ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Method-specific encodings for Gobra structs

Bachelor Thesis

René Čáky

September 5, 2023

Advisors: Prof. Dr. Peter Müller, Dionisios Spiliopoulos
Department of Computer Science, ETH Zürich

Abstract

Gobra is a program verifier designed for the Go language. Its workflow involves translating Gobra programs into the Viper intermediate verification language and subsequently performing the verification process on this intermediate representation. Given that the verification of large programs can introduce some peformance bottlenecks, it is desirable to address them. The objective of this bachelor thesis is to optimize the verification process for Gobra programs by improving the encoding of Gobra structs into Viper. The next objective is to extend Gobra to be able to handle alternative encodings for Gobra features concurrently, thereby further enhancing the versatility and efficiency of the verification process.

Contents

Chapter 1

Introduction

When it comes to software development, the significance of program correctness cannot be overstated, as it guards against the introduction of bugs. This important role is undertaken by verification languages, designed precisely for this purpose. Furthermore, when dealing with extensive programs, the performance of verification introduces a crucial aspect to consider.

Go [3], a widely used open-source programming language, was created by Google. It boasts built-in concurrency primitives, providing a significant advantage in building scalable concurrent programs. At ETH Zürich, a verification language named Gobra [15] was developed, aimed at ensuring the correctness of Go programs predominantly through adding method preconditions and postconditions to the Go code. The verification process of Gobra consists of translating the code into an intermediate verification language called Viper [9] , also crafted at ETH Zürich. Viper, serving as an infrastructure for permission-based reasoning, leans on the capabilities of SMT solvers to validate proof obligations. The SMT solvers are automated tools for solving logical formulas.

When it comes to developing software with Gobra, one potential issue to consider is the implementation of structs. Structs are collections of variables of different types, also called struct fields. 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 com-

piled to Viper. 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.

Moreover, our objective is to enhance Gobra's capabilities by ensuring its proper handling of various encodings. This entails enabling the integration of methods that utilize diverse encodings for specific features. Leveraging distinct encodings under specific conditions can result in improved verification performance. Thus, the potential benefits are substantial. However, a challenge arises when a method employing a specific encodings is called inside of a method that uses different encoding, potentially leading to compatibility issues. This solution to this issue is a part of this thesis' efforts.

Chapter 2

Background

This chapter establishes the fundamental groundwork for deepening undestanding of the topics investigated within this bachelor thesis. It explains the relevant aspects of the programming languages Go, Gobra, and Viper. Furthermore, it explores structs and their functionalities in Go, acknowledging their importance as a central theme.

2.1 Go

Go [3] is a popular systems open-source 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. These properties make Go an excellent tool for addressing challenges introduced by multithreaded program execution.

2.2 Structs

In Go, a struct is a composite data type that contains a collection of fields, each with a specific type. They provide a convenient way to group related data together. An example of a struct in Go is demonstrated in listing 1. Struct literals in Go allow us to create instances of a struct by directly speci-

```
1 type Vertex struct {
2         X int
3         Y int
4 }
```

Listing 1: Example of Go Struct

fying the values for its fields, as depicted in listing 2. If we omit the values

of certain fields during initialization, Go will automatically assign default values to those fields. This behavior is demonstrated in lines 2 and 3 of listing 2.

```
1 func main () {
2 v1 = Vertex{1, 2}
3 v2 = Vertex{X: 1} // Y:0 is implicit
4 v3 = Vertex{} // X:0 and Y:0
5 }
```

Listing 2: Example of Struct Literals in Go

Let's now explore the functionalities of Go structs. Firstly, we can access a field of a struct using a dot notation, such as v.X = 4, where v is a struct of type Vertex, and this assignment statement sets the value of the X field.

Secondly, in Go, we can create pointers to structs by using the ampersand (&) symbol, as illustrated in listing 3 on line 3. Conversely, we can dereference the pointer without using the asterisk (*) symbol as we would in the C language, as shown in line 4 of the same example.

```
1 func main() {
2         v := Vertex{1, 2}
3         p := &v
4         p.X = 1e9

6 }
```

Listing 3: Pointers to Structs in Go

2.3 Gobra

Gobra [15] is an open-source automated verifier of correctness and security properties for Go programs. The verification of Go programs relies on the Viper infrastructure and uses separation logic [10]. In Gobra, which supports a substantial portion of Go, the verification process involves translating the Go program into the Viper intermediate language. Afterward, the program goes through a verification procedure utilizing an existing SMT-based verification backend, which is a tool for solving logical formulas. Gobra also allows developers to add contracts for the functions that can formally specify the intended behaviour of the program. The contracts consist of pre- and postconditions. `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.

