ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verifying Termination of Go Programs

Bachelor's Thesis

Cheng Xuan

September 16, 2021

Advisors: Prof. Dr. Peter Müller, João C. Pereira, Linard Arquint
Department of Computer Science, ETH Zürich

Abstract

Go (or Golang) is a programming language which facilitates the development of scalable concurrent programs due to its characteristics. In order to prove the functional correctness of Go programs, the Programming Methodology Group at ETH has developed a deductive verification tool called Gobra. Currently, Gobra is only able to verify the partial correctness which is insufficient in some cases to avoid bugs which may cause system crashes or unexpected behaviour. Therefore, we are interested in extending Gobra's functionality to support termination checks such that the total correctness of Go programs can be verified by Gobra. Typically, the verification of termination is proceeded by using termination measures. However, this approach is unsound while proving termination of programs in modern languages such as Go, which allow for subtyping and dynamic method binding. In order to check termination in this setting, we propose and implement an extension of the termination measure approach in Gobra based on behavioral subtyping. Furthermore, we design and realize heuristics to infer termination measures automatically when possible to free users from this task. Finally, we apply our implementation to verify termination of several Gobra programs, including verifying termination of code from the SCION codebase in order to evaluate the performance and effectiveness of our work

Contents

Chapter 1

Introduction

1.1 Motivation

The open-source programming language Go [1] was developed at Google in 2007. Nowadays, it has been widely used not only to develop web applications such as Netflix but also in implementing large-scale and open-source projects like Docker, Ethereum, and SCION. Similar to many other modern programming languages, Go provides features such as garbage collection, a strong static type system, and sophisticated testing libraries to avoid errors. However, there are still bugs which are usually missed by testing. Such bugs may cause system crashes or unexpected behaviour. To tackle this problem for Go, the Programming Methodology Group at ETH developed Gobra which serves as a verifier to prove that every execution of a given Go program satisfies certain properties, including (1) memory safety, i.e., no invalid memory location is accessed, (2) crash-freedom, and (3) freedom from race-conditions. In general, Gobra expects its users to annotate source code with specification in the form of contracts to functions and methods. Internally, Gobra translates the Go program and its annotations to a program in the intermediate verification language Viper which is finally verified by Viper's backend verifier. If the verification succeeds, the program is guaranteed to behave according to the specification and therefore it notifies the user that Gobra found no errors. Otherwise, Gobra translates the error message from Viper back to Gobra's level in such a way that the translated error message is understandable by Gobra users without needing any insights into the encoding

Viper is a verification infrastructure developed at ETH Zurich consisting of an intermediate language targeted at verification and corresponding verifiers that prove correctness of Viper programs. In fact, instead of verifying the correctness at its own level, Gobra as well as some other frontends encode a high-level programming language into Viper. This allows them to

build up on Vipers verification capabilities. A sound encoding guarantees that specifications and annotations in the original program hold if verification of the corresponding Viper program succeeds.

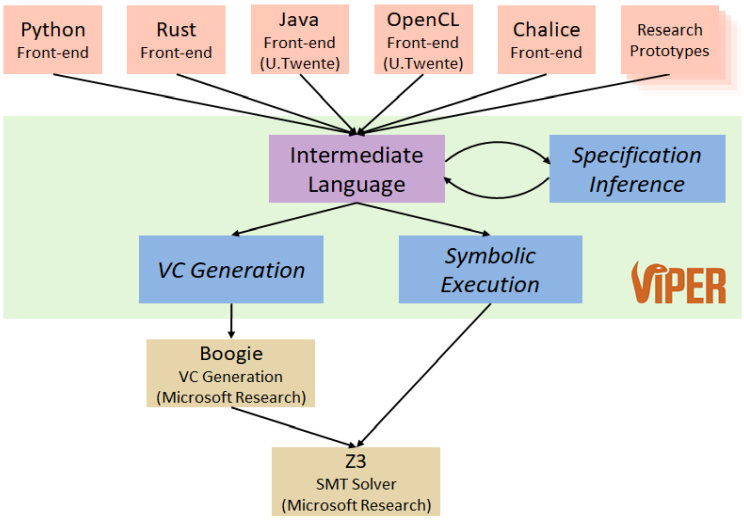


Figure 1.1: Viper's architecture[4]

Currently, Gobra only supports the verification of partial correctness for Go programs. That is, Gobra only verifies that the annotated postcondition holds when the function terminates. As a result, functions that do not terminate can be proven to satisfy arbitrary postconditions. In order to verify the total correctness for a Go program, it is required to check whether it terminates or not. However, the termination checking for programs is not straightforward. In 1936, Alan Turing proved that there does not exist an algorithm to determine whether a program terminates on every possible input, which is also named the halting problem. Due to the undecidability of this problem, any approach to prove termination in general requires guidance which is usually in the form of annotations provided by programmers. In previous work, the termination measure approach has been implemented in Viper to verify termination of Viper programs.[7] In general, it relies on termination measures which are specified by users as annotation to reason about termination. In section 2.2, we will go into details of this approach and its implementation in Viper. From our side, implementing a completely new termination checking for Gobra from scratch seems to be quite inefficient. Instead, we aim at reusing Viper's termination checker. In next section, we introduce our goals based on this core idea.

1.2 Goals

Recall that Gobra encodes the Go programs and additional annotations from programmers into Viper and we would like to reuse Viper's support for termination checking. Therefore, goals of this thesis are as follows: The first step of our work is to extend Gobra such that it could be able to accept and then soundly encode additional termination measures defined by the user at Gobra's level into Viper such that successful verification at Viper's level guarantees that the provided Gobra program terminates. Additionally, the current functionality of Gobra must not be affected by our extension and this is checked through a variety of regression tests. If the verification of termination fails, we back translate error messages provided by Viper to messages regarding the original Gobra program such that the user know which lines of code cause the termination check to fail. Furthermore, as already mentioned in the abstract, the termination measure approach cannot be directly applied to check termination of interface methods, which are dynamically bound. We will discuss the reason why this problem is not trivial and the problem of proving termination of mutual recursion in dynamic method binding in section 3.2. As a result, we devise an extension of the original termination measure approach to handle this issue. For convenience, we also design heuristics to infer termination measures automatically when possible such that they do not need to be manually defined by programmers. As the last step, we verify several Gobra programs including the code from SCION project to evaluate the effectiveness of our implementation.

1.3 Thesis Outline

This thesis mainly consists of four parts which are background, methodology, implementation and evaluation. In the first part, we introduce Gobra as well as Viper including simple syntax. We will also briefly explain how termination check is performed in Viper. In the second part, we discuss the methodology. That is, we will provide a high-level explanation on how we extend Gobra to accept and work with the additional feature of termination measures for termination checking. Furthermore, we explain why mutual recursion with dynamic method binding in Gobra leads to unsoundness of Viper's termination measure approach. We will also present our final proposal for solving this problem and introduce how we apply heuristics to infer termination measures. The third part involves the implementation of our approaches. Finally, an evaluation of our implementation is provided and we conclude our work with some ideas of possible extension for future work.

Chapter 2

Background

2.1 Gobra

In this section, we briefly introduce Gobra's syntax and its workflow.

2.1.1 Syntax Excerpt

In general, Gobra allows users to annotate the source code in order to state the functional behaviour of a program. Then, Gobra verifies the program according to these annotations. The example in Listing 1 presents a simple function annotated with pre- and postconditions.

```
1 requires n >= 0
2 ensures res == n * (n + 1) / 2
3 func factorial (n int) (res int) {
4   if n == 0 {
5      return 1
6   } else {
7      return n * factorial(n-1)
8   }
9 }
```

Listing 1: Factorial function in Gobra which recursively computes the factorial of a given input n.

Here `requires n >= 0` specifies the precondition for this function, which states that the parameter `n` should not be negative. The assertion following `ensures` corresponds to the postcondition, which guarantees that the return value is equal to `n * (n + 1) / 2`. Such preconditions and postconditions express the valid states of the function before and after the execution. The verifier checks whether the postcondition holds after the execution of the function if the precondition holds before the invocation. Otherwise, an er-

ror is reported. In this particular example, the verification fails since the postcondition can not be ensured. Note additional statements for instance unfold and fold, additional parameters or additional types just for verification purposes are referred to as Ghost Code in Gobra. As a result, adding or removing the ghost code does not change the semantics of the Go program. The reader can refer to the tutorial for a concrete description of Gobra's syntax.[3]

2.1.2 Gobra's Workflow

The whole process of verification in Gobra is divided into six phases which are parsing, type checking, desugaring, internal Gobra to Gobra transformation, encoding to program at Vipers level and verification of encoded program as shown in figure 1.2. All these steps are executed in a sequential order and the error is reported as soon as possible which means that the following steps will not be executed in case the error is detected in any of these phases. At step 1, Gobra parses the original Go program together with its annotations to Parser AST. The parsing succeeds if there is no syntactical error. Then at step 2, the program in Parser AST is passed to the type checker such that the type information is obtained for each node. Afterwards, the Parser AST is transformed to a simpler internal representation called Internal AST that does not longer contain syntactic sugar. At step 4, an internal representation to internal representation transformation is executed. At the beginning of this project, only overflow checks have been inserted into the program. As described in section 3.3, this project has added an additional transformation to this step, which infers termination measures. After the preprocessing is done, the program in Internal AST is then encoded into Viper AST. At step 6, the Viper program is verified by using one of Viper's backends, either Carbon or Silicon.[4] Carbon performs verification condition generation based on Boogie and Z3. Instead, Silicon verifies a Viper program by symbolically executing it. Gobra then obtains the verifier result from Viper, either a series of errors or success. Finally, Gobra reports the back translated messages to its user.

