Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Optimization of Slice Encoding in Gobra

Bachelor's Thesis

Zdenek Snajdr

May 2023

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

Abstract

Gobra is a deductive verifier for Go. It translates annotated Go code into the Viper intermediate verification language. Slices are a common data type in Go. The current way slices are translated by Gobra induces overhead for functionality that is not leveraged by all methods or functions. As a result, specific handling of slices in those methods provides a performance gain. This thesis introduces a new method-specific encoding for slices in Gobra. It can be used for methods without slicing expressions. The aim is to mitigate performance problems in Gobra. Finally, the syntactic check assures the use of the right encoding for each method.

Contents

Contents **iii**

1 Introduction **1**

2 Background **3**
2.1 Go 3
2.1.1 Slices 3
2.2 Viper 5
2.2.1 Domains 5
2.2.2 Contracts 5
2.2.3 Permissions 6
2.2.4 Inhaling and Exhaling 7
2.3 Gobra 8
2.3.1 Permissions in Gobra 9

3 Understanding The Existing Slice Encoding **11**
3.1 Design 11
3.1.1 Arrays as Basis 11
3.1.2 Injectivity 12
3.2 Specification 13
3.2.1 Domain 13
3.2.2 Initialization and Declaration 14

4 Proposed Slice Encoding **15**
4.1 Objectives 15
4.2 Design 15
4.2.1 Representation of Slices 16
4.2.2 Creating Slices 16
4.2.3 Injectivity 16
4.3 Specification 17

4.3.1 Domain 17
4.3.2 Slice Creation 18

5 Evaluation **19**

6 Syntactic Check for Slice Encoding **23**
6.1 Design 23
6.2 Implementation 24
6.3 Further Optimization 24

7 Future Directions **27**
7.1 Granularity 27
7.2 Method-specific Encoding for Other Data Types 28

8 Conclusion **31**

A Code Appendix **33**

Bibliography **37**

Chapter 1

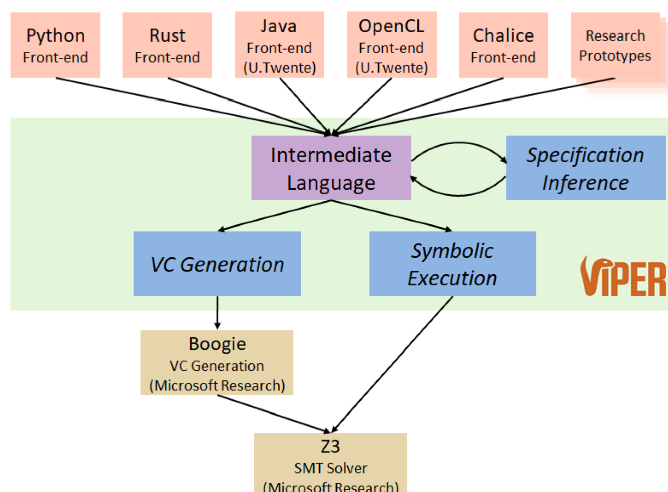
Introduction

Program correctness is the central goal of formal verification of software programs. The verification process has to be not only sound but also fast. The latter is especially of importance when dealing with large code bases.

Gobra [16] is an automated deductive verifier [7] for the programming language Go [4]. Automated verification using Gobra makes ensuring the correctness of Go programs easier. Verification is carried out by translating Go programs into the Viper intermediate verification language [8], which is part of the Viper verification infrastructure (Figure 1.1). Furthermore, this infrastructure contains two verification back-ends; both leverage the Z3 SMT solver [3]. Since the verification relies on Viper, functionality and behavior of Go programs have to be transferred faithfully. Such translation is maintained by the encoding process that defines how data types, functions, and other parts are translated. One way to make the verification faster is to adapt the translated code to the needs of specific functions or methods. As a result, the same data type can be represented through different Viper code depending on context, reducing the workload for the SMT solver. For simplicity reasons, functions and methods will be referred to as members in the rest of the thesis.

Slices are a commonly used data type in Go for a variety of tasks, such as manipulating and processing portions of data stored in large arrays. Despite their simple representation in Go, there are many ways to encode them in Viper; the main constraint being equivalent behavior and functionality to the original program. Since not every member requires all the features, the translated Viper code can leverage this by proper encoding based on context. The current implementation supports only one slice encoding that covers all the features of slices in Go. The way this encoding represents slices in Viper leads to verification overhead for members that only use a subset of the functionality of slices. Having a specialized encoding for slices based on the context and needs can speed up the verification and prevent timeout.

Figure 1.1: Viper infrastructure (Source: Viper Tutorial [15])



In the thesis an alternative simpler slice encoding for members that do not contain slicing expressions is introduced. The members which require the complete functionality will still use the existing encoding. Syntactic check of the program before the translation assures the use of suitable encoding for each member. The introduction of a specialized encoding promises less time spent in the Z3 Theorem Prover, the SMT solver used by Viper.

The first step was to design a function-specific encoding based on the requirements and the existing encoding. We prototyped and tested the design before implementing it in Scala. After the implementation of the encoding was done the next step was to evaluate the speed up and soundness. The final phase was to create a syntactic check to allow for further development and work with the simpler encoding.

The thesis is divided into eight chapters. Chapter 2 summarizes the technical background essential for the thesis. Chapters 3 to 4 introduce the current and the proposed encoding for slices in Gobra. Chapter 5 examines the evaluation of chosen programs in the proposed encoding against the current one; Chapter 6 focuses on a syntactic check for slice encodings and finally Chapter 7 discusses future directions of encodings in Gobra.

Chapter 2

Background

This chapter provides a technical background for the thesis. Section 2.1 describes the basics of slices in Go and Section 2.2 introduces code structure and heap access permissions in the intermediate verification language Viper. Last Section 2.3 comments on some basic features of Gobra. In this chapter, a substantial portion of the content has been sourced and adapted from the Viper and Gobra tutorial [15][6].

2.1 Go

The Go programming language [4], also known as Golang, is an open-source high-level programming language developed at Google. It was designed with simplicity, efficiency and reliability in mind to tackle problems introduced by multicore processors, networked systems, massive computation clusters, etc.

2.1.1 Slices

Slices are an important feature of Go offering a flexible way to work with dynamically-sized parts of arrays. A slice consists of a pointer to the underlying array, length, and capacity. A slice itself does not store any data but rather serves as a pointer to certain part of an array. For this reason, changes in the slice modify also the corresponding elements of the underlying array and vice versa, and these changes are visible to other slices which share the same part of the underlying array.

The length of a slice represents the number of elements currently accessible within the slice and the capacity indicates the maximum number of elements that the slice can hold without the need to resize. A slice might be empty, i.e. the length is zero. The value of such slice is `nil`.