Additionally, Gobra offers support for assert statements, allowing developers to verify conditions at specific points in their program. Moreover, it includes loop invariants, assume statements, and inhale and exhale statements, providing powerful tools for program analysis, verification, and data handling.

```
1  type pair struct {
2    left, right int
3  }

5  // incr requires permission to modify both fields
6  requires acc(&x.left) && acc(&x.right)
7  ensures acc(&x.left) && acc(&x.right)
8  ensures x.left == old(x.left) + n
9  ensures x.right == old(x.right) + n
10 func (x *pair) incr(n int) {
11   x.left += n
12   x.right += n
13 }
```

Listing 4: Gobra example program illustrating main Gobra functionalities

Listing 4 from Gobra tutorial [4] shows an example of a program written in Gobra. The program defines a struct type called pair with two integer fields: `left` and `right`. The program also defines a method called `incr` on the `pair` type. The method takes an integer parameter n and is associated with a pointer receiver (denoted by `*pair`). This means it operates on a mutable instance of the `pair` struct. The method has certain pre and post-conditions. The precondition specifies that the method requires write access `(acc(&x.left) && acc(&x.right))` to both the `left` and `right` fields of the x instance. This means that the caller must have permission to modify both fields. The `ensures` clauses specify the guarantees that hold after the method is called. It ensures that the write access permissions are maintained `(acc(&x.left) && acc(&x.right))`. Additionally, it ensures that the left field of x is updated correctly `(x.left == old(x.left) + n)` by adding the value of n to the original value. Similarly, it ensures that the right field of x is updated correctly `(x.right == old(x.right) + n)` by adding the value of n to the original value.

In summary, the program defines a method `incr` that takes an integer and increments both the `left` and `right` fields of a `pair` struct by that value, while enforcing permission requirements and ensuring the expected behavior of the modifications.

2.3.1 Permissions and Heap-Manipulation

Gobra utilizes implicit dynamic frames [13] to facilitate reasoning about mutable heap data. Heap locations can be considered as addresses of pointers and each heap location is linked to an access permission.. The ability to access a pointer x is represented by the accessibility predicate acc(x). Gobra also introduces fractional and quantified permissions. When dealing with fractional permission, each permission amount is represented as a rational number in interval from 0 to 1. Complete permission (acc(x, write)) is necessary for modifying the value of a heap location, while any non-zero permission amount suffices for reading from a heap location. When specifying permissions to a number of heap location we can use quantified permissions. The approach involves placing the resource assertions within the body of a forall quantifier, which is a fundamental concept in Gobra.

2.3.2 Predicates in Gobra

Predicates are parameterized assertions marked using the keyword `pred`. The number of parameters for a predicate is arbitrary. Their bodies consist of assertions that can only use these parameters as variable names. Recursive predicate definitions are supported, enabling them to represent permissions of recursive heap structures like linked lists and trees. Listing 5 presents a recursive predicate that grants access write permissions to the value and pointer of the next node for all nodes within a linked list.

```
1  type node struct {
2    value int
3    next *node
4  }

6  pred list(ptr *node) {
7    acc(&ptr.value) && acc(&ptr.next) && (ptr.next != nil ==> list(ptr.next)
         )
8  }
```

Listing 5: Gobra program with a Function-like Recursive Predicate

2.3.3 The Difference between Functions and Methods

In Gobra, functions and methods are both marked using the keyword `func`. Their distinction lies in the arrangement of arguments. Functions feature a listing of all arguments following the function name. Conversely, methods incorporate an initial argument called the receiver preceding the method name, with the remaining ones succeeding it. This pattern is evident in the

example provided in listing 6, where `add` is a function and `incrby` operates as a method.

Furthermore, Gobra encompasses both pure functions and pure methods. This mirrors the distinction between regular functions and methods, but the inclusion of the term `pure` indicates that their bodies consist of simple `return` statement with a pure expression.

```
1  type node struct {
2    value int
3    next *node
4  }
5  func add (a int, b int) int {
6  return a+b
7  }

9  func (ptr *node) incrby (a int){
10 ptr.value += a
11 }
```

Listing 6: Function and Method in Gobra

2.3.4 The Difference between Function-Like and Method-Like Predicates

To discern between predicates, we apply a comparable approach used in distinguishing functions from methods. If the predicate definition contains a receiver, it denotes a method-like predicate; whereas, the absence of the receiver signifies a function-like predicate. In reference listing 5, the predicate is labeled as a function predicate. Furthermore, the shift from this function predicate to a method predicate is observable in reference listing 7.

```
1  pred (ptr *node) list () {
2    acc(&ptr.value) && acc(&ptr.next) && (ptr.next != nil ==> ptr.next.list
         ()
3  }
```

Listing 7: Method-like Predicate in Gobra

2.4 Viper

Viper [9] is a verification infrastructure for permission-based reasoning developed at ETH Zurich. It consists of an intermediate verification language and corresponding verifiers that prove correctness of Viper programs. Viper can be used to implement verification techniques for sequential and concur-

rent programs with mutable state, as well as encode verification problems manually. Viper has several other front-ends besides Gobra for Go, like Prusti [7] for Rust and Nagini [8] for Python. Viper is also used in various research projects and teaching institutions worldwide. Viper provides 2 backends: Silicon [2] and Carbon [1]. 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. Viper uses pre- and postconditions and loop invariants to specify the intended behavior of the program.

```
1  field f: Int

3  method client(a: Ref)
4    requires acc(a.f)
5  {
6    set(a, 5)
7    a.f := 6
8  }

10 method set(x: Ref, i: Int)
11   requires acc(x.f) && x.f < i
12   ensures acc(x.f) && x.f == i
13 {
14   x.f := i
15 }
```

