# Beyond the Frame Rule: Static Inlining in Separation Logic 

## Master Thesis

## Author(s):

Dardinier, Thibault

## Publication date: <br> 2020

## Permanent link:

https://doi.org/10.3929/ethz-b-000417746

## Rights / license:

In Copyright - Non-Commercial Use Permitted

## ETH

Eidgenössische Technische Hochschule Zürich

# Beyond the Frame Rule: Static Inlining in Separation Logic 

Master's Thesis<br>Thibault Dardinier

April 2020

Advisors: Gaurav Parthasarathy, Professor Peter Müller
Programming Methodology Group Department of Computer Science, ETH Zürich

## Abstract

Various formal verification techniques can be used to automatically verify the absence of errors in programs. This provides an advantage over testing approaches, namely the guarantee that a program is correct for any possible execution. However, such approaches often require a user to provide additional specifications to guide the verification, in the form of loop invariants and method preconditions and postconditions, which places a burden on the user. When no specifications are provided, verifiers usually report potential errors which are not actual errors, hence lowering confidence in error reporting.

Users might want to learn quickly (without providing too many specifications) and with high confidence whether a program is incorrect. This would speed up the development and verification process by only having to provide specifications when one is fairly certain that the program is correct. Static inlining, that is inlining of method calls and unrolling of loop iterations, is an interesting approach to tackle this issue. Using approaches based on static inlining, verifiers could inform a user of the existence of fundamental errors, errors for which no annotation can make the program verify. The existence of fundamental errors indicates an error in the program itself, informing the user this program cannot be verified, without the user needing to waste time and energy in the search of the right annotation.
One would expect that errors reported in an inlined program always correspond to fundamental errors in the original program, since annotations only serve as approximations of method calls and loops. Surprisingly, this is not always the case. Indeed, Viper, a verification infrastructure for permissionbased reasoning, partly based on separation logic and implicit dynamic frames with fractional permissions, has special features (such as permission introspection) which give rise to examples where this is not the case. These examples have all in common that they do not satisfy the frame rule.

In this thesis, we find (and prove correct) a soundness condition which
characterizes the set of Viper programs for which static inlining is sound, that is errors in the inlined program correspond to fundamental errors in the original program. We define a parametric language which generalizes Viper, where program states are elements of a separation algebra, define the soundness condition in terms of this language, and prove the soundness property of inlining under this soundness condition, using the proof assistant Isabelle/HOL. We then show how one can instantiate this parametric language to transfer the results to Viper. We also explore a completeness property of static inlining, and consider extensions to different loop semantics and different ways of inlining.

## Acknowledgments

Most of all, I would like to thank Gaurav Parthasarathy for his dedication and his enthusiasm towards this project. Thanks for the many formal (and informal) meetings we had, the passionate discussions, and the constant feedback to keep me on the right track while letting me explore random directions.

I would also like to thank Professor Peter Müller for the opportunity to work on such an interesting topic, and for convincing me that this project was really interesting and challenging at a time when I could not picture what this project was really about.

Finally, I would like to thank my family, flatmates, friends and girlfriend (in alphabetical order) who, on top of being important in my life, helped me find a coherent story to this thesis with their confused faces while I was trying to explain what I was working on.

## Contents

Abstract ..... i
Acknowledgments ..... iii
Contents ..... v
1 Introduction ..... 1
1.1 Early detection of fundamental errors in Viper ..... 1
1.1.1 Formal verification and the burden of annotating ..... 1
1.1.2 Towards a useful feature: Static inlining to the rescue . ..... 3
1.1.3 Permission introspection in Viper: A fly in the ointment ..... 5
1.2 Problem statement: The meaning of inlining ..... 6
1.3 Overview of the thesis ..... 8
2 Background ..... 11
2.1 Separation logic and fractional permissions ..... 11
2.2 Viper ..... 13
2.2.1 Permissions and assertions ..... 13
2.2.2 Viper state model ..... 13
2.2.3 Building blocks of loops and method calls verification ..... 14
2.2.4 Black-box semantics of loops in Viper ..... 15
2.2.5 Semantics of method calls in Viper ..... 16
2.2.6 Features which break the frame rule ..... 17
3 Unsoundness of Inlining in Viper ..... 21
3.1 Soundness property and soundness condition ..... 21
3.2 Examples of Viper programs where inlining is unsound ..... 22
3.2.1 SafeMono: Stronger states verify more ..... 22
3.2.2 MonoOut: Stonger state in, stronger states out ..... 24
3.2.3 Framing: Statements which respect the frame rule ..... 25
3.3 Framing and mono ..... 26
3.3.1 Inclusions and examples ..... 26
3.3.2 Compositionality ..... 27
4 Abstracting Viper States with a Separation Algebra ..... 31
4.1 A preliminary separation algebra ..... 32
4.2 Motivation: Modeling the behavior of inhale and exhale on annotations ..... 33
4.3 Definition of the separation algebra ..... 34
4.4 Related work ..... 36
5 A Parametric Language for Resource-Based Inlining ..... 39
5.1 The language ..... 40
5.1.1 Parameters of the model ..... 40
5.1.2 Statements ..... 42
5.1.3 Semantics ..... 43
5.2 Store of a state ..... 44
5.2.1 Store function ..... 44
5.2.2 Var and havoc ..... 45
5.3 Assertions ..... 46
5.3.1 From syntactic to semantic assertions ..... 47
5.3.2 Annotations are supported and intuitionistic ..... 47
5.3.3 Well-defined assertions: A mismatch between syntactic and semantic assertions ..... 49
5.3.4 Minimal satisfying set of an assertion ..... 51
5.4 While loops ..... 53
5.5 Renaming interface ..... 54
5.6 Method calls ..... 58
5.7 Rest of the semantics ..... 59
5.7.1 Control structures ..... 59
5.7.2 Custom interface ..... 59
5.8 Well-definedness and verification of a program ..... 60
6 Soundness of Static Inlining ..... 63
6.1 Mono and framing ..... 63
6.2 Formalization of soundness ..... 64
6.2.1 Inlining ..... 64
6.2.2 Soundness condition ..... 67
6.3 Soundness theorem ..... 70
6.4 Induction case: method calls ..... 72
6.5 Induction case: loops ..... 76
7 Completeness of Static Inlining ..... 83
7.1 Syntactic transformation: bounded program ..... 84
7.2 Examples of incompleteness in Viper ..... 86
7.2.1 Loops and recursive methods without index ..... 86
7.2.2 Permission gap ..... 88
7.3 Sketch of completeness ..... 90
7.3.1 Strongest postcondition assumption ..... 91
7.3.2 Constructing the annotation ..... 91
8 Instantiating the Parametric Language with Viper ..... 95
8.1 A simplified version of Viper ..... 95
8.1.1 Disallowing references to previous states ..... 96
8.1.2 Inhale-exhale assertions ..... 96
8.1.3 Types of variables ..... 96
8.1.4 Other features ..... 97
8.2 Separation algebra, variables and store ..... 97
8.3 Assertions ..... 99
8.3.1 From syntactic to semantic assertions ..... 99
8.3.2 Annotations are supported and intuitionistic ..... 100
8.4 Rename interface ..... 101
8.4.1 Renaming an element ..... 101
8.4.2 Inverting a renaming quadruple ..... 104
8.4.3 Towards the rename interface ..... 104
8.5 Custom statements ..... 105
8.6 Completeness: Strongest postcondition in Viper ..... 106
8.7 Towards two useful features in Viper ..... 109
8.7.1 Static inlining for early error detection ..... 109
8.7.2 Speed up re-verification through caching ..... 110
9 Extensions to Different Loop Semantics and Inlinings ..... 111
9.1 Exploration of loop semantics ..... 111
9.1.1 Semantics of loops in Viper: Silicon and Carbon ..... 112
9.1.2 Three coherent ways to treat loops ..... 113
9.2 Different inlinings: A classification of inlinings in Viper ..... 116
9.2.1 A chronology of annotating ..... 116
9.2.2 Barrier ..... 117
9.2.3 Scoping of a barrier ..... 118
10 Conclusion and Future Work ..... 121
10.1 Conclusion ..... 121
10.2 Future work ..... 122
10.2.1 Improve the current framework and soundness proof . ..... 123
10.2.2 Completeness of inlining ..... 123
10.2.3 Towards useful features in Viper ..... 124
List of Figures ..... 127
Listings ..... 129
Bibliography ..... 131
A Appendix: Isabelle Formalization ..... 135
A. 1 Theories presented in this appendix ..... 135
A. 2 Separation algebra ..... 137
A.2.1 Preliminary separation algebra ..... 137
A.2.2 Separation algebra ..... 138
A. 3 Semantics ..... 145
A.3.1 Abstract language ..... 145
A.3.2 Semantics ..... 154
A. 4 Soundness ..... 173
A.4.1 Method case ..... 178
A.4.2 Loop case ..... 183
A.4.3 Soundness proof ..... 183
A. 5 Renaming ..... 184

## Chapter 1

## Introduction

### 1.1 Early detection of fundamental errors in Viper

### 1.1.1 Formal verification and the burden of annotating

Various formal verification techniques can be used to automatically verify the absence of errors in programs. This provides an advantage over testing approaches, namely the guarantee that a program is correct for any possible execution. In practice, the program is tested against a specification written by the programmer. A common way of expressing a specificiation is via a contract, containing:

1. A precondition, namely a specification describing under which conditions the program should be executed.
2. A postcondition, namely a specification describing a condition which has to hold after the execution of the program, when executed under the conditions described by the precondition.

As an example, imagine a program whose aim is to find a path between two cities, named A and B. A precondition for this program could be "A and B are the names of two existing cities", and a postcondition could be "if a path between A and B exists, then the output is a valid path between A and B, otherwise the output is an empty path". If this annotated program verifies, this means that every time it is called with two cities for which there exists a solution, then a solution is found and returned by this program.

Ideally, formal verification should be fully automated. That is, starting with a program annotated with a precondition and a postcondition, it should automatically check whether the contract is satisfied by the program. However, this problem is generally undecidable. One source of undecidability is the existence of statically unbounded loops.

One solution to overcome this theoretical limitation is to expect the programmer to annotate loops with loop invariants. A loop invariant is also a contract: If the loop invariant holds before the execution of an iteration of the loop, it has to hold afterwards. Loops in the program can then be approximated solely in terms of this contract. If the invariant holds before the loop, then it holds after the loop. Given a loop invariant, checking whether it is actually an invariant for a loop is decidable. ${ }^{1}$ Moreover, checking whether the modified program (where loops are approximated by their contracts) satisfies its contract is also decidable. ${ }^{2}$ The verification of a program with annotated loops therefore becomes a decidable problem, but not yet a practical one.

In practice, verifying such a program has a high computational complexity, and can become an intractable problem. Another useful improvement to reduce this complexity is to use modular verification. The idea is to divide the program in smaller methods, each of them doing a small part of the overall job. The programmer has to then annotate these methods with contracts, consisting of a precondition and a postcondition. Method calls are approximated by their contracts, which makes verification tractable. Finally, verification succeeds when each method satisfies its contract under the assumption that all other methods satisfy their contract. In this case, the whole program is verified. ${ }^{3}$

The required additional specifications described above, loop invariants and method preconditions and postconditions, place a burden on programmers who want to formally verify their programs. Indeed, on top of the initial effort of writing a program and an initial contract for this program, programmers also need to annotate the program with these additional specifications. However, if the annotations are not strong enough to prove the desired property, then the verifiers will report potential errors which are not actual errors, but spurious errors. As a result, the programmer needs to often spend a lot of time adjusting the annotations until they are strong enough for the verifier to prove the program correct, assuming the program is correct.

If the program is incorrect (and hence the verifier always reports that there is a potential error), then the programmer cannot easily distinguish whether the verifier reports an error because the annotation is not strong enough or whether the program is incorrect. Hence the progammer may waste a lot of time first trying to strengthen the annotations until he or she figures out that the program is actually incorrect and then adjusts the program. It would therefore be useful for programmers to learn quickly and with high

[^0]confidence whether a program is incorrect, before diving into the process of providing these specifications. This would speed up the development and verification process by only having to provide specifications when one is fairly certain that the program is correct.

### 1.1.2 Towards a useful feature: Static inlining to the rescue

Stratified inlining [12] is an approach which has the potential to improve this situation by iteratively performing static inlining of methods and loops, that is inlining of method calls and unrolling of loop iterations up to a bounded depth, in order to detect errors. This approach has been successfully applied to the intermediate verification language Boogie [2] in the tool Corral [13].

However, this approach based on static inlining cannot in general be straightforwardly applied to other intermediate verification languages. The intermediate verification language Viper [15] is an example where the use of static inlining to detect incorrect programs is unclear, mainly because of Viper's powerful support for fractional permissions to heap locations. Let us first illustrate the idea of static inlining with Viper, in a case where there no are no issues.

A fractional permission, namely a rational number between 0 and 1 , is associated to a heap location (a pair of a reference $x$ and a field $f$ ) in a program state. At least some (that is non-zero) permission is required for a program to read a heap location, and full permission (1) is required to write a heap location. Viper has also a special kind of assertion, accessibility predicates, in the form of acc(x.f, p). This assertion represents a fractional permission of $p$ to the heap location $x . f . \operatorname{acc}(x . f)$ is a special form of it, where $p$ is 1 , that is full permission to $x$. $f$. Take the Viper program shown on Listing 1.1 as an example.

The situation is the following: The programmer has finished writing a program, with an initial contract, basically saying "If the method example is called with full permission to $x . f$, then no error happens". Without any loop invariant, one of Viper's verifier would report an error message on line 12 (underlined with a dotted red line), saying that there is not enough permission held by the program state to read and modify the value of $x . f$. This issue can be solved by annotating the loop with the loop invariant commented on line 9 , acc (x.f), since there is enough permission to $x . f$ held by the program state before the loop. Since it suffices to annotate the program to resolve this error, this is a not an actual error of the program, but a spurious error since the original annotation was too weak. In other words, this does not represent a "fundamental error". Let us define more precisely what we call a "fundamental error".

Listing 1.1: An example where an early error detection feature would be useful (original).

```
field f: Rational
method example(x: Ref, n: Int)
    requires acc(x.f)
    ensures true
{
    var y: Ref
    assume x != y
    var i: Int := 0
    while (i < n)
        // invariant acc(x.f)
        // invariant i >= 0
    {
        x.f := x.f + 1 / (i + 1)
```



```
        i := i + 1
    }
}
```

Definition 1.1 (Informal) A program contains a fundamental error if it is not possible to provide an annotation such that the program verifies.

Annotating the loop with the invariant on line 9 makes the permission error on line 12 disappear. Another error is then reported on line 12, with the error message "divisor $i+1$ might be zero". Indeed, since the division by 0 is not defined, dividing (potentially) by zero yields an error. As before, this does not correspond to a fundamental error: $i=0$ during the first iteration, thus $i+1=1 \neq 0$. Furthermore, $i$ is incremented at each iteration, thus $i+1$ will never be zero. Adding the invariant $i>=0$ commented on line 10 solves this issue.

The next error is reported on line 13 (underlined in red), another potential division by zero. However, this corresponds to a fundamental error. Indeed, $i-1$ will be zero during the second iteration. This is therefore an example of an error we want to report. As shown on Listings 1.2, unrolling the loop enables the verifier to easily detect this fundamental error. Moreover, the verifier can report a more precise error message, by reporting which iteration contains the error. The usefulness of this feature is a main motivation for this thesis. However, as shown below, understanding when the inlined program can be used to find fundamental errors is non-trivial in Viper, as opposed to Boogie.

Listing 1.2: An example where an early error detection feature would be useful (inlined).

```
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)
                {
                }
        }
    }
}
```


### 1.1.3 Permission introspection in Viper: A fly in the ointment

The previous subsections explain why a feature to detect fundamental errors early, based on static inlining, would be useful in the practical context of formal verification. Therefore, one important part of this thesis is to explore the following soundness property of static inlining:

Definition 1.2 (Informal) Static inlining is sound if errors in the inlined program correspond to fundamental errors in the original program.

This thesis focuses partly on Viper for two reasons. The first reason is that Viper is an intermediate language with many front-ends built on top of it, as illustrated in Figure 1.1. Therefore, working directly at the Viper level enables to leverage this work for many different languages. The relationship between static inlining at the front-end level and at the Viper level is however not clear, and may need further exploration.
The second and more important reason is the existence of Viper programs for which static inlining is unsound (see Chapter 3 for examples). All Viper programs for which static inlining is unsound have a common point: They contain statements which break the frame rule (see Chapter 2). Moreover,


Figure 1.1: An illustration Viper architecture (figure taken from [15]).
static inlining is always sound for Viper programs where all statements respect the frame rule. This is probably a reason why this topic has not been explored before, since verification languages usually respect the frame rule.

The frame rule does not always hold in Viper, because of its advanced support for permissions, which encode ownership of parts of the heap. Amongst others, Viper has a built-in feature called perm, which evaluates to the permission amount held to a field location by the program state. For example, $\operatorname{perm}(x . f)$ returns the amount of permission currently held to x.f. Permission introspection is a powerful feature which allows, for example, to encode a leak check for monitors (see [15], Section 2.2), or to branch on the permission amount held in the state. It is a powerful debugging tool. It is also a reason why it is possible to write a Viper program for which static inlining is not sound. We show in Chapter 3 three examples illustrating how permission introspection makes static inlining unsound in Viper programs.

### 1.2 Problem statement: The meaning of inlining

The core of this thesis is the exploration of the soundness property of static inlining, in a general model which abstracts Viper. We illustrate some parts of the problem this thesis focuses on in Figure 1.2. We start with a program


Figure 1.2: Illustration of the problem.
$\operatorname{Pr}$, that is a set of methods, and an initial statement $s$. This initial statement $s$ is the entry point of the program, similar to the body of a main method, and can call methods from the program. Our aim, formalized by the soundness property, is to use static inlining to answer the following question:

Is there any fundamental error in the program $P r$, with the entry point $s$ ?
This question is represented on the left side of Figure 1.2. In particular, we represent on the diagram the program inlined up to the depth bound $n$ as inline $e_{P r}^{n}(s)$. The definition of a fundamental error relates to annotated versions of the program, we therefore denote annotate $(\operatorname{Pr}, A)$ the program $\operatorname{Pr}$ annotated with an annotation $A$. This annotation $A$ represents all the annotations of the program, that is one precondition and one postcondition for each method, and one invariant for each loop.
Since there exist examples where soundness does not hold, the problem is actually about determining under which condition an error reported in the inlined program is a fundamental error. This condition is represented as "???" on the Figure, and the soundness property is represented by its contraposition: If there exists an annotation $A$ such that the program verifies (in this case there are no fundamental errors), then the inlined program (up to any finite depth) should verify.

Moreover, we also want to know how efficient static inlining is in detecting fundamental errors. We call this property completeness. It is obvious that inlining up to a depth of $n$ can only give us information on similarly bounded
executions of the original program. Therefore, we first define a bounded program bound ${ }^{n}(s, P r)$ which is a program behaving similarly to the original one, in which we simply stop the execution when we reach a depth greater than the bound. Completeness means that if no error is reported in the inlined program (the inlined program verifies), then there is no fundamental error in bound ${ }^{n}(s, P r)$ (the program can be annotated in a way which makes it verify).

On top of these two properties, soundness and completeness, we also present two extensions. The first is to explore different possibilities for defining the semantics of while loops in Viper, which then influence the soundness and completeness of static inlining. The second is to explore other useful ways of inlining (we only presented one way of inlining in this introduction, probably the simplest one).

Related work. A similar problem is tackled in [3], but with fundamental differences. In this work, the authors define a simple while language (with no method calls) and an operational semantics for this language. They mostly consider two verification workflows. A deductive verification workflow, which approximates loops with invariants, and a bounded model checking workflow, where loops are unrolled up to a fixed bound. They prove a soundness property and a completeness property for both workflows, with respect to the operational semantics. In this thesis, we do not consider any operational semantics, but study the direct link between the verification of the inlined program and the verification of the original annotated program. Moreover, the parametric language defined in this thesis is based on Viper, a concrete verification infrastructure with powerful features. One key difference is that we consider in this thesis features which make inlining unsound, and explore the conditions under which inlining is sound.

### 1.3 Overview of the thesis

This thesis is structured as follows. Chapter 2 presents some background on separation logic and Viper, needed to understand the rest. In Chapter 3, we define a desired soundness property for static inlining, and show examples of Viper programs for which this property does not hold. Based on these examples, we define two properties of statements, mono and framing, which are necessary to describe the conditions under which inlining is sound.

Since these properties and the soundness of inlining are more general than Viper, we define a general framework based on an abstraction of Viper, and prove soundness of static inlining in this framework. This is the core of this thesis. Chapter 4 defines a separation algebra [5] which abstracts the Viper state model. Building on this separation algebra, Chapter 5 defines a parametric language which abstracts the Viper language. Chapter 6 formalizes static
inlining, and proves soundness of static inlining in the general framework, under a formal condition based on the mono and framing properties. The definition of this framework with the proof of soundness of static inlining has been formalized in Isabelle/HOL [16] (see Appendix A).

Soundness is not the only interesting property about static inlining. Indeed, we are also interested in a completeness property of static inlining. Chapter 7 defines a completeness property for static inlining, and presents examples where this completeness property does not hold in Viper. We then sketch how to prove this completeness property in our general framework, under some assumptions.

Chapter 8 discusses how to instantiate the general framework for Viper, to leverage its theoretical results. We sketch a way to prove for Viper the assumption needed for the completeness property for inlining in the general framework. Moreover, we explain how to use this instantiation to create two useful features for Viper: An early error detection feature based on static inlining, and a feature to speed-up reverification ${ }^{4}$ based on caching and the properties mono and framing.
Chapter 9 proposes and explores two extensions for this work. The first idea is to extend this work to different loop semantics, and the second idea is to explore different ways of inlining to gain more information.
Finally, Appendix A shows the Isabelle/HOL formalization of the general framework (separation algebra and parametric language) in which we prove soundness of static inlining, and an instantiation of some renaming functions necessary to express the semantics of method calls and for inlining method calls.

[^1]
## Chapter 2

## Background

### 2.1 Separation logic and fractional permissions

Separation logic [17,21] is an extension of Hoare logic [10] for reasoning about programs which manipulate shared mutable data structures. It is based on the notion of partial heaps, and enables local reasoning thanks to the frame rule.

Partial heaps and states. A partial heap represents a part of the heap. A partial heap is formally a partial function, which maps heap locations (pairs of a reference and a field) to values. Two partial heaps are compatible if they are defined on disjoint domains, and their union is the partial heap which combines their mappings. A state consists of a partial heap and a store, which maps the names of local variables to their values. Two states are compatible if they agree on all values of local variables and their partial heaps can be combined.

Separating conjunction. Separation logic introduces the separation conjunction connective, written $*$. For two assertions $A$ and $B$, a state $\varphi$ satisfies the assertion $A * B$ if and only if there exist two states $\varphi_{1}$ and $\varphi_{2}$ such that $\varphi$ is the combination of $\varphi_{1}$ and $\varphi_{2}, \varphi_{1}$ satisfies $A$, and $\varphi_{2}$ satisfies $B$.

Points-to assertions. An important type of assertion introduced by separation logic is the points-to assertion. As an example, $x . f \mapsto v$ (read x.f points to $v$ ) where $v$ is some value is satisfied by a state if and only if its partial heap maps the heap location $x . f$ to the value $v .{ }^{1}$ In particular, combined with the separating conjunction, the assertion $x . f \mapsto v_{1} * y . f \mapsto v_{2}$ is satisfied by a state if and only if its heap is defined at least for two heaps locations: $x$.f and $y . f$, mapping $x$.f to $v_{1}$ and $y . f$ to $v_{2}$. In particular, this assertion implies

[^2]that $x . f$ and $y . f$ are disjoint parts of the heap. Therefore, $x . f \mapsto v$ represents ownership of the heap location $x . f$. In the logic, a state needs to own a heap location to read it or write it.

Hoare triples in separation logic. Separation logic slightly modifies the meaning of a Hoare triple. A Hoare triple is written $\{P\} \subset\{Q\}$, where $C$ is a program, and $P$ and $Q$ are assertions ( $P$ is a precondition and $Q$ a postcondition). In separation logic, $\{P\} C\{Q\}$ means that if $C$ executes from a state satisfying the precondition $P$, then nothing will go wrong and the final state will satisfy the postcondition $Q$. In particular, the precondition $P$ expresses which heap locations can be accessed and modified by the program C.

Local reasoning: The frame rule. On top of the standard rules from Hoare logic, separation logic introduces a new important rule, the frame rule, which enables local reasoning. The frame rule is defined as follows:

$$
\frac{\{P\} \subset\{Q\}}{\{P * R\} C\{Q * R\}} \bmod (C) \cap \mathrm{fv}(R)=\varnothing
$$

$\bmod (C)$ is the set of local variables modified by $C$, and $f v(R)$ is the set of free variables in the assertion $R$. This rule means in essence that if a program executes correctly in a small state satisfying $P$, then it will also execute correctly in a bigger state satisfying $P * R$, and its execution will not affect the additional part of the state $R$. $R$ should not interfere with $C$, that is $R$ should not say anything about variables which are modified by $C$.

Fractional permissions. Fractional permissions [4] are a generalization of the points-to assertion. A points-to assertion is annotated with a fractional permission $q \in \mathbb{Q} \cap(0,1]$, noted $x . f \stackrel{q}{\mapsto} v$, and partial heaps map heap locations to pairs of a permission and a value. Having some fractional permission to a location enables a state to read the value of this location. However, a state can write a heap location if and only if it has a permission of 1 to this location (full permission). Partial heaps with permissions can be combined by adding the permissions they hold, if they agree on the values of heap locations they both define, and if the sum of permissions is at most 1. Moreover, assertions can now be combined as follows, for permission amounts $q_{1}$ and $q_{2}$ :

$$
x . f \stackrel{q_{1}}{\mapsto} v_{1} * x . f \stackrel{q_{2}}{\mapsto} v_{2} \Longleftrightarrow \begin{cases}x . f \stackrel{q_{1}+q_{2}}{\mapsto} v_{1} & \text { if } v_{1}=v_{2} \wedge q_{1}+q_{2} \leq 1 \\ \perp & \text { otherwise }\end{cases}
$$

Fractional permissions are especially useful in concurrent separation logic [19], where they allow concurrent reading of the same heap location.

### 2.2 Viper

Viper [15] is a powerful verification infrastructure with an intermediate verification language, based on implicit dynamic frames [14, 24, 20] (which are based on separation logic) and fractional permissions. The semantics of the Viper language are not formally defined yet. However, they have been partially formalized and characterized [6]. Moreover, most of Viper features have a clear role, and they are explained in an online tutorial [1].
We describe in this section some parts of Viper semantics needed for this thesis. These explanations are partly based on [6], and partly based on the online tutorial.

### 2.2.1 Permissions and assertions

Permissions. Viper, based on implicit dynamic frames, separates the permission and the value of a heap location. Holding at least some fractional permission $p$ to a heap location x.f, where $x$ is a reference and $f$ a field, is encoded by the assertion acc(x.f, p), whereas the value of $x . f$ being $v$ is encoded by the assertion x.f $==\mathrm{v}$. Viper assertions do not use the usual conjunction $\wedge$, they only use the separating conjunction, written \&\& in Viper. As an example, the assertions acc(x.f, 1/2) \&\& x.f == 5 is satisfied by a state which holds at least half permission to $x . f$, and whose partial heap defines at least $x$. $f$, and maps the heap location $x . f$ to the value 5 .

Assertions. An assertion of the shape acc(x.f, p) is called an accessibility predicate. Assertions which do not contain any accessibility predicates are called pure assertions. Viper assertions can be constructed by combining assertions with the separating conjunction $\& \&$, the disjunction $\|$, and the implication ==> among others. In particular, disjunctions are only allowed when both assertions are pure. As an example, acc(x.f, 1/2) \|| x.f == 5 is not a valid assertion in Viper. The left side of an implication also has to be a pure assertion.

Self-framing assertions. Assertions used as annotations, that is as loop invariants and method preconditions and postconditions, have to be selfframing. Simply put, an assertion is self-framing if and only if includes permissions to at least the locations it reads. A better and more formal definition is given in [20]. As an example, $\operatorname{acc}(x . f, 1 / 2) \& \& x . f==5$ is self-framing, whereas $x . f==5$ is not.

### 2.2.2 Viper state model

A Viper state is defined in [6] as a triple of a store, a global heap, and a permission mask. A permission mask is simply a mapping from heap
locations to permission amounts. This permission mask actually consists of three permission masks, for permissions to field locations, permissions to predicates [8], and permissions to magic wands [23].

For our purpose, we only present here a simplified version of a Viper state. We do not consider predicates and magic wands. Moreover, we do not consider global heaps but only local heaps.

For our purpose, a Viper state can be seen as a triple $(s, \pi, h)$ where:

1. $s$ is the store: A finite partial mapping from local variables to values.
2. $\pi$ is a permission mask: A mapping from field locations (a pair of a reference and a field) to permissions (elements of $\mathbf{Q} \cap[0,1]$ ).
3. $h$ is a local heap: A finite mapping from field locations to values. In particular, the domain of definition of this heap is the finite set of field locations with a permission strictly greater than zero.

Trace semantics and state semantics. Viper semantics is actually a trace semantics. In particular, Viper defines the statement label 1 (where $l$ is simply a name) which records the heap at this point of the execution. The expression old[1](e) (where $l$ is a label and $e$ an expression) evaluates the expression $e$ in the heap recorded at the point of the execution the label $l$ refers to. In most of this thesis, we simplify the matter by only considering a state semantics, that is disallowing the use of label and old features. Chapter 8 discusses how to go from a state semantics to a trace semantics.

### 2.2.3 Building blocks of loops and method calls verification

We introduce three kinds of statements in this subsection: havoc, inhale and exhale. They are necessary to describe the semantics of loops and method calls in Viper.

Havoc. To havoc a variable simply means to assign an arbitrary value to this variable. As an example, take a state which satisfies $x==5$, that is its store maps $x$ to 5 . If we havoc $x$ in this state, then we "forget" the value of $x$. $x$ has therefore a new value, on which we do not have any information. ${ }^{2}$

Inhale. The Viper language has an inhale statement. To inhale an assertion means to assume the pure assertions (the value constraints) in it, and to add the permissions denoted in the assertion (with accessibility predicates) to the state. As an example, inhale acc(x.f, 1/2) \&\& x.f $==5$ adds half permission to $x$.f in the permission mask of the state, and assumes that the

[^3]value of $x . f$ in the local heap is 5 (that is, if $x . f \neq 5$ in the program state, then the execution stops, and the rest of the program verifies).

Exhale. exhale is in a way the opposite of inhale. To exhale an assertion means to assert (verify that it is true, otherwise throw an error) the pure assertions in it, and to subtract the permissions denoted in the assertions from the permission mask of the program state. As an example, exhale $\operatorname{acc}(x . f, 1 / 2) \& \& x . f==5$ verifies if $x . f==5$ holds and if the permission mask has at least half permission to $x$. $f$. If this exhale verifies, then half permission to $x$. $f$ is removed from the state.

### 2.2.4 Black-box semantics of loops in Viper

The general shape of a while loop in Viper is as follows, where $b$ (the guard), $I$ (the loop invariant) are assertions, and $s$ (the loop body) is a statement.

Listing 2.1: General shape of a while loop in Viper
while (b)
invariant I
\{
s
\}

To be well-defined, the loop invariant $I$ should be self-framing, meaning that $I$ must include permissions to at least the locations it reads. Moreover, $I$ is not allowed to contain any permission introspection (see Section 2.2.6). In particular, I cannot use perm and forperm.

The verification of this while loop is then equivalent to verifying what we call here the external behavior of the loop and the internal behavior of the loop. The internal behavior of the loop refers to the verification that the loop invariant is actually an invariant for the loop. The external behavior of the loop refers to the program, where the loop is approximated using the loop invariant.
The verification of the external behavior of the loop proceeds as follows (with respect to the scope surrounding the loop):

1. Exhale the loop invariant: exhale I.
2. Havoc (ie. assign arbitrary values to) all variables that get assigned by the loop body $s$.
3. Inhale the loop invariant: inhale I.
4. Assume the guard is false: assume !b.

The verification of the rest of the program then proceeds from this state. However, we need to make sure that the loop respects its contract, namely
that executing the loop body $s$ from a state satisfying $b$ and $I$ then satisfies $I$, and this for any loop iteration, this is why we havoc the variables assigned by the loop body. The verification of the internal behavior of the loop proceeds as follows:

1. Begin with a state with no permission, only variables defined before the loop and their values are known.
2. Havoc (assign arbitrary values to) all variables that get assigned by the loop body s.
3. Inhale the loop invariant: inhale I.
4. Assume the guard: assume b.
5. Execute the loop body: s.
6. Make sure that the invariant is satisfied, namely exhale it: exhale I.

The while loop verifies if and only if both the internal behavior of the loop and the external behavior of the loop verify.

### 2.2.5 Semantics of method calls in Viper

Methods and method calls in Viper are verified in a modular way. Methods are verified on their own, and method calls are verified assuming that all methods verify.

Listing 2.2: General shape of a method in Viper

```
method m(arg: T1, ...)
    returns (ret: T2, ...)
    requires P
    ensures Q
{
    s
}
```

A method can take several arguments, specified with their types (T1 is for example the type of the argument $\arg$ ), and can return any fixed number of variables, with their types (here ret of type T2). $P$ is the precondition of the method, whereas $Q$ is its postcondition. Finally, $s$ is the method body. To be well-defined, a method declaration must at least respect the following rules:

1. The method body $s$ is not allowed to modify the value of the arguments (only to read them).
2. The precondition and postcondition, similarly to loop invariants, have to be self-framing, and are not allowed to use permission introspection (see Section 2.2.6 below).
3. The only variables which can be referred to in the precondition $P$ are the arguments.
4. The only variables which can be referred to in the postcondition $Q$ are the arguments and the return variables.
5. The arguments and the return variables have to be mutually distinct.

A method is verified as follows, beginning with an empty state:

1. The arguments and the return variables are added to the store of the empty state, with some values.
2. Assume that the state satisfies the precondition: inhale $P$.
3. Execute the method body: s.
4. Ensure that the final state satisfies the postcondition: exhale Q .

A method call is verified modularly, assuming that the method it refers to verifies. A method call in Viper is written as follows:

Listing 2.3: General shape of a method call in Viper
1 y, ... :=m(x, ...)
It is well-defined if the arguments and return variables match the ones declared (number and types). Informally (we use dots to represent other variables), this method call is equivalent to:

1. (Informal) Make sure the renamed method precondition is satisfied: exhale $P[x . . . / \text { arg...] }]^{3}$
2. Havoc the return variables $y, \ldots$.
3. (Informal) Assume the renamed method postcondition is satisfied:
inhale Q[x.../arg...][y.../ret...]

### 2.2.6 Features which break the frame rule

The permission evaluation feature. As explained in the introduction, Viper has a feature of permission evaluation, perm. perm (x.f) evaluates to the permission amount held to $x . f$ by the program state. As an example, the statement assert perm(x.f) <= $1 / 2$ verifies if and only if the program state holds at most half permission to $x . f$. This statement breaks the frame rule in the following sense:

1. The program
inhale acc(x.f, 1/2)
assert perm(x.f) <= $1 / 2$

[^4]exhale acc(x.f, 1/2)
verifies with an initial empty state (which holds no permission to any field).
2. However, the program
inhale acc(x.f, 1/2) \&\& acc(x.f, 1/2)
assert perm(x.f) <= 1/2
exhale acc(x.f, 1/2) \&\& acc(x.f, 1/2)
does not verify with an initial empty state.
The first program corresponds to the Hoare triple

```
\(\{\operatorname{acc}(x . f, 1 / 2)\}\)
assert perm(x.f) <= 1/2
\(\{\operatorname{acc}(x . f, 1 / 2)\}\)
```

whereas the second program corresponds to the Hoare triple

```
{acc(x.f, 1/2) && acc(x.f, 1/2)}
assert perm(x.f) <= 1/2
{acc(x.f, 1/2) && acc(x.f, 1/2)}
```

The frame rule is broken with this statement, otherwise the second program should verify with $R:=\operatorname{acc}(x . f, 1 / 2)$.

For-perm quantifiers. A forperm statement, such as

```
forperm x: Ref [x.f] :: false
```

asserts that, for all references $x$ such that some $(>0)$ permission is held to $x . f$, false should hold. This statement verifies if and only if there exists no reference $x$ such that the program state holds some permission to $x . f$. In this case, adding some permission to some field location $x . f$ makes the statement not verify anymore. The frame rule is thus broken with this statement.

Assuming an impure assertion. To assume an assertion means doing nothing if the program state satisfies this assertion. However, if this assertion is not satisfied, the execution stops and the rest of the program verifies. In Viper, it is possible to assume impure assertions, such as assume acc (x.f, 1/2). This statement breaks the frame rule. Indeed, the following Viper statement

```
inhale true
assume acc(x.f, 1/2)
exhale false
```

verifies with an initial empty state, since the assertion is not satisfied (the state has no permission to $x . f$ ).

However,

```
inhale true \(\& \& \operatorname{acc}(x . f, 1 / 2)\)
assume acc(x.f, 1/2)
exhale false \(\& \& \operatorname{acc}(x . f, 1 / 2)\)
```

does not verify, since the assumed assertion is satisfied, but false does not.

## Chapter 3

## Unsoundness of Inlining in Viper

As explained in Chapter 1, static inlining (inlining thereafter) is not always sound for Viper programs. In this section, we explore concrete examples where inlining is unsound to derive a soundness condition. The soundness condition characterizes those Viper programs for which inlining is sound. We first express the soundness property and the requirement for the soundness condition formally. We then show three examples of unsoundness in Viper, which give rise to three properties of statements (safeMono, monoOut and framing), needed to characterize the soundness condition. Finally, we explore these three properties in Viper, and express a soundness condition for a simplified case.

### 3.1 Soundness property and soundness condition

The main purpose of inlining is to detect fundamental errors, as explained in Chapter 1. The idea is that if the inlined statement (for any bound) does not verify, then there must exist a fundamental error in the original program.

## Definition 3.1 Soundness of inlining

If there exists an annotation $A$ such that the initial statement $s$, which can call methods from the annotated program annotate ( $\operatorname{Pr}, A$ ), verifies with an initial empty state, then the statement inlined with respect to the program Pr up to the depth bound $n$ inline $_{n}(s, \operatorname{Pr})$ verifies with an initial empty state.
We denote this property soundness ${ }_{P r}^{n}(s)$.
We show in this chapter that this property does not hold in general for Viper programs. We therefore want to find a soundness condition, ideally the weakest possible, such that soundness holds. We denote this soundness condition $S C_{P r}^{n}(s)$ for a program $P r$, an initial statement $s$, and a bound $n$. This soundness condition has to satisfy the following requirement:

Listing 3.1: Annotated original program.

```
field f: Int
    inhale acc(x.f)
    callee(x)
    assert perm(x.f) <= 1/2
}
method callee(x: Ref)
    requires acc(x.f)
    ensures true
{
    assert true
}
```

field f: Int
method initial_inlined(x: Ref)
\{
inhale acc(x.f)
assert perm(x.f) $<=1 / 2$
\}

Figure 3.1: Example of unsoundness of inlining in Viper: Statement not safeMono.

## Definition 3.2 Requirement for the soundness condition

$$
S C_{P r}^{n}(s) \Longrightarrow \text { soundness }_{P r}^{n}(s)
$$

This expresses that, if the soundness condition holds for a statement, a program, and a bound, then inlining this statement with respect to this program up to this bound is sound. Namely, if it is possible to find an annotation such that the annotated program and the statement $s$ (with respect to the annotated program) both verify, then inlining this statement up to a depth of $n$ verifies. The contraposition of the soundness property is particularly useful: Under the soundness condition, if the inlined statement does not verify (for the bound $n$ ), then it is not possible to find an annotation to make the program verify. This means we detected a fundamental error, without needing any annotation from the user.

The next parts of this section focus on examples in Viper where inlining is unsound, in order to determine a suitable soundness condition.

### 3.2 Examples of Viper programs where inlining is unsound

### 3.2.1 SafeMono: Stronger states verify more

The first example of unsoundness of inlining in Viper, shown in Figure 3.1, is based on permission leaks and permission introspection. The annotated program (Listing 3.1) verifies whereas the inlined program (Listing 3.2) does
not verify. The annotated program verifies because of the permission leak happening when calling the callee method. Before the call, full permission is held by the program state to $x . f$, then half of this permission is leaked to the method, which does not return anything. After the call, only half permission is held to $x . f$, which makes line 7 verify.
The inlined program (Listing 3.2) does not verify because of line 7. Line 5 gives full permission to $x . f$, line 6 is the inlined method body (which does nothing), thus full permission is still held on line 7 .
Because of possible permission leaks, the state after a method call in an inlined program is often stronger than the corresponding state in the annotated program. Let us define this notion of stronger and weaker states.

## Definition 3.3 Stronger and weaker Viper states

A Viper state $\left(s^{\prime}, \pi^{\prime}, h^{\prime}\right)$ is stronger than another Viper state $(s, \pi, h)$ if and only if

1. For all $x$ defined in $s$, then $x$ is also defined in $s^{\prime}$, and $s^{\prime}(x)=s(x)$.
2. For all references $r$ and fields $f, \pi^{\prime}(r, f) \geq \pi(r, f)$.
3. For all references $r$ and fields $f$ such that $\pi(r, f)>0, h^{\prime}(r, f)=h(r, f)$.

If the state $\varphi^{\prime}$ is stronger than $\varphi$, then $\varphi$ is weaker than $\varphi^{\prime}$.
$A$ set of states $A^{\prime}$ is stronger than a set of states $A$ if and only if, for all states $\varphi^{\prime} \in A^{\prime}$, there exists a state $\varphi \in A$ such that $\varphi^{\prime}$ is stronger than $\varphi .^{1}$

This stronger relation for states means two things:

1. The stronger state has at least the information contained in the weaker state, namely the store values and the heap values.
2. The stronger state holds at least as much permission to heap locations as the weaker state.

We can now clarify why inlining the program in Figure 3.1 is unsound. The issue comes from the statement assert perm(x.f) <= 1/2. Indeed, there exist situations in which a weaker state makes this statement verify, whereas a stronger state makes it fail. Such a statement does not have the safeMono property:

Definition 3.4 A statement s is safeMono if and only if, for any two states $\varphi$ and $\varphi^{\prime}$ such that $\varphi^{\prime}$ is stronger than $\varphi$ and s verifies with the state $\varphi$, then salso verifies with the state $\varphi^{\prime}$.

This property has already been described in [25] and [5], where it is called Safety Monotonicity. However, this property is not enough to express the soundness condition, as shown by the next example.

[^5]Listing 3.3: Annotated original program.
field $f$ : Int
field f: Int
field f: Int
method initial(x: Ref)
method initial(x: Ref)
{
{
inhale acc(x.f)
inhale acc(x.f)
inhale acc(x.f)
assert acc(x.f, 1/2)
assert acc(x.f, 1/2)
if (perm(x.f) >= 1/1) \{
if (perm(x.f) >= 1/1) {
if (perm(x.f) >= 1/1) {
exhale acc(x.f)
exhale acc(x.f)
\}
}
}
callee(x)
assert acc(x.f, 1/2)
assert acc(x.f, 1/2)
\}
}
}
method callee(x: Ref)
requires acc(x.f, 1/2)
ensures true
\{
assert acc(x.f, 1/2)
\}

Figure 3.2: Example of unsoundness of inlining in Viper: Statement not monoOut.

### 3.2.2 MonoOut: Stonger state in, stronger states out

The example shown in Figure 3.2, like the previous one, is based on permission leaks and permission introspection, but this time with two method calls instead of one. As before, the annotated program (Listing 3.3) verifies whereas the inlined one (Listing 3.4) does not. The first call to callee in the annotated program is made with full permission to $x . f$. The method call verifies since there is at least half permission to $x . f$, and leaks half permission. After this first method call, only half permission is held, therefore the state doesn't enter the if branch, and goes directly to the second method call. Once again, this method call verifies since there is at least half permission held.

On the other hand, full permission to $x . f$ is still held in the inlined program after the inlined method call, on line 6 . The if branch is therefore entered, where full permission is exhaled. The statement on line 10 therefore doesn't verify, since no permission is held anymore.

