Practical Inlining in Viper

Bachelor's Thesis Project Description

Matthias Schenk
Supervisors: Prof. Peter Müller, Thibault Dardinier

May 2022

1 Introduction

Viper [1] is an intermediate verification language and a suite of verification tools for this language. It facilitates the development of automatic verifiers on top of it for different programming languages like Go (Gobra [2]), Python (Nagini [3]), and more. One reason Viper is a powerful verification infrastructure is that it uses modular verification. Modular verification means that method or loop bodies are verified in isolation with respect to their annotations[1]. The main method does not have any information about the method body of a callee or a loop body. For method calls, only the preconditions, postconditions and method arguments are visible. For loops, only the invariants and the guards are visible[2]. During the verification of the caller, it is assumed that all called methods and loops verify with their defined annotations. These annotations are then used to restrict the range of possible executions.
Viper builds on separation logic [4], a logic that extends on the pre- and postconditions of Hoare logic [5] to reason about heap-manipulating programs. A state in Viper is a triplet consisting out of the store, the heap, and the permission mask. The store contains all information on the local variables, the heap contains the values of the fields of references in the program, and the permission mask is a set of mappings from heap locations to permission fractions. A permission fraction p is a rational between 0 and 1 that defines the access rights to heap locations in an execution environment. The fine-grained control of fractional permissions to heap locations is possible because of separation logic. Fractional permissions create three cases in Viper:

1) **p=0** means that there are not enough permissions to read from or write to a heap location.
2) **p∈(0,1)** means that there are enough permissions to read from a heap location but not enough to write to it.
3) **p=1** means that there is full permission and reading from or writing to a heap location is possible.

Fractional permissions are especially useful for reasoning about concurrent heap-manipulating programs since permissions can easily be divided and distributed among threads, and a thread can only write to a heap location if and only if it holds full permission to the desired location. This also implies that no other thread can read from that location until the full permission is released from the writing thread again.
Overall, automatic verification gives developers more confidence in their code compared to

[1]The term annotations refers to the pre- and postconditions for methods and to invariants for loops.
[2]However, the main method identifies which variables are modified (loop targets) and which are not.

traditional testing methods. It is the responsibility of the developer to provide annotations and guide the verification process. Annotations are directed at the three aspects of a state in Viper and hence contain permission specifications to the heap, constraints on heap values, and information about the store of local variables. Defining these annotations can still be time-consuming since they need to be defined for every method and loop in a program and might need to be adapted whenever changes to the code are made. The more complex the desired properties are the more time will have to be invested into defining the required annotations. Especially in these cases, one would like to have assurances that the code functions properly before time is spent on annotations.
Therefore, the goal of this thesis is to introduce a new inlining feature to Viper that can detect errors in the code before annotations are defined. Inlining method calls, or unrolling loops cannot replace the error-detection of a fully annotated program. It is not possible to inline a recursive method, or to unroll a loop, an infinite number of times. Inlining will only ever be able to give a verification guarantee up to a certain depth.
Nonetheless, inlining is very useful for early error detection. The program in Listing 1 contains a division-by-zero error. Without any annotations, it will throw the error that the method might not have enough permissions to *x.f*. The permission error is resolved by adding the invariant in line 8. This results in two new errors for potential division by zero in line 11 and 12. The error in line 11 is not a real division-by-zero error, since *i=0* at the beginning of the loop and *i* monotonically increases. The true error in line 12 is determined only after adding the second invariant in line 9. This is a *fundamental error*. A fundamental error is an error for which there exist no annotations such that the program will verify. These findings can be compared with the unrolled version of the program in Listing 2. The verifier can find the fundamental error without any annotations, and reasoning about the behavior of *i*. Additionally, the developer receives feedback in which iteration the error occurs.

