ETH zürich

Advanced Logical Proofs in a Verifier

Bachelor's Thesis

Dina Weiersmüller

March 20, 2023

Advisors: Prof. Dr. Peter Müller, Linard Arquint, Thibault Dardinier

Department of Computer Science, ETH Zurich

Abstract

Viper is an intermediate verification language with back-ends that verify whether a Viper program is correct. In this thesis, we extend Viper with reasoning features such as existential elimination, universal introduction and lemmas. Our implementation of these features checks the necessary side conditions to ensure their soundness. We require for instance that the proof of a universal introduction does not influence the rest of the program. Hence, we devise an information flow analysis to check this side condition. As Viper verifies programs in a modular way, this information flow analysis is also modular. Therefore, we introduced a new annotation for method specification that allows the analysis to handle method calls in a modular way.

Contents

Chapter 1

Introduction

Viper [1] is an intermediate verification language with back-ends that verify whether a Viper program is correct. A Viper program is correct if all its methods are partially correct. A method is partially correct with respect to a precondition and a postcondition iff the resulting state after executing the method satisfies the postcondition given that the initial state satisfies the method's precondition. A program is verified by Viper in a modular way, which means that the program is split into methods that are verified separately. Viper also includes other specifications, for instance the specification of loop invariants.

In general, Viper is capable of verifying non-trivial programs such as programs modifying the heap. Thus, Viper is being used by several front-ends to verify programs written in Go [2], Rust [3], Java [4], and Python [5]. Complex properties and proof obligations can be expressed using the restricted set of language features offered by Viper by finding a suitable encoding. This thesis adds three language features to Viper, which we implemented in a Viper plugin. A Viper plugin extends the input language of Viper by additional language features and internally encodes these features. The three language features added are lemmas, existential elimination, and universal introduction.

Before lemmas were added as a language feature, each user had to consider the following to prove a lemma. A user formulated a lemma as a method with the proof in the body. The proof of a lemma has to fulfill certain side conditions to be valid. For instance, the proof of a lemma should not influence the rest of the program executed at runtime, so there should be some separation of the proof and the rest of the code. Additionally, the code for the proof of the lemma should terminate. A user had to be thorough in checking these side conditions such that the lemma's proof is indeed valid.

Another added language feature is existential elimination. If we have an existentially quantified property and we want to obtain a witness for which the property is satisfied, we use existential elimination. However, obtaining a witness is only

possible if there exists such a witness for which the property is satisfied. Before, a user had to first show that such a witness exists by asserting that the existentially quantified property holds. Then this user picked a variable with an arbitrary value and assumed this is the witness by assuming the property for this variable. This is only a valid existential elimination if the property in the assert and assume statement is the same. If this property is complex, mistakes can be easily introduced.

The final language feature added to Viper is universal introduction which is a well known rule in predicate logic. The rule introduces the universal quantifier for a property. This is useful when proving that a certain property holds for any variable. Similar to the lemma universal introduction also has side conditions that need to be respected. For instance, the code which proves the universal introduction should be separated from the rest of the program during runtime, similar as for the proof code of lemmas. This and all further side conditions of universal introduction are checked in the language extension such that a user wanting to implement universal introduction does not have to check them.

The structure of this thesis is as follows. Firstly, we introduce the new language features of Viper. We show their encoding and explain the necessary side conditions for the features to be correct. In the case of the universal introduction, one such side condition requires us to check that values from the proof do not influence the program. Therefore, we explain in Chapter 3 a simple approach to track the information flow from the proof to the program. However, we will see that this approach is non-modular. Thus, in Chapter 4 we present a modular approach for analysing the information flow. Finally, we evaluate the implemented Viper plugin based on both approaches.

Chapter 2

Sound Reasoning Features

In this chapter we discuss the three language features that we extend the Viper with, in order to ensure the soundness of certain reasoning steps. The three features we added are the elimination of the existential quantifier, the introduction of the universal quantifier and lemmas. We will explain each of these features and we will show how the Viper language was extended with a Viper plugin to include them.

2.1 Lemma

Currently, Viper does not offer support for lemmas and therefore the user has to simulate lemmas with methods. For instance, if we want to prove something by induction we want the method that simulates the induction to be a lemma. A lemma requires the precondition of an induction and based on that proves the induction statement. The prove of the induction can then be written in the body of the lemma where the case split between base case and induction step can be implemented using an if statement and a recursive call of the lemma corresponds to the induction hypothesis. A lemma is essentially a method with two side conditions. For one, a lemma must be guaranteed to terminate.

```
1 method lemma()
2 ensures false
3 {
4     lemma()
5 }
```

Figure 2.1: Non-terminating lemma

For instance, this lemma does not terminate which is equivalent to an incorrect induction proof without a base case and one could simply prove false as shown in the example in Figure 2.1. Furthermore, a lemma should be a method with a

body that does not write to the heap. If a lemma were able to write to the heap it would be able to modify the behaviour of the rest program. However, the body of the lemma is the proof of this lemma and therefore should not influence the rest of the program.

2.1.1 Syntax in Viper

To declare that a method is a lemma the following syntax was added to the Viper language.

```
1 method lemma1()
2 isLemma
3 {
4     <user provided code>
5 }
```

Figure 2.2: New syntax for lemmas

The new expression `isLemma` can be added as either a precondition or a postcondition in the signature of the method. By adding this expression, two checks are executed to ensure that the two previously mentioned requirements are fulfilled. Firstly, to prove that the lemma terminates, a `decreases` clause should be added to the method signature by the user. The `decreases` clause triggers a check for termination by the termination plugin [6] of Viper. Moreover, it is checked that the lemma does not write to the heap. Therefore, all statements in the user provided code are considered. The following statements are disallowed in this code block: assignment to a field, `inhale`, `exhale`, `fold`, `unfold`, `apply`, and `package`. Moreover, method calls are not allowed, however lemma calls are allowed since they clearly fulfill the side conditions of a lemma. If any of the above listed statements were allowed it would be possible for lemmas to influence the behaviour of the rest of the program. But we want to be able to prove the rest of the program independent of this lemma.

2.1.2 Lemma in an Old Context

As mentioned in the previous section, a lemma cannot modify the heap. Therefore, it is now sound to call a lemma in the old context which means to prove a lemma on the old state of the heap. To enable such lemma calls in an old context, we extended the Viper language with the following new syntax.

```
1  label l
2  var t: T
3  t := oldCall[l](lemma1())
```

→

```
1  label l
2  assume old[l](precondition)
3  havoc t
4  assert old[l](postcondition)
```

Figure 2.3: oldCall syntax and encoding

Where `t` are optional target variables, which are variables that the lemma call can be assigned to. `precondition` is the precondition and `postcondition` is the postcondition of the lemma which need to be assumed and asserted respectively in the old context. Therefore, we can use the lemma in an old context. However, since we only consider the lemma's pre- and postcondition and not the body, we do not know what values the target variables might be assigned. Thus, they are havocked in line 3 of our encoding.

