search` and `search_n` (Sections 3.6 and 3.7) find a subsequence that is identical to a given sequence when compared element-by-element and returns the position of the first occurrence.
- `count` (Section 3.9, on Page 48) returns the number of occurrences of a given value in a range. Here we will employ some user-defined axioms to formally specify `count`.

3.1. The `find` algorithm

The `find` algorithm in the C++ Standard Library [14, §25.2.5] 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 val);
```

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

As an example, we consider in Figure 3.1 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.

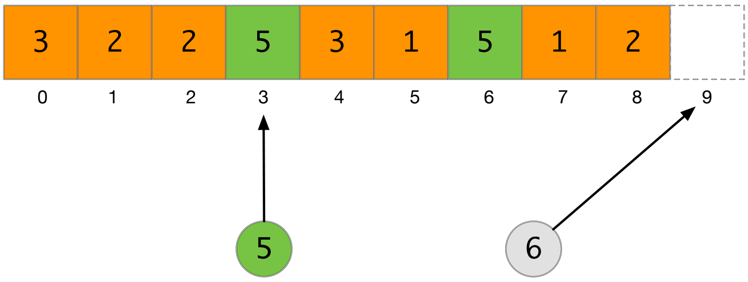

Figure 3.1.: Some simple examples for `find`

3.1.1. Formal specification of `find`

The formal specification of `find` in ACSL is shown in Listing 3.2.

```
/*@
  requires   \valid_read(a + (0..n-1));

  assigns    \nothing;

  ensures    0 <= \result <= \result;

  behavior some:
    assumes   \exists integer i; 0 <= i < n && a[i] == val;
    ensures   0 <= \result < n;
    ensures   a[\result] == val;
    ensures   \forall integer i; 0 <= i < \result ==> a[i] != val;

  behavior none:
    assumes   \forall integer i; 0 <= i < n ==> a[i] != val;
    ensures   \result == n;

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

Listing 3.2: 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 Section 1.3). The assigns-clause indicates that find (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 27).

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] == val 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] == val 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 Listing 3.2). 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*.

3.1.2. Implementation of **find**

Listing 3.3 shows a straightforward implementation of find. The only noteworthy elements of this implementation are the *loop annotations*.

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

  return n;
}
```

Listing 3.3: Implementation of find

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 (see Listing 3.2). 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.

3.2. The `find` algorithm reconsidered

In this section we specify the `find` algorithm in a slightly different way when compared to Section 3.1. Our approach is motivated by a considerable number of closely related formulas. We have in Listings 3.2 and 3.3 the following formulas

`\exists integer i; 0 <= i < n          &&  a[i] == val;`
`\forall integer i; 0 <= i < \result ==> a[i] != val;`
`\forall integer i; 0 <= i < n         ==> a[i] != val;`
`\forall integer k; 0 <= k < i         ==> a[k] != val;`

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

3.2.1. The predicate `HasValue`

In order to be more explicit about the commonalities of these formulas we define a predicate, called `HasValue` (see Listing 3.4), which describes the situation that there is a valid index `i` where `a[i]` equals `val`.

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

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

Listing 3.4: The predicate `HasValue`

Note that we needed to provide a label, viz. `A`, to the predicate, since the evaluation 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 `HasValue`, the label `A` is the only available label.

With this predicate we can encapsulate all uses of the universal and existential quantifiers in both the function contract of `find` and in its loop annotations. The result is shown in Listings 3.5 and 3.6.

3.2.2. Formal specification of `find`

Th revised contract for `find` in Listing 3.5 is more concise than the previous one in Listing 3.2. 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 `HasValue` predicate, as depicted with the last postcondition of behavior `some` shown in Listing 3.5.

```
/*@
  requires  valid:  \valid_read(a + (0..n-1));

  assigns   \nothing;

  ensures   result: 0 <= \result <= n;

  behavior some:
    assumes  HasValue(a, n, val);
    ensures  bound:   0 <= \result < n;
    ensures  result:  a[\result] == val;
    ensures  first:   !HasValue(a, \result, val);

  behavior none:
    assumes  !HasValue(a, n, val);
    ensures  result:  \result == n;

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

Listing 3.5: Formal specification of `find` using the `HasValue` predicate

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.

3.2.3. Implementation of `find`

The predicate `HasValue` is also used in the loop annotation inside the implementation of `find`. Note that, as in the case of the specification, we use labels to name individual annotations.

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

  return n;
}
```

Listing 3.6: Implementation of `find` with loop annotations based on `HasValue`

3.3. The `find_first_of` algorithm

The `find_first_of` algorithm [14, §25.2.7] is closely related to `find` (see Sections 3.1 and 3.2).

```
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]`.

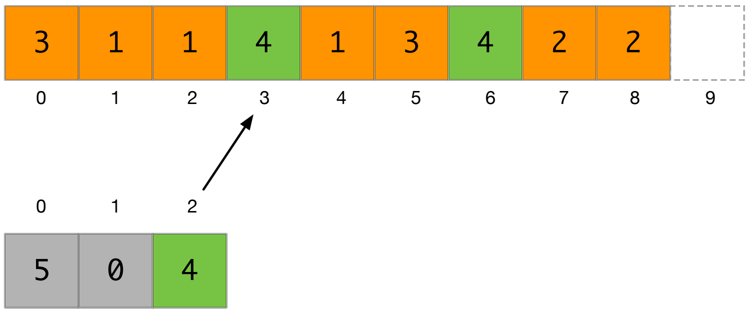

Figure 3.7.: A simple example for `find_first_of`

As an example, we consider in Figure 3.7 two arrays. The arrow indicates the smallest index where one of the elements of the three-element array occurs.

3.3.1. The predicate `HasValueOf`

Similar to our approach in Section 3.2, we define a predicate `HasValueOf` 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 `HasValue` (Listing 3.4) to define `HasValueOf` (Listing 3.8).

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

Listing 3.8: The predicate `HasValueOf`

3.3.2. Formal specification of `find_first_of`

The formal specification of `find_first_of` is shown Listing 3.9. The function contract uses the predicates `HasValueOf` and `HasValue` thereby making it very similar the specification `find` (Listing 3.5).

```
/*@
  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);
    ensures bound:  0 <= \result < m;
    ensures result: HasValue(b, n, a[\result]);
    ensures first:  !HasValueOf(a, \result, b, n);

  behavior not_found:
    assumes !HasValueOf(a, m, b, n);
    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 3.9: Formal specification of find_first_of

3.3.3. Implementation of find_first_of

Our implementation of find_first_of is shown in Listing 3.10.

```
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 = 0; i < m; i++) {
    if (find(b, n, a[i]) < n) {
      return i;
    }
  }

  return m;
}
```

Listing 3.10: Implementation of find_first_of

Note the call of the find function. We opted for an implementation of find_first_of that emphasizes reuse. Besides, leading to a more concise implementation, we also have to write fewer loop annotations.

3.4. The `adjacent_find` algorithm

The `adjacent_find` algorithm of the C++ Standard Library [14, §25.2.8]

```
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 3.11 indicates the smallest index where two adjacent elements are equal.

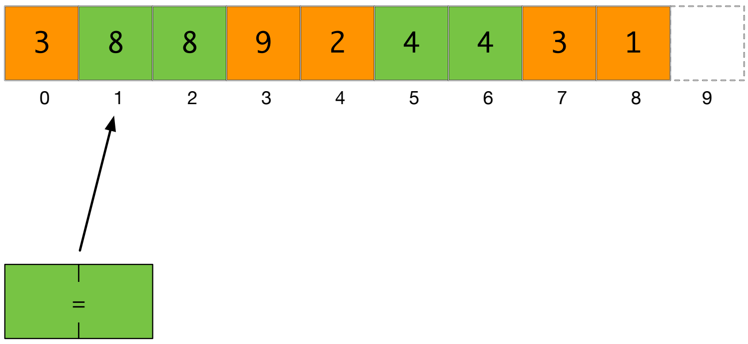

Figure 3.11.: A simple example for `adjacent_find`

3.4.1. The predicate `HasEqualNeighbors`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` (see Listing 3.12) that captures the essence of finding two adjacent indices at which the array holds equal values.

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

Listing 3.12: The predicate `HasEqualNeighbors`

3.4.2. Formal specification of `adjacent_find`

We use the predicate `HasEqualNeighbors` to define the formal specification of `adjacent_find` (see Listing 3.13).

```
/*@
  requires valid: \valid_read(a + (0..n-1));

  assigns \nothing;

  ensures result:  0 <= \result <= n;

  behavior some:
    assumes  HasEqualNeighbors(a, n);
    ensures  result:    0 <= \result < n-1;
    ensures  adjacent:  a[\result] == a[\result+1];
    ensures  first:     !HasEqualNeighbors(a, \result);

  behavior none:
    assumes  !HasEqualNeighbors(a, n);
    ensures  result:    \result == n;

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

Listing 3.13: Formal specification of `adjacent_find`

3.4.3. Implementation of `adjacent_find`

The implementation of `adjacent_find`, including loop annotations is shown in Listing 3.14. At the beginning we check whether the array contains at least two elements. Otherwise, there is no point in looking for adjacent neighbors . . .

```
size_type
adjacent_find(const value_type* a, size_type n)
{
  if (n > 1u) {
    /*@
      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 3.14: Implementation of `adjacent_find`

Note the use of the predicate `HasEqualNeighbors` in the loop invariant to match the similar postcondition of behavior `some`.

3.5. The `equal` and `mismatch` algorithms

The `equal` [14, §25.2.11] and `mismatch` [14, §25.2.10] algorithms in 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.

3.5.1. The `EqualRanges` 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.[13] 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 in Listing 3.15 several *overloaded* versions of the predicate `EqualRanges`.

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

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

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

Listing 3.15: Overloaded versions of predicate `EqualRanges`

The letters `K` and `L` in the definition of `EqualRanges` are so-called *labels*[14] 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`, 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 [9, §2.6.9].

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

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

Using predicate `EqualRanges` we can formulate the specification of `equal` in Listing 3.16, 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 `EqualRanges` 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 <==> EqualRanges{Here,Here}(a, n, b);
*/
bool equal(const value_type* a, size_type n, const value_type* b);
```

Listing 3.16: Formal specification of `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 EqualRanges{Here,Here}(a, n, b);
    ensures result: \result == n;

  behavior some_not_equal:
    assumes !EqualRanges{Here,Here}(a, n, b);
    ensures bound:   0 <= \result < n;
    ensures result:  a[\result] != b[\result];
    ensures first:   EqualRanges{Here,Here}(a, \result, b);

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

Listing 3.17: Formal specification of `mismatch`

The formal specification of `mismatch` in Listing 3.17 is more complex than that of `equal` because the return value of mismatch provides more information than just reporting whether the two arrays are equal. On the other, the specification is conceptually quite similar to that of `find` (Listing 3.5). While `find` returns the smallest index `i` where `a[i] == val` holds, `mismatch` finds the smallest index `a[i] != b[i]`.

Note in particular the use of `EqualRanges` in the specification of `mismatch`. As in the specification of `find` 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.

3.5.3. Implementation of `equal` and `mismatch`

Listing 3.18 shows an implementation of the `equal` algorithm by 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 3.18: Implementation of `equal` with `mismatch`

Listing 3.19 shows an implementation of `mismatch` that we have 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:  EqualRanges{Here,Here}(a, i, b);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++) {
    if (a[i] != b[i]) {
      return i;
    }
  }

  return n;
}
```

Listing 3.19: Implementation of `mismatch`

We use the predicate `EqualRanges` 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.

3.6. The search algorithm

The search algorithm in the C++ Standard Library [14, §25.2.13] 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 m,
                 const value_type* b, size_type n)
```

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 < n (see Figure 3.20). If no such index exists, then search returns the length m of the array a.

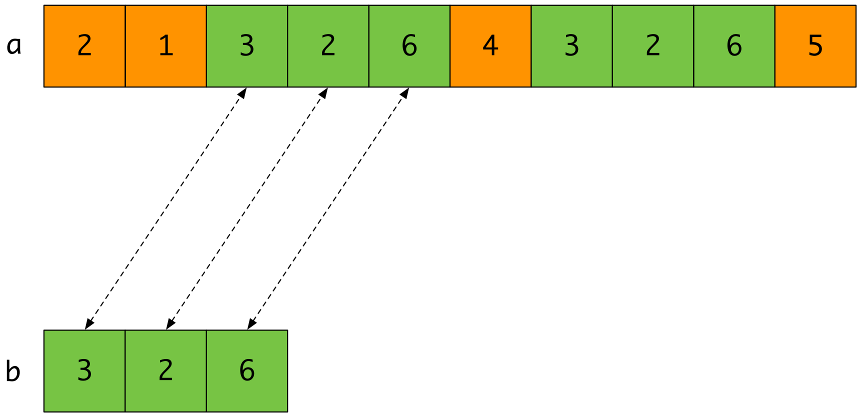

Figure 3.20.: Searching the first occurrence of b[0..n-1] in a[0..m-1]

3.6.1. The predicate HasSubRange

Our specification of search starts with introducing the predicate HasSubRange in Listing 3.21. This predicate formalizes, using the predicate EqualRanges defined in Listing 3.15, that the sequence a contains a subsequence which equal the sequence b. Of course, in order to contain a subsequence of length n, a must be at least that large; this is expressed by lemma HasSubRangeSizes.

```
/*@
  predicate
    HasSubRange{A}(value_type* a, integer f, integer l, value_type* b, integer n) =
      \exists integer k; (f <= k <= l-n) && EqualRanges{A,A}(a+k, n, b);

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

  lemma
    HasSubRangeSizes:
      \forall value_type *a, *b, integer f, t, n;
        HasSubRange(a, f, t, b, n) ==> n <= t-f;
*/
```

Listing 3.21: The predicate HasSubRange

3.6.2. Formal specification of search

The ACSL specification of `search` is shown in Listing 3.22. Conceptually, the specification of `search` is very similar to that of `find` (Section 3.1). 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`.

```
/*@
  requires \valid_read(a + (0..m-1));
  requires \valid_read(b + (0..n-1));

  assigns  \nothing;

  ensures  result: 0 <= \result <= m;

  behavior has_match:
    assumes HasSubRange(a, 0, m, b, n);
    ensures bound:  0 <= \result <= m-n;
    ensures result: EqualRanges{Here,Here}(a+\result, n, b);
    ensures first:  !HasSubRange(a, 0, \result+n-1, b, n);

  behavior no_match:
    assumes !HasSubRange(a, 0, m, b, n);
    ensures result: \result == m;

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

Listing 3.22: Formal specification of `search`

The ensures clause `bound` of behavior `has_match` indicates that the return value must be in the range `0..m-n`. 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+n-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 `m` 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*.

3.6.3. Implementation of search

Our implementation of search is shown in Listing 3.23. It follows the C++ Standard Library implementation in being easy to understand, but needing an order of magnitude of m*n operations. In contrast, the sophisticated algorithm from [15] needs only m+n operations.[15]

```
size_type search(const value_type* a, size_type m,
                 const value_type* b, size_type n)
{
  if (n <= m) {

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

  //@ assert no_match: !HasSubRange(a, 0, m, b, n);
  return m;
}
```

Listing 3.23: Implementation of search

The loop invariant not_found is needed for the proof of the postconditions of the behavior has_match (see Listing 3.22). It expresses that the subsequence b has not been found up to the current iteration step.

The trivial case m < n is caught separately in order to prevent an overflow in computation of m - n in the loop. Neither n == 0 nor m == 0 need to be handled separately, not even for efficiency reasons: in the former case, equal(a+i, n, b) will succeed in the first iteration, while in the latter, n > m will apply.

[15] 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

3.7. The `search_n` algorithm

The `search_n` algorithm in the C++ Standard Library [14, §25.2.13] 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 m, size_type n, value_type b)
```

Note the similarity to the signature of `search` (Section 3.6). The only difference is that `b` now is a single value rather than an array.[16] The function `search_n` returns the first index `s` of the array `a` where the condition `a[s+k] == b` holds for each index `k` with `0 <= k < n` (see Figure 3.24). If no such index exists, then `search_n` returns the length `m` of the array `a`.

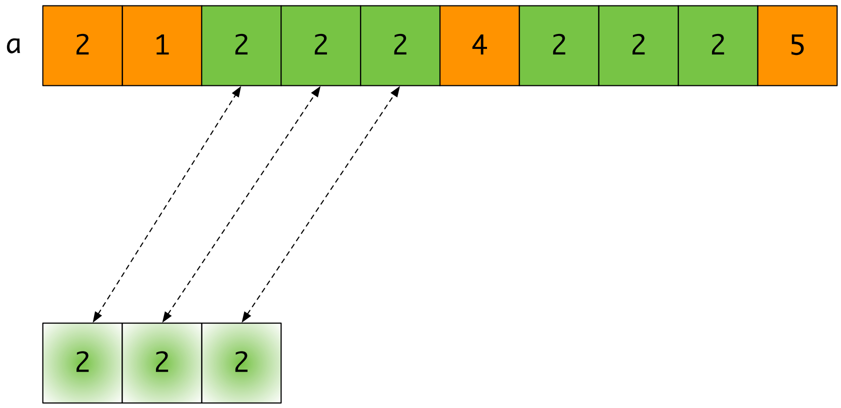

Figure 3.24.: Searching the first occurrence a given constant sequence in `a[0..m-1]`

3.7.1. The predicates `ConstantRange` and `HasConstantSubRange`

Our specification of `search_n` starts with introducing the predicate `ConstantRange` in Listing 3.25, which expresses that each member `a[first..last-1]` equals `val`.

```
/*@
  predicate
    ConstantRange(value_type* a, integer first, integer last, value_type v) =
      \forall integer i; first <= i < last ==> a[i] == v;

  predicate
    ConstantRange(value_type* a, integer first, integer last) =
      ConstantRange(a, first, last, a[first]);

  predicate
    ConstantRange(value_type* a, integer n, value_type v) =
      ConstantRange(a, 0, n, v);
*/
```

Listing 3.25: The predicate `ConstantRange`

There are two additional overloaded versions of `ConstantRange`. The first one uses the `a[first]` as `val`. We will use this particular version later on during the specification of `unique_copy` (Listing 6.16).

[16] For some reason, the C++ Standard Library has swapped the `b` and `n` parameter; we followed that order.

The second version is just a shortcut when the first index is 0.

Based on that, the predicate HasConstantSubRange in Listing 3.26 formalizes that the sequence a of length m contains a subsequence of n times the value b. Similar to HasSubRange, in order to contain n repetitions, a must be at least that large; this is what lemma HasConstantSubRangeSizes says.

```
/*@
  predicate
    HasConstantSubRange{A}(value_type* a, integer m, integer n, value_type b) =
      \exists integer i; 0 <= i <= m-n && ConstantRange(a, i, i+n, b);


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

Listing 3.26: The predicate HasConstantSubRange

3.7.2. Formal specification of search_n

The ACSL specification of search_n is shown in Listing 3.27. Like for search, the specification of search_n is very similar to that of find. 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.

```
/*@
  requires valid: \valid_read(a + (0..m-1));

  assigns  \nothing;

  ensures  result:  0 <= \result <= m;

  behavior has_match:
    assumes HasConstantSubRange(a, m, n, b);
    ensures result:  0 <= \result <= m-n;
    ensures match:   ConstantRange(a, \result, \result+n, b);
    ensures first:   !HasConstantSubRange(a, \result+n-1, n, b);

  behavior no_match:
    assumes !HasConstantSubRange(a, m, n, b);
    ensures result:  \result == m;

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

Listing 3.27: Formal specification of search_n

The result ensures clause of behavior has_match indicates that the return value must be in the range [0..m-n]. The match ensures clause expresses that the return value of search_n actually points to an index where b can be found n 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 m of the range a.

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*.

3.7.3. Implementation of search_n

Although the specification of search_n strongly resembles that of search, their implementations in the C++ Standard Library are significantly different. The former has a time complexity of $O(m)$, whereas the latter employs an easy, but a non-optimal algorithm needing $O(m \cdot n)$ time, cf. Section 3.6.3. Despite its linear complexity, the Standard Library search_n implementation uses two nested loops which is somewhat hard to understand; in Listing 3.28, we give an equally efficient implementation with only one loop.

```
size_type
search_n(const value_type* a, size_type m, size_type n, value_type b)
{
  if (0u < n) {

    if (n <= m) {

      size_type start = 0;

      /*@
        loop invariant constant:  ConstantRange(a, start, i, b);
        loop invariant start:     0 < start ==> a[start-1] != b;
        loop invariant bound:     start <= i + 1;
        loop invariant not_found: !HasConstantSubRange(a, i, n, b);
        loop assigns i, start;
        loop variant m - i;
      */
      for (size_type i = 0; i < m; ++i) {
        if (a[i] != b) {
          start = i + 1;
        }
        else if (n == i + 1 - start) {
          return start;
        }
      }
    }

    return m;
  }
  return 0;
}
```

Listing 3.28: Implementation of search_n

Our implementation maintains in the variable start the beginning of the most recent consecutive range of values b. This property is expressed by three loop invariants:

- constant states that b is the only value that can occur in a[start..i-1];
- start states that it cannot be left-extended, i.e. a[start-1] (if defined) is different from b;
- bound states that the sequence's bounds are actually given left to right.

The loop invariant `not_found` states that we didn't find an `n`-fold repetition of `b` up to now; if we find one, we terminate the loop, returning `start`.

We handle the trivial case `m < n` separately before entering the loop, thus avoiding unnecessary work. The other trivial case `n <= 0` is caught since our loop relies on the range searched for being non-empty. Only in this case checking `a[i] != b` makes sense in the very first iteration.

3.8. The `find_end` algorithm

The `find_end` algorithm in the C++ Standard Library [14, §25.2.6] 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 m,
                   const value_type* b, size_type n)
```

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 < n` (see Figure 3.29). If no such index exists, then `find_end` returns the length `m` of the array `a`. One has to remark the special case `n == 0`. In this case the last position of the empty string is found (the length `m`) and returned.

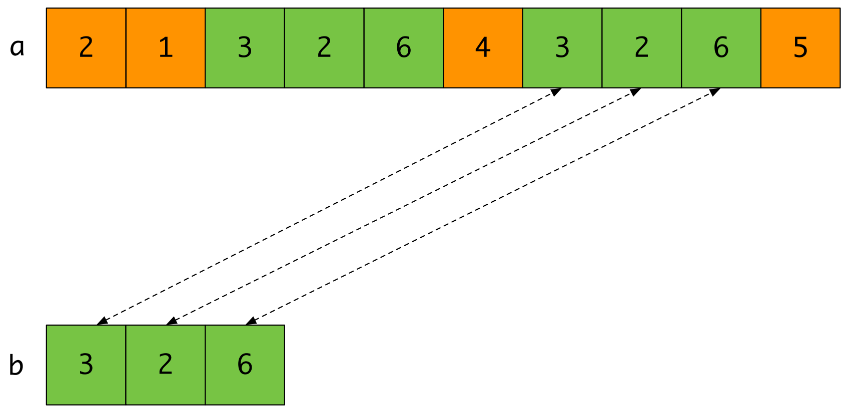

Figure 3.29.: Finding the last occurrence `b[0..n-1]` in `a[0..m-1]`

3.8.1. Formal specification of find_end

The ACSL specification of find_end is shown in Listing 3.30. Conceptually, the specification of the function find_end is very similar to that of find in Section 3.2. We therefore use again behaviors to capture the essential aspects of find_end.

```
/*@
  requires valid: \valid_read(a + (0..m-1));
  requires valid: \valid_read(b + (0..n-1));

  assigns \nothing;

  ensures result: 0 <= \result <= m;

  behavior has_match:
    assumes HasSubRange(a, 0, m, b, n);
    ensures bound:  0 <= \result <= m-n;
    ensures result: EqualRanges{Here,Here}(a + \result, n, b);
    ensures last:   !HasSubRange(a, \result + 1, m, b, n);

  behavior no_match:
    assumes !HasSubRange(a, 0, m, b, n);
    ensures result: \result == m;

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

Listing 3.30: Formal specification of find_end

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. The ensures clause bound indicates that the return value must be in the range 0..m-n. 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 m of the range a.

It is quite clear that these behaviors are *complete* and *disjoint*.

3.8.2. Implementation of `find_end`

Our implementation of `find_end` is shown in Listing 3.31. Similar to our `search` implementation (Section 3.6), it follows the C++ Standard Library implementation in being easy to understand, but needing an order of magnitude of `m*n` rather than only `m+n` operations.

```
size_type find_end(const value_type* a, size_type m,
                   const value_type* b, size_type n)
{
  size_type ret = m;

  if (0u < n && n <= m) {

    /*@
      loop invariant bound:  ret <= m - n || ret == m;
      loop invariant result: ret == m ==> !HasSubRange(a, n+i-1, b, n);
      loop invariant result: ret < m  ==> EqualRanges{Here,Here}(a + ret, n, b);
      loop invariant last:   ret < m  ==> !HasSubRange(a, ret+1, i+n-1, b, n);
      loop assigns i, ret;
      loop variant m - i;
    */
    for (size_type i = 0; i <= m - n; ++i) {
      if (equal(a + i, n, b)) {
        ret = i;
      }
    }
  }

  return ret;
}
```

Listing 3.31: Implementation of `find_end`

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

Invariant `bound` states that `ret` either still has the value `m` or has a value up to `m-n`. For the former case, invariant `result1` indicates that no occurrence of `b` has been found. For the latter case, invariant `result2` indicates that an occurrence at `ret` has been found, and invariant `last` states that none was found so far after `ret`.

3.9. The count algorithm

The count algorithm in the C++ Standard Library [14, §25.2.9] 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 val);
```

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

3.9.1. An axiomatic definition of counting on array sections

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.[17] 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 CountSection(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 Listing 3.32.[18]

```
/*@
  axiomatic Count
  {
    logic integer
    Count{L}(value_type* a, integer m, integer n, value_type v) reads a[m..n-1];

    axiom
      CountSectionEmpty{L}:
        \forall value_type *a, v, integer m, n;
          n <= m  ==>  Count(a, m, n, v) == 0;

    axiom
      CountSectionHit{L}:
        \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;

    axiom
      CountSectionMiss{L}:
        \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);

    axiom
      CountSectionRead{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 3.32: The logic function Count

[17] This statement is not quite true because the ACSL documentation lists numof as one of several *higher order logic constructions* [9, §2.6.7]. However, these *extended quantifiers* are mentioned only as experimental features.

[18] This definition of Count is a generalization of the *logic function* nb_occ of the ACSL specification [9, p. 55].

- The ACSL keyword `axiomatic` is used to gather the logic function `Count` and its defining axioms. Note that the interval bounds `m` and `n` and the return value for `Count` are of type `integer`.
- Axiom `CountSectionEmpty` covers the case of an empty range.
- Axioms `CountSectionOneHit` and `CountSectionOneMiss` reduce counting of a range of length $n + 1$ to a range of length n.
- The `reads` clause in the axiomatic definition of `Count` specifies the set of memory locations on which `Count` depends.

  Axiom `CountSectionRead` 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 `L1` and `L2`. We use predicate `Unchanged` (Listing 6.1 in Section 6.1) to express the premise of Axiom `CountSectionRead`. Axiom `CountSectionRead` is helpful if one has to verify *mutating* algorithms that rely on `Count`, e.g., `remove_copy` in Section 6.14. It is an inductive consequence of axioms `CountSectionEmpty`, `CountSectionOneHit`, and `CountSectionOneMiss`, but we don't prove it here.

Listing 3.33 shows some additional properties of `Count`.

```
/*@
  lemma
    CountSectionOne:
      \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
    CountSectionUnion:
      \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);
*/
```

Listing 3.33: Some lemmas for `Count`

3.9.2. Counting on a whole array

We also provide in Listing 3.34 an overloaded version of the logic function `Count` that only takes the starting address and the size of an array. This is just a convenience function for the use in specifications that do not need to consider array sections. Note how the accompanying lemmas in Listing 3.34 correspond to the axioms in Listing 3.32.

```
/*@
  logic integer
    Count{L}(value_type* a, integer n, value_type v) = Count{L}(a, 0, n, v);

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

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

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

  lemma
    CountRead{L1,L2}:
      \forall value_type *a, v, integer n;
        Unchanged{L1,L2}(a, n)  ==>  Count{L1}(a, n, v) == Count{L2}(a, n, v);
*/
```

Listing 3.34: The logic function `Count`

The lemmas for `Count` in Listing 3.35 are just reformulated versions of those in Listing 3.33.

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

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

*/
```

Listing 3.35: Some lemmas for `Count`

3.9.3. Formal specification of count

Listing 3.36 shows how we use the logic function Count from Listing 3.34 to specify count in ACSL. 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, val);
*/
size_type count(const value_type* a, size_type n, value_type val);
```

Listing 3.36: Formal specification of count

3.9.4. Implementation of count

Listing 3.37 shows a possible implementation of count. 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 val)
{
  size_type counted = 0;

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

  return counted;
}
```

Listing 3.37: Implementation of count

4. Maximum and minimum algorithms

In this chapter we discuss the formal specification of algorithms in the C++ Standard Library [14, §25.4.7] that compute the maximum or minimum values of their arguments. As the algorithms in Chapter 3, 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.

- `max_element` returns an index to a maximum element in a range. Similar to `find` it also returns the smallest of all possible indices. It is discussed in Section 4.2, on Page 55. In Section 4.3, on Page 57, we will introduce an alternative specification which relies on user-defined predicates.
- `max_seq` (Section 4.4, on Page 59) 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` can be used to find the smallest element in an array (Section 4.5).

First, however, we discuss in Section 4.1 general properties that must be satisfied by the relational operators.

4.1. A note on relational operators

Note that in order to compare values, the algorithms in the C++ Standard Library [14, §25.4.7] usually rely solely on the *less than* operator < or special function objects. To be precise, the operator < must be a *partial order*,[19] which means that the following rules must hold.