Notice that the statement if (perm(x.f) >= 1/1) \{exhale acc(x.f)\} (lines 7 to 9 ) is safeMono: Indeed, this statement always verifies. Hence, as this example shows, the safeMono property is not enough to guarantee soundness of inlining. The issue is that a stronger state before the statement (full permission to $x . f$ for example, compared to half permission) can result in a weaker state after the statement (no permission compared to half permission). Such a statement does not have the monoOut property.

```
    Listing 3.5: Annotated original program.
field f: Int
field f: Int
method initial(x: Ref)
method initial(x: Ref)
{
    inhale acc(x.f)
    callee(x)
        inhale acc(x.f)
        exhale acc(x.f, perm(x.f))
    x.f := 5
        x.f := 5
}
}
method callee(x: Ref)
    requires true
    ensures true
{
    exhale acc(x.f, perm(x.f))
}
```

Figure 3.3: Example of unsoundness of inlining in Viper: Statement not framing.

Definition 3.5 A statement s is monoOut if and only if, for any two states $\varphi$ and $\varphi^{\prime}$ such that $\varphi^{\prime}$ is stronger than $\varphi$ and $s$ verifies with both states, then executing $s$ from the state $\varphi^{\prime}$ results in a stronger set of states than executing it from the state $\varphi$.

Combining both properties results in mono.
Definition 3.6 A statement s is mono if and only if it is safeMono and monoOut.
However, even if all statements and substatements of all methods of the program and of the initial statement are mono, we still do not have soundness.

### 3.2.3 Framing: Statements which respect the frame rule

The two previous examples showed us some conditions needed on substatements of the initial statement, but do not say anything about the body of a method which is called. In the example shown in Figure 3.3, the body of callee is mono. Indeed, it always verifies, and results in a state identical as the one before the statement, except that there is no permission to $x . f$.

The annotated program (Listing 3.5) verifies because it does not transfer any permission to callee, thus it still has full permission on line 7, and full permission is needed to write the value of $x . f$. On the other hand, the body of callee on line 6 of the inlined program (Listing 3.6) removes all permission to $x . f$, hence making line 7 fail. The underlying reason is that the idea of modular verification of methods derives from the frame rule (see Section 2.1), which not all statements satisfy in Viper, due to permission introspection.

Definition 3.7 (Informal) A statement is framing if and only if it mono and it respects the frame rule.

Intuitively, a statement $s$ is framing if it satisfies the following: "If we add information and permissions to a state which already verifies with the statement $s$, then we still have these additional information and permissions after the statement has finished executing". This view is of course a bit informal, since we need to make sure there is no interference between the information provided and the statement. For example, if we add information about the value of a variable $x$ and $s$ modifies $x$, then we lose this information after the statement. Another way to say it is that a framing statement only acts on a small part of a state, and does not "consume" the rest of it. However, we allow a framing statement to add more than what we added initially. As an example, a statement which doubles the permission held to $x . f$ (such as inhale $\operatorname{acc}(x . f, \operatorname{perm}(x . f))$ is framing. This property has also been called Framing Property [25,5]. A formal definition is given in Chapter 6.

These three examples give us a first impression of a general soundness condition, a sufficient condition for soundness of inlining to hold. Let us first define what it means to be inlinable.

Definition 3.8 A statement is inlinable if and only if it is a loop, a method call, or if it contains an inlinable substatement.

We have now a first idea of what the soundness condition should look like:

1. Non-inlinable substatements have to be mono.
2. Bodies of methods and loops have to be framing.

This first idea is really informal and only aims at giving an intuition, since for example we do not refer here to the bound $n$. The complete soundness condition is formally expressed in Chapter 6. The purpose of the next and last part of this section is to illustrate and explore these properties a bit more, before diving into the formalization.

### 3.3 Framing and mono

### 3.3.1 Inclusions and examples

By definition, mono is the intersection of safeMono and monoOut, and framing is included in mono, as represented in Figure 3.4. Examples to show that these inclusions are strict are given in Table 3.1.

Statement 1 is the sequential composition of statement 2 and statement 3. Statement 2 is not safeMono, since it can fail when adding more permission. It is otherwise monoOut since it does not modify the state if it verifies. Statement 3 is not monoOut, but it is safeMono since it always verifies.


Figure 3.4: Representation of sets of statements satisfying safeMono, monoOut and framing.
Table 3.1: Examples of safeMono, monoOut and framing statements.

| $\#$ | Statement | safeMono | monoOut | framing |
| :---: | :--- | :---: | :---: | :---: |
| 1 | assert perm(x.f) <= $1 / 2$ <br> exhale perm(x.f) >= $1 / 1 ~==>~ a c c(x . f) ~$ |  |  |  |
| 2 | assert perm(x.f) <= $1 / 2$ |  | $\checkmark$ |  |
| 3 | exhale perm(x.f) >= $1 / 1==>$ acc(x.f) | $\checkmark$ |  |  |
| 4 | exhale acc(x.f, perm(x.f)) | $\checkmark$ | $\checkmark$ |  |
| 5 | exhale acc(x.f) | $\checkmark$ | $\checkmark$ | $\checkmark$ |

Because of these properties, statement 1 is neither safeMono nor monoOut. Statement 4 is mono but not framing, as explained before. Finally, statement 5 is framing since it respects the frame rule.

### 3.3.2 Compositionality

safeMono is not stable by sequential composition. As an example, take $s_{1}:=$ exhale perm(x.f) $>=1 / 1==>\operatorname{acc}(x . f)$, and $s_{2}:=$ assert $\operatorname{acc}(x . f, 1 / 2)$. Both $s_{1}$ and $s_{2}$ are safeMono, but $s_{1} ; s_{2}$ is not: A state with half permission to $x$. $f$ verifies, but a stronger state with full permission to $x$. $f$ fails.
mono (and monoOut) are stable by sequential composition. Indeed, take $s:=s_{1} ; s_{2}$ with $s_{1}$ and $s_{2}$ mono. Take two states $\varphi$ and $\varphi^{\prime}$ such that $\varphi^{\prime}$ is stronger than $\varphi$ and $\varphi$ verifies with $s$ (thus $\varphi$ verifies with $s_{1}$ ). Since $s_{1}$ is mono, we have that it verifies with $\varphi^{\prime}$ too, and that executing it with the initial state $\varphi^{\prime}$ results in stronger states than executing it with the initial state $\varphi$. By using that $s_{2}$ is mono, we can show that $s$ is mono.

This reasoning can be extended for framing, and for non-deterministic if (if (*)):

Lemma 3.9 Assume that $s_{1}$ and $s_{2}$ are mono (resp. framing). Then

1. $s_{1} ; s_{2}$ is mono (resp. framing).
2. if (*) \{s1\} else $\{s 2\}$ is mono (resp. framing).

This result is particularly useful in practice. As shown in Chapter 6, it is necessary to determine whether some statements are framing and mono to guarantee soundness. This result means that showing that substatements of a statement are mono or framing is sufficient to prove that this statement is mono or framing.

However, the converse also does not hold. As an example, consider

$$
\begin{aligned}
s_{1} & :=\text { var } p: \operatorname{Perm}:=\operatorname{perm}(x . f) ; \text { exhale acc(x.f, p }-1 / 2) \\
s_{2} & :=\operatorname{inhale} \operatorname{perm}(x . f)<=1 / 2==>\operatorname{acc}(x . f, p-1 / 2) \\
s & :=s_{1} ; s_{2}
\end{aligned}
$$

$s_{1}$ and $s_{2}$ are not framing, but $s_{1} ; s_{2}$ is. Indeed, $s_{1}$ records the old permission held to $x . f$ in $p$, and then removes enough permission to $x . f$ to reach half permission (it fails if there is less than half permission). Therefore, any state after $s_{1}$ has half permission to $x . f . s_{2}$ makes the permission to $x . f$ increase by $p-\frac{1}{2}$ if the permission held to $x$.f is lower than half, therefore $s_{2}$ is not framing (actually not even mono, take for example two states with $p=\frac{1}{2}$, one with $\frac{1}{2}$ permission to $x$.f and one with $\frac{3}{4}$ ). Finally, $s_{1} ; s_{2}$ is framing since when it verifies, the state after this statement is the same as the one before.

The converse does not hold also with non-deterministic if. As an example, consider

$$
\begin{aligned}
s_{1} & :=\text { exhale perm(x.f) }==1 / 2==>\text { acc(x.f, perm(x.f)) } \\
s_{2} & :=\text { exhale perm(x.f) }!=1 / 2==>\operatorname{acc}(x . f, \operatorname{perm}(x . f)) \\
s & :=\text { if }(*)\{s 1\} \text { else }\{s 2\}
\end{aligned}
$$

Both statements always verify, thus they are safeMono. However, none of them is monoOut, thus they are neither mono nor framing. The statement $s$ always verifies and creates two states from an initial one: The same one, and a similar one where no more permission to $x . f$ is held. This statement is therefore monoOut, thus mono.

These compositionality properties mean that we have a sufficient condition to determine whether a statement is mono and framing by looking only at its substatements, but this is not enough to show that a statement is not mono or framing. To show that, one has to find explicit counter-examples.

To summarize, we explored examples of Viper programs where static inlining is unsound, in order to sketch a soundness condition. Our aim is to
find and express the weakest condition under which soundness of inlining holds. Through three examples, we exhibited three properties of statements, safeMono, monoOut (combined in mono), and framing, which are required to have soundness. These properties can be lifted to more general settings. The next sections define (1) a general separation algebra to abstract Viper's state model, and (2) a language abstracting resource-based languages such as Viper. Using these two ingredients, we then formalize and prove soundness under the soundness condition for a more general setting

## Chapter 4

## Abstracting Viper States with a Separation Algebra

The previous chapter focused on exploring examples of Viper programs where inlining is unsound, in order to find a general soundness condition. This chapter and Chapter 5 define a general model for a verification language, based on Viper. The idea is to lift the results obtained in Chapter 3. This chapter abstracts the Viper state model, using the concept of separation algebra [5]. Chapter 5 then defines a general parametric language and a semantics for this language, using the algebra from this chapter to represent states. Finally, Chapter 6 formally defines the mono and framing properties in this framework. Using these properties, we formally express a soundness condition, and prove that it implies soundness of inlining. The content and the proofs presented in this chapter, Chapter 5 and Chapter 6 have been mechanized and proved using the proof assistant Isabelle/HOL [16] (see the formalization in Appendix A).

A state in Viper contains resources, and more precisely two kinds of resources, pure and impure resources. The store contains pure resources, in the sense of duplicable resources. As an example, the assertion $n=5$ is equivalent to the assertion $n=5 \& \& n==5$. On the other hand, permissions to heap locations are no duplicable: Permissions are impure resources. As an example, the assertion acc(x.f, $1 / 2$ ) is not equivalent to the assertion $\operatorname{acc}(x . f, 1 / 2) \& \& \operatorname{acc}(x . f, 1 / 2)$. We define in this chapter a separation algebra, whose aim is to abstract the Viper state model. A separation algebra [5] defines what it means to add two states, and defines an order relation based on this addition. The algebra we define here makes a clear distinction between pure resources, namely the store, and impure resources, corresponding to the permission mask and the local heap in Viper. In that respect, it does not correspond exactly to a separation algebra (as in $[5,7]$ ) but it can be seen as a combination of a separation algebra with a pure part (the store).

The plan of this chapter is as follows: We first define a preliminary separation algebra, the basis of our separation algebra. We then describe, using this preliminary separation algebra, the behavior of the Viper statements inhale and exhale, restricted to self-framing assertions. These statements constitute the core of our framework, since they are the blocks at the basis of the semantics of loops and method calls. We then fully define the separation algebra, and finally highlight differences with related works.

### 4.1 A preliminary separation algebra

We first define a preliminary separation algebra, to be able to speak about addition of states as well as stronger and weaker states in an abstract way.

Definition 4.1 A preliminary separation algebra is a partial commutative monoid $(\Sigma, \oplus, u)$, where $\Sigma$ is a set of states, $\oplus$ (addition) is a partial binary operation $\Sigma \times \Sigma \rightarrow \Sigma$, and $u$ (neutral state) is an element of $\Sigma$ with the following properties, for all states $a, b, c \in \Sigma$ (we note a\#b if and only if $a \oplus b$ is defined):

1. Commutativity (definedness): $a \# b \Leftrightarrow b \# a$
2. Commutativity: $a \# b \Rightarrow a \oplus b=b \oplus a$
3. Associativity (definedness): $a \# b \# c \Leftrightarrow a \# b \wedge(a \oplus b) \# c \Leftrightarrow b \# c \wedge a \#(b \oplus c)$
4. Associativity: $a \# b \# c \Rightarrow a \oplus(b \oplus c)=(a \oplus b) \oplus c$
5. Neutral state: $a \# u$ and $a \oplus u=a$

This definition is similar to the one from [5], with one major difference: We do not require $\oplus$ to be cancellative (that is, $\lambda y . x \oplus y$ is not necessarily injective). The reason is that we also want to include pure resources in the states, and pure states are duplicable (by definition), which therefore makes this requirement unsatisfiable.

Definition 4.2 A state $\varphi$ is pure if and only if $\varphi \oplus \varphi=\varphi$.
One way to understand this definition is that a state is pure if and only if it is infinitely duplicable. This definition allows us to extract from a state its core, namely its maximum pure part. To define the core of a state, we first need to define a partial order on states:

Definition 4.3 Induced order on states: $\varphi_{1} \ll \varphi_{2} \Longleftrightarrow \exists \varphi^{\prime} . \varphi_{2}=\varphi_{1} \oplus \varphi^{\prime}$

## Lemma 4.4 The induced order is a partial order:

1. $\ll$ is reflexive.
2. $\ll$ is transitive.
3. $\forall \varphi \in \Sigma . u \ll \varphi$.
```
        Listing 4.1: Pure assertion.
field f: Int
method \(m(n\) : Int, \(x\) : Ref)
\{
    inhale \(\mathrm{n}>=0\)
    exhale \(n>=0\)
    assert \(\mathrm{n}>=0\)
\}
```

Listing 4.2: Accessibility predicate.

```
field f: Int
method m(n: Int, x: Ref)
{
    inhale acc(x.f)
    exhale acc(x.f)
    assert perm(x.f) == 0/1
}
```

Listing 4.3: Combination of a pure assertion and an accessibility predicate.

```
field f: Int
method m(n: Int, x: Ref)
{
        inhale n >= 0 && acc(x.f)
        exhale n >= 0 && acc(x.f)
        assert n >= 0 && perm(x.f) == 0/1
}
```

Figure 4.1: Inhaling and exhaling the same assertion.

### 4.2 Motivation: Modeling the behavior of inhale and exhale on annotations

The preliminary separation algebra is the basis of our separation algebra. We explain in this section the reason why we distinguish two kinds of resources, pure and impure. Inlining deals with method calls and loops. Moreover, in Viper, the verification of loops and method calls is based on two statements, inhale and exhale, for self-framing (as defined in [20]) assertions. In this section, we illustrate the behavior of inhale and exhale with pure assertions and assertions containing accessibility predicates. Annotations in Viper, such as loop invariants, can be seen as having two roles: One role, based on pure resources, is simply to make the approximation of the behavior of the loop more precise. The second role, based on impure resources, is to model the transfer of ownership of heap locations.
In Figure 4.1 are presented three programs, all of whom verify. These three programs begin with an almost empty state (only two variables, $n$ and $x$, are defined). They then inhale and exhale the same assertion. The purpose of the assertions in the end is to show what we know about the final states.

The first example (Listing 4.1) inhales and exhales a pure assertion, that is an assertion speaking about a pure resource (the local variable $n$ ). inhale in the
case of a pure assertion is equivalent to assume, namely only the states which satisfy this assertion go through, the execution is stopped for the other ones. exhale in the case of a pure assertion is equivalent to assert, which verifies if and only if the assertion is true. Therefore, inhaling and exhaling a pure assertion is equivalent to assuming this assertion, and the exhale statement does not modify the state.

In the second example (Listing 4.2), the assertion is an accessibility predicate. Inhaling this assertion means adding the permission it contains to the current state, namely going from no permission to $x$.f to full permission to $x$.f. Exhaling this assertion then "cancels" this by removing the permission contained in the assertion, namely going from full permission to no permission. This is a fundamental difference from the previous example, where pure information is not forgotten.

The last example (Listing 4.3) shows what happens when pure and accessibility predicate are combined. The pure information ( $n \geq 0$ ) stays, whereas the permission is removed. This difference is at the core of our separation algebra.

### 4.3 Definition of the separation algebra

To finally define the separation algebra, we need to define the core of a state, which represents the pure part of the state, namely the store.

## Definition 4.5 Core of a state

The core set of a state is defined as

$$
\operatorname{coreset}(\varphi):=\{p \in \Sigma . p \text { is pure } \wedge p \ll \varphi\}
$$

The core of a state $\varphi$ is denoted $|\varphi|$, and is defined as ${ }^{1}$

$$
|\varphi|:=\sum_{p \in \operatorname{coreset}(\varphi)} p
$$

## Lemma 4.6 Core properties:

1. $u \in \operatorname{coreset}(\varphi)$
2. $\operatorname{coreset}(\varphi)$ is stable by $\oplus: \forall a, b \in \operatorname{coreset}(\varphi) \cdot a \# b \wedge a \oplus b \in \operatorname{coreset}(\varphi)$
3. If coreset $(\varphi)$ is finite, then $|\varphi|=\max (\operatorname{coreset}(\varphi))$

The situation we are interested in is when $\operatorname{coreset}(\varphi)$ is finite for all states, namely when all states have a finite store. We now define the separation algebra, using a new function $C: \Sigma \rightarrow \Sigma$, which computes the complementary

[^6]state of the core of a state. The core of a state $|\varphi|$ contains the pure resources of the state, whereas the complementary $C(\varphi)$ contains the impure resources of the state (permissions to and values of heap locations):

Definition 4.7 A separation algebra is a quadruple $(\Sigma, \oplus, u, C)$ which satisfies the following properties:

1. $(\Sigma, \oplus, u)$ is a preliminary separation algebra.
2. Finiteness: $\forall \varphi \in \Sigma . \operatorname{coreset}(\varphi)$ is finite.
3. Partially cancellative: $\forall \varphi, a, b \in \Sigma . C(\varphi) \oplus a=C(\varphi) \oplus b \Longrightarrow a=b$
4. Decomposition: $\forall \varphi \in \Sigma . \varphi=|\varphi| \oplus C(\varphi)$
5. Empty core: $\forall \varphi \in \Sigma .|C(\varphi)|=u$
6. Uniqueness: $\forall \varphi, a \in \Sigma . \varphi=|\varphi| \oplus a \wedge|a|=u \Longrightarrow a=C(\varphi)$
7. Positivity: $\forall a, b \in \Sigma . a \oplus b=u \Longrightarrow a=u$
8. Pure reducibility: $\forall p, a, b \in \Sigma . p \ll a \oplus b \wedge p$ is pure $\Longrightarrow p \ll|a| \oplus|b|$

We already presented the first axiom. The purpose of the finiteness axiom is to have the core of a state being the maximum of the coreset, which means that all stores are finite. Axiom 3 gives us the cancellative property that we gave up from [5] by allowing pure states to exist. Axioms 4, 5 and 6 ensure that $C(\varphi)$ is the unique state with no pure part such that, added to the core of $\varphi$, gives $\varphi$. Axiom 7 means that nothing can come out of an empty state. Axiom 8 means that if a pure state is smaller than $a \oplus b$, then its resources are contained in the combination of the pure resources from $a(|a|)$ and the pure resources of $b(|b|)$.

Remark 4.8 From the finiteness axiom and Lemma 4.6, we already get that

$$
\forall \varphi \in \Sigma . \exists c \in \Sigma . \varphi=|\varphi| \oplus c
$$

without the uniqueness and the empty core axioms.
Remark 4.9 It is possible to inject the original separation algebra (from [5]) in this separation algebra, by getting rid of the store and of pure states, namely by having that $\forall \varphi \in \Sigma .|\varphi|=u \wedge C(\varphi)=\varphi$. Everything (except positivity) can then be directly proven.

Here are some useful properties which can be proven from these axioms:
Lemma 4.10 Suppose $(\Sigma, \oplus, u, C)$ is a separation algebra. Then

1. $\ll$ is antisymmetric.
2. $\forall \varphi_{1}, \varphi_{2} \in \Sigma . \varphi_{1}=\varphi_{2} \Longleftrightarrow\left|\varphi_{1}\right|=\left|\varphi_{2}\right| \wedge C\left(\varphi_{1}\right)=C\left(\varphi_{2}\right)$
3. $\forall \varphi_{1}, \varphi_{2} \in \Sigma . \varphi_{1} \ll \varphi_{2} \Longleftrightarrow\left|\varphi_{1}\right| \ll\left|\varphi_{2}\right| \wedge C\left(\varphi_{1}\right) \ll C\left(\varphi_{2}\right)$
4. $\forall \varphi \in \Sigma .|\varphi| \ll \varphi \wedge\|\varphi\|=|\varphi|$
5. $\forall \varphi \in \Sigma . C(\varphi) \ll \varphi \wedge C(C(\varphi))=C(\varphi)$

Finally, we define an addition and an order on sets, which formalize the corresponding definitions from Chapter 3.

Definition 4.11 Addition and order on sets
Let $A, B \subseteq \Sigma$. Then

$$
\begin{gathered}
A \oplus B:=\{a \oplus b \mid a \in A, b \in B, a \# b\} \\
A \gg B \Longleftrightarrow(\forall a \in A \cdot \exists b \in B . b \ll a)
\end{gathered}
$$

## Lemma 4.12 Properties of addition of sets

1. Commutativity: $A \oplus B=B \oplus A$.
2. Associativity: $A \oplus(B \oplus C)=(A \oplus B) \oplus C$.
3. Transitivity: $A \gg B \wedge B \gg C \Longrightarrow A \gg C$.
4. Neutral: $A \oplus\{u\}=A$.
5. Transitive closure: $A \gg B \Longleftrightarrow\left\{a^{\prime} \mid \exists a \in A . a \ll a^{\prime}\right\} . \subseteq\left\{b^{\prime} \mid \exists b \in\right.$ B. $\left.b \ll b^{\prime}\right\}$.
6. Biggest element: $\varnothing \gg A$.
7. Addition is bigger: $A=B \oplus C \Longrightarrow A \gg B$.

Remark 4.13 The reciprocal of property 7 is not true. As an example, take three states $x_{f}, x_{h}, y_{f}$ where $x_{f}\left(\right.$ resp. $y_{f}$ ) has full permission to $x . f$ (resp. y.f) and $x_{h}$ has half permission to $x . f$.
Then $A:=\left\{x_{h} \oplus y_{f}, x_{f}\right\} \gg B:=\left\{x_{h}, x_{f}\right\}$. Now imagine there exists $C$ such that $A=B \oplus C$. We have $y_{f} \in C$, thus $x_{f} \oplus y_{f} \in B \oplus C$, but $x_{f} \oplus y_{f} \notin A$.

### 4.4 Related work

The concept of abstract separation logic and separation algebra was introduced in 2007 [5]. A separation algebra is defined in this work as a "cancellative, partial commutative monoid". Separateness and substate relations are induced from this separation algebra. We based our work on these separation algebra axioms and these induced relations. However, we do not consider the same cancellative axiom. As we explained in this section, we want to consider duplicable states (pure states) representing stores in our algebra.

This then implies, by definition, that our algebra cannot be cancellative. We still require a cancellative property, but only for states with empty stores.
A "fresh look at separation algebras" was presented in 2009 [7]. In this work, the authors begin with the same axioms as before. They then weaken the existence of a neutral element (the unit) axiom with the existence of $a$ unit for all elements. They go from

$$
\exists u . \forall x . u \oplus x=x
$$

to

$$
\forall x \cdot \exists u_{x} \cdot u_{x} \oplus x=x
$$

These units can be seen in a way as corresponding to pure states and cores (since $|\varphi| \oplus \varphi=\varphi$ ), even though the existence of a neutral state in our algebra makes this axiom trivially satisfied.
They also add some more axioms. A positivity axiom

$$
a \oplus b=c \wedge c \oplus c=c \Longrightarrow a \oplus a=a
$$

that we reproduce in a way. This axiom, directly intepreted in our algebra, means that if adding two states results in a state with only a store (and no permission), then both states are pure (only stores and no permission). We have a similar axiom (also called "positivity", Definition 4.7) saying that one can only obtain empty states from the empty state. These axioms ensure that the induced relation is antisymmetric.
They also add a disjointness axiom (stronger than the positivity one), requiring that only pure states can be joined (added) to themselves. We do not use this axiom since Viper uses fractional permissions, because fractional permissions do not make the difference between two equivalent amounts of permission. Indeed, two Viper states with permission to read the same field can be joined as long as the sum of their permissions is at most one (the full permission).

Finally, they consider a splittability axiom requiring that it is always possible to split a non-empty state in two non-empty states, and the following crosssplit axiom:

$$
\begin{aligned}
& a \oplus b=z \wedge c \oplus d=z \\
& \Longrightarrow \exists a c, a d, b c, b d . a c \oplus a d=a \wedge b c \oplus b d=b \wedge a c \oplus b c=c \wedge a d \oplus b d=d
\end{aligned}
$$

We do not use these axioms, but use a similar one (pure reducibility, Definition 4.7) requiring that if a pure state (a store) is a substate of the sum of two states, then this pure state is smaller than the sum of the cores of the summands.

Finally, we use the same name and notation for the core as Iris [11]. In this work, the authors define a resource algebra with a duplicable core noted |_|, which has properties similar to our own. This core is duplicable $(a \oplus|a|=a)$, idempotent $(||a||=|a|)$, and monotone ( $a \ll b \Longrightarrow|a| \ll|b|)$, similarly to ours. However, the core operator they consider is only partial, whereas ours is always defined.

To conclude this chapter, we defined a general separation algebra to capture the key properties of the Viper state model as well the behaviour of inhale and exhale statements on self-framing assertions. In particular, we define a separation algebra which considers impure as well as pure (duplicable) resources. We first presented a simple preliminary separation algebra on which we build on. We then showed, through examples, the motivation for the shape of our separation algebra, based on the behavior of inhale and exhale statements, statements which are at the basis of loops and method calls. Finally, we defined this separation algebra with several axioms, showed some interesting properties, and compared it to related works.
The next chapter builds a language on top of this separation algebra. The purpose of this language is to be as general as possible while modeling the semantics of Viper loops and method calls, in order to show general soundness and completeness properties for inlining.

## Chapter 5

## A Parametric Language for Resource-Based Inlining

In this chapter, we define a general parametric language for studying resourcebased inlining. The states for the semantics of this parametric language are elements of a separation algebra, as defined in Chapter 4. The structure of the model is represented in Figure 5.1. The inputs for this model consist in a separation algebra, two types, and four interfaces. Using these inputs, we define a language and its semantics. We make this language as general as possible, by just requiring the minimum needed for reaching two objectives:

1. Model as closely as possible the semantics of loops and method calls in Viper.
2. Express a soundness condition as sketched in Chapter 3 and prove soundness of inlining.

This chapter is organised as follows. We fix a separation algebra $(\Sigma, \oplus, u, C)$ throughout the chapter. Section 5.1 presents the parameters of the model, the statements of the language, and the general shape of its semantics. Section 5.2 then describes the type of variables, the store interface, and defines the semantics of defining and havocing variables. Section 5.3 explores how to encode syntactic assertions from Viper into semantic assertions, and defines the role of the assertions interface, the semantics of inhaling, exhaling and assuming, all needed for encoding loops and method calls. Section 5.4 describes the semantics of while loops. Section 5.5 describes the need for and the role of the rename interface, as well as its requirements. Section 5.6 describes the semantics of method calls, using the previously defined renaming functions. Section 5.7 completes the definition of the semantics, by describing the role of the type of custom statements, and the requirements for the custom interface. Finally, Section 5.8 describes what it means for a program to be well-defined and to verify. Everything presented


Figure 5.1: Illustrations of the input parameters for the model. $P$ is a semantic assertion, $\varphi$ is an element of $\Sigma, t$ is a renaming quadruple, and $o$ is a custom statement of type $O$.
in this chapter has been mechanized and proved with the proof assistant Isabelle/HOL [16] (see Appendix A).

### 5.1 The language

### 5.1.1 Parameters of the model

Figure 5.1 represents all the input parameters of our framework. Based on these parameters, our framework defines a language and its semantics. This model requires first a separation algebra, whose elements are the states for the semantics. On top of this separation algebra, two types are required:

1. Type $V$ is the type of variable names in our language (string in Viper).
2. Type $O$ is a type of custom statements. This allows the model to be instantiated with statements that are not defined initially in the statements of our language. As an example, our language does not define an assert statement, and does not define a statement for assigning variables. These two statements can therefore be defined through this
type $O$ (see Chapter 8 for more details).
The remaining parameters are four interfaces. These interfaces have to provide the functions listed on the figure. Moreover, these functions must satisfy some requirements. These requirements are given throughout the chapter.

The first interface, the store interface, simply defines a store function $\sigma$, such that

$$
\sigma(\varphi)
$$

is the set of the names of the variables defined in the state $\varphi$. Section 5.2 explains this.

The assertions interface defines a function, whose aim is to model the welldefinedness of an assertion in a given state. The function

$$
\text { wellDefinedAssert }(P, \varphi)
$$

returns a boolean: true if the semantic assertion $P$ is well-defined in the state $\varphi$, false otherwise. It is explained in Section 5.3.
A method call $\vec{y}:=m(\vec{x})$ requires to rename the formal arguments of the methods to $\vec{x}$, and the result variables of the method to $\vec{y}$. This is one role for the rename interface. The other role is to rename method bodies when inlining, as defined in Chapter 6 . This interface should provide four functions:

1. rename $(t, x)$ (variable names).
2. renameState $(t, \varphi)$ (states).
3. renameCustom $(t, o)$ (custom statements).
4. renameInv( $t$ ) (inverts a renaming quadruple).
where $t$ is a renaming quadruple (a configuration with four lists of variable names which defines how to rename). rename renames a variable name to another, renameState renames a state to another state (that is it renames the variables in the store of the state), renameCustom renames a custom statement of type $O$ to another. Finally, renameInv takes a renaming quadruple and returns another, which inverts the renaming. The meaning of these functions and the requirements they should satisfy are explained in Section 5.5.

Finally, the custom statements allowed in our language are characterized by the following four functions:

1. semanticsCustom Pr $(\varphi, o)$ (semantics).
2. $w^{\prime}$ Custom $_{P r}(o)$ (well-formed statement).
3. modifCustom (o) (variables modified by a statement).
4. readCustom (o) (variables read by a statement).

These functions are also explained in Section 5.7.
The following requirements deal with the rename interface:

1. Requirement 5.37: renaming an element.
2. Requirement 5.38: renaming a state.
3. Requirement 5.40: renaming an assertion.
4. Requirement 5.48: renaming a custom statement.

Finally, the following requirements deal with the other interfaces:

1. Requirement 5.6: store interface.
2. Requirement 5.23: assertions interface.
3. Requirement 5.47: custom interface.

### 5.1.2 Statements

The statements of the language presented in this chapter are as follows:

## Definition 5.1 Statements of the language

$S$ represents statements, $A$ assertions, $\vec{V}$ lists of elements of $V$ (variable names) and $O$ is the type of custom statements.

$$
\begin{aligned}
S:= & S ; S \mid \text { if }(*)\{S\} \text { else }\{S\} \mid \text { while }(A) \operatorname{inv} A\{S\} \mid \vec{V}:=m(\vec{V}) \\
& \mid \text { inhale } A \mid \text { exhale } A \mid \text { assume } A|\operatorname{var} \vec{V}| \text { havoc } \vec{V} \mid \text { custom } O \mid \text { skip }
\end{aligned}
$$

$S ; S$ is a sequential composition, if $(*)\{S\}$ else $\{S\}$ a non-deterministic conditional branching, while $(b)$ inv $I\{s\}$ a loop, $\vec{V}:=m(\vec{V})$ a method call, assume assumes an assertion, var declares a list of variables while havoc havocs a list of variables, custom allows one to encode statements not presented here, and skip does nothing.

Remark 5.2 We already wrote about inhale and exhale in Viper. The inhale and exhale statements presented here are different and behave differently. We simply require them to be equivalent when applied to annotations (eg. $\operatorname{acc}(x . f) \& \& x . f==5$ ), but they can behave differently with other assertions.

When injecting Viper into this framework, we therefore distinguish inhale and exhale which come from loop and method call verification with the ones which do not (which should be encoded with custom).

Most of these statements are given the usual semantics, and their semantics are described formally throughout this chapter. We use a non-deterministic if in this framework, since it is simpler to define and more general than a deterministic if, which we define as follows:

## Definition 5.3 Deterministic Ifs

$$
\text { if } \begin{aligned}
&(b)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\} \\
& \text { if }(b)\left\{s_{1}\right\}:=\text { if }(*)\left\{\text { assume } b ; s_{2}\right\} \text { else }\left\{\text { assume } \neg b ; s_{2}\right\} \\
&\text { assume } b ; s\} \text { else }\{\text { assume } \neg b\}
\end{aligned}
$$

Moreover, the statements for declaring and havocing variables, as well as the statements for calling methods, are defined with lists of variables instead of single variables, to allow more expressivity.

### 5.1.3 Semantics

We define as follows the semantics with respect to a program:

## Definition 5.4 Method, program and semantics

- A method is a quadruple $(m, \overrightarrow{\operatorname{args}}, \overrightarrow{r e t s}, P, Q, s)$ where $m$ is a string (name of the method), $\overrightarrow{\operatorname{args}}$ and $\overrightarrow{\text { rets are lists of variable names (representing arguments }}$ and return variables), $P$ (precondition) and $Q$ (postcondition) are semantic assertions (functions from states $\Sigma$ to booleans), and s is a statement (the method body).
- A program Pr is a finite set of methods.
- We denote semantics ${ }_{P r}(\varphi, s)$ the semantics of the statement s executed with the initial state $\varphi$, with respect to the program Pr. It is either an error or a set of states. ${ }^{1}$
We consider a fixed semantics function in this section, which we characterize throughout this chapter. Finally, we define two auxiliary functions based on the semantics function:

Definition 5.5 Let Pr be a program, A be a set of states and s be a statement. We define two functions ver and sem such that

$$
\begin{aligned}
\operatorname{ver}_{P r}(A, s) & :=\left(\forall \varphi \in A \cdot \operatorname{semantics}_{P r}(\varphi, s) \neq \text { Error }\right) \\
\operatorname{sem}_{P r}(A, s) & :=\cup_{\varphi \in A} \operatorname{semantics}
\end{aligned}
$$

and $\operatorname{sem}_{P_{r}}(A, s)$ is defined if and only if ver $_{P r}(A, s)$.

[^7]$\operatorname{ver}_{P r}(A, s)$ is true if and only if the statement $s$ verifies with all states from the set $A$, with respect to the program $\operatorname{Pr}$. In this case, $\operatorname{sem}_{P r}(A, s)$ is the set of all states resulting from executing $s$ from all states of $A$. In this chapter, we mostly characterize the semantics using these notations. The original semantics function can be fully reconstructed from these two auxiliary functions.

### 5.2 Store of a state

In order to define the semantics of var and havoc, we need to introduce the concept of store of a state. We first explain the requirements on the store function $\sigma$ for a state, and then define the semantics of var and havoc.

### 5.2.1 Store function

In Viper, the store of a state can be seen as a finite mapping from variable names to values of the right type. In this framework, we do not need as much precision and information. A state in our framework may still use a mapping to define a store. However, for the purpose of our framework (proving properties on static inlining), we only require a projection of this mapping to the sets of variables it defines. The validity of assertions such as $x==5$ is handled by the separation algebra.

Requirement 5.6 The store function $\sigma$ from states to sets of variable names must satisfy the following rules (for all states $\varphi, a, b$, and all variable names $x$ ):

1. $\sigma(u)=\varnothing$
2. $x \in \sigma(\varphi) \Longrightarrow(\exists c \in \Sigma . \sigma(c)=\{x\} \wedge c \ll \varphi)$
3. $a \# b \Longrightarrow \sigma(a \oplus b)=\sigma(a) \cup \sigma(b)$
4. $\sigma(\varphi)=\sigma(|\varphi|)$
5. $a \# b \wedge a$ is pure $\wedge \sigma(a) \subseteq \sigma(b) \Longrightarrow a \ll b$
6. $\sigma(a) \cap \sigma(b)=\varnothing \wedge a$ is pure $\Longrightarrow a \# b$

As an example, if a state $\varphi$ has a store which maps $x$ to $5, y$ to 7 , and $z$ to 42 , then $\sigma(\varphi)=\{x, y, z\}$. The first three axioms deal with combining states. The empty state has an empty store (axiom 1). It is always possible to find a weaker state than $\varphi$ with a store containing only one variable $x$ (which is defined in the store of $\varphi$ ) (axiom 2). The store of the addition of two states is the union of their stores (axiom 3). Axioms 4 and 5 formalize that the core of a state represents exactly its store. The store of a state is the store of its core (axiom 4). If $a$ and $b$ can be added, this means that they agree on the values of local variables which both stores define. Therefore, if $a$ is pure (only a store) and its store is included in the store of $b$, then $a$ is weaker than $b$
(axiom 5). Finally, it is always possible to add new variables to a state (axiom 6).

Using the finiteness axiom of the separation algebra, we can prove the following lemma:

Lemma 5.7 $\forall \varphi \in \Sigma . \sigma(\varphi)$ is finite.

### 5.2.2 Var and havoc

Declaring variables verifies if and only if no variable from this list is already declared, whereas havocing variables verifies if and only if all variables from this list are already declared. For readability purposes, and throughout this thesis, we consider implicit conversions between lists of variable names and sets of variable names.

## Definition 5.8 Verification of var and havoc:

$$
\begin{aligned}
\operatorname{ver}_{P r}(\{\varphi\}, \operatorname{var} \vec{l}) & \Longleftrightarrow \vec{l} \cap \sigma(\varphi)=\varnothing \\
\operatorname{ver}_{P r}(\{\varphi\}, \text { havoc } \vec{l}) & \Longleftrightarrow \vec{l} \subseteq \sigma(\varphi)
\end{aligned}
$$

Remark 5.9 This definition of $\operatorname{ver}_{P r}(\{\varphi\}, \operatorname{var} \vec{l})$ actually makes var not safeMono. An alternative definition where var defines variables which are not already defined and leaves the other variables unchanged would make all var statements safeMono. This alternative definition can also be used with the custom interface. If it can be externally justified that no variable names collide, then the two statements are equivalent in terms of semantics.

The semantics of var and havoc, when they verify, are a bit different. var has to define (add to the store) the new variables, with all possible values. havoc has to remove the variables, and then define them again with all possible values. To define these two semantics, we define two auxiliary functions: One to define variables with all possible values, and one to remove variables from a state. We define below the former:

Definition 5.10 h function:

$$
h(\vec{l})=\{\varphi \mid \varphi \text { is pure } \wedge \sigma(\varphi)=\vec{l}\}
$$

$h(\vec{l})$ is the set of all states with no impure resources, only a store, where the variables which are defined are exactly $\vec{l}$. This function is sufficient to define the semantics of var when it verifies:

Definition 5.11 Semantics of var:

$$
\operatorname{sem}_{P r}(\{\varphi\}, \operatorname{var} \vec{l})=\{\varphi\} \oplus h(\vec{l})
$$

This is the definition we want. Indeed, adding this set to the singleton $\{\varphi\}$ results in all states which correspond to $\varphi$ plus the definition of the variables contained in $\vec{l}$, with all possible values.

To define the semantics of havoc, we need a function which "undefines" (removes) variables from a state. Such a function exists because of the following lemma:

Lemma 5.12

$$
\forall \varphi \in \Sigma . \exists!p \in \Sigma . p \ll \varphi \wedge \sigma(p)=\sigma(\varphi)-V \wedge p \text { is pure }
$$

This lemma shows that there exists a unique store (pure state) which defines the variables in $\sigma(\varphi)-\vec{l}$ and matches exactly the store of $\varphi$ excluding $\vec{l}$. It is now easy to define a function which only removes the part of the store corresponding to $\vec{l}$ :

Definition $5.13 \bar{h}$ function:
Let $p$ be the unique $p$ such that $p \ll \varphi \wedge \sigma(p)=\sigma(\varphi)-\vec{l} \wedge p$ is pure. Then

$$
\bar{h}(\varphi, \vec{l})=p \oplus C(\varphi)
$$

We can finally define the semantics of havoc.
Definition 5.14 havoc semantics:

$$
\operatorname{sem}_{P r}(\{\varphi\}, \text { havoc } \vec{l})=\{\bar{h}(\varphi, \vec{l})\} \oplus h(\vec{l})
$$

This is the semantics described above: We remove $\vec{l}$ from the store of $\varphi$, and then we add again the variables from $\vec{l}$, but with all possible values.

### 5.3 Assertions

The purpose of this section is to define the semantics of inhale, exhale and assume. We first explain how to encode syntactic assertions from Viper into semantic assertions in our separation algebra. We then define two properties, supported and intuitionistic, which are satisfied by semantic assertions encoded from Viper annotations. Afterwards, we discuss a mismatch between syntactic and semantic assertions, how this gap could be bridged, and how we currently define the verification of inhale, exhale and assume in this framework. This means that we only need to model the behavior of inhale and exhale on this restricted class of semantic assertions. Eventually, we define the concept of the minimal satisfying set of an assertion, and use this concept to express the semantics of inhale and exhale.

### 5.3.1 From syntactic to semantic assertions

Viper assertions written in Viper programs are syntactic assertions. However, we do not want to define a syntax for assertions in our framework. We therefore encode syntactic assertions into semantic assertions. Following [21, $25,5,18$ ], we define semantic assertions as follows:

Definition 5.15 A semantic assertion on a set of states $\Sigma$ is a function $\Sigma \rightarrow$ Bool.
Remark 5.16 It is equivalent to see semantic assertions as elements of $\mathcal{P}(\Sigma)$.
Semantic assertions aim at capturing syntactic assertions (as in Viper). For an infinite number of states, there is an uncountable number of semantic assertions (Cantor's theorem), whereas there is only a countable number of syntactic assertions (defined on a finite alphabet). Therefore, our hope is that every syntactic assertion can be represented as a semantic assertion, and thus that working with semantic assertions is more general than working with syntactic assertions. However, Section 5.3.3 argues that there is a mismatch (at least for assertions in Viper).
Definition 5.15 is sufficient to define the semantics of assume when the statement verifies (the verification of assume statements is defined in Section 5.3.3).

Definition 5.17 Partial semantics of assume

$$
\operatorname{sem}_{P r}(\{\varphi\}, \text { assume } P)= \begin{cases}\{\varphi\} & \text { if } P(\varphi) \\ \varnothing & \text { otherwise }\end{cases}
$$

assume statements simply filter states which do not satisfy the assertion. That is, they stop the execution of these states.

### 5.3.2 Annotations are supported and intuitionistic

Semantic assertions are too general for our purpose, since we just want to match the behavior of inhale and exhale on annotations, and annotations are syntactic assertions with restrictions. Indeed, annotations have to be self-framing and are not allowed to use permission introspection. We can relate this restriction with two notions for semantic assertions.
As explained in Chapter 1, not using permission introspection introduces a certain monotonicity, which can be related to the notion of intuitionistic assertions [21] :

Definition 5.18 A semantic assertion P is intuitionistic if and only if

$$
\left(\forall \varphi, \varphi^{\prime} \in \Sigma . \varphi \ll \varphi^{\prime} \Longrightarrow\left(P(\varphi) \Rightarrow P\left(\varphi^{\prime}\right)\right)\right)
$$

This means that if a state satisfies an intuitionistic assertion, then all stronger states satisfy it too.
Moreover, syntactic self-framing assertions from Viper are encoded into semantic supported assertions (more explanations are given in Chapter 8). As explained in Section 2.2.1, an assertion is self-framing if it includes permissions to at least the locations it reads. This property, combined with other restrictions on assertions in Viper, implies that its representation as a semantic assertion is supported [18]:

Definition 5.19 An assertion $P$ is supported if and only if

$$
\begin{aligned}
& (\forall \varphi \in \Sigma \cdot P(\varphi) \\
& \left.\stackrel{\Longrightarrow}{\Longrightarrow} \varphi_{m} \in \Sigma \cdot \varphi_{m} \ll \varphi \wedge P\left(\varphi_{m}\right) \wedge\left(\forall \varphi^{\prime} \in \Sigma \cdot \varphi^{\prime} \ll \varphi \wedge P\left(\varphi^{\prime}\right) \Rightarrow \varphi_{m} \ll \varphi^{\prime}\right)\right)
\end{aligned}
$$

In other words, a semantic assertion $P$ is supported if and only if, for any state $\varphi$ that satisfies $P$, it is possible to find a support $\varphi_{m}$ weaker than $\varphi$, which also satisfies $P$. Being a support means that all states weaker than $\varphi$ satisfying $P$ are stronger than the support $\varphi_{m}$.