2.2 Existential Elimination

Another reasoning feature we added is existential elimination. Existential elimination means that if it can be proven that $\exists x.P(x)$ holds, then the existential quantifier can be removed and a witness w can be obtained such that $P(w)$ holds. Up until now this had to be done by the programmer. This meant that if we wanted to find witnesses ws for some expression P with an arbitrary number of variables, P had to be written twice as can be seen in the encoding, which lead to code duplication. Furthermore, if P is complex and one variable had to be changed, mistakes could easily be introduced since P had to be changed in two places.

2.2.1 Syntax and Encoding in Viper

The newly added syntax for existential elimination in Viper is encoded the following way:

```
1  obtain x:T where {P(x)} P(x)
```

→

```
1  assert exists x: T :: {P(x)} P(x)
2  var w: T
3  assume P(w)
```

Figure 2.4: Existential elimination encoding

This encoding is straightforward. Here, `P(x)` is some arbitrary expression that

takes `x` as an argument. The assert statement on line 1 is necessary since if we would omit this statement we could assume false in line 3 if `P(x)` does not hold for any `x`, and therefore we could prove false. With the assert now added, if there does not exist `x` for which `P(x)` holds, the execution is simply stopped and an assertion error is thrown. If we know that there exists such a variable `x` we can then assume for an arbitrary variable `w` that `P(w)` holds. Therefore, we have found a witness `w` for our expression `P(w)`.

2.3 Universal Introduction

After proving that $P(x)$ holds for an arbitrary variable x, universal introduction can be applied to obtain $\forall x.P(x)$. In general we would like to be able to prove that forall k for which $P(k)$ holds in an old context, that $Q(k)$ holds in the current context after the execution of the universal introduction body.

2.3.1 Syntax and Encoding in Viper

The new syntax and encoding for universal introduction is the following:

```
1  prove forall k:T {P(x)} assuming P(k) implies Q(k) {
2      <user provided code>
3  }
```

```
1  var b: Bool
2  var k: T
3  label l
4  if(b) {
5      assume P(k)
6  }
7  <user provided code>
8  assert b ==> Q(k)
9  assume forall k1:T :: {P(x)} old[l](P(k1)) ==> Q(k1)
```

Figure 2.5: Universal introduction encoding

Here, `P` and `Q` are two arbitrary expressions. The assumption on `k` needs to be guarded by a boolean value. Otherwise, if `P(k)` is not satisfiable, then `assume P(k)` kills all traces and the rest of the program trivially verifies, even though it might be incorrect. Therefore, the if statement on line 4 is added including an arbitrary boolean value `b` which ensures that the rest of the program is still checked. Moreover, a label on line 3 is added such that we can use `P(k1)` in the forall assumption on line 9 in the state it had before the provided code was executed.

Furthermore, we need `k` to be immutable. If we allow `k` to be modified we would assert `Q(k)` for a different `k` than we assumed `P(k)` with. This is incorrect because we want to prove the universal introduction for the same `k` in both cases.

Additionally, we require that all quantified variables in the universal introduction statement do not influence other variables that are in scope outside of the universal introduction code block. The program executes independent of the execution of the proof of the universal introduction. To understand why this is a problem, consider the following example.

```
1  var x:T
2  prove forall k:T assuming true implies k == x {
3      x := k
4  }
```

Figure 2.6: Example of an assignment to a variable in scope outside of the universal introduction block

In this example, the value of `x` after the universal introduction statement, has been altered and we have somehow proven that `x` is equal to all values of type `T`. This is obviously wrong and therefore we cannot allow the quantified variables in the universal introduction statement to influence any of the variables in scope outside of this statement and present in `P` or `Q` which includes the heap. In Figure 2.6 this is the case in the expression `k==x`. To avoid such faulty proofs, we need to check whether a quantified variable in the universal introduction statement influences a variable that is also present outside of the universal introduction scope. Therefore, we need to perform an information flow analysis of the code block inside the universal introduction statement to check the information flow from the proof to the rest of the program.

Chapter 3

Information Flow Analysis - First Attempt

As mentioned in the previous chapter, we want to collect all the variables whose value was influenced by the value of a quantified variable in the universal introduction and check whether these variables are also present outside of the scope of the proof. In particular, we must ensure that the quantified variable in a universal introduction is not assigned to a regular program variable or the heap and hence cannot modify the rest of the program. Therefore, we treat such quantified variables as tainted and perform an information flow analysis. In this first attempt, we track tainted variables which we store in a set. Variables are considered tainted when their value might depend on the value of an already tainted variable. For each variable that we consider tainted at the beginning of a statement, we compute a set that contains all the variables whose value was influenced by the originally tainted variable and consider these variables now tainted as well. For example, if the value of an initially tainted variable changes, the values of all variables in its tainted set might change as well. This chapter is structured as follows: Section 3.1 provides details on how the analysis handles individual statements, and Section 3.2 illustrates the limitations of this method of analysis that we overcome in Chapter 4 with a different analysis.

3.1 Analysis Details

In Section 2.3.1 we establish that we want to track which variables are influenced by the quantified variables in order to determine whether the universal introduction might influence the rest of the program. Therefore, the initially tainted variables are the quantified variables from the universal introduction and our information flow analysis is triggered by a universal introduction statement. Thus, when the analysis is triggered it adds the quantified variables to the set of tainted variables. Then the analysis iterates over all the statements in the code block inside the

universal introduction statement and adds variables to or removes them from the set according to the statement. In this section, we look at several statements and explain how they influence the set of tainted variables. We use f as the function that takes a tainted set S and a statement $stmt$ as inputs and returns the updated set of tainted variables after executing $stmt$.

3.1.1 Sequential Composition

$$f(S, s1; s2) = f(f(S, s1), s2) \tag{3.1}$$

For sequential composition $s1; s2$, we first compute the tainted set for executing $s1$ and use the resulting set as input for the analysis of $s2$. This is correct, since the two statements happen consecutively and therefore all the variables that are added to the tainted set in statement $s1$ are tainted when statement $s2$ is executed.

3.1.2 Local Variable Assignment

$$f(S, lhs := rhs) = \begin{cases} S \cup lhs & \text{if } tainted(rhs) \\ S \setminus lhs & \text{otherwise} \end{cases} \tag{3.2}$$

The analysis either adds the variable on the left-hand side to or removes it from the tainted set based on whether the right-hand side expression is tainted or not. If the expression rhs is tainted, the variable lhs is added to the tainted set because it has a tainted value after the assignment. Similarly, if the expression is untainted, the variable lhs stores an untainted variable after the assignment and thus we can remove the variable lhs from the set of tainted variables. To decide whether the expression rhs is tainted or not we use the auxiliary function $tainted$. The function $tainted(exp)$ takes an expression exp as an argument and returns whether this expression contains a variable that is present in the tainted set.