An array in Go always has a fixed size whereas a slice supports dynamic resizing which allows to append or remove arbitrary number of elements. This might be one of the reasons why in practice, slices are more common than arrays. A slice can be initialized and declared directly or it can be implicitly created from an array by slicing. The following code listings show both variants.

```go
arr := [5]int{1, 2, 3, 4, 5}
s1 := arr[1:4]
s2 := arr[:3]
s3 := arr[3:]
```

Listing 1: Slicing in Go

`arr` is an array with five elements and `s1`, `s2` and `s3` are slices created implicitly by slicing `arr`. Any array can be sliced by specifying the inclusive lower and the exclusive upper bound, separated by a colon (`arr[low:high]`). However, one may omit the lower or upper bound and their default values will be used instead. The default values are zero and the length of the slice respectively.

```go
    s1 := make([]int, 3, 5)
    s2 := make([]int, 5)
    s3 := []int{1, 2, 3}
```

Listing 2: Direct slice declaration in Go

Another way to declare a slice is through a slice literal `[]t{elems}`. `t` represents the member type and `elems` the elements accessible within the slice. The expression `make([]t, len, cap)` declares a slice with the length `len` and optionally capacity `cap` with default elements of the type `t`. When no value for `cap` is specified, the value of `len` is used. A slice can be resized by specifying new length (if still within capacity) or by simply slicing again with new bounds.

```go
slice1 := []int{1, 2, 3, 4, 5}
slice1 = append(slice1, 20, 21)

slice2 := []int{4,5,6}
slice1 := append(slice1, slice2...)
copy(slice1, slice2)
```

Listing 3: Append and copy

Two very useful methods are `append(slice, elems)` and `copy(dest, src)`. The method `append` appends an element or a slice at the end of the current slice. The dots, '`...`', after `slice2` are necessary when appending the elements of one slice to another. `copy` copies data from `src` to `dest`. The method returns the number of elements copied.

Other operations such as retrieving and setting a value of a slice element work the same way as for arrays and are not mentioned here.

2.2 Viper

Viper [8] is a verification infrastructure which contains Viper intermediate verification language as well as two verification backends; this language is designed to simplify the verification of programs that involve a mutable state. It uses a form of separation logic [11] to provide built-in support for reasoning about program state. Viper has a syntax similar to C, and it supports both sequential and concurrent programming paradigms. Viper supports writing specifications and contracts, which makes it easy to verify programs using formal methods. It is mainly used as an intermediate verification language between a frontend (e.g. Gobra [16], Prusti [1], etc.) and an SMT encoding. The goal of Viper is to improve the overall reliability and security of programs by reducing the risk of bugs or vulnerabilities.

2.2.1 Domains

Domains in Viper allow the introduction of additional types not innate to Viper. Functions and axioms are used to define their properties. A domain consists of a name for the new type and a block where function declarations, also called domain functions, and axioms can be defined. Functions declared in a domain are global and can be used anywhere in the Viper program. These domain functions are always abstract, lacking a defined implementation body, and cannot have preconditions; hence, they can be applied in any state. The meaning of these domain functions is derived from domain axioms, which are global and establish properties assumed to be true in all states. Domain axioms should be well-defined across all states and must not reference heap values or permission amounts. Typically, they are expressed as first-order logic assertions, often involving quantification [15].

2.2.2 Contracts

Contracts in program verification refer to formal specifications that provide means to specify the intended behavior or properties of a member. There are three main types of contracts commonly used in program verification:

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

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

Listing 4: Domain in Viper

preconditions, postconditions, and invariants. A precondition specifies the conditions that must be satisfied for a member to be called. Further reasoning about the program behavior is based upon assumptions defined by preconditions. A postcondition describes the expected behavior or properties that should hold true after returning from a member call, e.g. it can describe the desired output. The third most common type of contract is an invariant. An invariant is a property that remains true throughout the execution of a program or a specific code section. Invariants are necessary for reasoning about loops, data structures, or critical sections of code.

2.2.3 Permissions

Viper is used for verifying heap-manipulating programs and employs implicit dynamic frames [12], a variant of separation logic [11], to support reasoning about mutable heap data. In Viper, every heap location is linked to an access permission, which has to be transferred between members during calls and returns through contracts. Access to a location or member invocation is only possible if the appropriate permission is held [15]. The permission to access a heap location `x` is denoted by `acc(x)`. This accessibility predicate is not duplicable, i.e. `acc(x)` does not entail `acc(x) && acc(x)`, where `&&` is separating conjunction.

When a method reads from or modifies heap locations, it requires access permissions in its precondition. This guarantees that the caller must transfer permissions to the method. The access permissions in the postcondition ensure that the they are returned upon returning from a call. `acc(x)` is semantically equal to `acc(x, write)`. Viper offers fractional permissions, a rational number between 0 and 1. Non-zero means read and 1 means write access. An exclusive permission (`write`) to a heap location can be held at most once. Fractional permission to the same heap location can be held more than once are summed up, however, the permission amount to a location can never exceed 1.

Permissions can also be quantified with the `forall` quantifier like in Listing 6. The general form of this quantifier is `forall [vars] :: [triggers]`

```
1  method set(x: Ref, i: Int)
2    requires acc(x.f) && x.f < i
3    ensures acc(x.f) && x.f == i
4  {
5    x.f := i
6  }
```

Listing 5: Pre- and post-conditions in Viper

`A` where `[vars]` is a list of comma-separated declarations of variables which are being quantified over, `[triggers]` consists of trigger expressions in curly braces, and `A` is a Viper assertion potentially including the quantified variables. The purpose of the triggers is to inform the SMT solver to instantiate the quantifier only when it encounters expressions of forms matching the trigger. This allows for element-wise specification of data structures such as arrays or graphs. Listing 6 models a binary tree, i.e. each node has at most two children. The preconditions provide permission to access the `first` and `second` fields of all nodes `n`.

```
1  field first : Ref
2  field second : Ref

4  method inc(nodes: Set[Ref], x: Ref)
5    requires forall n:Ref :: { n.first } n in nodes ==>
6      acc(n.first) && (n.first != null ==> n.first in nodes)
7    requires forall n:Ref :: { n.second } n in nodes ==>
8      acc(n.second) && (n.second != null ==> n.second in nodes)
9    requires x in nodes
10 {
11   var y : Ref
12   if(x.second != null) {
13     y := x.second.first // permissions covered by preconditions
14   }
15 }
```

Listing 6: Quantified permissions in Viper

2.2.4 Inhaling and Exhaling

Permissions to heap locations can also be managed by `inhale` and `exhale` statements. The process of gaining permission (which happens in the callee), is called inhaling permissions; the opposite process of losing permission (in the caller) is called exhaling. Both operations update the amount of held permissions. From the caller's perspective, permissions required by a precondition are removed before the call, and permissions guaranteed by a postcondition are gained after the call returns. Conversely, from the callee's

perspective, the opposite occurs.

```
1  field f: Int

3  method set_inex(x: Ref, i: Int) {
4    inhale acc(x.f)
5    x.f := i
6    exhale acc(x.f)
7  }
```

Listing 7: Inhale and exhale in Viper

Inhaling and exhaling is not only used for permissions, moreover, constraints on values may also be specified. Viper allows explicit exhaling or inhaling via the statements `exhale A` and `inhale A`, where `A` is a Viper assertion such as `acc(x.f) && i < x.f`. Inhaling `A` means gaining permissions required by `A` and also assuming that the constraints in `A` hold; exhaling `A` means asserting that the constraints in `A` hold and giving up permissions expressed by `A`. From a caller's perspective, pre- and post-condition in Listing 5 can be seen as syntactic sugar for appropriate exhale and inhale statements before and after a call to the member.