In Viper, a state needs at least some ( $>0$ ) permission to a location to have information about it. As an example, the assertion x.f == 5 (not self-framing) is not a supported assertion, since it is impossible to find a support for this assertion and a state satisfying this assertion. Indeed, any state satisfying this assertion has to have some permission to $x . f$, and so does its support if it would exist. However, any state satisfying $x . f==5$ with less permission (but not zero) than the support would be weaker than the support while satisfying the assertion. Such a state exists: It is for example sufficient to divide the permission by two. This contradicts the definition of the support.
Another example of a not supported assertion is acc(x.f) \| acc(y.f) (such an assertion is not allowed in Viper). Take the state $\varphi$ which has full permission to both $x . f$ and $y . f . \varphi$ satisfies the assertion. Take then the state $\varphi_{x}$ (resp. $\varphi_{y}$ ) with only full permission to $x . f$ (resp. y.f). $\varphi_{x}$ and $\varphi_{y}$ satisfy the assertion and are weaker than $\varphi . \varphi_{x}$ and $\varphi_{y}$ cannot be the support, since $\varphi_{x}$ is not weaker than $\varphi_{y}$, and vice-versa. Moreover, a state weaker than both $\varphi_{x}$ and $\varphi_{y}$ (as a support for $\varphi$ should be) would have no permission to $x . f$ and $y . f$, therefore not satisfying the assertion.

Semantic assertions encoded from Viper annotations are well-formed:
Definition 5.20 Well-formed assertions
A semantic assertion $P$ is well-formed, written wfAssert $(P)$, if and only if $P$ is supported and intuitionistic.

Remark 5.21 This definition of well-formed semantic assertions is a bit too restrictive. Some Viper annotations, such as disjunctions, do not correspond to supported
assertions. To prove the soundness theorem (Chapter 6), a semantic assertion P used as an annotation needs only to satisfy the following property, which is satisfied by all well-formed assertions:

$$
\forall \varphi \in \Sigma \cdot P(\varphi) \Longleftrightarrow\{\varphi\} \gg \operatorname{Inh}(P)
$$

The definition of well-formed assertions will be modified with this property in the future.

In the next subsections, we model the behavior of inhale and exhale based on supported and intuitionistic semantic assertions. Chapter 8 explains why Viper annotations (syntactic assertions) are encoded into supported and intuitionistic semantic assertions, which justifies this choice.

### 5.3.3 Well-defined assertions: A mismatch between syntactic and semantic assertions

assume statements in Viper do not always verify. They verify if and only if they are well-defined in a state. As an example, assume $\mathrm{x}>=7 \& \& \mathrm{y}<=9$ verifies only in a state where $\vec{x}$ and $\vec{y}$ are defined. We want to transpose this concept of well-definedness from syntactic assertions to semantic assertions, in order to describe when assume, inhale and exhale verify. This is encoded via the wellDefinedAssert function from the assertions interface. The difficulty boils down to the definition of such a property, and by extension to the encoding of a syntactic assertion into a semantic assertion.

It does not actually work, because of a mismatch between syntactic and semantic assertions. To illustrate this mismatch, consider two Viper assertions $P$ and $Q$, where $P:=(x . f==x . f)$, and $Q:=(a==a) . P$ is well-defined with respect to a state if and only if this state has at least some permission to $x . f$, and $Q$ is well-defined with respect to a state if and only if $a$ is defined in this state. $P$ and $Q$ are always true when they verify. If we decide to define $P$ and $Q$ as being always true, then we cannot distinguish them from the semantic assertion true, which is well-defined in all states. If we decide to define $P$ and $Q$ to be true when they verify and false otherwise, then $P$ is indistinguishable from the assertion $\operatorname{perm}(x . f)>0$ (well-defined in all states) and $Q$ would be indistinguishable from an assertion such as $\operatorname{defined}(a)$. Such an assertion (defined(a)) does not exist in Viper, but it could exist. In this case, it would always be well-defined, and would be true in states where $a$ is defined in the store.

These two examples ( $P$ and $Q$ ) show the impossibility to distinguish different syntactic assertions when they are encoded into semantic assertions. This is an issue we want to tackle in the future, by using extended semantic assertions, which would be functions from states to some boolean (welldefined) or some special value signifying that the semantic assertion is not well-defined.

In our current model, to express when a semantic assertion is well-defined, we first need define the set of variables read by an assertion:

Definition 5.22 Set of variables read by an assertion

$$
\operatorname{readAssert}(P):=\cup_{i \in \operatorname{Inh}(P)} \sigma(i)
$$

This function captures the set of variables which have an effect in determining the value of an assertion. Intuitively, it corresponds to the minimum set of variables needed to syntactically define the assertion. We now define the three requirements we impose on the wellDefinedAssert function for semantic assertions as follows:

## Requirement 5.23 Well-defined assertion

wellDefinedAssert is a function which takes an assertion and a state as input, and returns a boolean. This function must satisfy the following properties, for all assertions $P$ and all states $\varphi$ and $\varphi^{\prime}$ :

1. wfAssert $(P) \Longrightarrow$ (wellDefinedAssert $(P, \varphi) \Leftrightarrow \operatorname{readAssert}(P) \subseteq \sigma(\varphi))$
2. wellDefinedAssert $(P, \varphi) \Longleftrightarrow$ wellDefinedAssert $(\neg P, \varphi)$
3. $P(\varphi) \Longrightarrow$ wellDefinedAssert $(P, \varphi)$

The first requirement says that, for well-formed assertions (that is supported and intuitionistic assertions, in particular annotations), this assertion is welldefined in a state if and only if the state defines all the variables it reads. The intuition is that well-formed semantic assertions represent self-framing syntactic assertions, and self-framing assertions cannot fail, by definition, because of permission. The issue has therefore to come from variables not defined. The second requirement simply follows a syntactic logic, in which if an assertion is well-defined, then its negation is also well-defined. The third requirement states that if a state satisfies an assertion, then this assertion is well-defined in this state. This interface and these requirements should be modified in the future with an extended form of semantic assertions.

We can now express the verification of inhale, exhale, and assume statements:

Definition 5.24 Verification of inhaling, exhaling, and assuming

$$
\begin{aligned}
\operatorname{ver}_{P r}(\{\varphi\} \text {, inhale } P) & \Longleftrightarrow \text { wellDefinedAssert }(P, \varphi) \\
\operatorname{ver}_{P r}(\{\varphi\} \text {, exhale } P) & \Longleftrightarrow \text { wellDefinedAssert }(P, \varphi) \wedge P(\varphi) \\
\operatorname{ver}_{P r}(\{\varphi\} \text {, assume } P) & \Longleftrightarrow \text { wellDefinedAssert }(P, \varphi)
\end{aligned}
$$

That is, these statements verify if and only if they are well-defined in the current state.

### 5.3.4 Minimal satisfying set of an assertion

We now introduce another concept before defining the semantics of inhale and exhale. As explained in Section 4.2, inhaling an assertion adds permission from accessibility predicates and assumes pure information. In the case of adding permission, this works similarly to adding a state to the initial state. In the case of assuming an assertion talking about pure resources, the execution is stopped if this assertion is not satisfied, and this is modeled using the partial definedness of the addition of our separation algebra. Inhaling, when verifying, corresponds to adding a set of states:

Definition 5.25 Let $P$ be an assertion. The minimal satisfying set of $P$ is defined as

$$
\operatorname{Inh}(P):=\left\{\varphi \in \Sigma \mid P(\varphi) \wedge\left(\forall \varphi^{\prime} \in \Sigma \cdot \varphi^{\prime} \ll \varphi \wedge \varphi^{\prime} \neq \varphi \Rightarrow \neg P\left(\varphi^{\prime}\right)\right)\right\}
$$

$\operatorname{Inh}(P)$ is the set of minima of the states satisfying $P$. Using this set, we define the semantics (when it verifies) of inhale as follows:

Definition 5.26 Semantics of inhale:

$$
\operatorname{sem}_{P r}(\{\varphi\}, \text { inhale } P):=\{\varphi\} \oplus \operatorname{Inh}(P)
$$

This works indeed as expected for modeling Viper. The addition of states adds the permissions. As an example, inhale acc(x.f, 1/2) adds half permission to the heap location $x$.f. If this new permission is strictly greater than 1 , this addition is not defined, it therefore results in an empty set, which models magic states in Viper. The same thing happen with assertions which deal with values, such as $x==5$ or $y . f==7$. These assertions act as filters: It they are satisfied by a state, then the addition is defined. However, if they are not satisfied, the addition is not defined, and thus also results in an empty state.

Example 5.27 Let $P:=\operatorname{acc}(x . f, 1 / 2)$ \&\& x.f $==5$. From the definition, $\operatorname{Inh}(P)$ is the set of all states whose stores only define $x$ (a reference), have half permission to $x . f$, and where $x . f=5$.

- If $\varphi$ has no permission to $x$.f, then $\{\varphi\} \oplus \operatorname{Inh}(P)$ results in a set with only one state, this state being the sum of $\varphi$ and the unique state of $\operatorname{Inh}(P)$ which defines the same $x$ as $\varphi$. Therefore, the resulting state has half permission to $x . f$, and $x . f=5$.
- If $\varphi$ has some permission to $x . f$, then it also carries a value for x.f.
- If this value is not 5 , then the sum of sets becomes the empty set (magic), since no element of $\operatorname{Inh}(P)$ is compatible with $\varphi$.
- If this value is 5 , then we have almost the same situation as in the first point, only in this case the resulting permission is $\frac{1}{2}$ more than before.
- Finally, if $\varphi$ has more than half permission to $x . f$, then $\varphi$ is not compatible with any element from Inh $(P)$, resulting once again in an empty set (magic).

As shown in Section 4.2, an exhale statement in Viper removes the permissions added by an inhale, but keeps the pure part. We define the semantics of exhale as follows:

Definition 5.28 Semantics of exhale

$$
\operatorname{sem}_{P r}(\{\varphi\}, \text { exhale } P):=\{|i| \oplus r \mid i \in \operatorname{Inh}(P) \wedge r \in \Sigma \wedge \varphi=i \oplus r\}
$$

It is not clear from the definition that this matches exhale in Viper. It actually does not have the same behavior in general, but it does in a special case, when an assertion is supported and intuitionistic. The situation is similar with inhale, which matches inhale in Viper in the case of an assertion supported and intuitionistic (well-formed).

Recall that self-framing assertions in Viper are encoded into semantic assertions which are both supported and intuitionistic, thus well-formed. The next lemma shows why this property is sufficient to have a correspondence between Viper and this framework when dealing with annotations.

## Lemma 5.29

$$
w f \operatorname{Assert}(P) \Longrightarrow(P(\varphi) \Leftrightarrow \exists!i \in \operatorname{Inh}(P) . i \ll \varphi)
$$

Proof Assume $w f$ Assert $(P)$, that is $P$ is intuitionistic and supported.
$\exists!i \in \operatorname{Inh}(P) . i \ll \varphi \Longrightarrow P(\varphi)$ follows from the definition of intuitionistic and the definition of $\operatorname{Inh}(P)$.
$P(\varphi) \Longrightarrow \exists!i \in \operatorname{Inh}(P) . i \ll \varphi$ follows from the definition of supported.
Using this lemma, we can now formalize the motivation from Section 4.2. That is, exhale verifies after inhale, and inhaling and exhaling the same assertion only deals with pure resources.

Lemma 5.30 Assume wfAssert $(P)$ and $\operatorname{ver}_{P r}(\{\varphi\}$,inhale $P)$. Then

1. $\operatorname{ver}_{P r}(\{\varphi\}$, inhale $P$; exhale $P)$
2. $\operatorname{sem}_{\operatorname{Pr}}(\{\varphi\}$,inhale $P$; exhale $P)=\{\varphi\} \oplus\{\mid i \| i \in \operatorname{Inh}(P)\}$

Proof We have $\operatorname{sem}_{\operatorname{Pr}}(\{\varphi\}$,inhale $P)=\{\varphi\} \oplus \operatorname{Inh}(P)$. Let $\varphi^{\prime} \in\{\varphi\} \oplus$ $\operatorname{Inh}(P)$, then $\exists i \in \operatorname{Inh}(P) \cdot \varphi^{\prime}=\varphi \oplus i$. Since $P$ is intuitionistic, we have $P\left(\varphi^{\prime}\right)$, thus we have wellDefinedAssert $\left(P, \varphi^{\prime}\right)$ from Requirement 5.23. Therefore, $\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}\right.$, exhale $\left.P\right)$, hence $\operatorname{ver}_{P r}(\{\varphi\}$, inhale $P$; exhale $P)$.

Moreover, using Lemma 5.29 , we have that there is a unique $i \in \operatorname{Inh}(P)$ such that $i \ll \varphi^{\prime}$. Hence,

$$
\begin{aligned}
\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \text { exhale } P\right) & =\left\{\left|i^{\prime}\right| \oplus r \mid i^{\prime} \in \operatorname{Inh}(P) \wedge r \in \Sigma \wedge \varphi^{\prime}=i^{\prime} \oplus r\right\} \\
& =\{|i| \oplus \varphi\} \\
& =\{\varphi\} \oplus\{|i|\}
\end{aligned}
$$

It follows that

$$
\operatorname{sem}_{P r}(\{\varphi\} \text {, inhale } P ; \text { exhale } P)=\{\varphi\} \oplus\{\mid i \| i \in \operatorname{Inh}(P)\}
$$

Finally, the correspondence between the semantics of inhale and exhale statements in Viper and the semantics of inhale and exhale in our language does not work in general when assertions are not annotations.

Example 5.31 Let $P:=\operatorname{acc}(x . f, \operatorname{perm}(x . f))$. $P$ is well-defined in all states. In Viper, inhale $P$ corresponds to doubling the permission held to $x . f$, and exhale $P$ removes all the permission to x.f. In our framework, on the other hand, $P$ is always true and well-defined, it therefore always verifies and inhale $P$ and exhale $P$ do nothing.

### 5.4 While loops

We define the semantics of while loops in this section. As explained in Section 2.2.4, the semantics of a loop involves havocing the variables modified by the body of the loop. We therefore first define which variables are modified by a statement, and then define the semantics of while loops.

## Definition 5.32 Variables modified by a statement

```
    \(\operatorname{modif}\left(s_{1} ; s_{2}\right)=\operatorname{modif}\left(s_{1}\right) \cup \operatorname{modif}\left(s_{2}\right)\)
\(\mid \operatorname{modif}\left(\mathbf{i f}(*)\left\{s_{1}\right\}\right.\) else \(\left.\left\{s_{2}\right\}\right)=\operatorname{modif}\left(s_{1}\right) \cup \operatorname{modif}\left(s_{2}\right)\)
\(\mid \operatorname{modif}(\vec{y}:=m(\vec{x}))=y\)
\(\mid \operatorname{modif}(\mathbf{w h i l e}(b) \operatorname{inv} I\{s\})=\operatorname{modif}(s)\)
\(\mid \operatorname{modif}(\) assume \(P)=\varnothing\)
\(\mid \operatorname{modif}(\) inhale \(P)=\varnothing\)
\(\mid \operatorname{modif}(\) exhale \(P)=\varnothing\)
\(\mid \operatorname{modif}(\operatorname{var} \vec{l})=\vec{l}\)
\(\mid \operatorname{modif}(\) havoc \(\vec{l})=\vec{l}\)
\(\mid \operatorname{modif}(\mathbf{s k i p})=\varnothing\)
\(\mid \operatorname{modif}(\mathbf{c u s t o m} o)=\operatorname{modifCustom}(o)\)
```

Using this function, we define the semantics of while loops as follows:
Definition 5.33 Semantics of while loops
Let $\vec{l}:=$ modif $(s) \cap \sigma(\varphi)$. Then

$$
\begin{aligned}
& \operatorname{ver}_{\operatorname{Pr}}(\{\varphi\}, \text { while }(b) \text { inv } I\{s\}) \\
& \Longleftrightarrow \operatorname{ver}_{\operatorname{Pr}}(\{|\varphi|\}, \text { havoc } \vec{l} ; \text { inhale } I ; \text { assume } b ; s ; \text { exhale } I) \\
& \wedge \operatorname{ver}_{\operatorname{Pr}}(\{\varphi\}, \text { exhale } I ; \text { havoc } \vec{l} \text {; inhale } I ; \text { assume } \neg b)
\end{aligned}
$$

and

$$
\begin{aligned}
& \operatorname{sem}_{P r}(\{\varphi\}, \text { while }(b) \text { inv } I\{s\}) \\
:= & \operatorname{sem}_{P r}(\{\varphi\}, \text { exhale } I ; \text { havoc } \vec{l} ; \text { inhale } I ; \text { assume } \neg b)
\end{aligned}
$$

The first part of the loop verification verifies if the invariant is an actual invariant. We begin with a state with no impure resources (the core), and then forget (havoc) the values of variables which can be modified by the loop. We then inhale the invariant and assume that we are in an iteration of the loop (b), execute the loop body $s$, and then make sure the state still satisfies the invariant. If this verifies, the possible states after the loop, represented by $\operatorname{sem}_{P r}(\varphi$, while $(b)$ inv $I\{s\})$, can be computed by exhaling the loop invariant, forgetting the values of the variables modified by the loop, and then inhaling the invariant as well as assuming we are out of the loop $(\neg b)$.

Remark 5.34 This definition defines $\vec{l}$ as modif $(s) \cap \sigma(\varphi)$, but the Isabelle formalization (Appendix A) defines it as modif(s) only. The Isabelle formalization should be modified with the definition presented here. Indeed, in the case where variables are declared inside the loop body, using $\vec{l}:=$ modif(s), any verification of this loop would fail. The variables would have both to be defined before the loop (to be havoced), and not to be defined inside the loop body (where they are declared).

### 5.5 Renaming interface

We have now defined the semantics for while loops, and want to define the semantics of method calls. When a method is called, the variables in the precondition and the postcondition of the method must be renamed. If the method is $(m, \overrightarrow{\operatorname{args}}, \overrightarrow{\text { rets }}, P, Q, s)$, and the method call is $\vec{y}:=m(\vec{x})$, we need to rename the variables of the precondition $P$, which only speaks about $\overrightarrow{\operatorname{args}}$. We also need to rename the variables of the postcondition $Q, \xrightarrow{\text { which only }}$ speaks about $\overrightarrow{\operatorname{args}}$ and $\overrightarrow{\text { rets. }} . \overrightarrow{\operatorname{args}}$ should be renamed to $\vec{x}$, and $\overrightarrow{\text { rets }}$ to $\vec{y}$.

On top of this, inlining a method call requires renaming the method body. As we explain in Chapter 6, we need to make sure variables declared in
the method body do not interfere with variables already defined elsewhere, to avoid variable capturing. This is the reason why we require a powerful renaming interface, which is able to both rename $\overrightarrow{\operatorname{args}}$ to $\vec{x}$ and $\overrightarrow{\operatorname{rets}}$ to $\vec{y}$, but which can also avoid a list of variables already defined elsewhere.

The renaming functions we present in this section are built on a single renaming function, which renames one element (a variable name) to another one. This renaming function, on top of the element to rename, takes as a parameter a renaming quadruple:

Definition 5.35 A renaming quadruple is a quadruple of lists of variables

$$
(\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{a v o i d}, \overrightarrow{\text { domain }})
$$

where

- $\overrightarrow{\text { old }}$ is a list of variables which should be renamed to variables from $\overrightarrow{n e w}$.
- $\overrightarrow{\text { avoid }}$ is a list of variables to avoid.
- $\overrightarrow{\text { domain }}$ is the list of variables on which this renaming function should be applied.

The idea of this definition is as follows. When calling a method, we need to rename its precondition and postcondition. As an example, take the method $(m, \overrightarrow{a r g s}, \overrightarrow{r e t s}, P, Q, s)$, and take the method call $\vec{y}:=m(\vec{x})$. We want to rename $P$ and $Q$, which speak about $\overrightarrow{\operatorname{args}}$ and $\overrightarrow{\text { rets, to speak about } \vec{x} \text { and }}$ $\vec{y}$. In this case, $\overrightarrow{o l d}$ would be the concatenation of $\overrightarrow{\operatorname{args}}$ and $\overrightarrow{\text { rets }}$, where $\overrightarrow{\text { new }}$ would be the concatenation of $\vec{x}$ and $\vec{y}$.
$\overrightarrow{\text { avoid }}$ and $\overrightarrow{\text { domain }}$ are useful for inlining. When we inline a method call, we want to rename variables declared in the method body with names of variables which are not defined in the rest of the program, to avoid interferences with the calling context (i.e., to avoid variable capturing). $\overrightarrow{a v o i d}$ is the list of variables defined elsewhere which should be renamed to something else. Finally, we want to be able to invert the renaming, to easily show properties about renamed statements. This is why we need a $\overrightarrow{\text { domain }}$ list. All variables in $\overrightarrow{\text { domain }}$ are renamed to variables which are not in $\overrightarrow{\text { avoid. However, to }}$ make this renaming invertible, we need to rename some variables to variables in $\overrightarrow{\text { avoid }}$, to make the renaming surjective. We therefore get the avoid property only on $\overrightarrow{\text { domain }}$, thus $\overrightarrow{\text { domain }}$ should represent the variables which are renamed.

To get these properties, we first impose some conditions on a renaming quadruple:
Definition 5.36 A renaming quadruple $(\overrightarrow{\text { old }}, \overrightarrow{\text { new }}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }})$ is well-formed, denoted by wfRenaming $((\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }}))$, if and only if

1. length $(\overrightarrow{\text { old }})=$ length $(\overrightarrow{n e w})$.
2. distinct $(\overrightarrow{o l d})$ (elements of old are all distinct).
3. $\operatorname{distinct}(\overrightarrow{n e w})$ (elements of $\overrightarrow{n e w}$ are all distinct).

We formally express the required properties as follows:

## Requirement 5.37 Rename an element, invert a renaming quadruple

Let $t:=(\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }})$, and assume wfRenaming $(t)$. Then

1. wfRenaming (renameInv $(t))$
2. $\operatorname{rename}(\operatorname{renameInv}(t), \operatorname{rename}(x, t))=x$
3. $\forall i<$ length $(\overrightarrow{o l d}) \cdot \operatorname{rename}(t, \overrightarrow{o l d}[i])=\overrightarrow{\text { new }}[i]$
4. $x \in \overrightarrow{\text { domain }}-\overrightarrow{\text { old }} \Longrightarrow$ rename $(t, x) \notin \overrightarrow{\text { avoid }}$

The first one means that the inverse of a renaming quadruple is a wellformed renaming quadruple. The second one means that the inverse of a renaming quadruple inverts the renaming. The third one says that the renaming function actually renames $\overrightarrow{o l d}$ variables to $\overrightarrow{n e w}$ variables. Finally, the last one shows that variables from the $\overrightarrow{\text { domain }}$ are renamed in something which is not in $\overrightarrow{\text { avoid }}$. These are the core functions and properties we need.

We now impose requirements on how to rename states, leveraging the properties of rename:

Requirement 5.38 Assume wfRenaming $(t)$. Then

1. $\sigma($ renameState $(t, \varphi))=\{$ rename $(t, x) \mid x \in \sigma(\varphi)\}$
2. renameState $\left(t, \varphi_{1} \oplus \varphi_{2}\right)=$ renameState $\left(t, \varphi_{1}\right) \oplus \operatorname{renameState}\left(t, \varphi_{2}\right)$
3. $(\forall x \in \sigma(\varphi) \cdot \operatorname{rename}(t, x)=x) \Longrightarrow$ renameState $(t, \varphi)=\varphi$
4. wfRenaming $\left(t_{1}\right) \wedge$ wfRenaming $\left(t_{2}\right) \wedge$

$$
\left(\forall x \in \sigma(\varphi) \cdot \operatorname{rename}(t, x)=\operatorname{rename}\left(t_{2}, \operatorname{rename}\left(t_{1}, x\right)\right)\right)
$$

$$
\Longrightarrow \text { renameState }(t, \varphi)=\text { renameState }\left(t_{2},\left(\text { renameState }\left(t_{1}, \varphi\right)\right)\right)
$$

These requirements follow a simple definition of renaming states, which just renames the variables defined in the stores. The first requirement is about the store of a renamed state. The second one shows that renaming a sum of states is equivalent to adding the renamed states. The third one says that, if all variables are left unchanged by the renaming function, then the state is left unchanged. Finally, the last one says that if the composition of two renaming functions is the same as one on the variables defined by $\varphi$, then it is also the same at the state level.

We now define how to rename semantic assertions, based on renaming states:

## Definition 5.39 Rename assertions

$$
\text { rename } A(t, P)(\varphi):=P(\text { renameState }(\text { renameInv }(t), \varphi))
$$

Renaming a semantic assertion is the same as applying it to the state renamed in the inverse direction. As an example, take the state $\varphi_{y}$ which only defines $y$ and the assertion $P:=(x==5)$. Let $t:=([x],[y],[],[])$. Renaming the assertion with $t$ gives $y==5$, which is the same as applying $P$ to $\varphi_{y}$ renamed with the invert of $t$ : In this case, renameState (renameInv $\left.(t), \varphi_{y}\right)$ only defines $x$.

We impose two more requirements for renaming assertions:

Requirement 5.40 Assume wfRenaming $(t)$. Then

1. wellDefinedAssert $(P, \varphi)$
$\Longrightarrow$ wellDefinedAssert $($ rename $A(t, P)$, renameState $(t, \varphi))$
2. $\operatorname{wfAssert}(P, \varphi) \Longrightarrow$ wfAssert(rename $A(t, P)$, renameState $(t, \varphi))$

That is, renaming an assertion keeps the assertion well-defined with respect to a state (but renamed), and well-formed (namely supported and intuitionistic).

We leverage these renaming functions for elements, states and assertions to define a function to rename statements. We first define how to rename a list of variable names:

## Definition 5.41 Rename a list of variable names

$$
\forall i<\operatorname{length}(\vec{l}) \cdot \operatorname{renameList}(t, \vec{l})[i]=\operatorname{rename}(t, \vec{l}[i])
$$

This simply means renaming every element of the list. We can now define the renameS function to rename statements:

## Definition 5.42 Rename a statement

```
renameS \((t\), inhale \(P)=\) inhale rename \(A(t, P)\)
\(\mid\) rename \(S(t\), exhale \(P)=\) exhale rename \(A(t, P)\)
\(\mid \operatorname{renameS}\left(t, s_{1} ; s_{2}\right)=\operatorname{rename} S\left(t, s_{1}\right) ; \operatorname{renameS}\left(t, s_{2}\right)\)
\(\mid \operatorname{renameS}(t, \operatorname{var} \vec{l})=\mathbf{v a r} \operatorname{renameList}(t, \vec{l})\)
\(\mid \operatorname{renameS}(t\), havoc \(\vec{l})=\mathbf{h a v o c} \operatorname{renameList}(t, \vec{l})\)
\(\mid \operatorname{rename} S\left(t\right.\), if \((*)\left\{s_{1}\right\}\) else \(\left.\left\{s_{2}\right\}\right)=\) if \((*)\left\{\right.\) rename \(\left.S\left(t, s_{1}\right)\right\}\) else \(\left\{\operatorname{rename} S\left(t, s_{2}\right)\right\}\)
\(\mid\) renameS \((t\), while \((b)\) inv \(I\{s\})=\)
while (rename \(A(t, b))\) inv rename \(A(t, I)\{\operatorname{rename} S(t, s)\}\)
\(\mid \operatorname{renameS}(t, \vec{y}:=m(\vec{x}))=\operatorname{renameList}(t, \vec{y}):=m(\operatorname{renameList}(t, \vec{x}))\)
|rename \(S(t\), assume \(b)=\) assume rename \(A(t, b)\)
\(\mid\) renameS \((t\), custom \(o)=\) custom renameCustom \((t, o)\)
\(\mid\) rename \(S(t\), skip \()=\mathbf{s k i p}\)
```

This renaming function is useful for defining the inlining function in Chapter 6. Requirements on renameCustom are given in Section 5.7.

### 5.6 Method calls

Using our function to rename semantic assertions, we can now define the semantics of method calls:

Definition 5.43 Semantics of method calls
Assume $(m, \overrightarrow{a r g s}, \overrightarrow{r e t s}, P, Q, s) \in \operatorname{Pr}$.
Let $t:=(\overrightarrow{\operatorname{args}}++\overrightarrow{\operatorname{rets}}, \vec{x}++\vec{y}, \varnothing, \varnothing)$ (where ++ denotes concatenation), $P^{\prime}:=$ $\operatorname{rename} A(t, P)$ and $Q^{\prime}:=\operatorname{rename} A(t, Q)$. Then

$$
\begin{aligned}
& \operatorname{ver}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x})) \\
\Longleftrightarrow & \operatorname{ver}_{P r}\left(\{\varphi\}, \text { exhale } P^{\prime} ; \text { havoc } \vec{y} ; \text { inhale } Q^{\prime}\right) \wedge \vec{x} \cup \vec{y} \subseteq \sigma(\varphi)
\end{aligned}
$$

and

$$
\operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x})):=\operatorname{sem}_{P r}\left(\{\varphi\}, \text { exhale } P^{\prime} ; \text { havoc } \vec{y} ; \text { inhale } Q^{\prime}\right)
$$

To verify, the state $\varphi$ needs to at least define the variables from $\vec{x}$ and $\vec{y}$. The verification and behavior of the method call is then equivalent to exhaling the renamed precondition, havocing the return variables, and inhaling the renamed postcondition.

### 5.7 Rest of the semantics

We now proceed to complete the definition of the semantics, by first defining the semantics of control structures, and then defining the requirements on custom statements.

### 5.7.1 Control structures

A state enters both branches of a non-deterministic if, hence the following semantics:

Definition 5.44 Semantics of non-deterministic if

$$
\begin{gathered}
\operatorname{ver}_{P r}\left(\{\varphi\} \text {, if }(*)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\}\right) \Longleftrightarrow \operatorname{ver}_{P r}\left(\{\varphi\}, s_{1}\right) \wedge \operatorname{ver}_{P r}\left(\{\varphi\}, s_{2}\right) \\
\operatorname{sem}_{P r}\left(\{\varphi\}, \text { if }(*)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\}\right):=\operatorname{sem}_{P r}\left(\{\varphi\}, s_{1}\right) \cup \operatorname{sem} \operatorname{sem}_{P r}\left(\{\varphi\}, s_{2}\right)
\end{gathered}
$$

The sequential composition applies $s_{1}$ to a state, and then $s_{2}$ to the resulting states:

## Definition 5.45 Semantics of sequential composition

$$
\begin{gathered}
\operatorname{ver}_{P r}\left(\{\varphi\}, s_{1} ; s_{2}\right) \Longleftrightarrow \operatorname{ver}_{P r}\left(\{\varphi\}, s_{1}\right) \wedge \operatorname{ver}_{P r}\left(\operatorname{sem}_{P r}\left(\{\varphi\}, s_{1}\right), s_{2}\right) \\
\operatorname{sem}_{P r}\left(\{\varphi\}, s_{1} ; s_{2}\right):=\operatorname{sem}_{P r}\left(\operatorname{sem}_{P r}\left(\{\varphi\}, s_{1}\right), s_{2}\right)
\end{gathered}
$$

The behaviour of skip is standard:
Definition 5.46 Semantics of skip

$$
\begin{gathered}
\operatorname{ver}_{P r}(\{\varphi\}, \mathbf{s k i p}) \Longleftrightarrow \top \\
\operatorname{sem}_{P r}(\{\varphi\}, \mathbf{s k i p}):=\{\varphi\}
\end{gathered}
$$

### 5.7.2 Custom interface

Finally, we impose some requirements on the behavior of custom statements.

## Requirement 5.47 General requirements for custom statements

1. $w_{f C u s t o m}^{P r}(0) \wedge \varphi^{\prime} \in$ semanticsCustom $_{P_{r}}(\varphi, o)$
$\Longrightarrow \sigma(\varphi) \subseteq \sigma\left(\varphi^{\prime}\right) \wedge \sigma\left(\varphi^{\prime}\right) \subseteq \sigma(\varphi) \cup$ modifCustom $(o)$
2. modifCustom $(0) \subseteq$ readCustom ( $o$ )

The first general requirement states that a custom statement cannot undefine variables, and that it modifies the variables it defines. The second one imposes that modifying a variable counts as reading a variable.

We then impose requirements on the consequences of renaming custom statements:

## Requirement 5.48 Rename requirements for custom statements

Assume wfRenaming $(t)$. Then

1. $\operatorname{readCustom}($ renameCustom $(t, o))=\{$ rename $(t, x) \mid x \in \operatorname{readCustom}(o)\}$
2. modifCustom(renameCustom $(t, o))=\{$ rename $(t, x) \mid x \in \operatorname{modifCustom}(o)\}$
3. renameCustom $($ renameInv $(t)$,renameCustom $(t, o))=o$
4. semanticsCustom $_{P r}(\varphi, o) \neq$ Error $^{\prime}$ $\Longrightarrow \operatorname{semanticsCustom}_{\text {Pr }}($ renameState $(t, \varphi)$, renameCustom $(t, o)) \neq$ Error
5. $\varphi^{\prime} \in \operatorname{semanticsCustom}_{P r}($ renameState $(t, \varphi)$, renameCustom $(t, o))$
$\Longrightarrow$ renameState (renameInv $\left.(t), \varphi^{\prime}\right) \in \operatorname{semanticsCustom}_{P r}(\varphi, o)$
6. wfCustom $_{P_{r}}(o) \Longrightarrow$ wfCustom $_{\text {Pr }_{r}}$ (renameCustom $\left.(t, o)\right)$

The first two requirements describe the variables read and modified by a renamed statement. The third one imposes that we can invert the renaming. The fourth and fifth ones describe how to relate the semantics of a renamed statement to the semantics of the original statement. The last requirement says that renaming a well-formed statement results in a well-formed statement.

### 5.8 Well-definedness and verification of a program

We have presented the semantics and the requirements on the parameters of the framework. These semantics are defined with respect to a program, and are modular. For example, when verifying a method call, we assume the method verifies on its own. This motivates the following definition expressing what it means for a complete program to verify:

Definition 5.49 Verification of a program

$$
\begin{aligned}
& \operatorname{verProg}(\operatorname{Pr}) \\
\Longleftrightarrow & \forall(m, \overrightarrow{\operatorname{args}}, \overrightarrow{\operatorname{rets}}, P, Q, s) \in \operatorname{Pr} \operatorname{ver}_{P r}(h(\overrightarrow{\operatorname{args}}) \oplus h(\overrightarrow{\operatorname{rets}}) \oplus \operatorname{Inh}(P), s ; \text { exhale } Q)
\end{aligned}
$$

A program verifies if all of its methods verify, assuming that all other methods verify, since method calls are approximated by their contracts.

Remark 5.50 The verification of a method, as written in the previous definition, is equivalent to verifying the method after $\overrightarrow{\operatorname{args}}$ and $\overrightarrow{\text { rets }}$ have been defined, beginning with a state which satisfies the precondition:

$$
\begin{aligned}
& \operatorname{ver}_{P r}(h(\overrightarrow{\operatorname{args}}) \oplus h(\overrightarrow{\operatorname{ret}}) \oplus \operatorname{Inh}(P), s ; \text { exhale } Q) \\
\Longleftrightarrow & \operatorname{ver}_{P r}\left(\{u\}, \operatorname{var} \overrightarrow{\operatorname{args}} ; \text { var } \overrightarrow{\operatorname{rets}^{*}} ; \text { inhale } P ; s ; \text { exhale } Q\right)
\end{aligned}
$$

However, we can only really speak about the verification of a program when it is well-formed, that is when all of its statements are well-formed. We define these notions as follows:

## Definition 5.51 Well-formed statements

```
    \(w f S t m t_{P r}\left(s_{1} ; s_{2}\right) \Longleftrightarrow w f S t m t_{P r}\left(s_{1}\right) \wedge w f S t m t_{P r}\left(s_{2}\right)\)
```




```
\(\mid w f S t m t_{P r}(\vec{y}:=m(\vec{x})) \Longleftrightarrow(\exists \overrightarrow{\text { args }}, \overrightarrow{r e t s}, P, Q, s .(m, \overrightarrow{\text { args }}, \overrightarrow{r e t s}, P, Q, s) \in \operatorname{Pr}\)
    \(\wedge \operatorname{length}(\vec{x})=\) length \((\overrightarrow{\operatorname{args}}) \wedge\) length \((\vec{y})=\) length \((\overrightarrow{\text { rets }})\)
    \(\wedge\) elements of \(\vec{x}\) and \(\vec{y}\) are distinct)
\(\mid w f S t m t_{P r}(\) inhale \(P) \Longleftrightarrow \operatorname{supported}^{(P)}\)
\(\mid w f S t m t_{P r}(\) exhale \(P) \Longleftrightarrow \operatorname{supported}^{(P)}\)
\(\mid w f S t m t_{P r}(\) custom \(o) \Longleftrightarrow w f C u s t o m ~_{P r}(o)\)
\(\mid w^{\prime}\) Stm \(_{P r}(\mathbf{s k i p}) \Longleftrightarrow \top\)
```

A while loop is well-formed if and only if its invariant is well-formed (intuitionistic and supported) and its body is well-formed. No condition is imposed on the loop guard $b$. A method call is well-formed if the numbers of arguments and return variables provided match the definition of the method. Moreover, for simplicity, we do not allow method calls such as $\left[x_{1}, x_{2}\right]:=m\left(\left[x_{1}, x_{1}, x_{2}, x_{2}\right]\right)$. Indeed, all variables in a method call have to be distinct (this is not the case in Viper, where only return variables have to be distinct). Finally, given our definitions of inhale and exhale semantics, we want to use only supported assertions. The reason is that the minimal satisfying set of an assertion which is not supported does not have the same meaning. Inhaling general assertions (such as $x . f==5$ ) can be encoded via custom statements.

We finally define what it means for a program to be well-formed:
Definition 5.52 Well-defined methods and programs
$A$ method ( $m, \overrightarrow{a r g s}, \overrightarrow{r e t s}, P, Q, s$ ) is well-formed, written

$$
\text { wfothethod }_{P r}((m, \overrightarrow{\operatorname{args},} \overrightarrow{\text { rets }}, P, Q, s))
$$

if and only if all the following properties are satisfied:

1. All elements from $\overrightarrow{\text { args }}$ and $\overrightarrow{\text { rets }}$ are distinct.
2. The precondition is well-formed: wfAssert $(P)$.
3. The precondition only deals with arguments: readAssert $(P) \subseteq \overrightarrow{\operatorname{args}}$.
4. The postcondition is well-formed: wfAssert( $Q$ )
5. The postcondition only deals with arguments and return variables: readAssert $(Q) \subseteq$ $\overrightarrow{\operatorname{args}} \cup \overrightarrow{\text { rets }}$.
6. The arguments are not modified: $\overrightarrow{\operatorname{args}} \cap \operatorname{modif}(s)=\varnothing$.
7. The method body is well-formed: wfStmt $\operatorname{Pr}_{P r}(s)$.

A well-formed program is defined as follows:

$$
\left.\operatorname{wffProg}(\operatorname{Pr}) \Longleftrightarrow\left(\forall \text { method } \in \text { Pr. }_{\text {wfMethod }}^{\text {Pr }} \text { ( method }\right)\right)
$$

We have now finished the definition of our framework. Given a separation algebra ( $\Sigma, \oplus, u, C$ ), a type $V$ of variable names, a type $O$ to encode custom statements, and four interfaces (store, assertions, rename and custom) which satisfy some requirements, we have defined a language with clearly defined semantics. We can now go back to the core of this thesis, inlining, express a soundness condition, and finally prove soundness of inlining in this framework.

## Chapter 6

## Soundness of Static Inlining

The previous two chapters define the framework we work with. This chapter deals with the soundness of inlining in this framework, as sketched in Chapter 3. We first formally define the mono and framing properties (introduced in terms of Viper in Chapter 3). We then define inlining up to a bound and we express the soundness condition (i.e., the condition under which soundness of inlining holds). The last four sections deal with the proof of the soundness. Section 6.3 describes the soundness theorem, as well as the soundness invariant we use to prove this theorem (by induction). We only present the method call case and the loop case of the proof by induction, respectively in Section 6.4 and Section 6.5, since they are the two most interesting proofs. Everything we present in this chapter has been mechanized and proved with the proof assistant Isabelle/HOL [16] (see Appendix A).

### 6.1 Mono and framing

As sketched in chapter 3, mono is the combination of safeMono and monoOut. These concepts are easy to define in our separation algebra:

Definition 6.1 safeMono, monoOut and mono

$$
\begin{aligned}
\operatorname{safeMono}_{P r}(s) & \Longleftrightarrow\left(\forall A, B \subseteq \Sigma .\left(B \gg A \wedge \operatorname{ver}_{P r}(A, s)\right) \Rightarrow \operatorname{ver}_{P r}(B, s)\right) \\
\text { monoOut }_{P r}(s) & \Longleftrightarrow\left(\forall A, B \subseteq \Sigma .\left(B \gg A \wedge \operatorname{ver}_{P r}(A, s) \wedge \operatorname{ver}_{P r}(B, s)\right)\right. \\
& \left.\Longrightarrow \operatorname{sem}_{P r}(B, s) \gg \operatorname{sem}_{P r}(A, s)\right) \\
\text { mono }_{P r}(s) & \Longleftrightarrow \operatorname{safeMono~}_{P r}(s) \wedge \text { monoOut }_{P r}(s)
\end{aligned}
$$

safeMono simply expresses that the function $\lambda A$.ver $P_{r}(A, s)$ is non-decreasing, ${ }^{1}$ while monoO ut expresses that the function $\lambda A \cdot \operatorname{sem}_{\operatorname{Pr}}(A, s)$ is non-decreasing ${ }^{2}$

[^8]$\left(\lambda A . \operatorname{semPr}(A, s)\right.$ is defined on $A$ when $\left.\operatorname{ver}_{P r}(A, s)\right)$.
We defined framing as respecting the frame rule. That is, when we have a state which verifies with a framing statement, if we add another state which does not interfere with the statement, then we get at least this other state afterwards. This is expressed formally as follows:

## Definition 6.2 framing

$$
\begin{aligned}
\text { framing }_{P r}(s) \Longleftrightarrow \text { mono }_{P r}(s) \wedge\left(\forall \varphi, r \in \Sigma . \operatorname{ver}_{P r}(\{\varphi\}, s) \wedge \operatorname{modif}(s) \cap \sigma(r)=\varnothing\right. \\
\left.\Longrightarrow \operatorname{sem}_{P r}(\{\varphi \oplus r\}, s) \gg \operatorname{sem}_{P r}(\{\varphi\}, s) \oplus\{r\}\right)
\end{aligned}
$$

A framing statement is mono. The non-interference condition, saying that $r$ does not define any variable which is modified by $s$, is expressed as modif $(s) \cap \sigma(r)=\varnothing$. Then, for any two states $\varphi$ and $r$ such that $s$ verifies with $\varphi$ and $r$ does not interfere with $s$, executing $s$ from the state $\varphi \oplus r$ results in a set of states stronger than what we would get by adding $r$ to all ${ }^{3}$ output states when $s$ is executed from the state $\varphi$.

### 6.2 Formalization of soundness

Now that we have expressed mono and framing, we can formally express the soundness condition, sketched in Chapter 3. We first define formally what it means to inline up to a bound.

### 6.2.1 Inlining

We first define inlining for all cases but method calls, which are a bit more complicated. The inline function $i n l$ is parametrized by a program $\operatorname{Pr}$ (to handle method calls), and a depth bound $n$. It takes as input the statement $s$ to inline, but also a list of variable names $\vec{l}$ which are already defined elsewhere in the inlined program, and have thus to be avoided when inlining method calls, to avoid interferences (see Definition 6.7).

