Verification of closures for Go programs

Master Thesis Project Description

Stefano Milizia

Supervisors: Felix Wolf, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

Start: 16th Feburary 2022
End: 15th August 2022

1 Introduction

Go is a programming language that focuses on scalability and concurrency. It is used in several large-scale web-based software systems. Such systems include security-critical applications, where it is fundamental to verify that the implementation does not have any bugs.

Gobra [1] is a tool that supports formal verification of Go programs. The tool takes Go programs, annotated with logical statements (specifications) that describe the intended behaviour of the program, and automatically verifies whether the code satisfies the specification or not. Verification is achieved by translating annotated Go programs to the intermediate verification language Viper [2], and using available tools to verify the generated Viper program.

Gobra currently supports a large subset of Go's language features, but not all of them. The aim of this project is to introduce support for closures in Gobra.

2 Motivation

Closures are anonymous functions that can use variables declared outside of their body. As an example, consider the closure `counter` defined in Figure 1. `closure` has a signature, on line 2, declaring that the closure takes no arguments and returns an integer. The body is similar to the body of any standard method. This includes return statements (line 4 in the example) and recursive calls. Variables declared outside of the body can be used inside the closure. We refer to these variables as 'captured' variables. In the example, the closure `counter` captures variable `count`, which is read and updated. The closure can be invoked an arbitrary number of times. Lines 6 and 8 show invocations of the closure, which is then given as an argument to function `hof`, which might, in turn, invoke it an arbitrary number of times. Different invocations, even with the same arguments, can have different results, because the state of the captured variables can change.

Closures are used to abstract over computations, which is useful in multiple ways:

```
1   count := 0
2   counter := func() int {
3     count = count + 1
4     return count
5   }

6   x := counter()
7   assert x == 1
8   x := counter()
9   assert x == 2

10  hof(counter)
```

Figure 1: Example of closure.

```
1   func apply(nums []int, f func(*int)) {
2       for i := 0; i < len(nums); i++ {
3               f(&nums[i])
4       }
5   }

6   numbers := []int{1,2,3,4}
7   times2 := func(x *int) { *x = 2 * *x }
8   apply(numbers, times2)
9   assert numbers[0] == 2 && numbers[1] == 4
```

Figure 2: Usage of closure in higher order function.

```
1   func partial(f func(int, int) int, a int) func(int) int {
2       f_partial := func(b int) int {
3               return f(a, b)
4       }
5       return f_partial
6   }
```

Figure 3: Partial application.

- Encapsulating computation to avoid code duplication, as shown in example 1. In the example, a call to `counter` is equivalent to increasing `count` and returning its value. In general, functions and closures are useful for improving code reuse.
- Passing computation to higher order functions. Higher order functions can take closures as arguments. This is useful, for example, to apply a single function over a data structure, or for callbacks. Figure 2 shows the higher order function `apply`, which takes the closure `f` as argument and calls it on all elements of array `nums`.
- Allowing functions to return computation. Consider the partial application example in Figure 3. The function `partial` receives a closure `f` as an argument and returns a new closure `f_partial`, which is the partial application of `f`, with the argument `a`. This is an example of how a function can return computations (in the form of a closure).

3 Context

Wolff et al. [3] introduced support for closures in a verifier for programs written in Rust (Prusti). In their work, they introduced a set of specification primitives for closures, and a Viper encoding suitable to verify them. Such primitives include, among others:

- Specification entailment. A higher order function can specify which conditions a closure, that is given as an argument, must satisfy so that it can be called. For instance, consider the specification entailment in Figure 4 - note that the code in the figure is written in Go to help with readability, but the aforementioned paper deals with Rust. In the example, there is a higher order function `hof` that receives, as an argument, a closure `f`, which is used inside its body. The precondition for `hof` restricts the closure instances that can be passed to it. Intuitively, the precondition of any instance must be weaker than specified, while the postcondition must be stronger. Closure `cl`, which is passed to `hof` at line 11, is suitable because `x >= 10` entails `x >= 0` (weaker precondition) and `result >= 0 && ...` entails `result >= 0` (stronger postcondition).
- Call descriptions. The specification of higher order functions can include information on how the result of a closure call is used. Consider the call description in figure 5. Here, the function `foo`, which takes closure `f` as an argument, specifies on line 2 that `f` is called with argument 42, and that the result of `foo` is the same as the result of this call to `f`. The expression `outer(result)` references the result of function `foo`.

The specification primitives introduced in the paper can be used as a starting point for the specification and verification of closures in Gobra. However, Gobra has some key differences compared to Prusti, which make it impractical to apply the same approach directly. The differences are presented in 3.1, and the resulting challenges are discussed in 3.2.

```
1   requires f |= (x) { requires x >= 10; ensures result >= 0 }
2   ensures result >= 0
3   func hof(f func(int) int) {
4       return f(10) + f(11)
5   }

6   cl := requires x >= 0
7         ensures result >= 0 && result % 2 == 0
8         func (x int) int {
9                 return 2 * x
10        }

11  y := hof(cl)
```

Figure 4: Specification entailment.

```
1   requires f |= (i) { requires i == 42 }
2   ensures f(42) {} ~~> { outer(result) == result }
3   func foo(f func() int) int {
4       return f(42)
5   }
```

Figure 5: Call description.

3.1 Permissions

The approach presented in the Prusti paper mentioned above cannot be applied directly to Gobra. The reason is that Prusti exploits the guarantees provided by Rust's type system, which are not provided by Go. To exemplify this, compare the specification of a simple function in Gobra (figure 6) with the specification of an equivalent function in Prusti (figure 7). The function adds values `*y` and `*x`, and writes the result to `*x`. This means that `x` requires write permissions, and `y` requires read permissions. Such permissions are implied by the Rust type system in figure 7 (`x` has type `&mut i32` and `y` has type `&i32`). On the contrary, additional annotations are required in Gobra (lines 1 and 3 in figure 6). In particular, permissions in Gobra can be a fractional amount between 0 and 1, where 0 means that there are no permissions, any positive value means that there are read permissions and 1 means that there are write permissions.