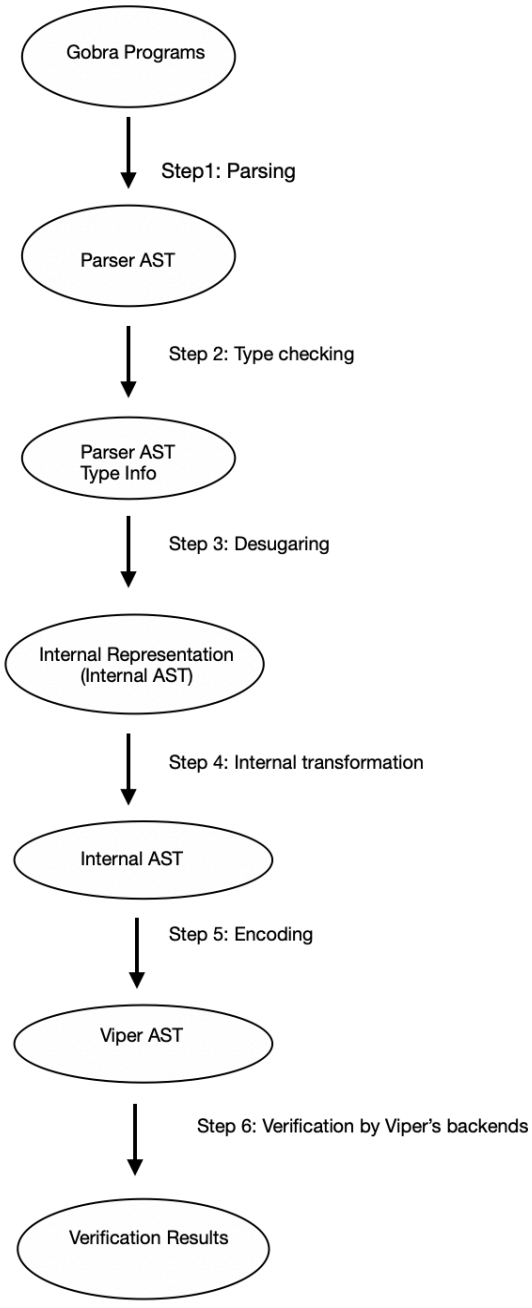


Figure 1.2: Gobra's workflow

2.2 Viper

Programs at Viper's level can not only be encoded from its frontends such as Gobra, but also directly written at its own level. Listing 2 shows the factorial method in Viper:

```
method factorial (n: Int) returns (res: Int)
   requires n >= 0
   ensures n == n * (n + 1) / 2
 {
   if (n == 0 ) {
     res := 1
   } else {
     var temp: Int
     temp := factorial(n - 1)
     res := temp * n
   }
 }
```

Listing 2: Factorial method in Viper which is encoded from the factorial function in Gobra shown in Listing 1

Viper allows the definition of similar annotations for the program as in Gobra. The keyword `requires` plus an assertion for a function or method is defined to express which precondition needs to be satisfied when it is called and the declaration of postcondition is defined as the keyword `ensures` followed by an assertion. It states what condition is guaranteed to be given after the execution by assuming the precondition holds. Listing 2 shows the corresponding method in Viper which is encoded from the `factorial` function in Listing 1. Also, the verification here fails since the postcondition can not be guaranteed.

One of the most important features of Viper is the ability to reason about the heap of a Viper program. This is managed by a permission-based approach. That is, Viper introduces field permissions and each field permission associates with an allocated heap location. Consider the following code example in Listing 3:

Here `acc(y.f, 1/2)` is called the accessibility predicate where `y.f` denotes the heap location that the permission refers to and `1/2` is called the permission amount that indicates the permission type. Quotients of two Int-typed expressions are used to denote a fractional permission or equivalently read permission. In this example, `method copy` requires the write permission to `x.f` and read permission to `y.f` since in the method body, it reads from the value from `y.f` and then write it to `x.f`. The precondition requires the caller to transfer the permissions to it which means the caller must hold these per-

```
field val: Int

method copy(x: Ref, y: Ref)
  requires acc(x.f) && acc(y.f, 1/2)
  ensures acc(x.f) && acc(y.f, 1/2)
{
  x.f := y.f
}
```

Listing 3: Viper access permission

missions before the transfer and temporarily loses it afterwards. The post-condition then corresponds to back transfer of permissions, ie. from callee to caller. The caller holds the permission again if the permission is involved in callee's postcondition. Otherwise, the caller fully loses the permission. In oder to abstract over assertions and denote permission to recursive heap structure e.g. linked lists, Viper also supports predicate. We do not present its exact syntax here. The reader can refer to Viper's tutorial for a more concrete description for predicate as well as other features in Viper.[4]

2.2.1 Termination Measure

In the first chapter, we have discussed the halting problem proved by Alan Turing. That is, determining whether a program terminates on every possible input is undecidable. As such, reasoning about termination of program in general requires input from users in the form of annotations. One of the most well-known methods is using the so-called termination measure introduced by Floyd.[12] It expects users to annotate functions and loops with termination measures, also known as variants, which are expressions with the following properties: (1) its value is bounded from below and (2) decreases in each loop iteration or recursive function call. Hence, there is an upper bound on the number of loop iterations and recursive function calls and the program is guaranteed to terminate. Again, consider the Gobra code in Listing 1: There, we can assume that `n >= 0` from the precondition. In each recursive call, the value of `n` decreases by 1 and the recursion terminates once `n` reaches 0. Hence, if we intuitively choose `n` as the termination measure, then we prove that `factorial` terminates, since the program performs, at most, n recursive function calls. The same strategy works for loops as well.

In Listing 4, we assume `s` is a sequence of integers and `len(s)` returns the length of sequence `s`. The difference between the length of sequence `s` and index `i` decreases by 1 in each loop iteration and it cannot be smaller than 0 since the loop terminates once `i` is equal to `len(s)`. Thus, `len(s) - i` as the termination measure decreases with respect to the well-founded order and

```
1 res := 0
2 for i := 0; i < len(s); i++ {
3   res += s[i]
4 }
```

Listing 4: Simple loop statement in Gobra

is bounded from below by 0. Thus, the termination of this loop is proven.

2.2.2 Support for termination check in Viper

Viper already supports termination proofs based on the user provided termination measures [7]. Actually, Viper does not only have its own built-in functionality of termination checking, but also allows its frontends to encode their termination proofs. That is, if no termination measure is specified at Viper's level, then the termination check will not be taken into account or in other words, Viper only proves the partial correctness in this case. As a result, a high flexibility is provided since the user can freely decide to ignore the termination verification or perform termination check either in the frontend before encoding programs into Viper or use Viper's termination checker for the proof. In the following section, we introduce how Viper uses termination measure approach to prove whether the program terminates or not based on call-graph analysis.

Approach

In general, the following cases are considered by Viper's termination checker which means that the termination measure can be defined at the place of functions or methods and loops as well:

1. **Directly recursive functions**: The directly recursive function refers to a function `f` that only calls itself recursively and calls to other functions which do not call `f` either directly or indirectly. In Viper, it is checked for directly recursive functions that the termination measure strictly decreases by each recursive call in it with respect to a well-founded order.
2. **Indirectly recursive functions (mutual recursion)**: The mutually recursive functions refer to functions that call each other recursively. In Viper, such functions are detected via the call graph analysis. In order to prove the termination of such functions, Viper checks whether the termination measure of an indirectly recursive function decreases at each indirectly recursive call. Consider the following case: If function `foo` calls function `bar` and `bar` calls `foo`, viper checks that the termination measure of `bar` is less than `foo`'s in the call to bar and also that the termination measure of `foo` is less than `bar`'s, in the call to `foo` in

function `bar`. This implies that the termination measure in the second call to `foo` is less than that in the first call.

3. **Loops**: Also, Viper is able to verify the termination of loop statements e.g the while loops. That is, Viper verifies if the defined termination measure decreases at the end of an execution of the loop body in comparison to its value at the beginning. Note that Viper does not check whether each single statement in the loop terminates for simplification.

4. **Directly or indirectly recursive methods**: Similar to the termination check for functions, Viper can also verify the termination of methods. However, different from the verification for functions, Viper not only checks the termination measure decreases by each recursive method call, but also the termination of other non-recursive method calls and loops contained in this method. In other words, Viper cannot verify the termination of a method if it contains a loop or a non-recursive method call and the termination of the loop or method call can not be proved, even if the termination measure is correctly defined for the method. Moreover, Viper does not check the termination of function calls inside the method. The reader who is interested in this point can check the reason for this design decision in the Viper tutorial.[4]

Syntax of Termination Measures

Currently, Viper is not able to infer termination measures automatically which means that the verification of termination relies on the measures defined by users manually. These measures should be specified in a certain format. In this section, we introduce Viper's syntax of termination measures

In general, there are five types of termination measures supported in Viper. As discussed before, the termination measure can be defined at the position of a precondition and for a loop, at the position of an invariant. The general form of a termination measure is ***decreases*** <***tuple***> where ***decreases*** is a keyword and <***tuple***> here refers to a tuple of comma-separated Viper expressions, in particular a single expression as the measure is also allowed. The lexicographical order over the elements in the tuple will be used as the well-founded order if a tuple of expressions is chosen as the termination measure. Note ***decreases*** <***tuple***> is called the decreases clause while <***tuple***> is called the termination measure. In the remaining part of this thesis, we will not explicitly distinguish between decreases clause and termination measure.

In Viper, the well-founded order for the termination measure of most built-in types is already offered in Viper's standard library and can

be imported. For instance, the well-founded order for integers can be imported by the statement ***import*** <***decreases/int.vpr***>. As an alternative, the user is also able to define the well-founded order over custom types.

In addition, in order to handle some special cases, Viper supports the definition of some specific termination measures as follows:

The user can define an **empty measure** by using the keyword ***decreases*** without any measure. The empty measure instructs Viper to prove termination of non-recursive methods, for which a constant termination measure would also suffice. Another situation in which the empty measure could be applied is verifying a method which contains a call to one non-recursive method. In order to verify this method, we should also verify that this non-recursive method terminates even though we actually do not need to prove that. In this case, we can define the empty measure for this non-recursive method. In other words, by defining an empty measure, the user can force Viper to generate the termination check with a constant as the termination measure which is inferred by Viper in a suitable way and if the method terminates intuitively, Viper verifies the termination successfully.

Besides, the user can specify the **wildcard measure** with the syntax ***decreases*** _ to express that a function is to be considered terminating, although no termination checks are generated. In other words, a wildcard measure is an unchecked assumption.