Listing 1: Error Detection in Loops

```
0  field f: Rational
1
2  method example(x: Ref, n: Int)
3      requires acc(x.f)
4      ensures true
5  {
6      var i: Int := 0
7      while (i < n)
8          //invariant acc(x.f)
9          //invariant i >= 0
10     {
11         x.f := x.f + 1 / (i + 1)
12         x.f := x.f - 1 / (i - 1)
13         i := i + 1
14     }
15 }
```

Listing 2: Error Detection in Loops (unrolled)

```
field f: Rational

method example(x: Ref, n: Int)
    requires acc(x.f)
    ensures true
{
    var i: Int := 0
    if (i < n) {
       x.f := x.f + 1 / (i + 1)
       x.f := x.f − 1 / (i − 1)
       i := i + 1
       if (i < n) {
         x.f := x.f + 1 / (i + 1)
         x.f := x.f - 1 / (i - 1)
         i := i + 1
         if (i >= n)
         { assume false }
       }
    }
}
```

This example shows the usefulness of inlining as a new feature of Viper. Not only does a preliminary verification through inlining save time in writing annotations, but it also makes debugging easier by narrowing down the source of the failed verification. It answers the question of whether there is a mistake in the code, or if the annotations are not sufficient.

2 False Positives

The initial example has shown that inlining can be used to find fundamental errors in programs. Finding such errors is not always this straightforward with inlining. There are statements and annotations that can create verification errors that in the fully-annotated, non-inlined program would not be an issue. We define verification errors in the inlined program that do not correspond to fundamental errors as *false positives*. Such false positives are side-effects of using separation logic and permission resources.
Listings 3 and 4 show an example of a false positive due to permission introspection. The function *perm()* returns the permission fractions currently held for a reference. The program wants to assert that there are no longer enough permission fractions to read from *x.f* after the callee method call. The callee method acquires the permission fractions with the precondition from the caller, but it does not return the permission fractions back to the caller in the postcondition. This is called a permission leak and can, for example, be intentionally used to prevent any further writes or even reads to a reference after a method call. The statement in the callee's body will always verify, and modular verification will also verify Listing 3 with no errors. On the other hand, the inlined program has no longer a permission leak causing the assertion in line 8 of Listing 4 to fail. This implies that it is not a fundamental error. In other words, there exist annotations with which the program verifies and are thus confronted with a false positive.

Listing 3: False Positive

```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f, 1/2)
4      ensures true
5  {
6      callee(x)
7      assert perm(x.f) == 0/1
8  }
9
10 method callee(x:Ref)
11     requires acc(x.f, 1/2)
12     ensures true
13 {
14     assert true
15 }
```

Listing 4: False Positive (unrolled)

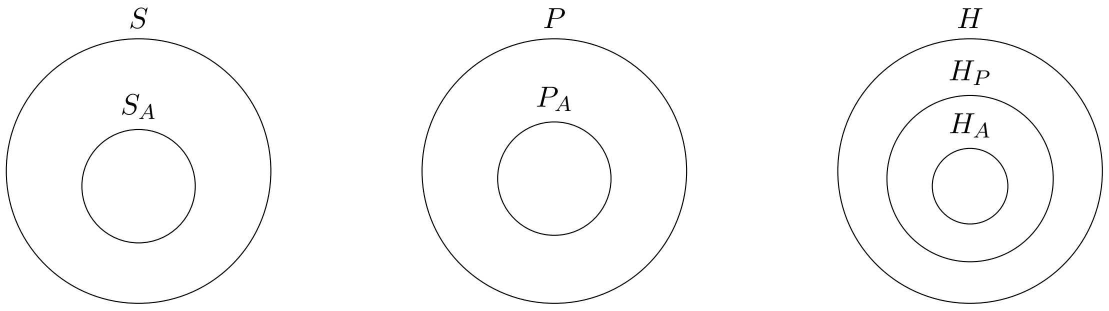
```
0  field f: Int
1
2  method caller(x: Ref)
3      requires acc(x.f, 1/2)
4      ensures true
5  {
6      assert true
7      assert perm(x.f) == 0/1
8  }
```

Having an understanding if inlining may cause false positives is important. There exist verification-preserving conditions for statements which imply that a statement does not cause any false positives when inlined [6]. These conditions are an active research topic, and there exists an extension of Viper that can check if they hold automatically using Boogie.

