ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verifying Go's Standard Library

Practical Work Report

Adrian Jenny

Tuesday 31st January, 2023

Advisors: Prof. Dr. Peter Müller, João Pereira
Department of Computer Science, ETH Zürich

Abstract

Program verification aims to reliably prove the correctness of a program according to the supplied specification. The Viper verification infrastructure offers various frontends for popular programming languages. The frontend for the Go programming language is called Gobra. One of the limitations of program verification is that the complete program, including library functions, has to be specified and verified in order for the result to be provably correct. Currently, Gobra only has specifications for a handful of the packages in the Go standard library, none of which are completely verified.
In this project, we, therefore, work on specifying and verifying further parts of Go's standard library. We derive criteria to help us choose which packages to work on. For parts of these packages, we provide specifications and can prove memory safety using Gobra. Additionally, we present an assessment of Gobra's usability. Based on our analyses, we report favourable verification performance across the chosen packages. Moreover, comparing our specified versions of two representative packages compared to the source code reveals that the annotation overhead for the specifications remained below a factor of 2.4.

Acknowledgements

I thank my project supervisor João Pereira. João showed great passion and constantly provided me with valuable feedback and took the time to expound upon more detailed background information on whatever issues I was facing at the time. His flexibility, support and intricate knowledge of the subject matter were of great assistance. His feedback and comments helped me learn a lot and grow my competencies in many areas besides program verification.

I further thank Prof. Dr. Peter Müller and the whole of the Programming Methodology Group at ETH for allowing me to work on this project.

Contents

Chapter 1

Introduction

Increasingly many components of our modernized world are guided by software-controlled systems. Consequently, having assurances on the functional correctness of these systems becomes ever more desirable. Depending on the application domain errors in implementations can be a threat to safety and cause enormous monetary losses. A prime example of a small software error and its disproportionate consequence is the crash of the Mars Climate Orbiter in 1999 [1]. For gauging the correctness of their programs, software developers typically use some sort of testing. This usually involves testing the implementation on sample inputs. Testing often forms an integral part or even the basis of many software development processes like test-driven development (TDD). However, testing alone is not enough to prove the absence of bugs. Sound software verification techniques and verification engines are used to definitively prove the absence of bugs on the level of the implementation. The Programming Methodology Group at ETH developed such a modern verification infrastructure called Viper [12]. Viper employs several different frontends in order to provide a verification toolchain to multiple languages. Gobra [15] is the frontend for the Go programming language and already supports many of Go's most important features. Thus, one can already provide specifications for Go programs and use Gobra to check whether a program behaves according to its specification. However, one key deficiency of verification is that it cannot check whether calls to unspecified library functions are guaranteed to succeed. Even if library functions are specified, if their implementations are not verified according to the specifications, then these specifications are nothing more than unchecked assumptions. This is especially problematic since this is also the case for Go's very own standard library [3] whose features are used in some way by most Go programs. Moreover, as can be seen on GitHub's issue tracker even the standard library ships with bugs [4,7]. Thus, blindly trusting the standard library implementation is misguided. This project's goal is to provide

specifications to a chosen subset of Go's standard library and verify them using Gobra. Typically, there is considerable effort involved when deriving specifications and applying verification. However, barring any large-scale design changes, this would be a one-time investment.

This project focuses on the following goals.

(1) *Applying verification*:
We derive criteria to select a suitable subset of the Go standard library and provide specifications for this subset's methods that can be verified using Gobra.

(2) *Providing a clean abstraction*
We use predicates to abstract over data structures' invariants end employ `ghost pure` functions to not reveal implementation details of public methods.

(3) *Issues and tests*
We want to test whether our specified version of the standard library is sufficient to catch known bugs. Furthermore, for available test cases, we want to check whether our specification is sufficient to guarantee whether a particular test will be successful.

(4) *Evaluating Gobra's usability*
There are continuous efforts to extend Gobra to be able to reason about all of Go's features. We wish to assess when Gobra's capabilities are lacking when it comes to verifying the selected functions. Moreover, using any kind of verification tool often requires considerable annotation overhead and workarounds in order to help the underlying theorem prover. We wish to identify cases where this overhead becomes impractical.

Chapter 2

Background

2.1 Go

Go [2] is a statically typed and compiled programming language that is already very well established in the industry and still continually gaining in popularity. It was designed at Google and as such is notable for its strong built-in support for building scalable and concurrent applications. Furthermore, Go employs a combination of nominal and structural typing.

2.1.1 The Standard Library

Go offers a large standard library [3] of functions and methods that are used in varying contexts. Apart from the definitions of built-in types and low-level platform-specific implementations the standard library contains many packages with high-level features and abstractions. Only a very small selection of which is listed here (in no particular order):

- `math`: This package offers optimized implementations of many commonly used mathematical functions.
- `container/list`, `container/ring`, `container/heap`: These packages define some fundamental and broadly used data structures. They achieve this by providing type definitions and methods to insert and remove elements from the data structure. Depending on the data structure, the corresponding package also provides functionality to rearrange or concatenate multiple instances of the data structure.
- `net`: The `net` package provides functionality to deal with network connections . This includes name resolution, TCP and UDP network sockets as well as network I/O.
- `database/sql`: This package defines the interfaces to be implemented by the drivers of a given relational database.

- `crypto`: The package `crypto` itself mainly defines some commonly used cryptographic constants and interfaces. These interfaces are then implemented in the packages defining more specific cryptographic primitives, e.g. `aes`, `ecdsa`, or `sha256`.
- `sort`: The `sort` package provides multiple different efficient sorting algorithms. It also defines interfaces that user-defined types can implement to make them sortable.
- `bytes`: The `bytes` package provides a multitude of functions for manipulating slices of bytes. Furthermore, the package defines the type `Buffer` whose methods in turn depend on the aforementioned bytes functionalities.
- `io`: The `io` package contains many well-documented interface definitions for `Reader` and `Writer` types.

Note: For convenience, we will shorten `container/list` *and* `container/ring` *to* `list` *and* `ring` *throughout the rest of this document.*

This list encompasses only a small selection of the many different packages provided by the standard library. Consequently, Go programmers make heavy use of the functionality provided by the standard library and rely on it being implemented correctly.

2.2 Testing and Verification

Apart from other design decisions, programmers mainly aim to produce code that is both reliable and correct. Several methodologies have been developed to help programmers achieve these goals. This section serves to highlight the differences between Testing and Program Verification.

2.2.1 Testing

Generally, whenever programmers implement a new feature or functional unit they also provide tests. These tests are often a sample set of predetermined inputs for the program along with a set of predetermined and approved outputs. A correctly written program will always produce the desired outputs on these given inputs. These test cases are usually chosen in such a way as to cover as many of the code's behaviours as reasonably possible. Tests are an integral part of any large software project and their inclusion in development pipelines helps to identify any breaking changes. Despite all the uses of testing and its importance in software projects it also has its limitations. Even for moderately simple and small projects testing cannot be used to give guarantees on all possible inputs. Most software projects are too complex to design test cases for all possible scenarios. Thus, testing alone is not sufficient to truly prove the absence of errors.

2.2.2 Program Verification

Program verification is an approach that is used to prove that a piece of code (e.g. a function) behaves according to its specification. Through the use of verification tools programmers can annotate their code with such a specification. Typically, a specification is of the form of a pre- and postcondition for a given function. The precondition describes the constraints on the inputs and the current state that need to be fulfilled to call that function. A verification engine combines the constraints given by the precondition and, together with the function implementation, tries to prove that the constraints described in the postcondition are satisfied. A function can call other functions and as long as it was possible to establish the precondition of the called function, the verification engine gets to assume the called function's postcondition in conjunction with its constraints. Thus, program verification can be done in a modular fashion. Correctly implemented verification engines provide proven soundness guarantees, i.e. a program that is not guaranteed to fulfill its specification will never be reported as fulfilling the specification. This is a clear benefit over just using test cases in that it is now possible to have guarantees as to the behaviour of a program. However, similar to the quality requirements on test cases, program verification is only useful if the specification expresses the desired or undesired properties. Deriving sensible specifications and verifying a program can often induce a significant overhead in developer effort. In fact, the overhead effort of providing meaningful function specifications and verifying a whole system is still large enough such that often only very critically important software projects get formally verified. Researchers have shown how building a verified system is a significant up-front investment in developer time [11]. In their example, they show how the additional annotation overhead is a factor of 3.6 times larger (measured in source lines of code) than the underlying implementation itself. Another limitation is that developers still need to trust the correctness of other parts of the system, including the compiler implementation, the hardware and unverified libraries.

2.3 Gobra

Gobra [15] is a modular verifier for the Go programming language. It is actively developed by the Programming Methodology Group at ETH. This section serves to give a short overview of Gobra's workings.

While Gobra still is a research project in development, it already supports a wide array of Go's more advanced features. These include support for interfaces and different methods of achieving concurrency. While the overarching goal of verification may be to prove the functional correctness of a program, this often relies on also proving memory safety and termination for all valid

inputs. Gobra's mechanism for proving memory safety is based on separation logic [14] and works by augmenting specifications with requirements regarding access permissions to given heap locations. This helps to ensure that functions only access valid memory locations and the permissions allow proving the absence of race conditions. Furthermore, Gobra assertions can also be parameterized and wrapped up in a predicate. These predicates can then be used to abstract over the permissions and consistency properties of a (possibly recursive) data structure.
For programs containing loops and recursive calls, termination can be enforced by providing variants. A variant is an expression whose value is guaranteed to decrease with every loop iteration or recursive call.

Internally, Gobra encodes annotated Go programs into programs written in the Viper verification language [12] and is verified by the Viper tooling based on functionality provided by the Z3 theorem prover [16].

Chapter 3

Methodology

The overall goal of this work is to use Gobra to verify a suitable subset of Go's standard library. While we recognize that verification can be a costly approach, the rationale behind this work is that having a properly verified standard library is very important. Given the large user base of Go, bugs in the standard library can have a far-reaching impact. Investing the effort to verify the standard library will therefore provide value to virtually all users of the language. Unless the used standard library functions are themselves verified, Go programs that rely on the standard library cannot be guaranteed to be correct.

