ETH zürich

Verification of closures for Go programs

Master's Thesis

Stefano Milizia

August 15, 2022

Advisors: Prof. Dr. Peter Müller, Felix Wolf

Department of Computer Science, ETH Zürich

Abstract

Closures are anonymous functions which are able to capture variables declared in the enclosing scope. They are a common and useful feature, present in many popular programming languages. They are a powerful tool for programmers, but it is hard to reason about them formally.

In this work, we introduce support for the formal specification and verification of closures within programs written in the popular Go language. Our work extends the modular, deductive verification tool Gobra. We introduce a design that is suitable to verify closures in several relevant scenarios. In particular, our design is capable of handling the internal state of a closure and capable of specifying the behaviour of higher-order functions. We use our approach to verify relevant programs showcasing real-life uses of closures, indicating that our design is powerful and flexible.

Acknowledgements

First of all, I would like to thank Felix Wolf for his supervision. He was always available to provide useful advice and explanations, both in our weekly meetings and via email. His observations helped the project steer in the right direction, and helped me identify the potential shortcomings of all the approaches I considered. He also provided key advice and feedback related to the presentation of my work and to the write-up of this report.

I would also like to thank Felix Wolf, again, and João Pereira for taking their time to review my contributions to Gobra's GitHub repository.

Lastly, I give my thanks to Prof. Peter Müller and to all the members of the Programming Methodology Group at ETH, for the feedback and observations that they provided at various stages of my work.

Contents

Chapter 1

Introduction

Closures are anonymous functions that can use variables declared outside of their body. They are first-class objects, so they can be assigned to variables, used as arguments to other functions, returned from other functions or stored inside data structures. Closures were first introduced as a feature of some functional programming languages [10]. Today, they are supported by many popular imperative and object-oriented programming languages, including C++, Java, Python, and Go.

Go is an increasingly popular programming language. Its focus on scalability and concurrency makes it the ideal candidate for the implementation of large-scale distributed systems. Such systems often include security-critical applications, so it is important that their implementations contain no bugs. Ensuring that such systems are bug-free, however, is often a hard problem, due to their complexity and to their extensive use of concurrency. In this context, automated deductive verifiers play an important role, as they allow us to formally prove that a program follows its intended behaviour.

Gobra [13] is a verifier for Go programs. Gobra proves that a program follows the specification provided by the user, and that the code is safe from crashes, memory-related bugs, and data races. Gobra supports many of the language features of Go, including Go's methods, interfaces, and concurrency primitives. However, before this thesis Gobra did not support closures.

1.1 Goals

With this thesis, we pursue the following high-level goals.

A mechanism for the verification of closures in Gobra. The mechanism should support closure definitions, closure calls and the usage of closures as arguments to higher-order functions, i.e. functions that receive a closure as an argument, or return a closure. Closures should be able to capture variables.

Higher-order functions should be able to describe, in their specification, how a closure argument is used.

A consistent, modular, powerful, and generalisable design. Our language design should be *consistent* with the existing Gobra language design. Most importantly, our design should follow the principle of *modularity*, i.e. functions should be specified and verified independently from their callers. It is important that the design is *powerful* enough to support the verification of the most important use cases of closures.

1.2 Challenges

Closures[1] can be considered a generalisation of ordinary functions. Similarly to ordinary functions, they have arguments and results. However, they are first-class objects and they can capture variables, i.e. they can read or modify variables declared outside of their body. These features introduce additional challenges in the verification of closures.

First, we often do not statically know the definition of a closure that we are calling. For instance, a closure variable can be reassigned. Furthermore, a closure received by a higher-order function as an argument may have any definition. We want our verification design to be *modular*, so a higher-order function should have no knowledge about its potential callers. Therefore, we require a mechanism that enables higher-order functions to make assertions about the behaviour of a closure argument, without knowing its actual behaviour.

Second, closures can capture variables. The values of the variables captured by a closure form the internal state of the closure. Handling the internal state of a closure is tricky, especially if we do not know statically which variables are captured by a certain closures. Since the result of a closure call may be affected by the internal state of the closure, it is also tricky to verify properties about the result.

1.3 Outline

This thesis introduces a mechanism for the verification of closures in Go, and the related encoding to the Viper language. Viper is the verification framework used by Gobra. The framework consists in an intermediate language for the representation of programs, and the tools required to verify programs written using this intermediate language.

[1] The features of closures described here refer specifically to the implementation of closures in Go. The behavior and capabilities of closures may be different in different programming languages.

Chapter 2 provides some background information necessary to understand the content of this thesis. In particular, it presents the syntax and behaviour of closures in Go, and it provides some necessary details about Gobra and Viper.

Chapter 3 presents our design for the verification of closures in Go, starting from the high-level ideas and then explaining the syntax and behaviour of the new Gobra language primitives. We also point out how our ideas and design were influenced by previous works on the verification of closures, and discuss some alternative designs.

Chapter 4 describes how the new language primitives are encoded into the Viper language. We also discuss the soundness of our encoding.

Chapter 5 provides a series of case studies showcasing the possible uses of closures in Go, and how our approach can be used to handle all these different use cases.

Chapter 6 presents our experimental evaluation. We choose several representative examples of Go programs using closures in order to evaluate the performance of our solution.

Chapter 7 concludes the thesis with a discussion about our work, and about potential improvements or extensions to our work.

Chapter 2

Background

This chapter includes some background information necessary to understand this thesis. In Section 2.1 we briefly present the syntax and behaviour of closures in Go. Then, we present the main features of Gobra (Section 2.2). In Section 2.3 we briefly introduce the intermediate verification language Viper, used within Gobra's verification pipeline.

2.1 Closures in Go

Closures in Go are similar to closures in other mainstream languages such as Java, C# or Python. As an example, consider the code snippet below. `cl` is a closure, declared at lines 2-5 using a closure literal. The signature of the literal (at line 2 in the example) declares the arguments and results of the closure. Unlike an ordinary function, a closure literal does not have a name (closures are anonymous). In the example, the closure `cl` accepts an integer value as an argument and returns an integer, so `cl` has type `func(int)int`. The body is analogous to the body of an ordinary function. However, a closure literal can capture variables from the surrounding scope. In the example, variable `b` is captured by `cl`. When we call `cl`, `b` is increased (line 3) and the result of the call depends on `b` (line 4). Since the value of `b` is changed by a call to `cl`, the calls at lines 6 and 7 have different results. We can also still change the value of the captured variables directly, as shown at line 8. This affects the result of further calls (line 9). Differently from other languages such as Java, variables can also be passed to other functions by taking their address. For instance, the address of variable `b` is `&b`.

```
1       b := 3
2       cl := func(a int)int {
3                 b += 1
4                 return a + b
5             }

6       r := cl(3) // r == 6, b == 4
7       r = cl(3) // r == 7, b == 5

8       b = 10
```

```
9       r = cl(1) // r == 11, b == 11
```

There are many uses for closures in Go. Chapter 5 presents the main use cases and provides relevant examples.

2.2 Gobra

Gobra [13] is a tool for the deductive verification of Go programs. Gobra checks that a program satisfies a set of properties for every possible execution. We call this set of properties the *specification* of a program.

In order to verify a Go program, a user adds *annotations* to it. *Specification annotations* allow the user to define the specification of a program. Some properties need a proof to be verified; in this case, a user needs to write the corresponding *proof annotations*. To automatically verify an annotated Go program, Gobra translates it into the intermediate verification language Viper. The generated Viper program is then verified using other available tools.

In this section, we will briefly present the main features of Gobra that are relevant to our work. First, we will look at how Gobra handles the access to heap-allocated variables. Second, we will present some of the different kinds of annotations that a user can employ in Gobra. Third, we will see how Gobra handles the verification of Go interfaces.

Further information about Gobra and its features can be found in the paper by Wolf et al. that introduces Gobra [13]. There is also a tutorial on Gobra's GitHub repository [3].

2.2.1 Permissions

Gobra is based on *separation logic* [8]. In a nutshell, separation logic allows us to reason about heap-allocated values by introducing access permissions. In particular, each value allocated on the heap is associated with a permission, which determines whether we can access the value, and whether we can read from the value. In Gobra, a *full* or *exclusive* permission (corresponding to a permission amount of 1) denotes that we have write access to the value. A fractional permission (corresponding to any permission amount greater than 0 and ≤ 1) denotes that we have read access to the value.

The total permission amount associated with a single value is 1, meaning that, if we have full permission to a value, then no one else has access to the value. Conversely, if we have a fractional permission to a value, no one else can have full permission to it, so the value cannot be modified.

Assume that `x` and `y` are pointers to integers (i.e. have type `*int`). Assertion `acc(x)`, or `acc(x, 1/1)` means that we have full permission to the corresponding

value, so we can modify the value of `*x`. Assertion `acc(y, 1/2)` means that we have a fractional permission to `*y`, so we can read the value of `*y`, but not write to it.

Gobra also supports predicates, which are named assertions with parameters. An example can be seen in the following snippet, where we define predicate `p1`. Note that predicate instances are not equivalent to their body. For instance, in the following example, `p1(&x)` does not hold at line 4. If we want to verify assertion `p1(&x)`, we need to fold the predicate first, as shown at line 5. When we fold a predicate, we lose permission to the body of the predicate, and get permission to the predicate instance. We can also unfold a predicate, which does the opposite.

```
1  pred p1(x *int) { acc(x) && *x > 0 }

2  func main() {
3      x@ := 1
4      assert acc(&x) && *x > 0
5      fold p1(&x)
6      assert p1(&x)
7  }
```

Predicates in Gobra can be recursive, so we can define access permissions over unbounded data structures. The snippet below shows an example. The `Node` struct, containing an integer value and the pointer to another node, can be used to define a list of integers. The recursive predicate `list` defines access to all the nodes in the list with head `l`.

```
1  type Node struct { elem int; next *Node }
2  pred list(l *Node) { l != nil ==> acc(l) && list(l.next) }
```

2.2.2 Annotations

We can annotate a Go program using *assertions*, *function specifications* and *loop invariants*.

Assertions in Gobra are used to verify that a particular logic statement holds at a certain point in the program, for every execution of the program. Consider the following example. At line 1, we declare `x` to be an integer variable, and we assign value 1 to it. Therefore, at line 2, it is always the case that `x == 1`, so the assertion verifies successfully.

```
1  x := 1
2  assert x == 1
```

Methods and functions in Gobra can be annotated using *preconditions* and *post-conditions*. Preconditions define properties about the arguments of the function that always hold when the function is called (so, whenever the function is called, the caller must guarantee the preconditions). Postconditions define properties about the arguments and results that are guaranteed to hold after the execution of the function. In the example below, the precondition (at line 1) specifies that

there is write access to the value corresponding to `x`, and that this value is greater than 0. The postcondition ensures that we still have write access to `x` when the function returns, that `*x` is increased by 1, and that the result is equal to `*x`.

```
1  requires acc(x) && *x > 0
2  ensures acc(x) && *x == old(*x) + 1 && res == *x
3  func add1(x *int)(res int) {
4    *x += 1
5    return *x
6  }
```

A function or method is *pure* if it is deterministic and it has no side effects (i.e. it does not modify any heap values). Pure methods or functions are annotated with the `pure` modifier. Their body must consist of a single return statement, which returns an expression that is deterministic and without side effects. Their postcondition cannot contain permissions, since all of the permissions specified in the preconditions are always returned. Also, the expression in the body is used as an implicit postcondition, which says that the result is equal to the value of the returned expression. Furthermore, calls to pure methods or functions can be used within specifications. Consider the example below. Function `pos`, defined at lines 1-4, is a pure boolean function that returns `true` if the value that we can access using pointer `n` is positive. The precondition of `pos` at line 1 says that we need read access to `n` (the wildcard `_` indicates any non-zero permission amount). At lines 5-8, function `add1` calls `pos` in its specification. `preserves` is used for assertions that are both in the precondition and in the postcondition of the function, so `acc(a) && pos(a)` is both the precondition and the postcondition of `add1`.

```
1  requires acc(n, _)
2  pure func pos(n *int)bool {
3    return *n > 0
4  }

5  preserves acc(a) && pos(a)
6  func add1(a *int) {
7    *a += 1
8  }
```

Loops in Gobra can be annotated with invariants, i.e. assertions that hold just before executing the loop, and after each loop iteration. Within a loop body, we lose all permissions to the heap-allocated variables that are accessed within the body, unless such permissions are included in an invariant. In the example below, value `*x`, initially 0, is incremented 100 times using a loop. The invariant at line 2 provides an upper bound and a lower bound to the values that iteration variable `i` can have. Invariant at line 3 ensures that the permission to `x` is preserved and, at each iteration, `*x` is always equal to `i` (this last invariant holds since both values are increased by 1 at each iteration).

```
1  assert *x == 0

2  invariant i >= 0 && i <= 100
3  invariant acc(x) && *x == i
4  for i := 0; i < 100; i += 1 {
```

```
5       *x += 1
6   }
7   assert *x == 100
```

The annotations introduced above are used to verify the partial correctness of a Go program, i.e. the specification holds if the program terminates. Gobra also supports termination measures for loops and methods, which are used to verify the termination of the program. Usually, termination measures have the shape '`decreases n`', where `n` is a bounded value that decreases at each loop iteration or recursive call. Further details about termination measures in Gobra can be found in Xuan's Bachelor thesis [17], which introduced termination measures in Gobra.

2.2.3 Interfaces

Interfaces in Go are named collections of method signatures. Any type (e.g. a struct, or a pointer to a struct) implementing all the method signatures of an interface automatically implements the interface.

In Gobra, interface methods can be annotated with pre- and postconditions. Additionally, an interface can define abstract predicates, used to abstract over the heap structure of a type implementing the interface.

Consider the example below. Interface `I` defines predicate `mem` and method `transform`. The specification of method `transform` declares that the method must preserve access to an instance of predicate `mem`, and that the result is greater than or equal to the argument. Struct `S` contains a pointer to an integer. The struct implements both predicate `mem` (at line 8) and method `transform` (at line 13).

Intuitively, we can see that the specification of the implementation of `transform` at line 13 satisfies the specification at line 5, since the body of `self.mem()` is equivalent to `acc(self.v) && *self.v >= 0`. However, the verifier cannot prove this automatically, so we write the proof at lines 16-22 to prove that struct `S` implements interface `I`. In the proof, we assume that the precondition of the interface method holds. Therefore, at line 18 predicate instance `self.mem()` holds. We can thus use the `unfold` operation at line 18 to obtain the property `acc(self.v) && *self.v >= 0`. It is now possible to call the implementation method `self.transform`. After folding `self.mem()` at line 20, the verifier can prove that all of the postconditions of the interface method hold, which concludes the proof.

```
1   type I interface {
2       pred mem()
3       preserves mem()
4       ensures res >= x
5       transform(x int) (res int)
6   }
7   type S struct { v *int }
8   pred (self S) mem() {
```

```
 9       acc(self.v) && *self.v >= 0
10   }
11   preserves acc(self.v) && *self.v >= 0
12   ensures res >= x
13   func (self S) transform(x int) (res int) {
14       return x + *self.v
15   }

16   S implements I {
17       (self S) transform(x int)(res int) {
18           unfold self.mem()
19           res = self.transform(x)
20           fold self.mem()
21       }
22   }
```

2.3 Viper

Viper [6] is a verification infrastructure built with separation logic in mind. The Viper intermediate language allows the encoding of programs, and the verification of the properties of such programs.

In order to verify the specification of Go programs, Gobra encodes the annotated program into a Viper program, so that the Viper program verifies if and only if the specification of the Go program is satisfied.

The Viper intermediate language supports a permission model that is suitable for this purpose. In particular, heap-allocated objects in Viper are represented via references (type `Ref`), and the corresponding values are encoded as fields. For instance, global declaration '`field i: Int`' declares integer field `i`. As long as we have access to the field `i` of a reference `p`, we can access and modify the corresponding value, as shown in the example below.

In the example, at line 2 we obtain full permission to `p.i`, so we can assign to the corresponding value. The `inhale` operation in Viper allows the user to obtain permissions to any predicate or heap-allocated object, and to assume properties about any value. For example, at line 2 we obtain the permission to access `p.i`, and we assume that the current value of `p.i` is 2. The opposite operation is `exhale`, which removes the permissions specified after asserting the specified properties. For example, after the `exhale` operation at line 4, we cannot access `p.i` anymore. Also, the `exhale` operation at line 4 would fail if we did not have access to `p.i` or if the value of `p.i` were not 3.

```
1  var p Ref
2  inhale acc(p.i) && p.i == 2
3  p.i := 3
4  exhale acc(p.i) && p.i == 3
```

Further information about Viper and its features can be found in the paper by Müller et al. [6]. There is also a tutorial on Viper's website [11].

Chapter 3

Design

This chapter presents the ideas behind our approach to verifying closures in Gobra. The first section introduces how we solve the verification of closures using the key ideas of *specification entailment* and *entailment interfaces*. The second section gives a descriptions of the new language primitives, which we have introduced to put those ideas into practice. The third section explains how our design has been influenced by previous works on the verification of closures and points out the main differences in our approach. The fourth section presents some alternative designs that have been considered.

3.1 Overview

This section is a brief overview of the key concepts that underpin our approach to the verification of closures in Gobra. First, the concept of *specification entailment* gives us a way to reason about a closure specification being strong enough to satisfy a desired specification. Second, *entailment interfaces* allow higher-order functions to reason about the state of closures without violating information hiding.

