Verifying Go's Standard Library

Practical Work Project Description

Adrian Jenny
Supervised by Prof. Dr. Peter Müller, João Pereira
Department of Computer Science
ETH Zürich
Zürich, Switzerland

I. Introduction

Go [1] is a statically typed and compiled programming language that is 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. Unlike other established languages in the enterprise field (e.g. C++, C#, Java, etc.), Go employs a version of structural subtyping instead of relying on a nominal type system. On top of this, Go does not provide classes but works purely with interfaces and type definitions. Furthermore, Go offers a large standard library of functions and methods that are used in varying contexts. These range from the provision of simple data structures, efficient implementations of commonly used mathematical functions, a cryptography library and an entire network stack. Consequently, Go programmers heavily rely on the functionality provided by the standard library. Given that the standard library is so heavily used, the correctness of a Go program is typically dependent on the standard library being implemented correctly. To help achieve this correctness the developers of the standard library rely on testing their implementation against a broad set of inputs. Unfortunately, testing is not sufficient to ensure the correctness of a program. Thus, even the standard library ships with bugs [2], [3]. This project aims at providing stronger correctness guarantees for the standard library implementations. Instead of testing, we want to verify the functions. Program verification is an approach that allows us to formally specify the intended behaviour of a program and automatically check whether the program behaves according to this specification. 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. Furthermore, considering other programs' reliance on the standard library, this would immediately benefit all users of Go. Thus, this project's overall goal is to apply verification to a carefully selected subset of Go's standard library.

II. Background

A. Testing and Verification

Testing has been a staple feature in the field of software engineering for decades now. Often, this is achieved via providing a suite of tests for any functions and even further tests to examine whether the interaction of multiple components still yields the desired results. Typically testing is done by providing sample inputs to a function and comparing its outputs to the expected results. These sample inputs have to be chosen carefully in the hope that they cover every edge case a function might have. Having functions that access mutable global state makes reliable testing even more difficult. While we recognize the benefits of testing, we also know its limits. Testing is simply not enough to prove the absence of errors entirely.
Software Verification on the other hand aims to provide tools to specify a given program and then verify that program according to this specification. To achieve this, verification engines rely on assumptions on the inputs to a function as well as on the global state before the function execution. These assumptions are called the pre-condition of a function. Correspondingly, a post-condition is what the verification aims to prove given a function and its pre-condition. Pre- and post-condition together are called a function's specification. The example in listing 1 shows a function `nextCollatz` together with its pre- and post-condition under the `requires` and `ensures` clause respectively. Verification aims to be a sound approach, i.e. it has

```
1 requires n > 0
2 ensures  (res == n/2) || (res == 3*n+1)
3 func nextCollatz(n int) (res int){
4         if n \% 2 == 0{
5                 return n/2
6         }
7         return 3*n+1
8 }
```

Listing 1: Example of pre- and post-condition of a function.

to disallow any behaviour that is not guaranteed to satisfy the specification. Due to the inherent difficulty in proving that a system's behaviour follows a given specification, coming up with a suitable specification that a verification engine is able to verify in reasonable time can sometimes be a challenge. Furthermore, 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 [4]. 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.

B. Gobra

With Go's rise in popularity, having a strong verification framework around the language grew ever more important. To fit this need, the Programming Methodology Group at ETH developed Gobra [5]. Gobra is a modular verifier for the Go language. 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. Gobra works by translating Go programs and their specifications to the Viper intermediate verification language [6] that was likewise developed by the Programming Methodology Group at ETH. 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 [7] and works by augmenting specifications with requirements regarding (exclusive) access permissions to given heap locations. This helps to ensure that functions only access valid memory locations and the exclusive write-permissions allow proving the absence of race conditions. For programs containing loops and recursion structures 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.

III. 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 implementation can have a far-reaching impact.Investing the effort to verify the standard library will therefore provide value to all those users. Furthermore, while there exist Gobra specifications for some standard library function definitions, these have never been verified themselves. Therefore, the verification of any Go program that relies on the standard library cannot be guaranteed to be correct unless the used standard library functions are themselves verified against their specifications.