Listing 8: Viper example program

The program in listing 8 from Viper tutorial [6] defines a field `f` of type Int and includes two methods: `client(a: Ref)` and `set(x: Ref, i: Int)`. The client method takes a reference a as a parameter and requires that `a.f` (the f field of object a) is accessible. Within the client method, it invokes the set method with a as the first argument and 5 as the second argument, which sets `a.f` to 5. It then updates `a.f` to 6 using the assignment operator :=.

The `set` method takes a reference x and an integer i as parameters. It requires access to `x.f` and that `x.f` is less than i. The method ensures that after its execution, `x.f` remains accessible and its value becomes equal to i. It accomplishes this by updating `x.f` to i using the assignment operator :=.

In summary, the provided Viper program demonstrates the manipulation of the f field of objects through the `set` and `client` methods. The program includes preconditions and postconditions to ensure the correctness of the program according to the specified requirements.

2.4.1 Method and Function Calls in Viper

A particularly interesting aspect related to the verification process in Viper, to be later explored in this thesis, involves the fact that when verifying a method or function call, Viper only considers the pre- and postconditions of the respective calls.

2.4.2 Domains

Domains in Viper provide the means to define additional types, mathematical operations, and fundamental principles that establish their properties, which is the role of axioms. Structurally, domains consist of a name and a section within which various domain function declarations and axioms can be introduced. The functions stated within a domain possess a global scope: they can be employed in any other part of the Viper program. Domain functions are devoid of preconditions. They can be employed in any state. Additionally, they are abstract, meaning they lack a specified body. The approach to attributing significance to these functions that lack clear interpretation is by means of domain axioms.

Domain axioms also possess a global influence. They establish characteristics of the program that are presumed to be true in all states. As axioms are not limited to specific states, they must be well-defined across all states. Consequently, domain axioms cannot reference the values of heap locations or permission levels. Normally, domain axioms are conventional assertions in first-order logic, often quantified. Axiom names are optional and utilized solely to enhance the program's readability. Listing 9 shows an example of a simple Viper domain. Equally, there exist domains in Gobra with the same syntax and functionalities.

```
2  domain MyDomain {
3    function foo(): Int
4    function bar(x: Bool): Bool

6    axiom axFoo { foo() > 0 }
7    axiom axBar { bar(true) }
8    axiom axFoobar { bar(false) ==> foo() == 3 }
9  }
```

Listing 9: Viper example Domain

Chapter 3

Current Struct Encoding

This chapter offers insight into the current translation of Go structs into Viper, supporting a wide range of struct functionalities in Go. The encoding primarily facilitates an automated conversion of structs from Gobra into Viper.

Presently, the usage of structs in Gobra may result in the generation of two potential domains in Viper, contingent on the circumstances. If struct literals are employed in the program, a Tuple domain will be created. Conversely, using pointers to a struct will generate a Shared Struct domain.

3.0.1 Shared Struct domain

Let's examine the translation of Gobra structs to Viper with a concrete example. Consider the Gobra program in listing 10 that defines a struct called Person with two string fields: `FirstName` and `LastName`.

```
1 type Person struct {
2     FirstName string
3     LastName string
4 }
```

Listing 10: Gobra Struct example

When we utilize this struct in a Gobra program that undergoes verification under the circumstances mentioned above resulting for the struct to be stored on the heap, it generates the corresponding Shared Struct domain in the Viper program shown in listing 11 . It's crucial to emphasize that regardless of the quantity of such structs with 2 fields present in the Gobra program, a single Shared Struct will be formed in the Viper program, representing all of them. Equally if there is a struct with more or less fields,

