Method-specific encodings for Gobra structs

Bachelor's Thesis Project Description

René Čáky
Supervised by Dionysios Spiliopoulos, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich
Zürich, Switzerland

March 2023

Introduction

Gobra

Gobra is an open-source automated verifier of correctness and security properties for Go programs. It is based on the Viper verification infrastructure and uses separation logic. Go is a popular systems programming language designed by Google that has great advantages for the development of scalable concurrent programs thanks to its built-in concurrency primitives, such as channel-based communication. This combination of features poses interesting challenges for static verification, which Gobra aims to address. Gobra supports a large subset of Go. Its implementation translates annotated Go programs into the Viper intermediate verification language and uses an existing SMT-based verification backend to compute and discharge proof obligations. Gobra also allows developers to add requires and ensures clauses to their code as contracts. Requires clauses specify preconditions that must be true at the beginning of a function, while ensures clauses specify postconditions that must be true at the end of a function. By using contracts, developers can formally specify the intended behavior of their code. Fig. 1 computes the sum of the first n integers in Gobra using pre- and post-conditions to specify the function's behavior.

```
requires 0 <= n // precondition
ensures  sum == n * (n+1)/2 // postcondition
func sum(n int) (sum int) {
  sum = 0

  invariant 0 <= i && i <= n + 1
  invariant sum == i * (i-1)/2
  for i := 0; i <= n; i++ {
    sum += i
  }
  return sum
}
```

Figure 1: Sum of first n integers in Gobra

Viper

Viper is a verification infrastructure for permission-based reasoning developed at ETH Zurich. It consists of an intermediate language and corresponding verifiers that prove correctness of Viper programs. It provides native support for reasoning about program state using permissions or ownership, similar to separation logic. Viper can be used to implement verification techniques for sequential and concurrent programs with mutable state, as well as encode verification problems manually. Viper has several other front-ends besides Gobra for Go, like Prusti for Rust and Nagini for Python. Viper is also used in various research projects and teaching institutions worldwide. Viper provides 2 backends: Silicon and Carbon.The Carbon backend uses a technique called verification condition generation.The Silicon backend, on the other hand, uses symbolic execution.Both backends rely on the Z3 SMT solver to check the validity of the proof obligations.The choice of which backend to use depends on the nature of the program being verified and the verification goals. Similar to Gobra, Viper uses pre- and post-conditions to

specify the intended behavior of the code. Fig. 2 shows an example of a method, which computes the sum of the first n integers in Viper.

```
method sum(n: Int) returns (res: Int)
  requires 0 <= n
  ensures  res == n * (n + 1) / 2
{
  res := 0
  var i: Int := 0;
  while(i <= n)
    invariant i <= (n + 1)
    invariant res == (i - 1) * i / 2
  {
    res := res + i
    i := i + 1
  }
}
```

Figure 2: Sum of first n integers in Viper

Motivation

When it comes to developing software with Gobra, one potential issue to consider is the implementation of structs. Gobra's struct syntax can result in suboptimal code when compiled to Viper. The resulting Viper code may contain a significant amount of inefficient code that could be optimized for a faster verification.

It's natural to want to optimize verification to ensure it runs as efficiently as possible. Faster verification times can make a big difference in the interactivity of the verifier. A current limitation of Gobra is that the encoding of some features can create overhead in the verification time, even if the whole encoding is not needed for a specific function. This presents the opportunity to investigate whether different encodings for some features can be used in a way that will speed up verification. To that end, we will focus on how structs are implemented in Gobra and compiled to Viper code. By analyzing and optimizing struct-based codes, we can not only reduce the verification time of Gobra programs but also gain insight on whether the use of different encodings for the same feature is a valid optimization strategy.