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

If you wish to check that the operator < of our `value_type`[20] satisfies these properties you can formulate lemmas in ACSL and verify them with Frama-C (see Listing 4.1).

[19] See `http://en.wikipedia.org/wiki/Partially_ordered_set`
[20] See Section 1.3

```
/*@
  lemma
    LessIrreflexivity:
      \forall value_type a; !(a < a);

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

  lemma
    LessTransitivity:
      \forall value_type a, b, c; (a < b) && (b < c) ==> (a < c);
*/
```

Listing 4.1: Requirements for a partial order on `value_type`

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 4.2 formulates conditions on the semantics of the derived operator >, <=, and >=.

```
/*@
  lemma
    Greater:
      \forall value_type a, b; (a > b) <==> (b < a);

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

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

Listing 4.2: Semantics of derived comparison operators

We also introduce in this chapter the predicates

- `UpperBound` in Listing 4.5
- `StrictUpperBound` in Listing 4.6
- `LowerBound` in Listing 4.12
- `StrictLowerBound` in Listing 4.13

These overloaded 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 5.

4.2. The max_element algorithm

The `max_element` algorithm in the C++ Standard Library [14, §25.4.7] 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`.

4.2.1. Formal specification of max_element

A formal specification of `max_element` in ACSL is shown in Listing 4.3.

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

  behavior empty:
    assumes n == 0;
    ensures result:  \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures result:  0 <= \result < n;
    ensures upper:   \forall integer i; 0 <= i < n        ==> a[i] <= a[\result];
    ensures strict:  \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 4.3: Formal specification of `max_element`

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

4.2.2. Implementation of max_element

Listing 4.4 shows an implementation of `max_element`. In our description, we concentrate on the *loop annotations*.

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

    size_type max = 0;

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

    return max;
  }

  return n;
}
```

Listing 4.4: Implementation of `max_element`

Loop invariant `max` is needed to prove postcondition `result` of behavior `not_empty` in Listing 4.3. Using loop invariant `upper` we prove postcondition `upper` of behavior `not_empty` in Listing 4.3. Finally, postcondition `strict` of this behavior can be proved with loop invariant `strict`.

4.3. 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 several user-defined predicates. First, we define in Listing 4.5 the overloaded predicate `UpperBound` which basically expresses that a given value is greater or equal than all elements of a given array (section).

```
/*@
  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);
*/
```

Listing 4.5: Definition of the `UpperBound` predicate

Closely related to the predicate `UpperBound` is the overloaded predicate `StrictUpperBound` from Listing 4.6.

```
/*@
  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 4.6: Definition of the `StrictUpperBound` predicate

We then define in Listing 4.7 the predicate `MaxElement` by stating 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.

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

Listing 4.7: Definition of the `MaxElement` predicate

4.3.1. Formal specification of max_element

The new formal specification of max_element in ACSL is shown in Listing 4.8. Note that we also use the predicate StrictUpperBound from Listing 4.6 in order to express that max_element 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;
    ensures result:   \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures result:   0 <= \result < n;
    ensures max:      MaxElement(a, n, \result);
    ensures strict:   StrictUpperBound(a, \result, a[\result]);

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

Listing 4.8: Formal specification of max_element

4.3.2. Implementation of max_element

Listing 4.9 shows implementation of max_element with only the loop invariants changed.

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

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

Listing 4.9: Implementation of max_element

4.4. The max_seq algorithm

In this section we consider the function max_seq (see Chapter 3, [8]) which is very similar to the function max_element of Section 4.2. 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_element (see Listing 4.11). Moreover, using only the formal specification of max_element in Listing 4.8 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_element that is separately proven to implement the contract in Listing 4.8 makes max_seq behave correctly. Once the contracts have been defined, the function max_element could be implemented in parallel, or just after max_seq, without affecting the verification of max_seq.

4.4.1. Formal specification of max_seq

A formal specification of max_seq in ACSL is shown in Listing 4.10.

```
/*@
  requires n > 0;
  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 4.10: 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.

4.4.2. Implementation of max_seq

Listing 4.11 shows the trivial implementation of max_seq using max_element. Since max_seq requires a non-empty range the call of max_element returns an index to a maximum value in the range. The fact that max_element 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_element(p, n)];
}
```

Listing 4.11: Implementation of max_seq

4.5. The `min_element` algorithm

The `min_element` algorithm in the C++ Standard Library [14, §25.4.7] 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`.

First we define in Listing 4.12 the overloaded predicate `LowerBound` that basically expresses that a given value is less or equal than all elements of a given array (section).

```
/*@
  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);
*/
```

Listing 4.12: Definition of the `LowerBound` predicate

Closely related to the predicate `LowerBound` is the overloaded predicate `StrictLowerBound` from Listing 4.13.

```
/*@
  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);
*/
```

Listing 4.13: Definition of the `StrictLowerBound` predicate

We then define in Listing 4.14 the predicate `MinElement` by stating 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.

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

Listing 4.14: Definition of the `MinElement` predicate

4.5.1. Formal specification of min_element

The ACSL specification of min_element is shown in Listing 4.15. Note that we also use the predicate StrictLowerBound from Listing 4.13 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;
    ensures result:  \result == 0;

  behavior not_empty:
    assumes 0 < n;
    ensures result:  0 <= \result < n;
    ensures min:     MinElement(a, n, \result);
    ensures strict:  StrictLowerBound(a, \result, a[\result]);

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

Listing 4.15: Formal specification of min_element

4.5.2. Implementation of min_element

Listing 4.16 shows the implementation of min_element with loop invariants where we also employ the predicates LowerBound and StrictLowerBound.

```
size_type min_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type min = 0;
    /*@
      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 = 0; i < n; i++) {
      if (a[i] < a[min]) {
        min = i;
      }
    }
    return min;
  }
  return n;
}
```

Listing 4.16: Implementation of min_element

5. Binary search algorithms

In this chapter, we consider the four *binary search* algorithms of the C++ Standard Library [14, §25.4.3], namely

- `lower_bound` in Section 5.1
- `upper_bound` in Section 5.2
- two variants for the implementation of `equal_range` in Section 5.3
- two variants for the formal specification of `binary_search` in Section 5.4

All binary search algorithms require that their input array is sorted in ascending order. There are two versions of predicate `Sorted` in Listing 5.1. The first one defines when a section of an array is sorted in ascending order. The second version uses the first one to express that the whole array is sorted.

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

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

Listing 5.1: The predicate `Sorted`

As in the case of the of maximum/minimum algorithms from Chapter 4 the binary search algorithms primarily use the less-than operator < (and the derived operators <=, > and >=) to determine whether a particular value is contained in a sorted range. Thus, different to the `find` algorithm in Section 3.1, 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 the following predicates

- `UpperBound` in Listing 4.5
- `StrictUpperBound` in Listing 4.6
- `LowerBound` in Listing 4.12
- `StrictLowerBound` in Listing 4.13

5.1. The `lower_bound` algorithm

The `lower_bound` algorithm is one of the four binary search algorithms of the C++ Standard Library [14, §25.4.3.1]. 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 val);
```

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

$$0 \leq \mathtt{lb} \leq n \tag{5.1}$$

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

$$0 \leq k < \mathtt{lb} \implies a[k] < \mathtt{val} \tag{5.2}$$

$$\mathtt{lb} \leq k < n \implies \mathtt{val} \leq a[k] \tag{5.3}$$

Conditions (5.2) and (5.3) imply that `val` 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 5.2 a sorted 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.

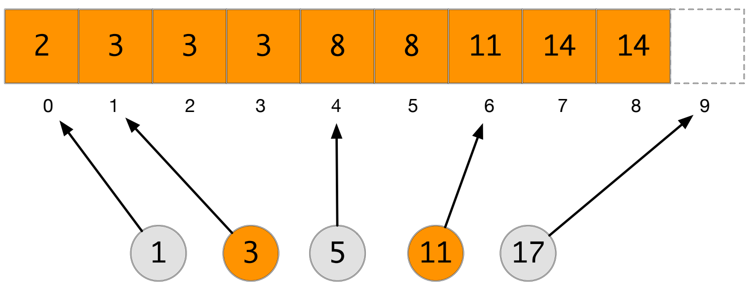

Figure 5.2.: Some examples for `lower_bound`

Figure 5.2 also clarifies that care must be taken when interpreting the return value of `lower_bound`. An important difference to the algorithms in Chapter 3 is that a return value of `lower_bound` that is less than n does not necessarily implies `a[lb] == val`. We can only be sure that `val <= a[lb]` holds.

5.1.1. Formal specification of `lower_bound`

The ACSL specification of `lower_bound` is shown in Listing 5.3. The preconditions `sorted` expresses that the values in the (valid) array need to be sorted in ascending order. The postconditions reflect the conditions listed above and can be expressed using predicates `LowerBound` from Listing 4.12 and `StrictLowerBound` from Listing 4.13, namely,

- Condition (5.1) becomes postcondition `result`

- Condition (5.2) becomes postcondition `left`
- Condition (5.3) becomes postcondition `right`

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result <= n;
  ensures left:    StrictUpperBound(a, 0, \result, val);
  ensures right:   LowerBound(a, \result, n, val);
*/
size_type lower_bound(const value_type* a, size_type n, value_type val);
```

Listing 5.3: Formal specification of `lower_bound`

5.1.2. Implementation of `lower_bound`

Our implementation of `lower_bound` is shown in Listing 5.4. 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 `val` and all indices not less than the temporary right bound `right` contain values that are greater or equal than `val`. 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 val)
{
  size_type left = 0;
  size_type right = n;

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

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

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

  return left;
}
```

Listing 5.4: Implementation of `lower_bound`

5.2. The `upper_bound` algorithm

The `upper_bound` algorithm of the C++ Standard Library [14, §25.4.3.1] is a variant of binary search and closely related to `lower_bound` of Section 5.1. The signature reads:

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

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

$$0 \leq \mathtt{ub} \leq n \tag{5.4}$$

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

$$0 \leq k < \mathtt{ub} \implies a[k] \leq \mathtt{val} \tag{5.5}$$
$$\mathtt{ub} \leq k < n \implies \mathtt{val} < a[k] \tag{5.6}$$

Conditions (5.5) and (5.6) imply that `val` 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 `val` can occur. It also means that the searched-for value `val` can *never* be located at the index `ub`.

Figure 5.5 is a variant of Figure 5.2 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 5.2, only the arrows from values that *are present* in the array change their target index.

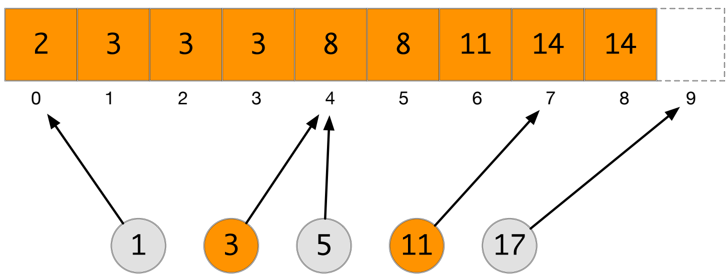

Figure 5.5.: Some examples for `upper_bound`

5.2.1. Formal specification of `upper_bound`

The ACSL specification of `upper_bound` is shown in Listing 5.6. The specification is quite similar to the specification of `lower_bound` (see Listing 5.3). The precondition `sorted` expresses that the values in the (valid) array need to be sorted in ascending order.

The postconditions reflect the conditions listed above and can be expressed using predicates `UpperBound` from Listing 4.5 and `StrictUpperBound` from Listing 4.6, namely,

- Condition (5.4) becomes postcondition `result`
- Condition (5.5) becomes postcondition `left`
- Condition (5.6) becomes postcondition `right`

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result <= n;
  ensures left:    UpperBound(a, 0, \result, val);
  ensures right:   StrictLowerBound(a, \result, n, val);
*/
size_type
upper_bound(const value_type* a, size_type n, value_type val);
```

Listing 5.6: Formal specification of upper_bound

5.2.2. Implementation of upper_bound

Our implementation of upper_bound is shown in Listing 5.7.

The loop invariants express that for each iteration step all indices less than the temporary left bound left contain values not greater than val and all indices not less than the temporary right bound right contain values greater than val.

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

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

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

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

  return right;
}
```

Listing 5.7: Implementation of upper_bound

5.3. The `equal_range` algorithm

The `equal_range` algorithm is one of the four binary search algorithms of the C++ Standard Library [14, §25.4.3.3]. As with the other binary search algorithms `equal_range` requires that its input array is sorted in ascending order. The specification of `equal_range` states that it *combines* the results of the algorithms `lower_bound` (Section 5.1) and `upper_bound` (Section 5.2).

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 5.9. Thus, the signature of `equal_range` now reads:

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

Figure 5.8 combines Figure 5.2 with Figure 5.5 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 $(\mathtt{lb}, \mathtt{ub})$, then the difference $\mathtt{ub} - \mathtt{lb}$ is equal to the number of occurrences of the argument `val` in the array.

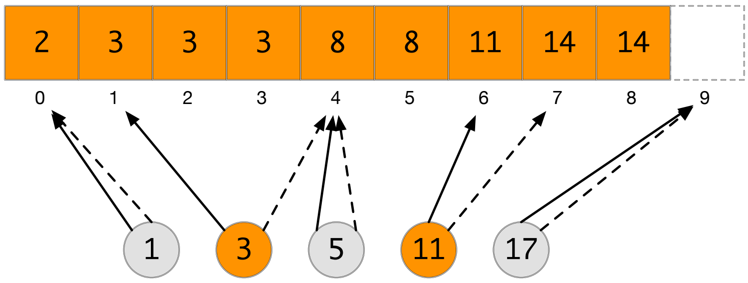

Figure 5.8.: Some examples for `equal_range`

We will provide two implementations of `equal_range` and verify both of them. The first implementation just straightforwardly calls `lower_bound` and `upper_bound` and simply returns their results (see Listing 5.11). The second, more elaborate, implementation follows the original STL code by attempting to minimize duplicate computations (see Listing 5.12).

Let $(\mathtt{lb}, \mathtt{ub})$ be the return value `equal_range`, then the conditions (5.1)–(5.6) can be merged into the inequality

$$0 \le \mathtt{lb} \le \mathtt{ub} \le n \tag{5.7}$$

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

$$0 \le k < \mathtt{lb} \implies a[k] < \mathtt{val} \tag{5.8}$$
$$\mathtt{lb} \le k < \mathtt{ub} \implies a[k] = \mathtt{val} \tag{5.9}$$
$$\mathtt{ub} \le k < n \implies a[k] > \mathtt{val} \tag{5.10}$$

Here are some justifications for these conditions.

- Conditions (5.8) and (5.10) are just the Conditions (5.2) and (5.6), respectively.

- The Inequality (5.7) follows from the Inequalities (5.1) and (5.4) and the following considerations: If ub were less than lb, then according to (5.8) we would have a[ub] < val. One the other hand, we know from (5.10) that opposite inequality val < a[ub] holds. Therefore, we have lb ≤ ub.
- Condition (5.9) follows from the combination of (5.3) and (5.5) and the fact that ≤ is a total order on the integers.

5.3.1. The auxiliary function `make_pair`

The type `size_type_pair` and the `make_pair` function in Listing 5.9 are used both for the first and second implementation of `equal_range`. The specification and implementation of this simple function is shown in Listing 5.9.

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

typedef struct size_type_pair size_type_pair;


/*@
    assigns \nothing;

    ensures \result.first  == first;
    ensures \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 5.9: The type `size_pair_type` and the function `make_pair`

5.3.2. Formal specification of equal_range

The ACSL specification of equal_range is shown in Listing 5.10.

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  requires sorted: Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result.first <= \result.second <= n;
  ensures left:    StrictUpperBound(a, 0, \result.first, val);
  ensures middle:  ConstantRange(a, \result.first, \result.second, val);
  ensures right:   StrictLowerBound(a, \result.second, n, val);
 */
size_type_pair
equal_range(const value_type* a, size_type n, value_type val);
```

Listing 5.10: Formal specification of equal_range

The ACSL specification of equal_range is shown in Listing 5.10. The precondition sorted expresses that the values in the (valid) array need to be sorted in ascending order.

The postconditions reflect the conditions listed above and can be expressed using the well-known predicates ConstantRange (Listing 3.25), StrictUpperBound (Listing 4.6) and StrictLowerBound (Listing 4.13), namely,

- Condition (5.7) becomes postcondition result
- Condition (5.8) becomes postcondition left
- Condition (5.9) becomes postcondition middle
- Condition (5.10) becomes postcondition right

5.3.3. First implementation of equal_range

Our first implementation of equal_range is shown in Listing 5.11. We just call the two functions lower_bound and upper_bound and return their respective results as a pair.

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

  return make_pair(first, second);
}
```

Listing 5.11: First implementation of equal_range

In an earlier 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 (Listing 5.6). The benefit of this reformulation is that both the assertion aux and the postcondition first <= second can now be verified automatically.

5.3.4. Second implementation of `equal_range`

The first implementation of `equal_range` does more work than needed. STL uses a slightly more complicated implementation of `equal_range` that performs as much range reduction as possible before calling `upper_bound` and `lower_bound` on the reduced ranges. In Listing 5.12 we translated the STL implementation to C code and verified it. It is a drop-in replacement for the first implementation and implements the same formal specification, provided in Listing 5.10.

```
size_type_pair
equal_range(const value_type* a, size_type n, value_type val)
{
  size_type first  = 0;
  size_type middle = 0;
  size_type last   = n;

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

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

  if (first < last) {
    //@ assert sorted: Sorted(a, first, middle);
    size_type left = first + lower_bound(a + first, middle - first, val);
    //@ assert constant: LowerBound(a, left, middle, val);
    //@ assert strict: StrictUpperBound(a, first, left, val);

    ++middle;

    //@ assert sorted: Sorted(a, middle, last);
    size_type right = middle + upper_bound(a + middle, last - middle, val);
    //@ assert constant: UpperBound(a, middle, right, val);
    //@ assert strict: StrictLowerBound(a, right, last, val);

    return make_pair(left, right);
  }
  else {
    return make_pair(first, first);
  }
}
```

Listing 5.12: Second implementation of `equal_range`

Due to the higher complexity of the second implementation, additional assertions had to be added to ensure that Frama-C is able to verify the correctness of the code. All of these 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 ACSL lemmas in Listing 5.13.

```
/*@
  lemma
    SortedShift{L}:
      \forall value_type *a, integer l, r;
      0 <= l <= r  ==>  Sorted{L}(a, l, r)  ==>  Sorted{L}(a+l, r-l);

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

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

Listing 5.13: Some lemmas to support the verification of `equal_range`

The `sorted` properties

Both `upper_bound` and `lower_bound` require that they operate on sorted data. This is also true for `equal_range`, however, inside our second implementation we need a more specific formulation, namely,

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

A three-argument form of the `Sorted` predicate from Listing 5.1 was added so we can spell out an intermediate step. This enables the provers to verify the preconditions of the call to `lower_bound` automatically. A similar assertion is present before the call to `upper_bound`.

The `strict` and `constant` properties

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

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

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

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, val)

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

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

5.4. The binary_search algorithm

The `binary_search` algorithm is one of the four binary search algorithms of the C++ Standard Library [14, §25.4.3.4]. 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  val);
```

Again, `binary_search` requires that its input array is sorted in ascending order. It will return `true` if there exists an index `i` in `a` such that `a[i] == val` holds.[21]

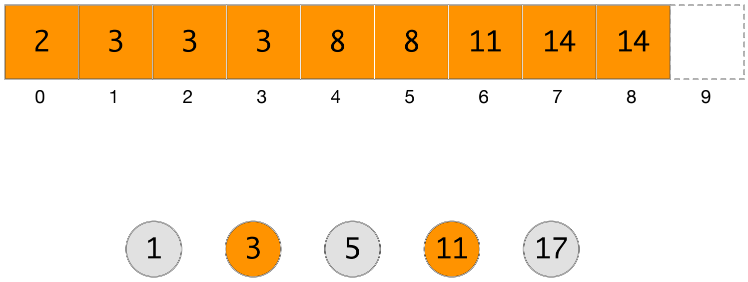

Figure 5.14.: Some examples for `binary_search`

In Figure 5.14 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.

5.4.1. Formal specification of binary_search

The ACSL specification of `binary_search` is shown in Listing 5.15.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: Sorted(a, n);

  assigns  \nothing;

  ensures  result:  \result <==> \exists integer i; 0 <= i < n && a[i] == val;
*/
bool binary_search(const value_type* a, size_type n, value_type val);
```

Listing 5.15: Formal specification of `binary_search`

Note that instead of the somewhat lengthy existential quantification in Listing 5.15 we can use our previously introduced predicate `HasValue` (see Listing 3.4) in to achieve the more concise formal specification in Listing 5.16.

[21] To be more precise: The C++ Standard Library requires that `(a[i] <= val) && (val <= a[i])` holds. For our definition of `value_type` (see Section 1.3) this means that `val` equals `a[i]`.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: Sorted(a, n);

  assigns  \nothing;

  ensures  result: \result <==> HasValue(a, n, val);
*/
bool binary_search(const value_type* a, size_type n, value_type val);
```

Listing 5.16: Formal specification of `binary_search` using the `HasValue` predicate

It is interesting to compare this specification with that of `find` shown in Listing 3.5. Both `find` and `binary_search` 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.

5.4.2. Implementation of binary_search

Our implementation of `binary_search` is shown in Listing 5.17.

```
bool binary_search(const value_type* a, size_type n, value_type val)
{
  const size_type i = lower_bound(a, n, val);

  return i < n && a[i] <= val;
}
```

Listing 5.17: Implementation of `binary_search`

The function `binary_search` first calls `lower_bound` from Section 5.1. Remember that if the latter returns an index `0 <= i < n` then we can be sure that `val <= a[i]` holds.

6. Mutating algorithms

Let us now turn our attention to another class of algorithms, viz. *mutating* algorithms of the C++ Standard Library [14, §25.3], 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 [9, §2.3.2]. Below are the algorithms we will discuss in this chapter. First, however, we introduce in Sections 6.1 and 6.2 the auxiliary predicates `Unchanged` and `MultisetUnchanged`, respectively.

- `fill` in Section 6.3 initializes each element of an array by a given fixed value.
- `swap` in Section 6.4 exchanges two values.
- `swap_ranges` in Section 6.5 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 Section 6.6 copies a source array to a destination array.
- `copy_backward` in Section 6.7 also copies a source array to a destination array. This version, however, uses another separation condition than `copy`.
- `reverse_copy` and `reverse` in Sections 6.8 and 6.9, respectively, reverse an array. Whereas `reverse_copy` copies the result to a separate destination array, the `reverse` algorithm works in place.
- `rotate_copy` in Section 6.10 rotates a source array by `m` positions and copies the results to a destination array.
- `rotate` in Section 6.11 rotates *in place* a source array by `m` positions.
- `replace_copy` and `replace` in Sections 6.12 and 6.13, 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` in Section 6.14 copies a source array to a destination array, but omits each occurrence of a given value.
- `random_shuffle` Section 6.17 re-arranges the elements of an array in a random way.

6.1. 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*; this cannot always be expressed by an `assigns` clause. In Listing 6.1 we therefore introduce the overloaded predicate `Unchanged` together with some simple lemmas. The expression `Unchanged{K,L}(a,f,l)` is true if the range `a[f..l-1]` in state `K` is element-wise equal to that range in state `L`.

```
/*@
  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 6.1: The predicate `Unchanged`

We also provide a few lemmas for `Unchanged` that we need for the verification of some algorithms.

Lemma `UnchangedSection` in Listing 6.2 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 $0 \leq m \leq p \leq q \leq n$ holds.

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

Listing 6.2: The lemma `UnchangedSection`

Lemma `UnchangedStep` in Listing 6.3 expresses the simple fact that “unchangedness” is an inductive property.

```
/*@
  lemma
    UnchangedStep{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);
*/
```

Listing 6.3: The lemma `UnchangedStep`

Lemma `UnchangedTransitive` in Listing 6.4 expresses the transitivity of `Unchanged` with respect to program states.

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

Listing 6.4: The lemma `UnchangedTransitive`

6.2. The predicate `MultisetUnchanged`

Various algorithms in this document *rearrange* or *reorder* the elements of a given range such that the number of each element remains unchanged. In other words, reordering leaves the *multiset*[22] of elements in the range unchanged.

We use the predicate `MultisetUnchanged`, defined in Listing 6.5, to formally describe this property. This predicate, which is given in two overloaded versions, relies on the logic function `Count` that is defined in Listing 3.34.

```
/*@
  predicate
    MultisetUnchanged{L1,L2}(value_type* a, integer first, integer last) =
      \forall value_type v;
        Count{L1}(a, first, last, v) == Count{L2}(a, first, last, v);


  predicate
    MultisetUnchanged{L1,L2}(value_type* a, integer n) =
      MultisetUnchanged{L1,L2}(a, 0, n);
*/
```

Listing 6.5: The predicate `MultisetUnchanged`

[22] See `http://en.wikipedia.org/wiki/Multiset`

6.3. The fill algorithm

The fill algorithm in the C++ Standard Library [14, §25.3.6] initializes general sequences with a particular value. For our purposes we have modified the generic implementation to that of an array of type value_type. The signature now reads:

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

6.3.1. Formal specification of fill

Listing 6.6 shows the formal specification of fill in ACSL. We can express the postcondition of fill simply by using the overloaded predicate ConstantRange from Listing 3.25.

```
/*@
  requires valid: \valid(a + (0..n-1));

  assigns a[0..n-1];

  ensures constant: ConstantRange(a, n, val);
*/
void fill(value_type* a, size_type n, value_type val);
```

Listing 6.6: 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 [9, §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 3.

6.3.2. Implementation of fill

Listing 6.7 shows an implementation of fill.

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

Listing 6.7: Implementation of fill

The loop invariant constant expresses that for each iteration the array is filled with the value of val up to the index i of the iteration. Note that we use here again the predicate ConstantRange from Listing 3.25.

6.4. The swap algorithm

The `swap` algorithm [14, §25.3.3] in the C++ Standard Library exchanges the contents of two variables. Similarly, the `iter_swap` algorithm [14, §25.3.3] 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`.

6.4.1. Formal specification of swap

The ACSL specification for the `swap` function is shown in Listing 6.8. The preconditions are formalized by the `requires`-clauses which state that both pointer arguments of the `swap` function must be dereferenceable.

```
/*@
  requires \valid(p);
  requires \valid(q);

  assigns *p;
  assigns *q;

  ensures *p == \old(*q);
  ensures *q == \old(*p);
*/
void swap(value_type* p, value_type* q);
```

Listing 6.8: Formal specification of `swap`

Upon termination of `swap` the entries must be mutually exchanged. We can express those postconditions by using the `ensures`-clause. The expression `\old(*p)` refers to the pre-state of the function contract, whereas by default, a postcondition refers the values after the functions has been terminated.

6.4.2. Implementation of swap

Listing 6.9 shows the usual straight-forward implementation of `swap`. No interspersed ACSL is needed to get it verified by Frama-C.

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

Listing 6.9: Implementation of `swap`

6.5. The swap_ranges algorithm

The `swap_ranges` algorithm in the C++ Standard Library [14, §25.3.3] 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.

This function refers to the previously discussed algorithm `swap`. Thus, `swap_ranges` serves as another example for "modular verification". The specification of `swap` will be automatically integrated into the proof of `swap_ranges`.

6.5.1. Formal specification of `swap_ranges`

Listing 6.10 shows an ACSL specification for the `swap_ranges` 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:  EqualRanges{Old,Here}(a, n, b);
  ensures equal:  EqualRanges{Old,Here}(b, n, a);
*/
void swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 6.10: 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 `separated`-clause 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 `EqualRanges` (see Listing 3.15) together with the label `Old` and `Here` to express the postcondition of `swap_ranges`. In our specification in Listing 6.10, 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.

6.5.2. Implementation of swap_ranges

Listing 6.11 shows an implementation of swap_ranges together with the necessary loop annotations.

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

Listing 6.11: Implementation of swap_ranges

For the postcondition of the specification in Listing 6.10 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 a workaround that 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 from Listing 6.1 is used to express this property.

6.6. The copy algorithm

The `copy` algorithm in the C++ Standard Library [14, §25.3.1] 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 6.12.

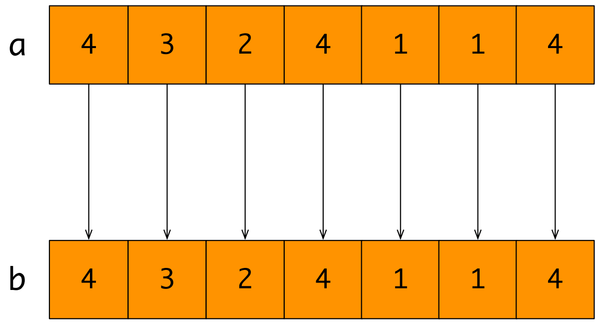

Figure 6.12.: Effects of `copy`

6.6.1. Formal specification of copy

Figure 6.12 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 6.13.

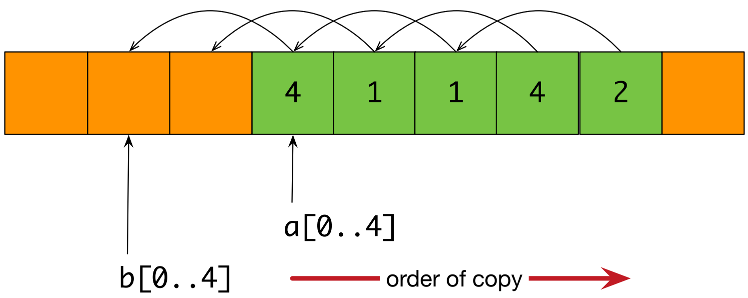

Figure 6.13.: Possible overlap of `copy` ranges

The ACSL specification of `copy` is shown in Listing 6.14. 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:   EqualRanges{Old,Here}(a, n, b);
*/
void copy(const value_type* a, const size_type n, value_type* b);
```

Listing 6.14: Formal specification of copy

Again, we can use the EqualRanges predicate from Section 3.5 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.