an adequate Shared Struct domain will be created with the same amount of fields. In this domain, we observe that for each field, there exists a pair of functions: a get function and a rev function. The get function of the first field with a two-field Shared Struct domain as an input will return the first field, while the rev function of the first field given the first field as an input will return the Shared Struct that this field is part of. The rev functions play a crucial role in defining the injectivity axioms. Injectivity [11] refers to the property of a function where distinct inputs yield distinct outputs. The injectivity axioms define the injectivity of every field of a Shared Struct, with injectivity being a property that ensures that for each location of a field there only exists one struct that has that field. Meaning it cannot happen that 2 different Shared Structs have a field with the same location. Additionally, in every Shared Struct domain, there is an equality axiom, which states that two structs are equal if and only if all fields at a specific position are equal.

```
1  domain ShStruct2[T0, T1] {

3    function ShStructget0of2(x: ShStruct2[T0, T1]): T0

5    function ShStructget1of2(x: ShStruct2[T0, T1]): T1

7    function ShStructrev0of2(v0: T0): ShStruct2[T0, T1]

9    function ShStructrev1of2(v1: T1): ShStruct2[T0, T1]

11   axiom {
12     (forall x: ShStruct2[T0, T1] ::
13       { (ShStructget0of2(x): T0) }
14       (ShStructrev0of2((ShStructget0of2(x): T0)): ShStruct2[T0, T1]) == x)
15   }

17   axiom {
18     (forall x: ShStruct2[T0, T1] ::
19       { (ShStructget1of2(x): T1) }
20       (ShStructrev1of2((ShStructget1of2(x): T1)): ShStruct2[T0, T1]) == x)
21   }

23   axiom {
24     (forall x: ShStruct2[T0, T1], y: ShStruct2[T0, T1] ::
25       { (eq(x, y): Bool) }
26       (eq(x, y): Bool) ==
27       ((ShStructget0of2(x): T0) == (ShStructget0of2(y): T0) &&
28       (ShStructget1of2(x): T1) == (ShStructget1of2(y): T1)))
29   }
30 }
```

Listing 11: Shared Struct domain with 2 fields

3.0.2 Tuple domain

If we work with struct literals that are stored on the stack, a Tuple domain will be created in the Viper program. This domain exhibits specificity in terms of the number of fields, requiring a new Tuple domain for verification purposes for each distinct number of fields. To illustrate this, listing 12 presents an example of such a Tuple domain.

```
domain Tuple2[T0, T1] {

  function get0of2(p: Tuple2[T0, T1]): T0

  function get1of2(p: Tuple2[T0, T1]): T1

  function tuple2(t0: T0, t1: T1): Tuple2[T0, T1]

  axiom getter_over_tuple2 {
    (forall t0: T0, t1: T1 ::
      { (tuple2(t0, t1): Tuple2[T0, T1]) }
      (get0of2((tuple2(t0, t1): Tuple2[T0, T1])): T0) == t0 &&
      (get1of2((tuple2(t0, t1): Tuple2[T0, T1])): T1) == t1)
  }

  axiom tuple2_over_getter {
    (forall p: Tuple2[T0, T1] ::
      { (get0of2(p): T0) }
      { (get1of2(p): T1) }
      (tuple2((get0of2(p): T0), (get1of2(p): T1)): Tuple2[T0, T1]) == p)
  }
}
```

Listing 12: Tuple domain with 2 fields

We can see the get functions, which are the same as in the Shared Struct domain and an additional tuple function that takes all of the fields as an input and returns a Tuple instance. In this domain there are 2 axioms. The first axiom `getter_over_tuple2` states that for any two values `t0` and `t1` of types `T0` and `T1` respectively, if we create a Tuple2 instance using these values `(tuple2(t0, t1))`, then retrieving the 0th element of this instance using the `get0of2` function should yield the value `t0`, and retrieving the 1st element using the `get1of2` function should yield the value `t1`. The second axiom `tuple2_over_getter` states that for any Tuple2 instance p, if you retrieve the 0th element of p using the `get0of2` function, and you retrieve the 1st element of p using the `get1of2` function, then creating a new Tuple2 using these two retrieved values should result in the same Tuple2 instance p.

Chapter 4

Optimizations

The central objective of this thesis is to explore alternative encodings of Gobra structs to Viper, potentially improving the overall performance of the verification process. This chapter presents two primary approaches that were explored to achieve the aforementioned goal. The first approach focuses on eliminating rev functions and injectivity axioms within the Viper Shared Struct domain, if they are not necessary for the verification of the program. The second approach involves replacing multiple `get` and `rev` functions in the Shared Struct and Tuple domain with individual functions that leverage an offset mechanism. These strategies were implemented and evaluated to optimize the encoding and subsequently enhance the efficiency of verification procedures.

4.1 Eliminating Rev Functions and Injectivity Axioms

As demonstrated in chapter 3, within the current Shared Struct domain, there exist numerous rev functions and corresponding injectivity axioms. However, in some cases, the requirement of injectivity may not be necessary for the program verification process. Consequently, we can eliminate the rev functions and the injectivity axioms from the Shared Struct domain, especially when their presence is unnecessary. This optimization becomes particularly beneficial when dealing with large Shared Struct domains that possess numerous fields, therefore numerous rev functions and injectivity axioms. By removing the rev functions and injectivity axioms, we can significantly reduce the size of the codebase, potentially leading to improved performance during the verification process.

While not all situations might allow for the application of the new encoding, Chapter 5 will demonstrate that using multiple diverse encodings simultaneously within a single codebase is indeed achievable.