3.1 Choice of subset

Because verification of the whole standard library would take a very long time, the scope of this project is a small subset of the standard library. There are multiple criteria to consider in the choice of this subset:

- *Impact:* Verification might either succeed or fail. In case of success, we can provide guarantees that the function is indeed correct according to its specification. In case of failure, we know that the function might have a bug. For this project to be impactful, it makes sense to verify functions and packages that are heavily used by other packages in the standard library and by users of the standard library.
- *Dependencies:* Like other packages, packages in the Go standard library may themselves also depend on the standard library. Starting with verifying packages with few or no dependencies reduces the need to re-adjust the specifications of packages with relatively more dependencies. This is especially true in the likely case that the specifications of the former need some fine-tuning. It, therefore, makes sense to be verifying packages with few or no dependencies first and work bottom-up.

However, when it comes to interfaces we might have to reverse our approach where applicable. While interface definitions can be seen as dependencies of their implementations, it still makes sense to first consider some of these implementations to get a sense of the intended shared behaviour.

- *Platform independence:* We will not work on any platform-specific packages. For one, we would quickly run into limitations of Gobra when dealing with inlined C or assembly code. Furthermore, as per the first criterion, our findings wouldn't be as broadly applicable.
- *Adequacy to program verification:* A subset that encompasses algorithmically interesting functions and data structures provides a good target for the application of verification techniques. Not only are the more complex algorithms the most likely to contain bugs, but they also offer the additional benefit of allowing us to test out the limits of Gobra's practicability in real-world use cases.

We chose parts of the following packages to verify and present our reasoning for this choice based on the above criteria. The contents of these packages are briefly described in the section on the standard library in 2.1.1.

- `list`: A list is a fundamental data structure. Since the list data structure defined in this package is doubly linked and circular, it has to fulfill several non-trivial invariants for all operations to work. Of particular interest is that there are already known issues [4] with the implementation and it will be interesting to see whether our specification is sufficient to catch these bugs.
- `ring`: The data structure defined in `ring` is very similar to the list above in that it is also doubly linked and circular. However, a notable difference is the lack of any clear "owner" object of the ring.
- `sort`: Sorting is an essential operation. Furthermore, the `sort` package naturally offers a number of algorithmically interesting implementations.

3.2 Approach

3.2.1 Bottom-up

The general idea is to follow a bottom-up approach. This applies to the overall structure concerning the dependencies between packages (as mentioned in 3.1), as well as with regard to the dependencies between functions within a particular package. Within a given package we first verify the functional building blocks lowest in the respective dependency hierarchy. The public-facing methods and functions that are built using these building blocks get

verified at a later stage. The advantage of this approach is that we can reduce the assumptions on the specifications for lower-level functions. Furthermore, we can gather all constraints on the pre- and postconditions from these lower-level functions first and use them to build up and establish a clean public-facing interface.

3.2.2 Clean abstraction

The specifications for public-facing functions should not reveal information about the internal representation like the existence of private fields. For a public method's contract to be explainable purely in terms of publicly available members, we plan to introduce additional `ghost pure` functions. Any invariants of the underlying data structures should be abstracted under a predicate whenever possible.

3.2.3 Testing

To check whether our specifications are applicable, we augment available test cases with our annotation and assertions to see whether our specifications suffice to cover the same behaviour as the test cases. That is, using our specification on a correct test case should guarantee that the test is successful. Furthermore, for the functions we can specify, we want to track down whether there are any open issues in the Go issue tracker[1] and see whether they would have been caught if verification was applied. For example, there is a known issue with a leaky abstraction in the `list` package [4].

[1] https://github.com/golang/go/issues

Chapter 4

Implementation

This chapter highlights some of the details and considerations that went into deriving specifications for chosen parts of the packages. To distinguish comments from the original developers from our newly added comments, we generally included a `#` sign at the start of our comments. E.g.

```
1 //  This is a comment left by the original developers.
2 //# This is a comment left by us pertinent to the specification.
```

Moreover, for better readability, we chose not to include the triggers in quantified expressions in our code excerpts.

4.1 List

In this section, we will describe some of our efforts that went into specifying and verifying parts of the `list` package as well as highlight some of the challenges. The `list` package describes a circular doubly-linked list via various defined types and methods. The two types that are used to implement this structure are `List` and `Element`. A full list of the package's methods that operate on these types is shown in the package overview on Go's website[1].

4.1.1 Types

Listing 1 shows the definition of the `Element` type. Since we are dealing with a doubly-linked list, every list element has a pointer `prev` and a pointer `next` pointing to the previous and next list element respectively. Storing values in the list works via the `Value` field of `any` type. The `list` field points to the list `l` that currently owns the element `e`. The developers of this package rely on

[1] https://pkg.go.dev/container/list

```
1  type Element struct {
2      // Next and previous pointers in the doubly-linked list of elements.
3      // To simplify the implementation, internally a list l is implemented
4      // as a ring, such that &l.root is both the next element of the last
5      // list element (l.Back()) and the previous element of the first list
6      // element (l.Front()).
7      next, prev *Element

8      // The list to which this element belongs.
9      list *List
10     // The value stored with this element.
11     Value any
12 }
```

Listing 1: Definition of the `Element` type from the standard library's `list` package.

this field being up-to-date for all valid elements in scope. As such, they often perform the comparison `e.list == l` to check whether element `e` is in list `l`.

```
1  type List struct {
2      root Element     // sentinel list element, only &root, root.prev,
3                       // and root.next are used
4      lenT int // current list length excluding (this) sentinel element
5  }
```

Listing 2: Definition of the `List` type from the standard library's `list` package.

The `List` type shown in listing 2 contains the `root` field of type `Element`. This field only serves to anchor the circular list around a fixed element and its `Value` field is never intended to be used. This root element also does not count toward the length of the list. To be able to immediately report the length of the list, the `List` type also features a length field called `lenT` of type `int`.

4.1.2 Invariants of a list

We formalized the invariants of List in the predicate `Mem` shown in Listing 3. This predicate abstracts over the permissions and consistency criteria of a list `l` and its contained elements. The predicate has a list `l` as its receiver and accepts arguments `s` and `isInit` of types `set[*Element]` and `bool` respectively. The set `s` must contain all the elements of the list including the root whereas the argument `isInit` indicates whether the list was already initialized with the `Init` method. As shown in lines 2 - 6, holding this predicate includes permissions to the underlying list `l` and the fields of all its elements. Line 7 enforces the relationship between the number of elements in the set `s` and the `lenT` field of the list. An uninitialized list only contains the root element which points to no other elements as shown in lines 9 - 13. If `isInit` is

Note: The length field was originally called `len` but due to an issue in Gobra [8] we had to avoid field names that could be mistaken as Gobra keywords.

```
1  pred (l *List) Mem(ghost s set[*Element], ghost isInit bool) {
2      acc(&l.lenT) &&
3      &l.root in s &&
4      (forall i *Element :: i in s ==> (
5          acc(&i.next) && acc(&i.prev) &&
6          acc(&i.list) && acc(&i.Value))) &&
7      l.lenT == len(s)-1 &&
8      ((l.root.next == nil || l.root.prev == nil) ==> !isInit) &&
9      (!isInit ==> (
10         len(s) == 1 &&
11         s == set[*Element]{&l.root} &&
12         l.root.next == nil &&
13         l.root.prev == nil)) &&
14     (isInit ==> (
15         len(s) >= 1 &&
16         (forall i *Element :: i in s ==> (
17                 i.next != nil && i.prev != nil &&
18                 (i != &l.root ==> i.list == l))) &&
19         (forall i, j *Element :: (i in s && i.next == j) ==> (
20                 j in s && j.prev == i && i.next.prev == i &&
21                 j.prev.next == j)) &&
22         (forall i, j *Element :: (i in s && i.prev == j) ==> (
23                 j in s && j.next == i && i.prev.next == i &&
24                 j.next.prev == j))))
25 }
```

Listing 3: Definition of the memory predicate `Mem` for a list.

true then the predicate enforces stronger constraints on the structure of the list. The quantified expressions on lines 14 - 24 enforce the doubly-linked structure of an initialized list as well as the fact that every element's `list` pointer points to `l`. Note that the predicate is not equipped to enforce that all elements form a single doubly-linked cycle. However, the elements' relative positions in the list are mentioned in the methods' postconditions. In general, every correctly implemented method operating on an initialized list in a consistent state should return a consistent list. Thus, every public method assumes the predicate `Mem` and must preserves it.

4.1.3 Initialization

Listing 4 shows the specified version of the `Init` method. Note that the precondition requires the `Mem` predicate but passes the `isInit` parameter to it that the `Init` method received as an argument. This is because the `Init` method is not only used for the first initialization of a list but also to reset it (i.e. remove pointers to any elements and set `lenT` to 0). It is noteworthy to point out that this method does not go through and change every element's `list` pointer to `nil`. It is a known issue that since an element's `list` pointer is often used to check for membership, it could thus be that if a list is reset and a client still holds a pointer to one of the list's earlier elements, then that element could get falsely reported as belonging to the list and lead to an inconsistent state [4].

```
1  // Init initializes or clears list l.
2  requires l.Mem(elems, isInit)
3  ensures  res == l
4  ensures  l.Mem(set[*Element]{&l.root}, true)
5  ensures  isInit ==>
6      (forall i *Element :: (i in elems && i != &l.root) ==>
7      (acc(&i.next) && acc(&i.prev) && acc(&i.list) && acc(&i.Value)))
8  ensures  l.Len(set[*Element]{&l.root}, true) == 0
9  decreases
10 func (l *List) Init(ghost elems set[*Element], ghost isInit bool)
11     (res *List) {
12         unfold l.Mem(elems, isInit)
13         l.root.next = &l.root
14         l.root.prev = &l.root
15         l.lenT = 0
16         fold l.Mem(set[*Element]{&l.root}, true)
17         return l
18 }
```