[^9]Definition 6.3 Inline (except one method call case)

$$
\begin{aligned}
& i n l_{P r}^{0}(\vec{l}, \vec{y}:=m(\vec{x})) \quad:=\text { assume } \perp \\
& \operatorname{in} l_{P r}^{0}(\vec{l} \text {, while }(b) \text { inv } I\{s\}):=\text { assume } \neg b \\
& \operatorname{inl}_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x})) \quad:=\ldots \text { (see below) } \\
& \operatorname{inl} l_{P r}^{n+1}(\vec{l} \text {, while }(b) \operatorname{inv} I\{s\}):=\mathbf{i f}(b)\left\{\operatorname{inl} l_{P r}^{n}(\vec{l}, s) ; \operatorname{in} l_{P r}^{n}(\vec{l} \text {, while }(b) \operatorname{inv} I\{s\})\right\} \\
& \operatorname{inl} l_{P r}^{n}\left(\vec{l}, s_{1} ; s_{2}\right) \quad:=\operatorname{inl}_{P r}^{n}\left(\vec{l}, s_{1}\right) ; \operatorname{inl} l_{P r}^{n}\left(\vec{l}, s_{2}\right) \\
& i n l_{P r}^{n}\left(\vec{l}, \mathbf{i f}(*)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\}\right):=\mathbf{i f}(*)\left\{\text { inl } l_{P r}^{n}\left(\vec{l}, s_{1}\right)\right\} \text { else }\left\{i n l_{P r}^{n}\left(\vec{l}, s_{2}\right)\right\} \\
& \text { inl } l_{P r}^{n}(\vec{l}, s) \quad:=s \quad \text { (otherwise) }
\end{aligned}
$$

We begin the inlining with a bound $n$. Everytime we inline a loop or a method call (see below), we decrease this bound by 1 and recursively call the inline function. When we have already inlined up to a depth of $n$, we do not go further. For method calls, we simply stop the execution by assuming $\perp$. In the case of loops, we simply assume $\neg b$, which is equivalent to assuming $b \Rightarrow \perp$. Indeed, if $b$ is true, then we would have had to go on with the loop, but otherwise we are out of the loop.

To inline a loop (when we have not reached the depth bound yet), we use the deterministic if construct (which, as explained in Definition 5.3, can be encoded using the non-deterministic $\mathbf{i f}(*)$ ). If $b$ is true, then we enter the loop and we first inline one iteration of the loop (i.e., the loop body). After this inlined iteration, we are still in the loop, so we inline the loop (but with the depth decreased by one). If $b$ is false, then we do nothing. Finally, inlining sequential compositions and conditional branchings is straightforward.

Remark 6.4 The definition of inl $l_{P r}^{n}\left(\vec{l}, s_{1} ; s_{2}\right)$ uses the same list of variable names $\vec{l}$ for the two recursive calls (inlining of $s_{1}$ and $s_{2}$ ), instead of recording the newly defined variables in inl $l_{P r}^{n}\left(\vec{l}, s_{1}\right)$. Variable names can therefore collide and variables can be captured. This will be modified in the future.

We now need to define the inlining of a method call when the depth bound has not been reached. This is also related to the list of variables used by the inlining function. To define this inlining, we first need to define what we mean by "variables read by a statement":

Definition 6.5 Variables read by a statement

```
    \(\operatorname{read}\left(s_{1} ; s_{2}\right)=\operatorname{read}\left(s_{1}\right) \cup \operatorname{read}\left(s_{2}\right)\)
\(\mid \operatorname{read}\left(\mathbf{i f}(*)\left\{s_{1}\right\}\right.\) else \(\left.\left\{s_{2}\right\}\right)=\operatorname{read}\left(s_{1}\right) \cup \operatorname{read}\left(s_{2}\right)\)
\(\mid \operatorname{read}(\vec{y}:=m(\vec{x}))=\vec{x} \cup \vec{y}\)
\(\mid \operatorname{read}(\mathbf{w h i l e}(b) \operatorname{inv} I\{s\})=\operatorname{read}(s) \cup \operatorname{readAssert}(b) \cup \operatorname{readAssert}(I)\)
\(\mid \operatorname{read}(\) assume \(P)=\operatorname{readAssert}(P)\)
\(\mid \operatorname{read}(\) inhale \(P)=\operatorname{readAssert}(P)\)
\(\mid \operatorname{read}(\mathbf{e x h a l e} P)=\operatorname{readAssert}(P)\)
\(\mid \operatorname{read}(\operatorname{var} \vec{l})=\vec{l}\)
\(\mid \operatorname{read}(\boldsymbol{\text { havoc }} \vec{l})=\vec{l}\)
\(\mid \operatorname{read}(\) skip \()=\varnothing\)
\(\mid \operatorname{read}(\) custom \(o)=\operatorname{readCustom}(o)\)
```

This definition is similar to the definition of the modif function (Definition 5.32), with some differences. Since we care about all variables which are read, we include all variables read by assertions, as well as variables passed as arguments when calling a method. Since we imposed as a requirement that readCustom $(o) \subseteq$ modifCustom ( $o$ ), we can show the following lemma:

Lemma 6.6 Every variable that is modified is also read

$$
\operatorname{modif}(s) \subseteq \operatorname{read}(s)
$$

Using this read function, we can now express how we inline method calls (we use ++ to denote the concatenation of two lists):

## Definition 6.7 Inlining of method calls

First case: The method is defined, i.e., there exists $(m, \overrightarrow{\operatorname{args}}, \overrightarrow{r e t s}, P, Q, s) \in P r$.
Let $s^{\prime}=\operatorname{rename}((\overrightarrow{\operatorname{args}}++\overrightarrow{\operatorname{rets}}, \vec{x}++\vec{y}, \vec{l}, \operatorname{read}(s)), s)$, and $\overrightarrow{l^{\prime}}=\vec{l} \cup \operatorname{read}\left(s^{\prime}\right)$.
Then

$$
i n l_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x})):=\operatorname{inl} l_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)
$$

Second case: The method is undefined.

$$
\operatorname{inl}_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x})):=\text { exhale } \perp
$$

In the case of a method call not well-formed (if the method called is not defined), we simply exhale $\perp$, which fails with any state. It does not really matter, since the soundness we prove (see Section 6.3) assumes well-formedness, which in particular means that every method call is well-formed, that is the method which is called is defined. When the method call is well-formed, we

```
    Listing 6.1: Loop condition not mono.
var b: Bool := false
while (perm(x.f) <= 1/2) { 2
    b := true
}
assert b
    ...
```

Listing 6.2: Negated loop condition not mono.

```
Listing 6.2: Negated loop condition not mono.
    var b : Bool := true
    var b : Bool := true
    while (perm(x.f) >= 1/2) \{
    while (perm(x.f) >= 1/2) \{
        b := false
        b := false
\}
```

\}

```
```

        ...
    ```
        ...
assert b
```

assert b

```

Figure 6.1: Loops with conditions not well-formed monotonic
take the body of the method, rename it by modifying the arguments and the return variables using the ones provided in the method calls.

The meaning of the renaming quadruple \((\overrightarrow{\operatorname{args}}++\overrightarrow{r e t s}, \vec{x}++\vec{y}, \vec{l}, \operatorname{read}(s))\) is the following:
1. \(\overrightarrow{\operatorname{args}}\) variables are renamed to \(\vec{x}\) variables.
2. \(\overrightarrow{\text { rets }}\) variables are renamed to \(\vec{y}\) variables.
3. \(\vec{l}\) variables are avoided: If a variable has a name in \(\vec{l}\), then it is renamed to a variable with a name not in \(\vec{l}\).
4. Only variables from read(s) will be renamed by the functions using this quadruple. Therefore, the property to avoid variables is satisfied for these variables.

We also need to record the new variables defined by the renamed method body, to not capture them while inlining deeper. This is the meaning of \(\overrightarrow{l^{\prime}}=\vec{l} \cup \operatorname{read}\left(s^{\prime}\right)\) : We add the variables read by \(s^{\prime}\) to \(\overrightarrow{l^{\prime}}\). This new list \(\overrightarrow{l^{\prime}}\) is then used in the recursive call to the inline function, for inlining the renamed method body.

\subsection*{6.2.2 Soundness condition}

The sketch of the soundness described in Chapter 3 said the following:
1. Method and loop bodies have to be framing.
2. Non-inlinable statements, that is statements which do not contain method calls and loops, have to be mono.

We need to formally clarify several things. First, we only looked at examples of method calls. In the case of loops, we also have a loop condition \(b\). This condition cannot be any assertion.

Indeed, we show in Figure 6.1 two examples \({ }^{4}\) where all statements (without considering loop conditions) are framing, but we do not have soundness. Listing 6.1 is not sound for inlining, since a state with no permission to \(x\).f verifies, but a state with full permission does not. In this case, assume perm(x.f) <= \(1 / 2\) is not mono. Listing 6.2 shows another example where inlining is unsound. In this other case, assume ! (perm(x.f >= 1/2)) is not mono. Formally, the loop condition has to satisfy the following property:

Definition 6.8 Well-formed monotonic assertions
\[
w f m_{P r}(b) \Longleftrightarrow \text { mono }_{P r}(\text { assume } b) \wedge \text { mono }_{P r}(\text { assume } \neg b)
\]

It should be clear from the examples why we require this. What this means is that, when an assertion is well-defined in a state, then it will have the same value (true or false) for any stronger state. Another reason is that, when we inline a loop iteration, we use a deterministic if, which is translated to a non-deterministic if using assume \(b\) and assume \(\neg b\). mono \(_{P_{r}(\text { assume }} b\) ) (and especially monoOut) ensures that a stronger state will enter the loop if a weaker state enters the loop, whereas mono \({ }_{P r}(\) assume \(\neg b\) ) (also especially monoOut) ensures that a stronger state will avoid the loop if a weaker state avoids the loop.

Moreover, we informally speak about non-inlinable statements. What we mean is statements which do not contain any loop nor any method call. Formally:

\section*{Definition 6.9 Inlinable}
\[
\begin{aligned}
\text { inlinable }(\vec{y}:=m(\vec{x})) & \Longleftrightarrow \top \\
\text { inlinable }(\text { while }(b) \text { inv } I\{s\}) & \Longleftrightarrow \top \\
\text { inlinable }\left(s_{1} ; s_{2}\right) & \Longleftrightarrow \text { inlinable }\left(s_{1}\right) \vee \text { inlinable }\left(s_{2}\right) \\
\text { inlinable }\left(\mathbf{i f}(*)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\}\right) & \Longleftrightarrow \text { inlinable }\left(s_{1}\right) \vee \text { inlinable }\left(s_{2}\right) \\
\text { inlinable }(s) & \Longleftrightarrow \perp
\end{aligned}
\]

One simple consequence of this definition is the following:
Lemma \(6.10 \neg\) inlinable \((s) \Longrightarrow \operatorname{inl} l_{P r}^{n}(\vec{l}, s)=s\)
A statement not inlinable stays the same when inlined. We now have all the elements to formally express the soundness condition.

The soundness condition takes a statement \(s\), the list of read variables \(\vec{l}\) it is inlined with, the program \(\operatorname{Pr}\) and also the remaining depth \(n\). Recall that the

\footnotetext{
\({ }^{4}\) These are not possible Viper programs, since Viper forbids the use of perm in loop conditions.
}
list \(\vec{l}\) records the variable names which are already defined somewhere in the inlined program, and thus the variable names which should be avoided when renaming a method body, to avoid capturing variables.

\section*{Definition 6.11 Soundness Condition}

\section*{If \(\neg\) inlinable \((s)\) then \(S C_{P r}^{n}(\vec{l}, s) \Longleftrightarrow\) mono \(_{P r}(s)\).}

\section*{Otherwise for inlinable statements:}
\[
\begin{align*}
& S C_{P r}^{0}(\vec{l}, \text { while }(b) \text { inv } I\{s\}) \Longleftrightarrow \text { mono }_{P r}(\text { assume } \neg b)  \tag{}\\
& S C_{P r}^{n}\left(\vec{l}, s_{1} ; s_{2}\right) \Longleftrightarrow S C_{P r}^{n}\left(\vec{l}, s_{1}\right) \wedge S C_{P r}^{n}\left(\vec{l}, s_{2}\right) \\
& S C_{P r}^{n}\left(\vec{l}, \text { if }(*)\left\{s_{1}\right\} \text { else }\left\{s_{2}\right\}\right) \Longleftrightarrow S C_{P r}^{n}\left(\vec{l}, s_{1}\right) \wedge S C_{P r}^{n}\left(\vec{l}, s_{2}\right) \\
& S C_{P r}^{n}(\vec{l}, \vec{y}:=m(\vec{x})) \Longleftrightarrow \ldots\left(\text { see below }^{*}\right) \\
& S C_{P r}^{n+1}(\vec{l}, \text { while }(b) \text { inv } I\{s\}) \Longleftrightarrow\left(\text { framing }_{P r}\left(\text { inl }_{P r}^{n}(\vec{l}, s)\right) \vee \text { framing }_{P r}(s)\right) \wedge \text { wfm }_{P r}(b) \\
& \wedge S C_{P r}^{n}(\vec{l}, s) \wedge S C_{P r}^{n}(\vec{l}, \text { while }(b) \text { inv } I\{s\}) \\
& S C_{P r}^{0}(\vec{l}, s) \quad \text { (otherwise) }
\end{align*}
\]

Remark 6.12 (*): In the Isabelle formalization (Appendix A), we require \(^{\text {( }}\),
\[
S C_{P r}^{0}(\text { while }(b) \operatorname{inv} I\{s\}) \Longleftrightarrow w f m_{P r}(b) \wedge \operatorname{mono}_{P r}(s)
\]
because we use in the Isabelle proof the lemma
\[
S C_{P r}^{n}(s) \Longrightarrow \text { mono }_{P r}(s)
\]

This lemma does not hold without the additional requirement (one reason being that the body inside the loop can be not safeMono, which makes the loop not safeMono). However, this lemma is not necessary for the proof. Indeed, we do not use this lemma in the proofs presented here, and the Isabelle formalization should be modified in this direction in the future.

We define the missing method call case below (for \(n=0\), there is no condition to be satisfied). Let us explain the intuition behind the soundness condition for loops, given in Definition 6.11. Let us first consider a loop with condition \(b\). If \(n=0\), then the soundness condition requires that assume \(\neg(b)\) must be mono (which should be clear from our discussion above). However, when the remaining depth is not zero but \(n+1\), we require several properties:
1. \(w f m_{P r}(b)\) : As explained above, we use both assume \(b\) and assume \(\neg b\).
2. \(S C_{P r}^{n}(\vec{l}, s)\) : We need this property since we inline the loop body.
3. \(S C_{P_{r}}^{n}(\vec{l}\), while \((b) \operatorname{inv} I\{s\})\) : We need this property since we also inline the loop (with a smaller bound).
4. framing \(g_{P r}\left(\right.\) inl \(\left._{P r}^{n}(\vec{l}, s)\right) \vee\) framing \(_{P r}(s)\) : Interestingly, this means that it is sufficient that either the original loop body is framing (with any existing annotation) or the inlined loop body is framing.
We finally define the soundness condition for method calls:

\section*{Definition 6.13 Soundness condition for method calls}

First, the soundness condition for a method call implies that there exist \(\overrightarrow{\operatorname{args},} \overrightarrow{r e t s}, P\), \(Q\) and method body s such that
\[
(m, \overrightarrow{\operatorname{args}}, \overrightarrow{r e t s}, P, Q, s) \in P r
\]

In this case, let \(s^{\prime}=\operatorname{rename}((\overrightarrow{\operatorname{args}}++\overrightarrow{\operatorname{rets}}, \vec{x}++\vec{y}, \vec{l}, \operatorname{read}(s)), s)\), and \(\overrightarrow{l^{\prime}}=\) \(\vec{l} \cup \operatorname{read}\left(s^{\prime}\right)\). Then
\(S C_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x})) \Longleftrightarrow\left(\right.\) framing \(\left._{P r}\left(\operatorname{inl}_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \vee \operatorname{framing}_{P r}(s)\right) \wedge S C_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\)
Since we inline by renaming the method body and recursively calling the inline function on \(s^{\prime}\) (the renamed method body) and \(\overrightarrow{l^{\prime}}\) (the new list of variables to avoid), we express the soundness condition using them. As in the case of loop, we have the disjunction framing \(\operatorname{Pr}\left(\right.\) inl \(\left._{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \vee\) framing \(_{P r}(s)\). We also need \(S C_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\) since this is what we inline.

One important and useful lemma which can be proved by computational induction on the structure of the soundness condition is the following:
Lemma 6.14
\[
S C_{P r}^{n}(\vec{l}, s) \Longrightarrow \text { mono }_{P r}\left(\operatorname{inl} l_{P r}^{n}(\vec{l}, s)\right)
\]

Remark 6.15 It is interesting to note that we have neither framing \(\operatorname{Pr}_{P r}\left(\operatorname{inl}_{P r}^{n}(\vec{l}, s)\right) \Longrightarrow\) framing \(_{P r}(s)\), nor the reciprocal. It is easy to see, for example, that we can have framing \(_{P r}\left(\operatorname{inl}_{P r}^{0}(\vec{l}, s)\right)\) without having framing \(P_{P r}(s)\). On the other hand, take the Viper method call \(s:=y:=m(x)\), with the body of \(m\) being exhale acc(x.f, perm(x.f)). We know that the body of \(m\) is not framing, therefore we have \(\neg\) framing \(_{\operatorname{Pr}}\left(\operatorname{inl}_{P r}^{1}(\vec{l}, s)\right)\). However, any well-formed method call is framing (because of the semantics of method calls).

\subsection*{6.3 Soundness theorem}

We now have all the formalism needed to express the soundness theorem:

\section*{Theorem 6.16 Soundness}

Assume
1. wfProg \((\operatorname{Pr})\)
2. \(w f S t m t_{P r}(s)\)
3. \(S C_{P r}^{n}(\operatorname{modif}(s), s)\)
4. verProg (Pr)
5. \(\operatorname{ver}_{P r}(u, s)\) ( \(u\) is the empty state of the separation algebra).

Then \(\operatorname{ver}_{P r}\left(u, \operatorname{inl}_{P r}^{n}(\operatorname{modif}(s), s)\right)\).

\section*{Remark 6.17 Contraposition of the soundness theorem}

By contraposition, we obtain that, under the following conditions:
1. wfProg (Pr)
2. \(w f S t m t_{P_{r}}(s)\)
3. \(S C_{P_{r}}^{n}(\operatorname{modif}(s), s)\)

If
\[
\exists n . \neg \operatorname{ver}_{P r}\left(u, i n l_{P r}^{n}(\operatorname{modif}(s), s)\right)
\]
then there does not exist any way of annotating the program such that both verProg \((\operatorname{Pr})\) and ver \(_{P r}(u, s)\).

The contraposition corresponds to what was discussed in Chapter 1. Namely, if the inlined program does not verify, then there exists a fundamental error in the original program.

We do not prove the soundness theorem directly, but we define and prove a stronger property we call the soundness invariant. We first define the domain of a set of states, needed to express the invariant.

Definition 6.18 Domain of a set of states
\[
\operatorname{domain}(A):=\cup_{\varphi \in A} \sigma(\varphi)
\]

The domain of a set is simply the set of variables defined by all of its states. We use this definition to express the fact that, when inlining method calls, we rename the variables of the corresponding method bodies to avoid capturing variables already defined.

Definition 6.19 Partial Soundness Property
\[
\begin{aligned}
\operatorname{PSP}_{P r}^{n}(\vec{l}, s) & :=\left(\forall A^{\prime}, A \subseteq \Sigma \cdot A^{\prime} \gg A \wedge \operatorname{domain}\left(A^{\prime}\right) \subseteq \vec{l} \wedge \operatorname{ver}_{P r}(A, s)\right. \\
& \left.\Longrightarrow \operatorname{ver}_{P r}\left(A^{\prime}, \operatorname{inl_{Pr}^{n}}(\vec{l}, s)\right) \wedge \operatorname{sem}_{P r}\left(A^{\prime}, \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right) \gg \operatorname{sem}_{P r}(A, s)\right)
\end{aligned}
\]

This partial soundness property is parametrized by a program Pr, a bound \(n\), the list of variables \(\vec{l}\) which records variables already defined somewhere in the program, and a statement \(s\). domain \(\left(A^{\prime}\right) \subseteq \vec{l}\) asserts that all variables
defined by all states of \(A\) are included in the list of variables \(\vec{l}\). This is to avoid interferences with newly defined variables when inlining the program. In the end, we instantiate this partial soundness property for \(A^{\prime}=A:=\{u\}\), a singleton with the empty state, for which this property is trivially satisfied for any list \(\vec{l}\). The partial soundness property says that, for any two sets of states \(A^{\prime}\) and \(A\) such that \(A^{\prime} \gg A\), if \(A^{\prime}\) does not interfere with \(\vec{l}\) as we just described, and if \(s\) verifies with the set of states \(A\), then \(A^{\prime}\) verifies with the inlined statement. Moreover, executing the inlined statement from the set of states \(A^{\prime}\) results in a stronger set of states than executing the original statement from the set of states \(A\).

This is intuitively what happens under the soundness condition: Loop invariants and method preconditions and postconditions allow hiding some information (such as the values of variables) and leak some permissions. On the other hand, all the information and the permissions are visible in the inlined program.

We now define the soundness invariant we prove, using this partial soundness property:

Definition 6.20 Soundness Invariant (induction hypothesis)
\(\operatorname{SI}_{P r}^{n}(\vec{l}, s) \Longleftrightarrow\left(w f S t m t_{P r}(s) \wedge \operatorname{modif}(s) \subseteq \vec{l} \wedge S C_{P r}^{n}(\vec{l}, s) \Longrightarrow \operatorname{PSP}_{P r}^{n}(\vec{l}, s)\right)\)
That is, we prove that for all well-formed statements such that the variables it modifies are included in \(\vec{l}\), under the soundness condition, we have the partial soundness property. Formally, we simply prove
Lemma 6.21
\[
S I_{P r}^{n}(\vec{l}, s)
\]

That is, this soundness invariant is true for any program \(P r\), any natural number \(n\), any list of variable names \(\vec{l}\) and any statement \(s\). We prove this lemma by computational induction on the structure of the inline function. The complete proof has been formalized in Isabelle/HOL. However, we only show in this chapter two interesting cases of the proof, namely loops and method calls. By instantiating this soundness invariant for \(\vec{l}:=\operatorname{modif}(s)\), we get Theorem 6.16, the soundness theorem.

\subsection*{6.4 Induction case: method calls}

In this section, we prove the following induction case, for method calls:

\section*{Lemma 6.22 Method calls induction case}

Assume
1. wfProg \((\operatorname{Pr}) \wedge\) verProg \((\operatorname{Pr})\)
2. \((m, \overrightarrow{a r g s}, \overrightarrow{r e t s}, P, Q, s) \in P r\)
3. \(s^{\prime}=\operatorname{rename}((\overrightarrow{\operatorname{args}}++\overrightarrow{\operatorname{rets}}, \vec{x}++\vec{y}, \vec{l}, \operatorname{read}(s)), s)\)
4. \(\overrightarrow{l^{\prime}}=\vec{l} \cup \operatorname{read}\left(s^{\prime}\right)\)
5. \(S I_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\)

Then \(S_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\).
Proof To prove \(S I_{P_{r}}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\), we assume \(\operatorname{wfStmt}_{P r}(\vec{y}:=m(\vec{x}))\), \(\operatorname{modif}(\vec{y}:=m(\vec{x})) \subseteq \vec{l}\), and \(\operatorname{SC}_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\), and we show \(\operatorname{PSP}_{P r}^{n+1}(\vec{l}, \vec{y}:=\) \(m(\vec{x}))\).
We get, by definitions, \(w f \operatorname{Stmt}_{P r}(s)\) and \(\vec{y} \subseteq \vec{l}\). Using \(\operatorname{verProg}(\operatorname{Pr})\) and that the method call is well-formed, we also get
\[
\operatorname{ver}_{P r}(\{u\}, \operatorname{var}(\overrightarrow{\operatorname{args}}++\overrightarrow{\text { rets }}) ; \text { inhale } P ; s ; \text { exhale } Q)
\]

Let \(t:=(\overrightarrow{\operatorname{args}}++\overrightarrow{\text { rets }}, \vec{x}++\vec{y}, \vec{l}, \operatorname{read}(s))\), which is well-formed. Then let \(P^{\prime}:=\operatorname{rename}(P, t)\) and \(Q^{\prime}:=\operatorname{rename}(Q, t)\) be the renaming of the precondition and postcondition with respect to the renaming quadruple \(t\).
From \(S C_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\) we get \(S C_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\). Then, from \(S I_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\) and the properties of well-formed renaming we get \(P S P_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\) and
\[
\operatorname{ver}_{P r}\left(\{u\}, \operatorname{var}(\vec{x}++\vec{y}) ; \text { inhale } P^{\prime} ; s^{\prime} ; \text { exhale } Q^{\prime}\right)
\]
which gives
\[
\operatorname{ver}_{P r}\left(h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right), s^{\prime}\right)
\]
and
\[
\operatorname{sem}_{P r}\left(h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right), s^{\prime}\right) \gg \operatorname{Inh}\left(Q^{\prime}\right)
\]
 \(\operatorname{domain}\left(A^{\prime}\right) \subseteq \vec{l}\) and \(\operatorname{ver}_{P r}(A, \vec{y}:=m(\vec{x}))\). Let then \(\varphi^{\prime} \in A^{\prime}, \varphi \in A\) such that \(\varphi \ll \varphi^{\prime}\). We have
\[
\operatorname{ver}_{P r}\left(\{\varphi\} \text {, exhale } P^{\prime} ; \text { havoc } \vec{y} ; \text { inhale } Q^{\prime}\right)
\]

We then can obtain \(i \in \operatorname{Inh}\left(P^{\prime}\right), r \in \Sigma\) such that \(\varphi=i \oplus r\). Let \(r^{\prime}:=\bar{h}(r, \vec{y})\), we have then \(\varphi=i \oplus|\varphi| \oplus r^{\prime}\).

Let us now describe the semantics of \(\vec{y}:=m(\vec{x})\) :
\[
\begin{aligned}
\operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x})) & =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, \vec{y}:=m(\vec{x})\right) \\
& =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, \text { exhale } P^{\prime} ; \text { havoc } \vec{y} ; \text { inhale } Q^{\prime}\right) \\
& =\operatorname{sem}_{P r}\left(\left\{|i| \oplus|\varphi| \oplus r^{\prime}\right\}, \text { havoc } \vec{y} ; \text { inhale } Q^{\prime}\right) \\
& =\operatorname{sem}_{P r}\left(\left\{\bar{h}\left(|i| \oplus|\varphi| \oplus r^{\prime}, \vec{y}\right)\right\} \oplus h(\vec{y}) \text {, inhale } Q^{\prime}\right) \\
& =\operatorname{sem}_{P r}\left(\left\{\bar{h}(|\varphi|, \vec{y}) \oplus r^{\prime}\right\} \oplus h(\vec{y}), \text { inhale } Q^{\prime}\right) \\
& =\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(Q^{\prime}\right)
\end{aligned}
\]

We know \(\operatorname{inl} l_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))=\operatorname{inl_{Pr}^{n}}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\). Our aim is to prove two things:
1. \(H_{1}: \operatorname{ver}_{P r}\left(\{\varphi\}, i n l_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right)\)
2. \(H_{2}: \operatorname{sem}_{P r}\left(\{\varphi\}, \operatorname{inl} P_{r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right) \gg \operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x}))\)

These two results are enough to conclude the proof. Indeed, we get
\[
\text { mono }_{P r}\left(\text { inl }_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right)
\]
from Lemma 6.14. Using this property, combined with \(\varphi \ll \varphi^{\prime}\), we get
1. \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl_{Pr}^{n+1}}(\vec{l}, \vec{y}:=m(\vec{x}))\right)\)
2. \(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right) \gg \operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x}))\)
which concludes the proof.
We prove therefore these two properties, \(H_{1}\) and \(H_{2}\), by considering the two cases corresponding to the disjunction of the soundness condition
\[
S C_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))
\]

Case 1: framing \(_{\text {Pr }}(s)\).
In this case, we also have framing \(_{P_{r}\left(s^{\prime}\right)}\) since the renaming quadruple \(t\) is well-formed and \(s^{\prime}\) is the renamed version of \(s\).
Let \(B^{\prime}:=\{\varphi\}\) and \(B:=h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right) \oplus\left\{r^{\prime}\right\}\). We have \(B^{\prime} \gg B\). Indeed, \(\varphi=i \oplus|\varphi| \oplus r^{\prime}\), with \(i \in \operatorname{Inh}\left(P^{\prime}\right)\), and \(\vec{x} \cup \vec{y} \subseteq \sigma(\varphi)\). Moreover, we have \(\operatorname{ver}_{P_{r}}\left(B, s^{\prime}\right)\), using mono \(_{\text {Pr }}\left(s^{\prime}\right)\) (which holds since \(s^{\prime}\) is framing) and \(\operatorname{ver}_{P r}\left(h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right), s^{\prime}\right)\). We also have domain \(\left(B^{\prime}\right)=\sigma(\varphi) \subseteq \vec{l} \subseteq \overrightarrow{l^{\prime}}\).
By applying \(P S P_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\) we get
\[
\operatorname{ver}_{P r}\left(B^{\prime}, \operatorname{inl}_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right)
\]
and
\[
\operatorname{sem}_{P r}\left(B^{\prime}, i n l_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \gg \operatorname{sem}_{P r}\left(B, s^{\prime}\right)
\]

Then
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\{\varphi\}, \operatorname{inl_{Pr}^{n+1}(\vec {l},\vec {y}:=m(\vec {x})))}\right. \\
& =\operatorname{sem}_{P r}\left(B^{\prime}, \operatorname{inl_{Pr}^{n}}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \\
& \gg \operatorname{sem}_{P r}\left(B^{\prime}, \operatorname{inl_{Pr}^{n}}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus h(\vec{y}) \\
& \left.\quad \quad \text { (since } \vec{y} \subseteq \sigma(\varphi) \text { and modif }\left(s^{\prime}\right) \cap \sigma(\varphi) \subseteq \vec{y}\right) \\
& \gg \operatorname{sem}_{P r}\left(B, s^{\prime}\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus h(\vec{y}) \\
& =\operatorname{sem}_{P r}\left(h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right) \oplus\left\{r^{\prime}\right\}, s^{\prime}\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus h(\vec{y}) \\
& \gg \operatorname{sem}_{P r}\left(h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right), s^{\prime}\right) \oplus\left\{r^{\prime}\right\} \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus h(\vec{y}) \\
& \text { (since } s^{\prime} \text { is framing) } \\
& \gg \operatorname{Inh}\left(Q^{\prime}\right) \oplus\left\{r^{\prime}\right\} \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus h(\vec{y}) \quad \\
& =\operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x}))
\end{aligned}
\]

Case 2: framing \(\left._{P_{r}\left(\text { inl }_{P r}^{n}\right.}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right)\).
Let \(B^{\prime}:=\{i \oplus|\varphi|\}\) and \(B:=h(\vec{x}) \oplus h(\vec{y}) \oplus \operatorname{Inh}\left(P^{\prime}\right)\). We have \(B^{\prime} \gg B\), since \(i \in \operatorname{Inh}\left(P^{\prime}\right)\). We also have domain \(\left(B^{\prime}\right) \subseteq \sigma(\varphi) \subseteq \vec{l} \subseteq \overrightarrow{l^{\prime}}\) and \(\operatorname{ver}_{P r}\left(B, s^{\prime}\right)\). By applying \(P S P_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\) we get
\[
\operatorname{ver}_{P r}\left(B^{\prime}, i n l_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right)
\]
and
\[
\operatorname{sem}_{P r}\left(B^{\prime}, i n l_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \gg \operatorname{sem}_{P r}\left(B, s^{\prime}\right) \gg \operatorname{Inh}\left(Q^{\prime}\right)
\]

We have mono \(_{P r}\left(\right.\) inl \(\left._{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right)\) since it is framing. Therefore, since \(\{\varphi\} \gg B^{\prime}\), we have \(\operatorname{ver}_{P r}\left(\{\varphi\}, \operatorname{inl} l_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right)\).

Moreover
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\{\varphi\}, \operatorname{inl}_{P r}^{n+1}(\vec{l}, \vec{y}:=m(\vec{x}))\right) \\
& =\operatorname{sem}_{P r}\left(B^{\prime} \oplus\left\{r^{\prime}\right\}, \operatorname{inl_{Pr}^{n}(\vec {l^{\prime }},s^{\prime }))}\right. \\
& =\operatorname{sem}_{P r}\left(B^{\prime} \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\}, \operatorname{inl_{Pr}^{n}(\vec {l^{\prime }},s^{\prime }))}\right. \\
& \gg \operatorname{sem}_{P r}\left(B^{\prime}, \operatorname{inl} l_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right)\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\} \\
& \quad \text { (since } \text { inl ll }_{P r}^{n}\left(\overrightarrow{l^{\prime}}, s^{\prime}\right) \text { is framing) } \\
& \gg \operatorname{sem}_{P r}\left(B, s^{\prime}\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\} \\
& \gg \operatorname{sem}_{P r}\left(B, s^{\prime}\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{y}) \\
& \gg \operatorname{Inh}\left(Q^{\prime}\right) \oplus\{\bar{h}(|\varphi|, \vec{y})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{y}) \\
& =\operatorname{sem}_{P r}(\{\varphi\}, \vec{y}:=m(\vec{x}))
\end{aligned}
\]

\subsection*{6.5 Induction case: loops}

We prove, in this section, the following induction case, for loops:

\section*{Lemma 6.23 Loops induction case}

Assume
1. \(w f \operatorname{Prog}(\operatorname{Pr}) \wedge v e r \operatorname{Prog}(\operatorname{Pr})\)
2. \(S I_{P r}^{n}(\vec{l}, s)\)
3. \(S I_{P r}^{n}(\vec{l}\), while (b) inv \(I\{s\})\)

Then \(S I_{P r}^{n+1}(\vec{l}\), while (b) inv \(I\{s\})\).
Proof Let \(w:=\) while (b) inv \(I\{s\}\).
To prove \(\operatorname{SI}_{P r}^{n+1}(\vec{l}, w)\), we assume \(w f \operatorname{Stmt}_{P r}(w)\), \(\operatorname{modif}(w) \subseteq \vec{l}\), and \(S C_{P_{r}}^{n+1}(\vec{l}, w)\), and we show \(\operatorname{PSP}_{P r}^{n+1}(\vec{l}, w)\).
We then get, by definitions, \(w f \operatorname{Stmt}_{P_{r}}(s)\), \(\operatorname{modif}(s) \subseteq \vec{l}, S C_{P r}^{n}(\vec{l}, w)\), and \(S C_{P r}^{n}(\vec{l}, s)\). From 2 and 3 we then obtain \(P S P_{P r}^{n}(\vec{l}, s)\) and \(P S P_{P r}^{n}(\vec{l}, w)\).
Assume \(w_{f S t m t}^{P r}(w)\), \(\operatorname{modif}(w) \subseteq \vec{l}\), and \(S C_{P r}^{n+1}(\vec{l}, w)\). We then get, by definitions, \(w f \operatorname{Stmt}_{P r}(s)\), \(\operatorname{modif}(s) \subseteq \vec{l}, S C_{P r}^{n}(\vec{l}, w)\), and \(S C_{P r}^{n}(\vec{l}, s)\). From 2 and 3 we then obtain \(\operatorname{PSP_{Pr}^{n}}(\vec{l}, s)\) and \(P S P_{P r}^{n}(\vec{l}, w)\).
Towards proving \(P S P_{P r}^{n+1}(\vec{l}, w)\), let \(A^{\prime}, A \subseteq \Sigma\) such that \(A^{\prime} \gg A\), \(\operatorname{domain}\left(A^{\prime}\right) \subseteq\) \(\vec{l}\), and \(\operatorname{ver}_{P r}(A, w)\). Let then \(\varphi^{\prime} \in A^{\prime}, \varphi \in A\) such that \(\varphi \ll \varphi^{\prime}\). We then have \(\sigma(\varphi) \subseteq \vec{l}, \sigma\left(\varphi^{\prime}\right) \subseteq \vec{l}\), and \(\operatorname{ver}_{P r}(\{\varphi\}, w)\).

Our aim is to prove two things:
1. \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, w)\right)\)
2. \(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl_{Pr}^{n+1}}(\vec{l}, w)\right) \gg \operatorname{sem}_{P r}(\{\varphi\}, w)\)

\section*{Moreover,}
\[
\begin{aligned}
\operatorname{inl_{Pr}^{n+1}(\vec {l},w)} & =\mathbf{i f}(b)\left\{\operatorname{inl_{Pr}^{n}(\vec {l},s);inl_{Pr}^{n}(\vec {l},w)\} }\right. \\
& =\mathbf{i f}(*)\left\{\text { assume } b ; i n l_{P r}^{n}(\vec{l}, s) ; \operatorname{inl_{Pr}^{n}}(\vec{l}, w)\right\} \text { else }\{\text { assume } \neg b\}
\end{aligned}
\]

Since we have \(w f m(b)\), we know that mono \(\left._{P_{r}(\text { assume }} b\right)\) and mono \(\left._{P_{r}(\text { assume }} \neg b\right)\).
Using the verification of the loop (i.e., \(\operatorname{ver}_{P r}(\{\varphi\}, w)\) ), we get that \(b\) and \(\neg b\) are well-defined with \(\varphi\). Therefore, \(\operatorname{ver}_{\operatorname{Pr}}\left(\left\{\varphi^{\prime}\right\}\right.\), assume \(\left.b\right)\), and \(\operatorname{ver}_{\operatorname{Pr}}\left(\left\{\varphi^{\prime}\right\}\right.\), assume \(\left.\neg b\right)\). We also get that \(\varphi \gg \operatorname{Inh}(I)\), thus there is an \(i \in \operatorname{Inh}(I)\) and an \(r \in \Sigma\) such that \(\varphi=i \oplus r\).
Let \(V:=\operatorname{modif}(s) \cap \sigma(\varphi)\). Let \(r^{\prime}=\bar{h}(r, \vec{V})\), then we have \(\varphi=i \oplus|\varphi| \oplus r^{\prime}\).
Let \(F:=\operatorname{Inh}(I) \oplus h(\vec{V}) \oplus\{\bar{h}(|\varphi|, \vec{V})\} \oplus\left\{r^{\prime}\right\}\).
Since \(V \subseteq \sigma(\varphi)\), we have
\[
\{\varphi\} \gg F
\]

Let us define a function filter to filter a set of states with respect to an assertion, such that
\[
\operatorname{filter}_{P}(A):=\{a \in A \mid P(a)\}
\]

Using this function, let us now describe the semantics of the loop:
\[
\begin{aligned}
\operatorname{sem}_{P r}(\{\varphi\}, w) & =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, w\right) \\
& =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, \text { exhale } I ; \text { havoc } \vec{V} ; \text { inhale } I ; \text { assume } \neg b\right) \\
& =\operatorname{sem}_{P r}\left(\left\{|i| \oplus|\varphi| \oplus r^{\prime}\right\}, \text { havoc } \vec{V} ; \text { inhale } I ; \text { assume } \neg b\right) \\
& =\operatorname{sem}_{P r}\left(\left\{\bar{h}\left(|i| \oplus|\varphi| \oplus r^{\prime}, \vec{V}\right)\right\} \oplus h(\vec{V}) \text {, inhale } I ; \text { assume } \neg b\right) \\
& =\operatorname{sem}_{P r}\left(\{\bar{h}(|\varphi|, \vec{V})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{V}) \text {, inhale } I ; \text { assume } \neg b\right) \\
& =\operatorname{sem}_{P r}\left(\{\bar{h}(|\varphi|, \vec{V})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I) \text {, assume } \neg b\right) \\
& =\operatorname{sem}_{P r}(F, \text { assume } \neg b) \\
& =\operatorname{filter}_{\neg b}(F)
\end{aligned}
\]

We now consider two cases, corresponding to the branch \(\varphi\) takes:
Case 1: \(\neg b(\varphi)\).
Then \(\neg b\left(\varphi^{\prime}\right)\) since \(w f m(b)\). We have \(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}\right.\), assume \(\left.b\right)=\varnothing\), which means that the conditional branch with the loop iteration verifies. Since we also have \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}\right.\), assume \(\left.\neg b\right)\), we get \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, w)\right)\).

Moreover
\[
\begin{aligned}
\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl}_{P r}^{n+1}(\vec{l}, w)\right) & =\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \text { assume } \neg b\right) \\
& =\left\{\varphi^{\prime}\right\} \\
& \gg\{\varphi\} \\
& \gg F \\
& \gg \operatorname{filter}_{\neg b}(F) \\
& =\operatorname{sem}_{P r}(\{\varphi\}, w)
\end{aligned}
\]
therefore concluding this case.
Case 2: \(b(\varphi)\).
Then \(b\left(\varphi^{\prime}\right)\) since \(w f m(b)\), thus we have
\[
\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, w)\right)=\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n}(\vec{l}, s) ; i n l_{P r}^{n}(\vec{l}, w)\right)
\]
and \(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}\right)\left(i n l_{P r}^{n+1}(\vec{l}, w)\right)=\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}\right)\left(i n l_{P r}^{n}(\vec{l}, s) ; i n l_{P r}^{n}(\vec{l}, w)\right)\).
We show four preliminary results before concluding:
1. \(\operatorname{sem}_{P r}\left(\right.\) filter \(\left._{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \gg\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus\) \(\operatorname{Inh}(I)\) and \(\operatorname{ver}_{P r}\left(\operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right)\)
We know from the verification of the loop that
\[
\operatorname{ver}_{P r}(\{|\varphi|\} \text {, havoc } \vec{V} \text {; inhale } I \text {; assume } b ; s \text {; exhale } I)
\]

We have
\[
\{\varphi\} \gg\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)
\]
and \(b(\varphi)\), thus
\[
\begin{aligned}
& \operatorname{sem}_{P r}(\{|\varphi|\}, \text { havoc } \vec{V} ; \text { inhale } I ; \text { assume } b) \\
= & \operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I))
\end{aligned}
\]

We then get
\[
\operatorname{ver}_{P r}\left(\operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right)
\]
and
\[
\operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \gg \operatorname{Inh}(I)
\]

Since \(s\) cannot undefine variables, and since \(\operatorname{modif}(s) \subseteq \vec{V}\), we get
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \\
& \gg\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)
\end{aligned}
\]
2. \(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right) \gg \operatorname{sem}_{P r}(\) filter \(b(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s) \oplus\) \(\left\{r^{\prime}\right\}\) and \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n}(\vec{l}, s)\right)\) :

We know that
\[
\begin{aligned}
\{i \oplus|\varphi|\} & \gg\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I) \\
& \gg \operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I))
\end{aligned}
\]

Let us consider the two cases corresponding to the disjunction in the soundness condition:

Case 2.1: framing \(_{P_{r}(s)}\)

By applying \(P S P_{P r}^{n}(\vec{l}, s)\) to \(A^{\prime}:=\left\{\varphi^{\prime}\right\}\) and \(A:=\{\varphi\}\), we get \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right)\) and
\[
\begin{aligned}
\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl_{Pr}^{n}(\vec {l},s))}\right. & \gg \operatorname{sem}_{P r}(\{\varphi\}, s) \\
& =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, s\right) \\
& \gg \operatorname{sem}_{P r}(\{i \oplus|\varphi|\}, s) \oplus r^{\prime} \quad \text { (since } s \text { is framing) } \\
& \gg \operatorname{sem}_{P r}\left(\text { filter }{ }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \oplus\left\{r^{\prime}\right\} \\
& \text { (since } s \text { is mono) }
\end{aligned}
\]

Case 2.2: framing \(_{P r}\left(\operatorname{inl}_{P r}^{n}(\vec{l}, s)\right)\)
We have \(\operatorname{ver}_{P r}\left(\right.\) filter \(_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I))\),s). By applying \(\operatorname{PSP}_{P r}^{n}(\vec{l}, s)\) to \(A^{\prime}=A:=\operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I))\), we get
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right) \\
& \gg \operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right)
\end{aligned}
\]
and
\[
\operatorname{ver}_{P r}\left(\operatorname{filter}_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right)
\]
thus \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl}_{P r}^{n}(\vec{l}, s)\right)\) (since mono \(_{P r}\left(\right.\) inl \(\left._{P r}^{n}(\vec{l}, s)\right)\) ).