Necessity of the Injectivity Axioms

Let's consider an example where the injectivity axioms are required to verify a function in Gobra. listing 13 illustrates such a scenario. The function f takes two pointers to a struct and then compares the adresses of the x field of these structs and tries to prove that these structs are the same struct. In this case, it becomes crucial to have a syntactic check that can identify the need for injectivity axioms to decide if the new encoding is suitable. In this case it needs to recognize that the original encoding must be utilized. This check is necessary to maintain the integrity of the encoding and ensure the accuracy of the verification process.

```
1  type A struct {
2  x int
3  y int
4  }

6  func f (z *A, w *A) {
7      if (&z.x==&w.x)
8          { assert (z==w)}
9  }
```

Listing 13: Example of a Gobra program, where injectivity axioms are needed

Evaluation

For the evaluation, I focused on assessing the optimization primarily on the programs extracted from the VerifiedScion GitHub repository, specifically the slayers module. VerifiedScion [5] is a project developed at ETH Zurich by Programming Methology Group for verifying the SCION Next-Generation Internet architecture. It serves as a good benchmark due to its engagement with large structs. The table below presents the obtained results. The values presented in the table represent the average of five test runs. Employing multiple runs for testing enhances the dependability of result consistency. Through these multiple runs, variations arising from external factors like system load are smoothed out, resulting in a more precise depiction of the overall performance.

The slayers module took around 100 seconds to verify using the original Gobra implementation. With the optimization, we observed an improvement of approximately 3 seconds. Furthermore, when verifying an example function of the dataplane.go file, the optimization resulted in an improvement of around 1 second, which is a speedup of 1,67%

To further examine the effectiveness of the optimization, I created a test pro-

gram that extensively utilizes the features targeted for optimization. This program contains large structs and numerous field accesses within a function. The optimization yielded an improvement of approximately 5 seconds for this particular test program, which is a speedup of 3,97%

In the final scenario, I created programs that include a single large struct along with a corresponding function. This struct consists of approximately 60 fields in the first program. Similarly, I created a second program with around 85 fields. The corresponding function repeatedly accesses fields of the struct and assigns values to them. However, there is still a very small improvement.

	Original Gobra [s]	**Optimization [s]**
slayers module	100.2	97.2
dataplane.go- example function	66.8	65.7
test program	125.8	121
struct with 60 fields	79.5	76.5
struct with 85 fields	186.8	183

Conclusion

The evaluation reveals that the impact of the optimization scales with the size of the testing programs. However, when considering the overall size of the programs, the benefits of the optimization become negligible. Moreover, to ensure the proper functioning of this optimization, it would be necessary to implement a syntactic check for the abstract syntax tree. This check would determine whether the rev functions and injectivity axioms are essential for the verification of a given program and assess the feasibility of applying this optimization. Nevertheless, as the enhanced encoding didn't appear to yield a significant speedup, I refrained from diving deeper into this syntactic check.

4.2 Replacing Multiple Get and Rev Functions with Offset-Based Functions

Depending on the circumstances, a Gobra struct can lead to the generation of two domains: Shared Struct and Tuple. In the previous section, we explored an optimization exclusively on the Shared Struct domain, while in this section, we focus on both domains. Our objective is to replace multiple get and rev functions with offset-based functions in the Shared Struct domain, whereas in the Tuple domain, we exclusively replace get functions, since there aren't any rev function inside of the Tuple domain. We saw the current implementation of both of these domains in chapter 3.

4.2.1 Tuple Domain

For the new encoding we are going to use a slightly changed idea from Robin Sierra's master thesis [12]. In listing 14 we can see the final looks of the new encoding.

```
1  domain Struct {

3    function struct_loc(s: Struct, m: Int): Int
4  }

6  domain StructOps[T] {

8    function default_tuple(l: Int): Struct

10   function struct_gettup(l: Int): T

12   function struct_lengthtup(x: Struct): Int

14   function struct_settup(s: Struct, m: Int, t: T): Struct

16   axiom axiom3 {
17     (forall m: Int ::
18       { (default_tuple(m): Struct) }
19       m == (struct_lengthtup((default_tuple(m): Struct)): Int))
20   }

22   axiom get_set_0_ax {
23     (forall s: Struct, m: Int, t: T ::
24       { struct_loc((struct_settup(s, m, t): Struct), m) }
25       (struct_gettup(struct_loc((struct_settup(s, m, t): Struct), m)): T)
             ==
26       t)
27   }

29   axiom get_set_1_ax {
30     (forall s: Struct, m: Int, n: Int, t: T ::
31       { struct_loc((struct_settup(s, n, t): Struct), m) }
32       m != n ==>
33       struct_loc(s, m) == struct_loc((struct_settup(s, n, t): Struct), m))
34   }
35 }
```

Listing 14: Replacement for the Tuple domain

The given program snippet introduces two new domains, namely Struct and StructOps[T], which serve as replacements for the original Tuple domain. This restructuring allows for more organized handling of struct variables. The Struct domain is responsible for defining the type for all struct variables, providing an unified approach. On the other hand, the StructOps[T] domain

contains the necessary operations and functionalities related to structs. This approach brings about an improvement by consolidating multiple Tuple domains with varying numbers of fields into just two domains.