Listing 4: Specified `Init` method

4.1.4 Update primitives

The methods `insert`, `move` and `remove` whose specified versions are shown in listings 5, 7 and 6 are used internally in the package to update the list's elements and their position. Notably, in `insert`'s postcondition we manage to verify the inserted element's position relative to its neighbors via the `ghost pure` method `comesBefore`.

```
1  // insert inserts e after at, increments l.lenT, and returns e.
2  requires l.Mem(elems, true)
3  requires acc(e)
4  requires at in elems
5  requires !(e in elems)
6  ensures  l.Mem(elems union set[*Element]{e}, true)
7  ensures  l.Len(elems union set[*Element]{e}, true) ==
8              1 + old(l.Len(elems, true))
9  ensures  at.comesBefore(e, elems union set[*Element]{e}, l)
10 ensures  e.comesBefore(old(at.nextPure(elems, l)),
11             elems union set[*Element]{e}, l)
12 ensures  res == e && res != nil && res != &l.root
13 decreases
14 func (l *List) insert(e, at *Element, ghost elems set[*Element])
15     (res *Element) {
16         unfold l.Mem(elems, true)
17         e.prev = at
18         e.next = at.next
19         e.prev.next = e
20         e.next.prev = e
21         e.list = l
22         l.lenT++
23         fold l.Mem(elems union set[*Element]{e}, true)
24         return e
25 }
```

Listing 5: Specified `insert` method

Contrary to this, in `remove`'s postcondition we do not explicitly enforce that the earlier neighbors of the removed element `e` are now neighboring each

```
requires l.Mem(elems, true)
requires e in elems
requires e != &l.root
ensures  l.Mem((elems setminus (set[*Element]{e})), true)
ensures  l.Len((elems setminus (set[*Element]{e})), true) ==
            old(l.Len(elems, true)) - 1
ensures  acc(e) && e.list == nil
decreases
func (l *List) remove(e *Element, ghost elems set[*Element]) {
        unfold l.Mem(elems, true)
        e.prev.next = e.next
        e.next.prev = e.prev
        e.next = nil // avoid memory leaks
        e.prev = nil // avoid memory leaks
        e.list = nil
        l.lenT--
        fold l.Mem((elems setminus (set[*Element]{e})), true)
}
```

Listing 6: Specified `remove` method

other. This is primarily because the invariants do not strictly enforce all elements to form a single cycle. Thus Gobra cannot rule out that `e` may only have been in a cycle with itself. Consequently, in `move`'s postcondition we also only reason about the neighborhood relation at the moved element's new position. A possible fix for this issue could be to enforce reachability between all elements. The problem is discussed in more detail in 5.4.

```
// move moves e to next to at.
requires l.Mem(elems, true)
requires e in elems
requires at in elems
ensures  l.Mem(elems, true)
ensures  (e != at ==> (unfolding l.Mem(elems, true) in
            (at.next == e && e.prev == at)))
ensures  (e != at && old(at.nextPure(elems, l) != e)) ==>
            at.comesBefore(e, elems, l)
ensures  (e != at && old(at.nextPure(elems, l) != e)) ==>
            e.comesBefore(old(at.nextPure(elems, l)), elems, l)
decreases
func (l *List) move(e, at *Element, ghost elems set[*Element]) {
        if e == at {
                return
        }
        unfold l.Mem(elems, true)
        //# remove e
        e.prev.next = e.next
        e.next.prev = e.prev
        //# insert e after at
        e.prev = at
        e.next = at.next
        e.prev.next = e
        e.next.prev = e
        fold l.Mem(elems, true)
}
```

Listing 7: Specified `move` method

4.1.5 List membership check

Instead of a linear traversals of the list to check for membership, the developers of this package prefer to use the condition `e.list == l` to check whether the element `e` is in list `l` as is shown in line 18 in listing 8. As already

```
1  // Remove removes e from l if e is an element of list l.
2  // It returns the element value e.Value.
3  // The element must not be nil.
4  requires e != nil
5  requires l.Mem(elems, true)
6  requires e != &l.root
7  //# The next two lines aim to establish: (e.list == l) IFF (e in elems)
8  requires !(e in elems) ==> (acc(e) && e.list != l)
9  requires unfolding l.Mem(elems, true) in (e.list == l) == (e in elems)
10 ensures  !(e in elems) ==> l.Mem(elems, true)
11 ensures  e in elems ==> l.Mem((elems setminus (set[*Element]{e})), true)
12 ensures  acc(e) && e.Value === res && (e in elems ==> e.list == nil)
13 decreases
14 func (l *List) Remove(e *Element, ghost elems set[*Element]) (res any) {
15     ghost if e in elems {
16         unfold l.Mem(elems, true)
17     }
18     if e.list == l {
19         // if e.list == l, l must have been initialized
20         // when e was inserted in l or l == nil
21         // (e is a zero Element) and l.remove will crash
22         fold l.Mem(elems, true)
23         l.remove(e , elems )
24     }
25     return e.Value
26 }
```

Listing 8: Specified `Remove` method

pointed out in 4.1.3, this membership check is buggy when a pointer to an element is leaked before the list was reset without clearing the element's `list` field. We thus wish to have an equivalence between an element's `list` field pointing to a list `l` and the fact that the element is contained in that list's set of elements in the `Mem` predicate. Concretely, given an element `e`, a list `l` and the corresponding predicate `l.Mem(elems, true)`, we need to establish

$$\texttt{e in elems} \iff \texttt{e.list == l} \tag{4.1}$$

With this restriction, we will never be able to observe the bug in verified code since calls to e.g. `Remove` are disallowed if the equivalence does not hold. This is further useful when gathering up constraints in branches in the code's control flow. In the above example of the `Remove` method, we enter the method not knowing whether the element `e` belongs to the list. The code does this check this via the if-statement `e.list == l`. The branch that then effectively removes the element from the list should only be entered if the element is in the list. Otherwise, the branch is not entered and it is

Note: We also require the list to be initialized, implying `l != nil`. This disallows the crash described by the developers in lines 19 - 21.

presumed the element was not part of the list. The predicate `l.Mem(elems, true)` already provides the fact that the `list` fields of the elements in the `elems` set all point to `l`. The equivalence 4.1 ensures that Gobra can infer that `e in elems` if the execution enters the branch where the element is to be removed. Establishing this equivalence happens in lines 8 and 9 in the above example and required a workaround that might seem counter-intuitive at first glance. While the equivalence itself is stated on line 9, the overall problem lies in the fact that it is unclear where the permissions to `e`'s `list` field should come from. While we use an `unfolding` of the predicate in line 9, this only yields us the permissions to this field if indeed `e in elems`. Therefore, if `e` were not in the `elems` set, the unfolding of the predicate would not give us the required permissions for the field access. In line 8 we, therefore, request permissions to `e`'s fields via `acc(e)` but only in the case where `e` is not in the `elems` set. In that case, the unfolding of the predicate in line 9 is without effect. This kind of membership check is used in various other methods throughout this package. We thus repeat this pattern in the preconditions of the methods where it is necessary. Specifically, this pattern had to be applied for the methods `InsertBefore`, `InsertAfter`, `MoveToFront`, `MoveToBack`, `MoveBefore` and `MoveAfter`. While this precondition constitutes an abstraction leak, i.e. it reveals how a membership check is done, it is nonetheless necessary in order to avoid running into a bug.

4.2 Ring

In this section, we describe our derived specifications for the `ring` package. The data structure of a ring is conceptually very similar to the doubly-linked circular list described in the `list` package with the main difference being that there is no clear "owner" element. A full list of the package's methods that operate on these types is shown in the package overview on Go's website[2].

4.2.1 Type

```
1  type Ring struct {
2      next, prev *Ring
3      Value any // for use by client; untouched by this library
4  }
```

Listing 9: Definition of the `Ring` type from the standard library's `ring` package.

In this package, only the `Ring` type gets defined as shown in listing 9. The package developers use the word "ring" interchangeably in their comments. They use "ring" to refer to both a single object instance of the `Ring` type

[2]https://pkg.go.dev/container/ring

as well as the whole data structure. We will differentiate these two cases and use "ring element" and "ring structure" respectively. A ring structure is composed of doubly-linked and cyclically connected ring elements via the `Ring` type's `prev` and `next` fields. Contrary to a list, we have no clear beginning or end to the ring structure, nor do we have an obvious "owner" element of the structure.

4.2.2 Invariants of a ring structure

The invariants of a ring structure are formalized in the `Mem` predicate shown in listing 10. This predicate is similar in many ways to the `Mem` predicate of a list. It abstracts over the permissions of all ring elements in the ring structure and additionally provides some consistency guarantees about the structure if it can be considered initialized. The parameters `s` and `isInit`

```
1  pred (r *Ring) Mem(ghost s set[*Ring], ghost isInit bool) {
2      r != nil &&
3      r in s &&
4      (forall i *Ring :: i in s ==> (
5          acc(&i.next) && acc(&i.prev) && acc(&i.Value))) &&
6      ((r.next == nil || r.prev == nil) ==> !isInit) &&
7      (!isInit ==> (
8          s == set[*Ring]{r} &&
9          r.next == nil &&
10         r.prev == nil)) &&
11     (isInit ==> (
12         (forall i *Ring :: i in s ==> (
13             i.next != nil && i.prev != nil)) &&
14         (forall i, j *Ring :: ((i in s && i.next == j) ==> (
15             j in s && j.prev == i && i.next.prev == i &&
16             j.prev.next == j))) &&
17         (forall i, j *Ring :: ((i in s && i.prev == j) ==> (
18             j in s && j.next == i && i.prev.next == i &&
19             j.next.prev == j)))))
20 }
```

Listing 10: Definition of the memory predicate `Mem` of a ring structure

are of types `set[*Ring]` and `bool` respectively. Similar to before, `s` is the set of all ring elements in the ring structure whereas `isInit` serves to indicate whether the ring structure pointed to via the ring element `r` was initialized by the `init` method. The quantified expressions in lines 12 - 19 establish that every ring element has to be connected to another ring element and that these connections are doubly-linked. Apart from this, the predicate by itself does not enforce that the elements in `s` form one single ring structure. Similar to `list`, this could be enforced by requiring reachability between all ring elements as described in more detail in 5.4.