Furthermore, the **star measure** is provided in Viper with the syntax ***decreases*** * to express that a function is not checked for termination. Hence, it may not terminate. This special measure is equivalent to providing no decreases clause at all. The common use of star measure is to document that the function is not yet checked for termination.

Finally, in order to combine different measures, Viper provides users the possibility to define **conditional termination clauses** for tuple and wildcard.

```
1 decreases <tuple> if <condition>
2 decreases _ if <condition>
3 decreases *
```

Listing 5: Conditional termination measure in Viper

Here <***condition***> is a Viper expression which indicates when the corresponding termination measure is chosen to check the termination. Note that the star ***decreases*** * is optional and can be defined for documentation purpose.

The implementation of termination proof is done by a plugin for termination in Viper. It is beneficial to avoiding changes for existing features and syntax of Viper such that various frontends of Viper will not be influenced by adding the support for termination check to Viper. The general workflow of the plugin is as follows: The decreases clauses introduced in the previous section are encoded as a decreases function call and placed together with postconditions in the function specification. After parsing is done, the resulting abstract syntax tree for parser will be type checked and it resolves to Viper AST. Then the decreases function calls in the postcondition are translated to decreases clauses nodes which are introduced by the plugin and function calls are removed from the postcondition. Note that the frontend which directly defines the Viper AST can skip these steps and in replacement, it is able to generate the Viper AST with such nodes. This is quite important for our implementation and will be discussed in the following chapter. Then Viper generates the termination proof and these decreases clauses nodes are removed from the AST. Finally, the original Viper program with the termination proof is obtained in pure Viper AST and will be passed to Viper backend verifiers.

For more details regarding the support for termination check in Viper, the reader can refer to the thesis in the previous work or the Viper tutorial. [7]

Limitations of Viper's approach

In the previous section, we have introduced the approach deployed in Viper which not only provides users the possibility to directly enable termination checks in Viper, but also facilitates the termination verification in Vipers frontend.

However, Viper's approach for termination check is still limited to a certain extent. Firstly, Viper does not check whether loop body terminate. As such, loops might not terminate even when the number of iterations that they can perform is bounded

Another limitation of Vipers approach becomes noticeable in the case of mutual recursion. In fact, for verifying the indirectly recursive functions, Viper performs the call-graph analysis. For each invocation in the resulting call-graph, it checks whether the termination measure of the callee is strictly smaller than the caller instead of performing the check transitively. An example is shown in Listing 6:

Here we can conclude that both functions terminate. However, Viper cannot verify the termination since by the call to `mutual_recursion1`

```
function mutual_recursion1(n: Int): Int
  requires n >= 0
  decreases n
{ n == 0 ? 0 : mutual_recursion2(n-1) }

function mutual_recursion2(n: Int): Int
  requires n >= 0
  decreases n
{ n == 0 ? 0 : mutual_recursion1(n) }
```

Listing 6: Unsound case of Viper's approach

in the function body of `mutual_recursion2`, the termination measure `n` stays unchanged and it violates the requirement that termination measures must decrease in every call in the chain even if the termination measure indeed decreases while `mutual_recursion2` is again called in the body of `mutual_recursion1`. In such a case, Viper relies on the user to observe such problems rather than handling it automatically. In this example, the user can extend the termination measure of `mutual_recursion1` to `(n, 1)` and `(m, 0)` for `mutual_recursion2` respectively. Then the verification in Viper succeeds

2.3 Interfaces & Behavioral subtyping

Go has support for interfaces and dynamic method binding, features for which Viper has no counterpart. In this section, we introduce the syntax of interface in Gobra as well as the behavioral subtyping. Similar to some other programming languages like Java, interfaces in Go consist of a set of method signatures without defining the exact implementation of the method body as the following code example shows:

```
// declaration of interface
type FactEven interface {

// signature of method
  factEven(int) int
}
```

Listing 7: Example of interface in Go

Here, `FactEven` declares an interface type and it contains a single method called `factEven` which takes an int-type argument as the input and returns an integer.

Interfaces in Go can also be implemented. In some programming lan-

guages, the interfaces are implemented explicitly by using the keyword like implements such that the subtype relation between a type and interface could be determined by such explicit declarations at compile time which is also called the nominal subtyping. Different from the nominal subtyping, Go uses the structural subtyping. That is, whether a type implements a particular interface is determined by checking if it implements all methods which are declared in that interface. Consider the following example:

```
1  // implementation for interface FactEven
2  type FactEvenImpl struct {}
3
4  func (x FactEvenImpl) factEven (n int) {
5    if n == 0 {
6       return 0
7    } else {
8       // some implementation
9    }
10 }
```

Listing 8: Implementation of FactEven interface

Here, `FactEvenImpl` is one implementation of `FactEven` interface since all methods (in this specific example, only `factEven`) are defined in this struct type.

In Gobra, interface methods can also be specified with pre- and postconditions:

This code example in Listing 9 corresponds to the previous one in Go with additional specifications at Gobra's level. Note that for the interface and its implementations, Gobra checks whether for all methods the precondition of the method in interface implies the precondition of its implementation method and the postcondition of its implementation method implies the postcondition of the interface method. When this holds, the implementation type is called a behavioral subtype of the interface method.[9] This is done in Gobra by generating the so-called implementation proof.

In general, the implementation proof is defined by the keyword `implements`. Inside the clause, the proof method for each method declared in the interface must be defined. Only `assert` statements as well as `fold` and `unfold` can be included in the proof method body in order to check whether the required conditions stated above are satisfied. Furthermore, the proof method must include a single call to the corresponding method in the implementation in a restricted syntax. Note that

```
type FactEven interface {
  requires n>= 0
  factEven(n int) int
}

type FactEvenImpl struct { }

requires n >= 0
func (x FactEvenImpl) factEven (n int) {
  if n == 0 {
     return 0
  } else {
     // some implementation
  }
}
```

Listing 9: Interface and its implementation in Gobra

```
// implementation proof
(* FactEvenImpl) implements FactEven {
  (x *FactEvenImpl) factEven (n int) {
    assert n >= 0 => n >= 0
    return x.factEven(n)
  }
}
```

Listing 10: Corresponding implementation proof of the interface and its implementation in Listing 9

users can either manually write the implementation proof or Gobra will infer one in case part of the implementation proof is missing or completely not provided by the user. For more details, readers can refer to the interface part in Gobras tutorial.[3]

2.4 Inference of termination measures

As already mentioned, reasoning about termination of program relies on annotations e.g. termination measures which are specified by programmers. In fact, they can be automatically inferred in some simple cases.

Dafny[8], for example, is a programming language with built-in features to specify and verify functional properties of program. In Dafny, the termination measure approach is used for checking termination of programs as well. However, Dafny differs from viper in that it treats

non-termination of methods as errors. Therefore, Dafny tries to infer the termination measure in case it is not supplied by the programmer. Typically, the measure is inferred as the list of arguments of the function or method in the given order as the following example shows:

```
1 function Fib(n: nat) : nat {
2 if n < 2 then n else Fib(n-2) + Fib(n-1) }
3 }
```

Listing 11: Fibonacci function in Dafny

Here we ignore the syntax of Dafny for simplification. Since the termination measure is not provided for the function `Fib` which computes the Fibonacci sequence for a given natural number, Dafny infers `n` as the termination measure during the verification. In this particular example, Dafny's heuristic infers the correct termination measure and the termination of `Fib` is verified. Currently, such heuristic is not provided in Viper. As a result, along with the support for termination checking, we are also motivated to introduce similar heuristic for inferring termination measures in Gobra such that Gobra's user can be freed from coming up with suitable termination measures when possible.

Chapter 3

Methodology

In general, our approach to verify termination in Gobra builds on top of Viper's support for verifying termination. More concretely, we extend Gobra with support for termination measures. That is, we extend each step of Gobra's workflow such that the termination measures can be proceeded through the pipeline. Finally, we propose a sound encoding such that these termination measures are translated to the analogous functionality in Viper. In the next section, we present the overall idea about how each phase in Gobra's workflow is extended for handling termination measures. Lastly, we show how we back-translate the errors produced by Viper when it cannot verify termination. Readers can refer to chapter 4 for the exact implementation of the approach. Not that for the purposes of this thesis, we will refer to Gobra methods and functions interchangeably.

3.1 Extension of Gobra's workflow

In section 2.2.2, we presented currently supported formats of a termination measure in Viper. We decided to allow users to define termination measures in the specification of functions, in particular after postconditions and before functions declaration for functions. In addition, they can also be defined after invariants and before the actual loop statement for loops. Note the syntax of termination measures in Gobra is exactly the same as those in Viper except for the ***infer termination measure.***

(1) Tuple Termination Measure: ***decreases*** <***tuple***> where tuple denotes a tuple of comma- separated expressions.

(2) Empty Termination Measure: ***decreases*** which is a particular case

of tuple termination measure (when <***tuple***> is empty)

(3) Wildcard Termination Measure: ***decreases*** _

(4) Star Termination Measure: ***decreases*** *

(5) Conditional Termination Measure:
decreases <***tuple***> ***if*** <***condition***>
decreases _ ***if*** <***condition***>
decreases *

Here condition refers to a boolean expression

6. Infer Termination Measure: ***decreases infer***

Note readers can refer to section 2.2.2 to find out the meaning of each termination measure and in which case it is suitable to be used. The special measure *infer* instructs Gobra to use its heuristics to come up with a suitable termination measure. We will discuss this feature in detail in section 3.3. This approach is chosen in the context of our thesis due to the following reasons: Firstly, the encoding of termination measures into Viper can be greatly simplified since it becomes relatively straightforward by coinciding the syntax of measures in Gobra with those in Viper. Secondly, since the verification is actually done using Viper, if we want to allow users to define new formats of termination measures which are not included in Viper, then Viper's termination plugin and its approach for termination check should be extended accordingly. As a result, other frontends of Viper could potentially be influenced by these changes. In order to translate these termination measures with the program together to Viper program, we extend Gobra's workflow as follows:

- **Parser and Parser AST**: In order to parse such termination measures, we extend the Parser AST by defining new Parser AST nodes and each node abstracts one type of termination measure introduced above. During the parsing, termination measures are parsed to these nodes which are then contained in the subtree of function, method or loop node, depending on where the termination measures are defined in the original Gobra program

- **Type checking**: After parsing the termination measures, we check that every expression occurring in them is well-typed and pure, that is, it does not produce any side effects.