3.1.1 Specification entailment

The concept of specification entailment is based on two key observations. First, each call to a closure satisfies some contract, which depends on the behaviour of the closure. This contract is a property of the closure, and we will call it the *specification* of a closure. Second, a specification can be used to entail a more general specification. Let us discuss these observations more in detail.

3.1.1.1 A call to a closure satisfies a contract

Calling a closure is similar to calling a function: some precondition must hold before the call, and the call guarantees some postcondition after it. We define a

specification of the closure to be any pair of precondition and postcondition that holds for any call to the closure.

Consider the closure `cl` defined in the following code snippet.

```
1  cl := func(x int)int {
2          return x
3        }
```

Following our definition of closure specification, we can assert that `cl` satisfies the following specification. In fact, if the argument received by `cl` is non-negative, the result will also be non-negative.

```
1  requires x >= 0
2  ensures res >= 0
3  func pos(x int)(res int)
```

This does not mean that `pos` is the only valid specification for `cl`. For example, `cl` also satisfies the specifications `g10pos` and `reqPos`, shown below.

```
1  requires x > 10
2  ensures res >= 0
3  func g10pos(x int)(res int)

4  requires x >= 0
5  ensures true
6  func reqPos(x int)(res int)
```

We can actually make a stronger statement than '`cl` satisfies `g10pos` and `reqPos`'. In fact, any closure that satisfies `cl` will also satisfy `g10pos` and `reqPos`, not just `cl`. In fact, `g10pos` has a stronger precondition than `pos`. So, any arguments that are valid for `g10pos` will also be valid for `pos`, and the postcondition guarantees are the same. We can also observe that `reqPos` has the same precondition, but a weaker postcondition than `pos`.

More formally, if we assume that a closure `c` satisfies pre- and postcondition pair (P_1, Q_1), then the closure `c` satisfies (P_2, Q_2) if there exists an assertion R such that $P_2 \Rrightarrow P_1 * R$ and $Q_1 * R \Rrightarrow Q_2$. R is any residual property (e.g. access permissions unused by P_1, which can then be assumed in the postcondition). The arrow $\Rrightarrow$ represents a *viewshift*. A *viewshift* is an implication that can also change the layout of permissions (e.g. by folding and unfolding predicates).

If the condition written above holds, then we say that the specification described by pair (P_1, Q_1) entails the specification described by pair (P_2, Q_2).

We can exploit specification entailments in Gobra to make the verifier aware that, if a closure satisfies a particular specification, then it also satisfies a second specification if we can show that the first specification entails the second.

3.1.2 Entailment interfaces

Unlike ordinary functions, closures can capture variables. The state of these variables, which we will refer to as *captured state*, can change both within and

```
1   func hof(f()int) {
2       // ...
3   }

4   func main() {
5       x := 0
6       cl := func() (res int) {
7                   x += 1
8                   return x
9             }
10      hof(cl)
11  }
```

Figure 3.1: Closure example: a counter.

outside the closure. The presence of a captured state has two consequences. First, whether a closure satisfies a specification may depend on its captured state. Second, it may be necessary to reason about the captured state of a closure, but without knowing the details (e.g. which variables are captured by the closure). The concept of *entailment interface* provides a solution to both of these problems.

Consider the example in figure 3.1. Closure `cl` captures variable `x` and increases it by one every time it is called. We would like to be able to pass `cl` as an argument to `hof`. What we need is a specification that `hof` can require, which allows `hof` to reason about the internal state of the closure without knowing the details. In particular, `hof` should not know about variable `x`. At the same time, the precondition of the specification should guarantee access to `x`.

An entailment interface allows a specification to abstract over the state of a closure, providing abstraction predicates and functions. To show that a closure satisfies the specification, the user will then need to provide a suitable instance of the interface.

Consider again the example in figure 3.1. A potential annotation for `hof` is in figure 3.2. The function `fspec` is coupled with entailment interface `Inv`, and asserts that the corresponding predicate `inv()` is preserved. `hof` will then require that the closure satisfies the specification `fspec` with entailment interface `inv`, and it will ensure that the predicate `inv()` is preserved. To call `hof`, any caller will then need to implement the interface, making sure that the implementation of `inv()` contains any required permissions and properties of its captured state.

In general, entailment interfaces can be used to express many kinds of closure properties. Such interfaces can provide abstraction functions and predicates to allow a higher-order function, for example, to specify how the result of a closure call is used, or how many times a closure has been called. Section 5.6 shows how this can be done, and how this allows us to successfully deal with several common closure use patterns.

```
type Inv interface { pred inv() }
preserves inv != nil && inv.inv()
func fspec(ghost inv Inv)(res int)

requires inv != nil && f implements fspec{inv}
preserves inv.inv()
func hof(ghost inv Inv, f()int) {
    // ...
}
```

Figure 3.2: A higher-order function allowing closure parameters with internal state.

3.2 Language extensions

3.2.1 Function literal specification

Function literals are specified, similarly to ordinary functions, using preconditions, postconditions, and termination measures. The syntax is shown below. Specified function literals can have a name, which is used to refer to the specification when calling the closure. Note that, in Go, function literals are not allowed to have a name, thus the name is considered a proof annotation.

```
trusted?
requires P
ensures Q
decreases M
pure? func specName?(arg1, ..., argN) (res1, ..., resN) { BODY }
```

Function literals are pure expressions that return a closure. If the literal captures variables, then the preconditions, postconditions, and termination measures can reference these variables. The captured variables must be addressable and access to them must be guaranteed by the precondition.

The visibility of the function literal name follows the same visibility rules as any variable. The name can be used within ghost code as an expression. The value of this expression is the same as the closure created by the function literal. For example, within the precondition `P` of `specName`, the expression `c == specName`, where `c` is an argument or a captured variable, means that `c` is the same closure as the one created by `specName`. The name `specName` can only be used as an expression within the specification or body of the corresponding literal. It cannot be used as an expression within the specification or body of any other function literal, even if the other literal is defined within the body of `specName`.

The `trusted` clause is used to tell Gobra not to verify the body. The body of the literal can be omitted (including the curly braces) to declare an abstract function literal. Note that omitting the function literal body is not allowed in Go, so it can only done within ghost code.

The `pure` clause is used to specify that the function literal does not have side effects. In this case, the body and specification must satisfy the same restrictions

as those required for ordinary pure functions. In particular, the body must consist of a single, pure return statement.

The following example shows a specified function literal, assigned to closure variable `c`. The literal is named `accumulate`, it captures variable `x`, and it accepts a single integer argument `n`. Variable `x` is increased by `n`, and then returned. This is reflected in the specification, which declares that access to `x` is preserved, that the final value of `x` is equal to the old value of `x` plus `n`, and that the result is equal to the final value of x.

```
1       x@ := 0
2       c := preserves acc(&x)
3            ensures x == old(x) + n && res == x
4            func accumulate(n int) {
5                x += n
6                return x
7            }
```

3.2.2 Closure specification instance

The name of any function, or function literal, constitutes a closure specification instance. The specification instance is used within closure calls and specification assertions, and it refers to the preconditions, postconditions and termination measures of the function, or function literal, with the same name. The function can also be a member of an imported package. In this case, standard package resolution (`package.name`) applies.

Optionally, a closure specification instance can also have parameters. The full syntax is shown below.

```
1       fName{(name1: )? exp1, ..., (nameN: )? expK}?
```

Any parameters are specified within curly braces, after the function name. If there are no parameters, the curly braces are allowed but not necessary. Either all parameters are named, or all parameters are unnamed. If a parameter is not named, the corresponding expression is assigned to the argument of `fName` with the same position in the list of arguments. If the parameter has a name, then the name must correspond to the name of one of the arguments of `fName`, and the expression is assigned to that parameter.

The type of a specification instance corresponds to the type of `fName`, after removing the arguments that are treated as parameters. Considering the function `addTo` below, for example:

- `addTo` has type `func(int,*int)int`
- `addTo{2}` has type `func(*int)int`
- `addTo{x: &y}`, where `y` is an addressable integer variable, has type `func(int)int`
- `addTo{2, &y}` has type `func()int`

```
1       preserves acc(x)
2       ensures *x == old(*x) + n && res == *x
3       func addTo(n int, x *int) (res int)
```

A closure that implements a specification instance must have the same type as the specification instance. A closure satisfies a specification instance with parameters `fName{params}` if and only if all calls to the closure satisfy the preconditions, post-conditions and termination measures of `fName`, where all parameters are replaced with the corresponding expression.

If `fName` is pure, then we consider its body to be part of the postconditions. This is consistent with the current behaviour of pure functions. In fact, after a pure function call, we can verify assertions that can only be inferred from the body.

3.2.3 Specification entailment assertion

Specification entailment assertions express that a closure, function, or method satisfies a specification instance. They are not heap-dependent, so, once they are established, they always hold. This applies even if the closure has a captured state, as any properties related to the state are included in the specification, which prevents a closure from being called if its state is invalid. A specification entailment assertion is written '`f implements spec`', where `f` is a closure, function or method, and `spec` is a closure specification instance.

3.2.4 Closure call

If a user wants to call a closure, they need to declare which specification to use. A closure call is written as '`c(exp1, ..., expN) as spec`', where `c` is an expression with function type, `spec` is a specification instance and `exp1`, ..., `expN` are the expressions passed as arguments to the function. The call is a pure expression if `spec` is declared as pure.

The call requires that '`c implements spec`' and the preconditions of `spec` hold. After a call, the postconditions of `spec` are ensured.

Function and method expressions that implement a spec instance can also be called using that spec, analogously to a closure. See line 18 of figure 3.3 for an example of a call to an ordinary function that explicitly uses a spec different from the function's own specification.

3.2.5 Specification entailment proof

A specification entailment proof is a statement used to establish an entailment assertion. The syntax is shown below.

```
1       proof f implements spec {
2           BODY
3       }
```

The body is similar to that of an interface implementation proof. It consists of:

- A single call to `f`. If `f` is a closure, the call must be accompanied by a specification, as explained in 3.2.4. If `f` is a function or method expression, it can also be called without an associated specification instance.
- `fold`, `unfold` and `assert` statements.

If `spec` is pure, the body must consist of a single return statement, where the return expression consists of a single call to `f` and `unfolding` sub-expressions. These restrictions are analogous to the restrictions for interface implementation proofs for pure methods.

The single call to `f` in the body must use, as arguments and results, exactly the names that the arguments and results have in the definition of `spec`. Within the proof, these names are visible and they always refer to the arguments and results of `spec`, shadowing any other variables with the same name.

If `spec` has termination measures, which prove that the function terminates, then the specification being used in the call must also have termination measures.

At the beginning of the proof body, the precondition of `spec` is assumed. The proof verifies whether the postcondition of `spec` holds at the end of the body. If that is the case, the fact '`f implements spec`' is established after the proof.

Let (P_1, Q_1) be the pre- and postconditions of the single call in the proof body, and (P_2, Q_2) the pre- and postconditions of `spec`. The only statements allowed other than the call are `assert`, `fold` and `unfold` statements. These statements can only change the permission layout. Therefore, a successful proof shows that there exists assertion R such that $P_2 \Rrightarrow P_1 * R$ and $Q_1 * R \Rrightarrow Q_2$, which means that the specification used for the call entails `spec` (as explained in 3.1.1). Therefore, it is correct to assume, after a successful proof, that `f` implements `spec`.

3.2.6 Entailment interface

Interfaces in Gobra support predicates and functions, so they provide all the elements necessary to implement the concept of entailment interfaces. Consider the following code snippet.

```
1  type InvRes interface {
2    pred inv()

3    ghost
4    requires inv()
5    pure res(n int)int
6  }

7  ghost
8  preserves inv != nil && inv.inv()
9  ensures res == old(inv.res(n))
10 func spec(ghost inv InvRes, n int)(res int)
```

The interface `InvRes` can be used as an entailment interface. It includes a predicate `inv()`, which allows the verification of closures with a captured state, since an implementation of `inv()` can include access to the captured variables. It also includes a pure function `res`, which requires access to `inv()`.

Function `spec` shows how to associate an entailment interface with a specificaton. `spec` accepts an instance `inv` of the interface as a ghost argument. The specification of `spec` requires that `inv.inv()` is preserved, and the result is the same as calling `inv.res(n)` in the state before the call.

The snippet below shows a possible implementation for the interface.

```
1  type InvResImpl struct { x *int }
2  pred (self InvResImpl) inv() {  acc(self.x) }
3  ghost requires self.inv()
4  pure func (self InvResImpl) res(n int)int {
5      return unfolding self.inv() in (*self.x + n)
6  }
```

Struct `InvResImpl` contains integer pointer `x`, and the predicate `inv()` contains access to `x`. The result, given argument `n`, is defined to be the value of `x` plus `n`. The struct implements the predicates and functions as indicated by `InvRes`, so `InvResImpl` implements `InvRes`. A closure of type `func(int)int` that captures an integer variable `y` and whose result, given argument `n`, is `y + n` can be shown to implement specification instance `spec{InvResImpl{&y}}` using a specification entailment proof.

3.2.7 Examples

The following two examples use all of the language features presented in this section. The first example shows how a function can be given as an argument to another function using specification entailment. The second example shows how an entailment interface can be used, together with specification entailment, to verify that a closure with a captured state can be given as an argument to a higher-order function.

3.2.7.1 Simple example

The example in figure 3.3 shows a simple specification entailment.

In the example, function `hof` requires that the closure `c` that it receives as an argument satisfies spec `pos`. This specification entails that `c` must be able to be called with any positive argument, and `c` must ensure that, if the argument is positive, then so is the result.

The caller calls `hof`, using function `plus2` as an argument (line 17). Before the call, the caller needs to show that the specification of `plus2` entails the specification of `pos`. The related proof is at lines 14-16. Inside the proof, a single call to `plus2` is enough to show the entailment. Line 18 shows that it is possible to call an ordinary function using a different specification that is entailed by the function.

```
1  ghost
2  requires a >= 0
3  ensures r >= 0
4  func pos(a int) (r int)

5  preserves c implements pos
6  func hof(c func(int) int) {
7          v := c(42) as pos
8          assert v >= 0
9  }

10 pure func plus2(a int)int {
11     return a + 2
12 }

13 func main() {
14     proof plus2 implements pos {
15         r = plus2(a)
16     }

17     hof(plus2)

18     r := plus2(3) as pos
19     assert r >= 0
20 }
```

Figure 3.3: Simple example, without captured variables.

3.2.7.2 Full example

Figure 3.4 shows an example where a higher-order function accepts a closure which is allowed to have a captured state.

In the example, function `hof` requires that the closure that it receives as an argument satisfies spec `pos{p}`, where `p` is an instance of an interface with predicate `inv()`. We can treat `p` as the instance of an entailment interface. The predicate allows `c` to have an internal state. In fact, a valid implementation of the interface can define `inv()` to include the permissions required to access any captured variables. The rest of the specification is analogous to the one in the previous example.

The caller defines an implementation of the entailment interface at lines 1-2, by declaring struct `Acc` and associated predicate `inv()`. Instance `Acc{&x}` of this struct is then used as a parameter in specification instance `pos{Acc{&x}}`.

The caller declares `c` to be an accumulator function which captures variable `x`. The value of `x` is updated by adding the value of the argument of the function to it at each call. Each call returns the value of `x` after the update. Note that this definition, by itself, does not guarantee that the result is always positive. However, the definition of `inv()` at line 2 says that `x` remains non-negative. Since any number of calls that satisfy the precondition of `pos` will keep the value of `x` non-negative, the proof at line 11 succeeds. The definition of `inv()` also allows us to assert, at line 19, that, after calling `hof`, the value of `x` is still non-negative.

```
1  type Acc struct { x *int }
2  pred (p Acc) inv() { acc(p.x) && *p.x >= 0 }

3  func main() {
4          x@ := 0
5          c := preserves acc(&x)
6               ensures x == old(x) + n && m == x
7               func f(n int) (m int) {
8                   x += n;
9                   return x
10              }

11         proof c implements pos{p: Acc{&x}} {
12                 unfold Acc{&x}.inv()
13                 r = c(a) as f
14                 fold Acc{&x}.inv()
15         }

16         fold Acc{&x}.inv()
17         hof(c, Acc{&x})
18         unfold Acc{&x}.inv()

19         assert x >= 0
20 }

21 ghost
22 requires p.inv() && a >= 0
23 ensures p.inv() && r >= 0
24 func pos(ghost p interface{pred inv();}, a int) (r int)


25 preserves p.inv() && c implements pos{p}
26 func hof(c func(int) int, ghost p interface{pred inv()}) {
27         v := c(42) as pos{p: p}
28         assert v >= 0
29 }
```

Figure 3.4: Full example, with captured variables.

3.3 Sources of inspiration

The concepts presented in this section have been inspired by two previous works on the verification of closures.

Wolff et al. [14, 15] introduced support for the verification of closures in Prusti, an automated verifier for the programming language Rust.

Their paper introduces the idea of specification entailments, which is similar to the one presented here, with some important differences. The differences are related to the fact that, due to the guarantees provided by Rust's type system, Prusti does not have user-defined permissions. Also, Rust imposes some restrictions on the use of captured variables. As a consequence, specification entailments in Prusti are checked automatically when a higher-order function is called. Instead, in order to make use of specification entailments in Gobra, a user needs to write a separate proof.