6.6.2. Implementation of copy

Listing 6.15 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:     EqualRanges{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 = 0; i < n; ++i) {
    b[i] = a[i];
  }
}
```

Listing 6.15: 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.[23]

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.

[23] 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.

6.7. The `copy_backward` algorithm

The `copy_backward` algorithm in the C++ Standard Library [14, §25.3.1] 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 see in Listing 6.14.

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.[24] 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 6.16 gives an example where `copy_backward`, but *not* `copy`, can be applied.

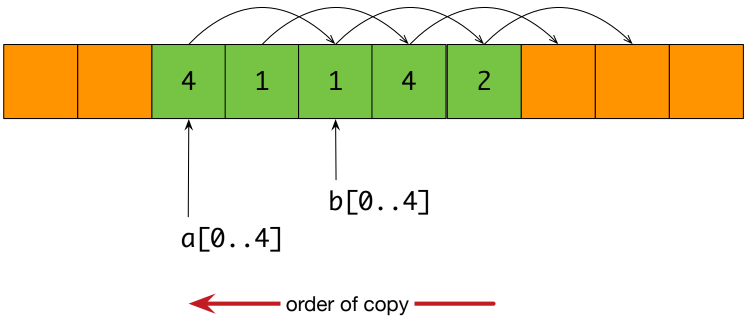

Figure 6.16.: Possible overlap of `copy_backward` ranges

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.

6.7.1. Formal specification of `copy_backward`

The ACSL specification of `copy_backward` is shown in Listing 6.17. 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.

[24] 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: EqualRanges{Old,Here}(a, n, b);
*/
void copy_backward(const value_type* a, size_type n, value_type* b);
```

Listing 6.17: 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 `EqualRanges` predicate from Section 3.5 to express that the array `a` equals `b` after `copy_backward` has been called.

6.7.2. Implementation of `copy_backward`

Listing 6.18 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:     EqualRanges{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 6.18: 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`).

6.8. The reverse_copy algorithm

The reverse_copy algorithm of the C++ Standard Library [14, §25.3.10] 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);
```

6.8.1. The predicate Reverse

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 Listing 6.19 the predicate Reverse (see also Figure 6.20). We also define several overloaded variants of Reverse that provide default values for some of the parameters. These overloaded versions enable us to write more concise ACSL annotations.

```
/*@
  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 6.19: The predicate Reverse

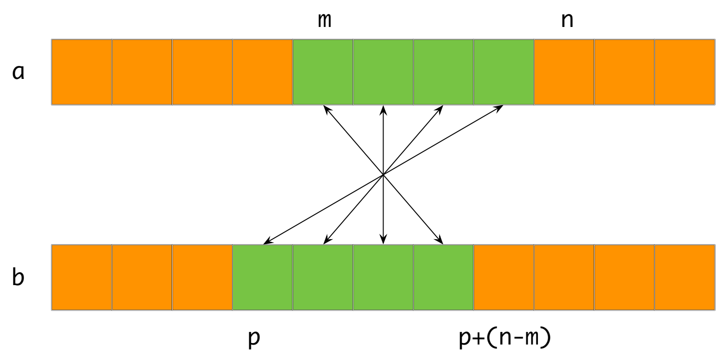

Figure 6.20.: Sketch of predicate Reverse

6.8.2. Formal specification of reverse_copy

The ACSL specification of reverse_copy is shown in Listing 6.21. We use the second version of predicate Reverse from Listing 6.19 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 6.21: Formal specification of reverse_copy

6.8.3. Implementation of reverse_copy

Listing 6.22 shows an implementation of the reverse_copy function. 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 from Listing 6.19.

```
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 6.22: Implementation of reverse_copy

6.9. The `reverse` algorithm

The `reverse` algorithm of the C++ Standard Library [14, §25.3.10] inverts the order of elements in a sequence. Note that `reverse` works *in place*, meaning that it modifies its input sequence. Our modified signature reads:

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

6.9.1. Formal specification of `reverse`

The ACSL specification for the `reverse` function is shown in listing 6.23.

```
/*@
  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 6.23: Formal specification of `reverse`

6.9.2. Implementation of `reverse`

Listing 6.24 shows an implementation of the `reverse` function. Since the `reverse` algorithm operates *in place* we use the `swap` function from Section 6.4 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` (Listing 6.19) and `Unchanged` (Listing 6.1) 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 6.24: Implementation of `reverse`

6.10. The `rotate_copy` algorithm

The `rotate_copy` algorithm of the C++ Standard Library [14, §25.3.11] 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 6.25 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]`.

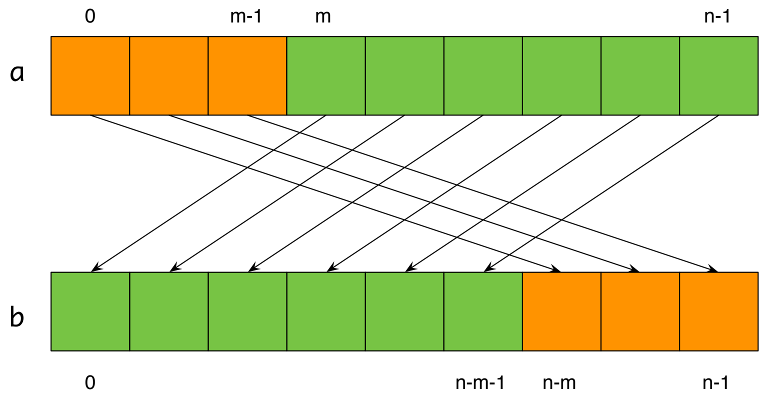

Figure 6.25.: Effects of `rotate_copy`

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);
```

6.10.1. Formal specification of `rotate_copy`

The ACSL specification of `rotate_copy` is shown in Listing 6.26. 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:      EqualRanges{Old,Here}(a, 0, m,   b, n-m);
  ensures right:     EqualRanges{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 6.26: Formal specification of `rotate_copy`

6.10.2. Implementation of rotate_copy

Listing 6.27 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 6.27: Implementation of rotate_copy

6.11. The rotate algorithm

The algorithm rotate is an *in-place* variant of the algorithm rotate_copy of Section 6.10. We have modified the generic specification of rotate [14, §25.3.11] 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);
```

6.11.1. Formal specification of rotate

Figure 6.28 shows informally the behavior of rotate. The figure is of course very similar to the one for rotate_copy (see Figure 6.25). The notable difference is that rotate operates *in place* of the array a[0..n-1].

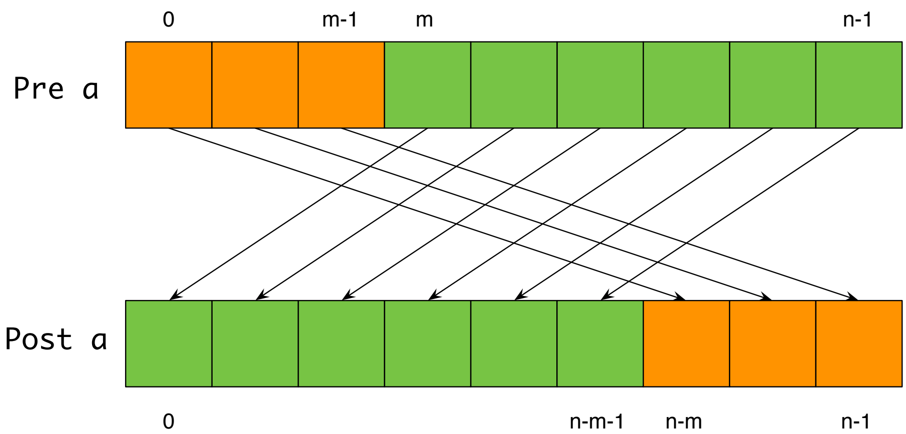

Figure 6.28.: Effects of rotate

The ACSL specification of rotate is shown in Listing 6.29.

```
/*@
  requires valid: \valid(a + (0..n-1));
  requires bound: m <= n;

  assigns a[0..n-1];

  ensures result: \result == n-m;
  ensures left:   EqualRanges{Old,Here}(a, 0, m, n-m);
  ensures right:  EqualRanges{Old,Here}(a, m, n, 0);
*/
size_type rotate(value_type* a, size_type m, size_type n);
```

Listing 6.29: Formal specification of `rotate`

6.11.2. Implementation of `rotate`

Listing 6.30 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* [14, §24.2.6] and which essentially consists of several calls to the algorithm `reverse` of Section 6.9.

Note the statement contract of the final call of `reverse` Listing 6.30. 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:   EqualRanges{Pre,Here}(a, 0, m, n-m);
    //@ assert right:  EqualRanges{Pre,Here}(a, m, n, 0);
  }

  return n - m;
}
```

Listing 6.30: Implementation of `rotate`

6.12. The `replace_copy` algorithm

The `replace_copy` algorithm of the C++ Standard Library [14, §25.3.5] 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.

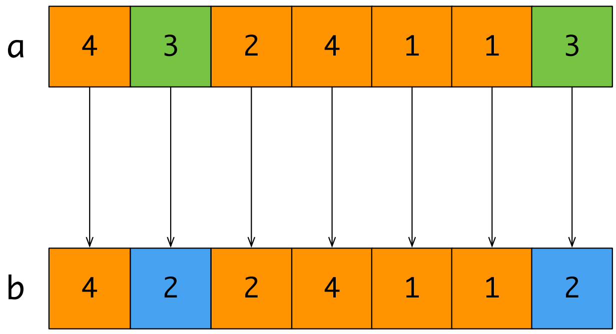

Figure 6.31.: Effects of `replace`

Figure 6.31 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]`.

6.12.1. The predicate `Replace`

We start with defining in Listing 6.32 the predicate `Replace` 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 [9, §2.2]).

```
/*@
  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 6.32: The predicate `Replace`

Listing 6.32 also contains a second, overloaded version of `Replace` which we will use for the specification of the related in-place algorithm `replace` in Section 6.13.

6.12.2. Formal specification of `replace_copy`

Using predicate `Replace` the ACSL specification of `replace_copy` is as simple as in Listing 6.33. Note that we require that the pointer `b` does not refer to elements of the source range `a[0..n-1]` (see also Section 6.6).

```
/*@
  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 replace:   Replace{Old,Here}(a, n, b, v, w);
  ensures result:    \result == n;
*/
size_type
replace_copy(const value_type* a, size_type n, value_type* b,
             value_type v, value_type w);
```

Listing 6.33: Formal specification of the `replace_copy`

6.12.3. Implementation of `replace_copy`

An implementation (including loop annotations) of `replace_copy` is shown in Listing 6.34. Note how the structure of the loop annotations resembles the specification of Listing 6.33.

```
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 invariant unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    b[i] = (a[i] == v ? w : a[i]);
  }

  return n;
}
```

Listing 6.34: Implementation of the `replace_copy` algorithm

6.13. The `replace` algorithm

The `replace` algorithm of the C++ Standard Library [14, §25.3.5] 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`.

6.13.1. Formal specification of `replace`

Using the second predicate `Replace` from Listing 6.32 the ACSL specification of `replace` can be expressed as in Listing 6.35.

```
/*@
  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 6.35: Formal specification of the `replace`

6.13.2. Implementation of `replace`

An implementation of `replace` is shown in Listing 6.36. 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 = 0; i < n; ++i) {
    if (a[i] == v) {
      a[i] = w;
    }
  }
}
```

Listing 6.36: Implementation of the `replace` algorithm

6.14. The `remove_copy` algorithm

The `remove_copy` algorithm of the C++ Standard Library [14, §25.3.8] 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 most important facts of this algorithms are:

1. The return value is the length of the resulting range.
2. 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]`.
3. The algorithm is stable, that is, the relative order of the elements in `b` is the same as in `a`.

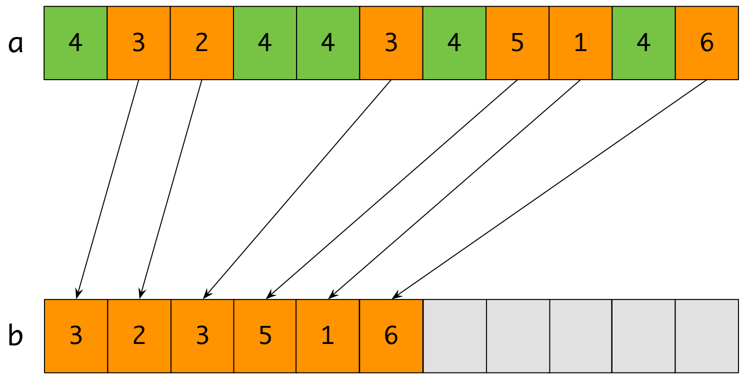

Figure 6.37.: Effects of `remove_copy`

Figure 6.37 shows how `remove_copy` is supposed to copy elements that differ from `v` from range `a` to `b`.

6.14.1. The predicate `MultisetRetainRest`

In order to achieve a concise specification we introduce the overloaded predicate `MultisetRetainRest` (see Listing 6.38). The expression `MultisetRetainRest{K,L}(a, m, b, n, v)` is true if the range `a[0..n-1]` at time `K` contains the same elements as `b[0..m-1]` at time `L`, except possibly for occurrences of `v`; the elements' order may differ in `a` and `b`.

There is also a more general version of `MultisetRetainRest` in Listing 6.38 that is defined over array segments.

```
/*@
  predicate
    MultisetRetainRest{K,L}(value_type* a, integer m1, integer m2,
                            value_type* b, integer n1, integer n2,
                            value_type  v) =

      \forall value_type x;
        x != v ==> Count{K}(a, m1, m2, x) == Count{L}(b, n1, n2, x);


  predicate
    MultisetRetainRest{K,L}(value_type* a, integer m,
                            value_type* b, integer n,
                            value_type  v) =
      MultisetRetainRest{K,L}(a, 0, m, b, 0, n, v);
*/
```

Listing 6.38: The predicate `MultisetRetainRest`

6.14.2. Formal specification of `remove_copy`

Listing 6.39 now shows our first attempt to specify `remove_copy`.

```
/*@
  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  bound:      0 <= \result <= n;
  ensures  size:       \result == n - Count{Old}(a, n, v);
  ensures  retain:     MultisetRetainRest{Old,Here}(a, n, b, \result, v);
  ensures  discard:    !HasValue(b, \result, v);
  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 6.39: Formal specification of `remove_copy`

- The informal specification of `remove_copy` states that the pointer `b` does not reference the range `a[0..n-1]`. This is expressed by demanding `\separated(a + (0..n-1), b)`.
- We use the predicate `MultisetRetainRest` to express that the number of elements different from `v` is the same in the source and target range. Note that this property does not guarantee the stability of `remove_copy` because given e.g. a range $\{1,0,5,2,0,5\}$ and the value `v=0` the expected result of `remove_copy` is the range $\{1,5,2,5\}$. However, since `Count` is invariant under permutations the specification in Listing 6.39 would also allow e.g. the result $\{5,5,1,2\}$.
- The predicate `Unchanged` from Listing 6.1 is used to express that `remove_copy` does not change `b[\result..n-1]`.
- Note the re-use of predicate `HasValue` (Listing 3.4) to express that the target range does not contain the value `v`.

6.14.3. Implementation of remove_copy

An implementation of remove_copy is shown in Listing 6.40.

```
size_type
remove_copy(const value_type* a, size_type n, value_type* b, value_type v)
{
  size_type j = 0;

  /*@
    loop invariant bound:      0 <= j <= i <= n;
    loop invariant size:       j == i - Count{Pre}(a, i, v);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, i, n);
    loop invariant retain:     MultisetRetainRest{Pre,Here}(a, i, b, j, v);
    loop invariant discard:    !HasValue(b, j, v);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, j, n);
    loop assigns i, j, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    if (a[i] != v) {
      b[j++] = a[i];
      //@ assert retain_pre:  MultisetRetainRest{Pre,Here}(a, i,   b, j-1, v);
      //@ assert retain_now:  MultisetRetainRest{Pre,Here}(a, i+1, b, j,   v);
    }
  }

  return j;
}
```

Listing 6.40: Implementation of remove_copy

Not surprisingly, the logical function Count as well as the predicates HasValue, Unchanged, and MultisetRetainRest also appear in the loop invariants of remove_copy. In a previous release of this manual, a series of complex assertions was necessary in order to guide the provers towards the loop invariant retain. In this version we still need the assertions retain_pre and retain_now to automatically verify the said loop invariant in a reasonable time.

Since the precondition sep does not guarantee that the range a[0..n-1] remains unchanged we have to refer to the pre-state of the array a[0..n-1] by using the logic label Pre instead of Here for all references of a where possible. This is necessary despite a[i] retaining its original value until the i-th iteration—a fact that is quickly verified through the loop invariant unchanged.

6.15. The `unique_copy` algorithm

The presentation of `unique_copy` in this section and Section 6.16 closely follows our discussion in an upcoming report of the VESSEDIA project.[25]

The `unique_copy` algorithm of the C++ Standard Library [14, §25.3.9] 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
unique_copy(const value_type* a, size_type n, value_type* b);
```

The requirements of `unique_copy` are:

Requirement	Description
Unique Copy Size	The output range must be able to store the same number of elements as the input range.
Unique Copy Separation	The input range and the output range do not overlap.
Unique Copy Consecutive	Only the first element from every consecutive group of equal elements of the input range is copied into the output range.
Unique Copy Return	The algorithm returns the number of copied elements.
Unique Copy Complexity	At most $n - 1$ comparisons of adjacent elements are performed.

Table 6.41.: Requirements of `unique_copy`

Note that of all these requirements **Unique Copy Consecutive** is the most important one because it describes the functionality of `unique_copy`. Figure 6.42 shows the behaviour of `unique_copy` at hand of an example array.

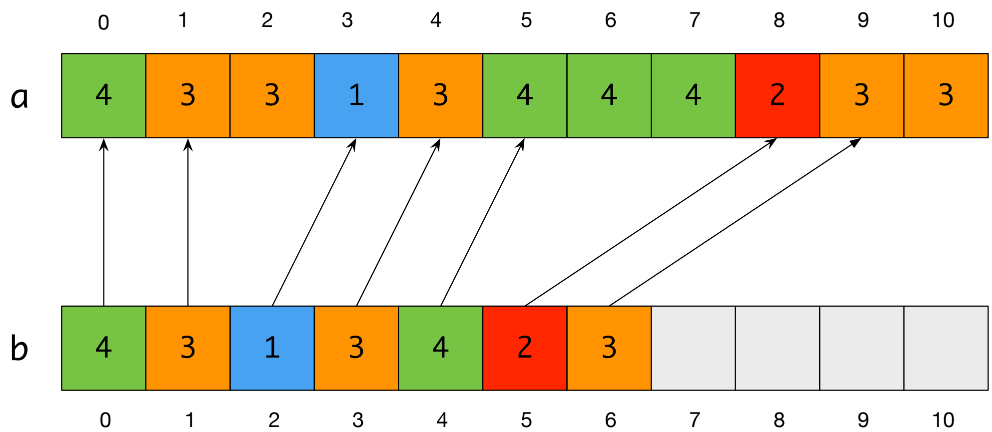

Figure 6.42.: Example of applying `unique_copy`

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

6.15.1. Formal specification of unique_copy

The main purpose of the formal specification in Listing 6.43 is to show that there are no equal neighbors in the output array. This property reflects an important consequence of **Unique Copy Consecutive**. In order to formalize this new property we use the predicate HasEqualNeighbors from Listing 3.12 We use the negation of HasEqualNeighbors from Listing 3.12 in the postcondition unique to verify the absence of equal neighbors throughout the output array b[0..n-1].

This contract also formalizes the requirements **Unique Copy Size** and **Unique Copy Separation** but only partially formalizes **Unique Copy Return** because we only provide some basic bounds for the number of copied elements.

```
/*@
  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:    0 <= \result <= n;
  ensures unique:    !HasEqualNeighbors (b, \result);
  ensures unchanged: Unchanged{Old, Here}(b, \result, n);
 */
size_type
unique_copy(const value_type* a, size_type n, value_type* b);
```

Listing 6.43: A short contract for unique_copy

Note that this contract does not cover all aspects of **Unique Copy Consecutive**. An extended specification and will be discussed in Section 6.16.

6.15.2. Implementation of unique_copy

The basic idea of our C-implementation of unique_copy in Listing 6.44 is to traverse the input array a[0...n-1] and copy an element a[i] to the output array b[0..n-1] whenever it has been detected that it is different from its predecessor a[i-1].

```
size_type
unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;
    b[k] = a[0];
    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant unique:    !HasEqualNeighbors(b, k+1);
      loop invariant unchanged: Unchanged{Pre, Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        b[++k] = val;
      }
    }
    return ++k;
  }
}
```

Listing 6.44: Annotations for a short contract of unique_copy

Assuming a non-empty array, the implementation starts with copying a[0] to b[0]. In order to detect whether the current value a[i] is different from its predecessor we compare it with the most recently copied value b[k]. In order to support the verification of the postcondition unique from Listing 6.43 we add a loop invariant unique to our implementation 6.44.

6.16. The `unique_copy` algorithm reconsidered

In this section we finally tackle the issue of formalizing the requirements of **Unique Copy Consecutive** in ACSL. The main idea is that we

1. capture in ACSL the properties of the partitioning sequence (see Section 6.16.1)
2. use these properties to specify and verify `unique_copy`

In order to get an understanding of our train of though for the verification contract of `unique_copy` we first provide a small formal analysis for `unique_copy`.

6.16.1. A closer look on the properties of `unique_copy`

Figure 6.45 is a slight modification of Figure 6.42. We show here only the indices of the source array whose values are copied into the target array. In addition, we have added another (dashed) arrow to link the indices that correspond to the *one past the end* locations of the input and output ranges, respectively. We use this additional arrow in order to be able to describe all sub sequences of consecutive equal elements in the source array.

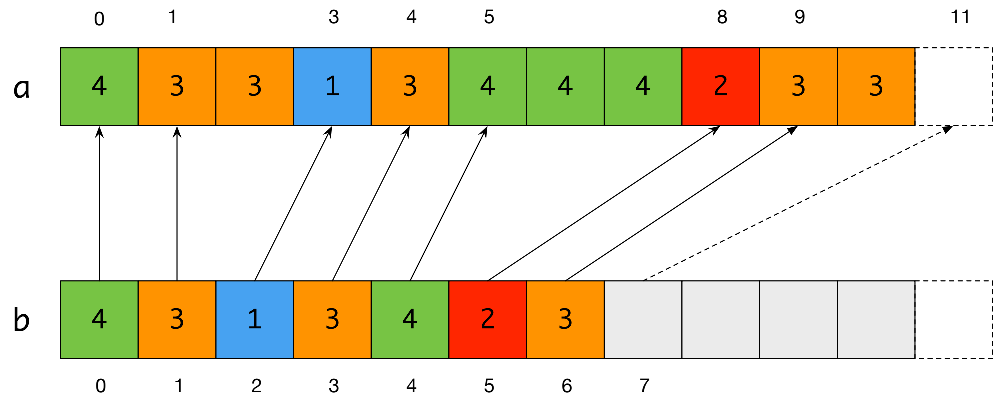

Figure 6.45.: Partitioning the input of `unique_copy`

These arrows between the indices of the array `b` and the array `a` define the following sequence p of eight indices where the index that points one past the end is underlined.

$$p = (0, 1, 3, 4, 5, 8, 9, \underline{11}) \qquad \text{for Figure 6.45}$$

More generally, we refer to the sequence p as *partitioning sequence* of `unique_copy` for the array `a[0..n-1]`. This sequence is characterized by the following properties: If $m + 1$ is the **length of a partitioning sequence**, then we observe that they are **strictly monotone increasing**

$$0 = p_0 < \ldots < p_{m+1} = n \tag{6.1}$$

and that the right-open index intervals

$$[p_i, p_{i+1}) \qquad \forall i : 0 \le i < m$$

mark **consecutive ranges** of equal elements in the source array, that is,

$$\mathtt{a}[p_i] = \mathtt{a}[k] \qquad \forall k : p_i \le k < p_{i+1} \tag{6.2}$$

We also have that the consecutive ranges are **maximal** in the following sense

$$\mathsf{a}[p_i] \neq \mathsf{a}[p_{i+1}] \qquad \forall i : 0 \leq i < m-1 \tag{6.3}$$

and last but not least for the **result of `unique_copy`** it must hold

$$\mathsf{b}[i] = \mathsf{a}[p_i] \qquad \forall i : 0 \leq i < m \tag{6.4}$$

6.16.2. Formalizing the number of elements copied by `unique_copy`

In order to improve our treatment of **Unique Copy Return** we define in Listing 6.46 the logic function `UniqueSize` which computes the number of elements that are to be copied by `unique_copy` from an array `a[0..n-1]`. In other words, `UniqueSize` represents the number m in the inequalities (6.1) of consecutive sub-ranges of equal elements. Our axioms in Listing 6.46 essentially give a *recursive* definition of `UniqueSize`.

```
/*@
  axiomatic UniqueSizeAxiomatic
  {
    logic integer UniqueSize(value_type* a, integer n) reads a[0..n-1];

    axiom UniqueSizeEmpty:
      \forall value_type *a, integer n;
        n <= 0  ==>  UniqueSize(a, n) == 0;

    axiom UniqueSizeOne:
      \forall value_type *a, integer n;
        n == 1  ==>  UniqueSize(a, n) == 1;

    axiom UniqueSizeEqual:
      \forall value_type *a, integer n;
        0 < n ==> a[n-1] == a[n] ==> UniqueSize(a, n+1) == UniqueSize(a, n);

    axiom UniqueSizeDiffer:
      \forall value_type *a, integer n;
        0 < n ==> a[n-1] != a[n] ==>  UniqueSize(a, n+1) == UniqueSize(a, n) + 1;

    axiom UniqueSizeRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n) ==> UniqueSize{K}(a, n) == UniqueSize{L}(a, n);
  }
*/
```

Listing 6.46: Axiomatic description of the function `UniqueSize`

Note that the axioms of `UniqueSize` cover also negative array size as can be seen in the definition of `UniqueSizeEmpty` in Listing 6.46. Moreover, they ignore whether the involved pointers can be dereferenced. The issue here is that the function `UniqueSize` must be defined as a *total function* regardless of the fact whether the involved function arguments make sense in C-code. For more details we refer to the rules for logic definitions in the description of ACSL [9, §2.2.2].

The following ACSL lemmas `UniqueSizeBound` (Listing 6.47) formulates some simple bounds on the number of copied elements. As trivial as these inequalities might look like, their not too complicated proofs rely on mathematical induction. Since automatic theorem provers are often not capable of performing induction proofs, we have proven this lemma with the interactive theorem prover Coq.

```
/*@
  lemma UniqueSizeBound:
    \forall value_type *a, integer n;
      0 <= n  ==>  0 <= UniqueSize(a, n) <= n;
*/
```

Listing 6.47: Lemma UniqueSizeBound

6.16.3. Formalizing the properties of the partitions of unique_copy

Listing 6.48 shows our axiomatic description of the function UniquePartition.

```
/*@
  axiomatic UniquePartitionAxiomatic
  {
    logic integer
      UniquePartition(value_type* a, integer n, integer i) reads a[0..n-1];

    axiom UniquePartitionEmpty:
      \forall value_type *a, integer n, i;
        n <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionLeft:
      \forall value_type *a, integer n, i;
        0 < n  ==>  i <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionRight:
      \forall value_type *a, integer n, i;
        0 < n  ==>  UniqueSize(a, n) <= i  ==>  UniquePartition(a, n, i) == n;

    axiom UniquePartitionMonotone:
      \forall value_type *a, integer n, i, j;
        0 <= i < j <= UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) < UniquePartition(a, n, j);

    axiom UniquePartitionSegment:
      \forall value_type *a, integer n, i, k;
        0 <= i < UniqueSize(a, n) ==>
        ConstantRange(a, UniquePartition(a, n, i), UniquePartition(a, n, i+1));

    axiom UniquePartitionMaximal:
      \forall value_type *a, integer n, i;
        0 <= i < UniqueSize(a, n) - 1  ==>
        a[UniquePartition(a, n, i)] != a[UniquePartition(a, n, i+1)];

    axiom UniquePartitionEqual:
      \forall value_type *a, integer n, m, i;
        n < m  ==>  0 <= i < UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) == UniquePartition(a, m, i);

    axiom UniquePartitionRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n) ==>
          UniquePartition{K}(a, n, i) == UniquePartition{L}(a, n, i);
  }
*/
```

Listing 6.48: Axiomatic description of the function UniquePartition

Before we begin to relate the axioms from Listing 6.48 to the formulas from Section 6.16.1 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.

- The monotonicity conditions (6.1) are described by the axioms `UniquePartitionEmpty`, `UniquePartitionLeft`, `UniquePartitionRight` and `UniquePartitionMonotone`.
- Equation (6.2) is represented by the axiom `UniquePartitionSegment`. Note the use of an overloaded version of predicate `ConstantRange` from Listing 3.25.
- Inequality (6.3) is described by axiom `UniquePartitionMaximal`.
- Axiom `UniquePartitionEqual` expresses that the value of `UniquePartition(a, n, i)` does not depend on the size of the array.
- Axiom `UniquePartitionRead`, finally states that `UniquePartition` is independent from the particular programme state in which it is used—as long as the respective array elements are equal in both states.