Eventually,
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n}(\vec{l}, s)\right) \\
& \gg \operatorname{sem}_{P r}\left(\{\varphi\}, \operatorname{inl} l_{P r}^{n}(\vec{l}, s)\right) \quad \text { (since } \operatorname{inl} l_{P r}^{n}(\vec{l}, s) \text { is mono) } \\
& =\operatorname{sem}_{P r}\left(\left\{i \oplus|\varphi| \oplus r^{\prime}\right\}, i n l_{P r}^{n}(\vec{l}, s)\right) \\
& \gg \operatorname{sem}_{P r}\left(\{i \oplus|\varphi|\}, i n l_{P r}^{n}(\vec{l}, s)\right) \oplus\left\{r^{\prime}\right\} \quad \text { (since } i n l_{P r}^{n}(\vec{l}, s) \text { is framing) } \\
& \gg \operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), \text { inl }_{P r}^{n}(\vec{l}, s)\right) \oplus\left\{r^{\prime}\right\} \\
& \text { (since } \operatorname{inl} l_{P r}^{n}(\vec{l}, s) \text { is mono) } \\
& \gg \operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \oplus\left\{r^{\prime}\right\}
\end{aligned}
\]
3. \(\operatorname{sem}_{P r}(F, w) \gg \operatorname{filter}_{\neg b}(F)\) and \(\operatorname{ver}_{P r}(F, w)\) :

Let \(f \in F\). There is some \(i_{f} \in \operatorname{Inh}(I)\) and \(h_{V} \in h(\vec{V})\) such that \(f=\) \(i_{f} \oplus h_{V} \oplus \bar{h}(|\varphi|, \vec{V}) \oplus r^{\prime}\). We only show the calculation of \(\operatorname{sem}_{P_{r}}(f, w)\), but \(\operatorname{ver}_{P r}(f, w)\) can easily be derived from it.
\[
\begin{aligned}
& \operatorname{sem}_{P r}(f, w) \\
&= \operatorname{sem}_{P r}\left(\left\{i_{f} \oplus h_{V} \oplus \bar{h}(|\varphi|, \vec{V}) \oplus r^{\prime}\right\}, w\right) \\
&= \operatorname{sem}_{P r}\left(\left\{i_{f} \oplus h_{V} \oplus \bar{h}(|\varphi|, \vec{V}) \oplus r^{\prime}\right\},\right. \\
&\quad \text { exhale } I ; \text { havoc } \vec{V} ; \text { inhale } I ; \text { assume } \neg b) \\
&= \operatorname{sem}_{P r}\left(\left\{\left|i_{f}\right| \oplus h_{V} \oplus \bar{h}(|\varphi|, \vec{V}) \oplus r^{\prime}\right\}, \text { havoc } \vec{V} ; \text { inhale } I ; \text { assume } \neg b\right) \\
&= \operatorname{sem}_{P r}\left(\left\{\bar{h}\left(\left|i_{f}\right| \oplus h_{V} \oplus \bar{h}(|\varphi|, \vec{V}) \oplus r^{\prime}, \vec{V}\right)\right\} \oplus h(\vec{V}) \text {, inhale } I ; \text { assume } \neg b\right) \\
&=\operatorname{sem}_{P r}\left(\left\{\bar{h}\left(\left|i_{f}\right|, \vec{V}\right)\right\} \oplus\{\bar{h}(|\varphi|, \vec{V})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{V}) \text {, inhale } I ; \text { assume } \neg b\right) \\
&= \operatorname{sem}_{P r}\left(\left\{\bar{h}\left(\left|i_{f}\right|, \vec{V}\right)\right\} \oplus\{\bar{h}(|\varphi|, \vec{V})\} \oplus\left\{r^{\prime}\right\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I) \text {, assume } \neg b\right) \\
&= \operatorname{sem}_{P r}\left(\left\{\bar{h}\left(\left|i_{f}\right|, \vec{V}\right)\right\} \oplus F, \text { assume } \neg b\right) \\
& \gg \operatorname{sem}_{P r}(F, \text { assume } \neg b) \quad \quad \text { (because } \text { mono }_{P r}(\text { assume } \neg b) \text { ) } \\
&= \operatorname{filter}_{\neg b}(F)
\end{aligned}
\]
4. \(\operatorname{sem}_{P r}\left(F, i n l_{P r}^{n}(\vec{l}, w)\right) \gg \operatorname{sem}_{P r}(F, w)\) and \(\operatorname{ver}_{P r}\left(F, \operatorname{inl}_{P r}^{n}(\vec{l}, w)\right)\) :

We apply \(P S P_{P r}^{n}(\vec{l}, w)\) to \(A^{\prime}:=F\) and \(A:=F\). We know that \(\operatorname{ver}_{P r}(F, w)\), so we just need to show \(\operatorname{domain}(F) \subseteq \vec{l}\) to satisfy the left hand side of \(P S P_{P r}^{n}(\vec{l}, w)\).
\[
\begin{aligned}
\operatorname{domain}(F) & =\operatorname{domain}(\operatorname{Inh}(I)) \cup \operatorname{domain}(h(\vec{V})) \cup \sigma(\bar{h}(|\varphi|, \vec{V})) \cup \sigma\left(r^{\prime}\right) \\
& =\operatorname{domain}(\operatorname{Inh}(I)) \cup \vec{V} \cup(\sigma(\varphi)-\vec{V}) \cup(\sigma(r)-\vec{V}) \\
& \subseteq \operatorname{domain}(\operatorname{Inh}(I)) \cup \vec{V} \cup(\sigma(\varphi)) \\
& \subseteq \sigma(\varphi) \\
& \subseteq l
\end{aligned}
\]

We then get the right hand side of \(\operatorname{PSP} P_{P r}^{n}(\vec{l}, w)\), which gives us the desired results.

\section*{Putting everything together:}

Using mono \(_{P r}\left(i n l_{P r}^{n}(\vec{l}, w)\right)\), we get \(\operatorname{ver}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, w)\right)\). We can now conclude:
\[
\begin{aligned}
& \operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n+1}(\vec{l}, w)\right) \\
& =\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, i n l_{P r}^{n}(\vec{l}, s) ; i n l_{P r}^{n}(\vec{l}, w)\right) \\
& =\operatorname{sem}_{P r}\left(\operatorname{sem}_{P r}\left(\left\{\varphi^{\prime}\right\}, \operatorname{inl} l_{P r}^{n}(\vec{l}, s)\right), i n l_{P r}^{n}(\vec{l}, w)\right) \\
& \gg \operatorname{sem}_{P r}\left(\operatorname{sem}_{P r}\left(\text { filter }_{b}(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I)), s\right) \oplus\left\{r^{\prime}\right\}, \operatorname{inl}_{P_{r}}^{n}(\vec{l}, w)\right) \\
& \text { (using result } 2 \text { and } \text { mono }_{P r}\left(\text { inl }_{P_{r}}^{n}(\vec{l}, w)\right. \text { )) } \\
& \gg \operatorname{sem}_{P r}\left(\text { filter }_{b}\left(\{\bar{h}(|\varphi|, \vec{V})\} \oplus h(\vec{V}) \oplus \operatorname{Inh}(I) \oplus\left\{r^{\prime}\right\}\right), \operatorname{inl}_{P r}^{n}(\vec{l}, w)\right) \\
& \text { (using result } 1 \text { and } \text { mono }_{P r}\left(i n l_{P r}^{n}(\vec{l}, w)\right) \text { ) } \\
& =\operatorname{sem}_{P r}\left(F, i n l_{P r}^{n}(\vec{l}, w)\right) \\
& \gg \operatorname{sem}_{P_{r}}(F, w) \quad \text { (using result 4) } \\
& \gg \text { filter }_{\neg b}(F) \quad \text { (using result 3) } \\
& =\operatorname{sem}_{P r}(\{\varphi\}, w)
\end{aligned}
\]

\section*{Chapter 7}

\section*{Completeness of Static Inlining}

In Chapter 6 we proved that, under the soundness condition, we have soundness of inlining. That is, if an inlined program (up to any bound \(n\) ) does not verify, we know for sure that the original program, with whatever annotation, does not verify either. \({ }^{1}\)

However, the soundness theorem goes in only one direction: It does not tell us anything about the situation where the inlined program verifies, and therefore does not inform us on how close the relationship is between the behaviour of the inlined program and the original program. As an example, we could define the following inlining function:
\[
\operatorname{inl}_{P r}^{n}(l, s):=\mathbf{s k i p}
\]

This inlining function is sound. Indeed, it always verifies, thus it is never wrong when it finds a fundamental bug (since it never finds any).
We therefore want a completeness property for the inlining function we have defined. The ideal completeness property would state that, under certain conditions, if the inlined program verifies, then there exists an annotation for the program (loop invariants, method preconditions and postconditions) such that this annotated program verifies. Such a property would indeed show us that our inlining function is a good model for what happens with the annotated program.

This completeness property does not hold, for one obvious reason: Inlining always only goes up to a bounded depth, so it cannot provide any information on executions which go deeper than this bound. Moreover, while loops and recursive methods are in general statically unbounded. Therefore, for any bound, we can find an example of incompleteness of static inlining, where the inlined program up to this bound verifies, whereas no annotation would make the original program verify.

\footnotetext{
\({ }^{1}\) Assuming completeness of the verifier.
}

In this chapter, we try to understand which conditions are required to get a completeness result. We then sketch how to prove such a completeness property. Intuitively, the inlined program behaves as if all loop iterations and method bodies would perfectly know the whole context. Therefore, when the inlined program verifies, we try to construct annotations which describe the whole context, in the hope that this reproduces the behavior of the inlined program, and thus makes the original annotated program verify. These annotations are based on the idea of a strongest postcondition, that is an assertion which describes exactly the program state after the execution of a statement from an initial empty state.

The structure of the chapter is as follows. We first define the syntactic transformation of "bounding" a program up to a bound, to align the execution of the original program on the one of the inlined program. The completeness property we study is with respect to this bounded program: If the inlined program verifies, does there exist an annotation which makes the bounded program verify? We show with two examples that this completeness property does not hold in general in Viper. One example is based on the lack of indices in loops and in recursive methods, and the other one is based on restrictions on how permissions can be used in Viper. We finally express this completeness property in our general framework, and sketch a proof for it, under conditions which disallow the previous examples, and with an assumption that there exists a strongest postcondition.

The concepts presented in this chapter have not been fully formalized yet. We therefore stay at an informal level, in order to convey the intuition behind it.

\subsection*{7.1 Syntactic transformation: bounded program}

As explained, we need to modify the original program, such that the execution stops if it reaches a depth which cannot be reached in the inlined program. This way, we can say something about the original program based on information from the inlined program. We define a syntactic transformation, bound \({ }^{n}(s, P r)\), which takes as input an initial statement \(s\) and a program Pr, and outputs another initial statement and another program, as illustrated in Figure 7.1. Even though bound \({ }^{n}(s, P r)\) is a syntactic transformation, it represents the same program semantically, except that the execution is sometimes stopped.

Listing 7.1 shows a simple program (with dots representing some statements), consisting of an initial statement with one loop and calling one method \(m 1\). The method \(m 1\) calls another method \(m 2\). Listing 7.2 shows the result of the syntactic transformation bound \({ }^{42}\) to this program. We intrument the program in two ways:
    ...
    m1(...)
    while (b)
    \{
    \}
    ...
\}
method m1(...) \{
    . .
    m2(...)
\}
\}
method m1(..., old_depth: Int) \{
    var depth: Int := old_d
    assume depth > 0
    m2(..., depth - 1)
\}

Listing 7.1: Original program.
```

method initial() {

```
```

method initial() {

```
method initial() \{
    var depth: Int := 42
    assume depth > 0
    m1(..., depth - 1)
    var before_depth: Int := depth
    while (b \&\& depth >0)
    \{
        depth := depth - 1
        ...
    \}
    assume ! b
    depth := before_loop_depth
    ...

Figure 7.1: An example of the syntactic transformation bound \({ }^{42}(s, P r)\).
1. We keep track of the current depth of the execution, via the variable depth.
2. We stop the execution if depth is zero or less, similarly to what happens with the inline function.

The current depth is transmitted as an additional argument to methods, which then assign it to the variable depth to be allowed to modify it. Every method call decrements the current depth by one.

We do the same for loops. Before every loop, we record the current depth, using an auxiliary variable. Moreover, we add the condition depth \(>0\) to the loop condition. This ensures that we do not enter the loop if we reached the bound. We assume \(!b\) after the loop, to stop executions which should have enter the loop one more, but which did not because of the condition depth \(>0\). Furthermore, at every iteration of the loop, we decrement the depth by one. Eventually, we go back to the depth before the loop.

Using this syntactic transformation, we can now express the completeness
property we want to have and prove:

\section*{Definition 7.1 Completeness property}

If
\[
\operatorname{inl}_{P r}^{n}(\operatorname{modif}(s), s)
\]
verifies, then there exists an annotation \(A\) such that
\[
\text { bound }{ }^{n}(s, \operatorname{annotate}(\operatorname{Pr}, A))
\]
verifies.
This completeness property does not hold in Viper, as we show in the next chapter. Similarly to what we did in Chapter 3 for the soundness property, we try to find a completeness condition such that this holds.
Since this chapter is not fully formalized, it is not clear whether this will be the definitive shape of the completeness. In particular, it could make more sense to define a semantic notion of a bounded program, instead of a syntactic one.

\subsection*{7.2 Examples of incompleteness in Viper}

We have now expressed the completeness property we want to have. Like the soundness property, it does not hold in general. We show, in this section, two cases in Viper in which this completeness does not hold. The first case occurs when loops or recursive methods do not have an index to use in the annotations. This case is a general one, and is relevant for our general framework. The second one occurs because of a permission gap in Viper, and is therefore restricted to Viper.

\subsection*{7.2.1 Loops and recursive methods without index}

We show the first example on Listing 7.3 and Listing 7.4. Listing 7.3 shows the bounded program (with a bound of 2), whereas Listing 7.4 shows the inlined program (with the same bound of 2).
We use two abstract methods (methods without any specified body), flip_coin and thow_dice, to encode havocing (which currently does not exist in Viper). flip_coin simply returns a boolean with no annotations, thereby havocing a boolean. throw_dice havocs an integer, and contrains it to be between 1 and 6 included.

This program does something simple. It keeps track of a counter \(c\), initialized at 0 . It then flips a coin \(b\), and if \(b\) is true, it throws a dice (result between 1 and 6 included), and adds the result of this dice to the counter. It then goes back to flipping a coin and does the same, until \(b\) is false.

Listing 7.3: Bounded program (bound of 2).
```

method flip_coin() returns (b: Bool)
method throw_dice() returns (i: Int)
ensures i >= 1 \&\& i <= 6
method initial_method() {
var depth: Int := 2
var b: Bool
var d: Int
var c: Int := 0
b := flip_coin()
var before_depth: Int := depth
while (b \&\& depth > 0) {
depth := depth - 1
d := throw_dice()
c := c + d
b := flip_coin()
}
assume !b
assert c <= 12.
}

```

Listing 7.4: Inlined program (bound of 2).
```

method flip_coin() returns (b: Bool)
method throw_dice() returns (i: Int)
ensures i >= 1 \&\& i <= 6
method initial_method() {
var b: Bool
var d: Int
var c: Int := 0
b := flip_coin()
if (b) {
d := throw_dice()
c := c + d
b := flip_coin()
if (b) {
d := throw_dice()
c := c + d
b := flip_coin()
assume !b
}
}
assume !b
assert c <= 12
}

```

The inlined program verifies. Indeed, the assertion assert c <= 12 (line 23) holds since we do at most two iterations. However, if we do not use depth in a loop invariant, then there is no way to write a loop invariant which would make the same assertion true in the bounded program, on line 20. Indeed, we only know that \(c=0\) before the first assertion, \(c\) is between 1 and 6 after the first iteration, and between 2 and 12 after the second one. Since we do not have any index (except depth) for knowing in which iteration we are, it is not possible to write the intermediate assertion saying that \(c\) is between 1 and 6.

This example shows, therefore, that we need indices to be able to speak about a particular iteration. Indeed, the annotation \(A\) we want in our completeness property cannot refer to the variable depth. The same reasoning also applies for recursive methods, for which we also need indices.

\subsection*{7.2.2 Permission gap}

The lack of indices is not the only cause of incompleteness. Another cause, at least in Viper, is a gap in describing permissions.

Take the example shown on Listing 7.5 and Listing 7.6. Listing 7.5 shows the original annotated program. The purpose of lines 4 to 6 is to get some permission to \(x . f\) which is strictly greater than zero, and less than half, but for which there does not exist any lower bound strictly greater than zero. More precisely, after line 6 , we know that \(\exists n . n>=2 \wedge \operatorname{perm}(x . f)==\frac{1}{n}\). In particular, \(\operatorname{perm}(x . f)>0\) and \(\operatorname{perm}(x . f)<=1 / 2\).

It is impossible to perfectly describe this permission amount in the precondition of callee, since we cannot refer to \(n\). We then call the function callee, and want to assert that the permission held to \(x . f\) is not zero, and less than half, which would be true with the same permission amount as before. The question boils down to what should be the precondition and postcondition of the method callee.
callee is a simple method, which only asserts something that is almost a tautology. Indeed, as long as we have some permission to \(x . f\), this assertion verifies. Therefore, we just need some permission in the precondition. However, we do not have a lower bound on the amount of permission we can transmit to callee. Moreover, we cannot use \(n\) in our precondition. Therefore, any possible precondition to make the method callee verify has to use a wildcard. wildcard in Viper means some permission amount, strictly between 0 and \(1 .^{2}\) To make line 8 verify, we need at least some permission amount, which is also less than half, but we do not know how much permission we have in callee.

\footnotetext{
\({ }^{2}\) As described in [1], this definition corresponds to inhaling a wildcard. Exhaling a wildcard results in exhaling some permission amount, strictly between 0 and the current permission held to the relevant location.
}

Listing 7.5: Original annotated program.
```

field f: Int
method initial_method(x: Ref) {
var n: Int
assume n >= 2
inhale acc(x.f, 1/n)
callee(x)
assert perm(x.f) > none \&\& perm(x.f).<=1/2
}
method callee(x: Ref)
requires acc(x.f, wildcard)
ensures acc(x.f, wildcard)
{
assert x.f < 0 || x.f >= 0
}

```
Listing 7.6: Inlined program (bound of 1)
```

field f: Int
method initial_method_inlined(x: Ref) {
var n: Int
assume n >= 2
inhale acc(x.f, 1/n)
assert x.f < 0 || x.f >= 0
assert perm(x.f) > none \&\& perm(x.f) <= 1/2
}

```

We therefore can only use a wildcard to give back some permission to the initial method. However, even with this annotation for the callee method, line 8 cannot verify. Indeed, by using wildcard, we forget the precise amount of permission to \(x\).f. In particular, we lost the information that this permission amount is less than half. Therefore, there does not exist any annotation for callee such that this program verifies.

On the other hand, the inlined program shown on Listing 7.6 verifies, the reason being that the amount of permission is not forgotten by the transmission to a method. This is, therefore, another example of incompleteness in Viper.
There exist several possibilities to bridge this gap:
1. Instrument the program with ghost variables, to record the permission amounts transmitted. These variables would be transmitted to the methods via their arguments, and would be assigned using the perm feature.

Listing 7.7: Original program.
```

field f: Int
method test(x: Ref) {
inhale acc(x.f, wildcard)
var p: Perm
var i: Int := 0
while (i < 1)
invariant ...
{
p := perm(x.f)
i := i + 1
}
assert p == perm(x.f)
}

```
Listing 7.8: Inlined program (bound of 1).
field f: Int
method inlined(x: Ref) \{
    inhale acc(x.f, wildcard)
    var p: Perm
    var i: Int := 0
    if (i<1)
    \{
        p := perm(x.f)
        i := i + 1
        assume ! (i < 1)
    \}
    assert \(\mathrm{p}==\operatorname{perm}(\mathrm{x} . \mathrm{f})\)
\}

Figure 7.2: Existential permissions require the soundness condition.
2. Use abstract read permissions [9, 22].
3. Add support for existential permissions in annotations. This would mean, for example, accept the use of syntactic assertions such as \(\exists n . n \geq\) \(2 \wedge \operatorname{acc}(x . f, 1 / n)\) as annotations. This would require giving a semantics to inhaling and exhaling assertions with existential permissions, and extending the definition of supported assertions, since assertions with existential permissions would not be supported in general.

Remark 7.2 The third solution, existential permissions, is not sufficient to get the completeness property, as we show in Figure 7.2. Even with existential permissions, one cannot write an invariant such that line 13 of the original program (Listing 7.7) verifies. However, the inlined program shown on Listing 7.8 verifies. A reason is that the body of the loop is not framing (and not even mono). It seems, however, that it is sufficient to prove the completeness property under the soundness condition. This is not a problem, since the purpose of this completeness result is to show that inlining under the soundness condition is a good approximation of what happens in the original program.

It is not clear which solution is the best. However, in Chapter 8, we tend towards the third solution for Viper.

\subsection*{7.3 Sketch of completeness}

We have seen two examples of incompleteness in Viper. We now explore completeness, in this section, for programs where we do not have these issues. Since we have not proven any completeness result yet, we only
show a possible result of completeness, with a sketch of a proof. This result is based on an assumption regarding the strongest postcondition: That is, we assume that we can take a statement and compute its strongest postcondition. We then use this assumption to describe how we would construct an annotation for the program, and why the bounded program verifies using this annotation.

The previous sections were mainly about Viper. In this section, we consider completeness in our general framework.

\subsection*{7.3.1 Strongest postcondition assumption}

The strongest postcondition assumption is formally defined as follows:

\section*{Definition 7.3 Strongest postcondition assumption}
\[
\exists S P .\left(\forall s . \operatorname{ver}_{P r}(\{u\}, s) \Longrightarrow \operatorname{sem}_{P r}(\{u\}, s)=\operatorname{Inh}(S P(s))\right)
\]

This assumes the existence of a function \(S P\) which takes a statement as input, and outputs a semantic assertion, which is similar to a strongest postcondition (hence the name \(S P\) ) in the following sense: Executing the statement \(s\) from the empty state (under the condition that it verifies) gives the minimal satisfying set of \(S P(s)\), that is \(\operatorname{Inh}(S P(s))\). Therefore, executing \(s\) from an empty state is almost equivalent to inhaling \(S P(s)\), that is
\[
s \approx \text { inhale } S P(s)
\]

These two statements are in general not equivalent in terms of semantics. Indeed, take \(s=\operatorname{var} x\). s declares the variable \(x\), something which inhaling an assertion cannot do. Indeed, the semantics of inhale \(P\) (for a well-formed assertion) is defined for an initial state \(\varphi\) only if
\[
\forall i \in \operatorname{Inh}(P) . \sigma(i) \subseteq \sigma(\varphi)
\]

This means, in particular, that inhale cannot define new variables. Hence, inhale \(S P(s)\) is not in general equivalent to \(s\). This is why we express this assumption using \(\operatorname{Inh}(S P(s))\), and not inhale \(S P(s)\).

This strongest postcondition assumption does not hold in general in Viper, because of the permission gap. However, if we bridge this gap, it seems that such a statement is provable. We discuss how to prove this assumption for Viper in Section 8.6.

\subsection*{7.3.2 Constructing the annotation}

The idea is the following. The strongest postcondition function, \(S P\), returns an assertion which essentially captures all the information available in the
inlined program. Therefore, adding annotations constructed with this \(S P\) function should provide the bounded program with the information used by the inlined program, thus making the verification of the bounded program equivalent to the verification of the inlined program.

We show through two examples, in Figure 7.3, how to construct an annotation such that the bounded program verifies, using the strongest postcondition assumption. We assume the strongest postcondition assumption, and we assume that all loops and method calls are indexed: That is, there is a way to uniquely refer to one instance of a method call or a specific loop iteration in an assertion. The idea is to construct annotations which correspond to the beginning and end of every method call, and the beginning and end of every loop iteration, in the inlined program.

The first example, Listing 7.9, deals with constructing annotations for a method. The inlined version of this program is shown on Listing 7.10, where \(s y\) corresponds to \(s x\) renamed with \(y\) instead of \(x\), and \(s z\) corresponds to \(s x\) renamed with \(z\) instead of \(x\). We have a simple method callee, with two arguments, an integer \(i\) and a variable \(x\) of type \(T\). We use \(i\) as an index, considering in this case that \(i=0\) during the first method call, and that \(i=1\) during the second method call.

We construct a precondition and a postcondition for each method call, two of each in this case. We use our \(S P\) function on the inlined program to do that. Before the first method call, \(i=0\), and only \(s 1\) has been executed. We therefore add the precondition \(i==0==>S P(s 1)\). This precondition should be satisfied before the method call, since the inlined program verifies until this point. The renamed method body, sy should then verify, and the set of states afterwards should be \(S P(s 1 ; s y)\), as hinted on the inlined program. We therefore add the postcondition \(i==0==>S P(s 1 ; s y)\) to the method. We do the same for the second method call, as shown on the figure (red before the method call, and black afterwards), which gives us the precondition \(i==1==>S P(s 1 ; s y ; s 2)\) and the postcondition \(i==1==>S P(s 1 ; s y ; s 2 ; s z)\).

Writing several preconditions (resp. postconditions) in Viper has the effect of conjoining them. Our two preconditions (resp. postconditions) are mutually exclusive (since \(i\) is either 0 or 1 ). Therefore, our reasoning for the first method call should also work for the second one.

We construct loop invariants similarly, as illustrated on Listing 7.11 (annotated program) and Listing 7.12 (inlined program). We assume that \(i\) is an index for the loop, incremented at every iteration, and starting at 0 . As illustrated in blue in Figure 7.3, we first write a loop invariant which holds when entering the loop, \(i==0==>S P(s 0)\). We then construct loop invariants which have to hold after each iteration, taking into account one more execution each

Listing 7.9: Example for a method.
Listing 7.10: Inlined.
```

method initial_method() {
s1
callee(i, y)
s2
callee(i, z)
s3
}
method callee(i: Int, x: T)
requires i == 0 ==> SP(s1)
requires i == 1 =>> SP(s1 ; sy ; s2)
ensures i == 0 ==> SP(s1 ; Sy )
ensures i == 1 ==> SP(s1 ; sy ; s2 ; sz )
{
Sx
}

```
```

method initial_method(n: Int) {
s0
while (i < n)

```

```

        invariant i == 1 ==> SP(s0 ; s) if (i < n) {
        invariant i == 2 => SP(s0 ; s ; s) s i in i=2
    {
        s
    }
    ...
    }
}

```

Figure 7.3: Illustration of how to construct annotations using the strongest postcondition assumption.
time, such as \(i==1==>S P(s 0 ; s), i==2==>S P(s 0 ; s ; s), \ldots\) We stop when we reach the inlining bound.

This is the current state of completeness. The formalization of everything presented here is planned in the future.

There still remains one gap between the examples of incompleteness presented earlier, and the proof as sketched here. For this proof, we assume that every method call is indexed, whereas examples of incompleteness we can find (for the moment) are only about recursive methods which are not indexed. We therefore need to continue investigate whether it is possible to write annotations in the case of several method calls without indices, or exhibit a counter-example.

We have now proved a soundness result in our framework, and sketched a completeness result assuming the existence of a strongest postcondition function. The next chapter describes how to define the parameters for instantiating our general framework to get a model of the Viper language, and also discusses (with a sketch of a proof) the strongest postcondition assumption in Viper.

\section*{Chapter 8}

\title{
Instantiating the Parametric Language with Viper
}

Chapter 4 and 5 define a general model of a language, whose semantics is built on a separation algebra. In Chapter 6, we expressed and proved the soundness of inlining under a soundness condition in this language. Moreover, we have informally expressed a completeness property for inlining in Chapter 7, and sketched a proof for it. By inputting the right parameters, this general framework can be instantiated as a model for concrete verification languages. Furthermore, this parametric language is strongly inspired by the Viper language. This chapter shows how to instantiate our framework to get a model for a simplified version of the Viper language.

Section 8.1 defines the simplified version of Viper we consider. Section 8.2 defines the separation algebra and the store interface for this simplified version of Viper. Section 8.3 discusses assertions. Section 8.4 shows an example of how to define a rename interface which satisfies all requirements. The core of this rename interface has been formalized and proved in Isabelle/HOL (see Appendix A). Section 8.5 discusses custom statements and the custom interface. Section 8.6 sketches how to prove the strongest postcondition assumption, to leverage the completeness of inlining result from Chapter 7. Finally, Section 8.7 briefly discusses how to use the theoretical results from the parametric language and the instantiation of this model for the Viper language to create two useful features for Viper: The early error detection feature described in the introduction, and a feature to speed up re-verification using caching.

\subsection*{8.1 A simplified version of Viper}

There are several reasons for working in this chapter with a simplified version of Viper. First, Viper does not have clearly defined semantics. Viper is a
powerful intermediate language for verification, with powerful features such as functions and predicates, and giving a formal semantics for these features is beyond the scope of this thesis. Second, the core of this thesis is the parametric language with the theoretical results. This parametric language with its requirements will be modified in the future, thus this chapter is more a proof-of-concept than a real instantiation.

In this simplified version, we do not permit referring to previous states, we do not accept inhale-exhale assertions in annotations, we do not consider typing for variables, and we ignore powerful features such as functions, predicates and magic wands.

\subsection*{8.1.1 Disallowing references to previous states}

As explained in the background chapter (Section 2.2.2), Viper semantics can be defined as a trace semantics, to enable referring to past heaps. This is mostly done through two features. Labels, which record the heap at a certain point of the execution, and old expressions, with or without a label, referring to the heaps of past states. In this simplified version, we do not allow the use of labels and old-expressions.

However, it seems possible to keep track of a trace in a state, by adding a fourth component. This fourth component would be a finite mapping from labels to states. Moreover, two states could be added if they agree on the states which are mapped from the same labels. The sum of two states would simply be the mapping obtained by combining both. This idea has to be explored more, to know if this is sufficient, and in general to know what happens with assertions, and the mono and framing properties.

\subsection*{8.1.2 Inhale-exhale assertions}

As explained in [1], inhale-exhale assertions \([A, B]\) are assertions which behave like \(A\) when inhaled, and like \(B\) when exhaled. These assertions can be used in a postcondition to encode a leak check, or to enhance verification by using properties justified externally [15]. It is quite clear that inhale-exhale assertions cannot be encoded as semantic assertions with the same meaning. We therefore disallow the use of inhale-exhale assertions.

\subsection*{8.1.3 Types of variables}

Our parametric language does not consider types of variables. Indeed, we define everything as if we had only type of variable.

Types of variables could be encoded in the store of a state, adding the type to the mapping. They could otherwise be checked as part of the wfProg function, which checks whether a program is well-formed. On top of that, we would
have to redefine the meaning of the \(h\) function, used in the semantics of var and havoc. We would also have to add a type of types in our parameters, and add a type when declaring variables with a var statement. This could be done in a future work, but is currently disallowed.

\subsection*{8.1.4 Other features}

As explained at the beginning of this section, predicates, magic wands and functions are examples of powerful features which lack clearly-defined semantics. This is why we currently disallow these features.

However, it seems that (recursive and non-recursive) predicates and magic wands could be encoded as part of another permission mask, and would not complicate the instantiation too much. This could be explored in the future.

\subsection*{8.2 Separation algebra, variables and store}

We now define our separation algebra, which captures Viper's state model.

\section*{Definition 8.1 Set of Viper states \(\Sigma\), store \(\sigma\) and impure part \(C\)}

Let \(V\) be the set of all strings (names of variables), Val be the set of possibles values for variables (at least integers, rationals and booleans), Addr be a set of integers (addresses in the heap), and Field be the set of fields defined by the program (strings).

We define the following types:
\[
\begin{aligned}
\text { Permissions } & :=\mathcal{Q} \cap[0,1] \\
\text { Stores } & :=V \rightharpoonup_{\text {fin }} \text { Val } \\
\text { PermissionMasks } & :=\text { Addr } \times \text { Field } \rightarrow \text { Permission } \\
\text { Heaps } & :=\text { Addr } \times \text { Field } \rightharpoonup_{\text {fin }} \text { Val }
\end{aligned}
\]
\(\Sigma\) is the set of all states \((s, \pi, h) \in\) Stores \(\times\) PermissionMasks \(\times\) Heaps such that
1. \(\{(0, f) \mid \pi(0, f)>0\}\) is finite.
2. The heap \(h\) is only defined on \(\{(o, f) \mid \pi(o, f)>0\}\).

Let domain be the function giving the domain of a mapping. The store of a state is
\[
\sigma((s, \pi, h)):=\operatorname{domain}(s)
\]

Let \(s \varnothing\) be the empty store (empty mapping). The impure part of a state is then
\[
C((s, \pi, h)):=\left(s_{\varnothing}, \pi, h\right)
\]

\section*{Definition 8.2 Combination of stores}

Let domain be the function giving the domain of a mapping. Two stores \(s_{1}\) and \(s_{2}\) can be combined, written \(s_{1} \# s_{2}\), if and only if
\[
\forall x \in \operatorname{domain}\left(s_{1}\right) \cap \operatorname{domain}\left(s_{2}\right) \cdot s_{1}(x)=s_{2}(x)
\]

In this case, the combination of two stores is then defined as
\[
\begin{aligned}
& \operatorname{domain}\left(s_{1} \oplus s_{2}\right)=\operatorname{domain}\left(s_{1}\right) \cup \operatorname{domain}\left(s_{2}\right) \\
& \forall x \in \operatorname{domain}\left(s_{1} \oplus s_{2}\right) \cdot\left(s_{1} \oplus s_{2}\right)(x)= \begin{cases}s_{1}(x) & \text { if } x \in \operatorname{domain}\left(s_{1}\right) \\
s_{2}(x) & \text { otherwise }\end{cases}
\end{aligned}
\]

\section*{Definition 8.3 Combination of permission masks}

Two permission masks \(\pi_{1}\) and \(\pi_{2}\) can be added, written \(\pi_{1} \# \pi_{2}\), if and only if
\[
\forall(o, f) \in \text { Addr } \times \text { Field. } \pi_{1}(o, f)+\pi_{2}(o, f) \leq 1
\]

In this case,
\[
\forall(o, f) \in \text { Addr } \times \text { Field. }\left(\pi_{1} \oplus \pi_{2}\right)(o, f)=\pi_{1}(o, f)+\pi_{2}(o, f)
\]

\section*{Definition 8.4 Combination of heaps}

Let domain be the function giving the domain of a mapping. Two heaps \(h_{1}\) and \(h_{2}\) can be added, written \(h_{1} \# h_{2}\), if and only
\[
\forall(o, f) \in \operatorname{domain}\left(h_{1}\right) \cap \operatorname{domain}\left(h_{2}\right) \cdot h_{1}(o, f)=h_{2}(o, f)
\]

In this case, the combination of two heaps is then defined as
\(\operatorname{domain}\left(h_{1} \oplus h_{2}\right)=\operatorname{domain}\left(h_{1}\right) \cup \operatorname{domain}\left(h_{2}\right)\)
\(\forall(o, f) \in \operatorname{domain}\left(h_{1} \oplus h_{2}\right) .\left(h_{1} \oplus h_{2}\right)(o, f)= \begin{cases}h_{1}(o, f) & \text { if }(o, f) \in \operatorname{domain}\left(h_{1}\right) \\ h_{2}(o, f) & \text { otherwise }\end{cases}\)
Definition 8.5 Addition of states \(\oplus\) and neutral element \(u\)
Two states \(\left(s_{1}, \pi_{1}, h_{1}\right)\) and \(\left(s_{2}, \pi_{2}, h_{2}\right)\) can be added if and only if
\[
\left(s_{1}, \pi_{1}, h_{1}\right) \#\left(s_{2}, \pi_{2}, h_{2}\right) \Longleftrightarrow s_{1} \# s_{2} \wedge \pi_{1} \# \pi_{2} \wedge h_{1} \# h_{2}
\]

In this case, the sum of two states is defined as
\[
\left(s_{1}, \pi_{1}, h_{1}\right) \oplus\left(s_{2}, \pi_{2}, h_{2}\right)=\left(s_{1} \oplus s_{2}, \pi_{1} \oplus \pi_{2}, h_{1} \oplus h_{2}\right)
\]

Let \(s_{\varnothing}\) represent the empty store, that is an empty mapping. Let \(h_{\varnothing}\) represent the empty heap, that is also an empty mapping. Let \(\pi_{0}\) be the zero mask, that is
\[
\forall(o, f) \in \text { Addr } \times \text { Fields. } \pi_{0}(o, f)=0
\]

The neutral state \(u\) is then defined as
\[
u:=\left(s_{\varnothing}, \pi_{0}, h_{\varnothing}\right)
\]

All requirements described in Chapter 4 can be proved from these definitions. In particular, the core of a state corresponds exactly to the store:

Lemma 8.6 Core of a state
\[
|(s, \pi, h)|=\left(s, \pi_{0}, h_{\varnothing}\right)
\]

\subsection*{8.3 Assertions}

As we already discussed in Section 5.3.3, there is a mismatch between syntactic assertions from Viper and semantic assertions. In this section, we do not dive deep into syntactic assertions. We first sketch a way for defining an encoding from syntactic to semantic assertions, and how to define the wellDefinedAssert function. We then briefly discuss why syntactic annotations are encoded into supported and intuitionistic semantic assertions.

\subsection*{8.3.1 From syntactic to semantic assertions}

As we saw, different syntactic assertions in Viper must be encoded into the same semantic assertion. Let us denote SynAssert the set of all syntactic assertions one can write in Viper.

We encode syntactic assertions into semantic ones using the following makeSemantic function:

Definition 8.7 Encoding a syntactic assertion into a semantic one
The function makeSemantic takes an element from SynAssert and returns the semantic assertion such that
\[
\text { makeSemantic }(A)(\varphi):= \begin{cases}\top & \text { if } A \text { is well-defined and true in } \varphi \\ \perp & \text { otherwise }\end{cases}
\]

That is, the corresponding semantic assertion is true when the syntactic assertion is both well-defined and true, and false otherwise. For example, this means that we encode the syntactic assertion false and \(1 / 0==1\) into the same semantic assertion, the one which is always false, hence
\[
\operatorname{makeSemantic}(1 / 0==1)=\operatorname{makeSemantic}(\mathbf{f a l s e})
\]

Another example is
\[
\text { makeSemantic }(\mathrm{x} . \mathrm{f}==\mathrm{x} . \mathrm{f})=\text { makeSemantic }(\mathbf{p e r m}(\mathrm{x} . \mathrm{f})>0)
\]

Indeed, x.f == x.f is always true when it is defined, but it is defined only when some permission amount strictly greater than zero is held to the heap location \(x\).f.

We define the wellDefinedAssert function, which encodes whether a semantic assertion is well-defined in a state, as follows:

Definition 8.8 Well-defined assertion
\[
\begin{aligned}
& \text { wellDefinedAssert }(P, \varphi) \\
\Longleftrightarrow & (\exists A \in \operatorname{Syn} \text { Assert. makeSemantic }(A)=P \wedge A \text { is well-defined in state } \varphi)
\end{aligned}
\]

This definition means that if it is possible to rewrite the syntactic assertion \(A\) into \(A^{\prime}\) such that both assertions are encoded into the same semantic assertion, and one of them is well-defined in the state \(\varphi\), then their common encoding into a semantic assertion is also well-defined in the state \(\varphi\). In a way, it acts as if any syntactic assertion is rewritten into a minimal syntactic assertion which is equivalent. As an example, the semantic assertion corresponding to \(\mathrm{x} . \mathrm{f}=\mathrm{x}=\mathrm{f}\) is always well-defined, since perm(x.f) \(>0\) is always welldefined.

As discussed in Section 5.3.3, there is an inherent mismatch in trying to encode syntactic assertions into semantic ones. This definition is therefore not perfect. A consequence of this definition is that if we have well-definedness in Viper, we have well-definedness in the parametric language. Another one is that if we define a function which returns the set of variables read by a syntactic assertion readSynAssert, we have
\[
\operatorname{readAssert}(\text { makeSemantic }(A)) \subseteq \operatorname{readSynAssert}(A)
\]

\subsection*{8.3.2 Annotations are supported and intuitionistic}

We only give a high view of why annotations are encoded into well-formed semantic assertions, since a formal proof would require to first define syntactic assertions, and then to formally define the meaning of self-framing assertions (following [20]).

First, annotations cannot use the keywords perm and forperm. Moreover, Viper disallows implications with accessibility predicates on the left side. For example, the assertion \(\operatorname{acc}(x . f)==>\ldots\) is not valid in Viper. Using these restrictions, it seems possible to show that Viper annotations are encoded into intuitionistic semantic assertions.

Furthermore, annotations are self-framing. As explained in the background chapter (Section 2.2.1), a self-framing assertion is an assertion which carries at least permissions for the locations it reads. Moreover, in Viper, it is not possible to write a disjunction with accessibility predicates. For example, the syntactic assertion acc(x.f) || acc(y.f) is not allowed in Viper. The corresponding semantic assertion would not be supported.

Let us describe a bit more why a self-framing assertion in Viper is supported, and why assertions which are not self-framing can be not supported. Consider \(P:=\operatorname{acc}(x . f, 1 / 2) \& \& x . f==5\) and \(Q:=x . f==5 . P\) is self-framing, whereas \(Q\) is not. \(P\) is encoded into a supported assertion. Indeed, any state satisfying \(P\) has at least half permission to \(x . f\), and \(x . f==5\). Therefore, all states have the same support, the state \(\varphi\) which has only half permission to \(x . f\), and where \(x . f==5\). A state satisfies \(P\) if and only if it is stronger than \(\varphi\). On the other hand, \(Q\) is encoded into an assertion which is not supported. Indeed, a state \((s, \pi, h) \in \Sigma\) satisfies \(Q\) if and only if \(h(x, f)=5\). This implies \(\pi(x, f)>0\). However, one cannot find a support for this state, since this support must satisfy \(\pi(x, f)>0\), and there does not exist any lower bound for \(\pi(x, f)\) other than zero.
These restrictions on Viper annotations therefore imply that they are encoded into well-formed (supported and intuitionistic) semantic assertions. This result should be formally proved in the future. This means that the semantics of inhale and exhale statements in Viper have to match the ones from the parametric language only for annotations which are encoded into well-formed semantic assertions. We describe the behavior of inhale and exhale for other kinds of assertions in the custom inferface, in Section 8.5.

\subsection*{8.4 Rename interface}

We show in this section one way of instantiating the rename interface, which satisfies all requirements. Viper variables are strings, and strings are countable, they are therefore in bijection with natural numbers. To simplify the formalization, we base our renaming framework on natural numbers. It is easy to transform this into a renaming interface based on strings, using a bijection between strings and numbers.
We first define two ways of renaming a natural number into another, depending on the renaming quadruple. We then show how to invert a renaming quadruple. We finally describe how to lift the function renaming a natural number to construct the different functions of the rename interface. The core part of this section, namely the renaming of natural numbers and the inversion of a renaming quadruple, have been formalized in Isabelle/HOL (see Appendix A), with the corresponding results (Requirement 5.37).

\subsection*{8.4.1 Renaming an element}

A renaming quadruple is a quadruple of lists of natural numbers
\[
(\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{a v o i d}, \overrightarrow{\text { domain }})
\]

Elements of \(\overrightarrow{\text { old }}\) should be replaced by the corresponding elements of \(\overrightarrow{n e w}\),


It is well-formed if and only if \(\overrightarrow{\text { old }}\) and \(\overrightarrow{n e w}\) have the same length, elements of \(\overrightarrow{o l d}\) are distinct, and elements of \(\overrightarrow{n e w}\) are distinct.

We distinguish two cases for renaming. The first case is a really special case we mostly use for inverting a renaming quadruple. The second one is a general case.

First case: old and new correspond to the same set, avoid is empty
This case is a special case, whose only purpose is to make it simple to invert a renaming quadruple. We only use this case when there is no number to avoid in the domain \((\overrightarrow{\text { avoid }}=[])\), and when \(\overrightarrow{\text { old }}\) and \(\overrightarrow{n e w}\) contain exactly the same set of numbers. Renaming in this case is simple:

Definition 8.9 Renaming a number (first case)
\[
\text { rename }_{1}((\overrightarrow{\text { old }}, \overrightarrow{n e w},[], \overrightarrow{\text { domain }}))(x):= \begin{cases}\overrightarrow{n e w}[i] & \text { if } \exists i<\text { length }(\overrightarrow{\text { old }}) \cdot x=\overrightarrow{\text { old }}[i] \\ x & \text { otherwise }\end{cases}
\]

If \(x \in \overrightarrow{o l d}\), then we rename it to its corresponding element in \(\overrightarrow{n e w}\) (we do not have any choice). In the other case, this renaming function is the identity. rename \(_{1}\) is clearly a bijection, and fulfills the requirements (avoids what there is to avoid, namely nothing).