In Viper translation of structs, there are several opportunities for alternative encodings. For instance, Fig. 3 includes axioms that express the equality of the memory footprints of two two-field structs. However, if we don't compare the equality of struct field locations in Gobra verification, this part of code may not be needed. Observing the struct encoding, we see that these axioms would be triggered even in situations, where they are not useful. That's why we believe this optimization would have an impact on the performance. To utilize this optimization, it would be necessary to perform syntactic checks to determine if the injectivity axioms are needed. One potential approach for syntactic checking is to verify if an address of a struct field is accessed, though more precise syntactic checks could also be explored during the process.

```
axiom {
    (forall x: ShStruct2[T0, T1] ::
      { (ShStructget0of2(x): T0) }
      (ShStructrev0of2((ShStructget0of2(x): T0)): ShStruct2[T0, T1]) == x)
  }

  axiom {
    (forall x: ShStruct2[T0, T1] ::
      { (ShStructget1of2(x): T1) }
      (ShStructrev1of2((ShStructget1of2(x): T1)): ShStruct2[T0, T1]) == x)
  }

  axiom {
    (forall x: ShStruct2[T0, T1], y: ShStruct2[T0, T1] ::
      { (eq(x, y): Bool) }
      (eq(x, y): Bool) ==
      ((ShStructget0of2(x): T0) == (ShStructget0of2(y): T0) &&
      (ShStructget1of2(x): T1) == (ShStructget1of2(y): T1)))
  }
```

Figure 3: Equality axioms

As we examine the struct definition, we notice that there are several get functions that return the fields of the struct. We might be able to optimize this code by replacing these individual functions with a single function that uses an offset to return a concrete field of a struct. Which may seem more complex but it could provide performance benefits for large structs. The map encoding mentioned on page 35 of Robin Sierra's master thesis is almost exactly what we will try to use. Similarly, the code includes multiple rev functions that could be replaced by a similar approach. Both of these functions can be seen in Fig. 4.

```
domain ShStruct5[T0, T1, T2, T3, T4]  {

  function ShStructget0of5(x: ShStruct5[T0, T1, T2, T3, T4]): T0

  function ShStructget1of5(x: ShStruct5[T0, T1, T2, T3, T4]): T1

  function ShStructget2of5(x: ShStruct5[T0, T1, T2, T3, T4]): T2

  function ShStructget3of5(x: ShStruct5[T0, T1, T2, T3, T4]): T3

  function ShStructget4of5(x: ShStruct5[T0, T1, T2, T3, T4]): T4

  function ShStructrev0of5(v0: T0): ShStruct5[T0, T1, T2, T3, T4]

  function ShStructrev1of5(v1: T1): ShStruct5[T0, T1, T2, T3, T4]

  function ShStructrev2of5(v2: T2): ShStruct5[T0, T1, T2, T3, T4]

  function ShStructrev3of5(v3: T3): ShStruct5[T0, T1, T2, T3, T4]

  function ShStructrev4of5(v4: T4): ShStruct5[T0, T1, T2, T3, T4] }
```

Figure 4: Get and rev functions

However, this change raises questions about how the interface of functions that will be using either of the two get functions. If a function that still uses the original get functions calls a function that uses the offset get function, the calling function will need to determine which function to use. Similarly, if a function that uses the offset get function calls a function that still uses the original get functions, the calling function will need to decide which approach to use. Having different encodings for the same feature can create issues with compatibility, consistency, and maintenance. This is because different parts of the codebase may be using different encodings, which can lead

to errors or unexpected behavior. A possible solution to this issue is to have two different contracts per function stored in Gobra. One contract would be used when the function is called, and the other when the function is being verified. These two contracts should be equivalent.

Goals

Core Goals

- Implement a struct encoding in Gobra that does not rely on injectivity axioms or rev functions
- Implement a simple syntactic check to verify if a struct field's address is accessed, to determine the necessity of the equality axioms
- Implement a new encoding method for structs in Gobra that replaces the numerous get and rev functions with just two functions, one for each, that utilize an offset
- Evaluate the performance of the implementations
- If not proven beneficial, extend Gobra to be able to handle different encodings for some features simultaneously by creating various equivalent contracts to be used in different situations

Extension Goals

- Explore different, more precise syntactic checks to determine the most suitable encoding
- Explore more function-specific encodings

References

[1] "A Tutorial on Gobra." [Online]. Available: https://github.com/viperproject/gobra/blob/master/docs/tutorial.md

[2] "A Tour of Go." [Online]. Available: https://go.dev/tour/list

[3] " Robin Sierra's master thesis." [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Robin$_S ierra_M A_R eport.pdf$

[4] "Viper Tutorial." [Online]. Available: https://viper.ethz.ch/tutorial/