Their paper introduces the ideas of call descriptions, used by higher-order functions to describe with which arguments a closure is called and how the result is used. This idea inspired the research of a mechanism that would allow us to reach similar

objectives in Gobra. However, these objectives are achieved in a very different way. In fact, while Gobra uses entailment interfaces for this, Prusti introduced additional specification primitives for functions.

Their paper proposes to use ghost predicates for the verification of some higher-order functions. This idea has some similarities with the idea of entailment interfaces, as predicates within the interface can provide the same functionality. However, the concept of entailment interfaces is more general. For example, it allows to define abstraction functions in addition to predicates. As a result, entailment interfaces have a wider use in our approach than ghost predicates have in theirs.

B. Weber investigated the verification of closures in Python programs in their Masters thesis [12], within the automatic verifier Nagini. Their core idea was based on two ideas. First, closure calls can be associated with a contract. Second, the user should provide a proof, for each closure call, that the closure satisfies the contract.

Similarly to our approach, closure contracts have a name, and a contract must be specified for each call. Their thesis also introduces specification entailment to enable a user to prove that a closure satisfies a contract. However, within their approach each closure call must be associated with an entailment proof, whereas we support separate proofs. Furthermore, they do not support the verification of closures that capture variables, and they do not have entailment interfaces.

3.4 Alternative designs

During our research, we have considered a few different designs for the verification of closures. The following sections briefly describe some alternative designs that we have considered, and explains why we have decided not to use them. First, we will see why applying the approach used in Prusti [14] directly is problematic. Second, we will look at a design where call descriptions are handled explicitly.

3.4.1 Applying the approach from Prusti directly

In this section, we will explain the main issues that arise when we try to use, in Gobra, the approach used in Prusti for the verification of closures. The design for specification entailments used in Prusti (a verifier for Rust) has the following characteristics[1].

- The pre- and postconditions attached to a closure have no explicit access permissions, since the access permissions can be inferred automatically using Rust's type system.

[1] This is just a short summary, which does not intend to explain the characteristics of Prusti's design in detail.

```
1  requires f |= (i int) { requires i > 0; ensures res > 0 }
2  ensures res > 0
3  func hof(f func(int)int)(res int) {
4      // ...
5  }

6  func main() {
7    x@ := -1
8    cl := invariant x >= old(x)
9          requires i > 0
10         ensures res == old(x) && x == old(x) + i
11         func(i int)(res int) {
12             r := x
13             x += i
14             return r
15         }
16   // cannot use x here
17   cl(2)
18   hof(cl)
19   assert x >= 1
20 }
```

Figure 3.5: An example showing Prusti's design for specification entailments

- A user can define the *invariants* of a closure, which denote the properties that keep holding for a closure after an unbounded number of calls. These invariants can express the relationship between the initial state of the closure and any state reachable after calling the closure any number of times.

- The variables mutably captured by the closure (i.e. variables captured with write access) cannot be accessed before the end of the closure's lifetime. If the variables are captured immutably, they cannot be modified until the end of the closure's lifetime. These conditions are enforced by Rust's type system, which makes the job easier for Prusti.

The example in figure 3.5 is taken from Wolff et al. [14]. The example is simplified and adapted to use Gobra-like syntax. Higher-order function `hof` requires a specification entailment at line 1 (which states that, if the closure receives a positive argument, then the result of a closure call is positive). Function `main` defines closure `cl` at line 8. `cl` captures variable `x`, which is increased by as much as indicated by the closure argument `i`. The result of the closure is the value of `x` before the increase. The specification of `cl` has an invariant, which states that, after 0 or more calls to `cl`, the value of `x` increases or stays the same.

The call at line 17 increases the value of `x` by 2. Therefore, the value of `x` becomes 1. The invariant at line 8 specifies that the value of `x` will not decrease, so, from now on, the result of a call to `cl` will always be positive. Therefore, the specification entailment required by `hof` is satisfied, and we can call `hof` at line 18. After the call, thanks to the invariant, we know that the value of `x` is at least 1.

3.4.1.1 Issues

A design that consists in applying Prusti's approach for the verification of closures directly in Gobra has a series of issues.

The first issue is that the expressivity of such a design is limited by the fact that there cannot be any user-defined permissions, precluding the verification of closures where we cannot infer the required permissions automatically. Trying to extend the design, by naively introducing user-defined permissions and predicates, would also not work, for two reasons. First, we are not able to infer automatically that a certain predicate is equivalent to, or stronger than, a second one. Therefore, we cannot automatically verify that a specification entails a second specification which has a different permission layout. Hence, we are only able to call a higher-order function requiring exactly the same access permissions as the actual closure. This means that the higher-order function needs to require the permissions to access the captured variables, making the design violate encapsulation (a higher-order function, in general, should not know about the variables captured by a closure). Second, the encoding used in Prusti involves the use of a *snapshot*, i.e. a mathematical abstraction over the value of the arguments, captured variables and results of a closure. User-defined predicates make the generation of snapshots problematic, due for example to the potentially recursive nature of predicates.

The second issue is that we need to understand when the lifetime of a closure ends, and when we have access to its internal state. To see why this is problematic, consider the example in figure 3.5. The caller `main` does not know how the function `hof` uses the closure, so we cannot assume that we have access to `x` after the call, or that we can call the closure again. In fact, `hof` might lose permission to the internal state of the closure, for example by giving it as an argument in a `go` call to another higher-order function. Therefore, a design applying Prusti's approach directly would need to be extended, for example adding primitives to indicate that we have permission to access the internal state of the closure.

The third issue is that such a design does not fit well within the existing Gobra verification framework. In fact, this design precludes the verification of perfectly valid examples of Go programs, since it imposes restrictions on the usage of variables that are specific to Rust's type system. For example, it enforces that a closure and its captured variables cannot be accessed at the same time.

3.4.2 Handling call descriptions explicitly

In our design, we use entailment interfaces to allow higher-order functions to describe the usage of a closure, as demonstrated by the examples in Section 5.6. We have also considered a different approach, which introduces additional verification primitives for this purpose. In this section, we will briefly present this alternative approach, and justify our decision not to use this approach in the final design.

3.4.2.1 Call description primitives

The example below shows the call description primitives introduced in the alternative design. The first primitive is the call definition, at line 1. Here, the higher-order function gives a name to a closure call (`c1`), and asserts the properties of the argument values (in the example, that the value of `a` is 10). At line 3, `called#c1` indicates that call `c1` happened, and `call#c1(r)` refers to the value of the result of call `c1`. In general, the value of any expression `E` in the poststate of the call can be referred to as `call#c1(E)`.

```
1  call f#c1 { a == 10 }
2  requires f implements s1
3  ensures called#c1 && res == call#c1(r)
4  func hof(f func(a int)(r int)) (res int) {
5      return (f#c1(10) as s1) + 1
6  }
```

In order to use the information provided by `hof` using the call description primitives, the caller has to provide more information about the behaviour of the closure. An example is shown below, where `caller` calls the function `hof` defined above. At line 2, closure `f` is defined to be a closure that returns a result that is twice the argument, and the related specification is named `s2`. `hof` is called at line 5. The caller, at line 6, indicates that `f` should be used with specification `s2`. This specification allows to know more about the value returned by the call than what `hof` already knows. In particular, at line 7 there is an indication that the result of the call is 20 (two times the argument, which is 10, as specified by `hof`). In general, the caller of a higher-order function indicates the specification associated with each closure used as an argument to the higher-order function. Additionally, the caller uses the keyword `ensures` to specify properties about the values in the post-state of the closure calls executed by the higher-order function. The verifier then checks that these properties can be inferred from the closure specification indicated by the caller and from the information contained in the call descriptions within the higher-order function's specification. In the Viper encoding, a proof that the properties hold is then generated automatically. For instance, the encoding of the example contains a proof that the specification `s2`, combined with the information that `a == 10` (from the specification of `hof`), entails that the result of the closure call is 20.

```
1  func caller() {
2      f := ensures r == 2 * a
3           func s2(a int)(r int) { return 2*a }
4      // ...

5      r := hof(f)
6           using f as s2
7           ensures call#c1(r) == 20

8      assert r == 21
9  }
```

3.4.2.2 Issues

When we try to use the approach presented above, some issues arise if we want to introduce user-defined predicates. Consider the example below. Function `hof` receives argument closure `f`, and integer pointer `*x`. The function `f` is called, in call `c1`, using specification `s1`, which uses predicate `acc1` to state that the access to argument `a` is preserved. Therefore, to access `*a`, we need to unfold predicate `acc1` (as we can see at lines 4 and 6, where the `unfolding` operation is used). The caller wants to use closure `f`, defined at line 11, as an argument to `hof`. We can do it, since specification `s2` entails specification `s1` (assume that, at some point, the entailment has been proven).

```
1  pred acc1(a *int) { acc(a) }

2  preserves acc1(a)
3  func s1(a *int)

4  call f#c1 { unfolding acc1(a) in (*a == old(*x)) }
5  requires acc(x) && f implements s1
6  ensures acc(x) && called#c1 && *x == call#c1(unfolding acc1(a) in (*a))
7  func hof(f func(a *int), x *int) {
8    // ...
9  }

10 func caller() {
11     f := preserves acc(a)
12          ensures *a == old(*a) + 1
13          func s2(a *int) {
14             *a += 1
15          }
16     // ...

17     x@ := 2
18     hof(f, &x)
19         using f as s2
20         ensures call#c1(unfolding acc1(a) in (*a)) == 3
21 }
```

However, in this case we cannot generate automatically a proof that the statement at line 20 is entailed by the specification `s2`, because the specification of `s2` does not use predicate `acc1`. We need some additional annotation from the user, specifying how to transform the permission layout (using `fold` and `unfold` operations), from the one required by `s1` to the one required by `s2`, similarly to what we do with specification entailment proofs.

The example above shows that, in general, using the alternative design presented, the annotation overhead for the caller can be quite high. Indeed, it is comparable to the annotation overhead we need in a design where we just use entailment interfaces. Therefore, introducing the call description primitives presented in this section is not necessary, and, in general, it does not improve the annotation overhead. For this reason, these primitives have not been introduced in the final design.

3.5 Syntactic sugar

In this section we propose the introduction of some syntactic sugar for our design, to reduce the annotation overhead. This proposal has not been implemented yet, but it provides an idea for potential future work.

3.5.1 Syntactic sugar for closure literals

It is often the case that a closure specification proof is trivial, as it only involves calling the closure with its original specification. We propose to introduce some syntactic sugar to prove that a closure implements a particular specification directly within the specification of the corresponding closure literal. In this section, we sketch the proposal using an example. We do not provide details of a potential implementation of the proposal.

Consider the following example. The caller (defined at lines 9-19) defines closure `cl` at lines 10-14. The closure is only used as an argument to `hof`, which is called at line 18. In order to use `cl` as an argument to `hof`, the closure must implement specification `pos`, defined at lines 1-3.

With the current design, in order to use `cl` in the call at line 18, we need to provide a specification entailment proof, which we do at lines 15-17.

```
1  requires n >= 0
2  ensures res >= 0
3  func pos(n int)(res int)

4  requires c implements pos
5  ensures res >= 0
6  func hof(c func(int)int)(res int) {
7      return res
8  }

9  func caller() {
10     cl := requires n >= 0
11           ensures res >= 0
12           func spec(n int)(res int) {
13             return n + 1
14           }

15     proof cl implements pos {
16         return cl(n) as spec
17     }

18     hof(cl)
19 }
```

Our proposal is to allow the following syntax, which combines the closure definition with the specification entailment proof. The specification of the closure literal has an additional clause, at line 4, which specifies that the closure returned by the literal satisfies the specification `pos`. Since the entailment proof is trivial, we can verify that the entailment holds automatically, for example by adding the corresponding checks in the encoding of the literal.

With this syntax, it is not always necessary to give a name to the literal in order to use the corresponding closure.

```
1  func caller() {
2      cl := requires n >= 0
3            ensures res >= 0
4            implements pos
5            func(n int)(res int) {
6              return n + 1
7            }
8      hof(cl)
9  }
```

To make the annotation even more compact, we can also allow a user not to write the pre- and postcondition (at lines 10 and 11 in the original example), since in this example we only use the specification of `pos`. Therefore, we can rewrite the example as follows.

```
1  func caller() {
2      cl := implements pos
3            func(n int)(res int) {
4              return n + 1
5            }
6      hof(cl)
7  }
```

The syntax presented above allows the user to directly use a closure literal as an argument to a higher-order function. In fact, we can rewrite the example above as follows.

```
1  func caller() {
2      hof(implements pos
3          func(n int)(res int) { return n + 1 })
4  }
```

Chapter 4

Encoding

This chapter presents and justifies the Viper encoding of the language extensions introduced in 3.2. The first section presents the notation used in the rest of the chapter; the second presents the encoding for the new Gobra types, expressions and statements; the third describes the encoding for the new Viper members generated to enable the encoding of the new features. The fourth section briefly discusses the soundness of the encoding, while the fifth section presents an issue affecting the soundness of termination checks for closure literals.

4.1 Notation

Throughout this chapter, we will use the following notation:

- $[\![$`code`$]\!]$ denotes the Viper encoding of `code`.
- `fName{params}` denotes the name assigned, in the encoding, to the specification instance `fName{params}`. This name depends on `fName` and on the set of arguments used as parameters, not on the value of the parameters. When we say that two specification instances are *distinct*, we mean that their assigned names are different, i.e. their base functions are different or they use a different set of arguments as parameters.
- `recv.methName` denotes the name assigned, in the encoding, to the method `recv.methName`. This name depends on `methName` and on the type of the receiver.
- `Type` denotes a name that uniquely identifies Viper type `Type`.
- *`code[exp1 → sub1, ..., expN → subN]`* denotes the result of replacing, within `code`, expressions `exp1, ..., expN` with expressions `sub1, ..., subN`.
- *`fName.args`*, *`fName.res`*, *`fName.P`*, *`fName.Q`* and *`fName.BODY`* denote the arguments, results, precondition, postcondition, and body of `fName` respectively.

4.2 Encoding of new types, expressions and statements

4.2.1 Function type

$[\![\texttt{func(int)int}]\!] \triangleq \texttt{Closure}$

where `Closure` is a Viper domain (defined in 4.3.1).

4.2.2 Nil/default function variables

$[\![\texttt{nil: func(\_)\_}]\!] \triangleq \texttt{closureNil()}$

$[\![\texttt{dflt[func(\_)\_]}]\!] \triangleq \texttt{closureNil()}$

where `closureNil` is a Viper domain function (defined in 4.3.1).

4.2.3 Function literals

$[\![\texttt{pure? func litName(\_)\_ \{...\}}]\!]$

$\triangleq \texttt{closureGet\$litName(}[\![\textit{litName.capturedVariablePointers}]\!]\texttt{)}$

where `closureGet$litName` is a Viper function (defined in 4.3.4.2)
and *litName.capturedVariablePointers* is a list containing pointers to the variables captured by `litName`, in order of appearance within `litName`.

4.2.4 Function and method expressions

Function or method expressions are expressions that consist of a function or method name. The name of a function or method, in Go, can be used similarly to any other function-typed variable.

$[\![\texttt{funcName}]\!] \triangleq \texttt{closureGet\$funcName()}$

where `closureGet$funcName` is a Viper function (defined in 4.3.5).

$[\![\texttt{recv.methName}]\!] \triangleq \texttt{closureGet\$recv.methName(}[\![\texttt{recv}]\!]\texttt{)}$

where `closureGet$recv.methName(`$[\![\texttt{recv}]\!]$`)` is a Viper function (defined in 4.3.6).

The analogous expression for function literals, as explained in 3.2.1, can only be used within the literal itself.

$[\![\texttt{litName}]\!] \triangleq \texttt{closure}$

where `closure` is a parameter in the encoding of the callable Viper member corresponding to `litName` (defined in 4.3.4.3). This parameter is required, by a precondition of the generated member, to have the same value as the closure returned by the function literal (see 4.2.3).

4.2.5 Closure specification entailment assertions

⟦`cl implements fName{params}`⟧

$\triangleq$ `closureImplements$fName{params}(`⟦`cl`⟧`, `⟦`params.values`⟧`)`

where `closureImplements$fName{params}` is a Viper domain function that returns a boolean (defined in 4.3.3).

4.2.6 Closure calls

⟦`cl(args) as fName{params}`⟧

$\triangleq$ `closureCall$fName{params}`
(⟦`cl`⟧, ⟦*fName.captVarsFuncs(cl)*⟧, ⟦`combine(args, params)`⟧)

where

- `closureCall$fName{params}` is a Viper method, if `fName` is not pure, or function, if `fName` is pure. If there are no parameters and `fName` denotes a function literal, this Viper method or function is defined at 4.3.4.3, otherwise it is defined at 4.3.7.
- *fName.captVarsFuncs(cl)* is defined in 4.3.2.1.
- *combine(args, params)* is a sequence of expressions combining `args` and parameter values so that each parameter value is in the right position according to the list of arguments of `fName`.