4.2.3 Next - Prev

The package contains the two similar `Next` and `Prev` methods of which the former is shown in listing 11. We use this example to illustrate a few key

```
1  // Next returns the next ring element. r must not be empty.
2  //# We use 'owner' here to make calls from different receivers
3  //# in the same ring structure work (see e.g. Link)
4  requires r != nil
5  requires owner != r ==> r in elems
6  requires !isInit ==> owner.Mem(elems, false)
7  requires isInit ==> acc(owner.Mem(elems, true), ReadPerm)
8  ensures  !isInit ==> (owner.Mem(set[*Ring]{r}, true) && res == r)
9  ensures  isInit ==> (acc(owner.Mem(elems, true), ReadPerm) && res in elems)
10 ensures  isInit ==> unfolding acc(owner.Mem(elems, true), ReadPerm) in
11             res == r.next
12 decreases
13 func (r *Ring) Next(ghost elems set[*Ring], ghost owner *Ring,
14     ghost isInit bool) (res *Ring) {
15         ghost if isInit {
16                 unfold acc(owner.Mem(elems, true), ReadPerm)
17         } else {
18                 unfold owner.Mem(elems, false)
19         }
20         if r.next == nil {
21                 assert !isInit //# here we know !isInit
22                 fold owner.Mem(elems, false)
23                 return r.init(elems, false)
24         }
25         res = r.next
26         fold acc(owner.Mem(elems, true), ReadPerm)
27 }
```

Listing 11: Specified `Next` method

observations. Firstly, the developers' comment points out that the receiver `r` must not be empty. As described in further documentation, this means simply that `r` must not be `nil`. Given the above, this method then uses the check `r.next == nil` to establish whether the ring structure is initialized. Secondly, `Next` as well as `Prev` call `init` if it was determined that the ring structure was not yet initialized. This prevents us from declaring these methods as `pure`. However, both these methods are called on initialized ring structures from various other points in the code. To clearly indicate that the heap-locations hidden in the `Mem` predicate were not changed we, therefore, differentiate the level of permissions based on whether the ring structure is already initialized, i.e., we only need write permissions to the heap locations if `isInit` is false.

4.2.4 New

Listing 12 shows a specified and adapted version of the original `New` method present in the `ring` package. The purpose of `New` is to create a new ring structure with `n` elements. Given `n > 0`, the original version started with the first ring element `r`, then iterated `n - 1` times through a loop where it attached a newly created ring element. After the loop, the endpoints of this

```
1  // New creates a ring of n elements.
2  ensures n <= 0 ==> res == nil
3  ensures n > 0 ==> (len(elems) == n && res.Mem(elems, true))
4  func New(n int) (res *Ring, ghost elems set[*Ring]) {
5      if n <= 0 {
6          res = nil
7          elems = set[*Ring]{}
8          return
9      }
10     r := new(Ring)
11     p := r
12     //# Changed the implementation to already form a cycle here so that
13     //# we can use the memory predicate in the invariant and have an
14     //# easier time establishing it in the end.
15     r.next = r
16     r.prev = r
17     elems = set[*Ring]{r}
18     fold r.Mem(elems, true)

19     invariant r.Mem(elems, true)
20     invariant len(elems) == i
21     invariant p in elems
22     invariant 1 <= i && i <= n
23     decreases n - i
24     for i := 1; i < n; i++ {
25         unfold r.Mem(elems, true)
26         q := &Ring{}
27         q.prev = p
28         q.next = p.next
29         q.prev.next = q
30         q.next.prev = q
31         p = q
32         elems = elems union set[*Ring]{p}
33         fold r.Mem(elems, true)
34     }
35     res = r
36 }
```

Listing 12: Specified and adapted `New` method

chain would be connected. This original version ultimately made it difficult to derive suitable loop invariants such that the invariants of an initialized list could be proved in the end. This adapted version takes a different approach. A consistent ring structure always gets established between iterations of the loop. This way, we can use the `Mem` predicate for initialized ring structures as our loop invariant and do not have to fold it at the very end only from the gathered constraints. However, this adapted version of the method still passed all the tests in the `ring_test.go` file.

4.2.5 Link

The `Link` method accepts ring element `r` as its receiver and ring element `s` as its argument. Depending on whether these ring elements are in the same ring structure and how they relate to each other in their relative positions, the execution yields one or two ring structures. A subset of the possible behaviours of the `Link` method is illustrated in figure 4.1. A partly specified version of `Link` is shown in listing 13.

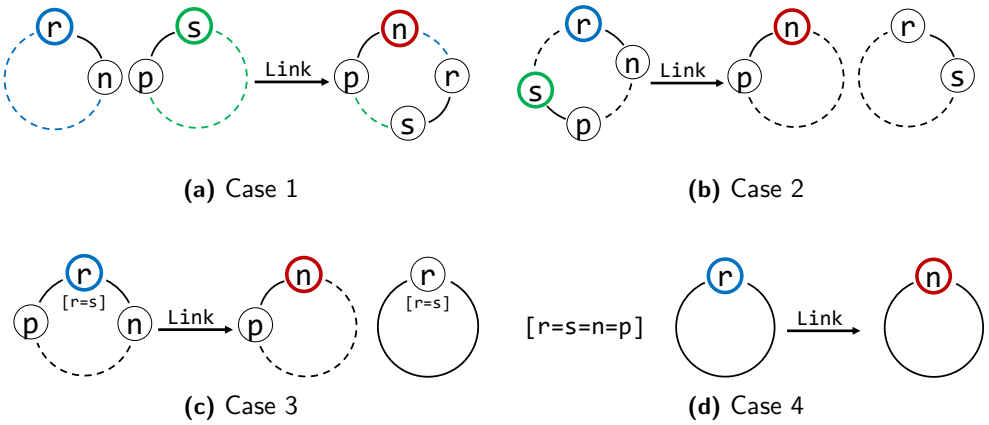


Figure 4.1: Subset of cases of `Link`'s behaviour

Note that in figure 4.1, `n` *is equal to* `r.next` *and* `p` *is equal to* `p.prev` *in the state before the method execution. The receiver of the method is circled in blue, the method argument (if not the same as the receiver) in green, and the returned pointer in red.*

We introduced the sets `elemsR` and `elemsS` as additional parameters. These sets contain the elements of the ring structures containing `r` and `s` respectively. Since we restrict the specification at the moment to only accept initialized ring structures, the return value will always be the original value of `r.next`.

- *Case 1:* The case depicted in figure 4.1a represents the case that is intuitively associated with the method name. The ring elements `r` and `s` belong to different ring structures. These ring structures get split up and recombined into a new ring structure now containing all elements of both `elemsR` and `elemsS`. Note that while the figure suggests there has to be more than one element in the initial ring structures, the above is still true even if any of the two or both of the structures only contained one element. We manage to successfully prove the `Mem` predicate for the resulting ring structure in line 33.

- *Case 2:* In this case, illustrated in figure 4.1b, the ring elements `r` and `s` belong to the same ring structure but are not equal. The idea here is that all elements between `r` and `s` in the `next` direction get ejected from the original ring structure. The two resulting chains are then closed to form two new ring structures. The method returns a reference to one of these ring structures via the returned ring element `n`. No specification was derived for this case since it was difficult to derive the subset of `elemsR` that contains all ring elements between `r` and `s`. There is a special sub-case when `r` and `s` were neighboring in the original ring structure. In that case, the structure remains unchanged and the method simply returns `n`.

```
1  requires r != nil
2  requires r.Mem(elemsR, true)
3  requires (s != nil && !(s in elemsR)) ==> s.Mem(elemsS, true)
4  requires (s != nil && !(s in elemsR)) ==> (
5      elemsR intersection elemsS) == set[*Ring]{}
6  requires (s != nil && s in elemsR) ==> elemsR == elemsS
7  ensures  res == old(unfolding r.Mem(elemsR, true) in r.next)
8  ensures  (s != nil && !(s in elemsR)) ==>
9      res.Mem((elemsR union elemsS), true)
10 ensures  (s != nil && s in elemsR && r == s && len(elemsR) == 1) ==>
11     res.Mem((elemsR), true)
12 decreases
13 func (r *Ring) Link(s *Ring, ghost elemsR set[*Ring],
14     ghost elemsS set[*Ring]) (res *Ring) {
15         n := r.Next( elemsR, r, true )
16         if s != nil {
17             ghost elemsX := (s in elemsR)?(elemsR):(elemsS)
18             ghost owner := (s in elemsR)?(r):(s)
19             p := s.Prev( elemsX, owner, true )
20             unfold r.Mem(elemsR, true)
21             ghost if !(s in elemsR) {
22                 unfold s.Mem(elemsS, true)
23             }
24             r.next = s
25             s.prev = r
26             n.prev = p
27             p.next = n
28             ghost if !(s in elemsR){
29                 assert n != p
30                 assert s != r
31                 assume forall i *Ring :: i in (elemsR union elemsS) ==>
32                     (i.next != i && i.prev != i)
33                 fold n.Mem((elemsR union elemsS), true)
34             } else {
35                 ghost if r == s {
36                     ghost if len(elemsR) == 1 {
37                         fold n.Mem(elemsR, true)
38                     }
39                     ghost if len(elemsR) > 1 {
40                         //# UNIMPLEMENTED Case 3
41                     }
42                 } else {
43                     //# UNIMPLEMENTED Case 2
44                 }
45             }
46         }
47         return n
48 }
```

Listing 13: Specified `Link` method

- *Case 3:* In the case shown in figure 4.1c, `r` and `s` are equal in a ring of multiple elements. `Link` simply removes `r` from the structure. We would like to prove `n.Mem((elemsR setminus (set[*Ring]r)), true)`. However, our abstraction is unsuitable for this purpose. Since the `Mem` predicate does not explicitly enforce that all ring elements form a single cycle, it could be that `r` was in a cycle with itself. Thus, we could have that `r == n` even though we know that `len(elemsR) > 1`
- *Case 4:* Figure 4.1d shows a scenario where `r == s` are the only ring element in a ring structure. No changes are made to the structure. We can prove `n.Mem(elemsR, true)` in line 37 and return `n`.

4.3 Sort

In this section, we describe the specifications we derived for parts of the `sort` package. Sort describes an interface called `Interface`. Types seeking to implement `Interface` must provide the methods `Len()`, `Less(i, j int)` and `Swap(i, j int)`. The package also offers some default implementations of `Interface` based on `int` and `float64` slices called `IntSlice` and `Float64Slice` respectively. Moreover, the package offers a sorting function called `pdqsort`. According to the developers, the implementation of `pdqsort` is based on pattern-defeating quicksort [13]. This function uses many subroutines such as `insertionSort`, `heapSort`, `partition` and `breakPatterns`.

4.3.1 Interface specification

We require implementations to abstract the underlying data structure into a `ghost` sequence. To be able to work with objects of varying types, we declare the type of the sequence as `seq[any]` and use it as a parameter in a predicate. An implementation of the predicate needs to map the values from the underlying data structure onto this sequence.

```
1  // Len is the number of elements in the collection.
2  preserves acc(Mem(elems), ReadPerm)
3  ensures   res == len(elems)
4  decreases
5  Len(ghost elems seq[any]) (res int)
```

Listing 14: Specification of `Len` in `sort.Interface`

The specification for `Len` shown in listing 14 simply describes that the result will be equal to the length of the sequence.