Listing 6.49 shows some lemmas that we have added to guide the automatic verification of the version of `unique_copy` presented in the Listings 6.51 and 6.53.

```
/*@
  lemma UniquePartitionZero:
     \forall value_type *a, integer n;
       UniquePartition(a, n, 0) == 0;

  lemma UniquePartitionLowerBound:
     \forall value_type *a, integer n, i;
       0 < n ==>
       0 <= i < UniqueSize(a, n) ==>
       0 <= UniquePartition(a, n, i);

  lemma UniquePartitionUpperBound:
     \forall value_type *a, integer n, i;
       0 < n ==>
       0 <= i < UniqueSize(a, n) ==>
       UniquePartition(a, n, i) < n;

  lemma UniquePartitionDiffer:
    \forall value_type *a, integer i, k, n;
     UniquePartition(a, n, k-1) < i <= UniquePartition(a, n, k)
     ==>  a[i-1] != a[i]  ==>  i == UniquePartition(a, n, k);
*/
```

Listing 6.49: Some lemmas for `UniquePartition`

6.16.4. Formal specification

With the definitions of the logic functions `UniqueSize` and `UniquePartition` we can now formulate the ACSL predicate `Unique` from Listing 6.50. This predicate reflects Equation (6.4) and therefore will serve a prominent role in our complete contract of `unique_copy`.

```
/*@
  predicate
    Unique(value_type* a, integer n, value_type* b) =
      \forall integer k; 0 <= k < UniqueSize(a, n) ==>
        b[k] == a[UniquePartition(a, n, k)];
*/
```

Listing 6.50: The predicate `Unique`

Listing 6.51 shows how we use the predicate `Unique` in the postcondition `unique` in order to formally specify **Unique Copy Consecutive** for `unique_copy`. It also shows how we use the logic function `UniqueSize` in the postcondition `size` to properly capture **Unique Copy Return**.

```
/*@
  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 size:      \result == UniqueSize(a, n);
  ensures unique:    Unique(a, n, b);
  ensures unchanged: Unchanged{Old, Here}(a, n);
  ensures unchanged: Unchanged{Old, Here}(b, \result, n);
 */
size_type
unique_copy(const value_type* a, size_type n, value_type* b);
```

Listing 6.51: An extended for `unique_copy`

A natural question is whether our postcondition `unique` is a generalization of the postcondition with the same name from Listing 6.43. Fortunately, this question can be answered in the affirmative. In fact, Lemma `UniqueImpliesNoEqualNeighbors` from Listing 6.52 states exactly the desired implication.

```
/*@
  lemma UniqueImpliesNoEqualNeighbors:
    \forall value_type *a, *b, integer n;
       Unique(a, n, b)  ==>  !HasEqualNeighbors(b, UniqueSize(a, n));
*/
```

Listing 6.52: The predicate `UniqueImpliesNoEqualNeighbors`

6.16.5. Implementation of unique_copy

Listing 6.53 shows that we need considerably more annotation in order to verify the contract from Listing 6.51.

```
size_type
unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (n == 0u) {
    return n;
  }
  else {
    size_type k = 0u;
    b[k] = a[0];
    //@ assert mapping:   0 == UniquePartition(a, n, k);

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant size:      k+1 == UniqueSize(a, i);
      loop invariant copy:      b[k] == a[i-1];
      loop invariant mapping:   UniquePartition(a, n, k) < i;
      loop invariant mapping:   i <= UniquePartition(a, n, k+1);
      loop invariant range:     ConstantRange(a, UniquePartition(a, n, k), i);
      loop invariant unique:    Unique(a, i, b);
      loop invariant unchanged: Unchanged{Pre, Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];
      if (b[k] != val) {
        //@ assert distinct:  a[i-1] != a[i];
        //@ ghost Before:
        b[++k] = val;
        //@ assert unchanged: Unchanged{Before,Here}(b, k);
        //@ assert unchanged: Unchanged{Before,Here}(a, n);
        //@ assert mapping:   i == UniquePartition(a, n, k);
        //@ assert range:     ConstantRange(a, UniquePartition(a, n, k), i+1);
        //@ assert size:      k == UniqueSize(a, i);
        //@ assert unique:    Unique(a, i, b);
      }
    }

    return ++k;
  }
}
```

Listing 6.53: Annotations for the extended specification of unique_copy

Listing 6.53 contains quite a few annotations. We concentrate here only on some of them. The basic idea of our assertions is: whenever the expression `b[k] != val` evaluates as true in the if-statement, then the index `i` enters a new partition. Note the assertion `distinct` which indicates that the evaluation of `b[k] != val` is equivalent to checking whether `a[i-1] != a[i]` holds.

We also rely on Lemma `UniquePreserve` from Listing 6.54 in order to verify the assertion `unique`.

```
/*@
  lemma UniquePreserve{K,L}:
    \forall value_type *a, *b, integer i, k, n;
      0 <= k <= i < n           ==>
      k == UniqueSize{L}(a, i) ==>
      Unique{K}(a, i, b)        ==>
      Unchanged{K,L}(b, k)      ==>
      Unchanged{K,L}(a, n)      ==>
      Unique{L}(a, i, b);
*/
```

Listing 6.54: The lemma `UniquePreserve`

The assertions `unchanged` in Listing 6.53 formulates preconditions for Lemma `UniquePreserve`. These assertions in turn rely on the C-label `Before`. As this label is not relevant for the execution of the code we have declared it as so-called *ghost code*. As explained in the ACSL documentations [9, §2.12], variables and statements that appear in comments marked as

```
/*@ ghost ... */
```

or

```
//@ ghost ...
```

are treated as C variables and statements, however, they are visible only in the specifications. In Listing 6.53 we declare the label `Before` as ghost. We could also have resorted to ACSL *statement contracts* [9, §2.4.4] but opted here for using ghost code.

6.17. The `random_shuffle` algorithm

The `random_shuffle` algorithm in the C++ Standard Library [14, §25.3.12] 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 random_shuffle(value_type* a, size_type n);
```

Figure 6.55 illustrates an example run of `random_shuffle`. In this figure, the values 1, 2, 3, and 4 occur twice, once, once, and three times, respectively, both before and after the `random_shuffle` run. This expresses that the range has been reordered.

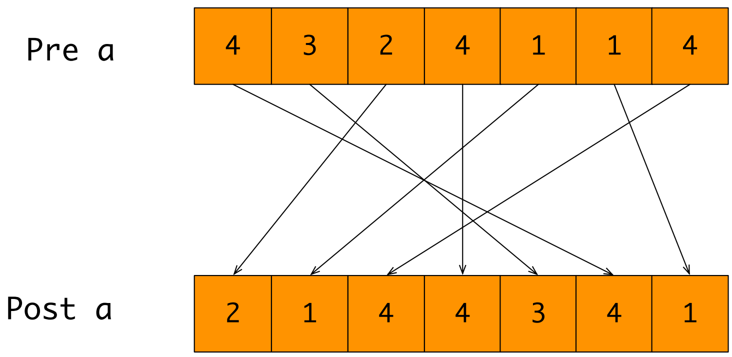

Figure 6.55.: Effects of `random_shuffle`

6.17.1. Formal specification of `random_shuffle`

The ACSL specification of `random_shuffle` is shown in Listing 6.56. The `random_shuffle` algorithm expects that the range `a` is valid for reading and writing. We use the predicate `MultisetUnchanged` defined in Listing 6.5 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 `random_seed` contains a seed for the random number generator used to randomize the shuffle. By specifying that the function assigns to `random_seed` we capture that the function may return a different permutation every time.

```
extern unsigned short random_seed[3];

/*@
  requires \valid(a + (0..n-1));

  assigns a[0..n-1];
  assigns random_seed[0..2];

  ensures  MultisetUnchanged{Old,Here}(a,n);
*/
void random_shuffle(value_type* a, size_type n);
```

Listing 6.56: Formal specification of `random_shuffle`

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 `random_shuffle` is required to be random. However, ACSL does currently not support the specification of temporal properties related to repeated call results.

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 [16, 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 [17].

6.17.2. Implementation of `random_shuffle`

Listing 6.57 shows an implementation of the `random_shuffle` function. It repeatedly calls the function `swap` from Section 6.4 to *transpose* (randomly) selected elements. The loop invariants `reorder` and `unchanged` of `random_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 two auxiliary assertions `reorder` which use the *ghost label* `Before`, to guide the automatic verification the loop invariant `reorder`.

```
void random_shuffle(value_type* a, size_type n)
{
  if (0u < n) {

    /*@
      loop invariant bounds:    1 <= i <= n;
      loop invariant reorder:   MultisetUnchanged{Pre,Here}(a,0,i);
      loop invariant unchanged: Unchanged{Pre,Here}(a,i,n);
      loop assigns   i, a[0..n-1], random_seed[0..2];
      loop variant   n - i;
    */
    for (size_type i = 1; i < n; ++i) {
      const size_type j = random_number(i) + 1;

      //@ ghost Before:
      swap(&a[j], &a[i]);
      //@ assert reorder: MultisetUnchanged{Before,Here}(a, 0, j);
      //@ assert reorder:          Unchanged{Before,Here}(a, j+1, i);
      //@ assert reorder: MultisetUnchanged{Before,Here}(a, j+1, i);
    }
  }
}
```

Listing 6.57: Implementation of `random_shuffle`

Verifying a random number generator

Our implementation presupposes a random-number generator named `random_number` which is specified in Listing 6.58. As in the case of `random_shuffle` itself, we do not formulate specific properties of randomness and only require its result to be in the specified range `[0..n-1]`. Again, the `assigns` clause to the array `random_seed` models the dependency on an external state.

```
extern unsigned short random_seed[3];

/*@
  requires 0 < n;
  assigns random_seed[0..2];
  ensures 0 <= \result < n;
*/
size_type random_number(size_type n);
```

Listing 6.58: Formal specification of `random_number`

Internally, the function `random_number` uses a custom implementation of the POSIX.1 random number generator `lrand48()`.[26] For sake of completeness, we have included our implementation of `lrand48()` (see Listing 6.59).

```
/*@
  lemma
    random_number_modulo:
      \forall unsigned long long a;
        (a % (1 << 48)) < (1 << 48);
*/

unsigned short random_seed[3] = { 0x243f, 0x6a88, 0x85a3 };

// see IEEE 1003.1-2008, 2016 Edition for specification
/*@
  assigns random_seed[0..2];
  ensures lower: 0 <= \result;
  ensures upper: \result <= 0x7fffffff;
*/
static long my_lrand48(void)
{
  unsigned long long state = (unsigned long long)random_seed[0] << 32
                           | (unsigned long long)random_seed[1] << 16
                           | (unsigned long long)random_seed[2];

  state = (0x5deece66dull * state + 0xbull) % (1ull << 48);
  //@ assert lower: state < (1 << 48);

  long result = state / (1ull << 17);
  //@ assert lower: 0 <= result;

  random_seed[0] = state >> 32 & 0xffff;
  random_seed[1] = state >> 16 & 0xffff;
  random_seed[2] = state >>  8 & 0xffff;

  return result;
}

size_type random_number(size_type n)
{
  return my_lrand48() % n;
}
```

Listing 6.59: Implementation of `random_number`

[26] See `http://pubs.opengroup.org/onlinepubs/9699919799/functions/lrand48.html`

Note the custom ACSL lemma `random_number_modulo` which we introduced to support the verification of the assertions `lower`.

The random number generator is a linear congruence generator with a 48 bit state and the iteration procedure

$$x_{n+1} = ax_n + c \bmod 2^{48} \tag{6.5}$$

where $a = 25214903917$ and $c = 11$ are relatively prime integers.

As a part of the iteration procedure in Equation (6.5) 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. [11, §6.2.5, 9]).

7. 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 [14, §26.7] 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 7.1: Limits of `value_type`

In order to refer to potential arithmetic overflows we introduce the two constants shown in Listing 7.1 which refer to the numeric limits of `value_type` (see also Section 1.3).

We consider the following algorithms.

- `iota` writes sequentially increasing values into a range (Section 7.1 on Page 116)
- `accumulate` computes the sum of the elements in a range (Section 7.2 on Page 118)
- `inner_product` computes the inner product of two ranges (Section 7.3 on Page 121)
- `partial_sum` computes the sequence of partial sums of a range (Section 7.4 on Page 124)
- `adjacent_difference` computes the differences of adjacent elements in a range (Section 7.5 on Page 127)

The formal specifications of these algorithms raise new questions. In particular, we now have to deal with arithmetic overflows in `value_type`.

7.1. The iota algorithm

The iota algorithm in the C++ Standard Library [14, §26.7.6] 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 val);
```

Starting at val, 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 val.

7.1.1. Formal specification of iota

The specification of iota relies on the logic function Iota that is defined in Listing 7.2.

```
/*@
  predicate
   Iota(value_type* a, integer n, value_type v) =
     \forall integer i; 0 <= i < n ==> a[i] == v+i;
*/
```

Listing 7.2: Logic function Iota

The ACSL specification of iota is shown in Listing 7.3. It uses the logic function Iota in order to express the postcondition increment.

```
/*@
  requires valid:  \valid(a + (0..n-1));
  requires limit:  val + n <= VALUE_TYPE_MAX;

  assigns a[0..n-1];

  ensures increment: Iota(a, n, val);
*/
void iota(value_type* a, size_type n, value_type val);
```

Listing 7.3: 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 7.1). In order to avoid integer overflows the sum val+n must not be greater than the constant VALUE_TYPE_MAX.

7.1.2. Implementation of `iota`

Listing 7.4 shows an implementation of the `iota` function.

```
void iota(value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant limit:     val == \at(val, Pre) + i;
    loop invariant increment: Iota(a, i, \at(val, Pre));

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

Listing 7.4: Implementation of `iota`

The loop invariant `increment` describes that in each iteration of the loop the current value `val` is equal to the sum of the value `val` in state of function entry and the loop index `i`. We have to refer here to `\at(val,Pre)` which is the value on entering `iota`.

7.2. The `accumulate` algorithm

The `accumulate` algorithm in the C++ Standard Library [14, §26.7.2] 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

$$\texttt{init} + \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i]$$

This implies that `accumulate` will return `init` for an empty range.

7.2.1. Axiomatic definition of accumulating over an array

As in the case of `count` (see Section 3.9) we specify `accumulate` by first defining a *logic function* `Accumulate` that formally defines the summation of elements in an array.

```
/*@
   axiomatic AccumulateAxiomatic
   {
     logic value_type
     Accumulate{L}(value_type* a, integer n, value_type init) reads a[0..n-1];

     axiom
       AccumulateEmpty{L}:
         \forall value_type *a, init, integer n;
           n <= 0 ==> Accumulate(a, n, init) == init;

     axiom
       AccumulateNext{L}:
         \forall value_type *a, init, integer n;
           0 < n ==> Accumulate(a, n, init) == Accumulate(a, n-1, init) + a[n-1];

     axiom
       AccumulateRead{K,L}:
         \forall value_type *a, init, integer n;
           Unchanged{K,L}(a, n) ==>
           Accumulate{K}(a, n, init) == Accumulate{L}(a, n, init);
   }
*/
```

Listing 7.5: The logic function `Accumulate`

With this definition the following equation holds for $n \geq 0$

$$\texttt{Accumulate}(\texttt{a}, \texttt{n}, \texttt{init}) = \texttt{init} + \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{7.1}$$

Both the `reads` clause and the axiom `AccumulateRead` in Listing 7.5 express that the result of the `Accumulate` function only depends on the values of `a[0..n-1]`.

Listing 7.6 shows an overloaded version of `Accumulate` that uses 0 as default value of `init`. Included in this listing is also a property corresponding to axiom `AccumulateNext` from Listing 7.5, here given as a lemma. We will use this version for the specification of the algorithm `partial_sum` (see Section 7.4).

Thus, for the overloaded version of `Accumulate` we have

$$\texttt{Accumulate}(\texttt{a},\texttt{n}) = \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{7.2}$$

```
/*@
  logic value_type Accumulate{L}(value_type* a, integer n) =
    Accumulate{L}(a, n, (value_type) 0);

  lemma
    AccumulateDefault0{L}:
      \forall value_type* a; Accumulate(a, 0) == 0;

  lemma
    AccumulateDefault1{L}:
      \forall value_type* a; Accumulate(a, 1) == a[0];

  lemma
    AccumulateDefaultNext{L}:
      \forall value_type* a, integer n;
        n >= 0 ==> Accumulate(a, n+1) == Accumulate(a, n) + a[n];

  lemma
    AccumulateDefaultRead{L1,L2}:
      \forall value_type *a, integer n;
        Unchanged{L1,L2}(a, n) ==>
        Accumulate{L1}(a, n) == Accumulate{L2}(a, n);
*/
```

Listing 7.6: An overloaded version of `Accumulate`

7.2.2. Preventing numeric overflows for accumulate

Before we present our formal specification of `accumulate` we introduce in Listing 7.7 a predicate `AccumulateBounds` that we will subsequently use in order to compactly express requirements that exclude numeric overflows while accumulating value.

```
/*@
  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;

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

Listing 7.7: The overloaded predicate `AccumulateBounds`

Predicate AccumulateBounds expresses that for $0 \leq i < n$ the *partial sums*

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \texttt{a}[k] \tag{7.3}$$

do not overflow. If one of them did, one couldn't guarantee that the result of accumulate equals the mathematical description of Accumulate.

Note that we also provide a second (overloaded) version of AccumulateBounds which uses a default value 0 for init.

7.2.3. Formal specification of accumulate

Using the logic function Accumulate and the predicate AccumulateBounds, the ACSL specification of accumulate is then as simple as shown in Listing 7.8.

```
/*@
  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 7.8: Formal specification of accumulate

7.2.4. Implementation of accumulate

Listing 7.9 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 = 0; i < n; ++i) {
    //@ assert rte_help: init + a[i] == Accumulate(a, i+1, \at(init,Pre));
    init = init + a[i];
  }

  return init;
}
```

Listing 7.9: Implementation of accumulate

Note that loop invariant partial claims that in the *i*-th iteration step result equals the accumulated value of Equation (7.3). This depends on the property bounds in Listing 7.8 which expresses that there is no numeric overflow when updating the variable init.

7.3. The `inner_product` algorithm

The `inner_product` algorithm in the C++ Standard Library [14, §26.7.3] computes the *inner product*[27] 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

$$\mathtt{init} + \sum_{i=0}^{\mathtt{n}-1} \mathtt{a}[i] \cdot \mathtt{b}[i]$$

thus, `inner_product` will return `init` for empty ranges.

7.3.1. The logic function `InnerProduct`

As in the case of `accumulate` (see Section 7.2) we specify `inner_product` by first defining a *logic function* `InnerProduct` that formally defines the summation of the element-wise product of two arrays.

```
/*@
   axiomatic InnerProductAxiomatic
   {
     logic integer
     InnerProduct{L}(value_type* a, value_type* b, integer n,
                     value_type init) reads a[0..n-1], b[0..n-1];

     axiom
       InnerProductEmpty{L}:
         \forall value_type *a, *b, init, integer n;
           n <= 0 ==> InnerProduct(a, b, n, init) == init;

     axiom
       InnerProductNext{L}:
         \forall value_type *a, *b, init, integer n;
          0 < n  ==> InnerProduct(a, b, n, init) ==
                     InnerProduct(a, b, n-1, init) + (a[n-1] * b[n-1]);

     axiom
       InnerProductRead{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 7.10: The logic function `InnerProduct`

Both Axiom `InnerProductRead` and the `reads` clause serve the same purpose in that they express that the result of the `InnerProduct` only depends on the values of `a[0..n-1]` and `b[0..n-1]`.

[27] Also referred to as *dot product*, see `http://en.wikipedia.org/wiki/Dot_product`

7.3.2. Preventing numeric overflows for inner_product

Before we present our formal specification of inner_product we introduce in Listing 7.11 two predicates that we will use subsequently in order to compactly express requirements that exclude numeric overflows while computing the inner product.

```
/*@
  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;
*/
```

Listing 7.11: The predicates ProductBounds and InnerProductBounds

Predicate ProductBounds expresses that for $0 \leq i < n$ the products

$$\mathsf{a}[i] \cdot \mathsf{b}[i] \tag{7.4}$$

do not overflow. Predicate InnerProductBounds, on the other hand, states that for $0 \leq i < n$ the *partial sums*

$$\mathtt{init} + \sum_{k=0}^{\mathtt{i}} \mathsf{a}[k] \cdot \mathsf{b}[k] \tag{7.5}$$

do not overflow.

Otherwise, one cannot guarantee that the result of inner_product equals the mathematical description of InnerProduct.

7.3.3. Formal specification of inner_product

Using the logic function InnerProduct, we specify inner_product as shown in Listing 7.12. 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 7.12: Formal specification of `inner_product`

7.3.4. Implementation of `inner_product`

Listing 7.13 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 = 0; 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 7.13: 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 (7.5). This depends of course on the properties `bounds` in Listing 7.12, which express that there is no arithmetic overflow when computing the updates of the variable `init`.

7.4. The partial_sum algorithm

The `partial_sum` algorithm in the C++ Standard Library [14, §26.7.4] 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

$$\begin{aligned} \mathsf{b}[0] &= \mathsf{a}[0] \\ \mathsf{b}[1] &= \mathsf{a}[0] + \mathsf{a}[1] \\ &\vdots \\ \mathsf{b}[n-1] &= \mathsf{a}[0] + \mathsf{a}[1] + \ldots + \mathsf{a}[n-1] \end{aligned}$$

More concisely, for $0 \leq i < n$ holds

$$\mathsf{b}[i] = \sum_{k=0}^{\mathtt{i}} \mathsf{a}[k] \tag{7.6}$$

7.4.1. The predicate PartialSum

Equations (7.6) and (7.2) suggest that we define the ACSL predicate `PartialSum` in Listing 7.14 by using the logic function `Accumulate` from Listing 7.6. Listing 7.14.

```
/*@
  predicate
    PartialSum{L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n ==> Accumulate(a, i+1) == b[i];
*/
```

Listing 7.14: The predicate `PartialSum`

7.4.2. Formal specification of partial_sum

Using the predicates `PartialSum` and `AccumulateBounds`, we specify `partial_sum` as shown in Listing 7.15.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires valid:      \valid(b + (0..n-1));
  requires separated: \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:     AccumulateBounds(a, n+1);

  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 7.15: Formal specification of `partial_sum`

Our specification requires 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*.

7.4.3. Implementation of `partial_sum`

Listing 7.16 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) {
    b[0] = a[0];

    /*@
       loop invariant bound:       1 <= i <= n;
       loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
       loop invariant accumulate: b[i-1] == Accumulate(a, i);
       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) {
      //@ ghost Enter:
      b[i] = b[i - 1u] + a[i];
      //@ assert unchanged:  a[i] == \at(a[i],Enter);
      //@ assert unchanged:  Unchanged{Enter,Here}(a, i);
      //@ assert unchanged:  Unchanged{Enter,Here}(b, i);
    }
  }

  return n;
}
```

Listing 7.16: Implementation of `partial_sum`

In order to facilitate the automatic verification of `partial_sum`, we had to add the assertions `unchanged` and provide the lemmas of Listing 7.17.

7.4.4. Additional lemmas

The lemmas shown in Listing 7.17 are needed for the verification of `partial_sum` and the algorithms in Sections 7.6 and 7.7.

```
/*@
  lemma
    PartialSumSection{K}:
      \forall value_type *a, *b, integer m, n;
      0 <= m <= n             ==>
      PartialSum{K}(a, n, b) ==>
      PartialSum{K}(a, m, b);

  lemma
    PartialSumUnchanged{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
    PartialSumStep{L}:
      \forall value_type *a, *b, integer n;
        1 <= n                      ==>
        PartialSum(a, n, b)         ==>
        b[n] == Accumulate(a, n+1) ==>
        PartialSum(a, n+1, b);

  lemma
    PartialSumStep2{K,L}:
      \forall value_type *a, *b, integer n;
        1 <= n                                 ==>
        PartialSum{K}(a, n, b)                 ==>
        Unchanged{K,L}(a, n+1)                 ==>
        Unchanged{K,L}(b, n)                   ==>
        \at(b[n] == Accumulate(a, n+1),L)  ==>
        PartialSum{L}(a, n+1, b);
*/
```

Listing 7.17: The lemma `PartialSumStep`

7.5. The `adjacent_difference` algorithm

The `adjacent_difference` algorithm in the C++ Standard Library [14, §25.7.5] 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

$$\begin{aligned}
\mathsf{b}[0] &= \mathsf{a}[0] \\
\mathsf{b}[1] &= \mathsf{a}[1] - \mathsf{a}[0] \\
&\vdots \\
\mathsf{b}[n-1] &= \mathsf{a}[n-1] - \mathsf{a}[n-2]
\end{aligned}$$

If we form the partial sums of the sequence `b` we find that

$$\begin{aligned}
\mathsf{a}[0] &= \mathsf{b}[0] \\
\mathsf{a}[1] &= \mathsf{b}[0] + \mathsf{b}[1] \\
&\vdots \\
\mathsf{a}[n-1] &= \mathsf{b}[0] + \mathsf{b}[1] + \ldots + \mathsf{b}[n-1]
\end{aligned}$$

Thus, we have for $0 \leq i < n$

$$\mathsf{a}[i] = \sum_{k=0}^{\mathsf{i}} \mathsf{b}[k] \tag{7.7}$$

which means that applying `partial_sum` on the output of `adjacent_difference` produces the original input of `adjacent_difference`.

Conversely, if `a[0..n-1]` and `b[0..n-1]` are the input and output of `partial_sum`, then we have

$$\begin{aligned}
\mathsf{b}[0] &= \mathsf{a}[0] \\
\mathsf{b}[1] &= \mathsf{a}[0] + \mathsf{a}[1] \\
&\vdots \\
\mathsf{b}[n-1] &= \mathsf{a}[0] + \mathsf{b}[1] + \ldots + \mathsf{b}[n-1]
\end{aligned}$$

from which we can conclude

$$\begin{aligned}
\mathsf{a}[0] &= \mathsf{b}[0] \\
\mathsf{a}[1] &= \mathsf{b}[1] - \mathsf{b}[0] \\
&\vdots \\
\mathsf{a}[n-1] &= \mathsf{b}[n-1] - \mathsf{b}[n-2]
\end{aligned} \tag{7.8}$$

We will verify these claims in Sections 7.6 and 7.7.

7.5.1. The predicate AdjacentDifference

We define the predicate AdjacentDifference in Listing 7.19 by first introducing the logic function Difference (Listing 7.18).

```
/*@
   axiomatic DifferenceAxiomatic
   {
      logic value_type
      Difference{L}(value_type* a, integer n) reads a[0..n];

      axiom
        DifferenceEmptyOrSingle{L}:
          \forall value_type *a, integer n;
            n <= 0 ==> Difference(a, n) == a[0];

      axiom
        DifferenceNext{L}:
          \forall value_type *a, integer n;
            0 < n ==> Difference(a, n) == a[n] - a[n-1];

      axiom
        DifferenceRead{K,L}:
          \forall value_type *a, integer n;
            Unchanged{K,L}(a, 1+n) ==> Difference{K}(a, n) == Difference{L}(a, n);
   }
*/
```

Listing 7.18: The logic function Difference

```
/*@
  predicate
    AdjacentDifference{L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n ==> b[i] == Difference(a, i);
*/
```

Listing 7.19: The predicate AdjacentDifference

7.5.2. Formal specification of adjacent_difference

We introduce here the predicate AdjacentDifferenceBounds (Listing 7.20) that captures conditions that prevent numeric overflows while computing difference of the form a[i] - a[i-1].

```
/*@
  predicate
    AdjacentDifferenceBounds(value_type* a, integer n) =
      \forall integer i; 1 <= i < n ==>
        VALUE_TYPE_MIN <= Difference(a, i) <= VALUE_TYPE_MAX;
*/
```

Listing 7.20: The predicate AdjacentDifferenceBounds

Using the predicates AdjacentDifference and AdjacentDifferenceBounds we can provide a concise formal specification of adjacent_difference (Listing 7.21). As in the case of the specification of partial_sum 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 separated:  \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 7.21: Formal specification of `adjacent_difference`

7.5.3. Implementation of `adjacent_difference`

Listing 7.22 shows an implementation of `adjacent_difference` with corresponding loop annotations.

In order to achieve the verification of the loop invariant `difference` we added

- the assertions `bound` and `difference`
- the lemmas `AdjacentDifferenceStep` and `AdjacentDifferenceSection` from Listing 7.23
- 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[0] = a[0];

    /*@
       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 7.22: Implementation of `adjacent_difference`

7.5.4. Additional Lemmas

The lemmas shown in Listing 7.23 are also needed for the verification of the algorithm in Section 7.7.

```
/*@
  lemma
    AdjacentDifferenceStep{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, 1+n, b);

  lemma
    AdjacentDifferenceSection{K}:
      \forall value_type *a, *b, integer m, n;
        0 <= m <= n                     ==>
        AdjacentDifference{K}(a, n, b)  ==>
        AdjacentDifference{K}(a, m, b);
*/
```

Listing 7.23: The lemma `AdjacentDifferenceStep`

7.6. Inverting partial_sum with adjacent_difference

In Section 7.5 we had informally argued that partial_sum and adjacent_difference are inverse to each other (see Equations (7.7) and (7.8)). In the current section, we are going to verify the second of these claims with the help of Frama-C, viz. that applying adjacent_difference to the output of partial_sum produces the original array. In Section 7.7, we will verify the converse first claim.

Listing 7.24 expresses the property from Equation (7.8) as lemma, on the ACSL logical level. This lemma is verified by Frama-C with the help of automatic theorem provers.

```
/*@
  lemma
    PartialSumInv{K,L}:
      \forall value_type *a, *b, integer n;
        0 <= n                          ==>
        PartialSum{K}(a, n, b)          ==>
        Unchanged{K,L}(b, n)            ==>
        AdjacentDifference{L}(b, n, a)  ==>
        Unchanged{K,L}(a, n);
*/
```

Listing 7.24: The lemma PartialSumInv

Since the lemma does not deal with arithmetic overflows or potential aliasing of data, we give a corresponding auxiliary C function which takes these issues into account.