\section*{Second case: General case}

This general case is less simple. We need to avoid renaming numbers from \(\overrightarrow{\text { domain }}\) into numbers from \(\overrightarrow{\text { avoid }}\), and we need to make sure our renaming is bijective.

This renaming function in the general case is built on the following lemma:
Lemma 8.10 Let \(\vec{a}\) and \(\vec{b}\) be two lists of numbers, such that
1. length \((\vec{a})=\) length \((\vec{b})\).
2. Elements of \(\vec{a}\) are distinct.
3. Elements of \(\vec{b}\) are distinct.

Then there exists a function \(f: \mathbb{N} \mapsto \mathbb{N}\) such that
1. \(f\) is a bijection.
2. \(\forall i \in \mathbb{N} . i<\operatorname{length}(\vec{a}) \Longrightarrow f(\vec{a}[i])=\vec{b}[i]\)
3. \(\exists M \in \mathbb{N} . \forall x \geq M \cdot f(x)=x\)

That is, with a well-formed renaming quadruple, we can find a bijective function \(f\) which maps \(\overrightarrow{o l d}\) to \(\overrightarrow{\text { new }}\) and which behaves like the identity function above a certain threshold.
We can now define the general case of renaming a number. \(\operatorname{sum}(\vec{l})\) denotes the sum of all elements of the list \(\vec{l}\).

Definition 8.11 Renaming a number (second case)
Let \(t:=(\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{a v o i d}, \overrightarrow{\text { domain }})\) be a well-formed renaming quadruple. Let \(\vec{a}\) be a list of distinct natural numbers, containing exactly the numbers from \(\overrightarrow{\text { old }}\) and \(\overrightarrow{\text { domain. Let } \vec{b}}\) be the list such that
1. length \((\vec{a})=\) length \((\vec{b})\)
2. \(\forall i, j \in \mathbb{N} . i<\operatorname{length}(\overrightarrow{\text { old }}) \wedge j<\operatorname{length}(\vec{a}) \wedge \vec{a}[j]=\overrightarrow{\text { old }}[i] \Rightarrow \vec{b}[j]=\) \(\overrightarrow{n e w}[i]\)
3. \(\forall j \in \mathbb{N} . j<\operatorname{length}(\vec{a}) \wedge \vec{a}[j] \notin \overrightarrow{\text { old }} \Rightarrow \vec{b}[j]=\vec{a}[j]+\operatorname{sum}(\overrightarrow{n e w})+\) \(\operatorname{sum}(\overrightarrow{\text { avoid }})+1\)
It follows that elements of \(\vec{b}\) are also distinct. Let \(f\) be a function as described in Lemma 8.10 for \(\vec{a}\) and \(\vec{b}\). Then
\[
\text { rename }_{2}((\overrightarrow{o l d}, \overrightarrow{n e w}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }}), x):=f(x)
\]
rename \(_{2}\) is a bijection, thanks to Lemma 8.10. Moreover, it maps elements of \(\overrightarrow{\text { old }}\) to elements of \(\overrightarrow{n e w}\). Finally, it maps elements from \(\overrightarrow{\text { domain }}\) to elements which are not in \(\overrightarrow{\rightarrow a v o i d}\). Indeed, since we use natural numbers and we shift
 it follows that the shifted element are greater than all elements in \(\overrightarrow{\mathrm{new}}\) and

Another important property of this renaming function rename \({ }_{2}\) is that it is equivalent to the identity function after a threshold, which makes it easy to invert.

\section*{Combining both renaming functions into one}

Our rename function is defined as follows:
Definition 8.12 Renaming a number
\(\operatorname{rename}((\overrightarrow{\text { old }}, \overrightarrow{\text { new }}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }}), x):=\left\{\begin{array}{c}\text { rename }_{1}((\overrightarrow{\text { old }}, \overrightarrow{n e w},[], \overrightarrow{\text { domain }}), x) \\ \left.\left(\begin{array}{c}\text { if } \\ \text { avoid } \\ \text { ava }\end{array}\right] \wedge \operatorname{set}(\overrightarrow{\text { old }})=\operatorname{set}(\overrightarrow{\text { new }})\right) \\ \text { rename }((\overrightarrow{\text { old }, ~}, \overrightarrow{\text { new }}, \overrightarrow{a v o i d}, \overrightarrow{\text { domain }}), x) \\ \text { (otherwise })\end{array}\right.\)

\subsection*{8.4.2 Inverting a renaming quadruple}

We have defined a renaming function, which works differently on two different cases. However, these two cases have in common that they behave like the identity function after a certain threshold. We use this property, combined with the first case of the rename function, to invert a renaming quadruple:

Definition 8.13 Let \(t:=(\overrightarrow{\text { old }}, \overrightarrow{n e w}, \overrightarrow{\text { avoid }}, \overrightarrow{\text { domain }})\) be a well-formed renaming quadruple. Let \(M \in \mathbb{N}\) be such that
\[
\forall x \geq M . \operatorname{rename}(t, x)=x
\]

Let \(\vec{l}:=[0 . . M-1]\) be the sorted list containing all numbers from 0 to \(M-1\), thus length \((\vec{l})=M\). Let \(\overrightarrow{l^{\prime}}\) be the list such that
1. length \(\left(\overrightarrow{l^{\prime}}\right)=M\)
2. \(\forall x<M \cdot \overrightarrow{l^{\prime}}[x]=\operatorname{rename}(t, x)\)

Let \(\vec{d}\) be any list of natural numbers (we just need to choose one). Then
\[
\operatorname{renameInv}(t):=\left(\overrightarrow{l^{\prime}}, \vec{l},[], \vec{d}\right)
\]

Since we have length \((\vec{l})=\operatorname{length}\left(\overrightarrow{l^{\prime}}\right)\), and
\[
\operatorname{set}(\vec{l})=\operatorname{set}\left(\overrightarrow{l^{\prime}}\right)=\{x \mid x \in \mathbb{N} \wedge x<M\}
\]
by definition, our inverted renaming quadruple always falls into the first case of the renaming function (since avoid is empty):
\[
\forall x \in \mathbb{N} . \operatorname{rename}(\text { renameInv }(t), x)=\text { rename }_{1}(\operatorname{renameInv}(t), x)
\]

Using this property, we can show that our renameInv function satisfies the required properties:

Lemma 8.14 Let \(t\) be a well-formed renaming. Then renameInv \((t)\) is also wellformed, and
\[
\forall x \in \mathbb{N} . \operatorname{rename}(\operatorname{renameInv}(t), \operatorname{rename}(t, x))=x
\]

\subsection*{8.4.3 Towards the rename interface}

Lifting the previous results to build a rename interface is quite obvious from this point, we therefore do not go into details. We need two functions, toString and toNatural, which respectively convert a natural number to a string and a string to a natural number. These functions are bijections, and in particular they are inverse of each other. Using these two functions, we can
convert any string renaming quadruple into a number renaming quadruple and vice versa. It is straightforward to instantiate the rename and renameInv functions of the interface. Renaming a state is simple. It suffices to use this function to modify the store \(s\) in the following way:
\[
\text { renameState }(t,(s, \pi, h)):=\left(s^{\prime}, \pi, h\right)
\]
where \(\operatorname{domain}\left(s^{\prime}\right)=\{\operatorname{rename}(t, x) \mid x \in \operatorname{domain}(s)\}\) and
\[
\forall x \in \operatorname{domain}(s) \cdot s^{\prime}(\operatorname{rename}(t, x)):=s(x)
\]

The instantiation of renameCustom is also straightforward given a custom type of statements.

\subsection*{8.5 Custom statements}

The purpose of the custom interface is to encode all statements which are not built in the parametric language. As an example, the parametric language defines neither assert statements nor variable assignments. Moreover, it only defines a special case of inhale and exhale. This section illustrates how to use the custom interface. We first show how to define assert and variable assignment, and then discuss how to encode inhale and exhale in the general case.

We assume we have a function (not described here) \(\lambda e . w \operatorname{def}(e, \varphi)\) which is true if and only if \(e\) (an expression or a syntactic assertion) is well-defined in the state \(\varphi\). We also assume we have a function \(\lambda e . e v a l(e, \varphi)\) which evaluates a boolean expression or a syntactic assertion in a certain state. Since the store is a mapping, it is easy to encode a variable assignment using these two functions:

\section*{Definition 8.15 Variable assignment}
\[
\begin{aligned}
& \text { semanticsCustom }_{P r}((s, \pi, h), x:=e) \\
:= & \begin{cases}\{(s[x:=\operatorname{eval}(e,(s, \pi, h))], \pi, h)\} & \text { if wdef }(e,(s, \pi, h)) \wedge x \in \operatorname{domain}(s) \\
\text { Error } & \text { otherwise }\end{cases}
\end{aligned}
\]

A variable assignment \(x:=e\) verifies if and only if an expression is welldefined in the state \((s, \pi, h)\) and \(x\) is already defined by \(s\). If it verifies, then it simply modifies the mapping of \(x\), by evaluating the expression in the state \((s, \pi, h)\).

To define assert, inhale and exhale, we need to embed the type of syntactic assertions into the type \(O\) of custom statements. We can then encode the semantics of assert as follows:

Definition 8.16 Assert
\[
\text { semanticsCustom }_{P r}(\varphi, \text { assert } P):= \begin{cases}\{\varphi\} & \text { if wdef }(P, \varphi) \wedge \operatorname{eval}(P, \varphi) \\ \text { Error } & \text { otherwise }\end{cases}
\]
assert verifies if and only if the assertion is well-defined and true in the state \(\varphi\). If it verifies, it does not modify the state.

We do not give a description for the semantics of inhale and exhale, since they involve some subtleties which are beyond the scope of this thesis. We refer the reader to [6] for a first approach on how to define these semantics, as well as [20] for a more theoretical approach. The important message is that these statements can be encoded using syntactic assertions (embedded in the type \(O\) ) and custom statements, and they behave in general differently than the inhale and exhale statements provided by the parametric language. As an example, inhale acc(x.f, perm(x.f)) encoded with the inhale statement from the parametric language and with a semantic assertion would simply be equivalent to skip, whereas in Viper it would double the permission held to \(x . f\).

\subsection*{8.6 Completeness: Strongest postcondition in Viper}

To leverage the completeness result of Chapter 7, we need to prove the strongest postcondition assumption for Viper. That is, we need to show how to construct an assertion which completely characterizes a Viper statement, and then to prove it. As we saw with the "permission gap" example, this assumption is false in general. We therefore sketch how to prove this assumption, assuming that we can use existential permissions in Viper. \({ }^{1}\) We sketch in this section a proof for the strongest postcondition assumption in Viper.

The idea is to prove the assumption by induction on a statement \(s\), with a strongest postcondition of the following form:
\[
\begin{align*}
& \exists \vec{v} . \text { behavior }(\vec{v}) \wedge  \tag{8.1}\\
& x=v_{0} \wedge y=v_{7} \wedge z=v_{9} \wedge \ldots \wedge  \tag{8.2}\\
& \operatorname{acc}\left(x . f, v_{2}\right) \wedge\left(v_{2}>0 \Rightarrow x . f=v_{5}\right) \wedge \ldots \tag{8.3}
\end{align*}
\]
8.1 fully specifies the behavior of the program (assumptions, conditional branching, permissions inhaled, values assigned...), in terms of existentially quantified variables. 8.2 links the existentially quantified variables \(\vec{v}\) to the

\footnotetext{
\({ }^{1}\) This can easily be done for permissions to field locations, but extending this idea to predicates and magic wands seems more complicated.
}

Listing 8.1: Example of a program for which we construct a strongest postcondition.
```

field f: Int
method example_strongest_post()
{
var m: Int
var x: Ref
var n: Int
assume n >= 2
if (m >= 5) {
n := 2 * n
}
inhale acc(x.f, 1/n)
n := 0
}

```
\[
\begin{align*}
& \exists m_{0}: \text { Int, } x_{0}: \operatorname{Ref}, p_{0}: \text { Perm, } p_{1}: \text { Perm, } x f_{0}: \text { Int, }  \tag{8.4}\\
& n_{0}: \text { Int, } n_{1}: \text { Int }, n_{2}: \text { Int, }  \tag{8.5}\\
& e_{0}: \text { Bool, } e_{1}: \text { Int, } e_{2}: \text { Perm, } e_{3}: \text { Int. }  \tag{8.6}\\
& p_{0}=0 \wedge n_{0} \geq 2 \wedge  \tag{8.7}\\
& \left(e_{0} \Leftrightarrow m_{0} \geq 5\right) \wedge e_{1}=2 * n_{0} \wedge  \tag{8.8}\\
& \left(e_{0} \Rightarrow n_{1}=e_{1}\right) \wedge\left(\neg e_{0} \Rightarrow n_{1}=n_{0}\right) \wedge  \tag{8.9}\\
& e_{2}=1 / n_{1} \wedge p_{1}=p_{0}+e_{2} \wedge e_{3}=0 \wedge n_{2}=e_{3} \wedge  \tag{8.10}\\
& m=m_{0} \wedge x=x_{0} \wedge a c c\left(x . f, p_{1}\right) \wedge  \tag{8.11}\\
& \left(p_{1}>0 \Rightarrow x . f=x f_{0}\right) \wedge n=n_{2} \tag{8.12}
\end{align*}
\]

Figure 8.1: Constructing a strongest postcondition in Viper.
variables of the program \((x, y, z, \ldots)\). Finally, 8.3 links the existentially quantified variables to permissions held to field locations and the values of those fields (if the permission held is greater than 0 ). As discussed in Remark 7.2, an assertion of this shape, using existential permissions, would only satisfy the strongest postcondition assumption under the soundness condition.

To prove the strongest postcondition assumption by induction with this form, we add information to 8.1 , by adding new existential variables and new conjunctions. We then modify 8.2 and 8.3 accordingly.
We show an example of a strongest postcondition we could construct for Listing 8.1 in Figure 8.1. We have existential variables for all variables of
the program, with as many versions as the number of times this variable could potentially be modified. For example, \(n\) has an initial value, and can be modified on lines 11 and 14 , hence three versions \(n_{0}, n_{1}\), and \(n_{2}\). We also have existential variables for evaluating expressions: \(e_{0}\) is the condition of the if statement, \(e_{1}\) represents \(2 * n\) of line \(11, e_{2}\) is the \(1 / n\) of line 13 , and \(e_{3}\) is 0 of line 14 .

All lines except 8.11 and 8.12 specify the behavior of the program. 8.7 specifies that we have initially no permission to \(x . f\), and encodes the assume statement (line 9), 8.8 and 8.9 encode the conditional branching with its assignment (lines 10 to 12). Finally, 8.10 encodes the last two lines of the method, the inhale statement and \(n:=0\).

As we can see from this example, the size of this postcondition can grow really quickly. This is not an issue since we are only interested in a theoretical construction which allows us to show completeness. Other difficulties arise when trying to prove this assumption, which are not shown in this example, but which can be dealt with, amongst which:
- Aliasing: When assigning a value to \(x . f\), we should encode that this can also modify the value of \(y . f, z . f, \ldots\)
- Havocing when no more permission: When the permission held to a location goes to 0 , we need to forget the value of this location.

Viper magic wands and predicates (which we do not consider in the simplified subset defined in Section 8.1) also present another challenge, since they can hide information which can be lost when they are transferred through annotations. Consider Figure 8.2 as an example for predicates. We have a predicate \(P(x)\), which contains full permission to \(x . f\). Listing 8.2 shows the original annotated program, which does not verify because line 20 fails. We know that \(x . f=5\) because of line 10 . This information is then hidden in the predicate \(P(x)\), line 12. However, since we put this predicate in the loop invariant (line 14), we get back a predicate \(P(x)\), but we are not sure it is the same predicate as the one we gave to the loop through the loop invariant. Therefore, we lost the information that \(x . f=5\).
On the other hand, the inlined program (bound of 1) shown on Listing 8.3 verifies. Indeed, we are sure that we have the same version of the predicate \(P(x)\) throughout the method, so we know \(x . f=5\) after having unfolded the predicate. One way to solve this issue is to uncomment the loop invariant on line 15. This loop invariant make explicit the hidden information \(x . f=5\) from \(P(x)\). We also would have to do something similar for magic wands.

Moreover, even if recursive predicates can be stactically unbounded, we know that the number of times any predicate is unfolded in the bounded program is stactically bounded. Therefore, we only need to encode information into annotations up to a bounded depth, by using nested unfolding.

Listing 8.2: Original annotated program.
```

field f: Int
predicate P(x: Ref) {
acc(x.f)
}
method example(x: Ref)
requires acc(x.f)
{
x.f := 5
var i: Int := 0
fold P(x)
while (i < 1)
invariant P(x)
// \&\& unfolding P(x) in x.f == 5
{
i := i + 1
}
unfold P(x)
assert.x.f.== 5
}

```
Listing 8.3: Inlined (bound of 1 ).
field f: Int
predicate \(\mathrm{P}(\mathrm{x}:\) Ref) \{
    acc(x.f)
\}
method example(x: Ref)
    requires acc(x.f)
\{
    x.f := 5
    var i: Int := 0
    fold \(P(x)\)
    if (i<1)
    \{
                i := i + 1
        assume ! (i<1)
    \}
    unfold \(P(x)\)
    assert \(x . f==5\)
\}

Figure 8.2: Viper predicates can hide information.

To conclude, it seems possible to prove this strongest postcondition assumption in Viper. However, we need to extend assertions with existential permissions. Such a proof would be a complicated one and needs more exploration. \({ }^{2}\)

\subsection*{8.7 Towards two useful features in Viper}

\subsection*{8.7.1 Static inlining for early error detection}

As explained in the introduction, a major motivation for this project is the creation of a useful feature for early error detection for Viper. This feature would not only detect fundamental errors, but could also provide more information on spurious errors (as we explain in Chapter 9).

We can leverage some theoretical results obtained in this thesis to detect fundamental errors in Viper. First, we need to ensure that the soundness condition is satisfied. This mostly requires a way of checking whether a statement is mono or framing.

\footnotetext{
\({ }^{2}\) Another difficulty is dealing with Viper functions, which represent abstraction over expressions.
}

One interesting result is that these properties are stable by sequential composition and non-deterministic conditional branching. Moreover, Viper statements which are not mono (and thus not framing) necessarily use perm, forperm or contain an assume statement with an accessibility predicate. Viper statements which are not framing necessarily satisfy the latter condition, or contain an exhale statement with a wildcard. \({ }^{3}\) Combining these two results gives a straightforward conservative function to check the soundness condition. Leveraging the definition of inlining and the soundness result, we can report fundamental errors.

\subsection*{8.7.2 Speed up re-verification through caching}

This work yields an unexpected byproduct, which relates to caching verification results. Thanks to the modular verification of method calls, modifying the body of a method without modifying its annotation does not require one to re-verify all calls to this method. However, in Viper, it is unclear what we should do when a precondition or a postcondition is modified.

Thanks to the properties mono and framing, we can explore and express answers to this. We give some examples here:
- If we strengthen the precondition of a method, and if this method's body is mono, then we do not need to check again whether the method satisfies its contract.
- If we weaken the precondition of a method and the method still satisfies its contract, then we do not have to check again the program containing the method call if the statement after the method call is mono: Indeed, a weaker precondition with the same postcondition gives a stronger set of states, which still verifies in this case.
- Conjoining (with the separation conjunction) the same assertion to both the precondition and the postcondition of a method: We do not have to check the method again if its body is framing.

This problem will be explored deeper in a future work.

\footnotetext{
\({ }^{3}\) As an example, exhale acc(x.f, wildcard) is mono and not framing, but inhale acc(x.f, wildcard) is framing (and thus mono). See [1] for more details about wildcard.
}

\title{
Extensions to Different Loop Semantics and Inlinings
}

This chapter presents two possible extensions to the results already presented (soundness and completeness), based on Viper. The first section explores three different possibilities for defining the semantics of a loop in Viper. The second section explores three useful ways of inlining, corresponding to different stages of annotating a program.

\subsection*{9.1 Exploration of loop semantics}

Our soundness and completeness results only apply to one possible semantics for loops, as expressed in Definition 5.33. We refer to this semantics of loops as black-box semantics, since the verification of loops in this semantics does not take into account impure resources from outside the loop, except the ones explicitly specified in the loop invariant. For example, if full permission is held to both \(x\). \(f\) and \(y\). \(f\) before the loop, and the loop invariant only specifies \(\operatorname{acc}(x . f)\), then the loop body completely ignores the full permission to \(y . f\), and behaves as if this permission never existed.
However, this black-box semantics is only one of several possibilities to define loop semantics, and our general framework can only deal with this one. In this section, we explore two other possibilities for defining the semantics of loops in Viper. Based on this exploration, it should be possible to also define these different loop semantics in our parametric language, and prove the soundness of inlining for all of them.

This section only focuses on the exploration of different loop semantics in Viper. We first highlight a difference between how Silicon and Carbon, two verifiers for Viper, treat loops. Based on this difference, we present two more possibilities for loop semantics, on top of the black-box semantics. We only present these three different loop semantics, but no result of soundness or

Listing 9.1: Differences of loop treatment between the Viper verifiers Carbon and Silicon.
```

field f: Int
method example(x: Ref, y: Ref, n: Int)
requires acc(x.f, 1/2) \&\& x.f == 5
{
var i: Int := 0
while (i < n)
invariant true
{
inhale acc(x.f, 1/3) Verifies with Carbon
assert x.f == 5
exhale acc(x.f, 1/3) Fails with Silicon
inhale acc(x.f, 1/3)
assert x.f == 5 F Fails with Carbon
exhale acc(x.f, 1/3)
i := i + 1
}
}

```
completeness of inlining have been formalized yet with respect to the two new possibilities.

\subsection*{9.1.1 Semantics of loops in Viper: Silicon and Carbon}

Listing 9.1 presents a Viper program where the two Viper verification backends, Silicon and Carbon, differ in their treatment of the loop. Before entering the loop, the program holds half permission to \(x . f\), and the value of \(x . f\) is 5 .

The Silicon back-end actually treats loops as black-box, and thus line 9 does not verify with Silicon. Indeed, since the loop invariant given here (true) does not give any information, Silicon does not know anything about the value of \(x . f\) inside the loop.

On the other hand, line 9 verifies with the Carbon back-end. It is justified by the fact that the program holds half permission to \(x . f\), and this permission has not been transferred to the loop through the loop invariant. Therefore, this permission is still held by the program state after the loop, and so does \(x\).f \(==5\). The information \(x . f==5\) is true before and after the loop. The reason is that the Carbon back-end assumes that this heap value cannot be modified from inside the loop body, since half permission to this heap location is kept outside the loop. Following this assumption that this value cannot be modified from inside the loop body, Carbon considers this information true inside the loop body.

However, line 12 suprisingly does not verify with Carbon. The practical
reason is that the exhale line 12 removes all permission held (inside the loop body) to \(x . f\), and thus the value of \(x\). \(f\) is havoced. It would however be coherent for line 12 to verify, using the same reasoning as before. This is the idea developed in the second possible loop semantics.

\subsection*{9.1.2 Three coherent ways to treat loops}

We now present three coherent ways to define the semantics of loops in Viper. The first is the black-box semantics, the one we used in our framework, and the one used by the Silicon back-end. The second one is the logical extension of how Carbon treats loops. In Carbon, impure resources not specified in the loop invariant, which are accessible after the loop, are also accessible from inside the loop. Finally, the last loop semantics builds on top of the second one, and argues that giving access to reading permissions from outside the loop is coherent with respect to the role of a loop invariant.

\section*{Black-box semantics}

The black-box semantics for loops is the one from Definition 5.33 and described in Section 2.2.4 of the background chapter. In this semantics, the loop body has only access to the values of variables which were defined before the loop, and the impure resources specified by the loop invariant. It has no access to the impure resources, namely the permissions to and the values of heap locations, which are not specified in the loop invariant.

\section*{Access to knowledge of the heap}

Viper semantics can be seen as based on a global assumption about the program: At any point of the execution, the sum of all permissions held to a field location in a program sums to at most 1 . A corollary is that, if some permission to a field location is kept outside a loop, then the knowledge about this field can be framed around the loop. The method shown on Listing 9.1, for example, keeps half permission to \(x\).f outside the loop. We therefore know that, after the loop, x.f still equals 5 .
Logically extending this global assumption, we could allow the body of a loop to have access to impure resources not specified in the loop invariant. Carbon already does this, but only to a certain degree, as we illustrated on Listing 9.1. In practice, lines 9 and 12 on Listing 9.1 would both verify with this semantics for loops.

This also implies that the permissions held to a field location, inside and outside a loop, should sum to at most 1. Take Listing 9.2 as an example to illustrate this idea. Line 14 does not verify with Carbon. Half permission to \(x . f\) is kept outside the loop, and half permission is given to the loop. Line 12 verifies, since \(x . f\) is known to be 5 outside the loop, with some permission.

Listing 9.2: Example with too much permission inside and outside a loop.
```

field f: Int
method example(x: Ref, n: Int)
requires acc(x.f) \&\& x.f == 5
{
var i: Int := 0
while (i < n)
invariant acc(x.f, 1/2)
// invariant i >= 1 ==> x.f == 4
{
inhale acc(x.f, 1/2)
assert x.f == 5
x.f := 4
assert.in>=1.==> false
i := i + 1
}
}

```

However, at every loop iteration, we inhale enough permission to \(x\). \(f\) to write it, and we assign 4 to it. Therefore, from the second iteration on ( \(i \geq 1\) ), we should know that \(x . f\) is both 4 and 5 , which is incoherent and thus would make line 14 verify. We can make this program verify by uncommenting the loop invariant on line 9 , showing that this behavior is, in a way, already implied by how Carbon treats loops.

Role of a loop invariant and access to read permissions outside the loop
We can take this idea of making what is framed around the loop accessible from inside the loop body even further. Since we frame permissions around the loop, why not also make these permissions accessible from inside the loop body? As an example of what this would mean, take Listing 9.3. To make this program verify, we need in particular to make line 11 verify. In the black-box semantics, we need to uncomment both invariants on line 8 and 9 . In the previous semantics (the one inspired by Carbon), we only need to uncomment the invariant on line 8 . If we would allow access to read permissions outside the loop, then this program would verify without any invariant.

This raises the question of the role of a loop invariant. As explained in the introduction (Chapter 1), we have to annotate loops with loop invariants to make verification decidable. The role of a loop invariant in general is therefore to approximate the behavior of a loop.

In Viper, a loop invariant is not exactly an approximation of the behavior of a loop. A loop invariant in Viper also defines a transfer of resources,

Listing 9.3: Example of a program which would verify in the third loop semantics.
```

field f: Int
method example(x: Ref, n: Int)
requires acc(x.f) \&\& x.f == 5
{
var i: Int := 0
while (i < n)
// invariant acc(x.f, 1/2)
// invariant x.f == 5
{
assert x.f == 5
i := i + 1
}
}

```

Listing 9.4: Example of a loop whose behavior depends on the loop invariant.
```

while (i < n)
invariant ...
{
p := perm(x.f)
i := i + 1
}

```
and especially impure resources. As an example, take the loop shown on Listing 9.4. The behavior of this loop body depends on the loop invariant. Indeed, it assigns to \(p\) the permission held to \(x . f\) inside the loop, and this permission depends on the loop invariant. If the loop invariant is true, then \(p=0\), whereas \(p=1 / 2\) if the loop invariant is acc(x.f, \(1 / 2\) ).

One could think that the loop invariant should not influence the behavior of the loop. In this case, all permissions from outside the loop should be accessible from within the loop. However, a loop invariant should still be allowed to use accessibility predicates (acc(...)), for two reasons:
1. If some permission inside the loop is inhaled or exhaled, then the permission mask is modified. This modification is part of the program, and it should be possible to approximate it using the loop invariant.
2. Viper assumes that if some permission to a heap location is kept outside the loop, its value is not modified inside the loop. This form of framing would be lost if all permissions held outside the loop are accessible from inside the loop. Thus, permissions held outside the loop should only allow reading a heap location, but not writing it. Full permission
inside the loop should still be required to write a heap location.
Therefore, only allowing to read a field location, when some permission to this field location is detained, inside or outside the loop, does not have this issue. Moreover, as shown on Listing 9.3, it would reduce the size of loop invariants. This is another kind of loop semantics to explore.

\subsection*{9.2 Different inlinings: A classification of inlinings in Viper}

In this section, we show that there is not only one useful way to inline a program, but many. In particular, we first focus on Viper programs to illustrate how three different scopings for inlining could be useful, at different stages of annotating the program. These three scopings are related to the three loop semantics presented in the previous section. We present in this section a general way of describing and classifying inlinings, based on two concepts: barriers and scopings.

\subsection*{9.2.1 A chronology of annotating}

In this subsection, we give a simple description of annotating a program in Viper, based on experience, in three stages:

First stage: Writing the program. This is the situation we described in the introduction. We have a program, without any annotation, and we want to know if it is possible to find an annotation such that it verifies.

Second stage: Annotating the program with permissions. Once the program is written, Viper reports a lot of errors because of insufficient permissions. Since no permission is specified in preconditions, postconditions and loop invariants, no permission is transferred to loops and methods. This stage consists of writing all the required permissions to describe all transfers of permission to loops and methods, to remove the spurious errors reported by Viper.

Third stage: The interesting properties. Once the program verifies with respect to permissions, one can begin to write interesting properties, which describe the behavior of the program. This stage consists mostly in adding annotations describing values of variables and fields.

Partly based on this chronology, we define in the following subsections three different scopings for inlining, which correspond to these three stages. Using these different scopings, we could define an inlining function which takes annotations into account, and provides more information about an error to
Listing 9.5: A method call.
method initial() \{
    method initial() \{
    m()
\}
Listing 9.6: An inlined method call.
                                ... Barrier 1: precondition
\(\approx\) framing \(\begin{array}{c:c} & \text { body_m }\end{array}\)
    Barrier 2: postcondition
\}

Figure 9.1: Illustration of the concept of barriers.
the programmer. We first need to define the concept of a barrier. This section is based on the black-box semantics of loops in Viper.

\subsection*{9.2.2 Barrier}

We illustrate the concept of a barrier in Figure 9.1. A barrier acts like a loop invariant at the beginning of a loop: The loop invariant takes some permissions, specified by the annotation, transfer these permissions to the loop body, and let the remaining permissions outside the loop. Similarly, a barrier is parametrized by an annotation, which characterizes a transfer of permissions and information. Listing 9.5 shows a method call, and Listing 9.6 the inlined version of this method call, where we represent two barriers as blue lines. The first barrier is parametrized by the precondition of the method, while the second barrier is parametrized by the postcondition. These two barriers are connected, which also defines what is framed. An inlining simply defines what these barriers filter, what gets through and what is framed, with respect to the annotation they are parametrized with.

The inlining function from Definition 6.3 simply lets everything go through. It acts as if using barriers, where the annotations capture everything about the program state. Since both barriers, in this case, behave in the same way with different annotations, we refer to their scopings as no scoping (defined below). However, it is possible to define other scopings for these barriers.

An example of another scoping is to only let the permissions specified by the annotation go through the barrier. The permissions not specified by the annotation are either framed (in the case of barrier 1), or leaked (in the case of barrier 2). The next subsection defines more precisely such scopings.

This concept of barriers can be extended to loops, with a barrier at the entrance of the loop, a barrier at the exit of the loop, and intermediate barriers in between every iteration.

\section*{9. Extensions to Different Loop Semantics and Inlinings}


Figure 9.2: Degrees of freedom for scoping of barriers

\subsection*{9.2.3 Scoping of a barrier}

The scoping of a barrier defines what gets through and what does not, based on its annotation. What does not get through is either framed (precondition barrier for a method, entrance barrier for a loop) or leaked (postcondition for a method, intermediate and exit barriers for a loop).

A Viper state consists of a store, a permission mask and a heap (see Section 2.2.2). Variables, permissions and heap values are the three types of information we can filter through a barrier. Since variables are accessible from within the body of a loop, we do not filter out any information about variables. We are therefore left with permissions and heap information.

We define two levels of permission scoping, and three levels of heap information scoping, as shown in Figure 9.2. \(P\) represents the whole permission mask, whereas \(P_{A}\) represents the part of the permission mask described by the annotation \(A\). \(H\) represents the whole heap, \(H_{P}\) the part of the heap for which permission is held from \(A\), and \(H_{A}\) represents the heap information exactly defined by \(A\).

Consider, as an example, the assertion \(A:=\mathbf{a c c}(x . f, 1 / 2) * x . f>=3\), and the state \(\varphi\) with full permission to both \(x\) and \(y\), and where \(x . f==5\) and \(y \cdot f==7\). We have, in this settings:
- \(P=\operatorname{acc}(x . f) * \operatorname{acc}(y . f)\), that is full permission to both \(x . f\) and \(y . f\).
- \(P_{A}=\operatorname{acc}(x . f, 1 / 2)\), that is half permission to \(x . f\).
- H represents the full heap, where \(x . f==5\) and \(x . f==7\).
- \(H_{P}\) represents the information \(x . f=5\), but does not contain any information about the value of \(y . f\).
- \(H_{A}\) represents represents the information \(x . f \geq 3\).

Using this classification of information, we define three different scopings, corresponding to the three stages of annotating presented earlier.

No scoping: \(H\) and \(P\). This is the scoping used in Definition 6.3. Since \(H\) is the full heap, and \(P\) the full permission mask, it lets everything go through, and therefore does not frame anything. As we saw, inlining with no scoping barriers is useful when no annotations have been written yet.

Weak Scoping: \(H_{P}\) and \(P_{A}\). The second stage of annotating corresponds to a point where annotations define the transfers of permission. In this case, it is not useful anymore to inline by letting the full permission mask go through a barrier. This scoping therefore lets \(P_{A}\) go through. However, since the annotation does not contain any heap information yet, we give all of the heap information: \(H_{P}\). Our hope is that we can prove a strong soundness result for this scoping, which does not require a soundness condition. Indeed, this soundness result would not speak about annotating the program, but about adding information to the already existing annotation, in the form of store and heap information. It also seems that we can prove a stronger completeness result, where we do not need the soundness condition, and we do not need to bridge the permission gap.

Strong Scoping: \(H_{A}\) and \(P_{A}\). The last stage of annotating corresponds to adding interesting properties, in particular heap information. When all the heap information needed has been specified in the annotation, the only remaining thing one can add is information about the store. It seems, similarly to weak scoping, that we can prove strong soundness and completeness results, speaking about adding only information about the store to annotations.

These three scopings can also be combined to get hybrid forms of inlining. When, for example, one has finished writing a precondition for a method, but no postcondition has been written, we can inline with a strong scoping (or a weak scoping) barrier for the precondition (i.e., taking the precondition into account), and a no scoping barrier for the postcondition (i.e., letting all the permissions and information flow through the barrier).

Moreover, even when no annotation has been written, it is useful to use different scopings for inlining. This can give us more information on an error. We could, for example, report errors with messages such as:
- "Line 10 does not verify because of a permission error, it seems however that there is enough permission in the surrounding scope" (if no scoping verifies, but weak scoping does not).
- "There seems to be insufficient permission in the surrounding scope for line 10 to verify, in the third iteration of the loop" (if no scoping does not verify, with the soundness condition).
- "The assertion on line 10 seems true, but heap information needs to be added to the precondition" (if weak scoping verifies, but strong scoping

\section*{does not).}

These different combinations should be explored in the future, as well as the soundness and completeness of these different scopings, to build a useful feature in Viper.

\section*{Chapter 10}

\section*{Conclusion and Future Work}

\subsection*{10.1 Conclusion}

We have explored static inlining in Viper, defined a soundness property with respect to annotations and fundamental errors, and shown examples of Viper programs where this soundness property does not hold. We have derived from these examples a soundness condition, namely a sufficient condition under which this soundness property holds, based on two properties of statements, mono and framing, as illustrated in Figure 10.1.

We have defined a general framework to abstract Viper, consisting of a


Figure 10.1: Illustration of the soundness and completeness results.
separation algebra and a parametric language. This separation algebra builds on other works on separation algebra, with a difference: We combine pure and impure resources in a single algebra. \({ }^{1}\) This enabled us to define a parametric language centered around the semantics of inhaling and exhaling annotations, mimicking both transmission of pure information and transfer of impure resources.

This parametric language has many degrees of freedom. It can be instantiated as a model of a concrete verification language, given the right input parameters. We have formally defined inlining up to a bound, the soundness condition and the soundness property for this framework. We have formally proved the soundness of static inlining in this framework, and this proof has been mechanized with the proof assistant Isabelle/HOL (Appendix A).

We have also defined and explored completeness of inlining. We have shown examples of Viper programs where completeness does not hold, and identified two main issues, the lack of indices in loops and method calls, and a permission gap. We have also sketched a proof for completeness of inlining in our general framework, based on an assumption of a strongest postcondition.

To show how (and that) our general framework can be instantiated, we have discussed how to instantiate it for Viper. We have shown an example of how to instantiate the rename interface (proved in Isabelle/HOL), how to instantiate the other interfaces to get a model of the Viper language, and how this instantiation can be leveraged to build two useful features in Viper: an early error detection feature based on static inlining, and a cache re-verification feature based on mono and framing.

Finally, we have proposed and discussed two extensions for this work. We have explored three different yet coherent possible definitions of loop semantics. Our work deals with one kind of loop semantics, and could be extended to the other two. Moreover, we have shown that different inlinings are possible, that they correspond to different stages of annotating a program, and that combining them can yield a lot of useful information in the early stages of annotating.

\subsection*{10.2 Future work}

As we explained throughout this thesis, this work has opened many interesting directions, some of which should be addressed in our future work.

\footnotetext{
\({ }^{1}\) As noted in the related work section of Chapter 4, Iris also does this.
}

\subsection*{10.2.1 Improve the current framework and soundness proof}

Loops semantics. As explained in Remark 5.34, the definition of loop semantics in this thesis is slightly different than the one formalized in Isabelle/HOL. The semantics of while (b) inv \(I\{s\}\) for a state \(\varphi\) should havoc modif \((s) \cap \sigma(\varphi)\), but the Isabelle formalization havocs only modif(s), which makes it impossible to declare variables in a loop body. This should be fairly easy to fix.

Variable capturing when inlining As explained in Remark 6.4, the current definition of the inlining function in the case of a sequential composition allows variable capturing. The second recursive call should be modified to disable variable capturing.

Soundness condition. As explained in Remark 6.12, Definition 6.11 defines \(S C_{0}^{P r}(l\), while \((b)\) inv \(I\{s\})\) as mono \(_{P_{r}}(\) assume \(\neg b)\). However, the Isabelle/HOL formalization defines it as \(w f m_{P r}(b) \wedge \operatorname{mono}_{P r}(s)\), because a first version of the proof of soundness required \(s\) to be mono under the soundness condition. The proofs presented in Chapter 6 show that this is no longer needed, and these proofs should replace the ones in our formalization.

Well-formed assertions The current definition of well-formed assertions is too restrictive and disallows some Viper annotations. This definition should be modified as explained in Remark 5.21.

Extended semantic assertions. As explained in Section 5.3.3, there is a mismatch between syntactic and semantic assertions, and our current model is therefore limited. A solution to tackle this issue is to use extended semantic assertions, which can return an error on top of true and false, to model when assertions are not well-defined. This solution should be explored, as well as its relationship with annotations in Viper. This solution should then be used to solve this mismatch.

\subsection*{10.2.2 Completeness of inlining}

Completeness in our general framework. Chapter 7 only sketches a way of how to construct annotations for proving completeness. Completeness in this framework under the strongest postcondition assumption should be explored deeper. In particular:
1. The notion of bounded program should be formalized, maybe as a semantic notion instead of a syntactic one.
2. The completeness property should be formally defined, as well as what it means to annotate a program.
3. The gap between completeness with indices for all methods and the examples where completeness does not hold when recursive methods are not indexed should be bridged. That is, we should either find an example of incompleteness because of a non-recursive method which is not indexed, or we should prove completeness even in the case of non-recursive methods which are not indexed.
4. A completeness theorem, similar to the soundness theorem, should be formalized and proved in Isabelle/HOL, using the same framework.

Completeness in Viper. Section 8.6 sketches a proof for the strongest postcondition assumption in Viper.
1. The idea of existential permissions in Viper should be formalized, given a semantics, and the supportedness property for a semantic assertion in our general framework should be adapted consequently. Otherwise, another approach should be developed to bridge the permission gap, for example, approaches using abstract read permissions [9, 22] or ghost variables.
2. The strongest postcondition assumption should be formalized and proved, at least for a simplified version of Viper.

\subsection*{10.2.3 Towards useful features in Viper}

Instantiation of the general framework with Viper. The injection of Viper described in Chapter 8 is not complete.
1. The semantics of Viper should be formally defined.
2. The instantiation should be done with a more complex version of Viper, taking into account for example trace semantics, magic wands, and predicates.
3. The requirements for the instantiation with Viper should be proved.

Early error detection in Viper. The results proved or sketched in this thesis should be leveraged in Viper.
1. A function to check whether a Viper statement is in mono or framing should be developed in a conservative way, using the stability of these properties with sequential composition and conditional branching. perm, forperm and assume statements should be explored further to make this feature more precise.
2. Based on the notion of a barrier and the different scopings described in Chapter 8, several functions for inlining in Viper should be developed.
3. These functions should be used for developing a feature to provide more useful information on errors reported in early stages of annotating a Viper program.
4. Extensions of this feature should be considered, such as stratified inlining [ 13,12\(]\), adapting the inlining to the annotation (such as not inlining further when an annotation is already written, adapt the scoping based on the shape of the annotation...).

Caching verification results. As described in Section 8.7.2, this work opens the way for speeding up reverification \({ }^{2}\) in Viper, using mono and framing to know when it is not necessary to verify some parts of the code again. This problem should be explored in depth.

Inlining in other contexts. This work focuses on static inlining at the Viper level. However, Viper serves as an intermediate verification language, on which many front-ends are built. It is unclear whether inlining at the frontend level is related to inlining at the Viper level. Inlining at the front-end level should be explored, as well as its relationship to inlining at the Viper level.