For the specification of `Less`, shown in listing 15, we additionally introduced the pure function `less_spec` as well as the auxiliary lemma `LemmaLessIsTransitive`. The pure function `less_spec` was introduced for us to be able to use it in specifications. Any already available implementations of `Less` might not follow Gobra's restrictions on the form of pure functions. Therefore, any implementations would need to duplicate `Less`'s behaviour in `less_spec` while following the restrictions. Furthermore, it is required for `Less` to form a transitive relation on the elements it compares. Therefore, transitivity has to be proven by the implementations of the interface that have to satisfy `LemmaLessIsTransitive`'s postcondition. The way the developers implemented the `IsSorted` method makes it clear that `Less` must be "strictly less" which is why we explicitly enforce that comparisons between the same element yield `false` in line 18.

```
1  // Less reports whether the element with index i
2  // must sort before the element with index j.
3  ghost
4  preserves acc(Mem(elems), ReadPerm)
5  ensures   forall a, b, c int :: (0 <= a && a < len(elems) &&
6      0 <= b && b < len(elems) && 0 <= c && c < len(elems) &&
7      less_spec(a, b, elems) && less_spec(b, c, elems)) ==> (
8          less_spec(a, c, elems))
9  ensures   forall a, b, c int :: (0 <= a && a < len(elems) &&
10     0 <= b && b < len(elems) && 0 <= c && c < len(elems) &&
11     !less_spec(a, b, elems) && !less_spec(b, c, elems)) ==> (
12         !less_spec(a, c, elems))
13 decreases
14 LemmaLessIsTransitive(ghost elems seq[any])

15 requires acc(Mem(elems), ReadPerm)
16 requires 0 <= i && i < len(elems)
17 requires 0 <= j && j < len(elems)
18 ensures  i == j ==> !res
19 decreases
20 pure less_spec(i, j int, ghost elems seq[any]) (res bool)

21 preserves acc(Mem(elems), ReadPerm)
22 requires  0 <= i && i < len(elems)
23 requires  0 <= j && j < len(elems)
24 ensures   res == less_spec(i, j, elems)
25 decreases
26 Less(i, j int, ghost elems seq[any]) (res bool)
```

Listing 15: Specification of `Less` in `sort.Interface`

```
1  // Swap swaps the elements with indexes i and j.
2  requires Mem(elems)
3  requires 0 <= i && i < len(elems)
4  requires 0 <= j && j < len(elems)
5  ensures  Mem(swapped_elems)
6  ensures  haveSameElements(elems, swapped_elems)
7  ensures  0 <= i && i < len(elems)
8  ensures  0 <= j && j < len(elems)
9  ensures  elems[i] === swapped_elems[j]
10 ensures  elems[j] === swapped_elems[i]
11 ensures  forall x int :: (0 <= x && x < len(elems) &&
12     x != i && x != j) ==> (
13         elems[x] === swapped_elems[x])
14 ensures  old(less_spec(i, j, elems)) ==> !less_spec(i, j, swapped_elems)
15 ensures  old(less_spec(j, i, elems)) ==> !less_spec(j, i, swapped_elems)
16 decreases
17 Swap(i, j int, ghost elems seq[any]) (ghost swapped_elems seq[any])
```

Listing 16: Specification of `Swap` in `sort.Interface`

`Swap` depicted in listing 16 is the only operation described in the interface that should change the underlying data structure. It accepts the sequence `elems` as an argument and returns the sequence `swapped_elems` as a result. Since sorting functions may call `Swap` many times we introduced the `ghost pure` helper function `haveSameElements` and use it in `Swap`'s postcondition in line 6. When used in the specification of a sorting function, this gives us a concise way to express that the underlying data structure's elements remain the same. The `ghost pure` function `haveSameElements` maps `elems` and `swapped_elems` to multisets and compares these. In lines 9 - 13, the postcondition expresses how

the elements at indices `i` and `j` were swapped and that the elements at all other indices remained the same. Since sorting functions use the swapping of elements to change the ordering, we state the effect of swapping on the `Less` relation in lines 14 - 15.

4.3.2 IntSlice

In addition to the interface `Interface`, we specified and verified the interface's implementation `IntSlice`. Since the methods' functionality is straightforward and the specifications are only marginally changed, we refrain from repeating the same information here. Instead, we focus on how we defined the `Mem` predicate in this implementation example. Listing 17 shows the `Mem` predicate

```
1  ghost
2  requires forall j int :: 0 <= j && j < len(x) ==> acc(&x[j], _)
3  ensures  len(s) == len(x)
4  ensures  forall j int :: 0 <= j && j < len(x) ==> (
5      s[j] == res[j])
6  decreases len(x)
7  pure func toSeq(x []int) (s seq[any]) {
8    return (len(x) == 0 ? seq[any]{} :
9        toSeq(x[:len(x) - 1]) ++ seq[any]{x[len(x) - 1]})
10 }

11 pred (x IntSlice) Mem(ghost s seq[any]){
12         len(x) == len(s) &&
13         forall j int :: 0 <= j && j < len(x) ==> acc(&x[j]) &&
14         s == toSeq(x)
15 }
```

Listing 17: `IntSlice.Mem` predicate

for `IntSlice` and the `ghost pure` helper function `toSeq`. The `toSeq` function maps the slice of `int`s to a sequence of type `seq[any]`. The `Mem` predicate itself describes permissions to all elements of the `IntSlice x`. Additionally, the predicate accepts the sequence `s` as a parameter and enforces that `s` needs to be the same as `toSeq(x)`.

A notable point about the implementation of `Swap` for `IntSlice` is that we were not able to prove `haveSameElements`. Moreover, we had to introduce two assertions in lines 24 - 27 to help along the SMT-solver when proving the other postconditions. The main difficulty lay in the fact that we had to remind Gobra about how the `IntSlice x` relates to the sequences of elements `elems` and `swapped_elems` at any point in time. However, the first of these assertions cannot be proven by using the most recent version of Gobra as of January 23rd 2023. However, verification works fine when using the stable version of GobraIDE. Likely, an update to Gobra introduced incompleteness.

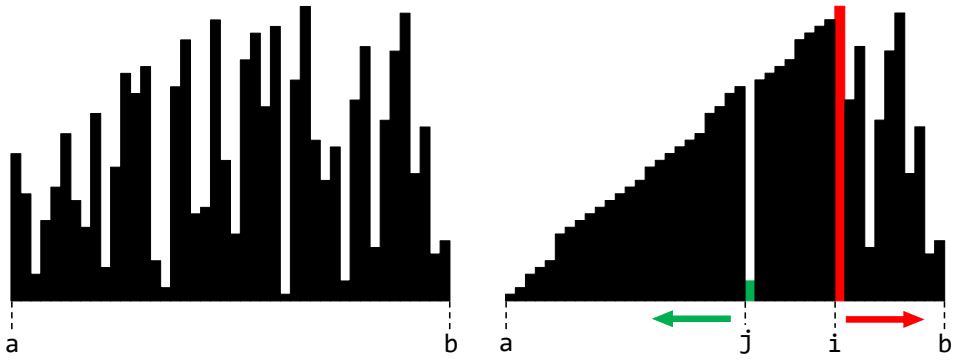
```
1  assert forall idx int :: (0 <= idx && idx < len(elems)) ==> (
2      elems[idx] == old(unfolding x.Mem(elems) in  x[idx]))
3  assert forall idx int :: (0 <= idx && idx < len(swapped_elems)) ==> (
4      swapped_elems[idx] == unfolding x.Mem(swapped_elems) in x[idx])
```

Listing 18: `IntSlice.Swap` assertions

4.3.3 Sorting function

One of the subroutines used by `pdqsort` is `insertionSort`. Our specified version of `insertionSort` is shown in listing 4.2. Since `insertionSort` is only used as a subroutine, it accepts arguments `a` and `b` which are indices into the underlying data structure. The `insertionSort` function is used to sort the elements between `a` *(inclusive)* and `b` *(exclusive)*. Our specified version of `insertionSort` additionally accepts the sequence `elems` and returns the sequence `sorted_elems`. The postcondition in line 7 shows that these two sequences contain the same elements.

Note that we had to adapt the inner loop condition in line 27 and move the second conjunct into the loop via an introduced if-statement. Gobra does currently not support short-circuiting [9]. Without this change, Gobra would not have recognized that calling `Less` with the argument `j - 1` is allowed.



(a) initial state **(b)** sort in progress

Figure 4.2: Illustration `insertionSort`

The figures depicted in 4.2 serve as a simple illustration and reminder of how insertion sort works. As in the context of the `insertionSort` function, the data shown in these figures goes from indices `a` to `b`. Initially, we start with our data in an unsorted state as shown in subfigure 4.2a. The outer loop in the `insertionSort` function then advances the pointer `i` from `a` towards `b`. In every such step, if the element at index `i` is not yet sorted, the inner loop will advance pointer `j` from `i` towards `a` and perform swaps between

```
1  // insertionSort sorts data[a:b] using insertion sort.
2  requires data.Mem(elems)
3  requires data != nil
4  requires 0 <= a && a < len(elems)
5  requires a <= b && b <= len(elems)
6  ensures  data.Mem(sorted_elems)
7  ensures  haveSameElements(elems, sorted_elems)
8  decreases
9  func insertionSort(data Interface, a, b int, ghost elems seq[any])
10     (ghost sorted_elems seq[any]) {
11         sorted_elems = elems
12         assert data.Mem(sorted_elems)

13         invariant haveSameElements(elems, sorted_elems)
14         invariant a < i
15         invariant a < b ==> i <= b
16         invariant data.Mem(sorted_elems)
17         //# invariant prefix [a:i] is always sorted
18         decreases b-i
19         for i := a + 1; i < b; i++ {
20             invariant haveSameElements(elems, sorted_elems)
21             invariant i < b
22             invariant a <= j && j <= i
23             invariant a < j ==> 0 <= j - 1
24             invariant data.Mem(sorted_elems)
25             //# invariant [a:j] and [j:i] is always sorted
26             decreases j-a
27             for j := i; j > a /*&& data.Less(j, j - 1, elems)*/; j-- {
28                 assert 0 <= j - 1
29                 assert i < b
30                 assert i < len(sorted_elems)
31                 if data.Less(j, j - 1, sorted_elems) {
32                     sorted_elems = data.Swap(j, j - 1, sorted_elems)
33                     assert !data.less_spec(j, j - 1, sorted_elems)
34                 } else {
35                     assert !data.less_spec(j, j - 1, sorted_elems)
36                     break
37                 }
38             }
39         }
40 }
```