3.1.3 If Statement

$$f(S, \text{if}(cond)\ \{s1\}\ \text{else}\ \{s2\}) = \begin{cases} S \cup mod(s1) \cup mod(s2) & \text{if } tainted(cond) \\ f(S, s1) \cup f(S, s2) & \text{otherwise} \end{cases} \tag{3.3}$$

Since either the if block $s1$ or the else block $s2$ is executed we need to compute the analysis for both of these blocks. The same set S is used in both computation because only one of the two blocks $s1$ or $s2$ will be executed. We cannot know which of these blocks is executed therefore we overapproximate the tainted variables by taking the union of the tainted variables resulting from the analyses of $s1$ and $s2$. The condition $cond$ is the determining factor which block is executed. Therefore the value of $cond$ influences the value of all the variables that are modified in either of the blocks $s1$ and $s2$. Hence if the condition is tainted, the variables that are modified in these block are tainted as well and must be added to the set of

tainted variables. To decide whether the condition is tainted the auxiliary function $tainted$ is used as explained in Section 3.1.2 and to determine which variables are modified in the respective blocks the auxiliary function mod is used. The function $mod(stmt)$ takes a statement $stmt$ as an argument and returns all variables that are modified by this statement. Modified means that the value of the variable was changed by this statement. This is the case, when the variable is assigned a new value by the statement.

3.1.4 While Statement

$$f(S, \text{while}(cond)\ \{body\}) = \begin{cases} S & \text{if } S = f(S, \text{if}(cond)\ \{body\}\ \text{else}\ \{\}) \\ f(S', \text{while}(cond)\ \{body\}) & \text{otherwise} \end{cases} \tag{3.4}$$

For the while statement there are several possibilities to be considered. First, the while body could be executed several times or never. If the body is not executed at all, all the variables that were previously in the set of tainted variables need to be kept inside this set. Therefore, one iteration of the loop body is the same as executing the analysis on an if statement where in one block the body is executed and in the other block nothing is computed. As explained in Section 3.1.3, the union of the two resulting sets from the if and else block is taken. Since we leave the else block empty, the union is taken from the resulting set of the execution of the body in the if block and the initial tainted set.

However, we cannot know how many times the body of the loop is executed, thus we need to overapproximate the variables that can be tainted. As in the if statement, we can overapproximate the set of tainted variables by taking the union of the tainted set after executing the loop body once, twice, and so on. Because there are a finite amount of variables in scope, there are also a finite amount of variables that can be added to the set of tainted variables. Thus, and because the set can only grow with each iteration, the set will not change after a certain amount of iterations of the loop body. When we reach this point, where the set does not grow anymore, we have found the fixed point of the set. In other words, when the set S is equal to the set that is returned after a further iteration we have reached the fixed point of set S and can return this fixed point. Otherwise, we compute a further iteration of the loop body.

3.2 Limitations

The presented information flow analysis computes for a given set of tainted variables the set of variables that are tainted after performing some statement. Since the analysis is specific to a particular set of initially tainted variables, this

analysis is non-modular as we will illustrate with a method call. Consider a method being called twice in the following manner.

```
1  method m(a:T,b:T) returns (c:T,d:T)
2  {
3      c:=a
4      d:=b
5  }
6   method main()
7   {
8      var r1: T
9      var r2: T
10     var r3: T
11     var r4: T
12     prove forall x:T {P(x)} assuming P(x) implies Q(x)
13     {
14         var y: T
15         r1,r2 := m(x,y)
16         r3,r4 := m(y,x)
17     }
18 }
```

Figure 3.1: Method calls with set flow analysis

We want to modularly determine which of the variables `r1` to `r4` are tainted. With a non-modular analysis we would simply go through the method body of the called method and execute the same analysis described in the previous section on the method body to determine the tainted return variables and therefore which variables `r1` to `r4` are tainted. Thus, it depends on method `m` which of the variables, that the method call is assigned to, are tainted. To be modular and thus not to analyse the body of method `m` when considering a call to `m`, we want to annotate the callee. This annotation should express the method's effects on the set of tainted variables. Since we call the variables in the tainted set tainted, it follows naturally to annotate each tainted variable as tainted when passed to a method. When we consider the first method call to `m` on line 15, we therefore want to add an annotation to the variable `x` since this is the quantified variable of the universal introduction statement and thus tainted. We simply add the annotation `tainted` to mark that when calling method `m`, the first argument can be a tainted variable. To avoid having to analyse the whole body each time, the return variables are annotated accordingly with the tainted keyword as well. Hence, the method specification for `m` for the method call on line 15 looks the following way.

```
1  method m(tainted a:T,b:T) returns (tainted c:T,d:T)
2  {
3      c:=a
4      d:=b
5  }
```

Figure 3.2: Method signature with tainted keyword

Now the callee in the method `main` in Figure 3.1 can analyse the method specification of `m` and determines that the variable `r1` should be added to the set of tainted variables. However, on line 16 the callee sees the same method specification of `m` and therefore assumes that again the first argument can be tainted and therefore the first returned variable, namely `r3` is added to the set of tainted variables. But this leads to an incorrect tainted set. We can easily see that on line 16 the second argument, in this case `x`, that is passed to method `m` is a tainted variable. Thus, looking at the body, variable `r4` should be added to the tainted set, which is not the case.

To solve this problem, we would therefore need a method specification of `m` that annotates the second argument of the method as well as the second return variable with the `tainted` keyword. One possibility is to have several method specifications with all possible method arguments annotated as tainted or not annotated. This would return the correct tainted set that results from the method call. However, this leads to specification duplication and every time method `m` is called, we need to check the method call against every specification which results in verification overhead. But this verification overhead is exactly what we want to avoid by modular verification. Another possibility is to generally annotate all method arguments as tainted and annotate the according return variables as tainted. However, in this case the computed tainted set might overapproximate the actual tainted set substantially. In Figure 3.1 the analysis would add all variables `r1` to `r4` to the tainted set instead of only `r1` and `r3`.

In the next chapter, we present a graph-based approach that allows us to express more precisely how input variables impact output variables. For the example above, we can express that variable `a` impacts only variable `c` and the can be expressed for variables `b` and `d`.

Chapter 4

Modular Flow Analysis with Graphs

In the previous chapter, we introduced an information flow analysis, in order to ensure that the quantified variables in the universal introduction statement do not influence the rest of the program. However, it was established that the handling of method calls, in particular the use of the `tainted` annotation does not work efficiently, see Section 3.2. Thus, in this chapter we introduce a more fine-grained approach to specify methods as shown in Section 4.1. This approach allows us to specify which variable is influenced by which other variables. This solves the issue of multiple specifications or the overapproximation of the set of tainted variables. The tainted analysis discussed in the previous chapter could be used to check this new specification. However, this does not scale well. For each method, we need to execute the analysis for each method argument and additionally for a variable that represents the heap. Therefore, we develop a direct compositional approach for the modular information flow analysis based on graphs as shown in Section 4.2.