4.2.7 Specification entailment proofs

⟦ `proof cl implements fName{params} { BODY }` ⟧ $\triangleq$

```
1  ∀ aX in fName.args: ⟦declare aX_S⟧
2  ∀ rX in fName.res: ⟦declare rX_S⟧
3  ∀ (p.name, p.value) in params: p.name_S := ⟦p.value⟧
4  CL_SUB := ⟦cl⟧
5  if (*)
6  {
7      var numIterations: Int
8      numIterations := closureProofIterator()
9      while(numIterations > 0)
10       decreases numIterations
11     {
12         inhale ⟦ fName.P[aX → aX_S] ⟧
13         label pr_lbl
14         ⟦ BODY[aX → aX_S, rX → rX_S, cl → CL_SUB, old(E) → old[pr_lbl](E) ] ⟧
15         exhale ⟦ fName.Q[aX → aX_S, rX → rX_S, old(E) → old[pr_lbl](E)] ⟧
16         numIterations := numIterations - 1
17     }
18     assume false
19 }
20 inhale ⟦ cl implements fName{params} ⟧
```

At lines 1 and 2, we define substitutes for the arguments and results of `fName`.

At line 3, we assign the parameter values to the corresponding argument substitutes (each parameter name `p.name` matches the name of an argument of `fName`).

At line 4, we assign the value of closure expression `cl` to the exclusive variable substitute `CL_SUB`. This assignment is necessary to ensure that no permissions are needed, inside the proof, to access the closure.

The arguments, results, and closure expression are substituted within the encoding of the proof precondition, body and postcondition, as indicated at lines 12, 14, and 15.

The body is enclosed within a loop, which in turn is enclosed within an if-statement. The if-condition is a non-deterministic boolean value, which forces the verifier to verify the body. The `assume false` at the end ensures that any changes to the state of the program made within the if-statement are discarded.

The purpose of the loop is to lose all permissions related to the arguments, including the parameters. The preconditions of `fName` are assumed at the beginning of the body, and they must hold at the end so that the `exhale` operation, at line 15, succeeds.

`closureProofIterator` is an abstract, terminating Viper method that returns an unknown integer. Therefore the number of executions of the loop is unknown, but not infinite. This allows the verifier to prove the termination of the enclosing method, if necessary, since `numIterations` is decreased at line 16. Because the number of iterations is unknown, the proof still works if a verifier backend attempts to inline loops.

The `inhale` statement at the end ensures that, from now on, the expression '`cl implements fName{params}`' is known to be true.

4.3 New Viper members

4.3.1 Closure domain

A new domain `Closure` is introduced to encode Go's function type.

```
1  domain Closure {
2      function closureNil(): Closure

4      [[ capturedVarDomainFunctions ]]

6      [[ specEntailmentDomainFunctions ]]
7  }
```

`closureNil` returns the `nil` function, which we encode as a variable of `Closure` type with no known properties. $[\![capturedVarDomainFunctions]\!]$ are the domain functions necessary to access the variables captured by a closure, defined in 4.3.2. [[specEntailmentDomainFunctions]] are the domain functions used for specification entailment assertions, defined in 4.3.3.

4.3.2 Domain functions necessary to access captured variables

capturedVarDomainFunctions is a list of functions that allows all closures in the package to access the variables that they capture.

Let *CaptTypes* be set of encoded types of pointers to all the variables captured by any function literal in the package. For example, if the package contains a function literal that captures a variable of type `int`, *CaptTypes* will contain type `Ref` (the encoding of type `*int`).

For each type `T` in *CaptTypes*, let N_T be the maximum number of distinct variables with encoded pointer type `T` that are captured *by a single function literal* in the package.

We define *capturedVarDomainFunctions* as follows.

$[\![$ *capturedVarDomainFunctions* $]\!] \triangleq$
  $\forall$ *T in CaptTypes* :
    $\forall$ *X := 1..*N_T :
      `function captVar`*X*`Closure_`$\underline{T}$`(closure: Closure): `*T*

4.3.2.1 List of variables captured by a specific function literal

Given a closure `cl` and function literal `fName`, we define *fName.captVarsFuncs(cl)* to be the list of domain function calls that correspond to the variables captured by `fName`, assuming that the value of `cl` is the same as the one returned by `fName`. If `cl` is not the closure returned by `fName`, the values returned by the call will either be undefined, or they will correspond to the variables captured by some other function literal.

If `fName` does not denote a function literal, then the list is empty.

Let `vlist` be the list of variables captured by `fName`, in the order in which they appear within `fName`. For each variable `vlist[i]`, let T_i be the encoded type of a pointer to `vlist[i]`, and let X_i be the number of variables `v[j]` such that `j` $\leq$ `i` and T_j == T_i.

We obtain each element of *fName.captVarsFuncs(cl)* as follows.

fName.captVarsFuncs(cl)`[i]` $\triangleq$ `captVar`X_i`Closure_`$\underline{T_i}$`(cl)`

Each of the functions `captVar`X_i`Closure_`$\underline{T_i}$ is defined within the `Closure` domain. In fact, as explained in 4.3.2, the encoding makes sure that, for each type T_i, there are enough domain functions to access captured variables whose pointer has this type.

4.3.3 Domain functions used for specification entailment assertions

For each distinct specification instance `fName{params}` in the package, we generate an abstract Viper function as follows.

```
function closureImplements$fName{params}(closure: Closure, ⟦params⟧])
                                                    : Bool
```

4.3.4 Function literal getter and callable member

The encoding of a function literal generates two Viper members: a *callable method or function* and a *getter function*. To make the process easier, the literal is first subject to a series of transformations.

4.3.4.1 Internal transformation

The function literal is transformed, internally, by:

- Replacing every occurrence of a captured variable with the dereference of a fresh variable. The variable's type is the type of a pointer to the variable. For example, captured variable `x`, of type `int`, is replaced by expression `*x_RN0`, where `x_RN1` is a fresh variable of type `*int`.
- Renaming it, so that the new name is unique in the package.
- Transforming the body and specification similarly to any other function.

After the transformation described above, the literal has the shape shown in the snippet below.

```
requires P
ensures Q
func litName<c1_T0, c2_T0, c1_T1, ...>(a1,, ..., aN) (r1, ..., rR) {
    BODY
}
```

`a1, a2, ..., aN` are the arguments, `r1, ..., rR` are the results. `c1_`T_0, `c2_`T_0, `c1_`T_1, `...` are the captured variables substitutes, where T_n is the type of the variable after the encoding. Note that, in the actual encoding, the names are different.

4.3.4.2 Closure getter

We generate a Viper function, so that a call to this function can be used to retrieve the corresponding closure variable. The function looks as follows.

```
function closureGet$litName(⟦c1_T0, c2_T0, c1_T1, ...⟧): Closure
  ensures closureImplements$litName(result)
  ∀ cX_Tn: ensures captVarXClosure_Tn(cl) == cX_Tn
```

The function ensures that the result implements `litName`, and the values of the relevant captured variables are the ones given as arguments. This allows the result to be used as an argument to the `closureCall$litName` method.

4.3.4.3 Callable method or function

A non-pure function literal is encoded as a Viper method. The encoding looks as follows.

```
1  method closureCall$litName$(closure: Closure, ⟦c1_0, ...⟧, ⟦a1, ...⟧)
2    returns (⟦r1, ..., rR⟧)
3    (if litName does not capture any variables)
4          requires closureImplements$litName(closure)
5    (if litName captures some variables)
6          requires closure == closureGet$litName(closure, ⟦c1_0, ...⟧)
7    requires ⟦P⟧
8    ensures ⟦Q⟧
9  {
10     ⟦BODY⟧
11 }
```

The arguments are:

- A closure parameter, which represents the closure being called;
- The substitutes of the captured variables (`c1_0, c2_0, ...`);
- The actual arguments.

The specification is the encoding of the original specification, with the additional precondition that:

- If the literal does not capture any variables, the closure must implement `litName` (line 4);
- If the literal captures some variables, `closure` must have the same value as the one returned by the closure getter (i.e. it must be the closure created using the function literal). This precondition is at line 6; note that this condition is stronger than the one at line 4.

The purpose of this Viper member is twofold: first, it allows the verification of the literal's body and specification; second, it is used for closure calls.

The encoding for a pure literal is a Viper function with the following signature. The specification and body are defined analogously to the non-pure version.

```
1  method closureCall$litName$(closure: Closure, ⟦c1_0, ...⟧, ⟦a1, ...⟧)
2                      : ⟦litName.res.type⟧
```

4.3.5 Function expression conversion

If function name `someFunc` is used as a function expression, we introduce a Viper conversion function, that we can call to retrieve a closure value implementing `someFunc`. This conversion function looks as follows.

```
1  function closureGet$someFunc(): Closure
2    ensures closureImplements$someFunc(result)
```

The function is very similar to a closure getter for function literals. The only difference is that there are no captured variables.

4.3.6 Method expression conversion

If a function name `recv.someMeth` is used as a method expression, we introduce a conversion function, similarly to what we do with functions. However, in this case the encoding depends on the type of the receiver.

Let `recvT` be the type of `recv`. The encoding of the corresponding conversion function for `recv.someMeth` depends on whether `recvT` is an interface type or not.

If `recvT` is not an interface type, the conversion function looks as follows.

```
1  function closureGet$recv.someMeth(self: ⟦recvT⟧): Closure
```

The function is very similar to the conversion for functions. The differences are the following:

- The receiver is accepted as a parameter;
- The result is not assumed to implement any spec. The reason for this absence is that specs, for now, can only be constructed from functions, or function literals, but not from methods. The support for method-based specs could be added in the future, but it is not necessary. In fact, to prove that the method implements a particular spec, a user can use normal method calls inside a specification entailment proof.

If `recvT` is an interface type, the conversion function looks as follows.

```
1  function closureGet$recv.someMeth(self: ⟦recvT⟧): Closure
2    requires ⟦self != nil⟧
3    ∀ tX implementing recvT :
4        ensures (⟦typeOf(recvT) == tX⟧) ==>
5                    result == closureGet$recv.someMeth(⟦self.(tX)⟧)
```

The postcondition at lines 4-6 ensures that the result depends on the concrete type of `self`: if the concrete type is `tX`, the result is the same as we would get after casting the type of `self` to `tX`. So, if there are variables `i1` and `s1` such that `i1` has interface type, `s1` has a type that implements the type of `i1`, and `i1 == s1`, then `i1.someMeth` and `s1.someMeth` have the same value.

4.3.7 Method or function used by closure calls

For each distinct specification instance `fName{params}` in the package, where `fName` is a non-pure function, we generate an abstract Viper method as follows.

```
1  method closureCall$fName{params}(cl: Closure, ⟦fName.args⟧)
2      returns (⟦fName.res⟧)
3      requires closureImplements$fName{params}(cl, ⟦params.names⟧)
4      requires ⟦fName.P⟧
5      ensures ⟦fName.Q⟧
```

Note that, by construction, each parameter in `params` corresponds to one of the arguments of `fName`, so `params.names` are valid values. A call to this method has

the same requirements and guarantees as a call to the original `fName` method, with the additional precondition that the closure argument `cl` implements the spec.

If `fName` is pure, we generate a Viper function instead, as follows.

```
1  function closureCall$fName{params}(cl: Closure, ⟦fName.args⟧)
2                                  : ⟦fName.res.type⟧
3      requires closureImplements$fName{params}(cl: Closure,⟦params.names⟧)
4      requires ⟦fName.P⟧
5      ensures ⟦fName.Q⟧
6      ensures result == ⟦fName.BODY.returnExpression⟧
```

The remarks made for the method also apply to the function. An additional observation is that there is an additional postcondition, at line 6. This additional postcondition asserts that the result of a call is equal to the return expression of the body of `fName`. The additional postcondition is necessary because, as explained in 3.2.2, we consider the body of pure functions to be part of their postcondition. Without the added postcondition, we would instead lose all information about the body when calling this function. We would obtain the same result by copying the body instead. However, adding a postcondition has the advantage that the verifier does not need to verify again that `fName.Q` holds, which is unnecessary, since this fact is already verified in the original function.

4.4 Soundness

This section provides an intuition as to why the encoding presented in the previous sections is sound. This section is not meant to provide a formal proof of soundness.

4.4.1 Closure domain

A single closure domain is enough to represent all variables of function type. In fact, the type checker makes sure that:

- A closure of a certain type cannot be assigned to a variable of a different type;
- Two closures with different types cannot be compared;
- In any closure call (e.g. '`cl(args) as spec`') and specification entailment assertion (e.g. '`cl implements spec`'), the types of the closure and of the spec match;
- In specification entailment proofs, the types of the closure and of the spec match.

Therefore, all type mismatches are detected by the type checker. Hence, at the encoding stage we can assume that the closures have the correct type.

4.4.2 Nil/default closure

As emphasised in 4.4.1, it is not a problem that all nil closures, regardless of their type, have the same encoding.

Additionally, the user cannot call a nil closure, because no spec instance is implemented by it (unless the user explicitly assumes that it does).

4.4.3 Function literals

If the `closureCall` method or function generated from a function literal verifies, then the function literal is correct. In fact, if we only consider normal arguments and results, the behaviour of a function literal is equivalent to that of an ordinary function. Also, capturing variables is equivalent to implicitly receiving pointers to the variables. So, assuming that the verification of ordinary functions is sound, then the verification of function literals is also sound.

For the same reason, if termination checks succeed for the generated member, then the function literal terminates.

4.4.4 Closure calls

The encoding of closure calls ensures that the call '`cl(args) as spec`' only verifies if the verifier knows that `cl` implements the specification of `spec`. Therefore, if a function verifies, it must be the case, by definition of specification entailment, that the precondition of `spec` is enough to call `cl`, and that such a call ensures the postcondition of `spec`. This guarantees that, if the call verifies, any assertion that can be inferred right after the call is sound.

4.4.5 Specification entailment proofs and assertions

If specification entailment proof '`proof cl implements spec { BODY }`' verifies, then the behaviour of `cl` entails the specification.

The body of the proof, when encoded, is put inside a loop. Within the loop, all knowledge about the properties of the parameters is lost, except for their value. In particular, all access permissions are lost. The variables used to represent non-parametrised arguments and results are completely new variables, so no information about them is known in the proof except for what the preconditions state. Therefore, any assertion within the proof body must be entailed by the precondition of the spec, and by the value of the parameters.

Therefore, for the call inside the body to succeed, the preconditions must be entailed by the preconditions of `spec`. Analogously, the `exhale` statement at the end only succeeds if the postconditions of the call, combined with any residual properties unused by the call, entail the postconditions of `spec`.

Hence, it is sound to assume, after a successful proof, that `cl` implements `spec`. Whether a specification entailment assertion is true depends on the specific values of the parameters, so the fact that the values of the parameters are not havoc-ed within the proof body does not invalidate the soundness of the proof.

Also, the success of a proof does not depend on the state of a closure, since the state can only be accessed through access permissions, which are havoc-ed in the proof. Therefore, it is sound to treat a specification entailment assertion as a heap-independent expression.

4.4.6 Termination

The type checker verifies that, if `spec` has any termination measures, then the call of `cl` within the specification must use a specification which also has termination measures. The purpose of this check is to ensures that only closures that are known to terminate can implement a specification that includes termination measures.

The termination of closure literals is checked analogously to the termination of regular functions, since closure literals are encoded as Viper functions or methods. Viper provides a *termination plugin* that Gobra uses to add termination checks to the encoded Viper program. However, in some scenarios, our encoding undercuts the generation of checks by the plugin, leading to unsound termination checks. We provide more details about this problem, and a potential solution, in the next section.

4.5 Problem with termination checks for closure literals

In this section, we explain a problem in the encoding of our design that leads to the presence of unsound termination checks for closures in some scenarios. First, we briefly explain how termination checks work in Gobra. Then, we provide an example showing a situation where termination checks fail to detect non-termination. Lastly, we sketch a potential solution to the problem.

4.5.1 Termination checks in Gobra

In order to add termination checks to a generated Viper file, Gobra uses a plugin provided by Viper. If the specification of a function contains termination measures, then the plugin adds termination checks to ensure that the function actually terminates. In particular, the checks ensure that loops terminate and that all function calls terminate. Intuitively, a function call terminates if the callee is guarantee to terminate.

A function `f` is recursive if it contains a *recursive* call. A function call in `f` is recursive if it calls `f` directly, or if it calls a function that depends on `f`. Recursive

```
1  requires n >= 0
2  decreases n
3  func a(n int) {
4      if (n > 0) {b(n-1)}
5  }

6  requires m >= 0
7  decreases m
8  func b(m int) {
9      if (m > 0) {a(m-1)}
10 }
```

Figure 4.1: An example of functions calling each other.

Figure 4.2: Call graph corresponding to the program in figure 4.1.

calls are treated differently from other calls, as they require additional checks. Therefore, it is necessary that the termination plugin is able to identify which calls are recursive.

To generate the correct termination checks, Viper's termination plugin generates a *call graph* to identify which functions are recursive. In the call graph, each node corresponds to a function, and each edge corresponds to one or more calls. In particular, there is an edge from node A to node B if the function A contains a call to the function B. If a node A belongs to a cycle in the graph, then the corresponding function is recursive.

Consider the example in Figure 4.1. Functions `a` and `b` call each other. Both `a` and `b` are encoded as Viper methods, and the corresponding call graph is in figure 4.2. Nodes `a` and `b` belong to a cycle. Therefore, the calls `b(n-1)` (line 4) and `a(m-1)` (line 9) are recursive, and they will be treated as recursive calls by Viper's termination plugin. Identifying call cycles, and therefore identifying which calls are recursive, is crucial for the soundness of termination checks, because recursive calls need to satisfy more stringent conditions than non-recursive calls.