Listing 19: Specified version of `insertionSort`

neighbors until the prefix from `a` to `i` is sorted. This behaviour is depicted in subfigure 4.2b. In lines 31 - 33, we see how a swap changes the ordering between two neighbors that were previously not ordered correctly. Ideally, we want to prove that `insertionSort` sorts the elements in the underlying data structure in increasing order according to `Less`. A natural choice for a loop invariant for the outer loop would be that the prefix `[a:i]` is always sorted. Since `i` eventually reaches `b`, loop termination would imply that elements in `[a:b]` are sorted correctly which is the goal. Since the inner loop sorts in the element pointed to by index `j` into this prefix, possible invariants would be that the elements in `[a:j]` and `[j:i]` have to be sorted.
However, we were not able to prove this using Gobra and our attempted specifications. For any definition of sortedness Gobra complains the inner loop invariant could not be preserved. One possible reason for this is that the inner loop's loop condition is a conjunction between `j > a` and `data.Less(j, j - 1, sorted_elems)` whereas the termination measure is only concerned

with the first of these conjuncts. The inner loop terminates as soon as the element is correctly sorted into the prefix. If the element was the smallest element according to `Less` then the loop terminates as soon as the termination measure reaches zero. Otherwise, the loop terminates earlier, i.e. as soon as `!Less(j, j-1)` holds.

```
1  forall i int :: (a <= i && i < b) ==> !less_spec(i, i - 1)
2  forall i, j int :: (a <= i && i <= j && j < b) ==> !less_spec(j, i)
```

Listing 20: Definitions of sortedness

Listing 20 shows two simplified versions of the ways we tried to define sortedness. The definition in line 1 enforces for all pairs of neighbors, that the one with the higher index is not `Less` than the one with the lower index. I.e. the one with the lower index is less than or equal to the one with the higher index according to `Less`. The transitivity of `!less_spec` then implies that all elements are sorted. We can successfully prove this definition of sortedness in the postcondition of the `sort` package's `IsSorted` function. In contrast, the definition in line 2 does not rely on the transitivity of `!less_spec`. Here, it is explicitly enforced that, for any given element, any element with a higher or equal index is not `Less`.
Another, or additional, reason why we cannot prove sortedness might be because Gobra "forgets" the ordering between pairs of elements across calls to the `Swap` method. However, explicitly extending `Swap`'s postcondition to mention the ordering of all pairs with `less_spec` based on the `old(..)` context and `Swap`'s actions did not solve the issue.

Chapter 5

Evaluation

In this chapter, we present and evaluate a few key characteristics of our specifications. Furthermore, we will outline some of the limitations of our specifications as well as describe the encountered issues when working with Gobra.

5.1 Annotation Overhead

Deriving specifications for a program comes with a significant overhead as already established in section 2.2.2. In this section, we measure and compare the amount of lines of annotation and amount of lines of original source code. We regard the following as constituting annotation:

- Any clause in the specification of a method
- Loop variants and invariants
- Any `ghost` statement
- Any inlined ghost code, i.e. `unfoldings` as well as method calls or function definitions with ghost parameters
- `ghost` functions
- Predicate definitions
- Implementation proofs

We only list methods for which the specification is working as of the end of this project.

List

Table 5.1 shows a complete overview comparing the overhead of lines of annotation for every specified method. We note that for every method

Method name	#lines of code	#lines of annotation	Method name	#lines of code	#lines of annotation
`Next`	6	9	`move`	11	11
`Prev`	6	9	`Remove`	6	15
`Init`	6	9	`PushFront`	4	11
`New`	3	5	`PushBack`	4	10
`Len`	3	7	`InsertBefore`	6	16
`Front`	6	7	`InsertAfter`	6	16
`Back`	6	7	`MoveToFront`	6	15
`lazyInit`	5	12	`MoveToBack`	6	15
`insert`	9	13	`MoveBefore`	6	18
`insertValue`	3	11	`MoveAfter`	6	18
`remove`	8	10			

Table 5.1: Annotation overhead in `list` package

we required at least as many lines of annotation as there were lines of source code. The method `insertValue` required 3.7 times as many lines of annotation as there were lines of source code. This is mostly explained by the fact that the method itself is short with only 3 lines of code and that we wished to carry forward `insert`'s postcondition which describes the inserted element's position. Further contributing to lengthy annotations are methods that require a distinction between receiving initialized vs. uninitialized lists. This is shown by the large relative annotation overhead of up to 2.8 in `Len`, `lazyInit`, `PushFront` and `PushBack`. The overall largest contributor was the requirement that we establish the equivalence described in 4.1. The methods `MoveBefore` and `MoveAfter` accept two `Element` parameters and this equivalence had to be established for both of these elements. Therefore, with only 6 lines of source code, we had to provide 18 lines of annotation for both of these methods. We also required a total of 45 lines for helper functions and the definition of the `Mem` predicate. In total, we required 2.4 times as many lines of annotation as there were original lines of code.

Ring

A complete overview comparing the overhead of lines of annotation for every specified method is shown in table 5.2. Note that for `New` we compare our adapted implementation instead of using the original version. Despite this adapted implementation having more lines of code, `New` is the only method where we required fewer lines of annotation than lines of actual source code. This shows the benefit of being able to use the `Mem` predicate in the loop invariant. `Next` and `Prev` on the other hand require 3 times as many lines

Method name	#lines of code	#lines of annotation
`init`	5	7
`Next`	6	18
`Prev`	6	18
`Move`	16	23
`New`	19	13
`Link`	11	31

Table 5.2: Annotation overhead in `ring` package

of annotation compared to lines of source code. This is the result of us distinguishing the permission amount to the `Mem` predicate based on whether the ring structure is already initialized. The highest annotation overhead in absolute terms is seen in `Link` where we had to distinguish between different cases. Since we did not treat all possible cases in our specification, the overhead of a complete version would likely be higher. There were 11 additional lines of annotation spent on the definition of the `Mem` predicate. Summing up the specified methods and any additional annotation, we have 1.9 times as many lines of annotation compared to lines of source code.

Sort

Method name	#lines of code	#lines of annotation
`Interface.Len`	1	4
`Interface.Less`	1	6
`Interface.Swap`	1	14
`IntSlice.Len`	1	4
`IntSlice.Len`	1	7
`IntSlice.Swap`	1	20
`IsSorted`	9	17
`insertionSort`	7	31

Table 5.3: Annotation overhead in `sort` package

The relative overhead of annotation in the `sort` package, shown in the table 5.3 is overblown for two reasons. Firstly, we only managed to specify a small amount of the package, which causes the relative overhead incurred by helper functions to increase. Secondly, the interface `Interface` naturally only has one line of code per defined method. Since the implementation `IntSlice` is relatively simple, its methods were originally also written on just one line. The more typical function `IsSorted` has 17 lines of annotation for 9 lines of source code. However, `insertionSort` has 31 lines of annotation compared to 7 lines of source code which is an overhead factor of 4.4. This overhead is likely to increase when the specification gets updated to prove functional

correctness. Summing up all specified parts of the `sort` package, including any predicates and helper functions, we have 160 lines of annotation for only 22 lines of source code.

5.2 Timing

Poorly designed specifications can have a significant impact on verification time. Consequently, whenever reasonably possible we tried to reduce the number of required `unfolding`s and replaced them with calls to pure functions. Also, the triggers for the quantified expressions were chosen in such a way as to minimize unnecessary instantiations of the quantifiers. In this section, we report the verification times of individual methods as well as whole packages. We report the verification times in seconds as displayed by Gobra as "Total time". All tests are conducted without the use of the `--cacheFile` flag. For measuring the time to verify a single method we use the `file.go@linenumber` notation. We recognize that by using this methodology many of the reported verification times for single small methods are dominated by the overhead of setting up the verification pipeline. This is shown by the fact that it takes a similar amount of time to verify the "fastest" methods regardless of whether they are annotated as `trusted`. However, these results are still valuable since they give an upper-bound on the time it takes to verify any given method.

- Windows 10 laptop
- Intel i7-3630QM CPU with 4 physical (8 logical) cores at 2.4 GHz
- 2 $\times$ 8GB SODIMMs at 1600MHz
- Java version 8 update 341
- z3 for Windows version 4.11.2 - 64 bit
- Gobra, Viper and Silicon fully updated as of January 17th 2023

Other user-facing programs were closed, malware scans halted and intensive background processes stopped. All test runs were conducted five times. In our results, we report the runtimes in seconds and the average with the longest and shortest runtimes excluded and rounded to one decimal.

List

As shown in table 5.4, verification times for most methods hovered around 25 seconds. In `insert` and `remove`, where we change the structure of the list, we measure up to 55.3 seconds on average. `MoveBefore` and `MoveAfter` yield verification times of 47.3 and 38.3 seconds respectively on average. Interestingly, the average verification time for `move` is 329 seconds and by far the longest. We think this is because of the implication in the postcondition

Method name	*(shortest)*				*(longest)*	avg
`Next`	24	25	25	26	26	25.3
`Prev`	23	24	26	26	26	25.3
`Init`	26	27	27	27	27	27
`New`	22	22	22	22	23	22
`Len`	30	30	34	35	38	33
`Front`	25	25	25	25	25	25
`Back`	25	25	25	25	25	25
`lazyInit`	23	23	23	23	24	23
`insert`	52	53	56	57	61	55.3
`insertValue`	24	24	24	24	24	24
`remove`	36	38	39	39	43	38.7
`move`	287	296	337	354	468	329
`Remove`	26	26	26	26	27	26
`PushFront`	23	23	23	23	25	23
`PushBack`	23	23	23	23	23	23
`InsertBefore`	28	28	29	29	29	28.7
`InsertAfter`	27	27	27	27	29	27
`MoveToFront`	27	27	28	28	30	27.7
`MoveToBack`	31	31	31	32	33	31.3
`MoveBefore`	42	45	47	50	50	47.3
`MoveAfter`	38	38	38	39	41	38.3

Table 5.4: Verification time measurements in `list` package

where we describe the moved element's new position. If we leave out these postconditions `move` verifies in under 60 seconds. Also, `move`'s verification time shows the highest variability with the shortest run only taking 287 seconds and the longest run taking 468 seconds. While it makes sense that we experience higher variability if the overhead is not the dominating factor, this large difference is still surprising. A reason for this behaviour could be that we likely chose our triggers sub-optimally. Verifying the `list.go` file as a whole, the average verification time is 490.7 seconds and shows a similar variability as in `move`.