4.1 Flow Annotation

To solve our problem from Section 3.2, namely having multiple method specifications, we introduce an annotation to each method instead of marking each variable separately as tainted. Instead of having different method specification for every method argument, we can simply add an annotation to the method for each return variable, that specifies a set of method arguments this return variable might be influenced by. Therefore, we can simply check if any of the variables in this set is tainted. If this is the case, the returned variable will be tainted as well, otherwise not. The problem with having multiple specifications or a considerably overapproximated set of tainted variables is therefore solved. Additionally, the heap as a whole is always considered a method argument as well as a method return variable, since a method can modify the heap and read from it. This flow annotation has the following new syntax.

```
1 method m(a1:T,a2:T) returns (r1:T,r2:T)
2 influenced r1 by {}
3 influenced heap by {heap,a2}
4 {
5     <user provided code>
6 }
```

Figure 4.1: New syntax for flow annotation

Based on the code block inside the method, the user can now specify these annotations. In this case, for `r1` it is specified that it is not influenced by any method argument. For `r2` there was no annotation written, therefore in case the information flow is analysed, it is assumed that `r2` might be influenced by all method arguments including the heap to overapproximate the actual information flow. Since the heap is always an implicit method argument and return variable, we can also specify this annotation for the heap. The heap must always be influenced by itself and in this case the heap is also influenced by method argument `a2`.

4.1.1 Verification of the Flow Annotation

For each method, it is verified that its flow annotations are correct. This means, that the annotation should be exact or overapproximate the method arguments that influence the return variable. If we underapproximate the method arguments that influence the return variable, we miss a variable that influences the return variable and therefore the information flow analysis is incorrect. However, if we overapproximate the variables by which a return variable is influenced, all the variables that actually influence this variable will be listed and therefore the information flow analysis is correct.

4.2 Analysis with Graphs

An intuitive way of representing this influenced by relation created by the new flow annotation shown in the previous section, is to show this relation with an edge in a graph where the vertices represent the variables. Therefore, in this approach we use a graph to create the information flow analysis. After trying out different types of graphs we decided on the directed bipartite graph $G(V_{init} \cup V_{final}, E)$ as the new data structure used for the information flow analysis. The idea is to build a graph that shows which initial value of a variable before a statement influences which final value of a variable after a statement. Therefore, we defined the vertices V_{init} and V_{final} as follows. V_{init} contains all variables in scope and their value before a statement. These vertices only have outgoing edges. V_{final} contains the corresponding final values of each variable after a statement. These vertices only have incoming edges. V_{init} and V_{final} both contain a variable heap that represents the state of the heap before and after a statement.

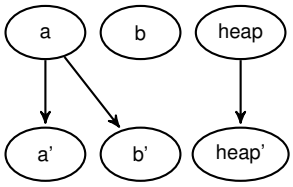

Figure 4.2: Example graph

In this example graph in Figure 4.2 `a`, `b`, and `heap` represent the variables in scope and their value before a statement. `a'`, `b'`, and `heap'` represent the same variables but their value after the statement. As previously stated, an edge shows an influenced by relation between two variables. Therefore, we learn from this graph, that `b` after the statement is not influenced by its value before the statement but rather by the value of `a` before the statement. We also learn that the value of `a` after the statement is still influenced by the value of `a` before the statement. The same holds for the `heap`. This leads directly back to the previously explained method annotation in Section 4.1. Each annotation refers to a set of edges between the method arguments and the return variable of the annotation. Thus, we can easily build a graph out of the flow annotations. The graph for Figure 4.1 looks the following way.

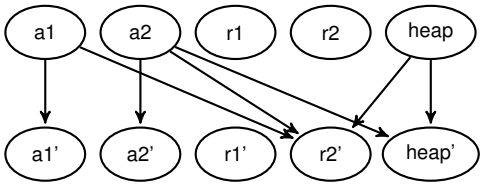

Figure 4.3: Flow annotation graph

We can use this flow annotation graph to further analyse method calls in a modular way, as shown in the next Section 4.3.

4.3 Computing the Flow Analysis Graph

Given this annotation, we can now define the rules to create the flow analysis graph for each statement. In the following subsection, `computeFlowGraph (S, stmt)` will denote the method that given the set `S` of current variables in scope, and a statement `stmt` computes the resulting graph. Furthermore, vertices labelled with the variable name represent the initial value, while vertices labelled with the variable name concatenated with the prime symbol "′" represent the value of the variable after the statement.

4.3.1 Local Variable Assignment

The analysis adds edges to the graph based on the left-hand side and the right-hand side of the assignment. In particular, after an assignment the right-hand side determines the value of the variable on the left-hand side and therefore the left-hand side is influenced by the right-hand side.

Example

```
1 method m(a:T) returns(c:T)
2 {
3     c:=a
4 }
```

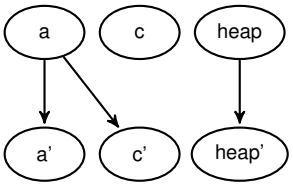

Figure 4.4: Local variable assignment example

In this example variable `a` is assigned to variable `c`. Therefore, the edge from `a` to `c'` needs to be added. However, for the values that have not been changed by this statement we also need to add the edges from their initial value to their final value since their final value is trivially influenced by their initial value. If these edges would not be added, the information would be lost by what variables their value is influenced by or we would assume that their value is not influenced by any variable in scope. We can formulate this statement the following way in a rule.

Rule: `computeFlowGraph(S, lhs:=rhs)`

```
  1 G := emptyGraph(S)
  2 if (containsHeapAccess(rhs)) {
  3     G.addEdges({heap},{lhs})
→ 4 }
  5 G.addEdges(rhs,{lhs})
  6 G.addIdentityEdges(S\lhs)
```

Figure 4.5: Local variable assignment rule

We start computing the graph for the local variable assignment with the empty graph `G`. To create the empty graph `G` the function `emptyGraph(S)` is called. It takes a set of variables `S` as an argument and returns a graph containing no edges and vertices representing the initial value and the final value of all variables in `S`.

If the expression `rhs` contains an access to the heap, we add an edge from the variable `heap`, denoting the whole heap, to the the vertex representing the final value of `lhs`, namely `lhs'`. Then we add an edge from all the variables in `rhs`

to `lhs'`. The auxiliary function `containsHeapAccess(rhs)` takes an expression `rhs` as an argument and returns whether this expression accesses the heap. Furthermore, to add edges to the graph we have function `G.addEdges(S1,S2)` which takes two sets of variables `S1`, `S2` as arguments and adds an edge in the graph `G` from each vertex representing the initial value of a variable `v1` in `S1` to to each vertex representing a final value of a variable `v2'` in set `S2`.