\footnotetext{
\({ }^{2}\) That is, verification after small modifications of a first verified version.
}

\section*{List of Figures}
1.1 An illustration Viper architecture (figure taken from [15]). ..... 6
1.2 Illustration of the problem. ..... 7
3.1 Example of unsoundness of inlining in Viper: Statement not safeMono. ..... 22
3.2 Example of unsoundness of inlining in Viper: Statement not monoOut. ..... 24
3.3 Example of unsoundness of inlining in Viper: Statement not framing. ..... 25
3.4 Representation of sets of statements satisfying safeMono, monoOut and framing. ..... 27
4.1 Inhaling and exhaling the same assertion. ..... 33
5.1 Illustrations of the input parameters for the model. \(P\) is a semantic assertion, \(\varphi\) is an element of \(\Sigma, t\) is a renaming quadruple, and \(o\) is a custom statement of type \(O\). ..... 40
6.1 Loops with conditions not well-formed monotonic ..... 67
7.1 An example of the syntactic transformation bound \({ }^{42}(s, P r)\). ..... 85
7.2 Existential permissions require the soundness condition. ..... 90
7.3 Illustration of how to construct annotations using the strongest postcondition assumption. ..... 93
8.1 Constructing a strongest postcondition in Viper. ..... 107
8.2 Viper predicates can hide information. ..... 109
9.1 Illustration of the concept of barriers. ..... 117
9.2 Degrees of freedom for scoping of barriers ..... 118
10.1 Illustration of the soundness and completeness results. ..... 121
A. 1 Theories presented in this appendix . . . . . . . . . . . . . . . . . 135

\section*{Listings}
1.1 An example where an early error detection feature would be useful (original) ..... 4
1.2 An example where an early error detection feature would be useful (inlined). ..... 5
2.1 General shape of a while loop in Viper ..... 15
2.2 General shape of a method in Viper ..... 16
2.3 General shape of a method call in Viper ..... 17
3.1 Annotated original program. ..... 22
3.2 Inlined program (bound of 1) ..... 22
3.3 Annotated original program. ..... 24
3.4 Inlined program (bound of 1) ..... 24
3.5 Annotated original program. ..... 25
3.6 Inlined program (bound of 1) ..... 25
4.1 Pure assertion ..... 33
4.2 Accessibility predicate ..... 33
4.3 Combination of a pure assertion and an accessibility predicate. ..... 33
6.1 Loop condition not mono. ..... 67
6.2 Negated loop condition not mono. ..... 67
7.1 Original program. ..... 85
7.2 Bounded program ..... 85
7.3 Bounded program (bound of 2). ..... 87
7.4 Inlined program (bound of 2) ..... 87
7.5 Original annotated program. ..... 89
7.6 Inlined program (bound of 1) ..... 89
7.7 Original program. ..... 90
7.8 Inlined program (bound of 1) ..... 90
7.9 Example for a method. ..... 93
7.10 Inlined. ..... 93
7.11 Example for a loop ..... 93
7.12 Inlined. ..... 93
8.1 Example of a program for which we construct a strongest postcondition. ..... 107
8.2 Original annotated program. ..... 109
8.3 Inlined (bound of 1) ..... 109
9.1 Differences of loop treatment between the Viper verifiers Car- bon and Silicon. ..... 112
9.2 Example with too much permission inside and outside a loop ..... 114
9.3 Example of a program which would verify in the third loop semantics. ..... 115
9.4 Example of a loop whose behavior depends on the loop invariant. 1 ..... 115
9.5 A method call. ..... 117
9.6 An inlined method call. ..... 117

\section*{Bibliography}
[1] Viper tutorial. http://viper.ethz.ch/tutorial. Accessed: 31 March 2020.
[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for objectoriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, Formal Methods for Components and Objects, pages 364-387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[3] Claudio Belo Lourenco, Maria Joao Frade, and Jorge Sousa Pinto. A Generalized Program Verification Workflow Based on Loop Elimination and SA Form. Proceedings - 2019 IEEE/ACM 7th International Workshop on Formal Methods in Software Engineering, FormaliSE 2019, pages 75-84, 2019.
[4] John Boyland. Checking interference with fractional permissions. In International Static Analysis Symposium, pages 55-72. Springer, 2003.
[5] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. Proceedings - Symposium on Logic in Computer Science, pages 366-375, 2007.
[6] Cyrill Martin Gössi. A Formal Semantics for Viper. Master thesis, ETH Zürich, 2016.
[7] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5904 LNCS:161-177, 2009.
[8] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In Giuseppe Castagna, editor, European Conference on Object-Oriented Programming (ECOOP), volume 7920 of Lecture Notes in Computer Science, pages 451-476. Springer, 2013.
[9] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Abstract read permissions: Fractional permissions without the fractions. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7737 LNCS:315-334, 2013.
[10] Charles Antony Richard Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-580, 1969.
[11] R. A.L.F. Jung, Robbert Krebbers, Jacques Henri Jourdan, Aleš Bizjak, Lars Birkedal, and D. E.R.E.K. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, (October), 2018.
[12] Akash Lal and Shaz Qadeer. Reachability modulo theories. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8169 LNCS:23-44, 2013.
[13] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7358 LNCS:427-443, 2012.
[14] K Rustan M Leino and Peter Müller. A basis for verifying multithreaded programs. In European Symposium on Programming, pages 378-393. Springer, 2009.
[15] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. Dependable Software Systems Engineering, pages 104-125, 2017.
[16] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. Isabelle/HOL: a proof assistant for higher-order logic, volume 2283. Springer Science \& Business Media, 2002.
[17] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2142:1-19, 2001.
[18] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. ACM Transactions on Programming Languages and Systems, 31(3), 2009.
[19] Peter W O'hearn. Resources, concurrency, and local reasoning. Theoretical computer science, 375(1-3):271-307, 2007.
[20] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. Logical Methods in Computer Science, 8(3):1-54, jul 2012.
[21] John C. Reynolds. Separation logic: A logic for shared mutable data structures. Proceedings - Symposium on Logic in Computer Science, 1(1):5574, 2002.
[22] Benjamin Schmid. Abstract Read Permission Support for an Automatic Python Verifier. Bachelor thesis, ETH Zürich, 2018.
[23] M. Schwerhoff and A. J. Summers. Lightweight Support for Magic Wands in an Automatic Verifier. In J. T. Boyland, editor, European Conference on Object-Oriented Programming (ECOOP), volume 37 of LIPIcs, pages 614-638. Schloss Dagstuhl, 2015.
[24] Jan Smans, Bart Jacobs, Frank Piessens, and K. U. Leuven. Implicit dynamic frames. ACM Transactions on Programming Languages and Systems, 34(1):1-58, 2012.
[25] Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2303:402-416, 2002.

Appendix A

\section*{Appendix: Isabelle Formalization}

\section*{A. 1 Theories presented in this appendix}


Figure A.1: Theories presented in this appendix

Figure A. 1 shows the theories presented in this appendix. The SepAlgebra theory defines two locales, the preliminary separation algebra and the actual separation algebra, as defined in Chapter 4. It also defines and proves properties on semantic assertions. The Semantics theory defines the language and its semantics defined in Chapter 5, using a locale to express the parameters and the requirements which should be satisfied. The Soundness theory formalizes Chapter 6: It defines the static inlining function, the soundness condition, and formally proves the soundness theorem. Unrelated to the other theories, the Renaming theory formalizes the rename interface described in Section 8.4. For space reasons, we do not show the proofs, only the definitions and the lemmas.

\section*{A. 2 Separation algebra}
theory SepAlgebra
imports Main
begin
type_synonym 'a assertion = "'a \(\Rightarrow\) bool"

\section*{A.2.1 Preliminary separation algebra}
```

locale pre_sep_algebra =
fixes orig_plus :: "'a \# 'a \# 'a option"
fixes u :: "’a"
assumes orig_comm: "orig_plus a b = orig_plus b a"
and asso1: "Some ab = orig_plus a b ^ Some bc =orig_plus b c \Longrightarrow
orig_plus ab c = orig_plus a bc"
and asso2: "Some ab = orig_plus a b ^ None = orig_plus b c \Longrightarrow
None = orig_plus ab c"
and asso3: "None = orig_plus a b ^ Some bc = orig_plus b c \Longrightarrow
None = orig_plus a bc"
and orig_neutral: "orig_plus a u = Some a"
begin
fun plus :: "’a option }=>\mathrm{ 'a option }=>\mathrm{ 'a option" (infixl "Ф" 60) where
"Some a \oplus Some b = orig_plus a b"
| "_ \oplus _ = None"
definition defined :: "'a \# 'a = bool" (infixl "\#\#" 60)
where "defined a b \longleftrightarrow (\neg Option.is_none (Some a \oplus Some b))"
definition smaller :: "'a \# 'a \# bool" (infixl "<<" 50) where
"smaller a b \longleftrightarrow(\existsc. Some b = Some a \oplus Some c)"
lemma commutative:
"a \oplus b = b \oplus a"
\langleproof\rangle
lemma associative:
"(a\oplusb)\oplusc=a\oplus(b\oplusc)"
<proof\rangle
lemma neutral:
"a \oplus Some u = a" \langleproof\rangle
lemma smaller_refl:
"a << a"
\langleproof\rangle

```
lemma smaller_trans:
    assumes "a << b"
```

            and "b << c"
    shows "a << c"
    \langleproof\rangle
definition pure :: "'a m bool" where
"pure \varphi}\longleftrightarrow\mathrm{ Some }\varphi\oplus\mathrm{ Some }\varphi=\mathrm{ Some }\varphi\mathrm{ "
definition pure_set :: "'a m 'a set" where
"pure_set \varphi = {\varphi'. pure \varphi'^\varphi'<<\varphi}"
definition core :: "'a m 'a" ("/_|")
where "|\varphi| = the (Finite_Set.fold (\oplus) (Some u) (Some ' (pure_set \varphi)))"
lemma pure_smaller_ok:
assumes "p << \varphi"
and "pure p"
shows "Some \varphi = Some }\varphi\oplus\mathrm{ Some p"
\langleproof\rangle
lemma pure_add:
assumes "pure a"
assumes "pure b"
assumes "a \#\# b"
shows "pure (the (Some a }\oplus\mathrm{ Some b))"
\langleproof\rangle
lemma pure_set_stable_add:
assumes "a \in pure_set \varphi"
and "b \in pure_set \varphi"
and "Some c = Some a }\oplus\mathrm{ Some b"
shows "c \in pure_set \varphi"
<proof\rangle
lemma pure_u [simp]: "pure u" \langleproof\rangle
lemma u_smaller [simp]: "u << \varphi" \langleproof\rangle
lemma empty_in_pure:
"u \in pure_set \varphi"
<proof\rangle
definition s_core :: "'a \# 'a option" where "s_core x = Some |x|"
end

```

\section*{A.2.2 Separation algebra}
```

locale sep_algebra = pre_sep_algebra +
fixes C :: "'a = 'a"

```
```

    assumes finiteness: "finite {\varphi'. pure \varphi'^ \varphi'<< \varphi}"
    and partially_cancellative: "Some (C \varphi) \oplus Some a = Some (C \varphi) \oplus
    Some b C a = b"
and decompo: "Some \varphi = Some |\varphi| \oplus Some (C \varphi)"
and c_empty_core: "|C \varphi = u"
and unique_c: 'Some \varphi = Some }|\varphi| | Some \varphi'^ |\varphi'| = u\Longrightarrow ' ' =
C }\mp@subsup{\varphi}{}{\prime\prime
and positivity: "Some a }\oplus\mathrm{ Some b = Some u \# a = u"
and pure_reducibility: "pure p ^p<< a ^ Some a = Some b \oplus Some
c\Longrightarrow
(\existsa'. Some a' = s_core b \oplus s_core c ^ p << a')''
begin
lemma pure_u:
"|u| = u"
\langleproof\rangle
definition some_core :: "'a \# 'a option" where
"some_core \varphi = (Finite_Set.fold (\oplus) (Some u) (Some ' (pure_set \varphi)))"
lemma commut_comp: "\y x. (\oplus) y ○ (\oplus) x = (\oplus) x ○ (\oplus) y"
\langleproof\rangle
interpretation add_pure: comp_fun_commute "plus"
<proof\rangle
lemma "fold_graph (\oplus) (Some u) (Some ' pure_set \varphi) (some_core \varphi)"
\langleproof\rangle
lemma " }|\varphi|=\mathrm{ the (some_core }\varphi\mathrm{ )" <proof>
lemma pure_set_u: "pure_set u = {u}"
\langleproof\rangle
lemma some_core_u:
"some_core u = Some u"
<proof>
definition some_prop :: "'a }=>\mathrm{ 'a option set }=>\mathrm{ 'a option }=>\mathrm{ bool" where
"some_prop \varphi S s \longleftrightarrow (S\subseteq Some ' pure_set \varphi\longrightarrow (S = {} ^ s = Some
u) \vee (s G Some ' pure_set }\varphi\wedge(\foralls'\inS. the s'<< the s)))"
lemma some_prop_proof:
"some_prop \varphi (Some ' pure_set \varphi) (some_core \varphi)"
<proof>
lemma core_is_max_uni:
"|\varphi| \in pure_set \varphi ^ (\forallx \in pure_set \varphi. x << |\varphi|)"
<proof\rangle

```
```

lemma core_is_max:
$" \varphi^{\prime}=|\varphi| \longleftrightarrow\left(\varphi^{\prime} \in \operatorname{pure} \text { _set } \varphi \wedge\left(\forall \varphi^{\prime}, \in \operatorname{pure} \text { _set } \varphi \cdot \varphi^{\prime}>\ll \varphi^{\prime}\right)\right)^{\prime}$
(is "?a $\longleftrightarrow$ ?b")
〈proof〉
lemma sum_pure:
assumes "pure a"
and "pure b"
and "Some $c=$ Some $a \oplus$ Some $b$ "
shows "pure c"
$\langle p r o o f\rangle$
lemma add_trans:
assumes "a << aa"
and " $b \ll b b$ "
and "Some $c c=$ Some aa $\oplus$ Some $b b$ "
and "Some $c=$ Some $a \oplus$ Some $b$ "
shows " $c \ll c c$ "
〈proof〉
lemma core_add:
assumes "Some $a=$ Some $b \oplus$ Some $c$ "
shows "s_core $a=s$ core $b \oplus$ s_core $c$ "
〈proof〉
lemma pure_set_finite:
"finite (pure_set $\varphi$ )"
〈proof〉
definition s_C :: "'a $\Rightarrow$ 'a option" where "s_C x = Some (C x)"
lemma core_properties:
shows "pure $|\varphi|$ "
and $"|\varphi| \ll \varphi$ "
and "pure $\varphi \longleftrightarrow \varphi=|\varphi|$ "
〈proof〉
lemma not_pure_core:
"ᄀ pure $\varphi \longleftrightarrow \varphi \neq|\varphi|$ " (is "?a $\longleftrightarrow$ ?b")
〈proof〉
definition add_set :: "'a set $\Rightarrow$ 'a set $\Rightarrow$ 'a set" (infixl " $\oplus \oplus$ " 60) where
" $A \oplus \oplus B=\{$ the (Some $a \oplus$ Some b) $\mid a b . a \in A \wedge b \in B \wedge a \# \# b\}$ "
lemma elem_add_set:
"x $\in A \oplus \oplus B \longleftrightarrow(\exists a b . a \in A \wedge b \in B \wedge$ Some $x=$ Some $a \oplus$ Some $b$ )"
〈proof〉

```
```

lemma simple_set_add_comm:
"A\oplus\oplusB=B\oplus\oplusA"
<proof>
lemma core_inv:
"||\varphi|| = |\varphi|"
<proof\rangle
lemma decompo_new_plus:
"Some \varphi = s_core \varphi }\varphi\mathrm{ s_C }\varphi
<proof>
lemma properties_c:
shows "a=b\longleftrightarrow |a| = |b| ^C a = C b"
and "C (C \varphi) = C \varphi"
<proof>
lemma definedness [simp]:
assumes "a \#\# b"
shows "C a \#\# b"
and "core a \#\# b"
and "b \#\# a"
\langleproof\rangle
lemma reduce_add:
assumes "a \#\# b"
and "Some a \oplus Some b = Some a \oplus Some c"
shows "s_core a }\oplus\mathrm{ Some b = s_core a }\oplus\mathrm{ Some c"
<proof>
definition intuitionistic :: "'a assertion }=>\mathrm{ bool" where
"intuitionistic P \longleftrightarrow (\forall\varphi \varphi'. \varphi << \varphi' \longrightarrow (P\varphi \longrightarrowP \varphi'))"
definition supported :: "'a assertion }=>\mathrm{ bool" where
"supported P}\longleftrightarrow(\forall\varphi.P\varphi\longrightarrow(\existsm.m<<\varphi\wedgePm^(\forall\varphi'.\varphi'<<
^P \varphi' }\longrightarrow(m<<\varphi\mp@subsup{\varphi}{}{\prime}))))
definition Inh :: "'a assertion }=>\mathrm{ 'a set" where

```

```

definition bigger_set :: "'a set => 'a set => bool" (infixl ">>" 50) where
"A >> B\longleftrightarrow (\foralla\inA.\existsb\inB. b<< a)"
lemma rel_bigger_add_set:
assumes " }A=B\oplus\oplusD\mathrm{ "
shows "A >> B"
<proof\rangle
lemma equiv_Inh [simp]:

```
```

    assumes " P \varphi"
    ```

```

        shows " }\varphi\in\operatorname{Inh P"
    <proof>
    lemma defined_trans_plus:
assumes "Some a = Some b \oplus Some c \oplus Some d"
shows "c \#\# d"
<proof\rangle
lemma neutral_parts:
shows "|u| = u"
and "Cu = u"
\langleproof\rangle
lemma c_add:
assumes "Some a = Some b \oplus Some c"
shows "s_C a = s_C b \oplus s_C c"
<proof>
lemma pure_c:
"pure }\varphi\longleftrightarrowC\varphi=u
<proof>
lemma smaller_pure:
assumes "a << b"
and "pure b"
shows "pure a"
<proof>
lemma antisym:
assumes "a << b"
and "b << a"
shows "a = b"
<proof>
lemma supported_inh:
assumes " P \varphi"
and "supported P"
shows "{\varphi} >> Inh P"
<proof\rangle
lemma supported_intui:
assumes "supported P"
and "intuitionistic P"
shows "P\varphi\longleftrightarrow " {\varphi} >> Inh P"
<proof>
lemma plus_and_bigger_set:

```
```

    assumes "A >> B"
    shows "(A\oplus\oplusD) >> (B\oplus\oplusD)"
    <proof>
lemma simple_set_add:
assumes "Some a = Some b \oplus Some c"
shows "{b} \oplus\oplus {c} = {a}" (is "?s = ?a")
<proof>
lemma set_add_elem:
assumes "x\inA\oplus\oplusB"
shows "\existsa b. a \inA\wedgeb\inB\wedge Some x = Some a }\oplus\mathrm{ Some b"
<proof>
lemma set_add_elem_reci:
assumes "a \inA\wedge b \inB ^ Some x = Some a }\oplus\mathrm{ Some b"
shows "x}\inA\oplus\oplusB\mathrm{ "
\langleproof>
lemma set_add_asso:
"(A\oplus\oplusB)\oplus\oplusD = A\oplus\oplus(B\oplus\oplusD)" (is "?g = ?d")
<proof\rangle
lemma bigger_set_union:
assumes "A' >> A"
and " }B\mathrm{ ' >> B"
shows "A'\cup B' >> A\cupB"
<proof\rangle
lemma smaller_core_comp:
"b<< a\longleftrightarrow |b|<< |a| ^C b << C a" (is "?a \longleftrightarrow ?b")
<proof>
lemma c_trans_ineq:
assumes "b << a"
and "Some a = s_C x \oplus Some aa"
and "Some b = s_C x \oplus Some bb"
shows "bb << aa"
<proof>
lemma frame_trans:
assumes "b << a"
and "Some a = Some i }\oplus\mathrm{ Some ra"
and "Some b = Some i }\oplus\mathrm{ Some rb"
and "Some c = s_core i }\oplus\mathrm{ Some ra"
shows "rb << c"
<proof\rangle
lemma bigger_set_forall:

```
```

    "A >> B\longleftrightarrow(\foralla\inA.{a} >> B)"
    \langleproof\rangle
    lemma subset_smaller:
assumes "A\subseteqB"
shows "A >> B"
<proof>
lemma bigger_set_trans:
assumes "A >> B"
and "B >> D"
shows " }A>>D
<proof>
lemma pure_set_add_smaller:
assumes " }A\oplus\oplusB=A\mathrm{ "
shows "A >> B"
<proof>
lemma decompose_set:
assumes "x
shows "\existsa b. a \inA\wedgeb\inB\wedge Some x = Some a }\oplus\mathrm{ Some b }\oplus\mathrm{ Some c }
Some d"
<proof>
lemma add_pure_singleton:
assumes "A >> B"
and "A >> {p}"
and "pure p"
shows "A >> B \oplus\oplus{p}"
<proof>
lemma add_sets_neutral:
"A \oplus\oplus {u} = A" (is "?a = ?b")
<proof>
end
end

```

\section*{A. 3 Semantics}
```

theory Semantics
imports SepAlgebra
begin

```

\section*{A.3.1 Abstract language}
```

datatype ('a, 'b, 'c) stmt =
Inhale "'a assertion"
| Assume "'a assertion"
| Exhale "'a assertion"
| Skip
| Seq "('a, 'b, 'c) stmt" "('a, 'b, 'c) stmt" (infixl ";" 60)
| If "('a, 'b, 'c) stmt" "('a, 'b, 'c) stmt"
| Var "'b list"
| Havoc "'b list"
| MethodCall "'b list" "string" "’b list"
| While "'a assertion" "'a assertion" "('a, 'b, ’c) stmt"
| Other 'c

```
type_synonym ('a, 'b, 'c) method \(=\) "string \(\times\) 'b list \(\times\) 'b list \(\times\) 'a
assertion \(\times\) 'a assertion \(\times(' a, \quad b, \quad\) 'c) stmt"
type_synonym ('a, 'b, 'c) program = "('a, 'b, 'c) method list"
fun get_method :: "('a, 'b, 'c) program \(\Rightarrow\) string \(\Rightarrow\) ('a, 'b, 'c) method
option" where
    "get_method ( \(t\) \# q) \(s=(i f\) fst \(t=s\) then Some \(t\) else get_method \(q\)
s)"
| "get_method _ s = None"
fun get_args :: "('a, 'b, 'c) method \(\Rightarrow\) 'b list" where
    "get_args (_, args, _, _, _, _) = args"
fun get_ret :: "('a, 'b, 'c) method \(\Rightarrow\) 'b list" where
    "get_ret (_, _, ret, _, _, _) = ret"
fun get_args_ret :: "('a, 'b, 'c) method \(\Rightarrow\) 'b list" where
    "get_args_ret \(m=\) get_args \(m\) @ get_ret m"
fun get_pre :: "('a, 'b, 'c) method \(\Rightarrow\) 'a assertion" where
    "get_pre (_, _, _, P, _, _) = P"
fun get_post :: "('a, 'b, 'c) method \(\Rightarrow\) 'a assertion" where
    "get_post (_, _, _, _, Q, _) = Q"
fun get_body :: "('a, 'b, 'c) method \(\Rightarrow(' a, \quad\) 'b, 'c) stmt" where
    "get_body (_, _, _, _, _, s) = s"
```

datatype 'a ss = S "'a set" | Error
type_synonym 'b rename_t $=$ "'b list $\times$ ' $b$ list $\times$ 'b list $\times$ 'b list"
fun get_S :: "'a ss $\Rightarrow$ 'a set" where
"get_S (S A) = A"
| "get_S _ = \{\}"
fun union_set_ss :: "'a ss set $\Rightarrow$ 'a ss" where
"union_set_ss $A=$ (if Error $\in A$ then Error
else $S(\cup a \in A$. get_S a))"
fun len :: "('a, 'b, 'c) stmt $\Rightarrow$ nat" where
"len (MethodCall y mx) = 5"
| "len (While b I s) = 5 + len s"
| "len (If s1 s2) = len s1 + len s2"
| "len (s1 ; s2) = len s1 + len s2"
| "len _ = 1"
lemma len_at_least_one:
"len $s \geq 1 "$
〈proof〉
definition lnot :: "'a assertion $\Rightarrow$ 'a assertion" where
"lnot $P=(\lambda s . \neg P s) "$
fun wf_renaming :: "'b rename_t $\Rightarrow$ bool" where
"wf_renaming (old_vars, new_vars, vars_to_avoid, domain) $\longleftrightarrow$
length old_vars = length new_vars $\wedge$ distinct old_vars $\wedge$ distinct
new_vars"
definition lfalse :: "’a assertion" where
"lfalse = ( $\lambda s$. False)"
locale semantics_algebra = sep_algebra +
fixes $\sigma::$ "'a $\Rightarrow$ 'b set"
fixes semantics_other :: "('a, 'b, 'c) program $\Rightarrow$ ' $a \Rightarrow$ ' $c \Rightarrow$ 'a ss"
fixes read_pred :: "'a assertion $\Rightarrow$ 'b list"
fixes read_other :: "’c $\Rightarrow$ 'b list"
fixes well_defined_assert :: "'a assertion $\Rightarrow$ 'a $\Rightarrow$ bool"
fixes modif_other :: "'c $\Rightarrow$ 'b list"
fixes rename_elem :: "'b $\Rightarrow$ 'b rename_ $t \Rightarrow$ ' $b$ "
fixes rename_other $::$ "'c $\Rightarrow$ ' $b$ rename_t $\Rightarrow$ ' $c$ "

```
fixes rename_state :: "'a \(\Rightarrow\) 'b rename_t \(\Rightarrow\) 'a"
fixes rename_pred :: "'a assertion \(\Rightarrow\) ' \(b\) rename_ \(t \Rightarrow\) 'a assertion"
fixes rename_inv :: "'b rename_ \(t \Rightarrow\) 'b rename_ \(t\) "
fixes wf_other :: "('a, 'b, ’c) program \(\Rightarrow\) 'c \(\Rightarrow\) bool"

\section*{assumes}
\[
\text { store_pure: } " \sigma \varphi=\sigma|\varphi| "
\]
and store_add: "Some \(a=\) Some \(c \oplus\) Some \(d \Longrightarrow \sigma a=\sigma c \cup \sigma d "\)
and store_empty: " \(\sigma u=\{ \}\) "
and compatible_stores: "a \#\# b \(\Longrightarrow\) pure \(a \Longrightarrow \sigma a \subseteq \sigma b \Longrightarrow a \ll b\) "
and unique_store_exists: \(" v a r \in \sigma a \Longrightarrow(\exists c \cdot \sigma c=\{\operatorname{var}\} \wedge c \ll a) "\)
and defined_disjoint_store: \(" \sigma a \cap \sigma b=\{ \} \Longrightarrow\) pure \(a \Longrightarrow a\) \#\# b"
and read_pred_def: "set (read_pred P) = ( \(\cup i \in \operatorname{Inh} P . \sigma\) i)"
and well_defined_assert_supported: "supported \(P \Longrightarrow\) (well_defined_assert
\(P \varphi \longleftrightarrow\) set (read_pred P) \(\subseteq \sigma \varphi\) )"
and can_read_not: "well_defined_assert \(P a \longleftrightarrow\) well_defined_assert (lnot P) a"
and p_implies_well_def: " \(P \varphi \Longrightarrow\) well_defined_assert \(P \varphi\) "
and modif_other_sem: "wf_other Pr other \(\wedge c \in\) get_S (semantics_other \(\operatorname{Pr}\) a other) \(\Longrightarrow \sigma a \subseteq \sigma c \wedge \sigma c \subseteq \sigma a \cup\) set (modif_other other)"
and modif_other_read_other: "set (modif_other other) \(\subseteq\) set (read_other other)"
and rename_inv_def_elem: "wf_renaming \(t \Longrightarrow\) rename_elem (rename_elem elem \(t\) ) (rename_inv \(t\) ) = elem"
and rename_inv_def_other: "wf_renaming \(t \Longrightarrow\) rename_other (rename_other other \(t\) ) (rename_inv t) = other"
and renaming_invert_wf: "wf_renaming \(t \Longrightarrow\) wf_renaming (rename_inv t)"
and rename_removes_vars_elem: "wf_renaming (11, 12, 13, do) \(\Longrightarrow\) var \(\in\) set do \(\Longrightarrow\)
rename_elem var (11, 12, 13, do) \(\in\) set \(11 \cup\) set \(13 \Longrightarrow\) rename_elem \(\operatorname{var}(11,12,13\), do) \(\in \operatorname{set} 12 "\)
and rename_elem_in_11_12: "wf_renaming (11, 12, 13, do) \(\Longrightarrow\) i \(<\) length \(11 \Longrightarrow\) rename_elem (11 ! i) \(\left(11,12,13\right.\), do) = 12 ! \(i^{"}\)
and rename_store: "wf_renaming \(t \Longrightarrow \sigma\) (rename_state a \(t\) ) \(=(\lambda e l e m\). rename_elem elem t) ' ( \(\sigma\) a)"
and rename_state_add: "wf_renaming \(t \Longrightarrow\) Some \(a=\) Some \(d \oplus\) Some c \(\Longrightarrow\)
Some (rename_state a \(t\) ) = Some (rename_state dt) \(\oplus\) Some (rename_state
c t)"
and rename_state_identity: "wf_renaming \(t \Longrightarrow(\forall x \in \sigma\) a. rename_elem
\(x t=x) \Longrightarrow\) rename_state \(a t=a "\)
and rename_state_double: "wf_renaming \(t \Longrightarrow w f \_r e n a m i n g ~ t 1 ~ \Longrightarrow w f \_r e n a m i n g ~\)
```

$t 2 \Longrightarrow$
( $\forall x \in \sigma$ a. rename_elem $x t=r e n a m e \_e l e m ~\left(r e n a m e \_e l e m ~ x t 1\right) ~$

```

and rename＿pred＿def：＂rename＿pred \(P t=(\lambda \varphi . P\)（rename＿state \(\varphi\)（rename＿inv t）））＂
and well＿defined＿assert＿rename：＂wf＿renaming \(t \Longrightarrow\) well＿defined＿assert \(P \varphi \Longrightarrow\) well＿defined＿assert（rename＿pred \(P\) t）（rename＿state \(\varphi\) t）＂
and read＿other＿rename＿other：＂read＿other（rename＿other other \(t\) ）＝map
（ \(\lambda\) elem．rename elem \(t\) ）（read＿other other）＂
and modif＿rename＿other：＂modif＿other（rename＿other other \(t\) ）＝map（ \(\lambda e l e m\) ． rename＿elem elem \(t\) ）（modif＿other other）＂
and wf＿other＿renaming：＂wf＿renaming \(t \Longrightarrow\) wf＿other Pr other \(\Longrightarrow\) wf＿other \(\operatorname{Pr}\)（rename＿other other \(t\) ）＂
and ver＿rename＿other：＂wf＿renaming \(t \Longrightarrow\) semantics＿other Pr \(\varphi\) other \(\neq\) Error \(\Longrightarrow\) semantics＿other Pr（rename＿state \(\varphi t\) ）（rename＿other other t）\(\neq\) Error＂
and semantics＿rename＿other：＂wf＿renaming \(t \Longrightarrow a \in\) get＿S（semantics＿other \(\operatorname{Pr}\)（rename＿state \(\varphi t\) ）（rename＿other other t））
\(\Longrightarrow\) rename＿state a（rename＿inv \(t\) ）\(\in\) get＿S
（semantics＿other \(\operatorname{Pr} \varphi\) other）＂
begin
lemma pure＿is＿exactly＿store：＂\(a \ll c \Longrightarrow b \ll c \Longrightarrow\) pure \(a \Longrightarrow \sigma a \subseteq\) \(\sigma b \Longrightarrow a \ll b^{\prime \prime}\)

〈proof〉
lemma rename＿state＿c＿same：
assumes＂wf＿renaming \(t\)＂
shows＂C（rename＿state a \(t\) ）＝C a＂
〈proof〉
lemma finite＿store：
＂finite（ \(\sigma x\) ）＂
〈proof〉
lemma rename＿state＿same：
assumes＂wf＿renaming t1＂
and＂wf＿renaming t2＂
and＂\(\backslash x . x \in \sigma a \Longrightarrow\) rename＿elem \(x\) t1＝rename＿elem \(x\) t2＂
shows＂rename＿state a t1＝rename＿state a t2＂
\(\langle p r o o f\rangle\)
lemma rename＿inv＿def＿state：
assumes＂wf＿renaming \(t\)＂
shows＂rename＿state（rename＿state a t）（rename＿inv t）＝a＂（is＂？a ＝a＂）
```

<proof>
lemma rename_store_same_if_11:
assumes "wf_renaming (l1, 12, 13, do)"
and "\sigma a\subseteq set 11"
shows "rename_state a (11, 12, 13, do) = rename_state a (l1, l2,
[], d)"
<proof>
definition read_mono :: "'a assertion \# bool" where
"read_mono P\longleftrightarrow(\forall\varphi \varphi'. \varphi << \varphi' \longrightarrow well_defined_assert P \varphi \longrightarrow
well_defined_assert P \varphi')"
definition wf_assert :: "'a assertion \# bool" where
"wf_assert P \longleftrightarrow supported P ^ intuitionistic P ^ read_mono P"
lemma can_read_false:
"well_defined_assert lfalse a"
<proof>
fun rename_set :: "'a set }=>\mathrm{ 'b rename_t }=>\mathrm{ ' 'a set" where
"rename_set A t = ( }\lambda\varphi\mathrm{ . rename_state }\varphi\mathrm{ t) ' A"
lemma rename_pred_same:
assumes "wf_renaming t"
shows "P \varphi\longleftrightarrow (rename_pred P t) (rename_state \varphi t)"
<proof>
lemma wf_renaming_order:
assumes "wf_renaming t"
and "a << b"
shows "rename_state a t << rename_state b t"
\langleproof>
lemma rename_doesnt_change_if_not_affected:
assumes "wf_renaming (11, l2, [], do)"
and "set (read_pred P)\subseteq set l1"
and "wf_assert P"
shows "rename_pred P (11, 12, [], do) = rename_pred P (11, l2, l,
d)"
<proof>
lemma wf_renaming_diff:
assumes "wf_renaming t"
and "a = b"
shows "rename_state a t f= rename_state b t"
\langleproof>
lemma inh_renamed:

```
```

    assumes "wf_renaming t"
    shows "Inh (rename_pred P t) = rename_set (Inh P) t" (is "?a = ?b")
    <proof>
lemma read_pred_rename_pred:
assumes "wf_renaming t"
shows "set (read_pred (rename_pred P t)) = (\lambdaelem. rename_elem elem
t) ' (set (read_pred P))" (is "?a = ?b")
<proof\rangle
lemma wf_rename_inv_other:
assumes "wf_renaming t"
shows "rename_state (rename_state x (rename_inv t)) t = x"
<proof>
definition rename_list :: "'b list }=>\mathrm{ 'b rename_t }=>\mathrm{ 'b list" where
"rename_list l t = map (\lambdaelem. rename_elem elem t) l"
lemma rename_pred_comp_simple:
assumes "wf_renaming t"
and "wf_renaming (l1, l2, [], do)"
and "set (read_pred P)\subseteq set l1"
and "wf_assert P"
shows "rename_pred P (11, rename_list 12 t, [], do) = rename_pred
(rename_pred P (l1, l2, [], do)) t" (is "?ra = ?rb")
<proof>
lemma rename_state_neutral:
assumes "wf_renaming t"
shows "rename_state u t = u" (is "?r = u")
\langleproof\rangle
lemma rename_removes_vars_other:
assumes "wf_renaming (l1, l2, l3, do)"
and "set (read_other other) \subseteq set do"
shows "set (read_other (rename_other other (11, 12, 13, do))) \cap (set
l1 U set l3) \subseteq set l2" (is "?a \subseteq ?b")
<proof\rangle
lemma pure_is_exactly_store_variant:
assumes "a << b"
and "pure b"
and "\sigmaa=\sigma b"
shows "a = b"
<proof>
definition h :: "'b list }=>\mathrm{ 'a set" where
"h V = {\varphi. pure }\varphi\wedge\sigma\varphi=\operatorname{set V}"

```
```

lemma h_store:
assumes "\varphi\inh l"
shows "\sigma\varphi = set l"
<proof>
lemma h_pure:
assumes "hx \inhx"
shows "pure hx"
<proof\rangle
lemma all_stores_exist:
"s\subseteq\sigmaa\Longrightarrow\existsc.c<< a ^\sigmac=s"
<proof>
lemma h_bigger:
assumes "set x\subseteq\sigma \varphi"
shows "{\varphi} >> h x"
<proof>
lemma h_lin_simpler:
assumes "set v = set v1 U set v2"
and "set v1 \cap set v2 = {}"
shows "h v = h v1 \oplus\oplus h v2" (is "?a = ?b")
<proof>
lemma h_v_add:
"h v = h v \oplus\oplus h v" (is "?a = ?b")
<proof\rangle
lemma exists_list_inter:
"\b. \existsc. set c = set a \cap set b"
<proof>
lemma exists_list_minus:
"\b. \existsc. set c = set a - set b"
<proof>
lemma h_lin:
assumes "set v = set v1 U set v2"
shows "h v = h v1 \oplus\oplus h v2" (is "?a = ?b")
\langleproof>
fun h_comp_some :: "'a m 'b list }=>\mathrm{ 'a" where
"h_comp_some a l = (THE b. b << a ^ \sigma b = \sigma a - set l ^ pure b)"
lemma h_comp_some_exists:
assumes "b = h_comp_some a l"
shows "b << a ^\sigma b = \sigma a - set l ^ pure b"
<proof\rangle

```

\section*{A．Appendix：Isabelle Formalization}
```

definition h_comp :: "'a $\Rightarrow$ 'b list $\Rightarrow$ 'a" where
"h_comp $\varphi V=\left(\right.$ the $\left(s \_C ~ \varphi \oplus\right.$ Some (h_comp_some $\left.\varphi V\right)$ ))"
lemma h_comp_some_sum:
"Some (h_comp $\varphi$ V) = s_C $\varphi \oplus$ Some (h_comp_some $\varphi$ V)"
〈proof〉
lemma h_comp_store:
" $\sigma($ h_comp $\varphi$ l) $=\sigma \varphi-\operatorname{set} 1 "$
〈proof〉
lemma h_comp_grows:
assumes "c << a"
shows "h_comp c $V$ << h_comp a $V^{\prime}$
〈proof〉
lemma h_comp_lin:
assumes "Some a = Some a1 $\oplus$ Some a2"
shows "Some (h_comp a V) = Some (h_comp a1 V) $\oplus$ Some (h_comp a2 V)"
〈proof〉
lemma h_comp_not_here:
assumes "set $V \cap \sigma a=\{ \}$ "
shows "h_comp a $V=a$ "
〈proof〉
lemma h_comp_sum:
"Some $\varphi=$ Some (h_comp $\varphi x$ ) $\oplus$ s_core $\varphi$ "
〈proof〉
lemma h_comp_h:
assumes " $\varphi \in h V^{\prime}$
shows "h_comp $\varphi V=u$ "
〈proof〉
lemma rename_removes_vars_list:
assumes "wf_renaming (11, 12, 13, do)"
and "set $1 \subseteq$ set do"
shows "set (rename_list 1 (11, 12, 13, do)) $\cap($ set $11 \cup$ set 13$) \subseteq$
set 12 " (is "?a $\subseteq$ ?b")
〈proof〉
lemma list_inclusion:
assumes "wf_renaming $t$ "
and "set $a \subseteq$ set $b$ "
shows "set (rename_list a $t$ ) $\subseteq$ set (rename_list bt)" (is "?a $\subseteq$ ?b")
〈proof〉

```
```

lemma rename_list_same:
assumes "wf_renaming (11, 12, 13, do)"
shows "rename_list l1 (11, 12, 13, do) = 12"
<proof>
lemma rename_modif_for_q:
assumes "wf_renaming (11, 12, 13, do)"
and "set (read_pred Q) \subseteq set 11"
shows "set (read_pred (rename_pred Q (11, 12, 13, do))) \subseteq set 12"
(is "?a \subseteq ?b")
<proof\rangle
lemma rename_removes_vars_pred:
assumes "wf_renaming (l1, l2, 13, do)"
and "set (read_pred P) \subseteq set do"
shows "set (read_pred (rename_pred P (11, l2, l3, do))) \cap (set l1 U
set 13) \subset set 12"
\langleproof\rangle
lemma rename_inv_def_pred:
assumes "wf_renaming t"
shows "rename_pred (rename_pred P t) (rename_inv t) = P"
\langleproof〉
lemma rename_pred_lnot:
"rename_pred (lnot b) t = lnot (rename_pred b t)"
<proof\rangle
lemma rename_list_same_length: "wf_renaming t \Longrightarrow length (rename_list
l t) = length l"
\langleproof\rangle
lemma rename_list_concat: "wf_renaming t \Longrightarrow rename_list (11 @ l2) t
= rename_list l1 t @ rename_list l2 t"
<proof>
lemma rename_elem_list:
assumes "i < length l"
and "wf_renaming t"
shows "rename_list l t ! i = rename_elem (l ! i) t"
<proof\rangle
lemma rename_list_distinct:
assumes "wf_renaming t"
and "distinct l"
shows "distinct (rename_list l t)"
\langleproof\rangle
fun modif :: "('a, 'b, 'c) stmt => 'b list" where

```
```