- **Desugaring**: For the next step in Gobra's workflow i.e. desugaring, we also extend the Internal AST of Gobra such that each

kind of termination measure can be represented by a corresponding node in the Internal AST. Recall that the syntactical sugar in Parser AST is removed at this step. Since the syntax of termination measures cannot be further simplified, no significant changes are made while translating termination measures from parser AST level to internal AST level.

- **Internal transformation**: Afterwards, the Gobra to Gobra transformation is performed to the obtained program represented in the internal AST from the last step. Our heuristics for inferring termination measures are implemented in this phase and we will discuss this topic in section 3.3

- **Encoding**: As the last step before verification, termination measure nodes in Internal AST are translated to corresponding decreases clauses nodes in Viper AST in such a way that the resulting decreases clauses nodes should represent termination measures in the original Gobra program without changing the functionality of these measures. As introduced in section 2.2.2, the termination plugin in Viper performs some additional transformations to Viper AST before verification by which the decreases clauses nodes are removed and termination proofs are generated. Since the Viper program generated by Gobra is directly passed to Viper verifier, we manually invoke the termination plugin to complete these required changes.

- **Error back translator**: Finally, error messages regarding the termination check are back translated in the following way: We reuse text messages about the type of error and the reason why this error occurs. Then we replace the positional information by the back tracked position in the source Gobra program. For instance, if Viper reports that a certain method in the Viper program does not terminate since its termination measure may not decreases or might not be bounded, in Gobra we back track the line of the corresponding Gobra method as well as its termination measure from which the non-terminating Viper method is encoded. Finally, we report that the method defined at that line may not terminate since its termination measure might not decrease or might not be bounded.

3.2 Specifying and Verifying Termination of interface methods

As mentioned in section 2.3, Go as well as Gobra have support for interfaces and dynamic method binding which are not involved in Viper. Therefore, the termination check for interface methods is not considered by the approach introduced in the last section. In this section, we present our approach for solving this issue. Readers can refer to section 4.2 for details on the implementation of this approach in Gobra. In fact, solving the problem of handling termination check for interface methods, especially checking termination of mutual recursive methods in presence of dynamic method binding is not trivial. We explain it based on an example. Consider the code snippet in Listing 12:

In that example, `factEven` and `factOdd` methods are declared in `FactEven` and `FactOdd` interface respectively. `FactEvenImpl` implements `FactEven` interface while `FactOddImpl` is an implementation for `FactOdd`. We can figure out that `factEven` method implemented in `FactEvenImpl` computes the factorial of even numbers and it recursively calls `factOdd` method at interface. At the same time, `factOdd` method in `FactOddImpl` computes the factorial of odd numbers and it calls the interface method `factEven`. In this case, we face some difficulties while verifying the termination. Firstly, Viper cannot prove the termination for call `y.factOdd(n-1)` if no termination is defined for interface method `factOdd`. Therefore, we should decide whether the termination measure is allowed to be defined for the interface method. If so, we need to concern about the problem that whether the termination measure can be specified for the implementation method as well. Then we should determine what kind of relation between termination measures of interface method and its implementation methods need be required.

Currently, Gobra allows users to define pre-and postconditions for the abstract method in the interface. Similarly, our approach is firstly to allow users to define the termination measure as well for both abstract methods in the interface and corresponding implementation methods in the subtype. Recall that the rules for behavioral subtyping in Gobra define which types are allowed to implement an interface. That is, to let a type to be a subtype or in other words an implementation of an interface, Gobra checks whether for all methods the precondition of the method in interface implies the precondition of its implementation method and the postcondition of its implementation method implies the postcondition of the interface method by generating the implementation proof. Inspired by that, we extend our notion of behavioral subtyping with the restriction that termination measures of interface methods are an upper bound to those of implementation methods.

```
type FactEven interface {
  requires n >= 0
  // termination measure?
  func factEven(n int) int
}

type FactOdd interface {
  requires n >= 0
  // termination measure?
  func factOdd(n int) int
}

type FactEvenImpl struct {}
  requires n >= 0
  // termination measure?
  func (x FactEvenImpl)factEven(n int) {
    if n == 0 {
      return 0
    } else {
      var y FactOdd = FactOddImpl{}
      return n * y.factOdd(n-1)
    }
  }

type FactOddImpl struct {}
  requires n >= 0
  // termination measure?
  func (y FactOddImpl)factOdd(n int) {
    if n == 1 {
      return 1
    } else {
      var y FactEven = FactEvenImpl{}
      return n * y.factEven(n-1)
    }
  }
```

Listing 12: Mutual recursion in dynamic method binding

Note that in principle, we would only need to enforce that the termination measure of the implementation is at most that of the interface when the precondition of the interface method holds. In the other cases, the implementation could actually not terminate without breaking behavioral subtyping. When verifying the termination of mutual recursive method, if the callee is an abstract method, then it is checked

whether the termination measure defined by the interface method is strictly smaller than the measure of the caller. According to our notion of behavioral subtyping, if the termination measure of the abstract method is indeed strictly smaller than the measure of caller, then we can also ensure this holds for all implementations in the subtype. As such, the termination of implementation methods `factEven` and `factOdd` can be proven. However, at this point, the approach is still not sound. Consider the simple code example in Listing 13: Method `f` calls the abstract method `g` in the interface and the implementation of this abstract method calls `f` in turn. As a result, this indeed leads to the existence of mutual recursion. However, the call-graph analysis results in a call graph where there is an arrow from method `f` to abstract method in interface and an arrow from its implementation to `f` as shown in figure 3.1.

```
1  type G interface {
2    requires ... // some preconditions
3    ensures ... // some postconditions
4    decreases ... // some termination measure
5    func g(m int) int
6  }
7
8  type GImpl struct {}
9    requires ... // some preconditions
10   ensures ... // some postconditions
11   decreases ... // some termination measure
12   func (h GImpl) g(m int) int {
13      // some implementation
14      return f(m - 1)
15 }
16
17 requires ... // some preconditions
18 ensures ... // some postconditions
19 decreases ... // some termination measuree
20 func f(m int) int {
21   // some implementation
22   var y G = GImpl{}
23   return m * y.g(m - 1)
24 }
```

Listing 13: Simple example of mutual recursion with dynamic method binding

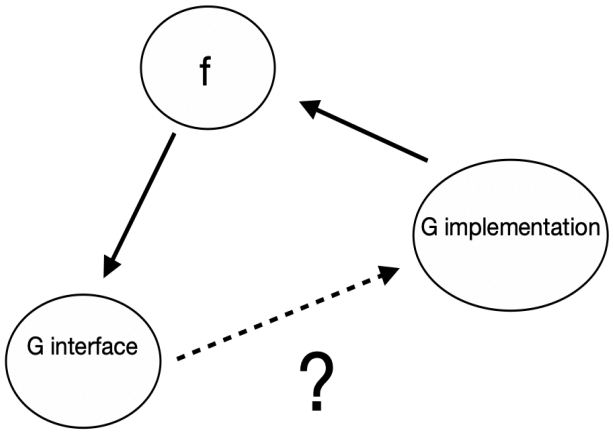


Figure 3.1: Call graph of code example in listing 14

Obviously due to the missing arrow from the abstract method g to its implementation method, Viper's call-graph analysis does not detect mutual recursion in this method. Thus, the proof obligation for termination of mutual recursion is not generated at all since there is no mutual recursion in the call graph. In order to solve this problem, our approach is to add the artificial call from the abstract method to its implementation. That is, we replace the empty body of the abstract method by a call to the implementation method which is chosen by a case split over all its implementations where it is checked which implementation is used as the dynamic type. But here the problem is that according to Viper's analysis introduced before, it proves for this additionally added call from abstract method to implementation, the termination measure must decrease as well which is not the case since here the call we add is only a resolution and therefore the termination measure actually stays the same. Therefore, what we need to do here is adding the artificial call to close the cycle while skipping the termination measure check for this call without affecting the analysis at other places in the cycle. In our approach, this issue is solved by using the assume false statement to skip the proof. That is, we add an additional statement `assume false` to the body of the abstract method and place this statement before the artificial call to implementation methods. The statement `assume false` makes all proof obligations trivially true, including proving that the function terminates. Thus, in effect, this ignores the proof for this artificial call that the termination measure decreases.

3.3 Heuristics for termination measures

In section 2.4, we briefly introduced that some programming languages such as Dafny are able to infer termination measures. As another goal of thesis, we implement heuristics in order to infer termination measures in case that they are not provided by users such that they can be freed from the task of coming up with termination measures when possible. In this section, we provide a high-level explanation on Gobra's support for termination heuristics. In section 4.3, the concrete implementation is presented.

Our approach is not to infer termination measures for all functions and loops that are not annotated, but instead to introduce a new termination measure, 'infer' as an annotation, which signals the functions or loops for which Gobra should try to infer termination measures. The reason is that if we infer termination measures everywhere, the computation effort increases significantly since Gobra would potentially try to prove termination of functions which are not meant to terminate. That means, the termination measure of functions or loops is inferred in case that the user specifies infer as the measure.

```
1  decreases infer
2  func infer_measure(ghost s seq[int]) int {
3    ghost res := 0
4    invariant i >= 0
5    invariant len(s) >= i
6    decreases infer
7    for i := 0; i < len(s); i++ {
8        res += s[i]
9    }
10 }
```

Listing 14: Termination measures for function as well as loop statement are not provided by user and will be inferred by Gobra

Furthermore, termination measure generation is implemented as a Gobra-to-Gobra transformation and it serves as an extension of the original Gobra's internal transformation phase since the internal transformation in Gobra is designed in such a way that new transformations can be modularly added. As a result, we define a new transformation which works on inferring the termination measures using multiple different heuristics. That is, the heuristics are defined as functions in the new internal transformation of Gobra. Each one takes the information about a function represented in internal AST as the input and infers a termination measure for it based on its specific strategy.