Finally, we add all the identity edges for the remaining variables that are not `lhs` but that are in scope. Therefore, the auxiliary function `G.addIdentityEdges(S)` is called on a graph `G` and takes a set of variables `S`. It adds all the identity edges of the variables in set `S`, meaning that the function adds all edges from the vertex representing the initial value to the vertex representing the final value of the variables in set `S`.

4.3.2 Sequential Composition

Given two sequential statements we compute the flow analysis graph for each of them. Because these two statements are sequential the vertex representing the final value of a variable in the first graph is the same as the vertex representing the initial value of the same variable in the second graph. Now we can follow the edges to see which initial value of a variable in the first graph influences which final value of a variable in the second graph and create the final graph for the sequential composition.

Example

```
1 method m(a:T) returns(c:T,d:T)
2 {
3       c:=a
4       d:=c
5 }
```

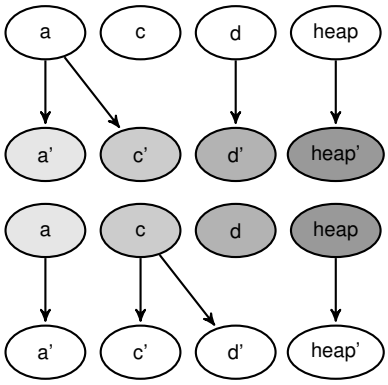

Figure 4.6: Sequential composition example

In this example, first the graph for the local variable assignment on line 3 is computed and then the graph for the local variable assignment on line 4 is computed as explained in the previous section. Therefore, we know that in the first graph `a` influences `c` and in the second graph `c` influences `d`. Thus, there exists a path from `a` to `d'` and we know that `a` influences `d`. We can build a new graph that

removes the vertices in the middle, particularly the vertices representing the final values in the first graph and the vertices representing the initial value in the second graph. This final graph only includes an edge from the vertices representing the initial value in the first graph to the vertices representing a final value in the second graph if there exists such a path.

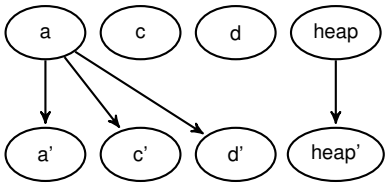

Figure 4.7: Final graph of sequential composition example

Hence, the rule for sequential composition follows.

Rule `computeFlowGraph(S,s1;s2)`

```
                1  mergeGraphs(computeFlowGraph(S,s1),
→               2               computeFlowGraph(S,s2))
```

Figure 4.8: Sequential composition rule

The function `mergeGraphs(g1,g2)` takes two graphs `g1` and `g2` and finds the path from any vertex representing an initial value of a variable from the first graph to any vertex representing a final value of a variable in the second graph and merges them into one graph.

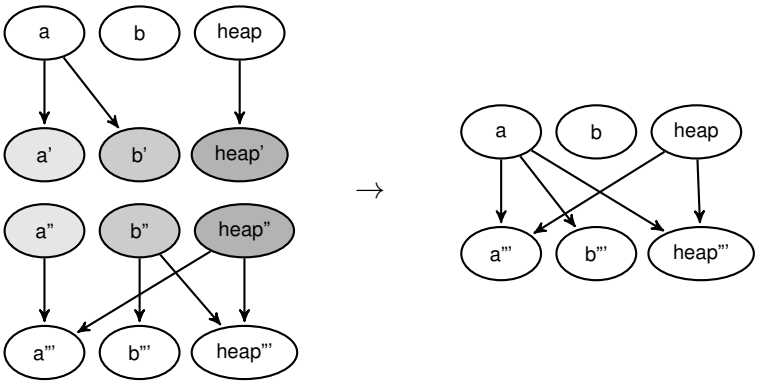

Figure 4.9: Merge two graphs

Here, we set the vertices that are single prime equal to the vertices that are

double prime and find all the paths. Hence, the resulting graph only contains the vertices representing the initial value before the first statement and the vertices representing the value of the variables after both the statements have been executed.

Therefore, this function computes the final graph for the sequential composition as explained in the previous paragraph.

4.3.3 If Statement

In an if statement the condition determines whether the if or the else block is executed. Therefore, each assignment in these two blocks is dependent on which branch is taken. Since there is this dependency on the condition, we can state that the value of the variables in the condition influences the final value of the variables that are assigned to in either the if or the else block. Therefore, an edge should be added between all the variables in the condition and the variables that are modified in either the if or the else block. Now, we have computed the first graph necessary for the if statement. We also need to compute the graphs resulting from the if block and the else block. When all three graphs are computed, we can take the union of these graphs. This means that we take the union of all edges and keep the same vertices. The union overapproximates the variables that influence each other, since we cannot know which branch is taken.

Example

```
1  method m(a:T,c:T)
2  returns (d:T,e:T)
3  {
4      if(c) {
5          d:=a
6      } else {
7          e:=a
8      }
9  }
```

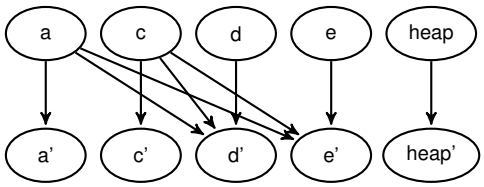

Figure 4.10: If statement example

As stated above, we add all the edges from the condition to all the variables that were modified in either the if or the else block, in this case variables `d` and `e`. Furthermore, we add in the if block the edge from `a` to `d'`. According to the local variable assignment rule, shown in Section 4.3.1, all the other identity edges from `a`, `c`, `e`, and `heap` need to be in the graph as well. In the else block, we add the edge from `a` to `e'` as well as all the identity edges from the variables other than `e`, once again because of the local variable assigment rule. The resulting graph then takes the union of all the edges and creates the final graph containing all edges.

Rule `computeFlowGraph(S,if (cond) {s1} else {s2})`

```
1  G := identityGraph(S)
2  if containsHeapAccess(cond) {
3      G.addEdges({heap},(modifiedVars(s1)
4                         ∪ modifiedVars(s2))
→  5  }
6  G.addEdges(cond,(modifiedVars(s1) ∪ modifiedVars(s2)))
7  G.removeIdentityEdges(modifiedVars(s1)
8                         ∩ modifiedVars(s2))
9  computeFlowGraph(S,s1) ∪ computeFlowGraph(S,s2) ∪ G
```

Figure 4.11: If statement rule

First, we check whether the condition contains a heap access, with the auxiliary function `containsHeapAccess`, as explained in Section 4.3.1. If the condition contains a heap access, we need to add an edge from the initial value of the `heap` to the final value of the variables that have been modified in either the if or the else block. These variables are computed using the auxiliary function `modifiedVars(stmt)`, which takes a statement and returns all variables that are modified by the statement. In particular, all variables which are on the left-hand side of an assignment in that statement. Then we go through all the variables in the condition and add the edges from these condition variables to the variables that were modified, using the `addEdges` function, as explained in Section 4.3.1. Now we can remove the identity edges from the variables that were modified in both the if and the else block. This is correct, because if a variable is modified in both the if and the else block we know that both graphs computed from the if and the else block will add or remove an edge and therefore we can remove the identity edge for this variable. To remove the identity edges the function `G.removeIdentityEdges(S)` is used, which for a graph `G` removes all edges from the vertices representing initial values in the set of variables `S` to the vertices representing the final values of these variables. Finally, we take the union of the three computed graph, namely the graph from the if block, the graph from the else block and the graph resulting from the condition.