2.3 Gobra

Gobra [16] is a modular, deductive program verifier [7] for Go. It was developed by the Programming Methodology Group at ETH Zurich and serves as a frontend for Viper, translating Go code annotated with contracts and assertions into the Viper intermediate verification language [8] and then using an existing SMT solver [3] to perform formal verification. Gobra supports a large subset of the Go programming language, allowing developers to leverage the benefits of formal verification directly in the existing Go code. Gobra helps to prove memory safety, crash safety and data-race freedom to prevent errors, bugs, and other issues that can lead to software vulnerabilities. This makes it a valuable tool helping developers to ensure correctness, safety, and security of their code. One of Gobra's applications is VerifiedSCION [14], a project focused on verifying the SCION Next-Generation Internet architecture [2] that aims to provide secure routing and forwarding, alongside numerous other desirable properties.

Gobra uses contracts as a basis for reasoning about code correctness. In Listing 8, a caller of the method `sum` must guarantee the preconditions and the postconditions must hold upon returning from a call. If a pre- or post-condition is not satisfied, Gobra throws an error and aborts the verification. Gobra supports deterministic boolean expressions (e.g. `x > y + z`), impli-

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

6    invariant 0 <= i && i <= n + 1 // conjoined invariant
7    invariant sum == i * (i-1)/2
8    for i := 0; i <= n; i++ {
9      sum += i
10   }
11   return sum
12 }
```

Listing 8: Gobra code

cations (`==>`), conditionals (`cond ? e1 : e2`), universal quantifiers (e.g. `forall x int :: x >= 5 ==> x >= 0`), and many other assertions. [6]

2.3.1 Permissions in Gobra

Since Gobra is a frontend for Viper, permissions in Gobra work similarly to permissions in Viper. `acc(&x.f)` can be understood as permission to the heap location `&x.f`. Permissions to heap locations are declared in pre- and postconditions which are in Gobra denoted with `requires` and `ensures` respectively. Analogously to Viper, permissions can be quantified for specifying permissions to potentially unbounded number of heap locations, as you can see in Listing 9.

```
1  requires forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/2)
2  ensures forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/2)
3  ensures isContained ==> 0 <= idx && idx < len(s) && s[idx] == x
4  func contains(s []int, x int) (isContained bool, ghost idx int) {

6    invariant 0 <= i && i <= len(s)
7    invariant forall k int :: 0 <= k && k < len(s) ==> acc(&s[k], 1/4)
8    for i := 0; i < len(s); i += 1 {
9      if s[i] == x {
10       return true, i
11     }
12   }

14   return false, 0
15 }
```

Listing 9: Function in Gobra with permissions

Chapter 3

Understanding The Existing Slice Encoding

This chapter examines the objectives, design, and specification of the current encoding of slices in Gobra.

The existing encoding of slices in Gobra supports a large subset of the features of slices in Go. The encoding process serves as a bridge between the Go programming language, which is widely used for software development, and Viper, a specialized language designed for formal verification. It aims to automate the translation process and reduce the manual effort required to verify programs written in Go. The primary goal of slice encoding is to facilitate formal verification of slices in Go by faithfully representing this data type in Viper.

3.1 Design

The existing slice encoding in Gobra is designed to faithfully encode slices and their features in annotated Go code into Viper for formal verification. This section introduces the key design points that make up the current encoding.

3.1.1 Arrays as Basis

Arrays serve as the underlying data structure for slices in Go. By using the arrays as basis for encoding slices after translation into Viper, the design maintains high compatibility with the semantics of the Go language; this also allows us to leverage the existing array representation in the Viper code for slices. Each slice stores its underlying array, length, and capacity. The latter two, length and capacity, define the part of the array that is accessible

to the slice. Information about the elements themselves is managed by its underlying array.

Translating slices directly into Viper can be challenging due to their dynamic nature and variable size. By relying on arrays as the basis, the translation process becomes more straightforward. In Viper, the concrete location a slice points to is represented as the underlying array with a fixed offset, this is the array index where the slice starts, to which the index within the slice itself is added. Consequently, the implementation of slices in Viper is based and relies on encoding of arrays in Viper. The existing encoding supports a large subset of slice properties in Go, especially all the properties discussed in Section 2.1.

3.1.2 Injectivity

In the context of verification of arrays in heap-manipulating programs, injectivity refers to a property ensuring that two different indexes of any array will *always* identify a different heap location. Injectivity plays a crucial role in program verification; it ensures that modifications or operations on a particular memory location are reflected consistently across all references to that location. Concretely for arrays and slices, if two or more arrays or slices point to the same memory location, then their corresponding elements at that location must also be the same.

For slices, injectivity guaranties that when two slices reference the same part of underlying array, modifying an element in one slice will also affect the corresponding element in the other slice. Injectivity is particularly relevant in situations where aliasing occurs, meaning that multiple variables or references point to the same memory location, e.g. when multiple slices are used to manipulate and access the same part of an array. Without injectivity, inconsistent states and unexpected behavior may arise due to inconsistent element values across aliased arrays or slices; It is hence used for maintaining data consistency and avoiding unexpected behavior.

Viper uses field permissions to reason about the heap. In order to denote permission to a potentially unbounded set of locations without prescribing a traversal order, permissions and predicates are allowed to occur under universal quantifiers (see Section 2.2.3). In addition, Viper requires for each assertion `acc(E.f)` under a `forall x:T` that `E` is injective [9].

Similarly to slices, arrays are also not innate to Viper and are therefore implemented through a domain (see Listing 10). An array access in a higher-level programming language, e.g. `arr[i]` in Go, is modeled as `loc(arr, i).val` in Viper. The `loc` function maps arrays and indices to a value of type `Ref`. Fields of these `Refs` are used to represent array slots (locations). The `val` field represents the value stored in this array slot. The `allDiff` ax-