4.5.2 Explanation of the soundness problem

Closure calls are encoded as calls to the Viper method or function corresponding to the specification used in the call. Therefore, calls to the same closure are encoded as calls to separate methods, if they use different specifications. Hence, the call graph corresponding to the encoding of a program that uses closures does not preserve all the information required by the verification plugin. As a consequence, in some situations, Viper's verification plugin is unable to detect closure recursion, leading to unsoundness in the verification of termination.

```
1  ghost
2  preserves acc(acl, 1/2) && *acl implements aspec{acl}
3  decreases n
4  func aspec(acl *func(int), n int)

5  func t() {
6      var c@ func(int)
7      c := preserves acc(&c, 1/2) && c implements aspec{&c}
8           decreases n
9           func f(n int) {
10             c(n) as aspec{&c}
11          }

12     proof c implements aspec{&c} {
13         c(n) as f
14     }

15     c(5) as f
16 }
```

Figure 4.3: A non-terminating recursive closure.

Consider the example in figure 4.3. Function `t`, defined at lines 5-16, declares closure variable `c` at line 7, and defines it at lines 8-12. Closure `c` calls itself at line 11, using the unchanged argument `n`. Intuitively, it is clear that `c` does not terminate, so the verification of the program should fail, since the specification at line 8 requires the verifier to check that the closure always terminates. However, the verification of the program succeeds.

Let us consider the call graph that Viper's verification plugin generates for this example. Constructing the call graph requires some facts about the encoding. Following the encoding presented in this chapter, we know the following (we simplify the names of the encoded methods for convenience).

- Closure literal `f` (lines 7-11) is encoded as method `clCall$f`.
- Closure call `c(n) as aspec{&c}` (line 10) is encoded as a call to method `clCall$aspec_1`, which is encoded from the function specification defined at lines 1-4.
- Function `t` (lines 5-16) is encoded as method `t`.
- Closure calls `c(n) as f` (line 13) and `c(5) as f` are encoded as calls to method `clCall$f`.

The call graph generated by Viper's termination plugin is shown in figure 4.4. The body of `t` contains calls `c(n) as f` (line 13) and `c(5) as f` (line 15), so there is an edge from node '`t`' to node '`clCall$f`'. The body of closure literal `f` contains call `c(n) as aspec{&c}` (line 10). Therefore, there is an edge from node '`clCall$f`' to node '`clCall$aspec_1`'. Since there are no other calls, this is the entire call graph (excluding nodes that are not relevant to our discussion). Note that there is no edge between '`t`' and '`clCall$aspec_1`' because the call at line 10, in the encoding, is not part of the body of '`t`', since closure literals are encoded as separate methods.

Figure 4.4: (Incorrect) call graph corresponding to the program in figure 4.3.

Figure 4.5: (Correct) call graph corresponding to the program in figure 4.3.

The problem is that the call graph does not reflect the fact that '`clCall$aspec_1`' might depend on '`clCall$f`', since the proof at lines 12-14 uses spec `f` to prove that `c` implements `aspec{&c}`.

4.5.3 Proposed solution

One possible solution to the problem is to ensure that the call graph of an encoded Gobra program reflects that callable methods corresponding to closure specification instances (whose encoding is defined at 4.3.4.3 and 4.3.7) might depend on other methods. In particular, we can infer all the requires dependencies from the body of the specification entailment proofs in the package.

Consider the example in figure 4.3 again. The correct call graph corresponding to the example is shown in figure 4.5. The edge shown using a dotted line represents the dependency between specification instance `aspec{_}` and specification instance `f`, which can be inferred from the proof at lines 12-14 in the code.

In practice, for each specification entailment proof `proof c implements spec1`, we need to add a dependency between the Viper method or function corresponding to `spec1` and the Viper method/function that is called within the body of the proof.

Currently, Viper's verification plugin does not support the explicit addition of dependencies in the call graph. However, we can make sure that the dependencies are added to the call graph using the approach sketched next. Note that the proposal is only a sketch and needs to be refined. The approach proposed here is similar to the approach currently used to ensure the soundness of termination checks for calls to interface methods. We make the assumption that Viper's termination plugin does not distinguish between dead code and alive code when computing the call graph.

Approach for non-pure closure specification instances

Consider non-pure specification instance `spec{params}` and consider the corresponding callable method `closureCall$spec{params}`. According to the encoding described in 4.3.7, `closureCall$spec{params}` has no body. Let `proof1`, `proof2`, ..., `proofN` be the specification entailment proofs in the package that are used to show that some closure implements `spec{params}`.

We define a new encoding, where the method `closureCall$spec{params}` has the same specification as before. However, we now define the body as follows.

```
1  method closureCall$spec{params}(cl: Closure, [[spec.args]])
2  // specification
3  {
4      inhale false

6      [[proof1.BODY.call]]

8      [[proof2.BODY.call]]

10     // ...

12     [[proofN.BODY.call]]
13 }
```

Statement `inhale false` at line 2 ensures that the calls at lines 4-10 have no effect on the verification of the partial correctness of the program. However, since the calls are contained inside the body of the generated method, Viper's termination plugin will add the corresponding dependencies in the call graph, as wanted.

Approach for pure closure specification instances

Consider pure specification instance `spec{params}` and consider the corresponding callable function `closureCall$spec{params}`. Similarly to the non-pure case, the function has no body according to the current encoding, defined at 4.3.7. Define `proof1`, `proof2`, ..., `proofN` analogously to the case where the specification instance is not pure.

We define a new encoding, where the function `closureCall$spec{params}` has the same specification as before. However, we now define the body as follows.

```
1  function closureCall$spec{params}(cl: Closure, [[spec.args]])
2  // specification
3  {
4      return true ? closureCall$spec{params}$true(cl, [[spec.args]])
5                  : (true ? [[proof1.returnExpr]]
6                          : (true ? [[proof2.returnExpr]]
7                                  // ...
8                                  : ([[proofN.returnExpr]])
9                          ...))
10 }
```

The body is a nested ternary operator. The condition in the outermost ternary operator is `true`. Therefore, the verifier knows that a call to the function is just equivalent to verifying a call to `closureCall$spec{params}$true`.

`closureCall$spec{params}$true` is defined analogously to the definition of `closureCall$spec{params}` in the original encoding. The rest of the body contains all of the expressions contained in all relevant specification entailment proofs. So, although the rest of the body is not taken into account by the verifier, Viper's termination plugin will add the dependencies corresponding to proofs `proof1`, ..., `proofN` in the call graph, as wanted.

Chapter 5

Case study

This chapter presents common use cases of closures. For each use case, we propose a relevant example, and we show how our design can be used to verify such examples.

5.1 Calling a closure within the function defining it

If a closure is called within the function that defines it, we fully know its specification. The following example shows how this simple case can be verified following our approach. The closure variable `c` is defined at line 3 using a function literal `ctr`. The literal defines a counter that captures variable `cnt`, increments it at each call and returns the new value of `cnt`.

The closure is called first at line 8, and then at line 14, within a loop. The assertions at lines 9 and 15 show that we can make assertions about the new value of `cnt` after each call. We can also change the value of the captured variables freely, as shown at line 10.

```
1  func main() {
2      cnt@ := 0
3      c := preserves acc(&cnt)
4           ensures cnt == old(cnt) + 1 && res == cnt
5           func ctr()(res int) {
6              cnt += 1; return cnt
7           }

8      r := c() as ctr
9      assert r == 1 && cnt == 1

10     cnt = 10

11     invariant i >= 0 && i <= 89
12     invariant acc(&cnt) && cnt == i + 10
13     for i := 0; i < 89; i += 1 {
14         r = c() as ctr
15         assert r == i + 11 && cnt == i + 11
16     }
17     assert cnt == 100
18 }
```

5.2 Recursive closures

A closure is recursive if it contains a call to itself. In Go, this is achieved as shown in the snippet below. The closure variable, called `c` in the example, is declared at line 1. The literal declared at lines 2-6 then calls closure `c` at line 4, with any arguments. The literal is assigned to variable `c`, which means that the call at line 4 will use the function declared by the literal, i.e. it is a recursive call.

```
1       var c func(int)int
2       c = func(a int)int {
3               // ...
4               c(??)
5               // ...
6           }
```

The code snippet below shows how our approach allows the verification of recursive closures in Gobra. The closure in the snippet implements the factorial function recursively, and the corresponding function literal `factorial` is defined at lines 7-14. The precondition, postcondition, and termination measures at lines 7,8, and 10 are analogous to those that a standard factorial function defines: the argument is non-negative and it is decreased in recursive calls, and the result is specified by pure function `fact`. `fact`, defined at line 3, is a pure ghost function used to define the value of the n-th factorial.

Closure variable `c`, defined at line 7, is used in a recursive call at line 17. Therefore, `c` is captured by the literal, and we need to specify access permissions at line 9. The remaining precondition `c == factorial` means that, for calls to succeed, closure `c` must have exactly the same value as the closure created by the literal `factorial`. This last condition allows the verifier to know that `c` implements `factorial`, and that it captures the same variables (`c` itself, in this case). This information is enough for the call `c(n-1) as factorial` (line 13) to be verified.

Line 17 shows a call to the recursively-defined closure `c`, which verifies and has the expected result.

```
1   ghost requires n >= 0
2   pure func fact(n int)int {
3       return n == 0 ? 1 : n * fact(n-1)
4   }

5   func main() {
6       var c@ func(int)int
7       c = requires n >= 0
8           preserves acc(&c, 1/2) && c == factorial
9           ensures r == fact(n)
10          decreases n
11          func factorial(n int) (r int) {
12              if n == 0 { return 1 }
13              else { return n * c(n-1) as factorial }
14          }

15      assert fact(0) == 1 && fact(1) == 1 && fact(2) == 2 &&
16          fact(3) == 6 && fact(4) == 24 && fact(5) == 120

17      r := c(5) as factorial
18      assert r == 120
19  }
```

5.3 Specification entailment for interface methods

We can use specification entailments to make assertions about the behaviour of interface methods. Consider the example in the following code snippet. Every type implementing interface `Box`, declared at lines 1-5, must implement a predicate `mem` and a pure function `value`, which returns an integer and preserves `mem`. Function `transformBox`, declared at lines 6-11, receives a `Box` variable as an argument, calls it at line 10 and returns the result of the call multiplied by 2. We want to ensure that this result is divisible by 4. The specification provided by the interface, however, is not enough. We need an additional requirement, informally written at line 6, which says that `box.value()` returns an integer divisible by 2, for the specific implementation of `box` received as an argument.

```
1  type Box interface {
2      pred mem()

3      preserves mem()
4      value()(r int)
5  }

6  requires box != nil && [the result of box.value() is even]
7  preserves box.mem()
8  ensures res % 4 == 0
9  func useEvenBox(box Box) (res int) {
10     return box.value() * 2
11 }
```

We can use a specification entailment to write the precondition that we need, as shown in the next code snippet. Lines 1-3 define the function `even`, which we will use as a specification. The only parameter of `even` is variable `box`, of type `Box`. The parameter `box` is necessary to link the specification defined by `even` with a particular instance of interface `Box`. The pre- and postconditions of `even` declare that the `mem` predicate of of `box` is preserved, and that the result of `even` is divisible by 2. Function `useEvenBox` can now define precondition `box.value implements even{box}` at line 4 to require that `box.value()` has an even result. Thanks to this precondition, we can add the specification instance `even{box}` to the call at line 8, allowing the verifier to know that the result of the call will be divisible by 2.

```
1  ghost preserves box.mem()
2  ensures r % 2 == 0
3  func even(ghost box Box)(r int)

4  requires box != nil && box.value implements even{box}
5  preserves box.mem()
6  ensures res % 4 == 0
7  func useEvenBox(box Box) (res int) {
8      return (box.value() as even{box}) * 2
9  }
```

A potential caller of `useEvenBox` is shown below. Function `main`, defined at lines 8-17, creates variable `box` at line 12, using struct type `TCtrBox`. `TCtrBox` is defined at line 1 as a struct type, which contains a pointer to an integer. The predicate `mem()`, defined at line 2, contains access to the value accessible through the

pointer, and method `value()` returns the value of the pointer, multiplied by 2. Since this implementation of `value()` always has a result that is divisible by 2, the entailment proof at lines 12-14 succeeds. Variable `box` can thus be used in the call to `useEvenBox` at line 15.

```
1  type TCtrBox struct { x *int }
2  pred (self TCtrBox) mem() { acc(self.x) }
3  preserves self.mem()
4  ensures res % 2 == 0
5  func (self TCtrBox) value()(res int) {
6      return unfolding self.mem() in (2 * *self.x)
7  }

8  func main() {
9      x@ := 0
10     box := TCtrBox{&x}
11     fold box.mem()

12     proof TCtrBox{&x}.value implements even{TCtrBox{&x}} {
13         return TCtrBox{&x}.value()
14     }

15     r := useEvenBox(box)
16     assert r % 4 == 0
17 }
```

5.4 Higher-order functions knowing the exact behaviour of a closure

Specification entailments can be used to verify higher-order functions that know the exact specification of a closure. To see an example of this, let us consider a variation of the example considered in Section 5.1. This variation of the example is shown below without annotations. The closure `c` is defined as in the original example. The difference is that the main function, instead of calling `c` directly, delegates this task to function `call_counter_n_times`, by calling it at line 6. The function `call_counter_n_times`, given an integer `n` and closure `ctr`, calls the closure `n` times.

```
1  func main() {
2      cnt := 0
3      c := func ctr()(res int) {
4              cnt += 1; return cnt
5           }

6      call_counter_n_times(100, c)
7  }

8  func call_counter_n_times(n int, ctr func()int) {
9      for i := 0; i < n; i += 1 {
10         ctr()
11     }
12 }
```

It is a common programming practice to split functions, in order to improve the readability of the code by making the functions shorter. Sometimes, as in the example shown above, this practice leads to the creation of higher-order functions. In this case, the programmer knows exactly which closures are the arguments to

the higher-order function. Therefore, in this specific case, our objective is writing a specification for the higher-order function that leverages the full knowledge of the behaviour of the closure. This objective can be achieved using specification entailment. The annotated version of the code, shown in the code snippet below, demonstrates how our mechanism is suitable for such a scenario.

Lines 1-7 are analogous to the same lines in the original example in Section 5.1. In addition to the actual arguments, the function `call_counter_n_times` has a ghost argument `cnt` (line 21), which corresponds to the pointer to the variable captured by the closure. The precondition at line 18 states that argument `n` must be non-negative (line 18). The other precondition at line 18 (`ctr implements ctr_spec{cnt}`) is used to specify the exact behaviour of `ctr`. In fact, the signature and specification of `ctr_spec` (lines 14-17) matches exactly the definition of function literal `ctr` at lines 3-7, with the only difference that any captured variables are replaced by pointer arguments. In our case, captured variable `cnt`, of type `int`, is replaced with argument `cnt`, of type `*int`. Because the specification of `ctr_spec` matches the behaviour of the closure exactly, the specification entailment proof at lines 8-10 succeeds.

Since function `call_counter_n_times` knows the exact behaviour of `ctr` via the precondition at line 18, it can track the value of the captured variable `cnt` at every call. Therefore, the remaining part of the specification can be verified and is meaningful. In particular, the rest of the specification states that access to `cnt` is preserved (line 19), and that, after the call, the value of `cnt` is increased by `n` (line 20).

```
1  func main() {
2      cnt@ := 0
3      c := preserves acc(&cnt)
4           ensures cnt == old(cnt) + 1 && res == cnt
5           func ctr()(res int) {
6              cnt += 1; return cnt
7           }
8      proof c implements ctr_spec{&cnt} {
9          return c() as ctr
10     }
11     call_counter_n_times(100, c, &cnt)
12     assert cnt == 100
13 }
14 ghost
15 preserves acc(count)
16 ensures *count == old(*count) + 1 && res == *count
17 func ctr_spec(count *int)(res int)
18 requires n >= 0 && ctr implements ctr_spec{cnt}
19 preserves acc(cnt)
20 ensures *cnt == old(*cnt) + n
21 func call_counter_n_times(n int, ctr func()int, ghost cnt *int) {
22     invariant i >= 0 && i <= n
23     invariant acc(cnt) && *cnt == old(*cnt) + i
24     for i := 0; i < n; i += 1 {
25         ctr() as ctr_spec{cnt}
26     }
27 }
```

5.5 Higher-order functions returning a closure

Functions can return a closure. The caller of such functions has the benefit that it does not need to know about the exact implementation of the closure, making the code easier to read and to maintain. The example below shows, as an example, a function that returns a counter function. Function `getCounter` declares integer variable `x` at line 2, and the variable is then captured by the closure `c` declared at line 2. The closure increases the variable and returns its value. `getCounter` then returns the closure `ctr`.

```
1  func getCounter() (ctr func()int) {
2      x := 0
3      ctr = func count() (res int) {
4                x += 1; return x
5            }
6  }
```

We can verify this example using our verification mechanism. Our objective, in this case, is that the caller `getCounter` can call the closure returned by `getCounter`, with knowledge about the values returned by each call but without knowing the implementation details. To achieve this goal, we define an appropriate entailment interface and closure specification, shown below. The interface predicate `inv` at line 2 allows the presence of an internal state, and the pure interface method at line 4 provides a way to make assertions about the value returned by the closure. The `counter` function, at lines 6-10, defines the specification of the closure returned by `getCounter`. The only parameter of `counter` is `ctrint`, an instance of the entailment interface `Ctr`. The specification states that the predicate `ctrint.inv()` is preserved (line 8), that `ctrint.value()` increases by 1 at each call (line 9), and that the result of `counter` is the same as `ctrint.value()`. Note that there is no indication that `ctrint.value()` corresponds to the value of a particular variable, or any other information about the implementation of the counter.

```
1  type Ctr interface {
2      pred inv()

3      ghost requires inv()
4      pure value() int
5  }

6  ghost
7  requires ctrint != nil
8  preserves ctrint.inv()
9  ensures ctrint.value() == old(ctrint.value()) + 1 && res == ctrint.value()
10 func counter(ghost ctrint Ctr)(res int)
```

Below, we show the annotated `getCounter`. At lines 18, we define struct `CtrImpl`, which implements entailment interface `Ctr`. The `inv` predicate, at line 19, contains access to an integer pointer, corresponding to the captured variable. The `value` function, defined at lines 20-23, returns the value accessible via the pointer. In

the body of `getCounter`, a specification entailment proof (at lines 10-14) is used to prove that the closure `ctr` implements the specification instance `counter{CtrImpl{&x}}`, where `x` is the variable captured by `ctr`. A ghost return value, with name `ctrint` and type `Ctr`, is returned together with the closure. The value of this additional return value is `CtrImpl{&x}`. However, the caller of `getCounter` will only know that `ctrint` is an instance of entailment interface `Ctr`. `ctrint.inv()` is folded at line 16, in order to ensure permission to `ctrint.inv()` in the postcondition.