Function partial_sum_inv, shown in Listing 7.25, calls first partial_sum and then adjacent_difference. 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 separated) occur. Under these precondition, Frama-C automatically verifies that the final adjacent_difference call just restores the original contents of a used for the initial partial_sum call.

```
/*@
  requires valid:     \valid(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires separated: \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:    AccumulateBounds(a, n+1);

  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 7.25: partial_sum and then adjacent_difference

7.7. Inverting adjacent_difference with partial_sum

In this section, we prove the converse property, viz. that applying adjacent_difference, and thereafter partial_sum, restores the original data array. Listing 7.26 expresses this property as a lemma on the level of ACSL predicates. It had to be proven interactively with Coq, by induction on n.

```
/*@
  lemma
    AdjacentDifferenceInv{K,L}:
      \forall  value_type *a, *b, integer n;
        0 <= n                          ==>
        AdjacentDifference{K}(a, n, b)  ==>
        Unchanged{K,L}(b, n)            ==>
        PartialSum{L}(b, n, a)          ==>
        Unchanged{K,L}(a, n);
*/
```

Listing 7.26: The lemma AdjacentDifferenceInv

As in the case discussed in Section 7.6, we give a corresponding C function in order to account for possible arithmetic overflows and potential aliasing of data. Function adjacent_difference_inv, shown in Listing 7.27, calls first 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 separated) occur.

```
/*@
  requires valid:     \valid(a + (0..n-1));
  requires valid:     \valid(b + (0..n-1));
  requires separated: \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:    AdjacentDifferenceBounds(a, n+1);

  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 7.27: adjacent_difference and then partial_sum

In order to improve the automatic verification rate of the function adjacent_difference_inv we also use lemma UnchangedTransitive from Listing 6.4. Both lemmas itself and (with its additional help) the contract of adjacent_difference_inv are proven by Frama-C without further manual intervention. This finishes the formal proof of our inversity claims from Section 7.5.

8. Heap Algorithms

The heap algorithms of the C++ Standard Library [14, 25.4.6] 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 [18].

The C++ standard[28] 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 a sorted sequence.

Figure 8.1 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.

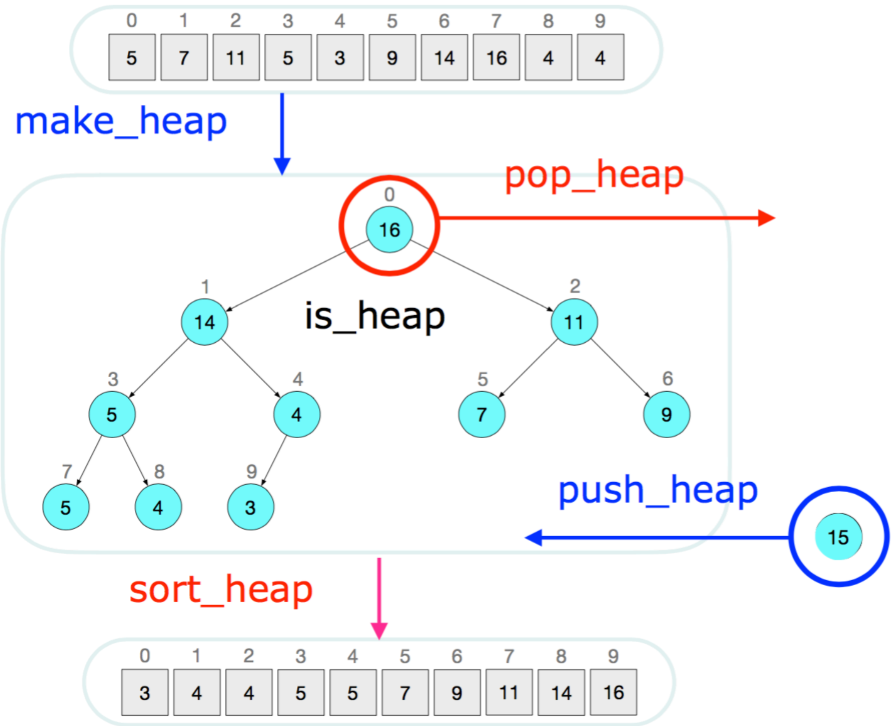

Figure 8.1.: Overview on heap algorithms

[28] See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`

Roughly speaking, the algorithms from Figure 8.1 have the following behavior.

- `is_heap` from Section 8.3 allows to test at run time whether a given array is arranged as a heap
- `push_heap` from Section 8.4 *adds* an element to a given heap in such a way that resulting array is again a heap
- `pop_heap`, which is currently not included in this document, *removes* an element from a given heap in such a way that the resulting array is again a heap
- `make_heap` from Section 8.6 turns a given array into a heap.
- `sort_heap` from Section 8.7 transforms a given heap into a sorted range.

In Section 8.1 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 Section 8.2.

```
#define SIZE_TYPE_MAX  UINT_MAX
```

Listing 8.2: Upper limits of `size_type`

In order to admit maximally large heaps, we had to catch border cases in ACSL as well as in C, cf. e.g. Listing 8.34 and 8.35. To this end, we introduced the constant from Listing 8.2. to refer to the upper bound of `size_type`. We don't need a corresponding constant `SIZE_TYPE_MIN` for the lower bound, since it is trivial.

8.1. 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:[29]

> 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:

$$\text{left child for index } i \qquad \mathrm{child_l} : i \mapsto 2i + 1 \tag{8.1}$$

$$\text{right child for index } i \qquad \mathrm{child_r} : i \mapsto 2i + 2 \tag{8.2}$$

and

$$\text{parent index for index } i \qquad \mathrm{parent} : i \mapsto \frac{i-1}{2} \tag{8.3}$$

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. [9, §2.2.4]).

$$\mathrm{parent}(\mathrm{child_l}(i)) = i \tag{8.4}$$

$$\mathrm{parent}(\mathrm{child_r}(i)) = i \tag{8.5}$$

In order to given an example for the usefulness of heaps we consider the following multiset of integers X.

$$X = \{2, 3, 3, 3, 6, 7, 8, 8, 9, 11, 13, 14\} \tag{8.6}$$

[29] See `http://stdcxx.apache.org/doc/stdlibug/14-7.html`

Figure 8.3 shows how the multiset from Equation (8.6) can, according to the parent-child relations of a heap, be represented as a tree.

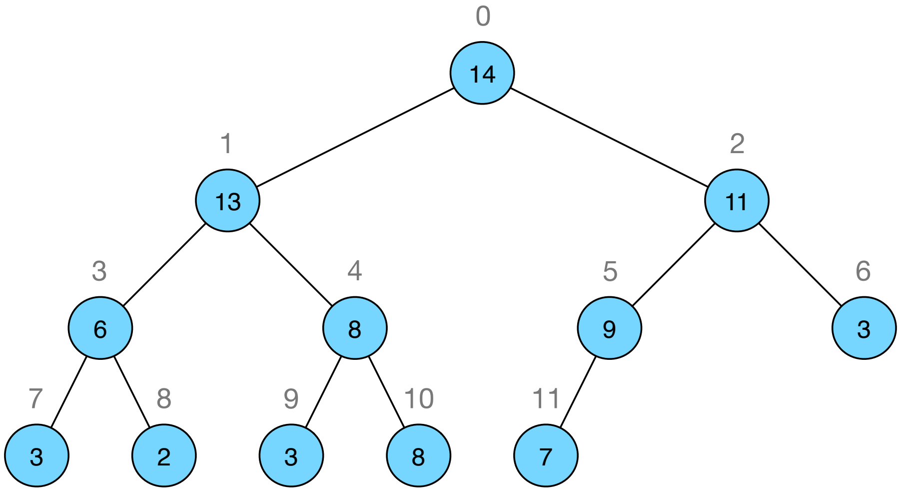

Figure 8.3.: Tree representation of the multiset X

The numbers outside the nodes in Figure 8.3 are the indices at which the respective node value is stored in the underlying array of a heap (cf. Figure 8.4).

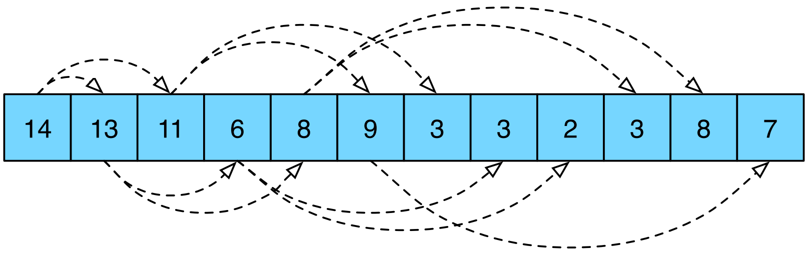

Figure 8.4.: Underlying array of a heap

It is important to understand that there can be various representations of a multiset as a heap. Figure 8.5, for example, arranges the elements of the multiset X as a heap in a different tree.

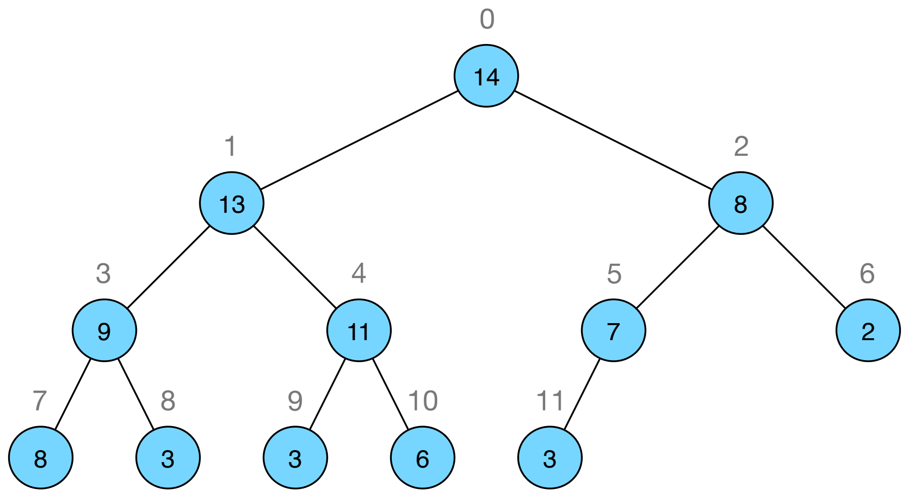

Figure 8.5.: An alternative representation of the multiset X

Figure 8.6 then shows the underlying array that corresponds to the tree in Figure 8.5.

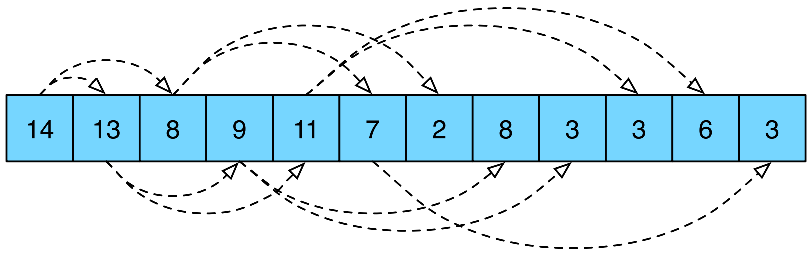

Figure 8.6.: Underlying array of the alternative representation

8.2. ACSL presentation of heap concepts

Listing 8.7 shows three logic functions `HeapLeft`, `HeapRight` and `HeapParent` that correspond to the definitions (8.1), (8.2) and (8.3), respectively.

```
/*@
  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
    HeapParentOfLeft:
      \forall integer p; 0 <= p ==> HeapParent(HeapLeft(p)) == p;

  lemma
    HeapParentOfRight:
      \forall integer p; 0 <= p ==> HeapParent(HeapRight(p)) == p;

  lemma
    HeapParentChild:
      \forall integer c, p;
        0 < c ==> HeapParent(c) == p ==>
        (c == HeapLeft(p) || c == HeapRight(p));

  lemma
    HeapChilds:
      \forall integer a, b;
        0 < a ==> 0 < b ==>
        HeapParent(a) == HeapParent(b) ==>
        (a == b || a+1 == b || a == b+1);

  lemma
    HeapParentBounds:
      \forall integer c; 0 < c ==> 0 <= HeapParent(c) < c;

  lemma
    HeapChildBounds:
      \forall integer p;
        0 <= p ==> p < HeapLeft(p) < HeapRight(p);
*/
```

Listing 8.7: Logic functions for heap definition

Listing 8.7 also contains a number of ACSL lemma that state among other things that

- the `HeapParent` function satisfies the equations (8.4) and (8.5) and
- the function `HeapParent` is the *left inverse* to the `HeapLeft` and `HeapRight` functions.[30]

[30] See Section *Left and right inverses* at `http://en.wikipedia.org/wiki/Inverse_function`

On top of these basic definitions we introduce in Listing 8.8 the predicate `IsHeap`.

```
/*@
  predicate
    IsHeap{L}(value_type* a, integer n) =
      \forall integer i; 0 < i < n  ==> a[i] <= a[HeapParent(i)];
*/
```

Listing 8.8: The predicate `IsHeap`

The root of a heap, that is the element at index 0, is always the largest element of the heap. Lemma `HeapMaximum` in Listing 8.9 expresses this property using the `MaxElement` predicate from Listing 4.7.

```
/*@
  lemma
    HeapMaximum{L} :
      \forall value_type* a, integer n;
        1 <= n ==> IsHeap(a, n) ==> MaxElement(a, n, 0);
*/
```

Listing 8.9: The lemma `HeapMaximum`

We use the following fact about division in C in the proof of lemma `HeapMaximum`.

```
/*@
  lemma
    C_Division_2:
      \forall integer a; 0 <= a  ==>  0 <= a/2 <= a;
*/
```

Listing 8.10: The lemma `C_Division_2`

The following Listing 8.11 contains the C counterpart of a logic heap function in Listing 8.7. Note that here we have to take into account the limitations of C types.

```
/*@
   requires bound:  0 < child;
   assigns          \nothing;
   ensures  parent: \result == HeapParent(child);
 */
static inline size_type heap_parent(size_type child)
{
  return (child - 1u) / 2u;
}
```

Listing 8.11: An auxiliary heap function

8.3. The `is_heap` algorithm

The `is_heap` algorithm of the C++ Standard Library [14, §25.4.6.5] 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 8.4.

8.3.1. Formal specification of `is_heap`

The ACSL specification of `is_heap` is shown in Listing 8.12. The function returns **true** if and only if its arguments satisfy the predicate `IsHeap` introduced in Section 8.2.

```
/*@
   requires valid: \valid_read(a +(0..n-1));

   assigns \nothing;

   ensures heap:  \result <==> IsHeap(a, n);
*/
bool is_heap(const value_type* a, size_type n);
```

Listing 8.12: Function Contract of `is_heap`

Before we discuss the implementation of `is_heap` we want to point out that (downward) sorted arrays are heaps. In Listing 8.13 we have formalized this claim which is easily verified by Frama-C/WP.

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


  lemma
    SortedDownIsHeap{L}:
      \forall value_type *a, integer n;
        SortedDown(a, n) ==> IsHeap(a, n);
*/
```

Listing 8.13: The lemma SortedDownIsHeap

8.3.2. Implementation of is_heap

Listing 8.14 shows one way to implement the function is_heap. 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.

```
bool is_heap(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:   IsHeap(a, child);

    loop assigns child, parent;
    loop variant n - child;
  */
  for (size_type child = 1u; child < n; ++child) {
    if (a[parent] < a[child]) {
      return false;
    }

    if ((child % 2u) == 0u) {
      ++parent;
    }
  }

  return true;
}
```

Listing 8.14: Implementation of is_heap

8.4. The push_heap algorithm

Whereas in the C++ Standard Library [14, §25.4.6.1] 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* an element on a heap.

8.4.1. Formal Specification of push_heap

Listing 8.15 shows our ACSL 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 the user of the algorithm has in mind.

```
/*@
   requires nonempty: 0 < n;
   requires valid:    \valid(a + (0..n-1));
   requires heap:     IsHeap(a, n-1);

   assigns a[0..n-1];

   ensures heap:      IsHeap(a, n);
   ensures reorder:   MultisetUnchanged{Old,Here}(a, n);
*/
void push_heap(value_type* a, size_type n);
```

Listing 8.15: Formal specification of push_heap

Pushing an element on a heap usually *rearranges* several elements of the array (cf. Figures 8.16 and 8.17). We therefore must be able express that push_heap only *reorders* the elements of the array. We re-use the predicate MultisetUnchanged, defined in Listing 6.5, to formally describe this property.

8.4.2. 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 8.16 shows the heap from Figure 8.3 together with the additional element `12` that is to be 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.

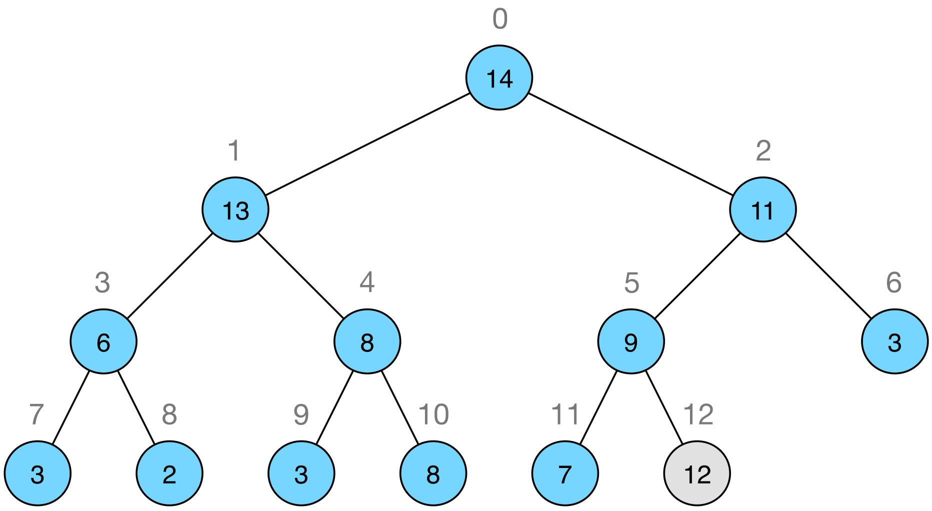

Figure 8.16.: Heap before the call of `push_heap`

Figure 8.17 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 changed their contents during the function call. The pushed element `12` is now at its correct position in the heap.

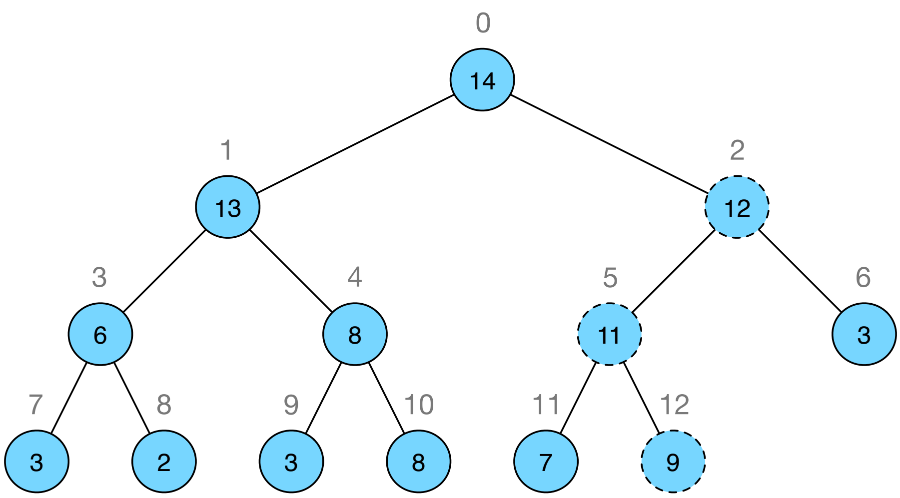

Figure 8.17.: Heap after the call of `push_heap`

8.4.2.1. Challenges during the verification

In order to properly describe different stages of `push_heap` and to accommodate the sheer size of our implementation we split the source code into three separate parts, to which we refer as

- *prologue* (see Section 8.4.2.2)
- *main act* (see Section 8.4.2.3)
- *epilogue* (see Section 8.4.2.4)

We will illustrate the changes to the array after each stage by figures of the array in tree form, based on the `push_heap` example from Figure 8.16.

Verifying `push_heap` is a non-trivial undertaking, and we will proceed, roughly speaking, as follows:

> We can establish the `heap` property of Listing 8.15 already in the prologue. However, the `reorder` property only holds at the function boundaries but 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.

8.4.2.2. Prologue

Our prologue initializes some important variables, checks whether the initial heap is nonempty, *and* also tries to move the new element upwards within the heap. In other implementations, the latter step is usually performed as part of `push_heap`'s main loop. In order to better understand our implementation decision we can look at Figure 8.18 which shows exemplarily its effects.

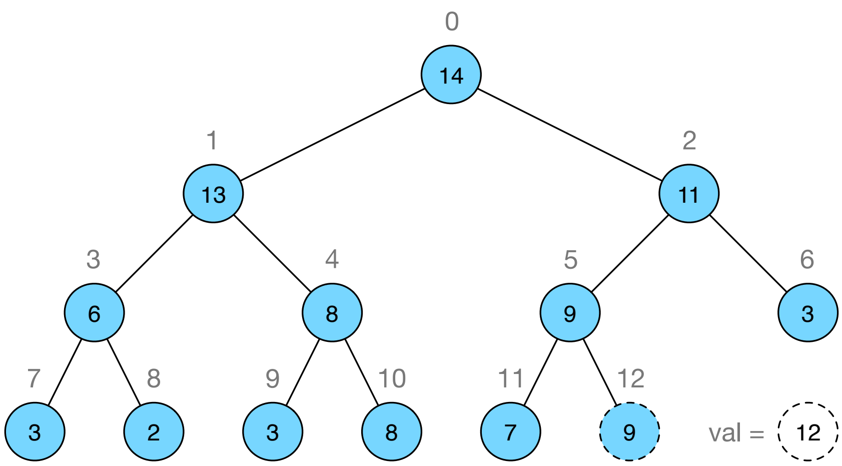

Figure 8.18.: Heap after the prologue of `push_heap`

If we compare this tree to the tree in Figure 8.16 we notice that the element in the node with the dashed outlining changed its value. The number of occurrences of `9` increased by one during the prologue, while the number of occurrences of `12` decreased by one. The number of occurrences for all other elements is maintained. We store the element `12` in the variable `val` so we can write it back into the array later. The increased number of occurrences of `9` and the decreased number of elements `12` means that at this stage the postcondition `reorder` is violated. On the other hand, the modified array is now a heap. We express this by coloring all elements blue (cf. Figure 8.16).

More generally speaking, the following properties hold after the prologue:

1. The modified array is now a heap.
2. The "parent value" `a[parent]` now occurs one time more often.
3. The value `a[n-1]`, on the other hand, now occurs one time fewer.
4. No other value changed its number of occurrences.

Listing 8.19 shows the implementation of the prologue. It starts with the listing of an auxiliary function `heap_parent` that computes the parent index of its argument. The prologue also deals with the trivial cases that

- the array contains only one element or
- if `a[n-1]` is less or equal than its parent element

At the end of the prologue we have added four assertions that formally express the properties we have just enumerated above. As we will see (in Listing 8.24), these properties will occur as loop invariants in the the main act. Adding these assertions makes the purpose of the prologue more explicit and thus supports the long-term maintenance of the annotated code. The auxiliary predicates that are used in the assertions are:

- predicate `MultisetAdd` (Listing 8.20)
- predicate `MultisetMinus` (Listing 8.21)
- and an overloaded version of predicate `MultisetRetainRest` (Listing 8.22)

```
void push_heap(value_type* a, size_type n)
{
  // start of prologue
  if (1u < n) { // otherwise nothings needs to be done

    const value_type v      = a[n - 1];
    size_type  hole = heap_parent(n - 1);

    if (a[hole] < v) {

      a[n - 1] = a[hole];
      //@ assert heap:   IsHeap(a, n);
      //@ assert add:    MultisetAdd{Pre,Here}(a, n, a[hole]);
      //@ assert minus:  MultisetMinus{Pre,Here}(a, n, v);
      //@ assert retain: MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);

      // end of prologue
```

Listing 8.19: Prologue of `push_heap` implementation

These auxiliary predicates are discussed in the following subsections.

The predicates `MultisetAdd` and `MultisetMinus`

The predicate `MultisetAdd` in Listing 8.20 expresses that the number of occurrences of a specific element in an array has increased by one between between two program points `K` and `L`.

```
/*@
  predicate
    MultisetAdd{K,L}(value_type* a, integer n, value_type val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) + 1;
*/
```

Listing 8.20: The predicate `MultisetAdd`

The predicate `MultisetMinus` in Listing 8.21, 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`.

```
/*@
  predicate
    MultisetMinus{K,L}(value_type* a, integer n, value_type val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) - 1;
*/
```

Listing 8.21: The predicate `MultisetMinus`

Note that we could have defined `MultisetMinus` also by calling `MultisetAdd` with the labels reversed.

```
  predicate
    MultisetMinus{K,L}(value_type* a, integer n, value_type val) =
      MultisetAdd{L,K}(a, n, val);
```

It is a often only a matter of taste how to decide which of several ways to define a predicate is more appropriate. However, one also has to take into account which definition can be handled more easily by Frama-C/WP and its associated theorem provers.

Overloading predicate `MultisetRetainRest`

The predicate `MultisetRetainRest` (Listing 8.22) overloads the predicate from Listing 6.38. The new version holds if the number of occurrences for all elements, except the two given ones, remains unchanged between two program points.

```
/*@
  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);
*/
```

Listing 8.22: An overloaded version of predicate `MultisetRetainRest`

The definition of this new version of the `MultisetRetainRest` predicate uses the simpler predicate `MultisetRetain` shown in Listing 8.23. This predicate holds if the number of occurrences of a given value in an array does not change between two program points. We will later also directly employ `MultisetRetain` to succinctly formulate additional assertions in `push_heap`.

```
/*@
  predicate
    MultisetRetain{K,L}(value_type* a, integer n, value_type v) =
      Count{K}(a, n, v) == Count{L}(a, n, v);
*/
```

Listing 8.23: The predicate `MultisetRetain`

8.4.2.3. Main act

The goal of the main act is to locate the array index to which the new element can be assigned. Listing 8.24 shows its implementation.

```
// start of main act
if (0u < hole) {
  size_type parent = heap_parent(hole);

  /*@
    loop invariant bound:  0 <= hole < n-1;
    loop invariant heap:   IsHeap(a, n);
    loop invariant heap:   parent == HeapParent(hole);
    loop invariant less:   a[hole] < v;
    loop invariant add:    MultisetAdd{Pre,Here}(a, n, a[hole]);
    loop invariant minus:  MultisetMinus{Pre,Here}(a, n, v);
    loop invariant retain: MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);
    loop assigns           hole, parent, a[0..n-1];
    loop variant           hole;
  */
  while (0u < hole && a[parent] < v) {
    //@ ghost Loop: // LoopEntry not yet supported!
    //@ ghost const value_type old_a = a[hole];
    //@ assert reorder:  old_a == \at(a[hole],Loop);

    if (a[hole] < a[parent]) {
      a[hole] = a[parent];
      //@ assert less:    old_a   < v;
      //@ assert less:    a[hole] < v;
      //@ assert retain:  MultisetUnchanged{Loop,Here}(a, 0, hole);
      //@ assert retain:  MultisetUnchanged{Loop,Here}(a, hole + 1, n);
      //@ assert minus:   MultisetMinus{Loop,Here}(a, n, old_a);
      //@ assert add:     MultisetAdd{Loop,Here}(a, n, a[hole]);
      //@ assert retain:  MultisetRetain{Loop,Here}(a, n, v);
      //@ assert retain:  MultisetRetain{Pre,Here}(a, n, old_a);
      //@ assert retain:  MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);
    }

    hole = parent;
    if (0u < hole) {
      parent = heap_parent(hole);
    }
  }
}
// end of main act
```

Listing 8.24: Main act of `push_heap` implementation

The loop invariant `heap` expresses that the predicate `IsHeap` is true for the array throughout the main act. Instead of an invariant `reorder` that reflects the postcondition with the same name, we now consider the invariants `add`, `minus`, and `retain`.

It is important to understand the use of the variable `hole` in these loop invariants. Before each loop iteration, `hole` stores the index of the node whose value was assigned to one of its children in the previous iteration or in the prologue (for the first loop run). Therefore, the value `a[hole]` appears in the loop invariants `add` and `retain`.

Verifying the various loop invariants and assertions has been far from being straightforward, and required additional assertions and the *ghost* label `Loop`. The following remarks highlight some of the issues.

1. The heap property implies that a[hole] <= a[parent] always holds. Thus, the assignment a[hole] = a[parent] might be redundant. We have not check whether guarding this assignment with the condition a[hole] < a[parent] is more efficient. Important for us is the following: the guard allows us to put additional assertions where they really matter and where they can more easily be verified.

2. In order to guide the automatic provers, we have also provided the lemmas MultisetAddDistinct (Listing 8.25), MultisetMinusDistinct (Listing 8.26), and MultisetPushHeapRetain (Listing 8.27). These lemmas formalize conditions under which the respective predicates MultisetAdd, MultisetMinus, and MultisetRetain apply.