```
1  field val: Int

3  domain Array {
4     function loc(a: Array, i: Int): Ref
5     function length(a: Array): Int
6     function rToA(ref: Ref): Array
7     function rToI(ref: Ref): Int

9     axiom allDiff {
10        forall a: Array, i: Int :: {loc(a, i)} rToA(loc(a, i)) == a && rToI
              (loc(a, i)) == i
11    }
12    axiom lengthNonneg {
13        forall a:Array :: length(a) >= 0
14    }
15 }
```

Listing 10: Encoding of Arrays in Viper

iom expresses that the `loc` function is injective [13]. Injectivity is necessary to maintain soundness for mapping from integers back to locations through the `loc` function [10].

3.2 Specification

3.2.1 Domain

The `Slice[T]` domain is the representation of Go slices in Viper. The complete `Slice[T]` domain in Viper can be found in Appendix A.

As described in 2.2.1, a Viper domain consists of functions and axioms. These functions serve as filed values that store important data about a concrete slice. The function `sarray` represents the underlying array, the functions `slen` and `scap` return the length and capacity of the slice respectively. The function `soffset` keeps the offset value from the beginning of the array. The offset is added to the array's base address to mark the beginning of the slice elements in the array. And the last function is `smake` which is used for representing each slice as a `ShArray` (analogous to Listing 10) with offset `o`, length `l`, and capacity `c`.

Domain axioms are used to attach meaning to the abstract domain functions. The properties defined in the axioms are assumed to hold universally in all states. These domain axioms assure the basic properties of slices as described in the Go documentation [5], e.g. the length of a slice is less or equal to its capacity or that the length and the offset must be at least zero. Moreover, they define properties which are connected to the `ShArray` domain. These are that the sum of offset and capacity cannot exceed the length of

the underlying array, or that each slice is represented as slicing operation of ShArray. The last group of axioms are deconstructing axioms. They define the return values of the domain functions by using the slice representation of smake.

3.2.2 Initialization and Declaration

Initially, slices have to be declared and initialized. The existing encoding supports the entire range of slice initializations innate to Go; this subsection covers how Gobra deals with slice initializations as seen in Listing 1 and Listing 2.

Slice created with the existing encoding always has an underlying array. The Viper code contains helper functions for slice initialization from an array or a slice. These functions take an array or a slice as an argument together with the lower and upper slicing bound. In the helper function the values for sarray, soffset, slen, and scap are checked and assigned. The general style of these helper functions can be observed on the example in Listing 11.