The postcondition of `getCounter` (lines 1-2) communicates, to the caller, that there is permission to `ctrint.inv()`, that the current value returned by `ctrint.value()` is 0, and that the closure returned implements the specification of `counter`, with entailment interface instance `ctrint`.

```
1  ensures ctrint != nil && ctrint.inv() && ctrint.value() == 0
2  ensures ctr implements counter{ctrint}
3  func getCounter() (ctr func()int, ghost ctrint Ctr) {
4      x@ := 0
5      ctr = preserves acc(&x)
6            ensures x == old(x) + 1 && res == x
7            func count() (res int) {
8               x += 1; return x
9            }

10     proof ctr implements counter{CtrImpl{&x}} {
11         unfold CtrImpl{&x}.inv()
12         res = ctr() as count
13         fold CtrImpl{&x}.inv()
14     }

15     ctrint = CtrImpl{&x}
16     fold ctrint.inv()
17 }

18 type CtrImpl struct { cnt *int }
19 pred (self CtrImpl) inv() { acc(self.cnt) }
20 ghost requires self.inv()
21 pure func (self CtrImpl) value() int {
22     return unfolding self.inv() in (*self.cnt)
23 }
```

The snippet below shows a potential caller of `getCounter`. Closure variable `ctr` is declared and defined at line 2, alongside entailment interface instance `ctrint`. Note that `ctrint` is a ghost variable. The closure is then called at lines 3 and 6, using specification instance `counter{ctrint}`. Assertions at lines 4 and 7 check that we can verify that the value returned by the calls increases by one at each call.

```
1  func main() {
2      ctr, ctrint := getCounter()

3      r := ctr() as counter{ctrint}
4      assert r == 1

5      r = ctr() as counter{ctrint}
6      assert r == 2
7  }
```

5.6 Higher-order functions describing their calls to a closure

In this section, we will consider several scenarios where a higher-order function receives a generic closure as an argument. In each case, we will use our verification mechanism for closures, so that the higher-order function can declare, within its own specification, how the closure is used.

In each scenario, our objective is that the specification of the higher-order function is:

- *General* with respect to the closure. We should make no unnecessary assumptions about the implementation or behaviour of the closure.
- *Understandable* by a human reader.
- *Informative* about how the closure is used by the higher-order function. The caller of the higher-order function should be able to infer useful information about how the results of the closure affect the results of the higher-order function, and about how the internal state of the closure changes after a call to the higher-order function.

For each scenario, we will present:

- A general description of the scenario, with a relevant example.
- A *specification pattern* suitable for this scenario. A *specification pattern* consists in an entailment interface used in combination with a closure specification function. These two elements can be used in instances of the scenario, in order to allow a higher-order function to describe the usage of a closure argument.
- The annotated version of the example shown at the beginning.
- An annotated call example, showing a potential caller of the higher-order function.

5.6.1 The closure is called an unspecified number of times

In this scenario, we want a higher-order function to communicate, via its post-condition, that a particular closure argument will be called 0 or more times, with any arguments. However, the function should also be able to communicate that some particular calls happened, in some unspecified order, and to express the relationship between these calls and the function's results.

5.6.1.1 Example

In the example below, higher-order function `hof` calls closure argument `f` in a different order depending on boolean parameter `choice`. The result of `hof` is the

sum of the calls `f(2)` and `f(3)`. Note that, depending on the order of these calls, the result might change.

```
1  func hof(f func(int)int)(res int) {
2      if choice {
3          res = f(2) + f(3)
4      } else {
5          res = f(3) +  f(2)
6      }
7      f(5)
8  }
```

5.6.1.2 Pattern

A potential specification pattern for this scenario is shown below.

```
1  type Calls interface {
2      pred inv()

3      ghost requires inv()
4      pure called(x int)bool

5      ghost pure res(x int, res int)bool
6  }

7  ghost
8  requires x >= 0 && cs != nil && cs.inv()
9  ensures cs.inv() && cs.called(x) && cs.res(x, res)
10 ensures forall x int :: old(cs.called(x)) ==> cs.called(x)
11 func spec(ghost cs Calls, x int) (res int)
```

The `Calls` entailment interface consists of a predicate, `inv`, and of two pure methods, `called` and `res`. `inv()` is used to capture all the properties about the state of the closure that will keep holding after any number of calls, assuming that the arguments satisfy the preconditions declared by `spec`. `called(x int)` is a pure boolean method, which is used to specify any properties of the internal state that hold, and will keep holding, after a call with argument `x`. `res(x int, r int)` is a pure boolean method, which encodes all the properties of the result `r` of a call to the closure, assuming that the argument used in the call is `x`. The properties described by `res` hold for the results of all the closure calls that use specification `spec`, regardless of the precise internal state of the closure.

Specification function `spec` has the same signature as the closure argument that we want to specify, with an additional ghost parameter `cs`, which is an instance of the interface `Calls`. The specification of `spec` makes sure that `cs.inv()` is preserved (line 8), that `cs.called(x)` holds after after the call (line 9), and that `cs.res(x, res)` holds (line 9), where `res` is the result of the call. Intuitively, `spec` makes sure that the implementations of the methods `called` and `res` provided by `cs` actually describe the behaviour of a closure that we want to specify. The postcondition at line 10 ensures that the `called` method keeps holding for old call values.

5.6.1.3 Annotated example

The snippet below shows the annotated version of function `hof` from the example at 5.6.1.1. The signature of `hof`, at line 6, accepts ghost parameter `cs`, which is an instance of entailment interface `Calls`. The precondition at lines 1-2 states that the closure argument `f` must implement closure specification instance `spec{cs}`, and that `cs.inv()` is preserved. The postconditions contain information about the usage of `f`. The postcondition at line 5 states that `f` is called with argument 2, and with argument 3. The postcondition at lines 3-4 states that the result of `hof` is obtained by adding up the results of two calls to `f`. All postconditions can be verified because the closure calls at lines 8, 10 and 12 are annotated with specification instance `spec{cs}`, so the verifier can leverage the specification of function `spec`.

```
1  requires f implements spec{cs}
2  requires cs != nil && cs.inv()
3  ensures cs.inv() && exists a int, b int ::
4      cs.res(2, a) && cs.res(3, b) && res == a + b
5  ensures cs.called(2) && cs.called(3)
6  func hof(ghost cs Calls, f func(int)int, choice bool)(res int) {
7      if choice {
8          res = (f(2) as spec{cs}) + (f(3) as spec{cs})
9      } else {
10         res = (f(3) as spec{cs}) +  (f(2) as spec{cs})
11     }
12     f(5) as spec{cs}
13 }
```

5.6.1.4 Call example

Consider the snippet below for an example of how a caller can verify a call to `hof`. In the example below, the caller defines closure `cl` at line 12. The closure captures variable `accum`. At each call, `accum` is incremented by the value indicated by the argument to the call, and it is returned as the result of the call. The specification entailment proof at lines 19-23 uses interface entailment instance `Acc{&accum}`.

The implementation of struct `Acc` is defined at lines 1-9. The predicate `inv()`, at line 2, provides access to the captured variable and states that the value remains non-negative. The implementation of `called` (line 4) says that, after a call, the value of the captured variable becomes at least as large as the argument to the call. The implementation of `res` (line 7) says that the result of the call is also at least as large and the argument.

With this implementation of the entailment interface, it is possible to verify the call at line 25, and to make the assertions at lines 26 and 28, giving a lower bound to the result of `hof` and to the new value of `accum`.

```
1  type Acc struct { accum *int }
2  pred (self Acc) inv() { acc(self.accum) && *self.accum >= 0 }
3  ghost requires self.inv()
4  pure func (self Acc) called(x int)bool {
5      return unfolding self.inv() in (*self.accum >= x)
6  }
7  ghost pure func (self Acc) res(x int, y int)bool {
```

```
8       return y >= x
9   }

10  func main() {
11      accum@ := 0
12      cl :=   requires x >= 0
13              preserves acc(&accum) && accum >= 0
14              ensures accum == old(accum) + x && y == accum
15              func accumulate(x int)(y int) {
16                  accum += x
17                  return accum
18              }

19      proof cl implements spec{Acc{&accum}} {
20          unfold Acc{&accum}.inv()
21          res = cl(x) as accumulate
22          fold Acc{&accum}.inv()
23      }

24      fold Acc{&accum}.inv()
25      r := hof(Acc{&accum}, cl, true)
26      assert r >= 5

27      unfold Acc{&accum}.inv()
28      assert accum >= 3
29  }
```

Under this scenario, as seen in the call example, the caller might not be able to know the exact state of the closure after a call to the higher-order function. In Section 5.6.3 we will see a scenario where a higher-order function provides more precise information about the usage of a closure, allowing the caller to know more precise information about the state of the closure after the call.

5.6.2 The closure is only called once

In this scenario, we want a higher-order function to communicate, via its post-condition, that a particular closure argument will be called at most once, with a particular argument. The result of the call needs to satisfy a particular specification. Since we know that the closure is only called once, however, the closure argument only needs to satisfy the required specification for the first call, not necessarily for any other call.

5.6.2.1 Example

In the example below, higher-order function `hof` calls closure argument `f` once, with argument `n`. The result of this call is then returned, so the value of the result of `hof` is the same as the value returned by the closure call. We want to ensure that the result of `hof` is greater than or equal to n, so it is necessary that the first call to `f` with any argument `n` returns a value that is at least as large as `n`.

```
1   // ensures res >= n
2   func hof(f func()int, n int)(res int) {
3       return f(n)
4   }
```

5.6.2.2 Pattern

A potential specification pattern for this scenario is shown below.

```
1  type Once interface {
2      pred invBefore()
3      pred invAfter(n int)
4  }

5  requires o != nil && o.invBefore()
6  ensures o.invAfter(n) && res >= n
7  func onceGT(ghost o Once, n int) (res int)
```

Interface `Once` has two separate predicates, one (`invBefore`) corresponding to the state before the call, one corresponding to the state after the call (`invAfter`).

Specification function `onceGT` has the same signature as the closure argument that we want to specify, with an additional ghost parameter `o`, which is an instance of the interface `Once`. The specification of `onceGT` states that `o.invBefore()` is required to hold before the call, and `o.invAfter(n)` is guaranteed to hold after the call. The result is guaranteed to be greater than or equal to `n`, as wanted. Since `onceGT` does not guarantee permissions to the `invBefore` predicate after the call, any higher-order function using this pattern will only be able to call the closure at most once.

5.6.2.3 Annotated example

The following snippet shows the annotated version of function `hof` from the example at 5.6.2.1. The function accepts ghost parameter `o`, an instance of entailment interface `Once`. As a precondition, there must be access to `o.invBefore()` and closure `f` must implement specification instance `onceGT{o}`. The function ensures that `o.invAfter()` holds after the call and that the result returned by `hof` is greater than or equal to `n`.

```
1  requires o != nil && o.invBefore() && f implements onceGT{o}
2  ensures o.invAfter(n) && res >= n
3  func hof(ghost o Once, f func(int)int, n int)(res int) {
4      return f(n) as onceGT{o}
5  }
```

5.6.2.4 Call example

The snippet below demonstrates that, as long as the postcondition `res >= n` is guaranteed by the first call to a closure, it is possible to write an implementation of `Once` that allows us to use the closure as an argument to `hof`. In the example, the closure `cl` is defined at line 6. The closure captures variable `accum`. At each call with argument `x`, `x` is added to `accum` and the new value of `accum` is returned as a result of the call.

It is not true, in general, that call `cl(n)` returns a value that is at least as large as `n`. In fact, if the value of `accum` is negative before the call, that is not the

case. However, the first call satisfies the condition, since `accum` has value 0 at the beginning. In the entailment interface proof at lines 12-16, we use struct `Acc{&accum}` as an entailment interface instance. `Acc` is defined at lines 1-3, and the definition of `invBefore` states that we are in the state before the call if the value of the captured variable `accum` is 0. Also, the definition of `invAfter` at line 3 it says that, after a call with argument `n`, the value of the captured variable is equal to `n`.

This implementation of the entailment interface allows the caller to prove the specification entailment that we need, and to call `hof`. After the call, we also know that the new value of `accum` is 5, since `Acc{&accum}.invAfter(5)` holds after the call.

```
1  type Acc struct { accum *int }
2  pred (self Acc) invBefore() { acc(self.accum) && *self.accum == 0 }
3  pred (self Acc) invAfter(n int) { acc(self.accum) && *self.accum == n }

4  func main() {
5      accum@ := 0
6      cl :=   preserves acc(&accum)
7              ensures accum == old(accum) + x && y == accum
8              func accumulate(x int)(y int) {
9                  accum += x
10                 return accum
11             }

12     proof cl implements onceGT{Acc{&accum}} {
13         unfold Acc{&accum}.invBefore()
14         res = cl(n) as accumulate
15         fold Acc{&accum}.invAfter(n)
16     }

17     fold Acc{&accum}.invBefore()
18     r := hof(Acc{&accum}, cl, 5)
19     assert r >= 5

20     unfold Acc{&accum}.invAfter(5)
21     assert accum == 5
22 }
```

5.6.3 The closure is called with specific values

In this scenario, we know exactly how a higher-order function will use a closure argument `f`. In particular, we know the precise sequence of calls to `f` within the higher-order function's body, including the arguments that are used for each call. We can use a sequence, which is a ghost type, to represent a sequence of calls, and to enable the higher-order function to add a relevant assertion in the postcondition. Each element of the call sequence will contain values that describe the arguments of the corresponding call. We will show a simple example where a single integer is enough to represent the closure call argument. However, in general we can define a type in Gobra to provide a mathematical abstraction over the argument values, in order to represent the value of the arguments without the need for access permissions.

5.6.3.1 Example

Consider the example below. In the example, higher-order function `hof` calls closure argument `f` three times. The argument is 2 for the first call, 3 for the second, and 5 for the third. As informally stated in the comments at lines 1-2, the goal is to add assertions, in the postcondition, that communicate this information to the caller.

```
1  // ensures sequence of calls to f is f(2),f(3),f(5)
2  // ensures res is the sum of the value returned by the calls
3  func hof(f func(int)int)(res int) {
4      res = f(2)
5      res += f(3)
6      res += f(5)
7  }
```

5.6.3.2 Pattern

In this case, the value returned by a call to a closure might depend on the sequence of previous calls. The pattern shown below verifies closures of type `func(ghost seq[int], int)int`, where the ghost argument represents the sequence of previous calls to the closure.

```
1  type Calls interface {
2      pred inv(ghost seq[int])

3      ghost
4      requires len(calls) > 0
5      pure res(ghost calls seq[int]) int
6  }


7  ghost
8  requires cs != nil && cs.inv(prev)
9  ensures cs.inv(prev ++ seq[int]{x}) &&
10             res == cs.res(prev ++ seq[int]{x})
11 func spec(ghost cs Calls, ghost prev seq[int], x int) (res int)
```

Consider the `Calls` entailment interface (lines 1-6) first. The predicate `inv`, at line 2, contains information about the internal state of the closure, *which depends on the sequence of calls made so far*. Function `res`, at line 5, returns an integer that corresponds to the result returned by the closure after the calls described by sequence `calls`.

The specification function `spec`, defined at lines 7-11, has ghost arguments `cs`, which is an instance of entailment interface `Calls`, and `prev`, a sequence describing the previous calls to the closure. The precondition at line 8 states that the predicate `inv(prev)` holds, which means that the closure has already been called with the arguments specified by `prev`. The postcondition at lines 9-10 states that `inv` now holds with argument `prev ++ seq[int]x`, and `res` matches the value returned by `cs.res` when called with the same argument. Intuitively, we are stating that, after the call, the internal state of the closure can depend on all the closure call arguments used so far, both in the previous calls, and in the current one.