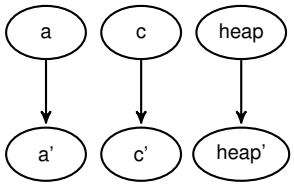

Figure 4.12: Identity graph

To have all the necessary edges in the if condition graph we need to start the analysis with the identity graph. To create this graph the auxiliary function

`identityGraph(S)` is used, which takes a set of variables `S` and returns the identity graph. This is the graph that has an edge between the two vertices representing the value before and after a statement, as shown in Figure 4.12.

4.3.4 While Statement

Computing one iteration or no iteration of the while loop is essentially the same as computing the if statement with the same condition, the body of the while loop as the if block, and an empty else block. If the body is never executed all the edges should stay the same. If the body is executed once the edges should be added accordingly. The next iteration is conceptually just a sequential composition of the first iteration. Therefore, we can now merge the two graphs each representing one iteration of the loop body and get the graph of the while loop after two iterations. In general, the graph for one iteration can be merged with the graph computed for the last iteration to compute the graph after an additional iteration. This graph can only grow and because there are a final amount of variables in scope and hence a final amount of edges, the graph will reach a fixed point at which no more edges are added to the graph. Since it is not clear how many iterations of the body will be executed, we overapproximate and use the fixed point graph as the flow analysis graph for the while loop.

Example

In the example in Figure 4.13, we see that in the second iteration only a single edge is added, namely the edge between the variable `a` and `d` which is marked red. This is the only edge that can be found by searching for paths when merging the graph after one iteration with itself to simulate two iterations. The graph of the second and third iteration stays the same and therefore we have found a fixed point and can stop the analysis of the while statement.

```
1 method m(a:T,c:T)
2 returns (d:T,e:T)
3 {
4     while(c) {
5         d:=e
6         e:=a
7     }
8 }
```

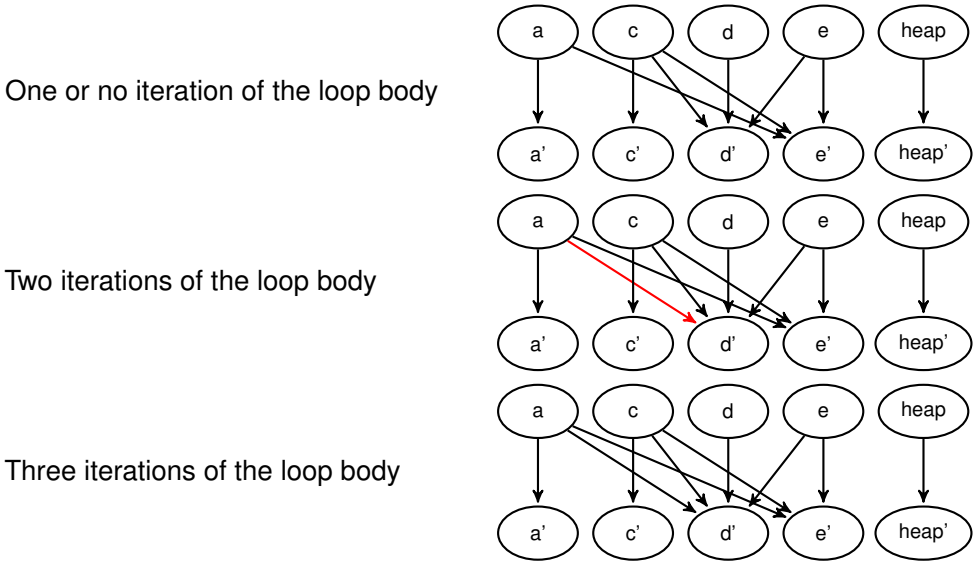

Figure 4.13: While statement example

Rule `computeFlowGraph(S,while(cond) {body})`

```
1 G' = identityGraph(S)
2 G1 = computeFlowGraph(S, if(cond) {body} else {})
3 G' = mergeGraphs(G1, G')
4 G = copy(G1)
→ 5 while(G' != G) {
6     G = copy(G')
7     G' = mergeGraphs(G',G1)
8 }
```

Figure 4.14: While statement rule

As in the if statement, this analysis starts with the graph containing only the identity edges that we create using the auxiliary function `identityGraph(S)` as discussed in Section 4.3.3. Here, `G1` is the graph after one iteration of the while loop. Therefore, we iterate over the graph `G1` until we find the fixed point of this graph. In our rule, `G` represents the graph of the last iteration and `G'` represents the graph of the most recent iteration. To compute a new iteration we can merge the graph representing our most recent iteration `G'` and the graph representing one iteration of the graph `G1` using the function `mergeGraphs` as explained in

Section 4.3.2. The function `copy(G)` takes a graph `G` and returns a copy such that we can save the graph of the last iteration.

4.3.5 Inhale/Assume

`Inhale` and `assume` result in the same graph. Every variable that is inhaled or assumed could potentially influence each variable that is in scope, because if the assume or inhale is false we do not know what code is executed and therefore each variable's value in the code block might be different based on what was assumed or inhaled.

Example

```
1  method m(b:Bool)
2  returns (d:Int, e:Int)
3  {
4      var b': Bool
5      if(b') {
6          assume b
7          d := 1
8      } else {
9          assume !b
10         d := 2
11     }
12 }
```

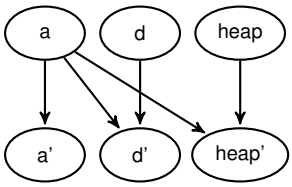

Figure 4.15: Assume statement example

This example can be encoded as an if statement and therefore we can show that our flow analysis graph is simply an overapproximation of the computation of the flow analysis graph for the if statement as shown in Section 4.3.3. Since variable `b'` has no assigned value we cannot know which branch is executed. However, because the assume statement on line 6 assumes `b` and the assume statement on line 9 assumes `!b`, we can rewrite this code example in Figure 4.15 the following way.

```
1  method m(b:Bool)
2  returns (d:Int, e:Int)
3  {
4      if(b) {
5          d := 1
6      } else {
7          d := 2
8      }
9  }
```

Figure 4.16: Assume statements translated to if statement

Therefore, `b` must influence `d`. However, since we do not know the rest of the

code when analysing the assume or inhale statement, we cannot make such transformations and therefore need to overapproximate the resulting graph. Any variable present in scope can be assigned to and therefore we need to add an edge in the graph from every variable in the assume or inhale statement to every other variable in scope.