A. Choice of subset

Considering the fact that verification of the whole standard library is a project that could 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:

- *High 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 further often used by users of the standard library.
- *Bottom-Up:* 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.
- *Nothing machine/platform-specific:* Somewhat contrary to the above yet still in tune with the first criterion is that we wouldn't want to 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.
- *Algorithm-heavy:* A subset that encompasses a lot of 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.

These criteria already provide some motivation for this work's impact. Moreover, they also outline this as a further opportunity to test and showcase Gobra's strengths and weaknesses. Based on these criteria we identified the following packages as candidates to verify:

- `list, ring, 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.
- `sort`: Sorting is one of the most central and essential operations in data handling pipelines. The `sort` package defines a type `Interface` whose implementations

provide comparability and other functionalities for collections. On top of this, the package also offers implementations of multiple sorting algorithms.
- `io`: Any useful real-world program has to deal with input and output at some point. The `io` package contains many well-documented interface definitions for `Reader` and `Writer` types. It would be interesting to specify the intended behaviour of these functions, based on their documentation.
- `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.

This list is quite extensive. So this project may only consider some selected units of functionality within these packages and merely verify those.

B. Challenges

The standard library is a large code base with many inter-dependencies. Since it was not originally developed with verification in mind, some design decisions might pose a challenge when it comes to deriving a suitable specification. The general idea is to follow a bottom-up approach. This applies to the overall structure concerning the dependencies between packages (as mentioned in III-A), 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 post-conditions from these lower-level functions first and use them to build up and establish a clean public-facing interface. The idea is to abstract any consistency conditions of the underlying data structure under this predicate whenever possible. Furthermore, for a method's contract to be explainable purely in terms of publicly available members, we plan to introduce additional pure functions and predicates. An issue with this approach is that the specifications for the lower-level functions might still need to be reevaluated when we later discover calls to these functions that are not wholly compatible. Moreover, as can be seen in listing 2 we will have to deal with cyclically linked elements. This is specifically the case for the packages `list` and `ring`. This poses additional challenges since it requires cleverly constructed predicates that differentiate the required permissions based on the data structure's current state (e.g. cyclical but empty or even temporarily broken cycles etc.).

As mentioned before in I, Go employs a version of structural subtyping. While this difference has many consequences, one of the main points for users of the language is that this represents a lack of opportunity to express specific intent within the language itself when it comes to subtyping relations. In some cases, it may even happen that a subtyping relation between two user-defined types is completely accidental. In this case, the expectations on the behaviour of a type's methods may not at all agree with the behaviour of this type's subtype's methods or vice-versa. We are hopeful that accidental subtyping will not occur in the standard library implementation. Nevertheless, the constraints of behavioural subtyping might still be challenging.

```
type Element struct {
  // Next and previous pointers in the
  // doubly-linked list of elements.
  // To simplify the implementation, internally
  // a list l is implemented as a ring, such that
  // &l.root is both the next element of the last
  // list element and the previous element of the
  // first list element.
  next, prev *Element

  // The list to which this element belongs.
  list *List

  // The value stored with this element.
  Value any
}
```

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

IV. Goals

Having outlined the importance of having verified standard library primitives we can define the following goals this project aims to accomplish:

(1) Applying verification:
Using Gobra to verify selected methods and functions from the packages listed in III-A.

(2) Providing a clean abstraction:
The specifications for public-facing functions should not reveal information about the internal representation. This will be achieved via the use of predicates and ghost pure functions in Gobra.

(3) 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.

(4) Issues and tests
Publicly known bugs and issues in the Go standard

library are observable on GitHub[1]. For the functions we are able to specify, we want to track down whether there are any remaining open issues 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 [2]. Similarly, where there are already tests available we want to check if our specification is sufficient to guarantee whether a particular test will be successful.

References

[1] Go programming language. [Online]. Available: https://go.dev/ref/spec

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

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

[4] C. Hawblitzel, J. Howell, M. Kapritsos, J. Lorch, B. Parno, J. Stephenson, S. Setty, and B. 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. [Online]. Available: https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/

[5] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification (CAV)*, ser. LNCS, A. Silva and K. R. M. Leino, Eds., vol. 12759. Springer International Publishing, 2021, pp. 367–379. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17

[6] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

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

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