In general, the transformer is applied to desugared Gobra programs, that

is, the AST representation of Gobra programs. Then, for each function in this program, it checks at every possible place including at the place of postconditions and invariants for loop statements whether the termination measure there needs to be inferred or not. If it is the case, it chooses one of the heuristics to infer the termination measure and construct a new function or loop statement without changing anything else except for adding the inferred measure. Otherwise, nothing is performed and the function or loop statement remains the same. Also if there is already a user-defined termination measure that is not the infer measure, the transformer does nothing for it. Additionally, in the first case, these annotations should be removed after the inference of termination measures is completed. After iterating over all functions, we obtain a new collection of functions which are still represented in the internal AST. Therefore, the output of this phase can be then passed to the encoder and gets verified by Vipers backend verifier. Since during the internal transformation, at each place where the annotation for enabling the heuristic is defined, only one certain heuristic is chosen to be applied to infer the termination measure and we actually would like to try out multiple heuristics in case that the inference fails, we slightly modify Gobra's workflow as follows:

- **Step 1**: The first three steps, i.e. parsing, type checking and desugaring are executed only once as before. After the desugaring is completed, we obtain a program including infer termination measures in the internal representation which is then stored. It will be reused for multiple times in the following steps. In the first iteration, we pass this program to the extended internal transformation introduced above and the result is then further passed to decoder and get verified.

- **Step 2**: Once the verification is done, we observe the verification result: If Viper successfully verifies, we report the success to Gobra's user as well. Otherwise, if all errors reported are related to the termination check for which the inferred termination measure is used, we inspect all these errors to determine for which functions the heuristic used in the internal transformation works and keep track of them. Then we repeat the process of internal transformation for which we pass the original program in internal AST that is stored in step 1 as input. During the internal transformation, before the heuristics are applied to functions as described above, we additionally check whether there is a mapping for the function to a heuristic used in previous iterations which is established at the beginning of step 2. If so, then in the current iteration, we use that heuristic to infer the termination measure instead of choosing a new one. Otherwise, we apply a new heuristic which has not been used yet to infer the measure. Afterwards, we pass the resulting program to encoder and further to the verifier. Then we repeat step 2.

If there is at least one error which is not caused by a wrong inference of termination measure, we stop the repetition and report all errors. The reason is that since all follow-up tries of other heuristics could potentially fail because of this error even though the right measure is actually inferred.

- **Step 3**: Once all heuristics are tried out, we stop the repetition of step 2. In case that for a certain function all heuristics fail, we notify the user that no provided heuristic works there and ask for the user to provide one

In the context of our thesis, we provide following heuristics:

- **Inferring termination measures for functions**: For simple functions, its relatively easy to infer the termination measure. In many cases, the functions are recursive with a low number of arguments. For example, if the function only has a single parameter, we can infer the termination measure as this single parameter. Once the list of its arguments grows up, the problem becomes more complex. If we want to infer the right measure, then it is required to analyze function's body which is quite complicated and inefficient in case that the complexity of the function is high and size of the function body is extremely large. Therefore, the heuristic provided by us for functions is inferring the measure as a tuple of all arguments in the given order. Note that in fact the empty measure is semantically equivalent to the tuple of arguments as the termination measure. This heuristic certainly does not work in all cases. However, we observe that termination measures of many functions in the SCION codebase for which we verify the termination can be inferred by this heuristic. We will further discuss this topic in chapter 5.

- **Inferring termination measures for loops**: We provide a heuristic which works on analyzing the loop condition to infer a suitable termination measure. The reason is that the loop condition can usually reflect which variant decreases by each loop recursion. For arithmetic operations of the form A op B for instance $A \leq B$, we can figure out that for each loop iteration, it is possible that the value of A increases and once the value of A reaches the value of B, the loop terminates. So in this case, the difference between A and B should decrease by each iteration and thus we infer the termination measure as B-A. Similarly, we infer the termination measure in the same way if the loop condition is of the form $A \geq B$ just in the opposite direction. Also for the condition like $A \neq B$, we can make a guess that the difference between B and A decreases by each iteration and the loop terminates once A is equal to B. Thus, either B-A or A-B is inferred as the measure. Then we

need to figure out which one of A or B has the greater value before the loop statement if executed. This is done by looking into the invariants associated with the loop. For instance, if the loop condition is A $\neq$ B and A is greater than B in the initial state, then typically it is stated in the invariant that A $\geq$ B. As a result, we can infer the termination measure as A-B.

Note that in our thesis, only simple heuristics are implemented. However, with the help of our architecture for inferring termination measures, more heuristics can be easily added in the future by defining new functions as heuristics in the internal transformation without doing other changes.

Chapter 4

Implementation

In this chapter, we present the implementations of the extension for Gobra's workflow. Furthermore, we also discuss the implementation for approaches handling the problem of mutual recursion with dynamic method binding as well as inferring termination measures which are explained in section 3.2 and 3.3 respectively.

4.1 Implementation of Gobra's extension

4.1.1 Parser & Parser AST

In chapter 3, we mentioned that the first step to extend Gobra's workflow is extending Parser AST and therefore, also the parser such that termination measures in the program can be accepted by Gobra. We represent all supported termination measures by new introduced nodes in Parser AST as shown in Listing 15

`PTerminationMeasure` is defined as a subclass of Parser AST node while all termination measures introduced before are separated to conditional or unconditional measures. Both `PUnconditionalTerminationMeasure` and `PConditionalTerminationMeasures` are defined as the case class of PTerminationMeasure. Note that in order to wrap all conditional termination measures into one single measure, `PConditionalTerminationMeasure` accepts a vector of PConditionalTerminationMeasureClause while each clause denotes a single conditional measure. Empty termination measure corresponds to the `PTupleTerminationMeasure` with an empty vector as its argument. Currently, in Parser AST, preconditions and postconditions are placed in the function specification. As a result, we add a new field to `PFunctionSpec`.

```
sealed trait PTerminationMeasure
  extends PNode

sealed trait PUnconditionalTerminationMeasure
  extends PTerminationMeasure

sealed trait PConditionalTerminationMeasureClause
  extends PNode

case class PConditionalTerminationMeasureIfClause
  (measure: PUnconditionalTerminationMeasure, cond: PExpression)
    extends PConditionalTerminationMeasureClause

case class PStarMeasure()
  extends PUnconditionalTerminationMeasure
  with PConditionalTerminationMeasureClause

case class PWildcardMeasure()
  extends PUnconditionalTerminationMeasure

case class PTupleTerminationMeasure(tuple: Vector[PExpression])
  extends PUnconditionalTerminationMeasure

case class PConditionalTerminationMeasures
  (clauses: Vector[PConditionalTerminationMeasureClause])
    extends PTerminationMeasure

case class PInferTerminationMeasure()
  extends PUnconditionalTerminationMeasure
```

Listing 15: Parser AST nodes which abstract termination measures

Since only at most one termination measure could be defined for function, the new field in `PFunctionSpec` is defined as an option of `PTerminationMeasure`. Similar to the function specification, the loop has an associated loop specification which currently only contains one field that stores invariants. Thus, we add the field of termination measure for `PLoopSpec` as well

After the Parser AST is extended for termination measures, we modify the parser such that when parsing functions and loops, the termination measure, if there is one provided, is parsed to `PTerminationMeasure` which is then placed together with other specifications for functions in `PFunctionSpec` or invariants for loops in `PLoopSpec` respectively. In type checker, we check

```
case class PFunctionSpec(
                      pres: Vector[PExpression],
                      preserves: Vector[PExpression],
                      posts: Vector[PExpression],
                      terminationMeasure: Option[PTerminationMeasure],
                      isPure: Boolean = false,
                      ) extends PSpecification
```

Listing 16: Extended PFunctionSpec node in Parser AST

```
case class PLoopSpec(
                    invariants: Vector[PExpression],
                    terminationMeasure: Option[PTerminationMeasure],
                    ) extends PSpecification
```

Listing 17: Extended PLoopSpec node in Parser AST

whether all `PExpression` which occur in termination measure nodes are pure and well-defined.

4.1.2 Desugaring

As shown in Listing 18, Similar to Parser AST, new Internal AST nodes need to be defined such that termination measures can be abstracted in the internal representation as well. The desugaring for termination measures is relatively straightforward since there is no significant difference between its syntax at Parser AST level and in internal representation. That is, we only need to desugar the argument of termination measure nodes and the overall structure can be kept. Note that `TupleTerminationMeasure` takes a vector of internal AST nodes instead of expressions. The reason is that predicate call could be chosen as the termination measure and the predicate access in the internal representation is a direct subclass of `Node` in internal AST instead of expressions.