Rule `g(S, inhale(exp))` or `g(S, assume(exp))`

```
    1  G := identityGraph(S)
→   2  G.addEdges(exp,S)
```

Figure 4.17: Inhale and assume statement rule

Here, the initial flow analysis graph `G` is the identity graph created using the auxiliary function `identityGraph` as explained in Section 4.3.3. Using the function `addEdges`, as explained in Section 4.3.1, the edges are added from every variable in the expression `exp` to every variable currently in scope represented by `S`.

4.3.6 Field Assignment

A field assignment is an assignment of some expression to a certain place in the heap. Since the heap is represented by a single variable, each right-hand side expression of an assignment influences the same variable representing the heap.

Example

```
1  field f:T
2  method m(a:T,b:Ref)
3  requires acc(b.f)
4  {
5      b.f:=a
6  }
```

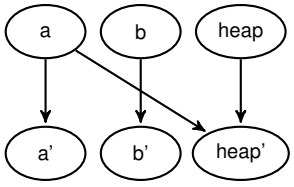

Figure 4.18: Field assignment example

This method requires access to the field `b.f` such that we can write to this field. In line 5, we assign `a` to this field and therefore assign a new value to this location in the heap. Thus the edge from `a` to `heap'` is added. However, it is important to note that different to the local variable assignment, the identity edge of the variable `heap` is still in the graph because the heap always might be influenced by itself.

Rule `g(S,x.f:=rhs)`

```
→   1  G := identityGraph(S)
    2  G.addEdges(rhs,{heap})
```

Figure 4.19: Field assignment rule

`G` represents the identity graph, created using the function `identityGraph`, as seen in Section 4.3.3. Since we defined one variable to represent the whole heap, we can add an edge from every variable in `rhs` to `heap`, using the function `addEdges`, as defined in Section 4.3.1.

4.3.7 Method Call

We can now use the newly added annotation to create the graph for a method call in a modular way. Unlike before, in the set-based approach, we can now simply create the graph described by the annotations of the called method. Additionally, we need to consider that the variables contained in the expression that is passed to the method influences the arguments of the method itself. The same holds for the return variables of the method. These return variables influence the variables that this method is assigned to. Hence, we will need to create two additional graphs to represent these influenced by relations.

Example

```
 1  method m(a1:T,a2:T)
 2  returns (r1:T)
 3  influenced r1 by {a1}
 4  influenced heap by {heap}
 5  {
 6      r1:=a1
 7  }

 9  method m1(a:T,b:T)
10  returns (d:T)
11  {
12      d:=m(a,b)
13  }
```

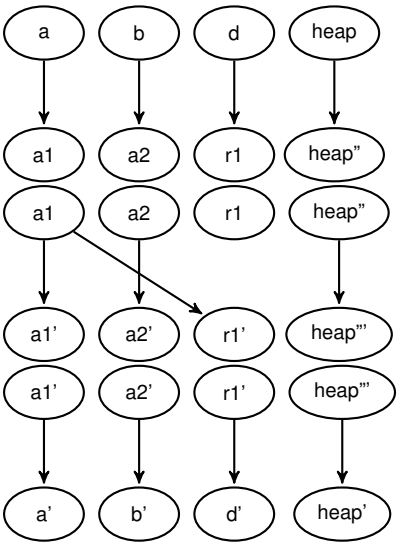

Figure 4.20: Method call example

In Figure 4.20, we create the graph that adds an edge between the variables passed to the method and the method arguments as well as a graph that shows which return variables of the method influence which variables that the method is assigned to. Then we compute the graph of the flow annotation as explained in Section 4.2. Now we can simply merge these graphs and get the final graph for the method call statement, as shown in Figure 4.21.

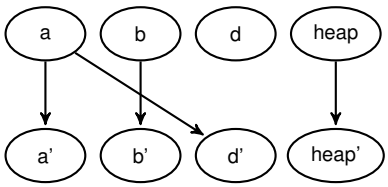

Figure 4.21: Merged graph for the method call example

Rule `g(S, t⃗ :=m(a⃗))`

```
    1  G := emptyGraph(S)

    3  G1 := emptyGraph(S)
    4  G1 := G1.addEdges'(a, m.arguments)

    6  G2 := emptyGraph(S)
→   7  G2 := G2.addEdges'(m.return_vars,ts')

    9  for (flow_annotation(t,args) in m) {
   10      G.addEdges(args,{t})
   11  }
   12  mergeGraphs(mergeGraphs(G1,G),G2)
```

Figure 4.22: Method call rule

Here `G` is the empty graph created using the auxiliary function, as seen in Section 4.3.1. $\overrightarrow{t}$ and $\overrightarrow{a}$ represent a set of target variables `ts` and variables `a` passed to method `m`. To compute the graph `G1` containing edges from all variables `a` passed to the method, to arguments of the method `m.arguments`, the auxiliary function `addEdges'(S1,S2)` is used. This function is similar to the function `addEdges` shown in Section 4.3.1. However, it is specified whether the vertices at either end of the edge represent the initial value or the final value of the variable. Similarly, we compute the graph `G2` containing the edges from all the return variables of the method `m.return_vars` to the variables `ts` that the method is assigned to, using the function `addEdges'`. To compute the graph of the annotations, all flow annotations of the called method `m` are checked. For each flow annotation there is the return variable `t` and the method arguments `args` that influence variable `t`. For each flow annotation all edges from the vertices in the argument set `args` to the return variable `t` are added, using the function `addEdges` as explained in Section 4.3.1.

The same analysis described for the method call is also used for `oldCall` statement, which is part of the newly added language features for lemmas in Viper.

4.3.8 Universal Introduction Statement

For the universal introduction statement, we simply compute the flow analysis graph for the body of the universal introduction statement. If any of the side-conditions are not met for the universal introduction statement an error will be thrown in another part of the verification process.

4.3.9 Exhale

If the exhale statement is pure, no edges need to be added to the flow analysis graph. However, if we have an exhale statement that is impure, it is possible that some access rights to the heap have been removed and therefore the heap is modified. Hence, we need to add an edge between every variable in the exhale statement and the heap.

The assert statement is conceptually the same as a pure exhale statement and therefore the graph does not have to be changed and we can simply return the identity graph.

Chapter 5

Evaluation

The modular flow analysis explained in the previous chapter heavily relies on graph computations. However, graphs are more computationally expensive than sets, which we use in our first approach. In this chapter, we evaluate our graph-based approach. In Section 5.1, we qualitatively compare the results from the set- and graph-based approaches. In Section 5.2, we evaluate the performance of our graph-based approach. The machine on which this code was timed has an Intel i7-8565U CPU processor, 16GB RAM and a Microsoft Windows 11 operating system.