```
/*@
  lemma
    MultisetAddDistinct{K,L}:
      \forall value_type *a, v, integer i, n;
        0 <= i < n                          ==>
        \at(a[i],K) != v                    ==>
        \at(a[i],L) == v                    ==>
        MultisetUnchanged{K,L}(a, 0, i)     ==>
        MultisetUnchanged{K,L}(a, i+1, n)   ==>
        MultisetAdd{K,L}(a, n, v);
*/
```

Listing 8.25: The lemma MultisetAddDistinct

```
/*@
  lemma
    MultisetMinusDistinct{K,L}:
      \forall value_type *a, v, integer i, n;
        0 <= i < n                          ==>
        \at(a[i],K) == v                    ==>
        \at(a[i],L) != v                    ==>
        MultisetUnchanged{K,L}(a, 0, i)     ==>
        MultisetUnchanged{K,L}(a, i+1, n)   ==>
        MultisetMinus{K,L}(a, n, v);
*/
```

Listing 8.26: The lemma MultisetMinusDistinct

```
/*@
 lemma
   MultisetPushHeapRetain{K,L,M}:
     \forall value_type *a, ap, ah, v, integer h, p, n;
       0 <= p < h < n-1                        ==>
       ah < ap < v                             ==>
       \at(a[h],L)  ==  ah                     ==>
       \at(a[p],L)  ==  ap                     ==>
       \at(a[h],M)  ==  ap                     ==>
       MultisetMinus{K,L}(a, n, v)             ==>
       MultisetAdd{K,L}(a, n, ah)              ==>
       MultisetRetainRest{K,L}(a, n, v, ah)    ==>
       MultisetUnchanged{L,M}(a, 0, h)         ==>
       MultisetUnchanged{L,M}(a, h+1, n)       ==>
       MultisetRetainRest{K,M}(a, n, v, ap);
*/
```

Listing 8.27: The lemma `MultisetPushHeapRetain`

Figure 8.28 shows the array after the main act. The contents of the dashed nodes have been overwritten with the values of their parents until `hole` reached a node to which `val` can be assigned, whilst maintaining the `heap` property.

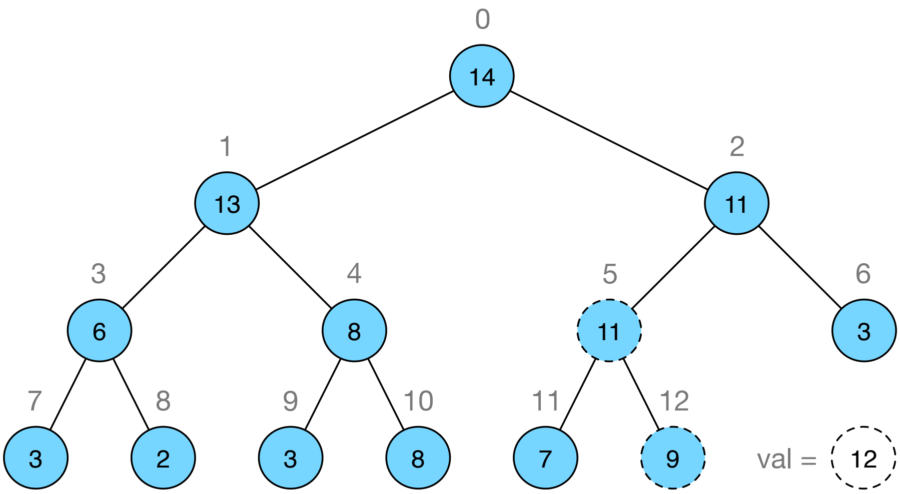

Figure 8.28.: Heap after the main act of `push_heap`

In our example, the loop performs just one assignment, viz. `a[5]=a[2]`, and then stops with `hole` being `2`. At this point, the new element `12` can be assigned to the node with the index `2` and the `heap` property stays intact. This assignment takes place in the epilogue.

8.4.2.4. Epilogue

The last part of the implementation is the epilogue, shown in Listing 8.29. It consists of exactly one assignment which re-establishes the `reorder` property while maintaining the `heap` property already established.

```
      // start of epilogue

      //@ ghost Epi:
      a[hole] = v;
      //@ assert value:      \at(a[hole],Epi)  != v;
      //@ assert value:      \at(a[hole],Here) == v;
      //@ assert unchanged:  MultisetUnchanged{Epi,Here}(a, 0, hole);
      //@ assert unchanged:  MultisetUnchanged{Epi,Here}(a, hole+1, n);
      //@ assert add:        MultisetAdd{Epi,Here}(a, n, v);
      //@ assert minus:      MultisetMinus{Epi,Here}(a, n, \at(a[hole],Epi));
      //@ assert reorder:    MultisetUnchanged{Pre,Here}(a, n);
    }
  }
  // end of epilogue
}
```

Listing 8.29: Epilogue of `push_heap` implementation

The ghost label `Epi`, together with a couple of assertions and the lemma `MultisetPushHeapClosure` are necessary to help the automatic theorem provers to prove the final assertion `reorder`.

```
/*@
 lemma
   MultisetPushHeapClosure{K,L,M}:
     \forall value_type *a, u, v, integer i, n;
       0 <= i < n-1                         ==>
       u != v                               ==>
       \at(a[i],M)  == v                    ==>
       MultisetAdd{K,L}(a, n, u)            ==>
       MultisetMinus{K,L}(a, n, v)          ==>
       MultisetRetainRest{K,L}(a, n, v, u)  ==>
       MultisetUnchanged{L,M}(a, 0, i)      ==>
       MultisetUnchanged{L,M}(a, i+1, n)    ==>
       MultisetAdd{L,M}(a, n, v)            ==>
       MultisetMinus{L,M}(a, n, u)          ==>
       MultisetUnchanged{K,M}(a, n);
*/
```

Listing 8.30: The lemma `MultisetPushHeapClosure`

Concerning the `reorder` property, the main act finished with an increased count of nodes with the value `11` and a decreased count of nodes with the value `12` (cf. Figure 8.28). The heap in Figure 8.31, on the other hand, shows the tree after the epilogue has assigned the value `12` to the node with the index `2`, which contained the value `11`. Hence the `reorder` property is re-established and the function can return.

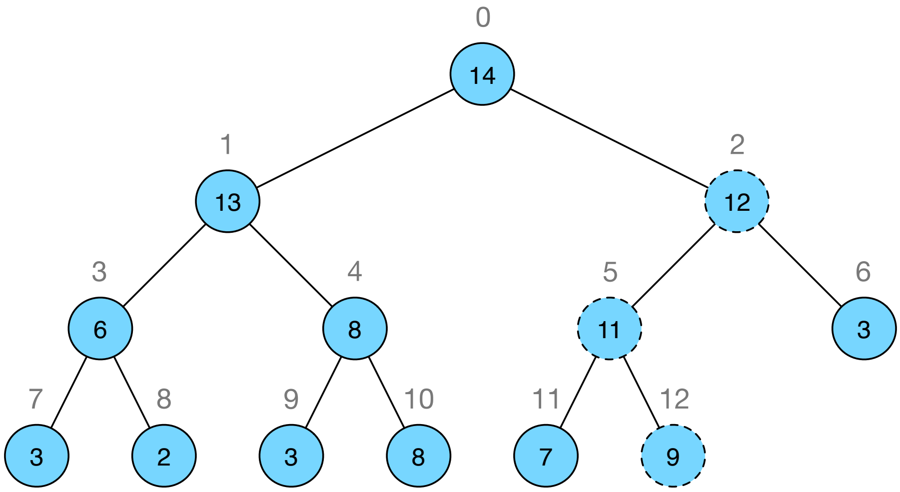

Figure 8.31.: Heap after the epilogue of `push_heap`

Moreover, since the `heap` property has been inferred, all nodes are now colored blue.

8.5. The pop_heap algorithm

Whereas in the C++ Standard Library [14, §25.4.6.2] 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[31] 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.

8.5.1. Formal Specification of pop_heap

The ACSL function contract of pop_heap is given in Listing 8.32.

```
/*@
   requires bounds: 0 < n;
   requires valid:  \valid(a + (0..n-1));
   requires heap:   IsHeap(a, n);

   assigns a[0..n-1];

   ensures heap:    IsHeap(a, n-1);
   ensures result:  a[n-1] == \old(a[0]);
   ensures max:     MaxElement(a, n, n-1);
   ensures reorder: MultisetUnchanged{Old, Here}(a, n);
*/
void pop_heap(value_type* a, size_type n);
```

Listing 8.32: Formal specification of pop_heap

[31] See Lemma HeapMaximum in Listing 8.9.

8.5.2. Implementation of pop_heap

Listing 8.33 shows our implementation of pop_heap together with ACSL annotations. Note that in this version the postcondition reorder of pop_heap, which states that the algorithm only *rearranges* the elements of the array, is not verified by Frama-C/WP. Verifying this postcondition would require more elaborate loop invariants which we will supply in a later version of this document.

```
void pop_heap(value_type* a, size_type n)
{
  if (1u < n) { // otherwise nothings needs to be done

    const value_type v = a[0u];
    //@ assert max:  MaxElement(a, n, 0);

    if (a[n - 1u] < v) { // otherwise nothings needs to be done
      //@ assert bounds: 2 <= n;
      size_type hole = 0u;
      size_type child = maximum_heap_child(a, n, hole);
      //@ assert heap: child < n - 1  ==>  hole == HeapParent(child);

      /*@
          loop invariant bounds: 0 <= hole < n-1;
          loop invariant bounds: hole < child;
          loop invariant heap:   IsHeap(a, n);
          loop invariant heap:   a[n-1] < a[HeapParent(hole)];
          loop invariant heap:   child < n - 1  ==>  hole == HeapParent(child);
          loop invariant child:  HeapMaximumChild(a, n, hole, child);
          loop invariant max:    UpperBound(a, 0, n, v);
          loop assigns           hole, child, a[0..n-2];
          loop variant           n - hole;
       */
      while (child < n - 1u && a[n - 1u] < a[child]) {
        a[hole] = a[child];
        hole    = child;
        //@ assert heap: IsHeap(a, n);
        child = maximum_heap_child(a, n, hole);
      }

      //@ assert child: child < n-1 ==> a[n-1] >= a[child];
      //@ assert child: HeapMaximumChild(a, n, hole, child);
      //@ assert heap: IsHeap(a, n);
      //@ assert heap: a[n-1] < a[HeapParent(hole)];
      a[hole]    = a[n - 1u];
      //@ assert heap: IsHeap(a, n-1);
      a[n - 1u] = v;
      //@ assert heap: IsHeap(a, n-1);
    }
  }
}
```

Listing 8.33: Implementation of pop_heap

Our implementation relies on the auxiliary function `maximum_heap_child` which determines for a given heap element "the" child element is not less than another child element. We have formalized the notion of a *maximum child of an element of a heap* in the ACSL predicate `HeapMaximumChild` shown in Listing 8.34.

```
/*@
  predicate
    HeapMaximumChild{L}(value_type* a, integer n,
                        integer p,     integer c) =
      0 <= p < n-1                                        &&
      (p < (SIZE_TYPE_MAX-1)/2   ==>  p == HeapParent(c))      &&
      (HeapLeft(p)  < n-1        ==>  a[HeapLeft(p)]  <= a[c]) &&
      (HeapRight(p) < n-1        ==>  a[HeapRight(p)] <= a[c]);
*/
```

Listing 8.34: The predicate `HeapMaximumChild`

Listing 8.35 shows the contract and the implementation of the function `maximum_heap_child`. This function returns a child index `c` of `parent` with the property that for a potential other child `d` of `parent` with `d < n - 1` the condition `a[d] <= a[c]` holds. Note that it explicitly handles the case that the child index computation would overflow; it returns `n` then.

```
/*@
   requires bound: 2 <= n;
   requires bound: 0 <= parent < n - 1;
   requires valid: \valid(a + (0..n-1));
   requires heap:  IsHeap(a, n);

   assigns         \nothing;

   ensures heap:   IsHeap(a, n);
   ensures max:    HeapMaximumChild(a, n, parent, \result);
   ensures less:   parent < \result;
   ensures less:   \result < n - 1  ==>  parent == HeapParent(\result);
*/
static inline size_type
maximum_heap_child(const value_type* a, size_type n, size_type parent)
{
  if (parent < (SIZE_TYPE_MAX - 1u) / 2u) {
    const size_type right = 2u * parent + 2u;
    const size_type left  = right - 1u;

    if (right < n - 1u) {
      // case of two children: select child with maximum value
      return a[left] >= a[right] ? left : right;
    }
    else {
      // at most one child that comes before n-1 can exist
      return left;
    }
  }
  else {
    return n;
  }
}
```

Listing 8.35: The auxiliary function `maximum_heap_child`

8.6. The make_heap algorithm

Whereas in the C++ Standard Library [14, §25.4.6.2] 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 8.36. 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 8.4.

8	8	7	3	14	11	3	6	2	3	13	9

Figure 8.36.: Array before the call of make_heap

8.6.1. Formal Specification of make_heap

Listing 8.37 shows the ACSL specification of make_heap.

```
/*@
   requires valid:  \valid(a + (0..n-1));

   assigns a[0..n-1];

   ensures heap:    IsHeap(a, n);
   ensures reorder: MultisetUnchanged{Old,Here}(a, n);
*/
void make_heap(value_type* a, size_type n);
```

Listing 8.37: The 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 MultisetUnchanged (see Listing 6.5) to ensure make_heap only rearranges the array elements.

8.6.2. Implementation of `make_heap`

The implementation of `make_heap`, shown in Listing 8.38, 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` 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:       IsHeap(a, i);
       loop invariant reorder:    MultisetUnchanged{Pre,Here}(a, i+1);
       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 = 1; i < n; ++i) {
      push_heap(a, i + 1u);
    }
  }

  //@ assert  heap: IsHeap(a, n);
}
```

Listing 8.38: The Implementation of `make_heap`

Since the loop statement consists just of a call to `push_heap` we obtain the both loop invariants `heap` and `reorder` by simply lifting them from the contract of `push_heap` (see Section 8.4.1).

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`.

8.7. The `sort_heap` algorithm

Whereas in the C++ Standard Library [14, §25.4.6.4] `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]` into an array that is sorted in increasing order. Thus, applying `sort_heap` to the heap in Figure 8.4 produces the sorted array in Figure 8.39.

2	3	3	3	6	7	8	8	9	11	13	14

Figure 8.39.: Array after the call of `sort_heap`

8.7.1. Formal Specification of `sort_heap`

Listing 8.40 shows our ACSL specification of `sort_heap`. The formal specification of `sort_heap` must ensure that the resulting array is sorted. Furthermore the multiset contained by the array must be the same as in the pre-state of the function. The postconditions `sorted` and `reorder` express these properties, respectively. The specification effort is relatively simple because we can reuse the previously defined predicates `MultisetUnchanged` (Listing 6.5) and `Sorted` (Listing 5.1).

```
/*@
   requires valid:  \valid(a + (0..n-1));
   requires heap:   IsHeap(a, n);

   assigns a[0..n-1];

   ensures sorted:   Sorted(a, n);
   ensures reorder:  MultisetUnchanged{Old, Here}(a, n);
*/
void sort_heap(value_type* a, size_type n);
```

Listing 8.40: Formal specification of `sort_heap`

8.7.2. Implementation of sort_heap

The implementation of sort_heap (Listing 8.41) is relatively simple because it relies on the pop_heap algorithm 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 sorted part of the array.

```
void sort_heap(value_type* a, size_type n)
{
  /*@
     loop invariant bound:    0 <= i <= n;
     loop invariant heap:     IsHeap(a, i);
     loop invariant sorted:   Sorted(a, i, n);
     loop invariant lower:    LowerBound(a, i, n, a[0]);
     loop invariant reorder:  MultisetUnchanged{Pre,Here}(a, 0, n);
     loop assigns i, a[0..n-1];
     loop variant i;
  */
  for (size_type i = n; i > 1; --i) {
    /*@
        requires heap:     IsHeap(a, i);
        assigns  a[0..i-1];
        ensures  heap:     IsHeap(a, i-1);
        ensures  max:      a[i-1] == \old(a[0]);
        ensures  max:      MaxElement(a, i, i-1);
        ensures  reorder:  MultisetUnchanged{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]);
  }
}
```

Listing 8.41: The Implementation of sort_heap

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 sorted.

Supporting lemmas

In order to facilitate the automatic verification of the property `sorted` we rely among others on the properties `lower` and `max` and the lemma `SortedUpperBound` from Listing 8.42.

```
/*@
  lemma
    SortedUpperBound{L}:
      \forall value_type *a, integer n;
        UpperBound(a, n, a[n])  ==>
        Sorted(a, n)            ==>
        Sorted(a, n+1);
*/
```

Listing 8.42: The lemma `SortedUpperBound`

To verify the property `reorder` we formulate in Listing 8.43 several lemmas that express that the properties

- `MultisetUnchanged{K,L}(a, 0, i)` and
- `Unchanged{Old,Here}(a, i, n)`

imply the desired loop invariant `MultisetUnchanged{K,L}(a, 0, n)`.

```
/*@
  lemma
    UnchangedImpliesMultisetUnchanged{L1,L2}:
      \forall value_type *a, integer k, n;
        Unchanged{L1,L2}(a, k, n) ==>
        MultisetUnchanged{L1,L2}(a, k, n);


  lemma
    MultisetUnchangedUnion{L1,L2}:
      \forall value_type *a, integer i, k, n;
        0 <= i <= k <= n                      ==>
        MultisetUnchanged{L1,L2}(a, i, k)     ==>
        MultisetUnchanged{L1,L2}(a, k, n)     ==>
        MultisetUnchanged{L1,L2}(a, i, n);


  lemma
    MultisetUnchangedTransitive{L1,L2,L3}:
      \forall value_type *a, integer n;
        MultisetUnchanged{L1,L2}(a, n)   ==>
        MultisetUnchanged{L2,L3}(a, n)   ==>
        MultisetUnchanged{L1,L3}(a, n);
*/
```

Listing 8.43: Some lemmas for `MultisetUnchanged`

9. Sorting Algorithms

In this chapter, we present algorithms of the C++ Standard Library [14, §25.4.1] that are related to the task of sorting a linear array.

- Section 9.1 shows an algorithm to check if a given array is already sorted in ascending order.
- The algorithm in Section 9.2 partitions a given array, and sorts only the resulting lower part.

In future releases we plan to handle the algorithms

- `partial_sort_copy` [14, §25.4.1.4]
- `sort` [14, §25.4.1.1] for the treatment of some classic sorting algorithms we refer the reader to Chapter 10
- `stable_sort` [14, §25.4.1.2]
- `merge` [14, §25.4.4]
- `inplace_merge` [14, §25.4.4]

9.1. The `is_sorted` algorithm

Our version of the `is_sorted` algorithm compared to the C++ Standard Library [14, §25.4.1.5] has the signature

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

It returns `true` if the given array is in ascending order, and `false` else.

9.1.1. Formal Specification of `is_sorted`

Listing 9.1 shows the ACSL specification of `is_sorted`. In the contract, we use the predicate `Sorted`, (see Listing 5.1) 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, see below.

```
/*@
  requires valid: \valid_read(a + (0..n-1));

  assigns  \nothing;

  ensures  \result <==> Sorted(a, n);
*/
bool is_sorted(const value_type* a, size_type n);
```

Listing 9.1: The Specification of `is_sorted`

9.1.2. Implementation of `is_sorted`

The implementation of `is_sorted` is shown in Listing 9.3. As usual, it 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 reason, we use the predicate `WeaklySorted`, shown in Listing 9.2, in the loop invariant of the implementation.

Users inexperienced in formal verification often have a blind spot at the difference between `Sorted` and `WeaklySorted`. Both versions are logically equivalent, and proving `Sorted` ⇒ `WeaklySorted` 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 at all.

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

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

Listing 9.2: The predicate `WeaklySorted`

Since our implementation uses `WeaklySorted` in its loop invariant, and follows the same principle in its code, its verification is straight-forward — except for the final reasoning that `WeaklySorted(a,n)` implies `Sorted(a,n)`. We have an own lemma for that step, shown in Listing 9.4, which needs to be proven manually with Coq. The converse lemma (Listing 9.5) is proven automatically, but isn't actually needed to verify our `is_sorted` implementation.

Alternatively, we could have dragged the predicate `Sorted` along the loop, which happens to cause no particular problems in this case.

```
bool is_sorted(const value_type* a, size_type n)
{
  if (0u < n) {
    /*@
        loop invariant sorted: WeaklySorted(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 9.3: The implementation of `is_sorted`

```
/*@
  lemma
    WeaklySortedImpliesSorted{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n             ==>
        WeaklySorted(a, m, n) ==>
        Sorted(a, m, n);
*/
```

Listing 9.4: The lemma `WeaklySortedImpliesSorted`

```
/*@
  lemma
    SortedImpliesWeaklySorted{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n      ==>
        Sorted(a, m, n) ==>
        WeaklySorted(a, m, n);
*/
```

Listing 9.5: The lemma `SortedImpliesWeaklySorted`

9.2. The `partial_sort` algorithm

Our version of the `partial_sort` algorithm compared to the C++ Standard Library [14, §25.4.1.3] 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 9.6 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.

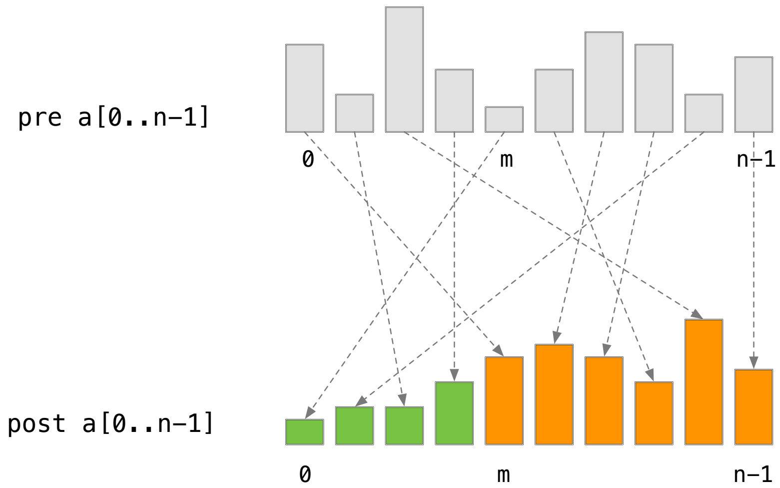

Figure 9.6.: Effects of `partial_sort`

.

9.2.1. Formal specification of `partial_sort`

We start this section by introducing in Listing 9.7 the new predicate `Partition` which formalizes the partitioning property.

```
/*@
  predicate
    Partition{L}(value_type* a, integer m, integer n) =
      0 <= m <= n ==>
      \forall integer i, k; 0 <= i < m <= k < n ==> a[i] <= a[k];
*/
```

Listing 9.7: The predicate `Partition`

The formal specification of the partial_sort function is shown in Listing 9.8. It uses the just introduced predicate Partition and reuses the previously defined predicates Sorted (shown in Listing 5.1) and MultisetUnchanged (Listing 6.5).

```
/*@
  requires valid: \valid(a + (0..n-1));
  requires split: 0 <= m <= n;

  assigns  a[0..n-1];

  ensures sorted:    Sorted(a, m);
  ensures partition: Partition(a, m, n);
  ensures reorder:   MultisetUnchanged{Old,Here}(a, n);
*/
void partial_sort(value_type* a, size_type m, size_type n);
```

Listing 9.8: The Specification of partial_sort

9.2.2. Implementation of partial_sort

Our implementation is shown in Listing 9.10 and 9.11. It initially calls make_heap (Section 8.6) 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 (8.5) and push_heap (8.4) appropriately. When the scan is done, the smallest elements are collected in the left part. We finally convert it from a heap into an ascending range, by sort_heap (8.7).

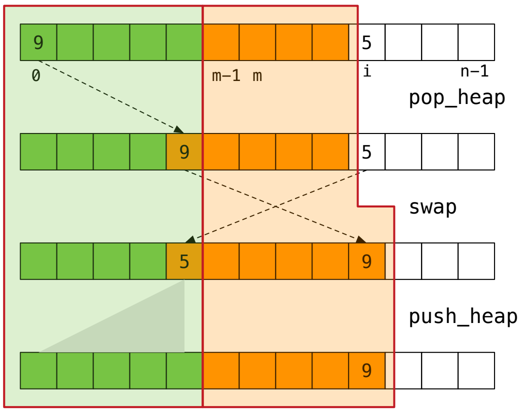

Figure 9.9.: An iteration of partial_sort

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).

```
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:      IsHeap(a, m);
      loop invariant upper:     UpperBound(a, 0, m, a[0]);
      loop invariant lower:     LowerBound(a, m, i, a[0]);
      loop invariant reorder:   MultisetUnchanged{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[0]) {

        /*@
          assigns             a[0..m-1];
          ensures heap:       IsHeap(a, m-1);
          ensures max:        a[m-1] == \old(a[0]);
          ensures max:        MaxElement(a, m, m-1);
          ensures reorder:    MultisetUnchanged{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:    MultisetUnchanged{Pre,Here}(a, i);
```

Listing 9.10: The Implementation of `partial_sort` (1)

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 9.9. The elements considered so far are shown colored similar to Figure 9.6; 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.

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.

```
        /*@
          assigns              a[m-1], a[i];
          ensures swapped:   SwappedInside{Old,Here}(a, m-1, i, n);
        */
        swap(a + m - 1u, a + i);
        //@ assert lower:       a[m-1] < a[i];
        /*@ assert lower:       \forall integer k; 0 <= k < m ==>
                                  LowerBound(a,m,i+1,a[k]);       */
        //@ assert upper:       UpperBound(a, 0, m-1, a[0]);
        //@ assert reorder:     MultisetUnchanged{Pre,Here}(a, i+1);
        //@ assert unchanged:   Unchanged{Pre,Here}(a, i+1, n);

        /*@
          assigns              a[0..m-1];
          ensures heap:        IsHeap(a, m);
          ensures reorder:     MultisetUnchanged{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 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 sorted:    Sorted(a, m);
      ensures reorder:   MultisetUnchanged{Old,Here}(a, m);
      ensures reorder:   MultisetUnchanged{Old,Here}(a, m, n);
    */
    sort_heap(a, m);
    //@ assert partition: Partition(a, m, n);
    //@ assert reorder:   MultisetUnchanged{Pre,Here}(a, n);
  }
}
```

Listing 9.11: The Implementation of `partial_sort` (2)

After calling `push_heap`, which just performs some more reorderings of the left part,[32] this property is preserved. 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 (Listing 9.10). In contrast, we use `assert` clauses to indicate intermediate reasoning to obtain subsequently needed properties.

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 in Listing 9.8, 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.

[32] We can't and we needn't tell which position q is moved to; the former is indicated in Figure 9.6 by the vague grey triangle.

Lemmas used in partition proofs

The following lemmas are use in proofs of properties and annotations related to the loop invariants `upper` and `lower`. Lemma `ReorderPreservesUpperBound` (Listing 9.12) informally says that a lower bound `v` of a range `a[0..n-1]` keeps its property even after the range is reordered.

```
/*@
  lemma
    ReorderPreservesUpperBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                        ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        UpperBound{K}(a, n, v)        ==>
        UpperBound{L}(a, n, v);
*/
```

Listing 9.12: The lemma `ReorderPreservesUpperBound`

Dually, lemma `ReorderPreservesLowerBound` (Listing 9.12) says that reordering a range doesn't affect any of its upper bounds.

```
/*@
  lemma
    ReorderPreservesLowerBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                        ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        LowerBound{K}(a, n, v)        ==>
        LowerBound{L}(a, n, v);
*/
```

Listing 9.13: The lemma `ReorderPreservesLowerBound`

Lemma `PartialReorderPreservesLowerBounds` (Listing 9.14) describes a more particular situation: if each element in `a[0..m-1]` is known to be a lower bound of `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` was called, the new heap maximum `a[0]`, is a lower bound of `a[m..i]`,

```
/*@
  lemma
    PartialReorderPreservesLowerBounds{K,L}:
      \forall value_type* a, integer m, n;
        0 < m <= n                                     ==>
        (\forall integer k; 0 <= k < m
          ==> LowerBound{K}(a, m, n, \at(a[k],K)))    ==>
        MultisetUnchanged{K,L}(a, 0, m)               ==>
        Unchanged{K,L}(a, m, n)                       ==>
        LowerBound{L}(a, m, n, \at(a[0],L));
*/
```

Listing 9.14: The lemma `PartialReorderPreservesLowerBounds`

Lemma `ReorderImpliesMatch` 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 lemmas above.

```
/*@
  lemma
    ReorderImpliesMatch{K,L}:
      \forall value_type *a, integer i, n;
        0 <= i < n                    ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        HasValue{K}(a, n, \at(a[i],L));
*/
```

Listing 9.15: The lemma `ReorderImpliesMatch`

Lemmas handling some effects of `swap`

The next group of lemmas is related to proofs dealing with the effect of the `swap` call. We used the predicate `SwappedInside`, shown in Listing 9.16 rather than the literal postcondition of `swap`, since this leads to better performance of the provers.

```
/*@
  predicate
    SwappedInside{K,L}(value_type* a, integer i, integer k, integer n) =
      0 <= i < k < n              &&
      \at(a[i],K) == \at(a[k],L)  &&
      \at(a[k],K) == \at(a[i],L)  &&
      Unchanged{K,L}(a, 0,   i)   &&
      Unchanged{K,L}(a, i+1, k)   &&
      Unchanged{K,L}(a, k+1, n);
*/
```

Listing 9.16: The predicate `SwappedInside`

Our next lemma, `SwappedInsideMultisetUnchanged`, which is shown in Listing 9.17, employs the predicate `SwappedInside` from Listing 9.16. It states that swapping the elements `a[i]` and `a[k]` is a particular kind of reordering on the range `a[i..k]`.

```
/*@
  lemma
    SwappedInsideMultisetUnchanged{K,L}:
      \forall value_type* a, integer i, k, n;
        SwappedInside{K,L}(a, i, k, n)     ==>
        MultisetUnchanged{K,L}(a, i, k+1);
*/
```

Listing 9.17: The lemma `SwappedInsideMultisetUnchanged`

This lemma is extended to lemma `SwappedInsidePreservesMultisetUnchanged` (Listing 9.18), which additionally considers a left context `a[0..k-1]` and a right context `a[k..n-1]`; if the left and right context is reordered and kept untouched, respectively, and `a[i]` and `a[k]` are swapped as before, then this whole action is a reordering on the range `a[0..k]`. These two lemmas are needed to prove that the loop invariant `reorder` is preserved.