In addition, functions in the parser AST are desugared to functions or pure functions according to the pure field defined in the `PFunctionSpec`. Furthermore, specifications in the `PFunctionSpec` are unwrapped to mutiple fields of the function node in the internal representation. As a result, we extend `FunctionMember`, `Function` and `PureFunction` nodes by adding a new field to place termination measures. Note that this holds for methods as well.

```
sealed trait TerminationMeasure
  extends Assertion

sealed trait UnconditionalTerminationMeasure
  extends TerminationMeasure

sealed trait ConditionalTerminationMeasureClause
  extends Assertion

case class ConditionalTerminationMeasureIfClause
  (measure: UnconditionalTerminationMeasure, cond: Expr)
  (val info: Source.Parser.Info)
    extends ConditionalTerminationMeasureClause

case class StarMeasure()(val info: Source.Parser.Info)
  extends UnconditionalTerminationMeasure
  with ConditionalTerminationMeasureClause

case class WildcardMeasure()(val info: Source.Parser.Info)
  extends UnconditionalTerminationMeasure

case class TupleTerminationMeasure
  (tuple: Vector[Node])(val info: Source.Parser.Info)
    extends UnconditionalTerminationMeasure

case class ConditionalTerminationMeasures
  (clauses: Vector[ConditionalTerminationMeasureClause])
  (val info: Source.Parser.Info)
  extends TerminationMeasure

case class InferTerminationMeasure()(val info: Source.Parser.Info)
  extends UnconditionalTerminationMeasure
```

Listing 18: Representation of termination measures in Internal AST

4.1.3 Encoding

Our final step before verification is encoding termination measures with the program together to Viper AST. As discussed in section 2.2.2, the decreases clause nodes in Viper are defined in the termination plugin as an extension for the Viper AST. In order to provide the convenience, Viper allows frontends to define the decreases clause node while constructing the program in Viper AST. As a result, we can import Vipers termination plugin and then

construct decreases clause nodes based on termination measure nodes in Internal AST. Afterwards, we append all generated nodes to invariants for loop, or preconditions for functions as well as methods depending on the position of these termination measure nodes in Internal AST. In the termination plugin, three kinds of decreases clause nodes are supported as shown in Listing 19.

```
case class DecreasesTuple
  (tupleExpressions: Seq[Exp] = Nil, override val condition: Option[Exp] = None)

case class DecreasesWildcard
  (override val condition: Option[Exp] = None)

case class DecreasesStar()
```

Listing 19: Decreases Clause Nodes in Viper's termination plugin

`DecreasesTuple` is used to abstract the tuple termination measure and thus it takes a sequence of Viper expressions in the first filed of its argument. Also, since the tuple termination measure can be used in the conditional termination measure, it also has an option of the Viper expression which indicates the condition. `DecreasesWildcard` denotes the wildcard termination measure and it also takes an option of Viper expression as the condition in case that it is used for conditional termination measure. And the star measure is abstracted by DecreasesStar node. Note that the empty termination measure is denoted by the DecreasesTuple node with an empty sequence of expressions and one can represent the conditional measure as a vector of `DecreasesTuple`, `DecreasesWildcard` and `DecreasesStar` nodes. Since the definition of termination measures in our internal representation is quite similar to that in the termination plugin, the encoding is straightforward:

- **Encoding wildcard measure**: We directly construct a `DecreasesWildcard` node by setting the argument to `None` since it does not appear in the conditional measure.

- **Encoding star measure**: We directly construct a `DecreasesStar` node.

- **Encoding tuple termination measure**: We construct a `DecreasesTuple` node. Note that the first argument of `TupleTerminationMeasure` is a vector of nodes in internal AST which contains expressions or predicate access. If the element is a predicate access, we encode it to `PredicateInstance` node that is defined in the predicate instance plugin. If it is an expression, then we apply Gobra's encoder for expressions. The reason why

predicate access in the internal representation which abstracts the predicate instances should be encoded to a PredicateInstance node is that in Viper, when a predicate instance is used as the termination measure, then the corresponding well-founded relation is `nested`. And the nested relation between two predicate instances can be checked by the domain function nested. However, it can only take first-class Viper type as its arguments which is not the case for predicate instances. Therefore, the `PredicateInstance` node is defined in order to represent the predicate instances. The second argument which denotes the condition is set to None.

- **Encoding conditional termination measure**: We encode the vector of `PConditionalTerminationIfClause` and `StarMeasure` to a vector of `DecreasesTuple`, `DecreasesWildcard` and `DecreasesStar` nodes in the following way:

  (1) If the first argument of `PConditionalTerminationIfClause` is the tuple termination measure, we do the same for encoding tuple termination measure and encode the second argument which is the expression in internal AST to Viper AST expression. It is then placed in the second argument in the constructed `DecreasesTuple` node.
  (2) If the first argument of `PConditionalTerminationIfClause` is the wildcard measure, we construct a `DecreasesWildcard` node and we encode the expression in the condition field to a Viper AST expression.
  (3) We directly construct a `DecreasesStar` node.

Note that in Viper, after the program represented in Viper AST with decreases clause nodes is generated and before the verification takes place, the termination plugin of Viper performs an additional step of transformation. That is, it takes the program in Viper AST as the input, removes decreases clauses from the AST and then appends them as additional information to the function, method or loop node. Then it generates the termination proof based on these information. Besides, for `PredicateInstance` nodes in the Viper AST, the predicate instance plugin in Viper also performs a transformation before the verification. During the transformation, the predicate instance function for predicates is generated and then all `PredicateInstance` nodes in AST are transformed to function calls to corresponding predicate instance functions. And in Viper, this transformation is performed before executing the transformation in the termination plugin. Thus, we invoke this transformation in the predicate instance plugin for the obtained program from encoding and then further perform the transformation in termination plugin. Finally, we get a program in Viper AST that satisfies all requirements of preprocessing for Viper's verification.

4.1.4 Error back translator

As mentioned, all error messages reported from Viper are related to the encoded Viper program which are actually hard to understand for Gobra's user. Therefore, we extend Gobra' back translator for errors regarding the termination check. Indeed, we reuse text messages from Viper that talk about the type of error and reason why this error occurs. In Listing 20, we show an example of error and error reason in Viper:

```
case class MethodTerminationError(override val offendingNode: ErrorNode,
                                  override val reason: ErrorReason,
                                  override val cached: Boolean = false)
                                  extends AbstractTerminationError {

  override val text = s"Method might not terminate."
}

case class TupleSimpleFalse(offendingNode: ErrorNode) extends AbstractErrorReason {
  override val id: String = "tuple.false"

  override val readableMessage: String =
    s"Termination measure might not decrease or might not be bounded."
}
```

Listing 20: Example of error and error reason regarding termination check in Viper

Based on that, we define error and error reason at Gobra's level as shown in Listing 21.

While back translating the error and error reason, in case that Viper passes `MethodTerminationError` and `TupleSimpleFalse` to Gobra, we generate `MethodTerminationError` and `TupleSimpleFalseError` at Gobra's level. For the info field, we back track the position in the original Gobra program based on the positional info which is provided in Viper's error and error reason. During reporting, we notify the user that the method might not terminate because termination measure might not decreases or might not be bounded based on back tracked lines in the input Gobra program.

```
 case class MethodTerminationError(info: Source.Verifier.Info)
   extends VerificationError {
  override def localId: String = "method_termination_error"
  override def localMessage: String =
    s"The method ${info.trySrc[PSendStmt](" ")}might not terminate"
}


case class TupleSimpleFalseError(info: Source.Verifier.Info)
  extends VerificationErrorReason {
  override def id: String = "tuple_simple_false_error"
  override def message: String =
    s"Termination measure ${info.origin.tag.trim}
    might not decrease or might not be bounded."
}
```

Listing 21: Corresponding error and error reason at Gobra's level for those in Viper shown in Listing 20

4.2 Handling interfaces

The approach for handling mutual recursion in the presence of dynamic method binding is presented in section 3.2. Since abstract methods which are defined in the interface also has an associated `PFunctionSpec` node in the parser AST and it has already been extended to accept termination measure nodes, same as the pre-and postconditions, the users can define termination measures at the place of interface methods. Thus what remains is to add the artificial call to the body of the abstract functions in the interface. This is completed in the desugar. That is, Gobra desugars the interface functions to methods or pure methods with an empty body in the internal representation. In general, for each interface function, we take this resulting method and replace its empty body by the artificial call without changing other parameters as follows: In order to skip the proof for the artificial call in the abstract function to the implementation function, we first introduce an "assume false" statement in the body of the function. In the resulting method from desugaring, the receiver field stores the dynamic type that indicates which implementation for this abstract function is used. Note that a map which contains mapping of each interface to a vector of all its implementations is constructed during the desugaring. Therefore, we can directly obtain all subtypes of the interface from this map. Then we construct a vector of If statements in the internal AST where by each one, the comparison between the type of receive and type of one subtype of the interface is chosen as the

condition. In case that the condition is true, we figure out that this subtype is the implementation which is used and thus, a method call node which indicates the call to the corresponding method in that implementation is placed in the then branch. And the else branch is set to an empty statement. In fact, this vector of If statement is actually equivalent to a nested If statement where we continue comparing the type of the dynamic type defined in the receiver field with the type of different implementations of the interface until the comparison succeeds. Then we call the corresponding implementation function. Finally, the new body of this method is constructed as the vector of If statements with the additional "assume false" prepended to it.

4.3 Inferring Termination Measures

In section 3.3, we introduced how we infer termination measures in Gobra. In this section, we present our concrete implementation for that approach. As mentioned, we allow users to add annotations to enable the heuristics. That is, at the place where the termination measure can be specified, users can write `decreases infer` to express that a termination measure is expected to be inferred there. Then in the Parser AST, the `PInferTerminationMeasure` node which does not take any argument is defined as a subclass of the `PTerminationMeasure` node. During parsing, `decreases infer` is parsed into the `PInferTerminationMeasure` node. That means, the annotation for heuristics is actually treated as a special type of termination measure and like other regular termination measures, it is placed in the `terminationMeasure` field of `PLoopSpec` and `PFunctionSpec`. Note that `decreases infer` cannot be specified in case that other termination measures are defined at the same place. Otherwise, the parser reports an error. Afterwards, no type checking is performed for `PInferTermination Measure` since it takes no argument and nothing should be checked there. In the internal representation, we also define a corresponding AST node called `InferTerminationMeasure` as a subclass of the `TerminationMeasure` node. For desugaring termination measures in Parser AST, if the measure is a `PInferTerminationMeasure` node, we directly construct a `InferTermination Measure` node. Thus after the desugaring is completed, we obtain a program in the internal representation for which the termination measure of functions, methods and loops could be either normal termination measure nodes in internal AST or `PInferTerminationMeasure` node. Then we modify Gobra's workflow: We introduce an `InferenceState` which stores the resulting program from desugaring and a map between integer numbers which denote heuristics and functions defined in that program. Each mapping between a function and the integer indicates that the heuristic which is represented through this number infers a correct termination measure for that function. The map is empty by default. Besides, we also

store another integer number in the `InferenceState` which indicates which heuristic should be applied in the current iteration. By initializing the `InferenceState`, this number is set to 1 and denotes that in the first iteration, the first heuristic should be chosen. The newly introduced internal transformation takes the program in internal representation, the integer number and the map stored in the `InferenceState` as the input. For each function and method, it first checks whether the associated termination measure is a `InferTerminationMeasure` node. If it is not the case, then it further searches for every loop statement contained in the body. For each loop, if the `InferTerminationMeasure` node is defined in the termination measure field, the heuristic for loops is applied and the `InferTerminationMeasure` node is replaced by the corresponding result returned from the heuristic. Otherwise, no changes are made. In case that `InferTermiantionMeasure` node is defined in the termination measure field of the function node and there is a mapping from this function to a integer number in the map, then the heuristic which is indicated by this number is applied to infer the termination measure. Otherwise, it chooses the heuristic for the current iteration indicated by the number in the input for the transformation and applies it. Afterwards, the `InferTerminationMeasure` node there is replaced by the result. Note for all termination measure nodes which are constructed here by the inference, we associate an annotation with each one such that we can distinguish between user-provided and generated termination measures. If the verification fails because of a failing termination check and the position where the error occurs is at an inferred termination measure, then the information in the corresponding error at Gobra's level contains this annotation as well.