5.1 Set vs. Graph-based Information Flow Analysis

We test the graph-based approach against a set-based approach to compare the two outputs. Ideally both approaches should result in the same information flow. The information flow was computed on test cases only including the implemented statements for sets. Hence, we consider Viper methods consisting of variable assignments, if and while statements, and sequential compositions. The remaining statements were not implemented because of the limitations of this approach, see Section 3.2. For each method, we compute the information flow using both approaches and compare their results. For the set-based approach, we execute the information flow analysis multiple times for each method. In particular, we mark a different input parameter as tainted in each iteration and store the resulting set of tainted output parameters. For the graph-based approach, we execute the information flow analysis with the method input and output parameters as the vertices. Afterwards, we compare the information flow for each variable. For each variable in the graph we gather the outgoing edges and store their target variables in a set. We can now simply compare the stored tainted sets from the set-based approach and the target variable sets from the graph-based approach for each input parameter. For all test cases, we have not identified any differences.

We have also measured the execution times for the set- and graph-based ap-

proaches. However, all considered test cases are fairly small and thus the measured execution times for each test file are aroung 500 ms. The execution times for the graph and set computation were up to 7 ms and the standard deviation for all test files' complete execution time was between 10 and 20 ms. Deducing meaningful conclusions is difficult because the variance of the underlying java virtual machine or operating system has potentially a big impact on these numbers.

5.2 Modular Flow Analysis Runtime

Due to the restricted set of supported statements in the set-based approach, we only consider small and hand-written Viper programs in Section 5.1. To obtain meaningful performance metrics for the graph-based approach, we separately evaluate the graph-based approach on large Viper programs generated by Gobra [2].

We measure the overall time to verify a given Viper program and the time spent for performing the modular flow analysis. However, the modular flow analysis is normally only performed for methods that are annotated with flow annotations. The large Viper programs generated by Gobra do not contain any flow annotation. Thus, we have modified the Viper plugin to execute the modular flow analysis for all methods. We repeat each test case 10 times. Table 5.1 lists the average execution time after discarding the slowest and fastest run of the overall verification and the modular flow analysis. The last column lists the modular flow analysis' impact on the overall verification time in percentage.

Our test cases are based on the artifact of Arquint et al. [7]. The first two test cases correspond to the initiator and responder implementations, respectively, of the Needham Schroeder Lowe (NSL) protocol. The latter two test cases correspond to the labeling and tracemanager package from the reusable verification library. Due to limitations of our implementation, we have manually transformed the generated Viper program to equivalent ones without goto and new statements.

Test case	**Lines of code**	**Verification time [s]**	**Flow analysis time [s]**	**Impact on verification time [%]**
initiator.gobra.vpr	18'214	74.859	0.84	1.1
responder.gobra.vpr	18'379	81.235	0.901	4.3
labeling.gobra.vpr	16'432	17.212	0.736	1.1
tracemanager.gobra.vpr	15'558	40.344	0.635	1.6

Table 5.1: Comparison of the modular information flow's execution time to the overall verification time for large Viper programs generated by Gobra

The table shows, that even for large Viper programs, the overhead introduced to perform the modular information flow analysis is negligible compared to the overall verification time. Our current implementation has not been optimized so far. Thus, we believe the overhead could be lowered even further by summarizing the effects to the graph of particular statements instead of computing many intermediate graphs and merging them, as we currently do. It is also important to mention that we forced the plugin to execute the information flow analysis on these test cases. In practice, the analysis will only be performed on methods that have at least one flow annotation.

Chapter 6

Conclusion

In this thesis, we extended the Viper language by three reasoning features, namely lemmas, existential elimination, and universal introduction, in a Viper plugin. These three reasoning features make reasoning in Viper easier and less error prone. In particular, all necessary side-conditions are checked such that users do not need to worry about them.

Firstly, lemmas can now be annotated as such, which induces proof obligations that their proofs terminate and do not modify the heap. Our plugin reports an error if these obligations are not met. Aside from these side conditions, lemmas also open up new possibilities, particularly we allow lemma calls in old contexts, since they do not modify the heap. Due to the same reason, lemmas allow us to encode lemma calls in a more efficient way compared to method calls because asserting and assuming their pre- and postconditions, respectively, is sufficient.

Secondly, we have added existential elimination. Compared to a manual encoding our new language feature removes code duplication, as the quantified property only has to be written once, and is less error prone, in particular when a code base evolves.

The final reasoning feature added is universal introduction. One of the side-conditions for universal introduction is, that the proof code inside the statement does not modify the execution of the rest of the program. Hence, we track which variables are influenced by the quantified variable as well as whether the heap is influenced by the quantified variable. We present a modular information flow analysis that is of independent interest and demonstrate how we apply this analysis to the particular use case of universal introduction.

In a first attempt the modular information flow analysis was done using a set-based approach. The new `tainted` keyword introduced to analyse method calls makes this approach non-modular. Therefore in summary, a method's annotation has to be permissive enough for all calls of this method. Hence, we introduce flow annotations in the method specification to express whether a given output

parameter is influenced by the heap and which input parameters it is influenced by. Thus, these flow annotations describe a graph of the information flow from method inputs to outputs. Throughout the analysis of the method body we use the same graph representation to check whether the flow annotations provided by the user are correct. Even though the modular flow analysis was initially developed for the purpose of verifying the universal introduction statement, it can be applied to other use cases.

6.1 Future Work

In this section, we elaborate what can be done to extend and improve the newly implemented features. Although our modular information flow analysis supports most statements in Viper, we want to extend the support to goto and new statements as well as to the new language feature for existential elimination as future work.

Furthermore, this thesis focuses on reasoning in Viper. However, Viper front-ends would benefit likewise from these reasoning features. As future work, we would like to explore to what extend, for example universal introduction in Gobra can directly be encoded to the corresponding counterpart in Viper or whether additional checks in the front-end are necessary.

Additionally, future work can look at the combination of the reasoning features introduced in this thesis and offer new syntax for common combinations. For example, one might want to prove a universal introduction using a lemma. Such a statement might look as follows for some function `P`, label `l`, and lemma `lemma`.

```
1 prove forall k:T {P(k)} assuming P(k)
2                         implies oldCall[l]lemma(k) {...}
```

Figure 6.1: New syntax for a lemma call acting as the proof of a universal introduction

This is not possible at the moment since we can only assume and imply expressions and not statements or lemmas.

Although our evaluation confirms that our modular information flow analysis adds a minimal overhead to verifying a program, the computation of flow analysis graphs could be optimized. Currently, a new graph is computed for every statement and then merged with the graph of the consecutive statement.

Bibliography

[1] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[2] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 367–379, Cham, 2021. Springer International Publishing.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[4] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

[5] Marco Eilers and Peter Müller. Nagini: A static verifier for Python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[6] Fabio Streun. Tool support for termination proofs. Bachelor's thesis, ETH Zurich, Zurich, 2019.

[7] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. A generic methodology for the modular verification of security protocol implementations. Technical Report 2212.02626, arXiv, 2022.

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):

Advanced Logical Proofs in a Verifier

Authored by (in block letters):

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

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

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 20.03.2023	D. Weiersmüller

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