The program adds a layer of indirection by adding the `struct_loc` function in the Struct domain, which takes a struct and a field index and maps it to an location, which is an integer. Instead of passing the field index as an input of the get function, we pass it this location and return the value that is stored at that location. When setting a value, the program specifies that the locations do not change, instead of specifying that the values don't change, which ensures that the get function will still return the same value. The axioms describe the behavior of the StructOps operations: `struct_gettup` and `struct_settup`.

The `get_set_0_ax` axiom states that if we set the mth field of a struct s using `struct_settup(s, m, t)` and then retrieve the mth element of that struct using `struct_gettup(struct_loc(struct_settup(s, m, t), m))`, then we should get back t. This essentially means that setting the mth element of s to t using `struct_settup` correctly updates the state of s.

The `get_set_1_ax` axiom states that if we set the nth field of a struct s using `struct_settup(s, n, t)` and then try to access the mth element of that new struct using `struct_loc(s, m)`, we should get the same result as if we had first created a struct s' without setting the nth element of s to t using `struct_settup(s, n, t)`, and then accessed the mth element of s' using `struct_loc(s', m)`. However, this only holds if m is different from n. This means that updating one element of the Struct using `struct_settup` should not affect the location of other elements in the Struct.

Two additional functions have been introduced to enhance the original StructOps[T] domain developed by Robin Sierra. These functions provide extended functionality and augment the capabilities of struct handling. The first function, default_tuple, serves as a partial replacement for the original tuple function of the Tuple domain. It accepts the length of the struct as input and generates a struct accordingly. The second function, `struct_lengthtup`, enables retrieval of the length of a struct. To establish the behavior and functionality of these functions, the `axiom3` axiom has been introduced. This axiom precisely defines the underlying principles and operations of `default_tuple` and `struct_lengthtup`.

Let's illustrate the process of struct creation under the new encoding using an example, since it is slightly different than the using the original encoding, in contrast to other applications of functions which are similar. Previously, if we wished to create a struct with two integer fields, such as 0 and 42, we could simply use the `tuple2(0, 42)` syntax. However, with the updated encoding, this method is no longer applicable. Instead, we now employ a more general approach to accommodate structs of varying lengths. For instance,

to create a struct of length 2 with fields 0 and 42, we would utilize the following syntax: `struct_settup(struct_settup(default_tuple(2), 0, 0), 1, 42)` In this new encoding, we first initialize a struct of length 2 using `default_tuple`, which sets the initial values of the struct's fields. Then, we utilize the `struct_settup` setter function to modify specific fields within the struct.

4.2.2 Shared Struct Domain

We have previously observed the original encoding of a Shared Struct domain, as shown in listing 11. In order to enhance the Shared Struct domain, I have decided to re-use the two separate domains idea for replacing the Tuple domain as can be seen in listing 15. I retained the existing Struct domain from this concept and refer to it as ShStruct to overcome overlapping issues. As for the other domain, which will contain the necessary operations for ShStruct, I called it ShStructOps[T]. In contrast to the previous approach where we had individual rev and get functions for each field of a struct, I have now consolidated them into two functions: `struct_rev` and `struct_get`. The `struct_get` function operates based on an offset value, which is passed indirectly using the `shstruct_loc` function.

To assist in defining the axioms, I have introduced the `struct_length` function. As we are now utilizing an offset for the get function, it is essential to determine the maximum offset to define the axioms in the most compact way possible. This approach allowed me to substitute the original Shared Struct domain, which could have varying numbers of fields and numerous instances with different field numbers, with just these two new domains. Furthermore, we can now create a single rev function since the domain definition only includes one field, T.

We define the axioms in a similar manner to the original domain, but we utilize the new functions. This provides the advantage of summarizing the potentially numerous injectivity axioms, particularly when dealing with large structs. I achieved this by employing a forall statement on the offset and establishing boundaries for that offset. The equality axiom also becomes more concise, especially when working with large structs, thanks to the same technique I used for the injectivity axioms.

Evaluation

During the testing phase of this optimization, my focus was primarily directed towards examining the performance on large structs, as they are expected to benefit the most from this new encoding technique. The test cases conducted were the same as those employed for the first optimization discussed in this bachelor thesis, with the exception of excluding the router and

```
domain ShStruct {

  function shstruct_loc(s: ShStruct, m: Int): Int
}

domain ShStructOps[T] {

  function struct_get(l: Int): T

  function struct_length(x: ShStruct): Int

  function struct_rev(v: T): ShStruct

  axiom {
    (forall x: ShStruct, l: Int ::
      { (struct_get(shstruct_loc(x, l)): T) }
      l >= 0 &&
      (l < (struct_length(x): Int) &&
      (struct_rev((struct_get(shstruct_loc(x, l)): T)): ShStruct) == x))
  }

  axiom {
    (forall x: ShStruct, y: ShStruct ::
      { (eq(x, y): Bool) }
      (eq(x, y): Bool) ==
      ((struct_length(x): Int) == (struct_length(y): Int) &&
      (forall l: Int ::l < (struct_length(x): Int) &&
        (l >= 0 &&
        (struct_get(shstruct_loc(x, l)): T) ==
        (struct_get(shstruct_loc(y, l)): T)))))
  }
}
```

Listing 15: Replacement for the Shared Struct domain

slayers module from the tests. It is worth noting that the results revealed a noticeable increase in verification time when using the optimized encoding compared to the original approach.