3 Differential Inlining

Verification-preserving guarantees are especially useful when inlining methods without annotations. However, developers often have some idea about the annotations required for the

intended functioning of their program. Rudimentary permission specifications, like *read and write*, *read only*, or intended permission leaks, can be of great help guiding the verification. An inlining tool should therefore also support partial annotations. T. Dardinier proposes the notion of barriers that consist of a combination of filters for the store, the heap, and the permission mask. Barriers can filter resources and frame them around the inlined body to create states that are dependent on the partial annotations of the callee [6]. Annotations for methods and loops can at most transfer all resources of the main method to the annotated body and in doing so replicate the state of the main method. Thus, barriers cannot create stronger states and only filter resources to create weaker states. This expresses itself through less precise information on local variables, less precise information on the values in the heap, or less permissions. Viper itself does not provide enough control over the state of the program to implement barriers. Barriers will ultimately be enforced in Boogie.
The diagrams below show potential levels to categorize the filters of the barriers. S represents all information in the store of the main method, P represents the whole permission mask of the main method, and H represents the whole heap of the main method. The subscripts signify additional filter properties. The most important subscript is A, which stands for the filter introduced by the annotations. However, it is possible that filters from annotations do not create weaker states but instead create states such that the same resources are available inside and outside the barrier, i.e., $S = S_A$.



Store: There are slight differences in the store that are passed on for modular verification of methods and of loops. For methods, only variables that are specified in the method arguments are passed on. For loops, the whole store needs to be visible to the loop body, but only local variables that are modified by the loop body require invariants. The store values for loop targets are havoced (are assigned arbitrary values) and Viper will only consider the corresponding annotations for the verification.

Permission Mask: Writing permission specifications is usually the first step when writing annotations. There are only two levels of filters for permissions. Either everything passes through the filter or only the annotations are considered. If the inlined program verifies with P but not with P_A, then there exist enough permission resources in the main method, which could be transferred to the callee such that the program verifies.

Heap: As already stated H entails all heap information from the caller. The next level H_P filters out any information about values of heap locations a method or loop does not have enough permissions to. Lastly, the filter H_A considers only annotations about the heap. Consider the annotation $A = acc(x.f) * x.f >= 0$ and let $x.f := 5$ and $y.f := 3$ in the state of the main method. The filter H_P will know that $x.f := 5$, since A contains access to $x.f$, but H_P will filter the value of $y.f$. The difference between H_P and H_A is that H_A will only know the information that $x.f >= 0$ but not the precise value of $x.f$, which is 5. Meaning, the filter H_P has greater precision.

The essence of differential inlining is to verify the inlined program with different combinations of barriers and narrow down the source of errors through the verification results. The levels outlined above result in 12 different filter combinations. Two of these combinations have already been discussed. $\{S, P, H\}$, called a no-scope barrier, does not filter any information, and corresponds to the inlining from the examples above. $\{S_A, P_A, H_A\}$ is a barrier that is equivalent to modular verification, since it only considers the annotations.
Below is a short comparison between different inlining barriers to demonstrate the intuition of differential inlining and see what information can be gained by comparing verification results. Table 1 will serve as a guideline for the hypothetical analysis, and the permission mask is ignored to focus on the store and heap information. Each row represents an inlined program which was verified with the different barriers from the first row. A checkmark indicates that the verification was successful. A "x" indicates that the verification failed.

	S and H	S_A and H	S and H_P	S_A and H_P
1)	✓	✓	✓	✓
2)	x	x	x	x
3)	✓	x	✓	x
4)	✓	✓	✓	x
5)	✓	x	x	x

Table 1: Barrier Comparison.

In 1), the inlined program verified for all barriers. This means that the annotations are sufficient. In 2), the code is faulty and needs to be revised before annotations are considered. Starting with 3), the information about the program becomes more specific. Verification is possible since the verification succeeds when the callee has access to all resources of the caller. In 3), the program does not verify with the annotations for the store. The attention needs to be focused on the store annotations. 4) offers more flexibility to the developer. The program can successfully be verified by either changing the annotations for both the store and the heap, only the store, or only the heap. And finally in 5), both heap and store annotations are not strong enough, and both need to be changed. But, as already stated before, the program verifies with a no-scope barrier and hence does not contain a fundamental error.