5.6.3.3 Annotated example

The following snippet shows the annotated version of function `hof` from the example at 5.6.3.1. The function accepts ghost parameter `cs`, an instance of entailment interface `Calls`. Also, the type of the closure parameter is changed to include, among the arguments, a sequence describing the arguments of the previous calls to the closure. As usual, the precondition at line 1 requires that the closure implements specification instance `spec{cs}`. The precondition at line 2, instead, requires that `cs.inv` holds with the empty sequence as argument, meaning that the closure has not been called yet. The postcondition at line 3 ensures that, when `hof` returns, the closure has been called three times, with arguments 2,3,5. Line 4 informs the caller that the result of `hof` is obtained by adding the values returned by all three calls to the closure. Each call in the body is annotated with specification instance `spec{cs}`. Also, a ghost argument is added to each call. The ghost argument is a sequence describing the argument values used for previous calls.

```
1  requires f implements spec{cs}
2  requires cs != nil && cs.inv(seq[int]{})
3  ensures cs.inv(seq[int]{2,3,5})
4  ensures res == cs.res(seq[int]{2}) + cs.res(seq[int]{2,3}) +
5      cs.res(seq[int]{2,3,5})
6  func hof(ghost cs Calls, f func(ghost seq[int], int)int)(res int) {
7      res = f(seq[int]{}, 2) as spec{cs}
8      res += f(seq[int]{2}, 3) as spec{cs}
9      res += f(seq[int]{2, 3}, 5) as spec{cs}
10 }
```

Consider the snippet below for an example of how a caller can verify a call to `hof`. In this case, the caller is able to know precisely the value of the captured variable after the call to `hof`, because the postcondition of `hof` gives a precise indication of which calls have been made.

The closure `cl` is defined at line 16, and it is an accumulator function analogous to the closure defined in the example analysed in 5.6.2.3. The implementation of entailment interface `Calls` is `Acc`, defined at lines 6-10. Predicate `inv`, defined at line 7, states that, after the calls described by argument `calls`, the value of the captured variable corresponds to the sum of the elements of `calls`. Similarly, `res` (at lines 10-13) states that the result, after the calls described in argument `calls` corresponds to the sum of all the elements of `calls`. In order to express these properties, it is necessary to define the sum of a sequence using pure function `seqSum`, defined at lines 1-6.

It is worth taking a look at the specification entailment proof at lines 23-28. As in the other examples, there are `unfold` and `fold` statements at lines 24 and 27. In this case, however, we also need the assertion at line 26, which helps the verifier to know that there are enough permissions for the `fold` statement at line 27. After the proof, it is possible to call `hof` at line 30. The postcondition of `hof` allows us to know the exact value of the result of the call (as shown by the assertion at line

30), and the new value of the captured variable `accum` (as shown by the assertion at line 33).

```
1  ghost
2  ensures forall k int :: k > 0 && len(s) == k ==>
3                  res == s[k-1] + seqSum(s[:(k-1)])
4  pure func seqSum(s seq[int]) (res int) {
5      return len(s) == 0 ? 0 : (s[len(s)-1] + seqSum(s[:(len(s)-1)]))
6  }

7  type Acc struct { accum *int }
8  pred (self Acc) inv(ghost calls seq[int]) {
9      acc(self.accum) && *self.accum == seqSum(calls)
10 }
11 ghost
12 pure func (self Acc) res(ghost calls seq[int])int {
13     return seqSum(calls)
14 }

15 func main() {
16     accum@ := 0
17     cl :=   preserves acc(&accum)
18             ensures accum == old(accum) + x && y == accum
19             func accumulate(ghost seq[int], x int)(y int) {
20                 accum += x
21                 return accum
22             }

23     proof cl implements spec{Acc{&accum}} {
24         unfold Acc{&accum}.inv(prev)
25         res = cl(prev, x) as accumulate
26         assert (prev ++ seq[int]{x})[:(len(prev))] == prev
27         fold Acc{&accum}.inv(prev ++ seq[int]{x})
28     }

29     fold Acc{&accum}.inv(seq[int]{})
30     r := hof(Acc{&accum}, cl)
31     assert r == 2 + (2 + 3) + (2 + 3 + 5)

32     unfold Acc{&accum}.inv(seq[int]{2, 3, 5})
33     assert accum == 2 + 3 + 5
34 }
```

5.7 Examples of common higher-order functions

The following examples show the verification of the two common higher-order functions `all` and `map` defined over slices. Both functions have in common the property that they call an argument closure over all elements of the slice. Due to their wide use, these functions are a common case study in the verification literature. In particular, the examples shown here were inspired and adapted from Wolff et al. [14], which uses them as case studies for the verification of higher-order functions in Rust. We also show the verification of the `fold` function, defined over a custom list type. The last example demonstrates that our mechanism for the verification of closures and higher-order functions is also applicable when the higher-order function uses a custom data type with custom permissions.

5.7.1 All

We define the `all` function on integer slices. The function, shown below, accepts a closure of type `func(int)bool` and an integer slice as arguments, and returns a boolean as a result, which is `true` if and only if all elements of the slice satisfy the condition defined by the closure. Here, we assume that the closure is pure and that it does not have an internal state. However, these limitations could be lifted by using a slightly more complex specification pattern than the one presented here. It is up to the programmer writing the higher-order function to decide on the right balance between having fewer limitations, and making the specification more complex.

An implementation of `all` in Go is shown below. In the implementation, we use a for-loop to go over the element, and we return `false` if we encounter a value that does not satisfy the condition described by `f`. If the loop terminates without returning `false`, then we return `true`.

```
1  func all(v []int, f func(int)bool)bool {
2      for i := 0; i < len(v); i += 1 {
3          if !f(v[i]) {
4              return false
5          }
6      }
7      return true
8  }
```

The annotated version is shown below. As usual, we use a specification pattern consisting of an entailment interface (`Property`, at lines 1-3), and of a specification function (`property`, at lines 4-6). It is a very simple pattern: the entailment interface just defines a pure boolean function `holds`, and the result of specification function `property` matches the value of this function, as specified at line 5.

Function `all` is defined at lines 7-20. In addition to the actual arguments, it receives ghost argument `prop`, an instance of entailment interface `Property`. The precondition at line 7 specifies that the closure `f` must implement specification instance `property{prop}`, ensuring that the result of any call to the closure with argument `n` matches the result of `prop.holds(n)`. Also, the `all` function preserves read permissions to all elements of slice `v`.

The call at line 17 is annotated with specification instance `property{prop}`, so that the call to the closure can be made. The loop invariant at line 9 asserts that, at each loop iteration, the property holds for every element that we have seen so far. This invariant enables the verification of the postcondition at line 9, which states that the result has value `true` if and only if the property holds for all elements of the slice.

```
1  type Property interface {
2      ghost pure holds(int)bool
3  }
4  ghost
```

```
5  ensures prop != nil && res == prop.holds(n)
6  pure func property(ghost prop Property, n int) (res bool)

7  requires prop != nil && f implements property{prop}
8  preserves forall i int :: i >= 0 && i < len(v) ==> acc(&v[i], 1/2)
9  ensures res == forall i int :: i >= 0 &&
10             i < len(v) ==> f(v[i]) as property{prop}
11 func all(v []int, ghost prop Property, f func(int)bool)(res bool) {
12     invariant i >= 0 && i <= len(v)
13     invariant forall j int :: j >= 0 && j < len(v) ==> acc(&v[j], 1/2)
14     invariant forall j int :: j >= 0 && j < i ==>
15             f(v[j]) as property{prop}
16     for i := 0; i < len(v); i += 1 {
17         if !(f(v[i]) as property{prop}) {
18             return false
19         }
20     }

21     return true
22 }
```

Any caller of this function will need to provide an implementation of `Property`, whose `holds` method returns the same result as closure `f`. Therefore, after the call, the caller knows whether the result is `true` or `false`, depending on the values in `v` and on the concrete behaviour of `f`.

5.7.2 Map

We define the `map` function on integer slices. Since `map` is a keyword that is already used in Go (it defines a type), the function will be named `map_vec` in the example. The function, shown below, accepts a closure of type `func(int)int` and an integer slice `v` as arguments. There are no results, but the values of `v` are changed. In particular, the closure is called for each element of the slice, in order, and each value `v[i]` is replaced by the result of the closure called giving, as argument, the original value of `v[i]`.

```
1  func map_vec(v []int, f func(ghost seq[int], int)int) {
2      for i := 0; i < len(v); i += 1 {
3          v[i] = v[i]
4      }
5  }
```

The annotated function is shown below. The verification of `map_vec` uses the same specification pattern as described in 5.6.3, since we know exactly the sequence of calls made inside the function, and the result of a call may depend on previous calls. Refer to 5.6.3.2 for a description of the pattern. The function also uses ghost pure function `toSeq` to define the sequence corresponding to a slice (i.e. the sequence that contains the same elements as the slice).

Function `map_vec` is defined at lines 20-38. In addition to the actual arguments, it receives ghost argument `cs`, an instance of entailment interface `Calls`. The precondition at line 20 states that `cs.inv` holds for the empty sequence, and closure `f` implements specification instance `fspec{cs}`. The precondition at line 21 provides write access to all the elements of `v`.

```
1  type Calls interface {
2      pred inv(ghost seq[int])
3      ghost
4      requires len(calls) > 0
5      pure res(ghost calls seq[int]) int
6  }

7  ghost
8  requires cs != nil && cs.inv(prev)
9  ensures cs.inv(prev ++ seq[int]{x}) && res == cs.res(prev ++ seq[int]{x})
10 func fspec(ghost cs Calls, ghost prev seq[int], x int) (res int)

11 ghost
12 requires forall i int :: i >= 0 && i < len(v) ==> acc(&v[i], _)
13 ensures len(s) == len(v)
14 ensures forall i int :: i >= 0 && i < len(v) ==> s[i] == v[i]
15 pure func toSeq(v []int) (s seq[int]) {
16     return len(v) == 0 ?
17         seq[int]{} :
18         (toSeq(v[:len(v)-1]) ++ seq[int]{v[len(v)-1]})
19 }

20 requires cs != nil && cs.inv(seq[int]{}) && f implements fspec{cs}
21 requires forall i int :: i >= 0 && i < len(v) ==> acc(&v[i])
22 ensures cs.inv(old(toSeq(v)))
23 ensures forall i int :: i >= 0 && i < len(v) ==> acc(&v[i]) &&
24     v[i] == cs.res(old(toSeq(v))[:(i+1)])
25 func map_vec(v []int, ghost cs Calls, f func(ghost seq[int], int)int) {
26     ghost s := toSeq(v)
27     invariant i >= 0 && i <= len(v)
28     invariant cs.inv(s[:i])
29     invariant forall j int :: j >= 0 && j < len(v) ==> acc(&v[j])
30     invariant forall j int :: j >= 0 && j < i ==> v[j] == inv.res(s[:(j+1)])
31     invariant forall j int :: j >= i && j < len(v) ==> v[j] == s[j]
32     for i := 0; i < len(v); i += 1 {
33         assert v[i] == s[i]
34         v[i] = f(s[:i], v[i]) as fspec{cs}
35         assert s[:(i+1)] == s[:i] ++ seq[int]{s[i]}
36         assert v[i] == cs.res(s[:(i+1)])
37     }
38 }
```

The most interesting part of the specification are the postconditions. The postcondition at line 22 states that the arguments used for calls to closure `f` correspond to the original values of the elements of `v`, in this order. The postcondition at line 23 states that the new value of each element `v[i]` is the result of the (i+1)-th call, and the arguments to the calls executed before this one correspond to the original values of elements of `v` placed before the i-th element.

The loop inside the body is annotated in order to verify the postconditions. The most interesting invariants are at lines 28 and 30. At line 28, we keep track of the calls that have already taken place. At line 30, we keep track of each of the updated values of `v`.

5.7.3 Fold

It is possible to verify the `fold` function over integer slices similarly to the `map` function. Here, we will analyse a different version, which uses a custom list type,

and which we will call `foldList`. This variation of the `fold` function was proposed by Svendsen at al. [9], as part of a case study involving the in-place reversal of a list. The example at the end of this section contains the full, annotated Gobra translation of the case study, which uses the `foldList` function to reverse a list in place.

First of all, let us introduce the list type, which we define via a `Node` struct that can be seen below. The struct consist of an integer value, and a pointer to another node, which is the next element in the list. We also introduce predicate `list` and ghost pure function `toSeq`. `list` is recursively defined to contain the permission to access all elements of the list. `toSeq` allows to convert, within ghost code, a list to the corresponding sequence of integers (i.e. the sequence which contains the same values in the same order).

```
1  type Node struct { elem int; next *Node }

2  pred list(l *Node) {
3      l != nil ==> acc(l) && list(l.next)
4  }

5  ghost
6  requires list(l)
7  ensures l != nil ==> len(res) > 0
8  ensures l != nil ==> unfolding list(l) in
9          (l.elem == res[0] && toSeq(l.next) == res[1:])
10 pure func toSeq(l *Node) (res seq[int]) {
11     return l == nil ?
12         seq[int]{} :
13         unfolding list(l) in (seq[int]{l.elem} ++ toSeq(l.next))
14 }
```

We can now define the `foldList` function, shown below. The function has argument `l`, a `Node` pointer which is the head of a list, and `f`, a closure that performs some operation on a `Node`. `f` has access to a single node, which it can transform in any way: it can change its next element, or its value. For example, it is possible to define `f` so that, when the `foldList` function terminates, the resulting list has a single node and its value is the sum of all values in the original list. A further example, showing the reversal of a list using `foldLeft`, is shown at the end of this section.

```
1  func foldList(l *Node, f func(*Node)) {
2      if l != nil {
3          tmp := l.next
4          f(l)
5          foldList(tmp, f)
6      }
7  }
```

The annotated function is shown below. The entailment interface `Inv` (lines 1-3) defines a predicate `inv`, which depends on a sequence of integers. The sequence describes the arguments of calls to the closure made so far. The specification function `fspec` (lines 4-8) has ghost parameters `inv` and `prev`. `inv` is an instance of entailment interface `Inv`, while `prev` is a sequence of integers. The precondition at line 5 makes sure that `prev` describes the arguments of previous calls to the

closure, while the postconditions at line 5 makes sure that `inv.inv(newseq)` is established, where `newseq` describes the argument values of calls made until now, including the current call. The precondition at line 6 states that there is access to the node `x`.

Function `foldList` is defined at lines 9-22. In addition to the original arguments, the function receives ghost arguments `inv` and `prev`. `inv` is an instance of entailment interface `Inv`, while `prev` is a sequence of integers, describing the argument of calls to `f` that have been made before the current `foldList` call. Closure `f` also accepts a sequence of integers as an additional ghost argument, representing the previous calls.

The pre- and postconditions at lines 10 and 11 state, intuitively, that `foldList` calls closure `f` with all the current values contained in list `l`, in order (the corresponding values are appended to `prev`).

```
1  type Inv interface {
2      pred inv(ghost seq[int])
3  }

4  ghost
5  requires inv != nil && inv.inv(prev)
6  requires x != nil && acc(x)
7  ensures inv.inv(prev ++ seq[int]{old(x.elem)})
8  func fspec(ghost inv Inv, ghost prev seq[int], x *Node)

9  requires f implements fspec{inv}
10 requires inv != nil && inv.inv(prev) && list(l)
11 ensures inv.inv(prev ++ old(toSeq(l)))
12 func foldList(l *Node, ghost inv Inv, ghost prev seq[int],
13                f func(ghost seq[int], *Node)) {
14     if l != nil {
15         ghost s := toSeq(l)
16         unfold list(l)
17         tmp := l.next
18         assert s == seq[int]{l.elem} ++ toSeq(tmp)
19         f(prev, l) as fspec{inv}
20         foldList(tmp, inv, prev ++ seq[int]{s[0]}, f)
21     }
22 }
```

5.7.3.1 Example: reverse a list using the foldList function

Figure 5.1 shows the annotated example of a caller using the function `foldLeft` (defined in 5.7.3) to reverse a list in place. The example is adapted from Svendsen at al. [9].

Method `reverseSeq` (lines 1-11) defines a reversed sequence.