```
/*@
  lemma
    SwappedInsidePreservesMultisetUnchanged{K,L,M}:
      \forall value_type* a, integer i, k, n;
        MultisetUnchanged{K,L}(a, k)       ==>
        Unchanged{K,L}(a, k, n)            ==>
        SwappedInside{L,M}(a, i, k, n)     ==>
        MultisetUnchanged{K,M}(a, k+1);
*/
```

Listing 9.18: The lemma `SwappedInsidePreservesMultisetUnchanged`

10. Classic Sorting Algorithms

Many issues in computer science can be exemplified in the field of sorting algorithms; see e.g. [19] for a famous textbook. In this chapter, we arrange some of the most common classic sorting algorithms. Following [20], we used (C rephrasings of) functions from the C++ Standard Library as far as possible to implement the different algorithmic approaches.

- `selection_sort` in Section 10.1 presents the classic *selection sort* algorithm.[33]
- `insertion_sort` in Section 10.2 the also well-known *insertion sort* algorithm.[34]
- `heap_sort` in Section 10.3 describes the quite efficient *heap sort*, which relies on the algorithms presented in Chapter 8.[35]

All algorithms essentially share the the following contract; it is their implementations that differ fundamentally.

```
/*@
   requires valid: \valid(a + (0..n-1));

   assigns  a[0..n-1];

   ensures sorted:  Sorted(a, n);
   ensures reorder: MultisetUnchanged{Old, Here}(a, n);
*/
void xxx_sort(value_type* a, size_type n);
```

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.[36]

[33] See `https://en.wikipedia.org/wiki/Selection_sort`
[34] See `https://en.wikipedia.org/wiki/Insertion_sort`
[35] See `https://en.wikipedia.org/wiki/Heapsort`
[36] See `https://en.wikipedia.org/wiki/Introsort`

10.1. 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 constructs the sorted array, left to right, by selecting in each step the minimum element of the remaining (yet unsorted) part and *swaps* it the first element of the unsorted part. This implies that each member of the sorted initial array segment is less or equal than each member of the unsorted part.

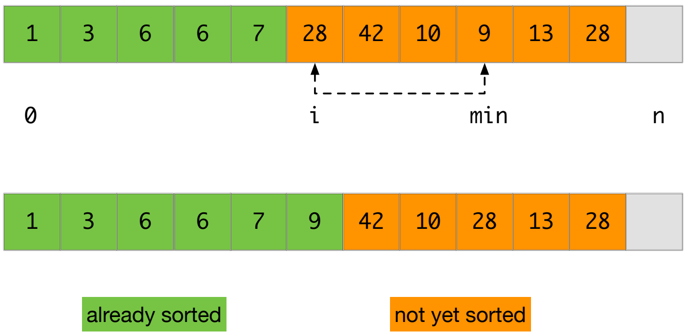

Figure 10.1.: An iteration of selection_sort

Figure 10.1 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 sorted area one field to the right.

10.1.1. Formal Specification of selection_sort

Listing 10.2 shows the ACSL specification of selection_sort.

```
/*@
   requires valid:  \valid(a + (0..n-1));

   assigns a[0..n-1];

   ensures sorted:   Sorted(a, n);
   ensures reorder:  MultisetUnchanged{Old, Here}(a, n);
*/
void selection_sort(value_type* a, size_type n);
```

Listing 10.2: The Specification of selection_sort

10.1.2. Implementation of selection_sort

The implementation of selection_sort is shown in Listing 10.3. We use min_element from Section 4.5 to find the minimum element of the unsorted area.

The loop invariants `sorted` and `lower` establish the sortedness of the initial array segment `a[0..i-1]` and, respectively, state that `a[i-1]` is a lower bound of the remaining array segment `a[i..n-1]`. Since the `min_element` call uses an address offset, we had to employ again the *shift lemmas* from Listing 5.13.

```
void selection_sort(value_type* a, size_type n)
{
  /*@
    loop invariant bound:   0 <= i <= n;
    loop invariant sorted:  Sorted(a, i);
    loop invariant sorted:  0 < i  ==> LowerBound(a, i, n, a[i-1]);
    loop invariant reorder: MultisetUnchanged{Pre,Here}(a, n);
    loop assigns   i, a[0..n-1];
    loop variant   n - i;
  */
  for (size_type i = 0; i < n; ++i) {
    const size_type sel = i + min_element(a + i, n - i);
    //@ assert reorder: i <= sel < n;
    /*@
       assigns a[sel], a[i];

       ensures reorder: a[i]   == \old(a[sel]);
       ensures reorder: a[sel] == \old(a[i]);
       ensures reorder: Unchanged{Old,Here}(a, 0,     i);
       ensures reorder: Unchanged{Old,Here}(a, i+1,   sel);
       ensures reorder: Unchanged{Old,Here}(a, sel+1, n);
    */
    swap(a + sel, a + i);
    //@ assert reorder: MultisetUnchanged{Pre,Here}(a, n);
  }
}
```

Listing 10.3: The Implementation of `selection_sort`

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` routine, from Section 6.4, is the only code that modifies `a`), it took considerable effort to prove it formally; including a statement contract that captures that captures the effects of calling `swap`. assertions in the loop body.

The main reason for introducing the statement contract is that it *transforms* the postcondition of the call to `swap` from Listing 6.8 into the hypotheses for the new lemma `SwapImpliesMultisetUnchanged` in Listing 10.4. This lemma, which relies on the lemmas from Listing 8.43, captures the fact that *swapping two elements of an array* is a *reordering*.

```
/*@
  lemma
    SwapImpliesMultisetUnchanged{K,L}:
      \forall value_type *a, integer i, k, n;
        0 <= i <= k < n                        ==>
        \at(a[i],K) == \at(a[k],L)            ==>
        \at(a[k],K) == \at(a[i],L)            ==>
        MultisetUnchanged{K,L}(a, 0,    i)    ==>
        MultisetUnchanged{K,L}(a, i+1, k)     ==>
        MultisetUnchanged{K,L}(a, k+1, n)     ==>
        MultisetUnchanged{K,L}(a, n);
*/
```

Listing 10.4: The lemma `SwapImpliesMultisetUnchanged`

10.2. 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 sorted. Unlike `selection_sort`, however, `insertion_sort` adds `a[i]` to the sorted range in the `i`th step (see Figure 10.5). It determines the (rightmost) appropriate position to insert `a[i]` by a call to `upper_bound` (Section 5.2), and then uses `rotate` (Section 6.11) to actually perform the insertion.

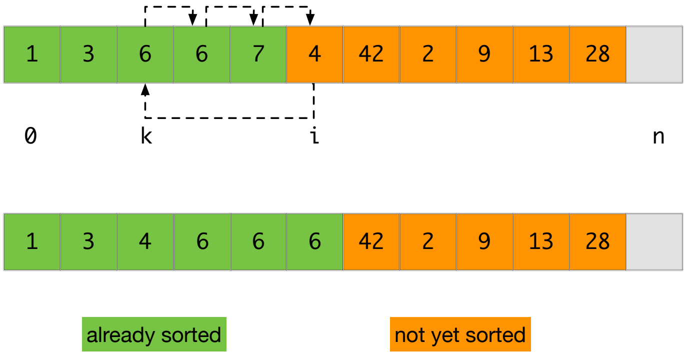

Figure 10.5.: An iteration of `insertion_sort`

10.2.1. Formal Specification of `insertion_sort`

Listing 10.6 shows our (generic sorting) contract for `insertion_sort`.

```
/*@
   requires valid:  \valid(a + (0..n-1));

   assigns a[0..n-1];

   ensures sorted:   Sorted(a, n);
   ensures reorder:  MultisetUnchanged{Old, Here}(a, n);
*/
void insertion_sort(value_type* a, size_type n);
```

Listing 10.6: The Specification of `insertion_sort`

10.2.2. Implementation of `insertion_sort`

The implementation of `insertion_sort` is shown in Listing 10.7. 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 ascending order are labelled `sorted`. 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 sorted:    Sorted(a, i);
    loop invariant reorder:   MultisetUnchanged{Pre,Here}(a, 0, 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 = 0; i < n; ++i) {
    const size_type k = upper_bound(a, i, a[i]);
    //@ assert bound:   0 <= k <= i;

    /*@
       requires sorted:  UpperBound(a, k, a[i]);
       requires sorted:  StrictLowerBound(a, k, i, a[i]);
       requires sorted:  Sorted(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:   MultisetUnchanged{Old,Here}(a, 0, k);
       ensures reorder:   EqualRanges{Old,Here}(a, k, i, k+1);
       ensures reorder:   EqualRanges{Old,Here}(a, i, i+1, k);
       ensures sorted:    Sorted(a, 0, k);
       ensures sorted:    UpperBound(a, k, a[k]);
    */
    rotate(a + k, i - k, i - k + 1);

    //@ assert sorted:  Sorted(a, k+1, i+1);
    //@ assert sorted:  StrictLowerBound(a, k+1, i+1, a[k]);
    //@ assert sorted:  Sorted(a, i+1);

    //@ assert reorder: MultisetUnchanged{Pre,Here}(a, 0, i+1);
  }
}
```

Listing 10.7: The 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 `rotate`'s contract (Listing 6.29) and repeat its verification proof.[37]

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 (Section 2.3). 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.

[37] ACSL allows to declare a function several times with different contracts; they are merged into a single one. Alternatively, non-disjoint `behaviors`, with empty `assumes` clauses, allow contract merging and provide finer control over the set of hypotheses generated from e.g. an `assert`.

Supporting lemmas related to property `sorted`

Lemma `EqualRangesPreservesSorted` (Listing 10.8) 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 `rotate` call preserves the order of those elements that are shifted upwards (cf. Figure 10.5).

```
/*@
   lemma
     EqualRangesPreservesSorted{K,L}:
       \forall value_type* a, integer m, n, p;
         Sorted{K}(a, m, n)             ==>
         EqualRanges{K,L}(a, m, n, m+p)  ==>
         Sorted{L}(a, m+p, n+p);
*/
```

Listing 10.8: The lemma `EqualRangesPreservesSorted`

Lemma `RotatePreservesStrictLowerBound` (Listing 10.9) is used to prove that the range `a[k..i-1]` having `a[i]` as strict lower bound before our `rotate` call ensures that it has `a[k]` as such a bound after the call. Note that this lemma reflects the situation at our particular `rotate` call site.

```
/*@
   lemma
     RotatePreservesStrictLowerBound{K,L}:
       \forall value_type* a, integer m, n;
         StrictLowerBound{K}(a, m, n, \at(a[n],K)) ==>
         EqualRanges{K,L}(a, m, n, m+1)          ==>
         EqualRanges{K,L}(a, n, n+1, m)          ==>
         StrictLowerBound{L}(a, m+1, n+1, \at(a[m],L));
*/
```

Listing 10.9: The lemma `RotatePreservesStrictLowerBound`

Supporting lemmas related to property `reorder`

Lemma `RotateImpliesMultisetUnchanged` (Listing 10.10) establishes that `rotate` by one produces just a reordering of the range it is applied to. Again, this lemma has been special-tailored to our `rotate` call; one could also think of a more general version that allows for arbitrary rotation amounts.

```
/*@
   lemma
     RotateImpliesMultisetUnchanged{K,L}:
       \forall value_type* a, integer m, n;
         0 <= m <= n                          ==>
         EqualRanges{K,L}(a, m, n, m+1)      ==>
         EqualRanges{K,L}(a, n, n+1, m)      ==>
         MultisetUnchanged{K,L}(a, m, n+1);
*/
```

Listing 10.10: The lemma `RotateImpliesMultisetUnchanged`

Finally, lemma `EqualRangesPreservesCount` (Listing 10.11) 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`. It is useful in the proof of the previous lemma, `RotateImpliesMultisetUnchanged`, since the predicate `MultisetUnchanged` (Listing 6.5) is defined via the `Count` function (Listing 3.32).

```
/*@
   lemma
     EqualRangesPreservesCount{K,L}:
       \forall value_type *a, v, integer m, n, p;
         0 <= m <= n                    ==>
         EqualRanges{K,L}(a, m, n, p)   ==>
         Count{K}(a, m, n, v) == Count{L}(a, p, p + (n-m), v);
*/
```

Listing 10.11: The lemma `EqualRangesPreservesCount`

10.3. 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 data structure discussed in Chapter 8 to efficiently bring the given array into an ascending order.

10.3.1. Formal Specification of heap_sort

Listing 10.12 shows the ACSL specification of heap_sort.

```
/*@
   requires valid:  \valid(a + (0..n-1));

   assigns a[0..n-1];

   ensures sorted:   Sorted(a, n);
   ensures reorder:  MultisetUnchanged{Old, Here}(a, n);
*/
void heap_sort(value_type* a, size_type n);
```

Listing 10.12: The Specification of heap_sort

10.3.2. Implementation of heap_sort

The implementation of heap_sort, shown in Listing 10.13, is straightforward. Given the unsorted array a, we use make_heap to arrange it into a heap; after that, we use sort_heap to draw from this heap the elements in ascending order.

```
void heap_sort(value_type* a, size_type n)
{
  make_heap(a, n);
  sort_heap(a, n);
}
```

Listing 10.13: The Implementation of heap_sort

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` [14, §23.6.5] 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 [21].

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.

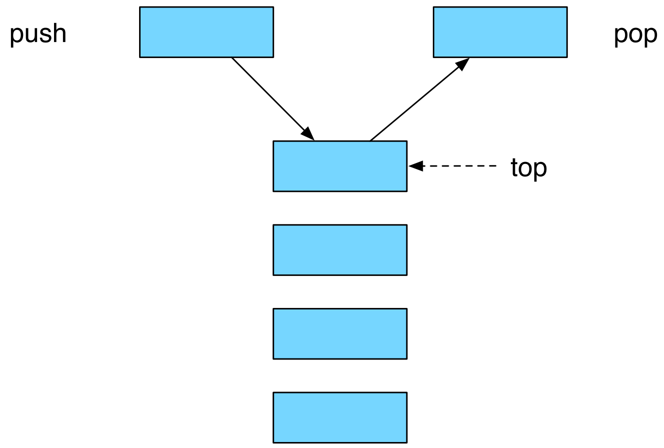

Figure 11.1.: Push and pop on a stack

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).

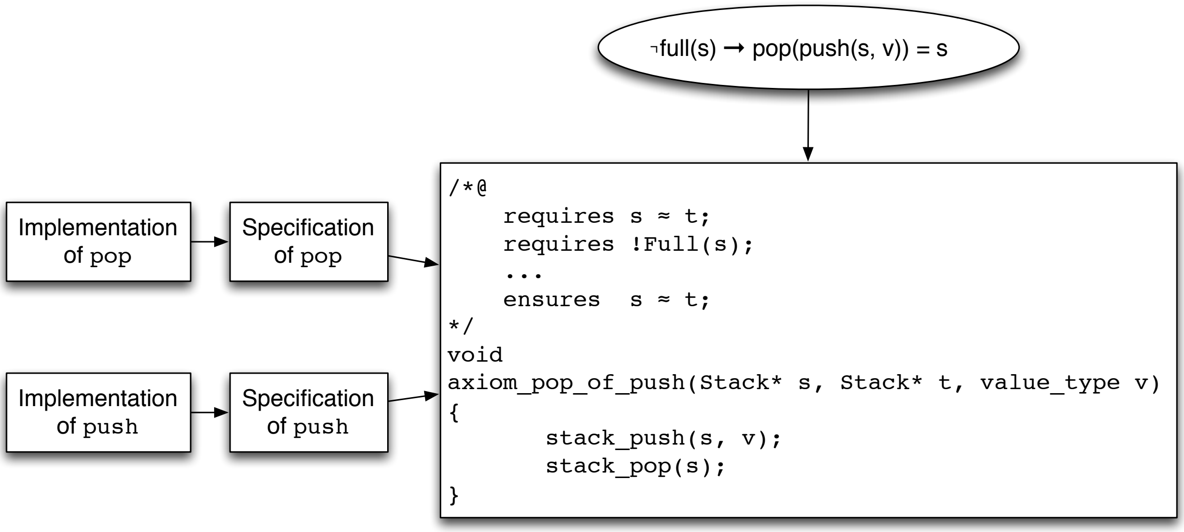

Figure 11.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (Section 11.2),
2. define a basic implementation of C data structures (only one in our example, viz. `struct Stack`; see Section 11.3) and some invariants the instances of them have to obey (Section 11.4),
3. provide for each stack function an ACSL contract and a C implementation (Section 11.7),
4. verify each function against its contract (Section 11.7),
5. transform the axioms into ACSL-annotated C code (Section 11.8), and
6. verify that code, using access function contracts and data-type invariants as necessary (Section 11.8).

Section 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 Section 11.6 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[38], 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

$$0 \leq \text{size}(s) \leq c. \tag{11.4}$$

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

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{11.6}$$

[38] There is an analogy in geometry: Euclid (e.g. [22]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [23] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

11.2.3. 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

$$\mathrm{size}(\mathrm{pop}(s)) = \mathrm{size}(s) - 1, \qquad \text{if} \quad \neg\mathrm{empty}(s). \tag{11.7}$$

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

$$\mathrm{pop}(\mathrm{push}(s, v)) = s, \qquad \text{if} \quad \neg\mathrm{full}(s). \tag{11.8}$$

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.

$$\mathrm{push}(\mathrm{pop}(s), \mathrm{top}(s)) = s, \qquad \text{if} \quad \neg\mathrm{empty}(s). \tag{11.9}$$

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

$$\mathrm{push}(s, \mathrm{err}) = s. \tag{11.10}$$

However, we then could derive for any non-full and non-empty stack `s`, that

$$\begin{aligned} \mathrm{size}(s) &= \mathrm{size}(\mathrm{pop}(\mathrm{push}(s, \mathrm{err}))) && \text{by 11.8} \\ &= \mathrm{size}(\mathrm{pop}(s)) && \text{as assumed in 11.10} \\ &= \mathrm{size}(s) - 1 && \text{by 11.7} \end{aligned}$$

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 Listing 11.3.

```
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.

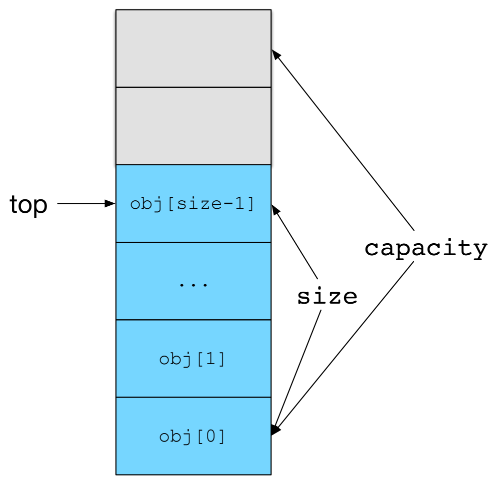

Figure 11.4.: Interpreting the data structure `Stack`

Based on the stack functions from Section 11.2, we declare in Listing 11.5 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 [14, §23.6.5]. 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. Below, we will define an ACSL-predicate Invariant that discriminates valid and invalid instances.

Before, we introduce in Listing 11.6 the auxiliary logical function Capacity and Size which we can use in specifications to refer to the fields capacity and size of Stack, respectively. This listing also contains the logical function Top 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.

```
/*@
  logic size_type Capacity{L}(Stack* s) = s->capacity;

  logic size_type Size{L}(Stack* s) = s->size;

  logic value_type* Storage{L}(Stack* s) = s->obj;

  logic value_type Top{L}(Stack* s) = s->obj[s->size-1];
*/
```

Listing 11.6: The logical functions Capacity, Size and Top

We also introduce in Listing 11.7 two predicates 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:

```
/*@
  predicate
    Empty{L}(Stack* s) =  Size(s) == 0;

  predicate
    Full{L}(Stack* s)  =  Size(s) == Capacity(s);
*/
```

Listing 11.7: Predicates for empty an full stacks

- 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 formalized in the predicate `Invariant` of Listing 11.8.

```
/*@
  predicate
    Invariant{L}(Stack* s) =
      0 < Capacity(s) &&
      0 <= Size(s) <= Capacity(s) &&
      \valid(Storage(s) + (0..Capacity(s)-1)) &&
      \separated(s, Storage(s) + (0..Capacity(s)-1));
*/
```

Listing 11.8: The predicate `Invariant`

Note how the use of the previously defined logical functions and predicates allows us to define the stack invariant without directly referring to the fields of `Stack`.

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* [24, 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 `Equal` as in Listing 11.9.

```
/*@
  predicate
    Equal{S,T}(Stack* s, Stack* t) =
      Size{S}(s) == Size{T}(t) &&
      EqualRanges{S,T}(Storage{S}(s), Size{S}(s), Storage{T}(t));
*/
```

Listing 11.9: Equality of stacks

Our use of labels in Listing 11.9 makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `EqualRanges` from Page 36). However, this definition of `Equal` will allow us later to compare the same stack object at different points of a program. The logical expression `Equal{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.[39]

If equal stacks can have different capacities then, according to our definition of the predicate `Full` in Listing 11.7, 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 11.10 are considered equal although there is are obvious differences in their internal arrays.

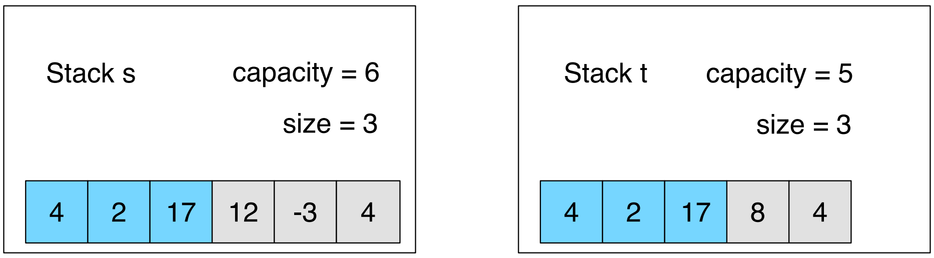

Figure 11.10.: Example of two equal stacks

[39] See `http://www.cplusplus.com/reference/vector/vector/capacity`

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.

$$\text{reflexivity} \qquad \forall s \in S : s = s, \tag{11.11a}$$
$$\text{symmetry} \qquad \forall s, t \in S : s = t \implies t = s, \tag{11.11b}$$
$$\text{transitivity} \qquad \forall s, t, u \in S : s = t \wedge t = u \implies s = u. \tag{11.11c}$$

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.

Listing 11.11 shows a formalization of these three rules for the relation `Equal`; it can be automatically verified that they are a consequence of the definition of `Equal` in Listing 11.9.

```
/*@
  lemma
    StackEqualReflexive{S} :
      \forall Stack* s; Equal{S,S}(s, s);

  lemma
    StackEqualSymmetric{S,T} :
      \forall Stack *s, *t;
        Equal{S,T}(s, t) ==> Equal{T,S}(t, s);

  lemma
    StackEqualTransitive{S,T,U}:
      \forall Stack *s, *t, *u;
        Equal{S,T}(s, t) && Equal{T,U}(t, u) ==> Equal{S,U}(s, u);
*/
```

Listing 11.11: Equality of stacks is an equivalence relation

The two stacks in Figure 11.10 show that an equivalence class of `Equal` can contain more than one element.[40] The stacks `s` and `t` in Figure 11.10 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*.[41] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `Equal`.[42] The question of well-definition will play an important role when verifying the functions of the `Stack` (see Section 11.7).

[40] This is a common situation in mathematics. For example, the equivalence class of the rational number $\frac{1}{2}$ contains infinitely many elements, viz. $\frac{1}{2}, \frac{2}{4}, \frac{7}{14}, \ldots$.

[41] This is why mathematicians know that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

[42] See `http://en.wikipedia.org/wiki/Well-definition`.

11.6. Runtime equality of stacks

The function `stack_equal` is the C equivalent for the `Equal` predicate. The specification of the function `stack_equal` is shown in Listing 11.12. Note that this specifications explicitly refers to valid stacks.

```
/*@
    requires valid: \valid(s) && Invariant(s);
    requires valid: \valid(t) && Invariant(t);

    assigns \nothing;

    ensures equal:     \result == 1  <==>  Equal{Here,Here}(s, t);
    ensures not_equal: \result == 0  <==> !Equal{Here,Here}(s, t);
*/
bool stack_equal(const Stack* s, const Stack* t);
```

Listing 11.12: Specification of `stack_equal`

The implementation of `stack_equal` in Listing 11.13 compares two stacks according to the same rules of predicate `Equal`.

```
bool stack_equal(const Stack* s, const Stack* t)
{
  return (s->size == t->size) && equal(s->obj, s->size, t->obj);
}
```

Listing 11.13: Implementation of `stack_equal`

11.7. Verification of stack functions

In this section we verify the functions `stack_init` (Section 11.7.1), `stack_size` (Section 11.7.2), `stack_empty` (Section 11.7.3), `stack_full` (Section 11.7.4), `stack_top` (Section 11.7.5), and `stack_push` (Section 11.7.6) `stack_pop` (Section 11.7.7), 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`[43] accompanied by a an ACSL contract that expresses that the implementation of `stack_foo` is well-defined. Figure 11.14 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `Stack` equality.

Specification of pop

```
/*@
  requires s ≈ t;
  requires !Empty(s);
    ...
  ensures  s ≈ t;
*/
void stack_pop_wd(Stack *s, Stack *t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Figure 11.14.: 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 Section 11.8 we will show that the *interplay* of these functions satisfy the stack axioms from Section 11.2.

[43] The suffix `_wd` stands for *well definition*

11.7.1. The function stack_init

Listing 11.15 shows the ACSL 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 invariant: Invariant(s);
  ensures capacity:  Capacity(s) == capacity;
  ensures empty:     Empty(s);
  ensures storage:   Storage(s) == storage;
*/
void stack_init(Stack* s, value_type* storage, size_type capacity);
```

Listing 11.15: Specification of `stack_init`

Listing 11.15 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     = 0;
}
```

Listing 11.16: Implementation of `stack_init`

11.7.2. The function `stack_size`

The function `stack_size` is the runtime version of the logical function `Size` from Listing 11.6 on Page 184. The specification of `stack_size` in Listing 11.17 simply states that `stack_size` produces the same result as `Size`.

```
/*@
    requires valid: \valid(s) && Invariant(s);

    assigns \nothing;

    ensures size: \result == Size(s);
*/
size_type stack_size(const Stack* s);
```

Listing 11.17: Specification of `stack_size`

As in the definition of the logical function `Size` the implementation of `stack_size` in Figure 11.18 simply returns the field `size`.

```
size_type stack_size(const Stack* s)
{
  return s->size;
}
```

Listing 11.18: Implementation of `stack_size`

Listing 11.19 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) && Invariant(s);
  requires valid: \valid(t) && Invariant(t);
  requires equal: Equal{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 11.19: Well-definition of `stack_size`

11.7.3. The function `stack_empty`

The function `stack_empty` is the runtime version of the predicate `Empty` from Listing 11.7 on Page 185.

```
/*@
    requires valid: \valid(s) && Invariant(s);

    assigns \nothing;

    ensures empty:     \result == 1  <==>  Empty(s);
    ensures not_empty: \result == 0  <==> !Empty(s);
*/
bool stack_empty(const Stack* s);
```

Listing 11.20: Specification of `stack_empty`

As in the definition of the predicate `Empty` the implementation of `stack_empty` in Figure 11.21 simply checks whether the size of the stack is zero.

```
bool stack_empty(const Stack* s)
{
  return stack_size(s) == 0;
}
```

Listing 11.21: Implementation of `stack_empty`

Listing 11.22 shows our check whether `stack_empty` is well-defined.

```
/*@
  requires valid: \valid(s) && Invariant(s);
  requires valid: \valid(t) && Invariant(t);
  requires equal: Equal{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: Well-definition of `stack_empty`

11.7.4. The function `stack_full`

The function `stack_full` is the runtime version of the predicate `Full` from Listing 11.7 on Page 185.

```
/*@
    requires valid: \valid(s) && Invariant(s);

    assigns \nothing;

    ensures full:     \result == 1  <==>  Full(s);
    ensures not_full: \result == 0  <==> !Full(s);
*/
bool stack_full(const Stack* s);
```

Listing 11.23: Specification of `stack_full`

As in the definition of the predicate `Full` the implementation of `stack_full` in Figure 11.24 simply checks whether the size of the stack equals its capacity.

```
bool stack_full(const Stack* s)
{
  return stack_size(s) == s->capacity;
}
```

Listing 11.24: Implementation of `stack_full`

Note that with our definition of stack equality (Section 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, `Full` is not well-defined.

11.7.5. The function stack_top

The function stack_top is the runtime version of the logical function Top from Listing 11.6 on Page 184. The specification of stack_top in Listing 11.25 simply states that for non-empty stacks stack_top produces the same result as Top which in turn just returns the element obj[size-1] of Stack.

```
/*@
    requires valid: \valid(s) && Invariant(s);

    assigns \nothing;

    ensures top: !Empty(s) ==> \result == Top(s);
*/
value_type stack_top(const Stack* s);
```

Listing 11.25: Specification of stack_top

For a non-empty stack the implementation of stack_top in Figure 11.26 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 - 1];
  }
  else {
    return s->obj[0];
  }
}
```

Listing 11.26: Implementation of stack_top

Listing 11.27 shows our check whether stack_top well-defined for non-empty stacks.

```
/*@
  requires valid: \valid(s) && Invariant(s) && !Empty(s);
  requires valid: \valid(t) && Invariant(t) && !Empty(t);
  requires equal: Equal{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 11.27: Well-definition of stack_top

Since our axioms in Section 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.

11.7.6. The function `stack_push`

Listing 11.28 shows the ACSL 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) && Invariant(s);

  assigns s->size;
  assigns s->obj[s->size];

  behavior not_full:
    assumes !Full(s);

    assigns s->size;
    assigns s->obj[s->size];

    ensures valid:     \valid(s) && Invariant(s);
    ensures size:      Size(s) == Size{Old}(s) + 1;
    ensures top:       Top(s) == v;
    ensures not_empty: !Empty(s);
    ensures unchanged: Unchanged{Old,Here}(Storage(s), Size{Old}(s));
    ensures storage:   Storage(s) == Storage{Old}(s);
    ensures capacity:  Capacity(s) == Capacity{Old}(s);

  behavior full:
    assumes Full(s);

    assigns \nothing;

    ensures valid:     \valid(s) && Invariant(s);
    ensures full:      Full(s);
    ensures unchanged: Unchanged{Old,Here}(Storage(s), Size(s));
    ensures size:      Size(s) == Size{Old}(s);
    ensures storage:   Storage(s) == Storage{Old}(s);
    ensures capacity:  Capacity(s) == Capacity{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void stack_push(Stack* s, value_type v);
```

Listing 11.28: Specification of `stack_push`

The implementation of `stack_push` is shown in Listing 11.29. 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)) {
    s->obj[s->size++] = v;
  }
}
```

Listing 11.29: Implementation of `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. 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 define in Listing 11.30 the predicate Separated.