	Original Gobra [s]	**Optimization [s]**
dataplane.go - example function	64.1	66.2
test programs	125.8	138.9
struct with 60 fields	79.5	86.8
struct with 85 fields	186.8	199.9

Conclusion

The evaluation results clearly demonstrate that the optimized encoding performs worse than the original encoding. Surprisingly, despite eliminating certain domain functions, particularly the get functions replaced by a single

function with an offset, no significant performance benefits were observed. This unexpected outcome suggests that the issue may lie in the repeated access of a single domain when multiple structs are involved.

A closer analysis reveals that when only one struct is present, as observed in the test programs with large structs, the performance difference between the optimized and original encodings is smaller. However, even in this scenario, there remains a performance loss compared to the original encoding.

In light of these findings, it appears that the primary advantage of this encoding lies in its ability to produce more compact programs rather than significantly improving performance. The optimization may still find relevance in cases where program readability is prioritized over runtime efficiency.

Chapter 5

Supporting Different Encodings for Some Features Simultaneously in Gobra

5.1 General Idea

Despite the extensive optimization efforts and the creation of new encodings from Gobra to Viper, it has become evident that the performance benefits of the optimizations presented until now are not significant. Therefore, the focus of this bachelor thesis now shifts towards addressing another crucial aspect. One of the main challenges lies in the fact that different functions within a codebase may utilize different encodings for the same feature, based on specific circumstances or requirements to ensure the most optimal verification. This situation becomes problematic when one function calls another function that employs a different encoding for the same feature. To illustrate this issue, let's consider the pseudocode example in listing 16.

```
2  requires A
3  ensures B
4  func f {
5  some body of the function
6  } //encoding 1

8  requires C
9  ensures D
10 func g {
11 f()
12 } // encoding 2
```

Listing 16: Pseudocode illustrating the problem of having 2 functions using 2 different encodings

Let's consider a scenario where function f uses Encoding 1 for a certain feature, while function g uses Encoding 2 for the same feature. The problem arises when g calls f within its body, but their encodings are not compatible. This means that the preconditions and postconditions of f are not compatible with g, assuming the encodings are incompatible.

To address this issue, we can let the function f use Encoding 1 since it would better suited for its verification, because there would normally be a program choosing the most suitable encoding for the function f. However, to ensure compatibility with g, we create a modified version of f, denoted as f_, which is abstract. The new function can be abstract, since the body of the function f got already verified in the function definition and because of the fact that Viper only cares about pre- and postconditions when verifying a function call.[11] That's why instead of verifying the entire body of f, we focus solely on the preconditions and postconditions of f, meaning we call the function f_ with modified pre- and postconditions of f, called A_ and B_ inside of the body of g instead of calling f. These preconditions and postconditions are encoded using Encoding 2, aligning them with the encoding used by g.

By separating the verification process for f and g, we can maintain the desired encodings for each function while ensuring compatibility between their preconditions and postconditions. This approach allows us to address the challenge of using different encodings within a codebase, enabling the verification of functions that rely on different encodings for the same feature. The pseudocode in Listing 17 illustrates the proposed implementation described above.

```
1  requires A
2  ensures B
3  func f {
4  some body of the function
5  } //encoding 1

7  requires A_
8  ensures B_
9  func f_ // abstract function f encoded with encoding 2

11 requires C
12 ensures D
13 func g { f_() } // encoding 2
```

Listing 17: Pseudocode illustrating the problem

Additionally, the issue of encoding compatibility extends to other functionalities within Gobra programs, such as methods, pure functions, pure methods, function predicates, and method predicates. We apply a similar approach as we described above with functions, with a minor distinction. For

methods, we employ the same methodology as we did for functions. However, there is a slight variation when it comes to pure methods, pure functions, method-like predicates, and function-like predicates.

One challenge with pure functions and pure methods is that their calls are considered expressions rather than statements, as in the case of function calls. This means they can be nested within other expressions, such as ”1 + return(),” where the pure function return() returns a numerical value. As Viper requires the return value of these pure functions, when generating a new version of the function in the program due to encoding mismatches, we cannot eliminate the function body. The same applies to predicates when utilizing the unfold statement, as we need to ascertain the contents of the predicate.

5.2 Coding the Concept into Reality

In order to demonstrate the functionality of the general idea, we will assume a specific encoding for the slice feature based on Zdenek Snajders' bachelor thesis [14]. This approach leads to two possible encodings for the slice feature in Go. Since these circumstances may require the presence of identical functions in the program, it becomes necessary to distinguish between them in the Viper program. To achieve this, we assign them unique names that clearly indicate the encoding used.