Ring

Table 5.5 shows the verification times for the specified methods in the `ring` package. The verification times for `init`, `Next`, `Prev` and `Move` are very consistent. Of these, `Move` has the highest average at still only 22 seconds. Our adapted version of `New` verifies after 73.6 seconds on average. `Link` has by far the longest verification time of 215.3 seconds on average. This is likely due to the case-split and the correspondingly convoluted postconditions and

Method name	*(shortest)*				*(longest)*	avg
`init`	19	19	19	19	20	19
`Next`	20	20	20	20	21	20
`Prev`	20	20	20	21	21	20.3
`Move`	22	22	22	22	23	22
`New`	52	55	75	91	93	73.6
`Link`	175	197	205	211	230	215.3

Table 5.5: Verification time measurements in `list` package

conditional folds of the `Mem` predicate.

Sort

Only a small fraction of the total number of functions and methods in the `sort` package could be specified and verified. On average, verification of all these together only takes 48 seconds. Therefore, a detailed listing of separate runtimes for single methods is omitted.

5.3 Tests

Successful verification of a program guarantees that the program fulfils the provided specification. However, the guarantees one gets from such a successful verification are only as valuable as what is explicitly stated in the program's specifications. To gauge the comprehensiveness of our specifications, we check whether we can use them to annotate the unit tests in the `list_test.go` file and prove the tested properties statically. Additionally, we will show how we can use the specified version of our code to detect a known bug. The tests in this section are only performed on our specified version of the `list` package. In this package, we managed to specify and verify the majority of methods which gives us enough to perform a few tests.

5.3.1 Test cases

The file `list_test.go` contains a variety of test cases for the implementation of the `list` package. The tests themselves are a collection of functions performing operations on a list by calling its methods. Originally, after some calls to these methods, either a call to `checkListPointers` or a manual check via an if-statement follows. The `checkListPointers` function accepts three arguments. One key argument is a slice of `Element` pointers. The function then checks whether the elements in that slice are contained in the list and if they are in the required order with all pointers correctly set. In this section,

we will present a small and adapted subset of these test cases along with some additional annotation.

```
1  func TestZeroList() {
2      var l1 = new(List)
3      fold l1.Mem(set[*Element]{&l1.root}, false)
4      e := l1.PushFront(1, set[*Element]{&l1.root}, false)
5      assert l1.Mem(set[*Element]{&l1.root, e}, true)
6      //# ...
7  }
```

Listing 21: Push to uninitialized list

The example in listing 21 shows a push to an uninitialized list. Since the list did not get instantiated via the `list` package's `New` function, we get an uninitialized list in line 2 and can consequently only fold the `Mem` predicate with `isInit == false`. Since both `PushBack` and `PushFront` call `lazyInit`, we can call `PushFront` on the uninitialized list in line 4. As a result, we obtain the `Mem` predicate for a now initialized list and the new element `e` included in the set of elements. It is noteworthy that `InsertBefore` and `InsertAfter` do not call `lazyInit`. Both these methods should only change the state if their `mark` argument is part of the list. The developers' implicit assumption was likely that if an element whose `list` field pointed to the receiver list was passed as an argument then the list is guaranteed to have already been initialized. However, as we have shown in the previous section, this assumption is wrong. A pointer to a list could have been leaked before. Furthermore, it is possible to reset a list `l` to an uninitialized state via `l = list.List` [5]. With this, `l` would keep its address and `mark.list == l` would still hold true. To avoid this issue altogether, we forbid this implicit assumption and require the list to always be initialized when a call to `InsertBefore` or `InsertAfter` is made.

```
1  func TestInsertBeforeUnknownMark() {
2      var l = new(List)
3      fold l.Mem(set[*Element]{&l.root}, false)
4      e1 := l.PushBack(1, set[*Element]{&l.root}, false)
5      e2 := l.PushBack(2, set[*Element]{&l.root, e1}, true)
6      e3 := l.PushBack(3, set[*Element]{&l.root, e1, e2}, true)
7      assert l.Mem(set[*Element]{&l.root, e1, e2, e3}, true)
8      mark := new(Element)
9      assert unfolding l.Mem(set[*Element]{&l.root, e1, e2, e3}, true)
10         in mark != &l.root
11     l.InsertBefore(1, mark, set[*Element]{&l.root, e1, e2, e3})
12     assert l.Mem(set[*Element]{&l.root, e1, e2, e3}, true)
13 }
```

Listing 22: `InsertBefore` not contained element

In listing 22 we see how `InsertBefore` is called in line 11. The `mark` argument for this call is not contained in the list (and properly `mark.list != l`). In this scenario, the call to `InsertBefore` should leave the list contents unchanged.

This is indeed the case as can be asserted in the following line. However, even though the `mark` element gets newly created in line 8, Gobra cannot derive itself that it is guaranteed to be different from the already existing `&l.root` and would thus fail to fulfil `InsertBefore`'s precondition. We can trigger this knowledge by unfolding the `Mem` predicate and manually asserting the fact.

```
1  func TestList() {
2      l := New()
3      //# ...
4      e2 := l.PushFront("2", set[*Element]{&l.root}, true)
5      assert l.root.comesBefore(e2, set[*Element]{&l.root, e2}, l)

6      e1 := l.PushFront(1, set[*Element]{&l.root, e2}, true)
7      assert l.root.comesBefore(e1, set[*Element]{&l.root, e1, e2}, l)

8      e3 := l.PushBack(3, set[*Element]{&l.root, e1, e2}, true)
9      assert e3.comesBefore(&l.root, set[*Element]
10         {&l.root, e1, e2, e3}, l)

11     e4 := l.PushBack("banana", set[*Element]{&l.root, e1, e2, e3}, true)
12     assert e4.comesBefore(&l.root, set[*Element]
13         {&l.root, e1, e2, e3, e4}, l)

14     l.Remove(e2, set[*Element]{&l.root, e1, e2, e3, e4})
15     assert l.Mem(set[*Element]{&l.root, e1, e3, e4}, true)
16 }
```

Listing 23: Insertion and Removal

The `TestList` function in listing 23 shows the creation of a list followed by some push operations and finally the removal of an element. We see how the set of elements in the `Mem` predicate gets extended with every push operation and how the correct element gets removed in the call to `Remove`. Moreover, after every push operation, we can observe the correct neighborhood relation between the list's root element and the newly created and inserted element. However, there are some strict limitations to checking these neighborhood relations. The details of these limitations will be expounded on in section 5.4.

5.3.2 Known issue

As already outlined in section 3.1, there is a known issue when it comes to the validity of using `e.list == l` to check whether `Element e` is in `List l` [4]. The underlying issue stems from a desire to have a constant time reset of a list via `Init` and a constant time membership check. A working, yet inefficient, solution would be to use a linear time lookup in all methods that have to check whether an element `e` is contained in list `l`. Contrary to this, and assuming that list resets happen less often than inserts or moves, it would seem more efficient to only incur this linear time penalty at times when the list is reset. In this case, one could simply clear the `list` field for every element in the list and users could also choose to simply allocate a new list instead of performing a reset via `Init`. As described in section 4.1.5,

we enforce the equivalence `e.list == l` $\iff$ `e in elems` under the predicate `l.Mem(elems, true)` in various methods' preconditions. The example shown

```
1  func github_issue_50152() {
2      l := New()
3      assert l.Mem(set[*Element]{&l.root}, true)
4      el := l.PushBack(42, set[*Element]{&l.root}, true)
5      assert l.Mem(set[*Element]{&l.root, el}, true)
6      l = l.Init(set[*Element]{&l.root, el}, true) // Reset the list.
7      assert l.Mem(set[*Element]{&l.root}, true)
8      l.Remove(el, set[*Element]{&l.root})
9      //# Call fails since precondition (el in elems) == (el.list == l)
10     //# does not hold.
11 }
```

Listing 24: GitHub Issue 50152

in listing 24 is an adapted version of an open issue on GitHub [6]. If executed normally, the call to `Pushback` in line 4 leaks the pointer to the newly created and inserted element `el`. After the list is reset in line 6, one could still call `Remove` and pass `el` as an argument. This is because `el.list` was still pointing to the list `l`. The list would then report a length of -1. With our verified version of the `list` package, we disallow this behaviour. For one thing, `Remove` could not preserve the list invariants if `l.lenT` were below zero. However, since we enforce the equivalence shown in 4.1 and we have `!(el in set[*Element]&l.root)` but `el.list == l`, the precondition of `Remove` is not fulfilled and the call is disallowed in the first place.

```
1  func github_issue53351() {
2      l0 := New()
3      assert l0.Mem(set[*Element]{&l0.root}, true)
4      e0 := l0.PushBack(nil, set[*Element]{&l0.root}, true)
5      assert l0.Mem(set[*Element]{&l0.root, e0}, true)
6      l1 := l0.Init(set[*Element]{&l0.root, e0}, true)
7      assert l0 == l1
8      assert l0.Mem(set[*Element]{&l1.root}, true)
9      e1 := l1.PushBack(nil, set[*Element]{&l1.root}, true)
10     assert l0.Mem(set[*Element]{&l1.root, e1}, true)
11     l0.MoveAfter(e0, e1, set[*Element]{&l1.root, e1})
12     //# Call fails since precondition (e0 in elems) == (e0.list == l0)
13     //# does not hold.
14 }
```

Listing 25: GitHub Issue 53351

The example in listing 25 is adapted from another GitHub issue [4] and is based on the same underlying problem. In line 4 it leaks the pointer to the

newly created and inserted element `e0`. The element is subsequently removed from the list via a reset in line 6. Normally, one could still call `MoveAfter` with the element `e0` as a parameter and it would try to perform the move operation since it believes `e0` to be in the list. If continued further, this would ultimately result in a runtime panic. The precondition of `MoveAfter` also enforces the equivalence shown in 4.1 for all the elements passed as arguments. This effectively prevents us from calling `MoveAfter` with these arguments.