```
1 function ssliceFromArray_Ref(a: ShArray[Ref], i: Int, j: Int): Slice[Ref]
2   requires 0 <= i
3   requires i <= j
4   requires j <= (ShArraylen(a): Int)
5   ensures (soffset(result): Int) == i
6   ensures (slen(result): Int) == j - i
7   ensures (scap(result): Int) == (ShArraylen(a): Int) - i
8   ensures (sarray(result): ShArray[Ref]) == a
```

Listing 11: Function for slicing

When initializing a slice an underlying array is created, or it already exists, and is passed as an argument to one of the helper functions to create a slice from that array. The only exception in this is when declaring a slice with make as can be seen in Listing 13. On lines 14 to 16, the translated code assigns the default value, in this case 0 for int, for each valid index and on lines 9 and 10 the length and the capacity values are inhaled.

Chapter 4

Proposed Slice Encoding

This chapter will focus mainly on introducing the objectives, design, and specification of an additional, simpler, encoding for slices in Gobra.

4.1 Objectives

The existing encoding of slices in Gobra is designed to be used universally for verification of Go programs with slices. This chapter proposes and describes the prototype of a function-specific encoding for slices that is simpler and covers a subset of the existing encoding; the proposed encoding does not cover slice expressions that use slicing and it is not intended to entirely replace the existing slice encoding. The aim is to speed up the verification of members that do not use slicing. Consequently, both encodings are meant to be used in parallel within one program.

In the existing slice encoding, `Slice[T]` domain in Viper keeps track of the underlying array and the offset from the beginning of the array; the slice itself does not store any elements, rather it stores the pointer to an array and an offset at which the slice elements begin. Hence, a lot of axioms contain expressions with the underlying array and additions of slice index and array offset (see Appendix A), making the tasks for the Z3 Theorem Prover [3] more complicated. Empirically it seems that the majority of the verification time is spent in Z3, so the proposed encoding should reduce this burden by simplifying the `Slice[T]` domain along with the axioms. The concrete design choices and the specification of the simple slice encoding will be addressed in the following sections.

4.2 Design

The simple slice encoding is designed to be faster in certain applications than the existing one. The representation of slices with the simple encoding is

similar to the representation of arrays in Viper, e.g. elements are addressed directly through functions in the slice domain. The following subsections address differences between the existing and the proposed encoding.

4.2.1 Representation of Slices

The key difference, between slices in the existing and the proposed encoding, is the way they are represented. A slice in both encodings has length and capacity functions. However, contrary to the description in Section 3.1, the proposed encoding does not use an array as a basis for slices anymore. A slice representation no longer has an underlying array and an offset marking the beginning of the actual slice elements. Consequently, the slices in the proposed encoding are similar to an array representation in Viper; the elements are stored in the slice itself and are accessed via a simple index.

4.2.2 Creating Slices

This simple slice encoding is only to be used when no slicing expression is present in a member. Many members do not require slicing and simply perform some operation on elements of the slice such as sorting or filtering. Therefore, with this encoding it is not possible to create a new slice by slicing an array or an existing slice as in Listing 1. Though, it is still possible to declare a slice without an underlying array as in Listing 2. Even though slicing is not supported, arrays can be sliced in other members and the resulting slices can then be passed as arguments to members, which use the simple slice encoding.

4.2.3 Injectivity

In Section 3.1.2 was mentioned that `E` in each assertion `acc(E.f)` under a `forall x:T` quantifier must be injective. In the existing encoding this property is maintained by the encoding of the underlying array. Unfortunately, the simple encoding does not work with arrays anymore and does not have injectivity axiom. Instead, the elements are accessed directly through the slice and these direct access permissions are inhaled at the beginning of a member. Although the simple slice encoding does not contain an axiom for injectivity, when a Gobra generated Viper program should be verified the flag *assumeInjectivityOnInhale* is enabled. This flag results in assuming injectivity for `inhale` statements, in our case for inhaled quantified access assertions. This means that despite injectivity not being ensured from within the encoding, it is not an issue due to the use of this flag.

4.3 Specification

The section will introduce and explain the implementation of the design choices.

4.3.1 Domain

Listing 12 shows the `Slice[T]` domain of the simple slice encoding. One may notice that it is much simpler that the one presented in Section 3.2.1. Indeed, the requirements for the simple encoding made many axioms redundant.

Some functions from the existing slice domain are still present. The function `scap` returns the slice capacity and `slen` the slice length. Since the elements of a slice are not represented by an underlying array and an offset now, the functions `sarray` and `soffset` were replaced with `sloc`. This function takes advantage of the way elements are stored in the proposed encoding. It takes two arguments, a slice `s` and an index `i` and returns the element stored at that index. `sloc(s,i)` models `s[i]` in Go and it works analogously to `loc` function in array domain (Listing 10).

The above mentioned functions are accompanied by two axioms. The axiom `slice_len_leq_cap` states that the length is less or equal to the capacity. The second axiom `slice_len_nonneg` bounds the length to be at least zero. Both define the necessary properties of length and capacity of a slice. The fairly minimal design of the domain has two main justifications.

The first justification is the use of the simpler encoding. It is designed and intended only for members that fulfill the given criteria and accompanies the existing encoding. The goal was to make it as simple as possible without sacrificing any substantial functionality. For example, as explained in Section 4.2.3, there is no need for injectivity axiom as in the array encoding. The essential properties of slices were tested and the proposed encoding passed the test suite (see Chapter 5 for more detail).

In addition, the proposed slice encoding was not the only encoding that was considered. During the design and creation of a prototype an idea for a different slice encoding emerged. This promising slice encoding contained additional domain functions and axioms compared to the proposed encoding These domain functions were inspired by the slice domain in the existing encoding, `smake(l: Int, c: Int): Slice[T]`, and the array domain, `rToS(r: T): Slice[T]` and `rToI(r: T): Int` for the `sloc` function. The full domain can be seen in Listing 15. The function `smake(l: Int, c: Int): Slice[T]` is a modified version of the original function `smake(a: ShArray[T], o: Int, l: Int, c: Int): Slice[T]`; it represents each slice with *length* `l` and *capacity* `c`. Functions `rToS` and `rToI` are used in axiom

slice_injectivity to mimic the injectivity axiom from ShArray. Although this axiom would not be necessary, it should show the influence of a similar injectivity axiom inside the Slice domain on the verification time. This alternative slice encoding was compared against the proposed encoding discussed in this chapter. Both completed successfully all tests, however, the verification time was slightly better only in a few cases. Simpler nature and faster verification in most cases demanded the decision to stick to the simpler and faster encoding of the both.

4.3.2 Slice Creation

The simple encoding is designed to work analogously to an array. In the former encoding accessing a slice element occurred through referencing the underlying array with the desired index and the offset of the slice, i.e. where the beginning if the slice in the array is. Accessing an element in the new encoding is straight-forward and there is no offset needed since the stored elements are accessible directly by the slice. This is done by the function sloc, which takes a slice s and an integer i as arguments and returns the element at the position i.

By directly referencing the elements in the slice and not in the underlying array, we can save time during verification since we do not have to check constraints with the offset, e.g. checking that $\texttt{slen(s)} + \texttt{soffset(s)} \leq \texttt{ShArrayLen(sarray(s))}$, and reference the ShArray domain as before. The Slice[T] and ShArray domains are not coupled anymore.

Since the simple encoding does not allow slicing operations the only way to create new slices is with make or the slice literal []t{elems}. A slice does not reference an underlying array like its counterparts in the existing encoding. When a slice is initialized, its length, and sometimes also its capacity (if no argument for capacity is given, a default value is used), are given as arguments and are inhaled in the Viper code. Hence, the return value of the slen and scap domain functions is defined by the inhale during slice initialization.

As for the elements, the existing encoding used an indirect way by creating an array first or using an existing array (see Section 3.2). The simpler encoding does not rely on an array to initialize the elements. The element at index i is directly mapped to the i-th element of the sequence containing the desired element values.

Chapter 5

Evaluation

In this chapter, the proposed slice encoding described in Chapter 4 will be compared to the existing slice encoding from Chapter 3. To assess the efficacy of both encodings, they were used in translating ten evaluation programs without any slicing operations, allowing us to measure the evaluation time of the resulting Viper code.