To transform the general concept into a functional implementation, it is necessary to annotate each function, method, and predicate that are nodes in the abstract syntax tree with the appropriate annotation that describes the encoding it utilizes. Our goal is to transform the original abstract syntax tree by restructuring it and generating a new abstract syntax tree as the resulting output. This process involves two main steps.

Firstly, we utilize Zdenek Snajdr's syntactic check, which assigns annotations to Nodes based on the encodings they employ. To achieve this, we create new instances of the nodes that require annotation and then substitute them within the abstract syntax tree.

In the rest of this chapter, we will further refer to functions, methods, pure functions, pure methods, function predicates, and method predicates as ”members.” In the second pass, we provide the modified abstract syntax tree to the program transformation. Here, we further modify the member names by adding encoding annotation, as well as adjust their bodies to ensure they call the appropriate members, considering the name changes. If a member called by another function does not use the same encoding and has not yet been created with the correct encoding, we create new members with the appropriate encoding, either abstract or not, depending on the type of the member, in order to maintain consistency. By following this approach,

we effectively process the abstract syntax tree, ensuring that all members align with the appropriate encodings while preserving the integrity of the original structure.

The program follows the following process: We traverse each node of the abstract syntax tree, performing a case distinction. If the node type is the member we defined above, we first extend the name to indicate the encoding it uses. Then we call a function that checks if there is a corresponding function with the correct encoding in the program for every member call within a member's body. If no such function exists, we create a new member with the same preconditions, postconditions and termination measures possibly with an empty body depending on the type of the member. Subsequently, we invoke another function to transform statements within the member's body, such as function calls, by adding the corresponding encoding annotation. This transformation involves appending an encoding annotation to the corresponding call.

Additionally, it is necessary to transform and inspect member calls in preconditions, postconditions, and termination measures, if they are part of a member. This is important because these sections may contain member calls that need to be accounted for. I also extended the functionalities of domains in Gobra to make them compatible with the extensions described in this chapter. The process of translating Gobra domains into Viper involves altering their names and introducing a default function that outputs the domain type. In cases where domain function calls appear in the Gobra program, these calls are expanded by appending an encoding annotation in both the program and the domain structure. When additional encodings are required for a specific function, the domain is expanded by incorporating all domain functions with the new encoding. The axioms are then recreated utilizing the functions from the new encoding. Importantly, the default function retains its original form, thereby preserving the domain's name as well.

To conduct thorough testing, I devised a comprehensive Gobra program that encompasses various scenarios for member call placements. During the first pass of the abstract syntax tree transformation, I randomly assigned encoding values to the member annotations as a replacement for Zdenek Snajdr's syntactic check. Subsequently, I manually inspected the resulting Viper program, ensuring that the encoding annotations remained consistent throughout.

Chapter 6

Future Work

The optimization concepts explored in this bachelor thesis proved to be ineffective, and it appears that further improvements cannot be made. Nevertheless, we successfully extended Gobra to be able to support simultaneous handling of different encodings for a certain feature. This extension holds promise for potential future use and can be readily expanded to accommodate additional encodings per feature or even multiple encodings for different features.

The primary objective of the extension was centered around creating precise annotations for different encodings. Nevertheless, a crucial task remains when applying this functionality in practical scenarios. Should a new method-specific encoding be introduced, my developed program would exclusively annotate the corresponding parts of the Gobra program, enabling effective differentiation between the encodings. This approach inherently permits the exploration of new encodings and their combined integration.

Bibliography

[1] Carbon. https://github.com/viperproject/carbon/. [Online; accessed 7-August-2023].

[2] Silicon: A Viper Verifier Based on Symbolic Execution. https://github.com/viperproject/silicon. [Online; accessed 7-August-2023].

[3] The Go Programming Language. https://go.dev. [Online; accessed 9-June-2023].

[4] Tutorial on Gobra. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md. [Online; accessed 9-June-2023].

[5] VerifiedSCION. https://github.com/viperproject/carbon/. [Online; https://www.pm.inf.ethz.ch/research/verifiedscion.html].

[6] Viper Tutorial. https://viper.ethz.ch/tutorial/. [Online; accessed 9-June-2023].

[7] Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 88–108. Springer, 2022.

[8] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the*

Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I, volume 10981 of *Lecture Notes in Computer Science*, pages 596–603. Springer, 2018.

[9] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[10] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[11] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill, 2011.

[12] Robin Sierra. Verification of Ethereum Smart Contracts Written in Vyper. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Robin_Sierra_MA_Report.pdf, 2019. [Online; accessed 9-June-2023].

[13] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[14] Zdenek Snajdr. Optimization of Slice Encoding in Gobra. [not yet published].

[15] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs (extended version). *CoRR*, abs/2105.13840, 2021.

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):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

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

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)**

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