Listing 5: Differential Inlining

```
field f: Int

method caller(x: Ref)
    requires acc(x.f) * x.f >= 1
    ensures true
{
    var i:Int := 1
    while (i < 2)
        invariant acc(x.f) * i >= 0 * x.f >= 0
    {
        assert x.f + i >= 1
        i:= i + 1
    }
}
```

Listing 5 shows a simple example to make clearer how the filters for different barriers look like. There are some annotations in the callee method, and the focus will again remain with the store and the heap for simplicity. The following overview shows the information contained in the Viper state after the filters of the barrier were applied.

$$\mathbf{S}\text{: i = 1} \qquad \mathbf{S_A}\text{: i >= 0} \qquad \mathbf{H}\text{: x.f >= 1} \qquad \mathbf{H_A}\text{: x.f >= 0}$$

This is the same scenario as in 4) of Table 1. The annotations for both the heap and the store are not precise enough such that the verifier succeeds with the verification, even though there is sufficient information outside of the loop. Strengthening either or both annotations for heap and store to reflect the information outside the loop would result in a successful verification.

4 Core Goals

4.1 Making inlining a real feature

There is an existing prototype of the inlining module that was developed as a proof of concept. The main goal of this thesis is to improve, rework, and clean up the code of the inlining module for Carbon. The current inlining module was developed with an older Carbon version. One aspect will be to update the code so that the module can be integrated into the current Viper version again. After the code has been updated, and the implementation has become cleaner, the documentation will be expanded on so that maintenance and further development becomes easier.
From the user side, it is important that the error reporting becomes reliable and relays comprehensive information. The verification-preserving conditions will play a key role for error reporting and warning messages. They will need to be studied in detail to be able to distinguish if inlining might cause a false positive. The verification-preservation and comprehensiveness of the feedback given to the user will be a deciding factor in the adoption of the feature.

4.2 Differential inlining

Another core goal is to explore differential inlining and to find a way to implement the required barriers. At time of writing, differential inlining in Viper is only conceptual and has no concrete implementation yet. The support of testing code with partial annotations through inlining will make the tool much more flexible and will open the possibility of providing feedback that combines the verification results of barriers with different filters.

4.3 Evaluation

The evaluation will primarily consist of two aspects. The first one will be the testing of the effectiveness and correctness of error-detection with the inlining tool. The second one will be the testing of differential inlining. For this purpose, the correctness of the barriers needs to be tested, and in a second step, the effectiveness of different barrier combinations in gaining information about the strength and completeness of our annotations can be explored.

5 Extension Goals

5.1 Performance and stratified inlining

Verification can computationally be very intensive. A recursive method that spawns multiple children will have exponential growth when inlined. Increasing the performance might allow to verify up to a greater depth. Another, more sophisticated approach, would be to use stratified inlining, which determines the more interesting execution paths of the program and selectively explores them in greater detail [7]. Both approaches increase the effectiveness of finding errors, and the more precise information, will increase confidence in the tool.

5.2 Extend verification-preserving conditions

The verification-preserving conditions mentioned before are useful to guarantee the absence of false positives when inlining. Nonetheless, there are still problems with false positives. There exist programs that cannot produce false positives, and yet the verification-preserving conditions are too coarse to capture these programs. Therefore, an extension of this thesis would be to extend the verification-preserving conditions such that more programs could be captured, or more statements could be supported. One such extension might be the exploration of wildcards for permissions.

5.3 Extend Differential Inlining

Extensions to differential inlining might be to formally define the algorithm and improve its performance. There may be potential gains by doing a dichotomic search, defining additional filter levels, or using parallelism.

References

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

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

[3] 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.

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

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[6] Thibault Dardinier. Beyond the frame rule: Static inlining in separation logic. Master's thesis, ETH Zurich, Zurich, 2020.

[7] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *Computer-Aided Verification (CAV)*, July 2012.