One evaluation program is specifically designed to evaluate the correctness of one of the extension goals. Extension goals are meant to accompany the core goals and for example widen the scope of the thesis or facilitate the integration of new or modified features. This evaluation program contains expressions of the form `arr[i:][j]`. These expressions create a slice from `arr` with offset `i` first and then access the element at position `j`. As a result, the syntactic check would mark members containing such expressions. However, `arr[i:][j]` is semantically equivalent to `arr[i+j]` which does not perform any slicing operation and does not trigger marking by the syntactic check. This extension goal was introduced because it allows for wider use of the simple slice encoding. Members with such expressions can in fact be found throughout the SCION code. Since Gobra is used by `VerifiedSCION`, it is reasonable to assume that there is going to be some improvement in the verification time. The goal is to transform the said expression into a simpler form and thus allow for the use of the simpler encoding. Due to incompleteness the existing encoding does not verify this program so it is also not present in the tables below; it was only used to test the implementation of the said extension goal.

The first evaluation program in the table is designed to test the basic properties of the simple encoding, including working with `append` and `copy`. This is not so much about the verification time but rather about the overall correctness of the encoding. The next three programs are sorting algorithms, *bubble sort*, *selection sort*, and *insertion sort*, which sort a given slice in an ascending order. The main focus of these is on working with the slice elements

(*get* and *set*). Next program is a `filter` function that filters out all the elements of a given slice that do not satisfy a certain property. In this program a new empty slice is created and all the elements that satisfy the property are appended to that slice via the `append` method. The main loop of the `filter` function is contained five times in the code to make it more challenging to verify. `matrixMul` contains standard implementation of matrix multiplication, `transClosure` performs transitive closure on a graph represented as an adjacency matrix, `testFile1` consists mainly of function calls in loops, and `testFile2` focuses purely on `copy` and `append` methods in a loop. The information about verification times from ten runs can be found in the tables.

File	t_W	t_B	t_A	σ_t
properties	2691	2187	2365.5	178.88
bubbleSort	2421	2162	2287.6	74.47
selectionSort	2758	2491	2648.2	89.63
insertionSort	1918	1434	1698.4	146.33
filter	12010	11603	11792.1	115.49
matrixMul	32715	27013	29347.0	1738.71
transClosure	6266	5858	6011.0	148.67
testFile1	6042	4952	5623.9	334.41
testFile2	41020	33137	35761.1	2700.86

Table 5.1: Verification times in ms of the evaluation programs in the existing encoding. Each benchmark file is listed along with its worst and best verification time t_W and t_B respectively, the average verification time t_A and the standard derivation σ_t rounded to two decimal places.

File	t_W	t_B	t_A	σ_t
properties	1972	1621	1841.9	92.96
bubbleSort	1916	1586	1702.3	91.27
selectionSort	2282	1600	2024.4	192.78
insertionSort	1488	1033	1263.6	135.93
filter	8328	6977	7640.0	439.39
matrixMul	14024	12359	13126.9	602.47
transClosure	4205	3872	4041.0	127.33
testFile1	4847	4403	4597.3	149.62
testFile2	26661	22343	24760.9	1324.93

Table 5.2: Verification times in ms of the evaluation programs in the simple encoding. Each benchmark file is listed along with its worst and best verification time t_W and t_B respectively, the average verification time t_A and the standard derivation σ_t rounded to two decimal places.

The average times of the tables above are captured in Table 5.3 together with their difference and measured speed-up. One can see that the proposed

encoding is faster than the existing encoding by about 20 to 25 percent for programs or rather members that do not use slicing operations. Both programs containing `copy` or `append` methods in a loop, `filter` and `testFile2`, are over 30 percent faster. `matrixMul` exhibits a remarkable speed-up of 2.24, which was confirmed by a second evaluation of ten consecutive runs; there even a speed-up of 2.33 was achieved. This speed-up is a perfect example of the verification time improvement for members which contain a lot of access permissions. As expected, representing a slice more like an array rather that an underlying array with an offset is beneficial to the verification time. The verification time in the tables represents only the duration Viper requires to verify the translated code; other actions carried out by Gobra, such as translation or a syntactic check, are not taken into account. Nevertheless, the time required for a syntactic check is negligible in relation to the overall process (parsing, desugaring, etc.) for small members.

File	t_C	t_S	Difference	Speed-up
properties	2365.5	1841.9	523.6	1.28
bubbleSort	2287.6	1702.3	585.3	1.34
selectionSort	2648.2	2024.4	623.8	1.31
insetionSort	1695.4	1263.6	431.8	1.34
filter	11792.1	7640.0	4152.1	1.54
matrixMul	29347.0	13126.9	16220.0	2.24
transClosure	6011.0	4041.0	1970.0	1.49
testFile1	5623.9	4597.3	1026.6	1.22
testFile2	35761.1	24760.9	11000.2	1.44

Table 5.3: Average verification times of the evaluation programs in the complete encoding (t_C) and the simpler encoding (t_S) in ms, the difference of average verification times in ms and the measured speed-up.

All the files listed in the tables were verified successfully by Gobra. The first testing file is mainly designed to verify basic features of the proposed encoding and not for the speedup.

Gobra developers themselves created a small test suites[1], among others also for the features of slices. The simpler encoding should also be evaluated with the tests from the said test suite that do not contain subslicing expressions. There are also tests that must not be successfully verified otherwise the encoding would be proven to be unsound. The proposed simpler encoding was tested and passed all the tests that it should pass and did not verify the tests that were expected to fail.

[1] https://github.com/viperproject/gobra/tree/master/src/test/resources/regressions/features/slices

Chapter 6

Syntactic Check for Slice Encoding

Chapters 3 to 4 describe the existing as well as the modified encoding for slices in Gobra. As mentioned in Section 4.1 already, both encodings should be used concurrently to achieve verification speedup from Chapter 5 without sacrificing features of slices in Go, e.g. slicing. This chapter focuses on the introduction of a syntactic check for Gobra programs to decide which encoding should be used for each member.

6.1 Design

The goal of having a syntactic check in place for a method-specific encoding is to check that a member is free of slicing expressions. It is useful for determining which slice encoding should be used. For members that do not contain any slicing expressions, the simple encoding can be used. Otherwise the existing encoding is to be used.

The syntactic check is a transformation of the internal abstract syntax tree (AST). It traverses the tree and marks the members that contain slicing expressions. If a slicing expression is found inside a member, the corresponding node is annotated with this information. The result of the syntactic check is an AST with additional information about slicing expressions. Afterwards, when the Viper code is generated, Gobra can decide easily which encoding for slices to use based on the node annotations. This leverages the speedup of the simpler encoding for specific members.

Additionally, a Gobra code can contain expressions that might be marked as slicing expressions even though there is an equivalent expressions without slicing. The expressions are of the type `arr[i:][j]` where `arr` is an array or a slice and `i` and `j` are indices. Such an expression would be marked by the syntactic check as containing a slicing expression. However, `arr[i:][j]` is semantically equivalent to `arr[i+j]` which does not contain any slicing ex-

pression and is not marked by the syntactic check. The AST should therefore be transformed in such a way that these expressions do not cause any trouble. The syntactic check has to be performed on the modified AST where the expressions of the said type do not occur.

6.2 Implementation

Initially, expressions of the type `arr[i:][j]` must be replaced with semantically equivalent expressions `arr[i+j]`. This is done before the actual syntactic check to ensure its correctness. If one would check for slicing operation before the replacement, the syntactic check would also mark members with nodes that would not be in the final AST. The syntactic check itself then traverses the modified AST where such expressions are no longer present.

Fortunately, both the node replacement and the syntactic check can be done in one traversal of the AST. This is due to the fact that the expression to replace can be spotted before the slicing operations get checked. The expression to replace is a node of type `IndexedExp` that has a slicing operation `Slice` as a base. In the AST, the base is then a child of `IndexedExp`. This parent node is replaced with the base of the slicing operation and the access index is changed to the addition of the lower slicing index and the access index of the original indexed expression. The traversal continues on the nodes of the modified AST where the node containing the slicing operation is no longer present.

In the internal AST, `Slice` represents the slicing expression (or operation) and not the slice itself. If the syntactic check finds a slicing expressions (an `Expr` of type `Slice`), be it from array of other slices, it creates an annotation for the member node. The internal AST with these annotations is then returned. The additional information created by the syntactic check can be used to determine which slice encoding to use for each member.

6.3 Further Optimization

Each traversal of the internal AST is connected with additional cost, especially for members with large bodies. Consequently, future modified encodings for other Gobra data types and data structures should consider this overhead and try to reuse the existing syntactic check for their annotations. Any extra traversal of the tree should only be done if the current implementation does not allow otherwise.

It might be the case that a member that uses the existing slice encoding calls another member that uses the simpler encoding. Since both encoding need different contracts there should exist an interface for handling such cases. It

should use the node annotations obtained from the syntactic check of the program and be scalable to accommodate future alternative encodings.

Chapter 7

Future Directions