3.2 Challenges

To summarise the paragraph above, function calls (and closure calls) in Prusti cannot fail due to missing permissions, because the correct permissions are entailed by the Rust typesystem. Instead, Gobra requires specific permission annotations to ensure that calls do not fail. Therefore, introducing the verification of closures in Go causes new challenges, including:

- Extending the specification primitives to take argument permissions into account.
- Changing the encoding of closure specification and closure calls, in order to adapt them to the semantics of the new specifications.
- Handling the variables captured by closures. Unlike Rust, Go does not provide guarantees on the usage of captured variables. In particular, a closure and its captured variables can be accessed at the same time.

B. Weber, in their Master's thesis [4], investigated the verification of closures in Python programs, allowing user-defined permissions. The approach presented in the thesis is to require the user to

```
1   requires acc(x, 1/1) && acc(y, 1/2)
2   requires *x >= 0 && *x <= 100 && *y >= 0 && *y <= 100
3   ensures acc(x, 1/1) && acc(y, 1/2)
4   ensures *x == *y + old(*x)
5   func add(x *int, y *int) {
6       *x = *y + *x
7   }
```

Figure 6: Function specification in Gobra, requiring permission annotations.

```
1   #[requires(*x >= 0 && *x <= 100 && *y >= 0 && *y <= 100)]
2   #[ensures(*x == *y + old(*x))]
3   fn add(x: &mut i32, y: &i32) {
4       *x = *y + *x
5   }
```

Figure 7: Function specification in Prusti, requiring no permission annotations.

specify, for each closure call, a contract that the invoked closure satisfies. A contract consists of the preconditions and postconditions of a closure call, including access permissions. Each contract has a name, which is used to identify it. The thesis introduces a new assertion primitive to express that a closure satisfies a contract. To establish such an assertion, the user has to write a proof that the closure satisfies the contract. The work presented in the thesis offers a solution for enabling specification entailments. However, the solution presented has some limitations. One relevant limitation is that their approach does not deal with captured variables. Furthermore, the thesis does not handle call descriptions.

4 Core goals

- **Sketch a baseline approach for the verification of closures.** For this goal, we will sketch how the approach for the verification of Rust programs [3] can be applied in Go directly. With such an approach, the permissions in the pre- and postconditions of a closure are determined by the signature and the body of the closure. In particular, the user cannot specify permissions. The purpose of this goal is to get into the topic of closure verification, and to understand the limitations that this basic approach has.
- **Support for specification entailments.** We will develop techniques to support specification entailments where permissions are specified by the user. The techniques presented in [3] and [4] can be used as a starting point, but we will have to extend such techniques to overcome the limitations outlined in 3.2. Achieving this goal involves designing new annotations at the Gobra level, and creating an encoding at the Viper level.
- **Support for call descriptions.** We will develop techniques to support call descriptions. Analogously to the previous goal, the techniques presented in [3] can be used as a starting point. However, we will have to adapt call descriptions to Gobra's setting, as they should support user-specified permissions.
- **Implementation.** We will implement our newly-developed techniques for the specification and verification of closures in Gobra. The objective is having a well-documented and maintainable implementation that is suitable to be merged into Gobra's main repository.
- **Evaluation of the final solution and its implementation.** We will evaluate our newly-developed techniques for the verification and specification of closure, and the corresponding implementation. This involves finding relevant example programs, and analysing the performance, capabilities, and limitations of the solution. Relevant examples could be taken, for example, from the Prusti repositories, or other verification literature, and translated to Gobra. Potential metrics for evaluation are the verification time, and the average number of specification lines needed. The limitations of the approach can be compared to any other approach that has been considered, and to other tools with similar functionalities (e.g. existing verifiers for other languages).

5 Extension goals

- **Support pure closures.** The aim of this goal is to add support for *pure*, i.e. terminating and side-effect-free, closures to Gobra. Pure closures have the benefit that they can be invoked within specifications. Wolff et al. [3] show how such pure closures can be used to specify higher order functions performing complex aggregations (e.g. the `fold` function). Wolff et al. present techniques to handle pure closures for Rust. We will adapt such techniques to Gobra's setting, with support for user-specified permissions.

- **Specification entailments and call descriptions for interface methods.** Go's interfaces are named collections of method signatures. Gobra already supports interfaces. However, specification entailments and call descriptions might be useful for interfaces too. The aim of this goal is to investigate whether adding the new primitives for interfaces is beneficial, and to implement them if they are useful.

- **Support for Go's `defer` statement.** We will add support for Go's `defer` statements to Gobra. `defer` statements allow to defer the invocation of functions or closures. Deferred calls are executed at the end of the enclosing function, in reverse order. Handling deferred calls is non-trivial in general. In particular, defer statements can occur in loop bodies. This case is hard to reason about because it results in a potentially unbounded number of deferred calls.

- **Add termination checks for closures.** Gobra supports termination checks for Go methods [5]. The aim of this goal is to extend the existing approach in order to support termination checks for closures.

References

[1] 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.

[2] 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.

[3] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers, "Modular specification and verification of closures in rust," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 5, no. OOPSLA. New York, NY, USA: ACM, oct 2021.

[4] B. Weber, "Automating modular reasoning about higher-order functions," Master's thesis, ETH Zürich, 2017.

[5] C. Xuan, "Verifying termination of go programs," Bachelor's thesis, ETH Zürich, 2021.