5.4 Functional Limitations

For all the methods discussed so far, we could successfully prove memory safety. However, there are still issues in our specification attempt. This section discusses some of the limitations of our specifications. We also explore the implications of these limitations as well as explore possible remedies in some cases.

Unsuitable abstractions

As pointed out multiple times, the invariants for packages `list` and `ring` do not enforce that all the elements form a single cycle. This posed a problem in the `list` package where we were unable to specify and verify the `PushBackList` and `PushFrontList` methods. This was because their loop invariants would have relied on the fact that iterating over the list by calling `Next` (or `Prev` respectively) repeatedly should take exactly `len(elems)` many iterations until we'd have seen every element exactly once. In `ring`'s case, most methods rely on the fact that all ring elements form a single cycle, namely `Move`, `Link`, `Unlink`, `Len` and `Do`. A first naive approach to enforcing this structure was based on reachability between elements and using the pure functions `moveNext(n int)` and `movePrev(n int)` where e.g. `e.moveNext(2)` would produce `e.next.next`. However, when trying to incorporate this approach, verification would not terminate in over 24 hours at which point we stopped Gobra. Having a proper solution to this would give us a notion of the ordering of the elements. As described in the next section, this could make framing easier.

Lack of proper framing

Our specifications are mostly a positive description of a method's behaviour. For example, our specification of the method `insert` in the `list` package depicted in listing 5 only talks about the fact that the to-be-inserted element `e` gets included in the set of elements `elems` and that it is neighboring element `at` and the element previously at `at.next`. However, the specification gives no further detail about the values at any of the other heap locations except that the invariants imposed by the `Mem` predicate could still be asserted.

In principle, solely going by the specification, `insert` could have changed the values of any elements in the list and arbitrarily shuffled any other neighborhood relations as long as it is still compliant with the list invariants. An example of this is the test case depicted in listing 23 at the end of the previous section. After the call to `PushBack` in line 11, we wished to be able to show the order of the elements is`root -> e1 -> e2 -> e3 -> e4 -> root`. However, we can only prove what we've been explicitly guaranteed by `PushBack`'s postcondition at that point. Any previous guarantees on the order of the elements have been lost since `PushBack` could have changed it. Instead of introducing additional postconditions, a likely better and more performant way to deal with this issue would be to adapt proper framing when dealing with permissions. With this, we would only demand write permissions to the heap locations a given method tries to update. However, since we are dealing with cyclical and doubly linked data structures in the cases of `list` and `ring`, this has a tendency to become convoluted in the current design. This is because it would require numerous implications and case distinctions to determine what heap locations require write permissions based on the length and order of the list. Due to time constraints, we were not able to implement a better solution to this issue. Updated versions of the `Mem` predicate where the abstraction provides a notion of the ordering of elements could make framing significantly easier.

Verification unstable

In section 5.3.1 we illustrated how we can leverage the verified `list` package when going over the test cases in the file `list_test.go`. While we could verify the package by itself, some test cases exhibited the behaviour that Gobra non-deterministically reported a postcondition of a called library method would not hold. This happened even though we could successfully assert the called method's precondition. In the example in listing 26, we create a new

```
1  func testingIssue() {
2      l := New()
3      e1 := l.PushFront(1, set[*Element]{&l.root}, true)
4      e2 := l.PushFront(2, set[*Element]{&l.root, e1}, true)
5      l.Remove(e1, set[*Element]{&l.root, e1, e2})
6      assert l.Mem(set[*Element]{&l.root, e2}, true)
7      l.MoveToFront(e2, set[*Element]{&l.root, e2})
8  }
```

Listing 26: Testing issue

list and insert two elements in lines 2 - 4. After the removal of element `e1` in line 5, we can successfully assert the `Mem` predicate in line 6. Even though we fulfil `MoveToFront`'s postcondition, Gobra reports the permissions to the `Mem` predicate might not suffice. One commonality of all these issues is that we

only get this error if we make a call to a `ghost pure` function in the `old(..)` context in the failing method's postcondition. One suspicion is that poorly chosen triggers in the library specifications' quantifiers might contribute to this unstable behaviour.

Lack of information-hiding

We did not achieve our goal of properly following information-hiding principles in the packages `list` and `ring`. There are many public methods whose specifications still reveal implementation details like private fields or how internal checks are made. For example, many of the specifications in the `list` package explicitly mention the list's `root` field. A better abstraction that provides a notion of the ordering of elements could also help hide any information about the `next` and `prev` pointers.

Lack of functional correctness proof for `ring.Move`

The pure functions `moveNext` and `movePrev` were initially introduced to be used in the loop invariants and postcondition of `ring`'s `Move` method. Testing these pure functions via the postcondition of dummy methods that would simply return e.g. `e.next.next` worked fine. However, when trying to use them in `Move`'s invariants, we encountered the same problem with seeming non-termination. Therefore, the current specification for `Move` only guarantees memory safety. Since `Unlink` relies on the return value of `Move` and a specific case of `Link`, we were also unable to specify `Unlink`.

Lack of functional correctness proof for `insertionSort`

As described in detail in section 4.3.3, we could not prove that `insertionSort` sorts the underlying data in the desired order. While it is good that we could prove memory safety, functional correctness of a sorting function is essential and required if we aim to ultimately prove functional correctness of `pdqsort`.

5.5 Observations concerning Gobra

This project was my first ever practical experience with program verification. While I already had some theoretical knowledge of code contracts and specifications, the whole concept of the permission model and predicates was completely new to me. This provided me with the valuable vantage point to evaluate Gobra as an outsider. This section will primarily deal with my observations. I will outline issues I discovered, most of which were already known as well as describe possible future features.

Lack of a `let` construct

Suppose we had a `ghost pure` helper function called `getElement` returning the pointer to an element. In this case, a `let` construct could look like `let e == getElement() && e.field1 == 1 && e.field2 == 2`. It would effectively introduce a new variable, i.e. `e` in this case, for use in the specification. A `let` construct could make the specification more compact many use cases since it allows us to give shorter names for long expressions that are used multiple times.

Gobra tutorial

The Gobra tutorial [10] has been an invaluable resource when first starting with Gobra. It gives a good overview of basic principles and even more advanced features such as concurrency and mutual exclusion. However, some parts of the tutorial are outdated and there are minor inconveniences. Examples include the following:

- Several of the internal links are not set correctly (this applies to the Viper tutorial as well). E.g. in the introduction chapter, there is a subheading "Predicates" with a link that should lead to the section on predicates. However, it leads back to the original subheading.
- It is no longer necessary to use a semicolon when a line ends with the `!<..!>` delimiter.
- `|set|` is not the correct way to obtain set cardinality.
- In the section on interfaces `typeOf(y) == int` does not work. Instead one should use `typeOf(y) == type[int]`.

Chapter 6

Conclusion

In this project, we derived criteria to choose a subset of Go's standard library to verify using Gobra. Based on these criteria, we attempted to verify parts of the packages `list`, `ring` and `sort`. In the `list` package, we could prove memory safety for all but two methods. This allowed us to use our specifications to disallow a known bug in the package. In the `sort` package, we specified and verified the interface `Interface` with its implementation `IntSlice`, as well as the functions `IsSorted` and `insertionSort`. We reported favourable verification performance and provided a detailed assessment of the required annotation overhead. Furthermore, we provided an overview of the current specifications' limitations and showed how they are not yet sufficient to provide meaningful guarantees for functional correctness.

6.1 Future work

As pointed out in the section about the specifcations' limitations in 5.4, there are still multiple avenues to explore in order to increase our results' usefulness and impact.

- The invariants of the `list` and `ring` packages, encapsulated in their respective `Mem` predicates, should be updated such that a single cycle of all the elements is enforced. This is more important for `ring` since its methods often explicitly rely on this property. Having accomplished this, specifying the remaining few methods should require less effort.
- With an updated abstraction that also enforces an ordering of the elements, our handling of permissions and enforcements of neighborhood relations could be vastly simplified. Given this, it should also be easier to prove functional correctness and data structure integrity across multiple method calls.

- Wherever our specifications reveal implementation details, it could make sense to introduce suitable `ghost pure` functions instead. Moreover, an updated abstraction would further help to hide implementation details. It is also worth considering that even though the `Element` type is publicly accessible, the fact that a list consists of elements is also an implementation detail. A possible alternative would be to use a sequence of values as the abstraction for a list.
- With the interface `Interface` and the implementation `IntSlice` in the `sort` package already being specified, one could continue to specify some of the other provided implementations of `Interface`, e.g. `FloatSlice`.
- The `sort` package consists of many more building block functions that are ultimately used in the `pdqsort` function in the `zsortinterface.go` file. Some of these building blocks are `heapSort`, `partialInsertionSort`, `partition`, `choosePivot` and `breakPatterns`. To ultimately verify the publicly exposed `Sort` function, it makes sense to first specify and verify these building blocks and also prove `insertionSort`'s functional correctness.

Bibliography

[1] Nasa Safety Center. Lost in translation. In S. Lilley, editor, *System Failure Case Studies*, volume 3, 2009.

[2] Go programming language. https://go.dev/ref/spec.

[3] Go standard library. https://pkg.go.dev/std.

[4] Go stdlib bug list package. https://github.com/golang/go/issues/53351.

[5] Go stdlib bug list package, manual reset. https://github.com/golang/go/issues/39014.

[6] Go stdlib bug list package, uninitialized. https://github.com/golang/go/issues/50152.

[7] Go stdlib bug net/url package. https://github.com/golang/go/issues/53763.

[8] Gobra keyword issue. https://github.com/viperproject/gobra/issues/118.

[9] Gobra short circuiting issue. https://github.com/viperproject/gobra/issues/511.

[10] Gobra tutorial. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md.

[11] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015.

[12] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[13] Orson R. L. Peters. Pattern-defeating quicksort. *CoRR*, abs/2106.05123, 2021.

[14] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[15] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.

[16] Z3 theorem prover. https://github.com/Z3Prover/z3.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verifying Go's Standard Library

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Jenny	Adrian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Jan 15th 2023, Rotherburg LU	A. Jenny

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.