This thesis introduced new and simple method-specific slice encoding. Before this encoding can be used, some problems need to solved and improvements made, however, these would go beyond the scope of a bachelor's thesis. This chapter summarizes the way forward and provides some ideas about possible future directions. Topics of granularity and the chances and limitations of using similar encoding approach for other data types in Gobra will be addressed.

7.1 Granularity

The granularity of the approach is an important criterion for the potential speedup. The finer the granularity, the greater the potential for using faster encoding. But it comes with a cost in the form of overhead, e.g. for a syntactic check or other additional procedure for context switching. The simpler slice encoding introduced in this thesis is designed to be method-specific, i.e. the slice encoding is decided per member.

Apart from implementing a syntactic check to decide which encoding can be used, there also have to be different pre- and post-conditions for members that use the simpler encoding. In consequence, a deeper understanding of whether the slice encoding can be chosen in a more fine-grained manner instead of per member, is needed. Seeing the difficulties with the method-specific slice encoding, one might be tempted to answer the previous question negatively. Nonetheless, pre- and post-conditions are not the only contracts used for formal verification. There are also invariants that are crucial for loop verification. This leads the author to the idea that it might be possible to introduce an even more fine-grained slice encoding than the method-specific one.

Since loops also have invariants, one could design an encoding that imple-

ments the same kind of optimization as the simpler one, namely a simpler encoding for loops that do not use slicing operations. With this, even a member that creates slices at the beginning but does not use slicing anywhere else, especially not in its loops, would benefit from the speed-up of such encoding structure. This would be a major benefit mainly for members with many iterations and large loop bodies. Finally, the method-specific and loop-specific slice encodings could both be used within the same program.

Additionally, specialized encodings could also be used per slice variable. Instead of marking members containing slicing expressions, syntactic check would go through the program and mark slice variables that are used in slicing expressions. The contribution of such an approach is not clear since it is highly dependable on the code it is used for. For example, if a concrete slice variable is only sliced once in one method, this would force the entire slice to use the complete encoding instead of the simpler one. Whereas a method-specific encoding would do much better in this case because only the method that uses slicing would use the complete encoding. Nonetheless, if the code was optimized for such encoding, e.g. create copies of slices to use in slicing expressions to avoid marking the entire slice variable, there might be a verification speed-up.

As mentioned above, even for this simpler slice encoding one needs different pre- and post-conditions for functions using different encodings. This will likely be handled by a new interface that is being implemented in the Programming Methodology Group at ETH. Coming up with a more fine- or coarse-grained encoding would probably require comparable or even larger amount of time and resources to make it work. One would have to weigh the gain and the price to pay.

7.2 Method-specific Encoding for Other Data Types

This thesis introduced a method-specific encoding for slices in Gobra. It appears that the achieved speed-up can be mostly attributed to uncoupling of the slice and array domains. Before, the two domains were always coupled through the `sarray` function in the slice domain that stores the underlying array. This is not the case with the simpler encoding anymore; the slice domain does not keep an underlying array (cf. Chapter 4). This method-specific approach opens up the possibility for further optimizations and improvements in performance for other data types in Gobra.

There are many data types innate to Gobra. Since the optimization of slice encoding involves mainly a method-specific domain, the list was reduced to data types that are known to have a domain in the translated Viper code, i.e. *arrays* and *structs*, together with *maps* which do not have a domain but could be handled differently based on whether they are mutable or not. By

tailoring other encodings to the needs of specific methods, it may be feasible to achieve similar benefits in terms of speed-up as with slices. However, the applicability of such an approach to other data types highly depends on their characteristics in the translated Viper code.

Let's begin by considering the potential application of method-specific encoding to arrays in Gobra. Arrays are fixed-size, ordered collections of elements of the same data type. Slices are coupled to arrays in the original encoding, hence, the speed-up achieved by optimizing arrays could improve the time needed for verification of slices in the original encoding as well. Compared to the original slice domain the array domain `ShArray` is relatively small; it comprises four functions and only two axioms (cf. Listing 10). One axiom guarantees the length to be non-negative and the other is for injectivity. Due to the array domain working well with minimum functions and axioms there is not a lot of potential for a method-specific domain. One possible improvement could be to drop the injectivity axiom and corresponding functions as we did with alternative simpler encoding (cf. Listing 15).

However, dropping only the injectivity axiom in the slice domain did not result in a significant performance improvement (cf. Section 4.3.1) and a similar result could be expected for arrays. Injectivity plays a crucial role in program verification and not including it could result in unsound behavior or more imprecision in some cases. The soundness of the slice domain was evaluated both with and without an injectivity axiom and both passed all the tests. The method-specific encoding for slices is limited to such members that it does not cause any problems and the slice domain was meant to be as simple as possible. By leaving out anything that is not necessary for verification of members that do not contain slicing expressions, it was possible to drop the injectivity axiom and the corresponding functions. Ultimately, the potential negative consequences outweigh the marginal improvement in verification time.

Next data type are structs, user-defined data types that allow you to group together variables of different types under a single name. The translation of structs already employs type-specific encoding. A struct is translated as `Tuple` if it is of *exclusive* type and as `ShStruct` if it is of *shared* type. Alongside to this thesis there was another thesis focused on designing and prototyping a method-specific encoding for structs in Gobra being done at the Programming Methodology Group at ETH. From the information given by the supervisor and the colleague who works on the thesis it seems that such optimization would unfortunately not yield significant speed-up for Gobra generated Viper code with structs [17].

Lastly, maps are unordered collections of key-value pairs that provide a way to associate and retrieve values based on unique keys. Although maps

do have a generated domain they were still considered as there might be for example redundant functions in some context. The consideration of maps was motivated by Scion where mutable and immutable maps are used. Gobra does not differentiate between mutable and immutable maps, so this distinction might result in speed-up during verification. The problem with improving maps is that they do not translate to domains like structs or slices but are implemented using maps innate to Viper; each map has a Viper map field with the given types. Upon inspecting the translation of maps from Gobra to Viper it was concluded that even if there were any possible optimizations they would have to be done on Viper level. This is due to the fact that maps are directly translated to features specific to Viper and thus do not fall under optimization of Gobra.

In conclusion, no data type could be found that would benefit from a similar method-specific optimization. The approximation done on slices cannot be easily replicated for the addressed data types. Further research and experimentation would be needed to determine the viability and potential advantages of implementing method-specific encodings for other data types.

Chapter 8

Conclusion

In the thesis a simpler encoding for slices in Gobra was designed and introduced. The purpose of the modified encoding is to assist the existing encoding in the translation of slices into Viper and it is used for representing slices in members that do not need the full specification of the existing encoding. The evaluation confirmed a verification speedup with the simpler encoding. The syntactic check assures that the AST is annotated with additional information such that for each member the suitable slice encoding can be chosen. During the AST traversal certain expressions are replaced with semantically equivalent but simpler expressions for the encoding.

The simple encoding reduces the verification time spent in the Z3 solver for slices. In practice, using both the existing and the simpler encoding would result in reduction of the overall verification time for large projects.

Since the simpler slice encoding only covers a subset of the complete encoding, there are some limitations to its use. It can only be used for members that do not contain any slicing expressions or where such expressions are replaced before the syntactic check. The simpler encoding stores and accesses the elements in a different way than the complete encoding. Additionally, to use both encodings we need an interface that can leverage the results of the syntactic check and generate the contracts, member's pre- and postconditions, accordingly.

In Chapter 7 we looked into some possible future directions for encodings in Gobra. Unfortunately, the primary contribution to optimizing the complete slice encoding is the decoupling of the slice and array domains in Viper. This complicates the task of extending a similar improvement to other data types in Gobra. The results of the thesis point to possible future optimizations, be it leveraging similar optimization strategy for other data types in Gobra or investigating more fine-grained approach to the presented optimization.

Appendix A

Code Appendix