Furthermore, in the back translator for errors, we extend it such that after the back translation is done, we check whether each error at Gobra's level is related to the termination check. If it is the case, then we inspect each error. When it contains the annotation mentioned above, we can figure out that the termination check which uses the inferred measure fails. Therefore, we translate the original error to a new defined one called `InferTerminationFailed` and the corresponding error reason to `InferTermination FailedReason`. The intention is that if we do not translate such errors, Gobra only reports the failure of the termination check at the place where the termination measure is inferred without clarifying whether the termination measure is provided by the user or inferred by heuristics.

As introduced in section 3.8, parsing, type checking and desugaring are performed only once. We repeat the following process until the integer number stored in the `InferenceState` reaches the total amount of heuristics provided: We pass the program, map and integer number stored in `InferenceState` to internal transformation for inferring termination measures and then pass the resulting new program in the internal representation

```
case class InferTerminationFailed(info: Source.Verifier.Info)
    extends VerificationError {

    override def localId: String =
        "infer_termination_measure_failed"

    override def localMessage: String =
        "Inferred termination measure cannot verify the termination"
}

case class InferTerminationMeasureFailedReason(node: Source.Verifier.Info)
    extends VerificationErrorReason {

    override def id: String =
        "infer_termination_measure_failed"

    override def message: String =
        s"Inferred measures cannot verify the termination.
           Please provide one by yourself"
}
```

Listing 22: Error as well error reason for unsuccessful inference

to encoder. After that, we create the verification task based on the encoded program in Viper AST and verify it. After the verification is completed, we inspect the verifier result. If the result is success, we stop the repetition and report the success to the user. Otherwise, if there is at least one error which is not `InferTerminationFailed`, we also stop the repetition and report all these errors. In case that all errors are `InferTerminationFailed`, we determine from which methods or functions these errors come by looking into the source information contained in the error. For each methods and function, if there is no `InferTerminationFailed` error which associates to it and there has been no mapping for it in the map of `InferenceState`, we map the method or function to the integer number stored in `InferenceState` to indicate that heuristic used in the last iteration works for it. Finally, we increase the integer number stored in `InferenceState` by 1 such that a new heuristic will be chosen in the next iteration.

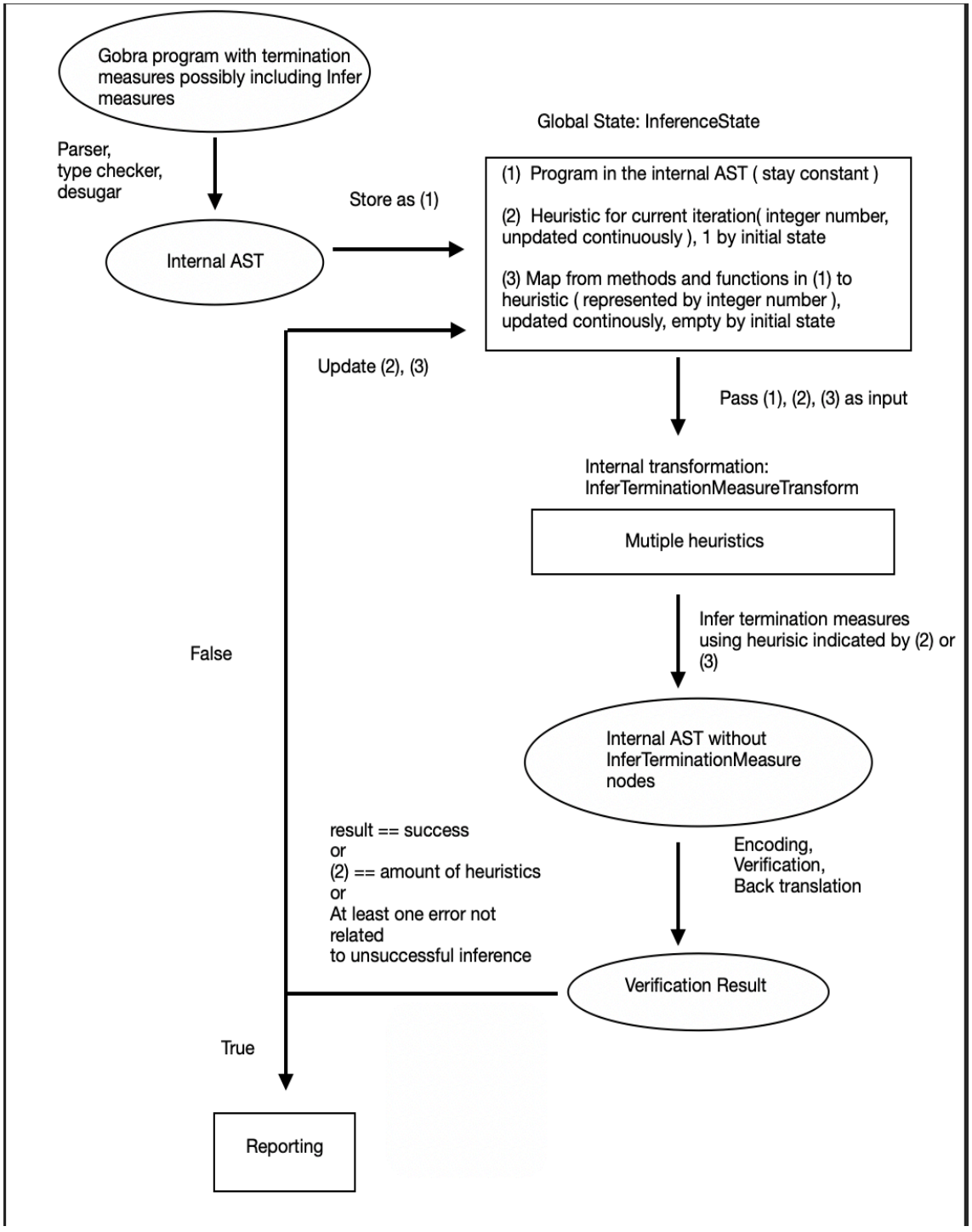


Figure 4.1: Adapted Gobra's workflow for inference of termination measures

Chapter 5

Evaluation

In order to evaluate our implementation, we prepared and ran various tests for different evaluation purposes such as testing the correctness, evaluating the performance and testing the effectiveness by practical examples.

5.1 Correctness Evaluation

In the first place, it should be guaranteed that the existing features and functionalities of Gobra must not be impacted while Gobra is extended by our implementation. In order to check that it is indeed the case, the already provided benchmark which contains regression tests was used. Furthermore, the correctness of our implementation need to be checked as well. Therefore, we extended the existing benchmark with a series of tests written by us which are targeted at checking the correctness of our implementation in different aspects. That is, we tested that our implementation behaves as intended while it is used to verify the termination of simple programs and program with presence of mutual recursion in dynamic method binding or verify the termination with inferred termination measure through the heuristics. For each aspect, we provided positive as well as negative tests. The positive test includes a series of example functions for which termination checking does not succeed. Correspondingly, the negative test contains examples which do not terminate. As such, we annotated the places where the termination check should fail for the CI workflow. Finally, we ran the extended benchmark by using CI workflow in Github and we observed that our implementation passes all tests. Additionally, we noticed two special cases in which the verification fails even if the function actually terminates. However, we figured out that these two issues are not related to our implementation. Consider the following example in Listing 23:

The problem is seeded in Viper's strategy for dealing with the conditional termination measure. In Viper tutorial[4], it states that by conditional ter-

```
decreases x if 1 <= x
decreases _ if x <= 0
decreases *
func sign(x int) int {
  if x == 0 {
    return 1
  } else {
    if 1 <= x {
      return sign(x - 1)
    } else {
      if x <= -1 {
        return sign(x + 1)
      } else {
        return x
      }
    }
  }
}
```

Listing 23: Issue 1: Downgrading from tuple measure to wildcard

mination measure: *"it is not allowed to downgrade from a tuple to a wildcard. That is, if the condition for tuple measure held in the caller's prestate, then the tuple measure must decreases even if the wildcard condition holds at a recursive call"*. In this particular example, the verification fails since if `x` is equal to 1, the termination measure `x` is used and at the recursive call, wildcard is chosen as the measure. Therefore, the problem of downgrading arises. In the future, this issue need to be taken into account by Gobra's user as long as Viper's approach remains unchanged. In some cases, the program may need to be adapted without changing the functionality in order to handle this issue. For this example, it is sufficient to change the termination measure to to `decreases x if 0 <= x; decreases _ if x <= -1;decreases *`. Afterwards, the termination can be verified.

Consider another example:

```
ghost
requires forall i int :: 0 <= i && i < len(s) ==> acc(&s[i], _)
decreases len(s)
pure func GhostSum(ghost s ghost []int) int {
    return len(s) == 0 ? 0 : s[len(s) - 1] + GhostSum(s[:len(s)-1])
}
```

Listing 24: Issue 2: Operations are encoded to Viper function

In this example, both slicing operation `s[:len(s)-1]` and function `GhostSum` are encoded to Viper functions. Therefore, the termination of the corresponding Viper function for slicing is checked while verifying the termination of `GhostSum` itself. Therefore, the verification here fails if the termination measure is not defined for the Viper function which is encoded from slicing. Note that this issue does not occur while verifying termination of non-pure Gobra functions. The reason is that they are encoded to Viper method. Different from functions, Viper does not check the termination of function calls inside the method body while verifying its termination. As a result, we expect Gobra's user to concern about this issue and add the termination measures for encoded Viper functions when needed.