Struct `Reverse` (lines 12-17) implements the entailment interface `Inv`. It contains a field `box` of type `*Box`, where `Box` is a structure containing a pointer to a node. Struct `Box` had to be introduced because Gobra does not support struct fields with double-pointer type yet (so, `Reverse` cannot have a field of type `**Node`). The `inv` predicate at lines 14-17 states that, at each call, node `self.box.head` is the head of a list that is the reverse of the list corresponding to `prev`.

```
1  ghost
2  ensures forall sq seq[int], n int :: reverseSeq(sq ++ seq[int]{n}) ==
3                                       seq[int]{n} ++ reverseSeq(sq)
4  ensures len(res) == len(s)
5  ensures forall i int :: i >= 0 && i < len(s) ==>
6      s[i] == res[len(s)-i-1]
7  pure func reverseSeq(s seq[int])(res seq[int]) {
8      return len(s) == 0 ?
9          seq[int]{} :
10         (seq[int]{s[len(s)-1]} ++ reverseSeq(s[:(len(s)-1)]))
11 }

12 type Box struct { head *Node }
13 type Reverse struct { box *Box }
14 pred (self Reverse) inv(ghost prev seq[int]) {
15     acc(self.box) && list(self.box.head) &&
16         toSeq(self.box.head) == reverseSeq(prev)
17 }

18 func main() {
19     fold list(nil)
20     n3@ := Node{3, nil}
21     fold list(&n3)
22     n2@ := Node{2, &n3}
23     fold list(&n2)
24     n1@ := Node{1, &n2}
25     fold list(&n1)
26     assert toSeq(&n1) == seq[int]{1,2,3}

27     var box@ Box = Box{nil}
28     flip := requires acc(x) && acc(&box) && list(box.head)
29             ensures acc(&box) && list(box.head) && box.head == x
30             ensures unfolding list(box.head) in
31                 (box.head.elem == old(x.elem) &&
32                     box.head.next == old(box.head) &&
33                     toSeq(x.next) == old(toSeq(box.head)))
34             func flip_(ghost seq[int], x *Node) {
35                 x.next = box.head
36                 box.head = x
37                 fold list(box.head)
38             }

39     proof flip implements fspec{Reverse{&box}} {
40         unfold Reverse{&box}.inv(prev)
41         flip(prev, x) as flip_
42         fold Reverse{&box}.inv(prev ++
43             seq[int]{unfolding list(box.head) in (box.head.elem)})
44     }

45     fold list(box.head)
46     fold Reverse{&box}.inv(seq[int]{})
47     foldList(&n1, Reverse{&box}, seq[int]{}, flip)
48     unfold Reverse{&box}.inv(seq[int]{1,2,3})
49     assert toSeq(box.head) == seq[int]{3,2,1}
50 }
```

Figure 5.1: A method implementing the in-place reversal of a list using the `foldList` function

The closure `flip`, defined at lines 28-38, captures variable `box` of type `Box`, which, at all times, points to the head of the reversed list being constructed. When `flip` is called, the node `x` given as an argument becomes the new head of the list, and its next element becomes the current `box.head`. As the assertion at line 49 verifies, a call to `foldList` with this closure reverses the list passed as an argument to `foldList`.

Chapter 6

Evaluation

This chapter presents the setup and results of our experimental evaluation. Section 6.1 explains how the example programs for the evaluation have been selected. Section 6.2 provides a brief descriptions of the examples that have not already been presented in the Case Studies chapter. Section 6.3 presents the setup of our experiments, and explains which data we have collected. In Section 6.4 we summarise the results of the evaluation in a table, and we make some comments about the results.

6.1 Selection of the examples

The example programs chosen for the evaluation come from two main sources.

The first source is the paper by Wolff et al. that introduced support for closures in Prusti [14]. In particular, we have taken the example programs used by the paper in their experimental evaluation[1]. We have translated each of these example programs to an annotated Go program, using our methodology to verify closures and higher-order functions. In particular, the example `fold_list_rev` is translated from Svendsen et al. [9], the paper which originally introduced the corresponding case study. For the other examples, even those inspired by the pre-existing literature, we have used the corresponding Rust implementation from Wolff et al. [14] as a source for the translation.

The remaining example programs are representative of the case studies analysed in chapter 5. For each case study, we have written a corresponding example program. We have not written an additional example program for case studies already covered by, or similar to, one of the example programs taken from Wolff et al. [14]. In particular, we have not written an additional example for case study

[1] We have ignored the example programs `option`, `option_map_err` and `result_uoe`, because they specifically involve the higher-order functions provided by Rust types that are not easily translatable to Go.

5.1 (corresponding to example program `counter`), 5.7.1 (`all`), 5.7.2 (`map_vec`), and 5.7.3 (`fold_list_rev`).

6.2 Brief description of the examples

Refer to table 6.1 for a full list of the example programs used in the evaluation. In this section, we provide a brief description of the examples which have not already been mentioned in chapter 5.

The examples with suffix `_err` are negative examples, which are expected to cause verification errors. They are analogous to the corresponding positive examples, with small changes that make the verification fail. For instance, some of the negative examples contain closure calls that do not satisfy the precondition of the call, or an incorrect specification entailment proof. In other negative examples, we make incorrect assertions after a call to a higher-order function.

The `delegation` example was originally introduced by Kassios and Müller [5]. The example consists of a function `f` which receives, as parameters, a pointer `y` to an integer and pointers to two closures. The purpose of the example is to show how we verify that `f` can call the two closures, and that either call does not interfere with the preconditions of the closures or with the value of `y`.

The `blameassgn` example was originally inspired by Findler and Felleisen [2]. The example verifies whether our implementation is able to assign the blame correctly for the violation of the contractual obligations defined by a higher-order function, which accepts a closure as a parameter. In particular, the higher-order function is to blame if it calls the closure without satisfying its precondition. Conversely, the caller of the higher-order function is to blame if it provides a closure which does not satisfy the specification requirements defined by the higher-order function.

The `repeat_with_n` example consists of a higher-order function that receives a closure as an argument and calls it `n` times, where `n` is a parameter of the higher-order function. The closure has no arguments, and the `n` results are saved in a vector, which is returned by the higher-order function. To verify this example, we followed the same strategy as the one used fot the *map* function (5.7.2).

The `any` example is similar to the *all* example (5.7.1). In fact, the higher-order function `any` accepts a closure of type `func(int)bool` and a slice of integer as arguments, and returns a boolean as a result, analogously to the `all` function. In this case, however, the result of `any` is true if *at least one* of the elements of the slice satisfies the condition defined by the closure.

6.3 Setup of the experiments

All experiments were executed on a warmed-up JVM on a laptop with a 1.6 GHz 8-Core Intel Core i5-8250U CPU and 8 GB of RAM, running Ubuntu 22.04 and

OpenJDK 17.0.2. We measured the total verification time for each experiment, i.e. the time it takes to encode the Gobra program to Viper, and to verify the encoded Viper program. The runtime is averaged over twelve executions, excluding the slowest and fastest outliers. For each experiment, we also collected the length (in lines of code) of the encoded Viper file.

We transformed five example programs into similar programs without closures. In these transformed programs, we removed all the closures and closure-related specifications. We replaced the higher-order functions with functions that, instead of calling a closure, compute on their own any result that they would originally get from a closure call. Therefore, the transformed programs have a similar size and complexity as the originals, except that they do not use closures. Thus, we can use the verification time of these transformed programs to get an idea of the impact on the verification time of our mechanism for the verification of closures.

We also ran the original experiments from the paper by Wolff et al. [14], in order to make a comparison with our results. We ran their experiments using the script provided as part of the related artefact [16], running it on the same machine as the other experiments. Analogously to the other experiments, we averaged the runtime over twelve executions, excluding the slowest and fastest outliers.

6.4 Results

Table 6.1 shows the results of our experiments. The caption explains the meaning of each column in the table.

The annotation overhead (lines of specification / lines of code) varies between 0.9 and 3.0, which is a typical range for deductive verifiers similar to Gobra. This range is also in line with the annotation overhead for Gobra programs that do not use closures. In fact, the annotation overhead reported in the Evaluation section of the paper introducing Gobra [13] is between 0.3 and 3.1.

The lines of encoded Viper code (excluding blank lines and comments) are between 4.5 and 8.9 times as many as the combined lines of code and annotations in the original Gobra file (where lines with both code and annotations are counted twice). These values are slightly higher than the ones we can find for Gobra programs without closures, which are between 1.9 and 7.8. The fact that the encoding is longer in the presence of closures is expected, since the encoding of closures generates several Viper members for each closure specification instance. It is worth noting, however, that a difference in the size of the generated Viper code does not affect the user, since the encoding is not exposed to the user.

As one might expect, the verification time for the examples where higher-order functions specify how a closure argument is used is generally higher than for the other examples. However, the verification time mostly depends on the complexity of the closures used by the caller of a higher-order function. For example, if the

Table 6.1: The list of the example programs considered in the evaluation, and the corresponding experimental results. The examples named `_err` are negative examples (whose verification fails as expected). For each experiment, we report: the corresponding case study in chapter 5, if there is one (CS); whether the example corresponds to an equivalent example from Wolff et al. [14] (FP); the lines of Go code (LOC); the lines of specification (LOS); the lines of code in the encoded Viper file (VLC); the verification time in seconds (T[s]). If a line includes both code and annotations, it is counted both in LOC and LOS. For some of the experiments, we also report the verification time for a similar program that does not use closures (T_L[s]), and the verification time for the corresponding example in the Prusti paper [14] (T_P[s]).

#	example	CS	FP	LOC	LOS	VLC	T[s]	T_L[s]	T_P[s]
1	counter	5.1	✓	15	29	277	1.01		6.45
	counter_err		✓	14	27	274	0.95		6.31
2	delegation		✓	20	36	331	1.13		6.48
3	blameassgn		✓	11	12	173	0.79		5.85
	blameassgn_err		✓	19	17	203	0.78		7.11
4	factorial	5.2		13	14	155	0.95	0.76	
5	even_box	5.3		16	19	307	1.05	0.85	
6	call_ctr_n_tms	5.4		14	19	211	0.86		
7	get_counter	5.5		13	34	325	1.17		
8	cdesc_imprecise	5.6.1		18	45	349	1.13		
9	cdesc_once	5.6.2		13	34	313	1.05		
10	cdesc_precise	5.6.3		15	45	320	1.16		
11	map_vec	5.7.2	✓	33	100	664	3.07		11.27
12	repeat_with_n		✓	17	41	390	1.56	1.34	14.43
13	fold_list_rev	5.7.3	✓	23	59	371	1.76		6.96[2]
14	any		✓	39	49	616	1.65	1.37	9.93
	any_err		✓	40	45	614	1.87		10.19
15	all	5.7.1	✓	39	48	615	1.64	1.43	9.04

result of a closure depends on the entire sequence of previous calls to the closure (which is the case in `map_vec`), the verification takes significantly more time than for closures whose result only depends on the arguments.

Since closures introduce more complexity to a program and its encoding, the verification of programs with closures takes slightly longer than the verification of similar examples which do not use closures. However, the increase in the verification time is limited (none of our experiments show an increase exceeding 25%). Furthermore, successful and failed verifications take around the same amount of time, which is desirable, especially if users verify their programs interactively.

Lastly, our results show that Gobra takes significantly less time to verify Go programs using closures than the amount of time required by Prusti to verify equivalent Rust programs. The absolute difference in performance is not particularly

[2] In the Prusti closures paper [14], example `fold_list_rev` is encoded directly in Viper, so the runtime reported in the table only refers to the verification of the encoded Viper file, and does not include the encoding time.

insightful, since the encodings used in Gobra and Prusti are very different. However, we can notice some similarities and differences in the relative verification time of the examples. Similarly to Gobra, examples with call descriptions take the longest. However, the verification time in Gobra does not always increase if the verification time of Prusti increases. Gobra has a much higher annotation overhead than Prusti (consult the Experimental Evaluation Section of Wolff et al. [14] for a comparison). This is expected, since Prusti exploits the restrictions of Rust's type system in order to allow simpler annotations.

Chapter 7

Conclusion

We have introduced support for the verification of closures in Go programs, extending the functionality of the Gobra verifier. We have devised a *modular* design that is consistent with the existing design of Gobra. The design consists of very simple new verification primitives, which give the user the flexibility required to verify a wide range of programs using closures.

Our design is based on two core ideas: *specification entailment* and *entailment interfaces*. Specification entailment is inspired by previous works [14, 15, 12], but we have adapted their ideas to a setting where both user-defined permissions and captured variables are allowed. Entailment interfaces, instead, are an original idea. Using the two core ideas of specification entailment and entailment interfaces, our design can effectively handle the captured state of closures, and higher-order functions. Furthermore, higher-order functions can describe, in a wide variety of cases, how they use a closure argument, without violating the modularity of our design.

In our case study, we have analysed several examples, which showcase that our design is suitable for the verification of a large range of scenarios. Our experimental evaluation shows that our implementation of the design is usable in practice. In fact, we have found that the verification time of our experimental example programs is in line with the verification time of similarly-sized Go programs which do not use closures.

Our design is not strongly dependent on any Go-specific features. In fact, the ideas of specification entailment and entailment interfaces can be implemented using any language that provides some form of inheritance or interfaces. Therefore, we expect our approach to be easily applicable to other verification tools similar to Gobra.

The next section briefly presents existing literature that has inspired our work. Then, we suggest potential future work, including outstanding issues that need to be addressed and possible improvements and extensions to the design.

7.1 Related work

Our design for specification entailments is inspired by two previous works. The first work that inspired our design is the paper by Wolff et al. [14] on the verification of closures in Prusti, a verifier for Rust. The second is Weber's Master thesis [12] on the verification of closures in Python.

The two main ideas behind Wolff at al.'s work are *specification entailment* and *call descriptions*. The concept of *specification entailment* is defined similarly to how we define it in Gobra. *Call descriptions* are primitives used, within the specification of a higher-order function, to specify how a closure argument is used and how the calls to the closure influence the results of the higher-order function. However, Wolff et al.'s approach for both specification entailment and call descriptions relies on guarantees provided by Rust's type system, which are not provided by Go. For example, their approach does not support user-defined permissions since Rust's type system allows Prusti to infer the required permissions automatically. Since Go's type system does not provide the same guarantees as Rust's type system, their approach is not suitable to be applied directly in Go (as explained in more detail in Section 3.3 and 3.4.1).

Weber's thesis introduces *call slots*, i.e. annotations for closure calls which include a specification for the call, and a proof that the closure implements the specification. Weber's approach supports user-defined permissions. However, their approach fails to verify closures with a captured state, which is an essential part of our work.

The problem of automatically verifying closures using an SMT-based verifier was also considered by Kassios and Müller [5] and by Nordio et al. [7]. Both works propose a methodology based on first-order logic, and they provide an encoding into Boogie, an SMT-based verifier that is also used by Viper. The methodologies proposed by the two works are similar, and they are both based on ideas related to the concept of specification entailment.

Other works, including those by Svendsen et al. [9] and Kanig et al. [4], address similar problems, but their mechanisms for the verification of closures or higher-order functions require the use of higher-order separation logic, which is unsuitable for SMT-based solvers. Such mechanisms can only be encoded using theorem provers that support higher-order logic, such as Coq.

7.2 Future work

Fix termination checks for closures

As pointed out in Section 4.4.6, the termination checks for closures are unsound in some scenarios. In Section 4.5.3, we have sketched an approach which can be used as a starting point to find a solution to the problem.

Invocation of closures in a goroutine

Goroutines are a feature of Go which enables concurrency. They are lightweight threads, which can be started by adding the keyword `go` before a function or method call. Gobra already supports the creation of goroutines using ordinary function or method calls [13]. Also, the encoding of closure calls is similar to the encoding of regular function calls. Therefore, we expect the addition of support for the creation of goroutines via closure calls to be straightforward.

Syntactic sugar for closure literals

With our current design, to call a higher-order function, it is always necessary to write a proof that the closure used as an argument satisfies the required specification. However, in many cases, the proof is trivial. We propose to introduce some syntactic sugar, so that, in these simple cases, a user can prove a specification entailment directly within the specification of a function literal. Section 3.5.1 explains this proposal in more detail.

Syntactic sugar for call descriptions

In Section 5.6 we have presented some common *specification patterns*. These patterns can be used to enable a higher-order function to describe how they use a closure argument. We propose to introduce some syntactic sugar for the most common specification patterns to reduce the annotation overhead.

Apply our approach to verify closures in other languages

As mentioned in the conclusion (Chapter 7), our design does not rely on features that are exclusive to Go or Gobra. Therefore, we propose to apply our approach to similar verification tools, to verify closures in other languages. For example, a potential candidate is Nagini [1], a verifier for Python which, analogously to Gobra, supports user-defined permissions and uses the Viper verification framework.

Bibliography

[1] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing. DOI: 10.1007/978-3-319-96145-3_33.

[2] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):4859, Sep 2002. DOI: 10.1145/583852.581484.

[3] A tutorial on gobra. URL: https://github.com/viperproject/gobra/blob/master/docs/tutorial.md (visited 10th Aug 2022).

[4] Johannes Kanig and Jean-Christophe Filliâtre. Who: A verifier for effectful higher-order programs. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, page 3948, New York, NY, USA, 2009. Association for Computing Machinery. DOI: 10.1145/1596627.1596634.

[5] Ioannis T. Kassios and Peter Müller. Specification and verification of closures. Technical report, ETH Zürich, 2010. Technical Reports D-INFK. DOI: 10.3929/ethz-a-006843251.

[6] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016. DOI: 10.1007/978-3-662-49122-5_2.

[7] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, pages 79–96, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-13953-6_5.

[8] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. DOI: 10.1109/LICS.2002.1029817.

[9] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 175–199, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-14107-2_9.

[10] D. A. Turner. Some history of functional programming languages. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-40447-4_1.

[11] Viper tutorial. URL: http://viper.ethz.ch/tutorial (visited 6th Aug 2022).

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

[13] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021. DOI: 10.48550/arXiv.2105.13840.

[14] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers. Modular specification and verification of closures in rust. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 5, New York, NY, USA, oct 2021. ACM. DOI: 10.1145/3485522.

[15] Fabian Wolff. Verification of closures in rust programs. Master's thesis, ETH Zürich, 2020. DOI: 10.3929/ethz-b-000444764.

[16] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. Modular Specification and Verification of Closures in Rust (artefact), September 2021. DOI: 10.5281/zenodo.5482557.

[17] Cheng Xuan. Verifying termination of go programs. Bachelor's thesis, ETH Zürich, 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):

Verification of closures for Go programs

Authored by (in block letters):

For papers written by groups the names of all authors are required.

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

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, 15/08/2022	Stefan Milizia

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