```
domain Slice[T] {
  function scap(s: Slice[T]): Int
  function slen(s: Slice[T]): Int
  function sloc(s: Slice[T], i: Int): T

  axiom slice_len_leq_cap {
    (forall s: Slice[T] ::
      { (slen(s): Int) }
      { (scap(s): Int) }
      (slen(s): Int) <= (scap(s): Int))
  }
  axiom slice_len_nonneg {
    (forall s: Slice[T] :: { (slen(s): Int) } 0 <= (slen(s): Int))
  }
}
```

Listing 12: Slice domain in the new encoding

```
1   var fn0: Slice[Ref]
2   exhale 0 <= 5 && 0 <= 8 && 5 <= 8
3   inhale (forall fn1: Int ::
4       { (ShArrayloc((sarray(fn0): ShArray[Ref]), sadd((soffset(fn0): Int),
5       fn1)): Ref) }
6       0 <= fn1 && fn1 < (scap(fn0): Int) ==>
7       acc((ShArrayloc((sarray(fn0): ShArray[Ref]), sadd((soffset(fn0): Int)
            ,
8       fn1)): Ref).val, write))
9   inhale (scap(fn0): Int) == 8
10  inhale (slen(fn0): Int) == 5
11  inhale (forall fn2: Int ::
12      { (ShArrayloc((sarray(fn0): ShArray[Ref]), sadd((soffset(fn0): Int),
13      fn2)): Ref) }
14      0 <= fn2 && fn2 < 5 ==>
15      (ShArrayloc((sarray(fn0): ShArray[Ref]), sadd((soffset(fn0): Int),
16      fn2)): Ref).val ==
17      0)
```

Listing 13: Translated Viper code of `make([]int,5,8)`

```
domain Slice[T] {
  function sarray(s: Slice[T]): ShArray[T]
  function scap(s: Slice[T]): Int
  function slen(s: Slice[T]): Int
  function smake(a: ShArray[T], o: Int, l: Int, c: Int): Slice[T]
  function soffset(s: Slice[T]): Int

  ...

  axiom {
    (forall s: Slice[T] ::
     { (sarray(s): ShArray[T]) }
     { (soffset(s): Int) }
     { (slen(s): Int) }
     { (scap(s): Int) }
     s ==
     (smake((sarray(s): ShArray[T]), (soffset(s): Int),
     (slen(s): Int), (scap(s): Int)): Slice[T]))
  }
  axiom {
    (forall s: Slice[T] ::
     { (slen(s): Int) }
     { (scap(s): Int) }
     (slen(s): Int) <= (scap(s): Int))
  }
  axiom {
    (forall s: Slice[T] ::
     { (soffset(s): Int), (scap(s): Int) }
     { (ShArraylen((sarray(s): ShArray[T])): Int) }
     (soffset(s): Int) + (scap(s): Int) <=
     (ShArraylen((sarray(s): ShArray[T])): Int))
  }
  axiom {
    (forall s: Slice[T] :: { (slen(s): Int) } 0 <= (slen(s): Int))
  }
  axiom {
    (forall s: Slice[T] :: { (soffset(s): Int) } 0 <= (soffset(s): Int))
  }
}
```

Listing 14: Slice domain

```
1  domain Slice[T] {
2    function scap(s: Slice[T]): Int
3    function slen(s: Slice[T]): Int
4    function smake(l: Int, c: Int): Slice[T]
5    function rToS(r: T): Slice[T]
6    function rToI(r: T): Int
7    function sloc(s: Slice[T], i: Int): T

9    axiom slice_constructor {
10     (forall s: Slice[T] ::
11     { (slen(s): Int) }
12     { (scap(s): Int) }
13     s == (smake((slen(s): Int), (scap(s): Int)))
14     )
15   }

17   axiom slice_len_leq_cap {
18     (forall s: Slice[T] ::
19       { (slen(s): Int) }
20       { (scap(s): Int) }
21       (slen(s): Int) <= (scap(s): Int))
22   }

24   axiom slice_len_nonneg {
25     (forall s: Slice[T] :: { (slen(s): Int) } 0 <= (slen(s): Int))
26   }

28   axiom slice_injectivity {
29     (forall s: Slice[T], i: Int ::
30       { (sloc(s, i): T) }
31       0 <= i && i < (slen(s): Int) ==>
32       (rToS((sloc(s, i): T)): Slice[T]) == s &&
33       (rToI((sloc(s, i): T)): Int) == i)
34   }
35 }
```

Listing 15: Alternative domain for slice encoding

Bibliography

[1] 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.

[2] Laurent Chuat, Markus Legner, David A. Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. *The Complete Guide to SCION - From Design Principles to Formal Verification*. Information Security and Cryptography. Springer, 2022.

[3] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] The Go Programming Language. https://go.dev.

[5] Go documentation. https://go.dev/doc/.

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

[7] Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State*

of the Art and Perspectives, volume 10000 of *Lecture Notes in Computer Science*, pages 345–373. Springer, 2019.

[8] 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.

[9] Peter Müller. Program verification, 2023. Available at https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Courses/SS2023/PV/slides/08-permissions-models.pdf.

[10] Severin Münger. Inference of pointwise specifications for heap manipulating programs. Master's thesis, ETH Zürich, 2017. Available at https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Severin_Muenger_MA_report.pdf.

[11] 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.

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

[13] Alexander J. Summers. Verification of unbounded heap data structures, 2017. Available at https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Courses/SS2017/Program%20Verification/10-UnboundedHeapDataStructures.pdf.

[14] VerifiedSCION. https://www.pm.inf.ethz.ch/research/verifiedscion.html.

[15] Viper Tutorial. https://viper.ethz.ch/tutorial/.

[16] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

[17] René Čáky. Method-specific encodings for gobra structs. Bachelor's thesis, ETH Zürich, 2023.

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):

Optimization of Slice Encoding in Gobra

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

Name(s):	First name(s):
Snajdr	Zdenek

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)
Zürich, 05.09.2023	Šnajdr

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