```
/*@
   predicate
     Separated(Stack* s, Stack* t) =
       \separated(s, s->obj + (0..s->capacity-1),
                  t, t->obj + (0..t->capacity-1));
*/
```

Listing 11.30: The predicate Separated

Listing 11.31 shows our formalization of the well-definition for stack_push.

```
/*@
  requires valid:     \valid(s) && Invariant(s);
  requires valid:     \valid(t) && Invariant(t);
  requires equal:     Equal{Here,Here}(s, t);
  requires not_full:  !Full(s) && !Full(t);
  requires separated: Separated(s, t);

  assigns s->size, s->obj[s->size];
  assigns t->size, t->obj[t->size];

  ensures  valid:     Invariant(s) && Invariant(t);
  ensures  equal:     Equal{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:   Top(s) == v;
  //@ assert top:   Top(t) == v;
  //@ assert equal: EqualRanges{Here,Here}(Storage(s), Size{Pre}(s), Storage(t));
}
```

Listing 11.31: Well-definition of stack_push

In order to achieve an automatic verification of the well-definition of stack_push we added in Listing 11.31 the assertions top and equal and introduced the lemma StackPushEqual from Listing 11.32.

```
/*@
  lemma
    StackPushEqual{K,L}:
      \forall Stack *s, *t;
         Equal{K,K}(s,t)              ==>
         Size{L}(s) == Size{K}(s) + 1 ==>
         Size{L}(s) == Size{L}(t)     ==>
         Top{L}(s)  == Top{L}(t)      ==>
         EqualRanges{L,L}(Storage{L}(s), Size{K}(s), Storage{L}(t)) ==>
         Equal{L,L}(s,t);
*/
```

Listing 11.32: The lemma `StackPushEqual`

11.7.7. The function `stack_pop`

Listing 11.33 shows the ACSL 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) && Invariant(s);

  assigns s->size;

  ensures  valid: \valid(s) && Invariant(s);

  behavior not_empty:
    assumes !Empty(s);

    assigns s->size;

    ensures size:      Size(s) == Size{Old}(s) - 1;
    ensures full:      !Full(s);
    ensures unchanged: Unchanged{Old,Here}(Storage(s), Size(s));
    ensures storage:   Storage(s) == Storage{Old}(s);
    ensures capacity:  Capacity(s) == Capacity{Old}(s);

  behavior empty:
    assumes Empty(s);

    assigns \nothing;

    ensures empty:     Empty(s);
    ensures unchanged: Unchanged{Old,Here}(Storage(s), Size(s));
    ensures size:      Size(s) == Size{Old}(s);
    ensures storage:   Storage(s) == Storage{Old}(s);
    ensures capacity:  Capacity(s) == Capacity{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void stack_pop(Stack* s);
```

Listing 11.33: Specification of `stack_pop`

The implementation of `stack_pop` is shown in Listing 11.34. 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 11.34: Implementation of `stack_pop`

Listing 11.35 shows our check whether `stack_pop` is well-defined. As in the case of `stack_push` we use the predicate `Separated` (Listing 11.30) in order to express that there is no aliasing between the two stack arguments.

```
/*@
  requires valid:     \valid(s) && Invariant(s);
  requires valid:     \valid(t) && Invariant(t);
  requires equal:     Equal{Here,Here}(s, t);
  requires separated: Separated(s, t);

  assigns s->size;
  assigns t->size;

  ensures valid: Invariant(s);
  ensures valid: Invariant(t);
  ensures equal: Equal{Here,Here}(s, t);
*/
void stack_pop_wd(Stack* s, Stack* t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Listing 11.35: Well-definition of `stack_pop`

11.8. Verification of stack axioms

In this section we show that the stack functions defined in Section 11.7 satisfy the stack Axioms of Section 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.

11.8.1. Resetting a stack

Our formulation in ACSL/C of the Axiom in Equation (11.1) on Page 181 is shown in Listing 11.36.

```
/*@
  requires valid:     \valid(s);
  requires size:      0 < n;
  requires valid:     \valid(a + (0..n-1));
  requires separated: \separated(s, a + (0..n-1));

  assigns s->obj, s->capacity, s->size;

  ensures valid: Invariant(s);
  ensures size:  \result == 0;
*/
size_type axiom_size_of_init(Stack* s, value_type* a, size_type n)
{
  stack_init(s, a, n);
  return stack_size(s);
}
```

Listing 11.36: Specification of Axiom (11.1)

11.8.2. 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) && Invariant(s);
  requires not_full: !Full(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures valid: Invariant(s);
  ensures size:  \result == Size{Old}(s) + 1;
*/
size_type axiom_size_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_size(s);
}
```

Listing 11.37: Specification of Axiom (11.5)

Except for the `assigns` clauses, the ACSL-specification refers only to encapsulating logical functions and predicates defined in Section 11.4. If ACSL would provide a means to define encapsulating logical functions returning also sets of memory locations, the expressions in `assigns` clauses would not need to refer to the details of our `Stack` implementation.[44] 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) && Invariant(s);
  requires not_full: !Full(s);

  assigns s->size;
  assigns 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 11.38: Specification of Axiom (11.6)

[44] In [9, §2.3.4], a powerful sublanguage to build memory location set expressions is defined. We will explore its capabilities in a later version.

11.8.3. 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) && Invariant(s);
  requires  empty: !Empty(s);

  assigns   s->size;

  ensures   size: \result == Size{Old}(s) - 1;
*/
size_type axiom_size_of_pop(Stack* s)
{
  stack_pop(s);
  return stack_size(s);
}
```

Listing 11.39: Specification of Axiom (11.7)

```
/*@
  requires  valid:    \valid(s) && Invariant(s);
  requires  not_full: !Full(s);

  assigns   s->size;
  assigns   s->obj[s->size];

  ensures   equal:    Equal{Pre,Here}(s, s);
*/
void axiom_pop_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  stack_pop(s);
}
```

Listing 11.40: Specification of Axiom (11.8)

```
/*@
  requires  valid:     \valid(s) && Invariant(s);
  requires  not_empty:  !Empty(s);

  assigns   s->size;
  assigns   s->obj[s->size-1];

  ensures   equal: Equal{Old,Here}(s, s);
*/
void axiom_push_of_pop_top(Stack* s)
{
  const value_type val = stack_top(s);
  stack_pop(s);
  stack_push(s, val);
}
```

Listing 11.41: Specification of Axiom (11.9)

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 3–11 could be deductively verified.

Within Frama-C, the Frama-C/WP plug-in [2] enables deductive verification of C programs that have been annotated with the ANSI/ISO-C Specification Language (ACSL) [1]. 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 and the associated provers that we used in this release we refer to Section A.1. In Sections A.2 and A.3 we show detailed verification results for different scenarios how the provers are called.

A.1. Verification settings

This section gives all settings that depend on the software release of Frama-C, Why3, or one if its employed provers. For our experiments we used Frama-C [3, v18.0 (Argon)] and the Why3 platform [25, v0.88.3]

Here are the most important options of Frama-C that we used in for almost all functions.[45]

```
-pp-annot
-no-unicode
-wp
-wp-rte
-wp-model Typed+ref
-warn-unsigned-overflow
-warn-unsigned-downcast
-wp-timeout 10
-wp-steps 1000
-wp-coq-timeout 10
```

The individual provers and their versions are listed in the following Table A.1. All provers, except Coq, are automatic provers.

[45] For the `my_lrand48()` function in `random_shuffle`, the option `-warn-unsigned-overflow` is disabled as explained in Section 6.17.2.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.2.0	[26]
CVC4	automatic	1.6	[27]
CVC3	automatic	2.4.1	[28]
Z3	automatic	4.8.3	[29]
E	automatic	2.2	[30]
Coq	interactive	8.7.2	[31]

Table A.1.: Information of automatic and interactive theorem provers

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*.[46] 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.2.

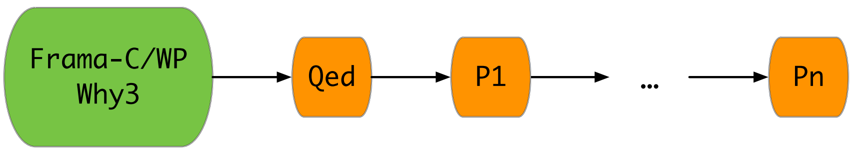

Figure A.2.: Verification pipeline of automatic and interactive theorem provers

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.2).

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
`find`	§3.1	19/19	(100%)	10	9	·	·	·	·	·
`find` (2)	§3.2	19/19	(100%)	10	9	·	·	·	·	·
`find_first_of`	§3.3	25/25	(100%)	16	9	·	·	·	·	·
`adjacent_find`	§3.4	22/22	(100%)	10	12	·	·	·	·	·
`mismatch`	§3.5	20/20	(100%)	10	10	·	·	·	·	·
`equal`	§3.5	7/ 7	(100%)	6	1	·	·	·	·	·
`search`	§3.6	30/30	(100%)	18	12	·	·	·	·	·
`search_n`	§3.7	29/29	(100%)	11	18	·	·	·	·	·
`find_end`	§3.8	28/28	(100%)	15	13	·	·	·	·	·
`count`	§3.9	19/19	(100%)	7	12	·	·	·	·	·

Table A.3.: Results for non-mutating algorithms

[46] Sequential processing is achieved by passing the option `-wp-par 1` to Frama-C/WP.

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
properties of operator <	§4.1	6/ 6	(100%)	4	2	·	·	·	·	·
`max_element`	§4.2	24/24	(100%)	13	11	·	·	·	·	·
`max_element` (2)	§4.3	24/24	(100%)	12	12	·	·	·	·	·
`max_seq`	§4.4	8/ 8	(100%)	5	3	·	·	·	·	·
`min_element`	§4.5	24/24	(100%)	12	12	·	·	·	·	·

Table A.4.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
`lower_bound`	§5.1	19/19	(100%)	5	14	·	·	·	·	·
`upper_bound`	§5.2	19/19	(100%)	7	12	·	·	·	·	·
`equal_range`	§5.3	20/20	(100%)	17	3	·	·	·	·	·
`equal_range` (2)	§5.3	64/64	(100%)	24	36	·	·	1	·	3
`binary_search`	§5.4	10/10	(100%)	8	2	·	·	·	·	·
`binary_search` (2)	§5.4	10/10	(100%)	8	2	·	·	·	·	·

Table A.5.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
`fill`	§6.3	12/12	(100%)	4	8	·	·	·	·	·
`swap`	§6.4	7/ 7	(100%)	7	·	·	·	·	·	·
`swap_ranges`	§6.5	22/22	(100%)	5	17	·	·	·	·	·
`copy`	§6.6	15/15	(100%)	4	11	·	·	·	·	·
`copy_backward`	§6.7	17/17	(100%)	7	10	·	·	·	·	·
`reverse_copy`	§6.8	17/17	(100%)	4	13	·	·	·	·	·
`reverse`	§6.9	24/24	(100%)	5	18	·	1	·	·	·
`rotate_copy`	§6.10	17/17	(100%)	5	12	·	·	·	·	·
`rotate`	§6.11	24/24	(100%)	10	14	·	·	·	·	·
`replace_copy`	§6.12	20/20	(100%)	7	13	·	·	·	·	·
`replace`	§6.13	15/15	(100%)	4	11	·	·	·	·	·
`remove_copy`	§6.14	34/34	(100%)	10	23	1	·	·	·	·
`unique_copy`	§6.15	26/26	(100%)	8	18	·	·	·	·	·
`unique_copy` (2)	§6.16	54/54	(100%)	9	33	5	2	1	1	3
`random_shuffle`	§6.17	49/49	(100%)	22	24	1	1	·	·	1

Table A.6.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
iota	§7.1	16/16	(100%)	7	9	·	·	·	·	·
accumulate	§7.2	14/14	(100%)	6	8	·	·	·	·	·
inner_product	§7.3	19/19	(100%)	6	13	·	·	·	·	·
partial_sum	§7.4	38/38	(100%)	9	25	4	·	·	·	·
adjacent_difference	§7.5	32/32	(100%)	11	17	4	·	·	·	·
partial_sum_inv	§7.6	18/18	(100%)	8	10	·	·	·	·	·
adjacent_difference_inv	§7.7	26/26	(100%)	7	16	2	·	·	1	·

Table A.7.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
is_heap	§8.3	24/24	(100%)	6	18	·	·	·	·	·
push_heap	§8.4	96/96	(100%)	33	49	10	3	·	·	1
pop_heap	§8.5	92/93	(98%)	49	39	3	·	·	·	1
make_heap	§8.6	34/34	(100%)	15	19	·	·	·	·	·
sort_heap	§8.7	50/50	(100%)	16	32	1	·	·	·	1

Table A.8.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
is_sorted	§9.1	15/ 15	(100%)	7	7	·	·	·	·	1
partial_sort	§9.2	114/114	(100%)	40	59	2	1	·	·	12

Table A.9.: Results for algorithms related to sorting

Algorithm		Verification Conditions		Individual Provers						
				QD	AE	C4	C3	Z3	EP	CQ
selection_sort	§10.1	54/54	(100%)	17	30	1	1	4	·	1
insertion_sort	§10.2	63/63	(100%)	18	36	2	2	·	1	4
heap_sort	§10.3	19/19	(100%)	8	11	·	·	·	·	·

Table A.10.: Results for classic sorting algorithms

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	EP	CQ
				Individual Provers						
`stack_init`	§11.7.1	14/14	(100%)	4	10	·	·	·	·	·
`stack_equal`	§11.6	18/18	(100%)	7	11	·	·	·	·	·
`stack_size`	§11.7.2	6/ 6	(100%)	1	5	·	·	·	·	·
`stack_empty`	§11.7.3	10/10	(100%)	5	5	·	·	·	·	·
`stack_full`	§11.7.4	11/11	(100%)	5	6	·	·	·	·	·
`stack_top`	§11.7.5	16/16	(100%)	6	10	·	·	·	·	·
`stack_push`	§11.7.6	43/43	(100%)	28	15	·	·	·	·	·
`stack_pop`	§11.7.7	32/32	(100%)	20	12	·	·	·	·	·

Table A.11.: Results for `Stack` functions

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	EP	CQ
				Individual Provers						
`stack_size_wd`	§11.7.2	12/12	(100%)	8	4	·	·	·	·	·
`stack_empty_wd`	§11.7.3	12/12	(100%)	8	4	·	·	·	·	·
`stack_top_wd`	§11.7.5	12/12	(100%)	8	4	·	·	·	·	·
`stack_push_wd`	§11.7.6	15/15	(100%)	3	9	3	·	·	·	·
`stack_pop_wd`	§11.7.7	12/12	(100%)	6	6	·	·	·	·	·

Table A.12.: Results for the well-definition of the `Stack` functions

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	EP	CQ
				Individual Provers						
`axiom_size_of_init`	§11.8.1	15/15	(100%)	11	4	·	·	·	·	·
`axiom_size_of_push`	§11.8.2	12/12	(100%)	9	3	·	·	·	·	·
`axiom_top_of_push`	§11.8.2	11/11	(100%)	8	3	·	·	·	·	·
`axiom_size_of_pop`	§11.8.3	11/11	(100%)	8	3	·	·	·	·	·
`axiom_pop_of_push`	§11.8.3	10/10	(100%)	6	4	·	·	·	·	·
`axiom_push_of_pop_top`	§11.8.3	15/15	(100%)	9	6	·	·	·	·	·

Table A.13.: 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 automatic provers from Table A.1.[47] Figure A.14 depicts this arrangement of provers. This arrangement of automatic theorem provers makes it a little bit easier to quantify their strength.

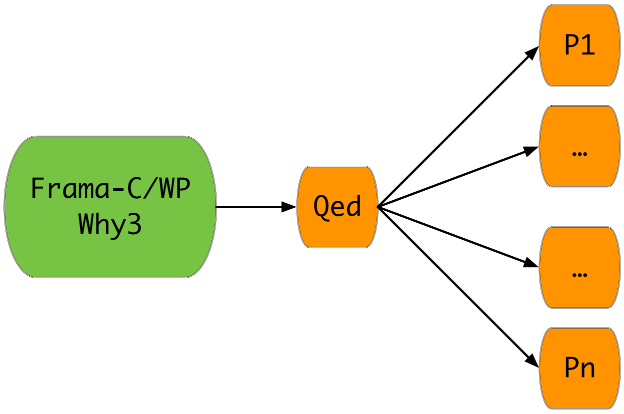

Figure A.14.: Parallel execution of automatic theorem provers

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 most proof obligations directly through Why3 to the individual provers. However, as in the sequential scenario, obligations for Alt-Ergo are directly generated by Frama-C/WP (without going through Why3).

Algorithm		**Verification Conditions**		**Individual Provers** QD	AE	C4	C3	Z3	EP
`find`	§3.1	19/19	(100%)	10	9	9	9	9	7
`find` (2)	§3.2	19/19	(100%)	10	9	9	9	6	5
`find_first_of`	§3.3	25/25	(100%)	16	9	9	9	5	3
`adjacent_find`	§3.4	22/22	(100%)	10	12	12	12	9	1
`mismatch`	§3.5	20/20	(100%)	10	10	10	10	9	4
`equal`	§3.5	7/ 7	(100%)	6	1	1	1	1	1
`search`	§3.6	30/30	(100%)	18	12	12	12	9	5
`search_n`	§3.7	27/29	(93%)	11	16	13	13	13	2
`find_end`	§3.8	28/28	(100%)	15	13	13	13	7	3
`count`	§3.9	19/19	(100%)	7	12	10	12	11	5

Table A.15.: Results for non-mutating algorithms

[47] We did not include the interactive theorem prover Coq in this setting.

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
properties of operator <	§4.1	6/ 6	(100%)	4	2	2	2	2	2
`max_element`	§4.2	24/24	(100%)	13	11	11	11	11	6
`max_element` (2)	§4.3	24/24	(100%)	12	12	12	12	9	5
`max_seq`	§4.4	8/ 8	(100%)	5	3	3	3	3	1
`min_element`	§4.5	24/24	(100%)	12	12	12	12	10	5

Table A.16.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`lower_bound`	§5.1	19/19	(100%)	5	14	14	12	12	4
`upper_bound`	§5.2	19/19	(100%)	7	12	12	10	11	2
`equal_range`	§5.3	20/20	(100%)	17	3	3	3	3	·
`equal_range` (2)	§5.3	61/64	(95%)	24	36	36	33	25	4
`binary_search`	§5.4	10/10	(100%)	8	2	2	2	1	·
`binary_search` (2)	§5.4	10/10	(100%)	8	2	2	2	1	·

Table A.17.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`fill`	§6.3	12/12	(100%)	4	8	8	8	6	3
`swap`	§6.4	7/ 7	(100%)	7	·	·	·	·	·
`swap_ranges`	§6.5	22/22	(100%)	5	17	17	17	11	6
`copy`	§6.6	15/15	(100%)	4	11	11	11	8	4
`copy_backward`	§6.7	17/17	(100%)	7	10	10	10	7	3
`reverse_copy`	§6.8	17/17	(100%)	4	13	12	13	11	3
`reverse`	§6.9	24/24	(100%)	5	19	17	17	15	3
`rotate_copy`	§6.10	17/17	(100%)	5	12	12	11	11	1
`rotate`	§6.11	24/24	(100%)	10	14	13	14	6	·
`replace_copy`	§6.12	20/20	(100%)	7	13	13	13	11	4
`replace`	§6.13	15/15	(100%)	4	11	11	11	8	5
`remove_copy`	§6.14	34/34	(100%)	10	23	20	23	20	7
`unique_copy`	§6.15	26/26	(100%)	8	18	18	18	15	3
`unique_copy` (2)	§6.16	51/54	(94%)	9	34	37	34	30	12
`random_shuffle`	§6.17	48/49	(97%)	22	24	22	24	20	6

Table A.18.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`iota`	§7.1	16/16	(100%)	7	9	9	9	7	3
`accumulate`	§7.2	14/14	(100%)	6	8	8	8	8	4
`inner_product`	§7.3	19/19	(100%)	6	13	13	13	13	5
`partial_sum`	§7.4	38/38	(100%)	9	27	29	26	15	29
`adjacent_difference`	§7.5	32/32	(100%)	11	19	21	20	19	18
`partial_sum_inv`	§7.6	18/18	(100%)	8	10	10	10	7	10
`adjacent_difference_inv`	§7.7	26/26	(100%)	7	15	18	16	10	19

Table A.19.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`is_heap`	§8.3	24/24	(100%)	6	18	18	17	18	4
`push_heap`	§8.4	95/96	(98%)	33	52	54	50	41	12
`pop_heap`	§8.5	91/93	(97%)	49	40	41	38	35	10
`make_heap`	§8.6	34/34	(100%)	15	19	16	17	16	6
`sort_heap`	§8.7	49/50	(98%)	16	32	30	30	25	11

Table A.20.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`is_sorted`	§9.1	14/ 15	(93%)	7	7	7	7	5	·
`partial_sort`	§9.2	101/114	(88%)	40	59	53	56	38	11

Table A.21.: Results for algorithms related to sorting

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`selection_sort`	§10.1	53/54	(98%)	17	30	25	30	28	8
`insertion_sort`	§10.2	59/63	(93%)	18	39	34	36	26	11
`heap_sort`	§10.3	19/19	(100%)	8	11	10	10	10	3

Table A.22.: Results for classic sorting algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`stack_init`	§11.7.1	14/14	(100%)	4	10	10	10	10	10
`stack_equal`	§11.6	18/18	(100%)	7	11	11	11	9	11
`stack_size`	§11.7.2	6/ 6	(100%)	1	5	5	5	5	5
`stack_empty`	§11.7.3	10/10	(100%)	5	5	5	5	5	5
`stack_full`	§11.7.4	11/11	(100%)	5	6	6	6	6	6
`stack_top`	§11.7.5	16/16	(100%)	6	10	10	10	10	10
`stack_push`	§11.7.6	43/43	(100%)	28	15	15	15	14	15
`stack_pop`	§11.7.7	32/32	(100%)	20	12	12	12	12	12

Table A.23.: Results for `Stack` functions

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`stack_size_wd`	§11.7.2	12/12	(100%)	8	4	4	4	4	4
`stack_empty_wd`	§11.7.3	12/12	(100%)	8	4	4	4	4	4
`stack_top_wd`	§11.7.5	12/12	(100%)	8	4	4	4	3	4
`stack_push_wd`	§11.7.6	15/15	(100%)	3	12	12	11	8	12
`stack_pop_wd`	§11.7.7	12/12	(100%)	6	6	6	6	5	6

Table A.24.: Results for the well-definition of the `Stack` functions

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	EP
`axiom_size_of_init`	§11.8.1	15/15	(100%)	11	4	4	4	4	4
`axiom_size_of_push`	§11.8.2	12/12	(100%)	9	3	3	3	3	3
`axiom_top_of_push`	§11.8.2	11/11	(100%)	8	3	3	3	3	3
`axiom_size_of_pop`	§11.8.3	11/11	(100%)	8	3	3	3	3	3
`axiom_pop_of_push`	§11.8.3	10/10	(100%)	6	4	4	4	4	4
`axiom_push_of_pop_top`	§11.8.3	15/15	(100%)	9	6	6	6	6	6

Table A.25.: 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 see Page 3.

The version numbers of this document are related to the versioning of Frama-C [3]. 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[48] 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 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.2. 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

[48] See `http://en.wikipedia.org/wiki/Atomic_number`

B.3. New in Version 16.1.0 (Sulfur, December 2017)

- special thanks to Aaron Rocha who provided various improvements for Chapters 3, 4, and 5
- 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 9 with more algorithms related to sorting
  - add algorithm `is_sorted` including predicate `WeaklySorted`
    * add lemma `SortedImpliesWeaklySorted`
    * add lemma `WeaklySortedImpliesSorted`
  - 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 `SortedUpperBound`
  - generalize lemma `UnchangedSection`
  - refactor lemma `HeapBounds` into `C_Division_2`

B.4. New in Version 15.1.2 (Phosphorus, October 2017)

- fix several typos reported by `seniorlackey@github` (thanks a lot!)
- add Chapter 10 on classic sorting algorithms which comprises
  - `selection_sort` including lemma `SwapImpliesMultisetUnchanged`

- insertion_sort including lemmas
    - EqualRangesPreservesSorted
    - RotatePreservesStrictLowerBound
    - RotateImpliesMultisetUnchanged
    - 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 HeapMaximumChild
    - provide the auxiliary function maximum_heap_child
    - 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.5. 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 8.4

- fix a typo in Figure 8.39
- 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.6. 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 `MultisetAddDistinct`
  - add predicate `MultisetMinus` and lemma `MultisetMinusDistinct`
  - add predicate `MultisetRetain` and lemma `MultisetPushHeapRetain`
  - provide an additional version of predicate `MultisetRetainRest`
  - and lemma `MultisetPushHeapClosure`

B.7. 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 (see Section A.3)
- 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 UnchangedShift and add more assertions to remove in order to simplify the task for automatic theorem provers

B.8. 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.9. 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 CountShift
- 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.10. 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 8. This new description of heap algorithms is based to a large extend on the bachelor thesis of one of the authors [18].

- 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 `CountBounds` 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`, `LowerBoundShift`, `StrictLowerBoundShift`, `UpperBoundShift` and `StrictUpperBoundShift` 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.11. 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.[49] 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 `AccumulateDefaultRead`
  - simplify implementation of `inner_product`
    - add predicates `ProductBounds` and `InnerProductBounds`
  - enable automatic verification of `partial_sum`

[49] See `https://bts.frama-c.com/view.php?id=2154`

    * add lemmas PartialSumSection, PartialSumUnchanged, PartialSumStep, 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 AdjacentDifferenceStep and AdjacentDifferenceSection 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 StackPushEqual to enable the automatic verification of the well-definition of stack_push

B.12. New in Version 11.1.1 (Sodium, June 2015)

- add Chapter on numeric algorithms

  - move iota algorithm to numeric algorithms (Section 7.1)

  - add accumulate algorithm (Section 7.2)

  - add inner_product algorithm (Section 7.3)

  - add partial_sum algorithm (Section 7.4)

  - add adjacent_difference algorithm (Section 7.5)

B.13. 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.14. 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.15. 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.16. 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.17. 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 [21] and now has a much better verification rate.

B.18. 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.19. 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.20. 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.21. New in Version 6.1.0 (Carbon, not published)

- changed definition of `Stack`
- renamed `reset_stack` to `init_stack`

B.22. 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.23. 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.24. 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.25. 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.26. New in Version 4.2.0 (Beryllium, January 2010)

- complete rewrite of pre-release
- adaption to Frama-C Beryllium 2 release

Bibliography

[1] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html, 2018.

[2] WP Plug-in. http://frama-c.com/wp.html.

[3] Frama-C Software Analyzers. http://frama-c.com, 2018.

[4] CEA LIST, Laboratory of Applied Research on Software-Intensive Technologies. http://www-list.cea.fr/gb/index_gb.htm.

[5] INRIA-Saclay, French National Institute for Research in Computer Science and Control . http://www.inria.fr/saclay/.

[6] LRI, Laboratory for Computer Science at Université Paris-Sud. http://www.lri.fr/.

[7] Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). http://www.fokus.fraunhofer.de.

[8] Virgile Prevosto. ACSL Mini-Tutorial. http://frama-c.com/download/acsl-tutorial.pdf.

[9] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL 1.13 Implementation in Chlorine-20180501. http://frama-c.com/download/acsl-implementation-Chlorine-20180501.pdf, May 2018.

[10] 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.

[11] Programming languages – C, Committee Draft. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf, 2009.

[12] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[13] 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.

[14] Lawrence Crowl and Thorsten Ottosen. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf, 2013. publicly available draft of C++ 14 standard.

[15] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, Jun 1977.

[16] Lincoln E. Moses and Robert V. Oakford. *Tables of Randon Permutations*. Stanford University Press, 1963.

[17] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate texts in computer science. Springer, New York, 1997.

[18] Timon Lapawczyk. Formale Verifikation von Heap-Algorithmen mit Frama-C. bachelor thesis, Humboldt-Universität zu Berlin, July 2016.

[19] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[20] 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.

[21] 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`.

[22] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. `http://farside.ph.utexas.edu/euclid.html`, Austin/TX, 2008.

[23] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[24] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[25] Why – Where Programs Meet Provers. `http://why3.lri.fr`, 2018.

[26] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. The Alt-Ergo SMT Solver. `http://alt-ergo.lri.fr`, 2018.

[27] Clark Barrett and Cesare Tinelli. Homepage of CVC4. `http://cvc4.cs.stanford.edu/web/`, 2018.

[28] Clark Barrett and Cesare Tinelli. Homepage of CVC3. `http://www.cs.nyu.edu/acsys/cvc3/`, 2010.

[29] Microsoft Research. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`, 2018.

[30] Stephan Schulz. The E Theorem Prover. `https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`, 2018.

[31] The Coq Consortium. The Coq Proof Assistant. `https://coq.inria.fr`, 2018.