5.2 Performance Evaluation

In the following two subsections, we present the methodology and testing environment used for performance evaluation as well as measurement results.

5.2.1 CAV Test Suite & Testing Environment

The original paper on Gobra[10], is evaluated against a suit of 14 relevant test cases. Our performance evaluation builds on top of that benchmark. In fact, we left out four test cases due to two issues introduced in the previous section since the program need to be adapted in order to fit the termination check. Therefore, this may affect the final result of evaluation. For each test case, we added the necessary termination annotations and collected the time that it takes to verify each file with and without termination checking. In order to compare the performance of Gobra in the case that verification fails, we introduced two additional versions for each example. In the first one, the specifications are adapted such that the failing verification is caused by a wrong specification. In another one, the implementation itself is mutated to introduce implementation errors. The verification time of each verification problem with three versions discussed above was measured in multiple experiments. We extended the test suite for the evaluation of our work. That is, based on the current code for each verification problem, we added termination measures such that Gobra can verify the termination of this example. Besides, we also provided two incorrect variants of it by which all errors are caused by failed termination check and they are related to wrong termination measures and wrong implementation respectively. Afterwards, we measured and compared the verification time of the original test suite and the extended test suite to evaluate the performance when the termination check is performed. We executed 7 experiments in total and took the average while the top and bottom outliers are removed. The experiments

are run on a MacBook Pro with a 2.3 GHz 4-Core Intel Core i5 CPU, 8 GB 2133 MHz LPDDR3 RAM and macOS 10.15.7 running on OpenJDK 11.0.1.

5.2.2 Results

The first table shows the result of average verification time in seconds, including its two versions for wrong specification and incorrect implementation in the extended test suite. The second table shows the corresponding result for the original test suite where the termination check is not performed. The results are rounded up to one decimal place.

Nr.	Example	$T[s]$	$T_{spec\ error}$[s]	$T_{impl\ error}$[s]
1	dense and sparse matrix	33.7 s	30.2s	29.3s
2	dutchflag	12.6s	10.1s	10.5s
3	example 2.1	13.3s	12.6s	12.5s
4	example 2.2	10.8s	10.2s	9.9s
5	heapsort	29.3s	27.5s	28.6s
6	pair insertion sort	23.2s	25.8s	21.8s
7	relaxed prefix	18.2s	12.5s	13.7s
8	parallel search replace	75.1s	31.6s	79.2s
9	parallel sum	61.3s	66.1s	30.3s
10	zune	10.4s	10.2s	9.7s

Nr.	Example	$T[s]$	$T_{spec\ error}$[s]	$T_{impl\ error}$[s]
1	dense and sparse matrix	29.6s	27.9s	28.5s
2	dutchflag	12.2s	11.1s	11.8s
3	example 2.1	11.6s	11.2s	11.7s
4	example 2.2	10.4s	10.6s	11.4s
5	heapsort	27.2s	26.9s	26.6s
6	pair insertion sort	22.8s	20.5s	22.3s
7	relaxed prefix	17.7s	16.9s	13.5s
8	parallel search replace	66.3s	61.6s	68.9s
9	parallel sum	58.5	52.2s	55.7s
10	zune	9.0s	9.3s	10.5s

Based on experiment results, we observed that performing termination check does not significantly increases the verification time.

5.3 Effectiveness Evaluation

Certainly, it is necessary to test whether our implementation can be deployed to verify termination of programs in real-world uses. Therefore, we

decided to verify termination of code from the SCION codebase in order to evaluate the effectiveness of our work.

5.3.1 SCION

The SCION project [5] is a network protocol developed at ETH Zurich which aims to provide route control, failure isolation, and explicit trust information for end-to-end communication. SCION is considered as a potential replacement for the current IP protocol because of its advantages. From our perspective, it is relevant that there is an implementation for SCION protocol which is written in Go programming language and therefore also an implementation in Gobra code which leads to a possible deployment of our work. As a result, based on SCION, we decided to evaluate the performance and effectiveness of our implementation under the environment of complex and practical programs in real-world. However, verifying the termination of the whole implementation of SCION could lead to an extremely high workload. As a result, we focused on a particular component of SCION which is the border router instead.

We chose the following files in Gobra code in the SCION code base for the evaluation: hopfield.gobra, infofield.gobra, base.gobra, decoded.gobra and raw.gobra under the path VerifiedSCION/gobra/lib/slayers/path. They serve as the key component for the router of SCION. The path is a data structure in SCION which describes how packages should be forwarded from one hop to the next and therefore, a path contains a sequence of hops. The file hopfield.gobra contains the definition of the hop field and some functions which realize some functionalities for it. Besides, the path also contains a number of info fields. The definition of info field can be found infofield.gobra which also includes functions to support some functionalities for info field data structure. In fact, the path in SCION has three forms which are base path, raw path and fully decoded path. The file base.gobra included the definition of the base path and a base path contains the important information of the path for instance the number of hop fields and info fields and it is used by both raw-and decoded path. The raw path is defined in raw.gobra which rawly represents the SCION path type. Finally, the the fully decoded path is defined in decoded.gobra. In all these three files, some functions are provided in order to support certain functionalities for each type of path respectively. Currently, the memory safety of functions defined in these files is already verified. At this point, we further verified the termination by using our implementation.

Note that since Go does not promote using recursion, we did not find recursive functions. That means, our heuristic could achieve a high success rate while inferring termination measures. Similar to the methodology discussed in subsection 5.2.1, we evaluated the performance of our implementation

when verifying the selected part of SCION code base. That is, we measured and compared the verification time when verifying the original chosen files without termination measures and the verification time in the case that termination measures are added to all functions and loop statements when needed in these files. The termination check was performed under the same testing environment as mentioned in subsection 5.2.1. Same to the previous evaluation, we executed 7 experiments, exclude the top and bottom outliers and take the average verification time. The result is presented in the next subsection.

5.3.2 Results

Nr.	File	$T_{with\ termination\ check}$[s]	$T_{without\ termination\ check}$[s]
1	hopfield	232.4s	215.7s
2	infofield	32.3s	28.9s
3	base	102.5s	100.2s
4	decoded	198.7s	177.8s
5	raw	189.6s	182.5s

We observed that performing termination check increases the verification time. However, this could be unnoticeable in practical use.

Chapter 6

Conclusion

The core goal of this thesis was to add support for termination checks in Gobra. In general, we extended the parser of Gobra such that the termination measure with the similar syntax as in Viper could be accepted by Gobra. Then each phase of Gobras workflow was modified without impacting the existing features such that termination measures could be processed at each step in the pipeline like other annotations specified by the users. That is, we type checked the termination measures in the type checker and the desuager is now able to desugar the measures to the internal representation by removing the syntactic sugars. Finally, with the help of the termination plugin supplied in Viper which allows the frontend to directly define the decreases node at the level of Viper AST, we devised a sound encoding of termination measures into Viper in the sense that successful verification guarantees that the provided Gobra program terminates. In addition, we extended the back translator for errors in Gobra such that Gobra is able to notify users that the termination check is not successful by providing understandable messages at Gobra's level. Otherwise, Gobra reports the successful result accordingly. Furthermore, except for the termination measures currently supported in Viper, we also introduced support for termination measure's inference via heuristics. If all heuristics failed, Gobra reports an error and asks the user to provide one. Due to our architecture, more heuristics can be easily added to Gobra. In the next section, we shortly discuss the possible extension goals and future work for our thesis.

6.1 Future Work

Further heuristics for termination measures in Gobra

In the context of our thesis, the support for heuristics regarding the termination measures is realized by a Gobra to Gobra transformation. The

modularity is considered as one of the advantages of our architecture. As a result, we expect more heuristics to be added in Gobra by the future work.

In our thesis, the heuristics are added at Gobra's level which means that they provide convenience for Gobras user. We believe that the termination measures could be inferred at the Viper instead of at the Gobra-level so that other Viper frontends can take advantage of this feature.

Other termination checkers as Gobras backend

Currently, we use Viper as Gobras backend verifier for the proof of termination. As a result, the functionality of Gobras termination check is then limited by the support in Viper for instance it is hard to introduce termination measures in Gobra in new forms which are not included in the Vipers termination plugin. In the future work Gobra could be connected to other state-of-the-art termination checkers to make termination checking and inference more powerful.

Computational Complexity Proofs

Since it is often crucial to calculate how many times a recursive function is invoked and how many iterations a loop has when analyzing the complexity of a program, in addition to proving termination, our approach could be extended to prove an upper and potentially a lower bound on the computational complexity of a program.

Bibliography

[1] "Go Programming Language." [Online]. Available: https://golang.org

[2] "Gobra Project." [Online]. Available: https://github.com/viperproject/gobra

[3] "Gobra Tutotial." [Online]. Available: https://github.com/viperproject/gobra/blob/master/docs/tutorial.md

[4] "Viper Tutotial." [Online]. Available: http://viper.ethz.ch/tutorial/?page=1&section=#introduction

[5] "SCION Internet Architecture." [Online]. Available: https://www.scion-architecture.net

[6] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[7] Fabio Streun, "Tool Support for Termination Proofs," Bachelor's thesis, ETH Zurich, 2019

[8] Richard L. Ford, K. Rustan M. Leino, Dafny Reference Manual, draft, 2017-08-14 09:40, https://homepage.cs.uiowa.edu/~tinelli/classes/181/Papers/dafny-reference.pdf

[9] Barbara Liskov, Jeannette Wing, "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, 1994

[10] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular Specification and Verification of Go Programs," In *Computer Aided Verification (CAV)*, 2021.

[11] Voirol, Nicolas; Kandhadai Madhavan, Ravichandhran; Kuncak, Viktor, "Termination of Open Higher-Order Programs," EFPL, 2017

[12] Robert W. Floyd. "Assigning Meanings to Programs", Dordrecht, 1993.

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):

Verifying Termination of Go Programs

Authored by (in block letters):

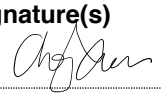
For papers written by groups the names of all authors are required.

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

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)**
Jiaxing, 29.September 2021	

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