    "modif (s1 ; s2) = modif s1 @ modif s2"
    | "modif (If s1 s2) = modif s1 @ modif s2"
| "modif (MethodCall y m x) = y"
| "modif (While b I s) = modif s"
| "modif (Assume b) = []"
| "modif (Inhale P) = []"
| "modif (Exhale P) = []"
| "modif (Var l) = l"
| "modif (Havoc l) = 1"
| "modif Skip = []"
| "modif (Other s) = modif_other s"
fun sigma_list :: "'a \# 'b list" where
"sigma_list }\varphi=(SOME l. set l = \sigma \varphi)"
lemma sigma_list_def:
"set (sigma_list \varphi) =\sigma \varphi"
<proof>

```

\section*{A.3.2 Semantics}
```

function semantics :: "('a, 'b, 'c) program = 'a = ('a, 'b, 'c) stmt
=> 'a ss" where
"semantics Pr \varphi Skip = S {\varphi}"
| "semantics Pr \varphi (Assume b) = (if well_defined_assert b \varphi then if b
\varphi then S {\varphi} else S {}
else Error)"
| "semantics Pr \varphi (s1 ; s2) = (let r = semantics Pr \varphi s1 in
if r = Error then Error
else let A = get_S r in
union_set_ss ((\lambda\varphi'. semantics Pr \varphi' s2) ' A))"
| "semantics Pr \varphi (If s1 s2) = (let r1 = semantics Pr \varphi s1 in
let r2 = semantics Pr \varphi s2 in
if r1 = Error \vee r2 = Error then Error
else S (get_S r1 U get_S r2))"
| "semantics Pr \varphi (Var x) = (if set x \cap\sigma \varphi = {} then S ({\varphi} \oplus\oplus hx)
else Error)"
| "semantics Pr \varphi (Other s) = semantics_other Pr \varphi s"
| "semantics Pr \varphi (Havoc x) = (if set x\subseteq\sigma \varphi then S ({h_comp \varphi x} \oplus\oplus
h x) else Error)"
| "semantics Pr \varphi (Inhale P) = (if well_defined_assert P \varphi then S ({\varphi}
\oplus\oplus Inh P) else Error)"
| "semantics Pr \varphi (Exhale P) =
(if P \varphi ^ well_defined_assert P \varphi then S {\varphi' | \varphi' i r. Some }\varphi=\mathrm{ Some
i}\oplus\mathrm{ Some r ^ i f Inh P ^ Some 的'=s_core i }\oplus\mathrm{ Some r}

```
```

    else Error)"
    | "semantics Pr \varphi (MethodCall y m x) = (
if set x \cup set y \subseteq\sigma \varphi then
let (_, args, ret, P, Q, _) = the (get_method Pr m) in semantics Pr
\varphi
(Exhale (rename_pred P (args @ ret, x @ y, [], [])); Havoc y ; Inhale
(rename_pred Q (args @ ret, x @ y, [], [])))
else Error)"
| "semantics Pr \varphi (While b I s) =
(let V = modif s in
if semantics Pr ( |\varphi| ) (Havoc V; Inhale I; Assume b; s ; Exhale I)
= Error then
Error
else
semantics Pr \varphi (Exhale I; Havoc V; Inhale I ; Assume (lnot b)))"
<proof>
termination
<proof\rangle
fun read :: "('a, 'b, 'c) stmt => 'b list" where
"read (s1 ; s2) = read s1 @ read s2"
| "read (If s1 s2) = read s1 @ read s2"
| "read (MethodCall y m x) = x @ y"
| "read (While b I s) = read s @ read_pred I @ read_pred b"
| "read (Assume b) = read_pred b"
| "read (Inhale P) = read_pred P"
| "read (Exhale P) = read_pred P"
| "read (Var 1) = 1"
| "read (Havoc 1) = 1"
| "read Skip = []"
| "read (Other s) = read_other s"
fun rename :: "('a, 'b, 'c) stmt }=>\mathrm{ ' 'b rename_t }=>\mathrm{ ('a, 'b, 'c) stmt"
where
"rename (Inhale P) t = Inhale (rename_pred P t)"
| "rename (Exhale P) t = Exhale (rename_pred P t)"
| "rename (s1 ; s2) t = (rename s1 t ; rename s2 t)"
| "rename (Var x) t = Var (rename_list x t)"
| "rename (Havoc x) t = Havoc (rename_list x t)"
| "rename (If s1 s2) t = If (rename s1 t) (rename s2 t)"
| "rename (While b I s) t = While (rename_pred b t) (rename_pred I t)
(rename s t)"
| "rename (MethodCall y m x) t = MethodCall (rename_list y t) m (rename_list
x t)"
| "rename (Assume b) t = Assume (rename_pred b t)"
| "rename (Other other) t = Other (rename_other other t)"
| "rename Skip t = Skip"

```
```

fun bigger_ss :: "'a ss $\Rightarrow$ 'a ss $\Rightarrow$ bool" (infixl ">>>" 60) where
"_ >>> Error $\longleftrightarrow$ True"
| "Error >>> _ $\longleftrightarrow$ False"
| "S A >>> S B $\longleftrightarrow A$ >> B"
fun sem :: "('a, 'b, 'c) program $\Rightarrow$ 'a set $\Rightarrow$ ('a, 'b, 'c) stmt $\Rightarrow$ 'a
set" where
"sem $\operatorname{Pr} A \mathrm{~s}=(\cup \mathrm{a} \in A$. get_S (semantics Pr a s))"
definition ver :: "('a, 'b, 'c) program $\Rightarrow$ 'a set $\Rightarrow$ ('a, 'b, 'c) stmt
$\Rightarrow$ bool" where
"ver Pr A st $\longleftrightarrow(\forall a \in A$. semantics Pr a st $\neq$ Error)"
definition mono :: "('a, 'b, 'c) program $\Rightarrow$ (' $a, ~ ’ b, ~ ' c) ~ s t m t ~ \Rightarrow b o o l " ~$
where
"mono Pr s $\longleftrightarrow(\forall a b . b \ll a \longrightarrow$ semantics $\operatorname{Pr}$ a $s \ggg$ semantics $\operatorname{Pr}$
bs)"
definition smonoIn :: "('a, 'b, 'c) program $\Rightarrow(' a, \quad, b, \quad c)$ stmt $\Rightarrow b o o l "$
where
"smonoIn Pr st $\longleftrightarrow(\forall A B . A \gg B \longrightarrow(v e r \operatorname{Pr} B$ st $\longrightarrow$ ver Pr $A$ st))"
definition smonoOut :: "('a, 'b, 'c) program $\Rightarrow$ ('a, 'b, 'c) stmt $\Rightarrow$ bool"
where
"smonoOut Pr st $\longleftrightarrow(\forall A B . \operatorname{ver} \operatorname{Pr} B$ st $\wedge A \gg B \longrightarrow$ (sem Pr A st >>
sem Pr B st))"
definition smono :: "('a, 'b, 'c) program $\Rightarrow(' a, \quad b, \quad c)$ stmt $\Rightarrow$ bool"
where
"smono Pr st $\longleftrightarrow$ smonoIn Pr st $\wedge$ smonoOut Pr st"
lemma v_singleton: "ver $\operatorname{Pr} A s \longleftrightarrow(\forall a \in A . v e r \operatorname{Pr}\{a\} s) "$
〈proof〉
lemma s_singleton: "sem $\operatorname{Pr} A s=(\cup a \in A . \operatorname{sem} \operatorname{Pr}\{a\} s) "$
$\langle p r o o f\rangle$
lemma elem_sem:
" $x \in$ sem $\operatorname{Pr} A s \longleftrightarrow(\exists a \in A . x \in \operatorname{sem} \operatorname{Pr}\{a\} s)$ "
〈proof〉
lemma smaller_error:
assumes "Error >>> x"
shows " $x=$ Error"
〈proof〉
lemma bigger_not_error:
assumes " $x$ >>> $y$ "
and " $y \neq E r r o r "$

```
```

    shows "x \not= Error"
    <proof\rangle
lemma smonoIn_singleton:
"smonoIn Pr s \longleftrightarrow (\foralla b. b << a \longrightarrow (ver Pr {b} s \longrightarrow ver Pr {a} s))"
(is "?a \longleftrightarrow ?b")
<proof\rangle
lemma smonoOut_singleton:
"smonoOut Pr s \longleftrightarrow (\foralla b. ver Pr {b} s ^ b << a \longrightarrow (sem Pr {a} s
>> sem Pr {b} s))" (is "?a \longleftrightarrow ?b")
\langleproof\rangle
lemma mono_smono:
"mono Pr s < smono Pr s" (is "?a \longleftrightarrow ?b")
<proof\rangle
fun wf_stmt :: "('a, 'b, 'c) program \# ('a, 'b, 'c) stmt => bool" where
"wf_stmt Pr (s1; s2) \longleftrightarrow wf_stmt Pr s1 ^ wf_stmt Pr s2"
| "wf_stmt Pr (If s1 s2) \longleftrightarrow wf_stmt Pr s1 ^ wf_stmt Pr s2"
| "wf_stmt Pr (While b I s) \longleftrightarrow wf_assert I ^ wf_stmt Pr s"
| "wf_stmt Pr (MethodCall y m x) \longleftrightarrow (let r = get_method Pr m in
r f None ^ (let (_, args, ret, _, _, _) = the r in length x = length
args
^ length y = length ret ^ distinct (x @ y)))"
| "wf_stmt Pr (Inhale P) \longleftrightarrow supported P"
| "wf_stmt Pr (Exhale P) \longleftrightarrow supported P"
| "wf_stmt Pr (Other other) \longleftrightarrow wf_other Pr other"
| "wf_stmt Pr _ \longleftrightarrow True"
fun wf_method :: "('a, 'b, 'c) program = ('a, 'b, 'c) method \# bool"
where
"wf_method Pr (m, args, ret, P, Q, s) \longleftrightarrow
wf_assert P ^ wf_assert Q ^ distinct (args @ ret) ^ set args \cap set
(modif s) = {} ^
set (read_pred P) \subseteq set args ^ wf_stmt Pr s ^ set (read_pred Q) \subseteq
set args U set ret"
fun wf_program_aux :: "('a, 'b, 'c) program \# ('a, 'b, 'c) program =
bool" where
"wf_program_aux Pr [] \longleftrightarrow True"
| "wf_program_aux Pr (t \# q) \longleftrightarrow wf_method Pr t ^ wf_program_aux Pr q"
fun wf_program :: "('a, 'b, 'c) program \# bool" where
"wf_program Pr \longleftrightarrow wf_program_aux Pr Pr"
lemma get_method_same_name:
"get_method Pr m = None \Longrightarrow fst (the (get_method Pr m)) = m"
\langleproof\rangle

```
```

lemma simple_method_exists:
assumes "wf_stmt Pr (MethodCall y m x)"
shows " }\exists\mathrm{ args ret P Q s. get_method Pr m = Some (m, args, ret, P, Q,
s)"
<proof\rangle
lemma sem_loop:
assumes "ver Pr {\varphi} (While b I s)"
shows "sem Pr {\varphi} (While b I s) = sem Pr {\varphi} (Exhale I; Havoc (modif
s); Inhale I; Assume (lnot b))"
<proof\rangle
lemma ver_method_real:
assumes "get_method Pr m = Some (m, args, ret, P, Q, s)"
shows "ver Pr {\varphi} (MethodCall y m x) \longleftrightarrow
ver Pr {\varphi} (Exhale (rename_pred P (args @ ret, x @ y, [], []))
; Havoc y ; Inhale (rename_pred Q (args @ ret, x @ y, [], []))) ^
(set x \cup set y) \subseteq\sigma \varphi'
<proof\rangle
lemma ver_loop: "ver Pr {\varphi} (While b I s) \longleftrightarrow
ver Pr {\varphi} (Exhale I; Havoc (modif s); Inhale I ; Assume (lnot b)) ^
ver Pr { |\varphi| } (Havoc (modif s); Inhale I; Assume b; s ; Exhale I)"
<proof>
lemma sem_method_real:
assumes "get_method Pr m = Some (m, args, ret, P, Q, s)"
and "ver Pr {\varphi} (MethodCall y m x)"
shows "sem Pr {\varphi} (MethodCall y m x) =
sem Pr {\varphi} (Exhale (rename_pred P (args @ ret, x @ y, [], []));
Havoc y ;
Inhale (rename_pred Q (args @ ret, x @ y, [], [])))"
<proof\rangle
definition well_defined_assert_set :: "'a assertion }=>\mathrm{ 'a set }=>\mathrm{ b bool" where
"well_defined_assert_set P A \longleftrightarrow (
lemma ver_inhale_single:
"well_defined_assert P \varphi \longleftrightarrow ver Pr {\varphi} (Inhale P)"
<proof\rangle
lemma ver_inhale:
"well_defined_assert_set P A \longleftrightarrow ver Pr A (Inhale P)"
\langleproof\rangle
lemma union_sum:

```
```

        "(\cupa\inA. {a} \oplus\oplusB) = A \oplus\oplus B" (is "?a = ?b")
    ```
        "(\cupa\inA. {a} \oplus\oplusB) = A \oplus\oplus B" (is "?a = ?b")
<proof\rangle
```

```
lemma sem_inhale:
    assumes "well_defined_assert_set P A"
    shows "sem Pr A (Inhale P) = A \oplus\oplus Inh P"
    <proof>
lemma ver_exhale:
    "ver Pr {\varphi} (Exhale P) = P \varphi"
    \langleproof\rangle
lemma singleton_sem:
    assumes "ver Pr {\varphi} s"
    shows "sem Pr {\varphi} s = get_S (semantics Pr \varphi s)"
    \langleproof\rangle
lemma sem_exhale:
    assumes " P \varphi"
    shows "sem Pr {\varphi} (Exhale P) = {\varphi' | \varphi' i r. Some \varphi = Some i }\oplus\mathrm{ Some
r ^i G Inh P ^ Some \varphi' = s_core i }\oplus\mathrm{ Some r}'"
\langleproof\rangle
lemma sem_skip:
    "(ver Pr A Skip) ^ (sem Pr A Skip = A)"
<proof>
lemma ver_if:
    "ver Pr A (If s1 s2) = (ver Pr A s1 ^ ver Pr A s2)"
    \langleproof\rangle
lemma sem_if_single:
    assumes "ver Pr {\varphi} (If s1 s2)"
    shows "sem Pr {\varphi} (If s1 s2) = sem Pr {\varphi} s1 U sem Pr {\varphi} s2"
    <proof>
lemma sem_if:
    assumes "ver Pr A (If s1 s2)"
    shows "sem Pr A (If s1 s2) = sem Pr A s1 U sem Pr A s2"
<proof\rangle
lemma sem_assume_true:
    assumes "b \varphi"
        and "well_defined_assert b \varphi"
    shows "sem Pr {\varphi} (Assume b) = {\varphi}"
    \langleproof\rangle
lemma sem_assume_false:
    assumes "\negb \varphi"
    shows "sem Pr {\varphi} (Assume b) = {}"
    <proof\rangle
```

```
lemma ver_havoc:
    "ver \(\operatorname{Pr}\{\varphi\}\) (Havoc 1) \(\longleftrightarrow\) set \(1 \subseteq \sigma \varphi\) "
    〈proof〉
lemma sem_havoc:
    assumes "ver \(\operatorname{Pr}\{\varphi\}\) (Havoc 1)"
    shows "sem \(\operatorname{Pr}\{\varphi\}(\) Havoc 1\()=\left\{h \_c o m p ~ \varphi l\right\} \oplus \oplus\) l"
    〈proof〉
lemma ver_var:
    "ver \(\operatorname{Pr}\{\varphi\}(\operatorname{Var} x) \longleftrightarrow\) set \(x \cap \sigma \varphi=\{ \} "\)
    〈proof〉
lemma sem_var:
    assumes "ver Pr A (Var x)"
    shows "sem \(\operatorname{Pr} A(\operatorname{Var} x)=A \oplus \oplus h x^{\prime}\)
    〈proof〉
lemma ver_seq_single:
    "ver \(\operatorname{Pr}\{\varphi\}(S e q ~ s 1 s 2) \longleftrightarrow \operatorname{ver} \operatorname{Pr}\{\varphi\} \operatorname{si} \wedge \operatorname{ver} \operatorname{Pr}(\operatorname{sem} \operatorname{Pr}\{\varphi\} s 1)\)
s2" (is "?a \(\longleftrightarrow ~ ? b ") ~\)
〈proof〉
lemma ver_seq:
    \(" v e r \operatorname{Pr} A(S e q ~ s 1 ~ s 2) \longleftrightarrow v e r \operatorname{Pr} A s 1 \wedge \operatorname{ver} \operatorname{Pr}(s e m \operatorname{Pr} A \operatorname{si}) s 2 "\)
〈proof〉
lemma union_ss:
    assumes "union_set_ss \(\left(\left(\lambda \varphi^{\prime}\right.\right.\). semantics \(\left.\operatorname{Pr} \varphi^{\prime} s\right)\) ‘ \(\left.A\right)=S r "\)
        and "ver \(\operatorname{Pr} A \mathrm{~s}\) "
        shows " \(r\) = sem \(\operatorname{Pr} A \mathrm{~s}\) "
\(\langle p r o o f\rangle\)
lemma sem_seq_single:
    assumes "ver \(\operatorname{Pr}\{\varphi\}\) (s1 ; s2)"
        shows "sem \(\operatorname{Pr}\{\varphi\}(S e q ~ s 1 ~ s 2)=s e m \operatorname{Pr}(\operatorname{sem} \operatorname{Pr}\{\varphi\}\) s1) s2" (is "?a
= ?b")
〈proof〉
lemma sem_seq:
    assumes "ver Pr A (s1 ; s2)"
    shows "sem Pr A (Seq s1 s2) = sem Pr (sem Pr A s1) s2"
\(\langle\) proof〉
lemma rename_set_empty:
    assumes "wf_renaming \(t\) "
    shows "rename_set \(\{u\} t=\{u\} "\) (is "?a = ?b")
〈proof〉
```

```
lemma store_same_pred_supp_inhale:
    assumes "supported P"
        and "ver Pr {a} (Inhale P)"
        and "b c sem Pr {a} (Inhale P)"
        shows "\sigma b = \sigma a"
<proof>
lemma exhale_verif:
    assumes "ver Pr {\varphi} (Exhale P)"
        shows "P \varphi"
    \langleproof\rangle
lemma inh_and_supported:
    assumes "supported P"
            and "a \in Inh P"
            and "b G Inh P"
            and "a << \varphi"
            and " }b<<\varphi\mathrm{ "
            and "P \varphi"
        shows "a = b"
    <proof\rangle
lemma exhale_sem_inh:
    assumes "supported P"
            and "i \in Inh P"
            and "Some \varphi = Some i }\oplus\mathrm{ Some r"
            and "ver Pr {\varphi} (Exhale P)"
            shows "sem Pr {\varphi} (Exhale P) = {the (s_core i \oplus Some r)}"
<proof>
lemma store_same_pred_supp_exhale:
    assumes "supported P"
            and "ver Pr {a} (Exhale P)"
            and "b f sem Pr {a} (Exhale P)"
            shows "\sigma b = \sigma a"
<proof>
lemma modif_in_read: "set (modif s) \subseteq set (read s)"
\langleproof\rangle
lemma rename_modif_list:
    assumes "wf_renaming t"
    shows "modif (rename s t) = rename_list (modif s) t"
<proof>
```

lemma rename_modif_no_inter_elem: "wf_renaming t $\wedge$ set $1 \cap$ set (modif
s) $=$ \{\}
$\Longrightarrow \operatorname{set}\left(r e n a m e \_l i s t l t\right) \cap \operatorname{set}(m o d i f(r e n a m e ~ s t))=\{ \}^{\prime \prime}$

## A．Appendix：Isabelle Formalization

```
(is "?a \Longrightarrow ?b")
<proof\rangle
lemma rename_store_inv: "wf_renaming t ^ set (modif (rename s t)) \cap
\sigma (rename_state r t) = {} \Longrightarrow set (modif s) \cap \sigma r = {}" (is "?a \Longrightarrow ?b")
\langleproof\rangle
lemma rename_modif_no_inter:
    assumes "wf_renaming (args @ ret, x @ y, l, do)"
    and "length args = length x"
    and "set args \cap set (modif s) = {}"
    and "set (modif s) \subseteq set do"
    shows "set x \cap set (modif (rename s (args @ ret, x @ y, l, do)))
= {}"
<proof\rangle
```

lemma rename_store_lemma: "wf_renaming $t \Longrightarrow \operatorname{set}(m o d i f(r e n a m e s t))$
$\cap \sigma r=\{ \} \Longrightarrow \operatorname{set}($ modif $s) \cap \sigma$ (rename_state $\left.r\left(r e n a m e \_i n v t\right)\right)=\{ \}^{\prime \prime}$
〈proof〉
lemma rename_removes_vars:
"wf_renaming (11, 12, 13, do) $\wedge$ set (read s) $\subseteq$ set do $\longrightarrow$ set (read
$($ rename $s(11,12,13, d o)))($ set $11 \cup$ set 13$) \subseteq$ set 12 "
〈proof〉
lemma rename_list_inv:
assumes "wf_renaming $t$ "
shows "rename_list (rename_list l t) (rename_inv t) = l"
〈proof〉
lemma renaming_invert:
"wf_renaming $t \longrightarrow$ rename (rename $s t$ ) (rename_inv $t$ ) = $s$ "
〈proof〉
lemma member_rename_set_inv:
assumes "wf_renaming $t$ "
shows "rename_state $x$ (rename_inv $t$ ) $\in A \longleftrightarrow x \in$ rename_set $A$ t"
〈proof〉
lemma h_rename_set:
assumes "wf_renaming $t$ "
shows " $h$ (rename_list $V$ t) $=$ rename_set ( $h$ V) t" (is "?a = ?b")
〈proof〉
lemma rename_set_add:
assumes "wf_renaming $t$ "
shows "rename_set $(A \oplus \oplus B) t=r e n a m e \_s e t A t \oplus \oplus$ rename_set $B$ t"
(is "?A = ?B")
〈proof〉

```
lemma h_comp_c_same:
    "C (h_comp x l) = C x"
    <proof\rangle
lemma wf_renaming_rename_list_set:
    assumes "wf_renaming t"
        and "set A = set B - set D"
        shows "set (rename_list A t) = set (rename_list B t) - set (rename_list
D t)" (is "?a = ?b")
\langleproof\rangle
lemma h_comp_rename:
    assumes "wf_renaming t"
    shows "h_comp (rename_state \varphi t) (rename_list V t) = rename_state (h_comp
\varphi V) t" (is "?a = ?b")
\langleproof\rangle
lemma wf_core_rename_same:
    assumes "wf_renaming t"
    shows "rename_state ( |x| ) t = | rename_state x t |"
<proof>
definition rename_sem_property :: "('a, 'b, 'c) program = ('a, 'b, 'c)
stmt }=>\mathrm{ 'b rename_t }=>\mathrm{ bool" where
    "rename_sem_property Pr s t \longleftrightarrow (\forall\varphi. wf_renaming t ^ wf_stmt Pr s
^wf_program Pr ^ ver Pr {\varphi} s }
            ver Pr {rename_state \varphi t} (rename s t) ^ sem Pr {rename_state \varphi t}
(rename s t) = rename_set (sem Pr {\varphi} s) t)"
lemma rename_sem_inhale:
    assumes "wf_renaming t"
        and "ver Pr {\varphi} (Inhale P)"
            shows "ver Pr {rename_state \varphi t} (rename (Inhale P) t)"
            and "sem Pr {rename_state \varphi t} (rename (Inhale P) t) = rename_set
(sem Pr {\varphi} (Inhale P)) t"
<proof>
lemma rename_sem_assume:
    assumes "wf_renaming t"
        and "ver Pr {\varphi} (Assume b)"
            shows "ver Pr {rename_state \varphi t} (rename (Assume b) t)"
        and "sem Pr {rename_state \varphi t} (rename (Assume b) t) = rename_set
(sem Pr {\varphi} (Assume b)) t"
<proof>
lemma rename_sem_exhale:
    assumes "wf_renaming t"
    and "ver Pr {\varphi} (Exhale P)"
```

```
    shows "ver Pr {rename_state \varphi t} (rename (Exhale P) t)"
    and "sem Pr {rename_state \varphi t} (rename (Exhale P) t) = rename_set
(sem Pr {\varphi} (Exhale P)) t"
<proof\rangle
lemma rename_sem_havoc:
    assumes "wf_renaming t"
            and "ver Pr {\varphi} (Havoc V)"
        shows "ver Pr {rename_state \varphi t} (rename (Havoc V) t)"
    and "sem Pr {rename_state \varphi t} (rename (Havoc V) t) = rename_set
(sem Pr {\varphi} (Havoc V)) t"
<proof\rangle
lemma rename_sem_var:
    assumes "wf_renaming t"
    and "ver Pr {\varphi} (Var V)"
    shows "ver Pr {rename_state \varphi t} (rename (Var V) t)"
    and "sem Pr {rename_state \varphi t} (rename (Var V) t) = rename_set
(sem Pr {\varphi} (Var V)) t"
<proof>
lemma rename_sem_seq:
    assumes "wf_renaming t"
            and "ver Pr {\varphi} (s1 ; s2)"
            and "wf_stmt Pr (s1 ; s2)"
            and "wf_program Pr"
            and "\\varphi. wf_stmt Pr s1 \Longrightarrowwf_program Pr \Longrightarrow ver Pr {\varphi} s1 \Longrightarrow
ver Pr {rename_state \varphi t} (rename s1 t) ^ sem Pr {rename_state \varphi t} (rename
s1 t) = rename_set (sem Pr {\varphi} s1) t"
            and "\\varphi. wf_stmt Pr s2 \Longrightarrow wf_program Pr \Longrightarrow ver Pr {\varphi} s2 \Longrightarrow
ver Pr {rename_state \varphi t} (rename s2 t) ^ sem Pr {rename_state \varphi t} (rename
s2 t) = rename_set (sem Pr {\varphi} s2) t"
    shows "ver Pr {rename_state \varphi t} (rename (s1 ; s2) t)"
        and "sem Pr {rename_state \varphi t} (rename (s1 ; s2) t) = rename_set
(sem Pr {\varphi} (s1 ; s2)) t"
<proof>
lemma seq_rename_sem_property:
    assumes "rename_sem_property Pr s1 t"
            and "rename_sem_property Pr s2 t"
            and "wf_renaming t"
            shows "rename_sem_property Pr (s1 ; s2) t"
<proof\rangle
lemma wf_program_method_aux:
    "wf_program_aux Pr P ^ get_method P name = Some m \Longrightarrow wf_method Pr
m"
<proof\rangle
```

```
lemma wf_method_exists_equiv:
    assumes "get_method Pr m = Some (m, args, ret, P, Q, s)"
    shows "wf_stmt Pr (MethodCall y m x) \longleftrightarrow length x = length args
    ^ length y = length ret ^ distinct (x @ y)"
    <proof\rangle
lemma verif_rename_pred:
    assumes "wf_renaming t"
    shows "(rename_pred P t) a \longleftrightarrowP (rename_state a (rename_inv t))"
    <proof\rangle
lemma smaller_same_one_side:
    assumes "wf_renaming t"
    and "a << b"
    shows "rename_state a t << rename_state b t"
<proof>
lemma smaller_same:
    assumes "wf_renaming t"
    shows "a << b < rename_state a t << rename_state b t"
    <proof\rangle
lemma rename_pred_same_supported:
    assumes "supported P"
        and "wf_renaming t"
    shows "supported (rename_pred P t)"
<proof>
lemma well_defined_assert_monoin:
    assumes "wf_assert P"
        and "well_defined_assert P a"
        and "a << b"
        shows "well_defined_assert P b"
    <proof\rangle
lemma rename_pred_same_intui:
    assumes "wf_assert P"
        and "wf_renaming t"
    shows "wf_assert (rename_pred P t)"
<proof>
lemma same_wf_rename:
    assumes "wf_assert P"
        and "wf_renaming t"
    shows "wf_assert (rename_pred P t)"
    <proof>
lemma wf_stmt_wf_renaming:
    assumes "wf_renaming t"
```

```
    shows "wf_stmt Pr s \longrightarrow wf_stmt Pr (rename s t)"
<proof\rangle
lemma rename_all:
    assumes "wf_renaming t"
    shows "\\varphi. wf_program Pr \Longrightarrow wf_stmt Pr s \Longrightarrow ver Pr {\varphi}s \Longrightarrow ver
Pr {rename_state \varphi t} (rename s t) ^ sem Pr {rename_state \varphi t} (rename
s t) = rename_set (sem Pr {\varphi} s) t"
\langleproof\rangle
lemma rename_ver:
    assumes "wf_renaming t"
        and "wf_program Pr"
        and "wf_stmt Pr s"
    shows "ver Pr {\varphi} s \longleftrightarrow ver Pr {rename_state \varphi t} (rename s t)" (is
"?a \longleftrightarrow ?b")
\langleproof\rangle
lemma rename_sem:
    assumes "wf_renaming t"
        and "wf_program Pr"
        and "wf_stmt Pr s"
        and "ver Pr {\varphi} s"
    shows "sem Pr {rename_state \varphi t} (rename s t) = rename_set (sem Pr
{\varphi}s) t"
    \langleproof\rangle
lemma rename_ver_set:
    assumes "ver Pr A s"
    and "wf_renaming t"
    and "wf_program Pr"
    and "wf_stmt Pr s"
    shows "ver Pr (rename_set A t) (rename s t)"
    <proof>
lemma method_verif_rename:
    assumes "ver Pr {u} (Var (args @ ret) ; Inhale P ; s ; Exhale Q)"
            and "wf_renaming (args @ ret, x @ y, l, do)"
            and "wf_program Pr"
            and "wf_stmt Pr (Var (args @ ret) ; Inhale P ; s ; Exhale Q)"
    shows "ver Pr {u} (Var (x @ y) ; Inhale (rename_pred P (args @ ret,
x @ y, l, do)) ;
            rename s (args @ ret, x @ y, l, do) ; Exhale (rename_pred Q (args
@ ret, x @ y, l, do)))"
<proof>
lemma sem_method:
    assumes "ver Pr {\varphi} (MethodCall y m x)"
        and "wf_stmt Pr (MethodCall y m x)"
```

```
    shows " }\exists\mathrm{ args ret P Q s. get_method Pr m = Some (m, args, ret, P, Q,
s) }
            sem Pr {\varphi} (MethodCall y m x) = sem Pr {\varphi} (Exhale (rename_pred
P (args @ ret, x @ y, [], [])); Havoc y ;
                Inhale (rename_pred Q (args @ ret, x @ y, [], [])))"
    \langleproof\rangle
lemma wf_program_method:
    assumes "wf_program Pr"
        and "get_method Pr name = Some m"
        shows "wf_method Pr m"
    \langleproof\rangle
lemma ver_method:
    assumes "ver Pr {\varphi} (MethodCall y m x)"
    and "wf_stmt Pr (MethodCall y m x)"
    and "wf_program Pr"
    shows "\existsargs ret P Q s.
                wf_method Pr (m, args, ret, P, Q, s) ^
                ver Pr {\varphi} (Exhale (rename_pred P (args @ ret, x @ y, [], []))
; Havoc y ; Inhale (rename_pred Q (args @ ret, x @ y, [], []))) ^
            (set x \cup set y) \subseteq\sigma \varphi^
            length args = length x ^ length ret = length y"
<proof>
lemma modif_property_other:
    "wf_program Pr ^ wf_other Pr other ^ ver Pr {a} (Other other) ^ c \in
sem Pr {a} (Other other) }\longrightarrow\sigmaa\subseteq\sigmac\wedge\sigmac\subseteq\sigma a\cup set (modif (Other
other))"
    <proof>
definition modif_property :: "('a, 'b, 'c) program = ('a, 'b, 'c) stmt
=> bool" where
    "modif_property Pr s }\longleftrightarrow (\foralla c. wf_program Pr ^ wf_stmt Pr s ^ ver
Pr {a} s ^c\in sem Pr {a}s\longrightarrow\sigma a\subseteq\sigmac\wedge\sigmac\subseteq\sigmaa\cupset (modif
s))"
lemma modif_property_inhale:
    "modif_property Pr (Inhale P)"
    <proof>
lemma modif_property_exhale:
    "modif_property Pr (Exhale P)"
    \langleproof\rangle
lemma modif_property_havoc:
    "modif_property Pr (Havoc y)"
<proof>
```

```
lemma store_modif_sem_seq:
    assumes "s = s1 ; s2"
    and "\a c. wf_program Pr ^ wf_stmt Pr s1 ^ ver Pr {a} s1 ^c\in
sem Pr {a} s1\Longrightarrow\sigma a\subseteq\sigmac^\sigma c\subseteq\sigmaa\cup set (modif s1)"
    and "\a c. wf_program Pr ^ wf_stmt Pr s2 ^ ver Pr {a} s2 ^c c\in
sem Pr {a} s2\Longrightarrow\sigmaa\subseteq\sigma c^\sigmac\subseteq\sigma a\cup set (modif s2)"
    and "wf_program Pr ^ wf_stmt Pr s ^ ver Pr {a} s ^c\in sem Pr
{a} s"
    shows "\sigmaa\subseteq\sigmac\wedge\sigmac\subseteq\sigmaa\cupset(modif s)"
<proof\rangle
lemma modif_property_seq:
    assumes "modif_property Pr s1"
            and "modif_property Pr s2"
        shows "modif_property Pr (s1 ; s2)"
<proof\rangle
lemma modif_property_assume:
    "modif_property Pr (Assume b)"
    <proof\rangle
lemma wf_wf_renaming:
    assumes "wf_program Pr"
            and "wf_stmt Pr (MethodCall y m x)"
            and "get_method Pr m = Some (m, args, ret, P, Q, s)"
            shows "wf_renaming (args @ ret, x @ y, l, do)"
            and "wf_renaming (args @ ret, x @ y, l, do)"
<proof\rangle
lemma store_modif_sem: "wf_program Pr ^ wf_stmt Pr s ^ ver Pr {a} s
\wedgec\in sem Pr {a} s\Longrightarrow\sigma a\subseteq\sigmac^\sigma c\subseteq\sigmaa\cup set (modif s)"
\langleproof\rangle
lemma ver_method_set_ret:
    assumes "ver Pr {\varphi} (MethodCall y m x)"
            and "get_method Pr m = Some (m, args, ret, P, Q, s)"
            and "a \in sem Pr {u} (Var (args @ ret) ; Inhale P ; s ; Exhale Q)"
            and "ver Pr {u} (Var (args @ ret) ; Inhale P ; s ; Exhale Q)"
            and "wf_program Pr"
            shows "set ret \subseteq\sigma a"
\langleproof\rangle
lemma h_comp_only_pure: "C \varphi = C (h_comp \varphi l)"
\langleproof\rangle
lemma h_comp_smaller: "h_comp \varphi x << \varphi"
    \langleproof\rangle
lemma var_sem_empty:
```

＂sem $\operatorname{Pr}\{u\}(\operatorname{Var} x)=h x^{\prime \prime}$〈proof〉

```
lemma havoc_concat:
    "h (a @ b) = h a \oplus\oplus h b"
    <proof\rangle
```

lemma havoc_store_bigger:
assumes "set $x \subseteq \sigma \varphi$ "
shows " $\{\varphi\} \gg\{\varphi\} \oplus \oplus h x^{\prime}$
〈proof〉
lemma supported_intui_exhale:
assumes "supported $P$ "
and "intuitionistic P"
shows "ver $\operatorname{Pr} A(E x h a l e P) \longleftrightarrow A \gg$ Inh $P "$
〈proof〉
lemma int_squared_exhale:
assumes "supported $P$ "
and "intuitionistic P"
and " $P \varphi$ "
and "Some $\varphi$ ' = Some $\varphi \oplus$ Some $r$ "
shows "sem $\operatorname{Pr}\{\varphi ’\}($ Exhale $P)=(\operatorname{sem} \operatorname{Pr}\{\varphi\}($ Exhale $P)) \oplus \oplus\{r\} "$
$\langle p r o o f\rangle$
lemma int_squared_inhale:
assumes "supported P"
and "Some $\varphi$ ' = Some $\varphi \oplus$ Some $r$ "
and "wf_assert P"
and "ver $\operatorname{Pr}\{\varphi\}$ (Inhale $P$ )"
shows "sem $\operatorname{Pr}\{\varphi ’\}($ Inhale $P)=(\operatorname{sem} \operatorname{Pr}\{\varphi\}($ Inhale $P)) \oplus \oplus\{r\} "$
〈proof〉
lemma int_squared_exhale_for_sets:
assumes "wf_assert P"
and $" \forall \varphi \in A . P \varphi$ "
shows "sem $\operatorname{Pr}(A \oplus \oplus\{r\})($ Exhale $P) \subseteq(\operatorname{sem} \operatorname{Pr} A(E x h a l e P)) \oplus \oplus$
$\{r\}$ " (is "?ae $\subseteq$ ?be")
〈proof〉
lemma wf_assert_monoIn:
assumes "wf_assert P"
shows "smonoIn Pr (Inhale P)"
and "smonoIn Pr (Exhale P)"
〈proof〉
lemma int_squared_inhale_for_sets:

```
    assumes "wf_assert P"
        and "ver Pr A (Inhale P)"
    shows "sem Pr (A \oplus\oplus{r}) (Inhale P) = (sem Pr A (Inhale P)) \oplus\oplus
{r}"
\langleproof\rangle
lemma smono_comp:
    assumes "smono Pr s1"
            and "smono Pr s2"
        shows "smono Pr (Seq s1 s2)"
<proof\rangle
lemma smono_if:
    assumes "smono Pr s1"
        and "smono Pr s2"
        shows "smono Pr (If s1 s2)"
<proof\rangle
fun framing :: "('a, 'b, 'c) program # ('a, 'b, 'c) stmt # bool" where
    "framing Pr st = (smono Pr st ^
    (\forall\varphi r. ver Pr {\varphi} st ^ list.set (modif st) \cap\sigmar = {} \longrightarrow (sem Pr
({\varphi} \oplus\oplus {r}) st >> (sem Pr {\varphi} st \oplus\oplus {r}))))"
lemma framing_set:
    "framing Pr s smono Pr s ^ (\forallA r. ((ver Pr A s ^ list.set (modif
s) \cap\sigmar={}) \longrightarrow(sem Pr (A \oplus\oplus{r}) s >> (sem Pr A s \oplus\oplus {r}))))"
\langleproof\rangle
lemma simple_rename_set_inv:
    assumes "wf_renaming t"
    shows "rename_set (rename_set A t) (rename_inv t) = A"
<proof\rangle
lemma simple_state_in_set:
    assumes "wf_renaming t"
    shows "rename_state a t frename_set A t \longleftrightarrow a\inA" (is "?a \longleftrightarrow ?b")
    <proof>
lemma simple_renaming_from_set:
    assumes "x \in rename_set A t"
    shows " }\exists\textrm{a}\inA.x=rename_state a t"
    <proof>
lemma rename_set_sum:
    assumes "wf_renaming t"
    shows "rename_set (A \oplus\oplusB) t = rename_set A t \oplus\oplus rename_set B t"
<proof\rangle
lemma rename_set_smaller:
```

```
    assumes "wf_renaming t"
    shows "A >> B \longleftrightarrow rename_set A t >> rename_set B t" (is "?a \longleftrightarrow ?b")
<proof\rangle
lemma mono_renamed:
    assumes "smono Pr s"
        and "wf_renaming t"
        and "wf_program Pr"
        and "wf_stmt Pr s"
        shows "smono Pr (rename s t)"
<proof>
lemma rename_singleton:
    "rename_set {a} t = {rename_state a t}"
    \langleproof\rangle
lemma rename_sem_set:
    assumes "wf_renaming t"
            and "wf_program Pr"
            and "wf_stmt Pr s"
    shows "sem Pr (rename_set A t) s = rename_set (sem Pr A (rename s (rename_inv
t))) t" (is "?a = ?b")
<proof>
lemma framing_renamed:
    assumes "framing Pr s"
    and "wf_renaming t"
    and "wf_program Pr"
    and "wf_stmt Pr s"
shows "framing Pr (rename s t)"
<proof>
lemma assume_false_smono:
    "smono Pr (Assume lfalse)"
<proof>
lemma inhale_smono:
    assumes "wf_assert P"
    shows "smono Pr (Inhale P)"
<proof>
lemma exhale_elem:
    assumes "ver Pr {a} (Exhale P)"
        shows "sa \in sem Pr {a} (Exhale P) \longleftrightarrow (\existsi r. Some a = Some i }\oplus\mathrm{ Some
r ^ i G Inh P ^ Some sa = s_core i }\oplus\mathrm{ Some r)" (is "?a }\longleftrightarrow ?b"
    \langleproof>
lemma exhale_smono:
    assumes "wf_assert P"
```


## A. Appendix: Isabelle Formalization

```
    shows "smono Pr (Exhale P)"
<proof>
lemma havoc_smono:
    "smono Pr (Havoc x)"
<proof\rangle
lemma smono_out_equiv:
    assumes "\\varphi. ver Pr {\varphi} s2 \Longrightarrow sem Pr {\varphi} s1 = sem Pr {\varphi} s2"
            and "smonoOut Pr s1"
            and "\\varphi. ver Pr {\varphi} s2 \Longrightarrow ver Pr {\varphi} s1"
            and "smonoIn Pr s2"
            shows "smonoOut Pr s2"
    <proof>
lemma method_smono_in:
    assumes "wf_program Pr"
        and "wf_stmt Pr (MethodCall y m x)"
        shows "smonoIn Pr (MethodCall y m x)"
<proof\rangle
lemma method_smono:
    assumes "wf_program Pr"
        and "wf_stmt Pr (MethodCall y m x)"
        shows "smono Pr (MethodCall y m x)" (is "smono Pr ?s")
<proof\rangle
fun ver_program_aux :: "('a, 'b, 'c) program # ('a, 'b, 'c) program }
bool" where
    "ver_program_aux Pr [] \longleftrightarrow True"
| "ver_program_aux Pr ((_, args, ret, P, Q, s) # q) \longleftrightarrow ver Pr {u} (Var
(args @ ret) ; Inhale P ; s ; Exhale Q) ^ ver_program_aux Pr q"
fun ver_program :: "('a, 'b, 'c) program # bool" where
    "ver_program Pr \longleftrightarrow ver_program_aux Pr Pr"
end
end
```


## A． 4 Soundness

```
theory Soundness
    imports Semantics
begin
context semantics_algebra
begin
```

definition wfm :: "('a, 'b, 'c) program $\Rightarrow$ 'a assertion $\Rightarrow$ bool" where
$" w f m \operatorname{Pr} b \longleftrightarrow$ smono $\operatorname{Pr}$ (Assume b) $\wedge$ smono $\operatorname{Pr}$ (Assume (lnot b))"
lemma wfm_not_same:
"lnot (lnot b) = b"
〈proof〉
lemma wfm_not:
"wfm Pr $b \longleftrightarrow w f m \operatorname{Pr}$ (lnot b)"
〈proof〉
lemma while_smono_in:
assumes "wf_assert I"
and "smono Pr s"
and "wfm Pr b"
shows "smonoIn Pr (While b I s)"
〈proof〉
lemma while_smono:
assumes "wf_assert I"
and "smono Pr s"
and "wfm Pr b"
shows "smono Pr (While bls)"
〈proof〉
lemma exhale_false_single:
" $\neg$ (ver $\operatorname{Pr}\{\varphi\}$ (Exhale lfalse))"
〈proof〉
fun inlinable :: "('a, 'b, 'c) stmt $\Rightarrow$ bool" where
"inlinable (MethodCall _ _ _) = True"
| "inlinable (While _ _ _) = True"
| "inlinable (Seq a b) = (inlinable a $\vee$ inlinable b)"
| "inlinable (If a b) = (inlinable a $\vee$ inlinable b)"
| "inlinable _ = False"
fun inline :: "('a, 'b, 'c) program $\Rightarrow$ nat $=>~ ' b$ list $\Rightarrow(' a, ~ ' b, ~ ' c)$
stmt $\Rightarrow$ ('a, 'b, 'c) stmt" where
"inline Pr $\mathrm{O}_{\text {_ }}($ MethodCall _ _ _) = Assume lfalse"
| "inline Pr $\mathrm{O}_{-}$(While b__) = Assume (lnot b)"
| "inline Pr (Suc n) l (MethodCall y name x) = (let $r$ = get_method Pr

```
name in
    if \(r=\) None then
        (Exhale lfalse)
    else
        let (_, args, ret, _, _, s) = the \(r\) in
        let new_s = rename s (args @ ret, x @ y, l, read s) in
        let new_l = 1 @ read new_s in
        inline Pr n new_l new_s)"
| "inline \(\operatorname{Pr}(S u c n) 1\) (While b I s) = If (Assume b ; inline Pr n l s
; inline \(\operatorname{Pr} n \mathrm{l}\) (While b I s)) (Assume (lnot b))"
| "inline \(\operatorname{Pr} n 1(S e q a b)=S e q(i n l i n e \operatorname{Pr} n l a)(i n l i n e \operatorname{Pr} n l b) "\)
| "inline \(\operatorname{Pr} n l(I f a b)=I f(i n l i n e \operatorname{Pr} n l a)(i n l i n e \operatorname{Pr} n l b) "\)
| "inline Pr \(0_{-} s=s "\)
| "inline Pr _ _ Skip = Skip"
| "inline Pr _ _ s = s"
lemma not_inlinable_id:
    \(" \neg\) inlinable \(s \Longrightarrow\) inline \(\operatorname{Pr} n \mathrm{l}=\mathrm{s}\) "
    〈proof〉
lemma empty_set_goes_empty:
    assumes "sem Pr A s1 = \{\}"
        and "ver Pr A s1"
    shows "sem Pr A (s1 ; s2) = \{\}"
〈proof〉
```


## lemma semantics＿union：

```
    \("(\cup a \in A\). sem \(\operatorname{Pr}(f a) s)=s e m \operatorname{Pr}(\cup a \in A . f a) s "\)
    〈proof〉
lemma rename＿set＿union：
＂\((\cup a \in A\) ．rename＿set \((f a) t)=r e n a m e \_s e t(\cup a \in A . f a) t "\)〈proof〉
fun
SC ：：＂（＇a，＇b，＇c）program \(\Rightarrow\) nat \(\Rightarrow\)＇b list \(\Rightarrow\)（＇a，＇b，＇c）stmt \(\Rightarrow\) bool＂and
inlinable＿SC ：：＂（＇a，＇b，＇c）program \(\Rightarrow\) nat \(\Rightarrow\)＇\(b\) list \(\Rightarrow\)（＇a，＇b，
＇c）stmt \(\Rightarrow\) bool＂
where
＂SC Pr \(n l\) st \(=(i f(\neg\) inlinable st）then smono Pr st else inlinable＿SC Pr n l st）＂
｜＂inlinable＿SC Pr（Suc n）l（MethodCall y m x）＝（let（＿，args，ret， ＿，＿，s）＝the（get＿method Pr m）in
let new＿s＝rename s（args a ret，x＠y，l，read s）in let new＿l＝ 1 ＠read new＿s in
（framing Pr（inline \(\operatorname{Pr} n\) new＿l new＿s）\(\vee\) framing \(\operatorname{Pr} s\) ）\(\wedge S C\) Pr n new＿l new＿s）＂
｜＂inlinable＿SC Pr（Suc n）l（While b I s）\(\longleftrightarrow ~ w f m ~ P r ~ b ~ \wedge ~(f r a m i n g ~ P r ~\)
```

```
(inline Pr n l s) \vee framing Pr s) ^ SC Pr n l s ^ SC Pr n l (While b
I s)"
| "inlinable_SC Pr n l (Seq a b) = (SC Pr n l a ^ SC Pr n l b)"
| "inlinable_SC Pr n l (If a b) = (SC Pr n l a ^ SC Pr n l b)"
| "inlinable_SC Pr O l (While b _ s) \longleftrightarrow wfm Pr b ^ smono Pr s"
| "inlinable_SC Pr O _ _ = True"
| "inlinable_SC Pr _ _ _ = undefined"
lemma sc_implies_smono:
    "wf_program Pr ^ wf_stmt Pr s ^ SC Pr n l s \Longrightarrow smono Pr s"
    "wf_program Pr ^ wf_stmt Pr s ^ inlinable s ^ inlinable_SC Pr n l s
"smono Pr s"
<proof>
lemma inlinable_same:
    "inlinable s \longleftrightarrow inlinable (rename s t)"
    <proof>
lemma not_inlinable_sc:
    assumes "\neg inlinable s"
        and "SC Pr n l s"
        and "wf_renaming t"
        and "wf_program Pr"
        and "wf_stmt Pr s"
        shows "SC Pr n l (rename s t)"
<proof\rangle
lemma wfm_same:
    assumes "wfm Pr b"
            and "wf_renaming t"
            and "wf_program Pr"
            and "wf_stmt Pr s"
            shows "wfm Pr (rename_pred b t)"
<proof>
lemma sc_method_implies_new:
    assumes "new_s = rename s (args @ ret, x @ y, l, read s)"
            and "new_l = l @ read new_s"
            and "SC Pr (Suc n) l (MethodCall y m x)"
            and "get_method Pr m = Some (m, args, ret, P, Q, s)"
            shows "(framing Pr (inline Pr n new_l new_s) \vee framing Pr s) ^ SC
Pr n new_l new_s"
\langleproof\rangle
lemma smono_inline:
    "SC Pr n l s mmono Pr (inline Pr n l s)"
<proof\rangle
definition no_inter_single :: "’a # 'b list }=>\mathrm{ b bool" where
```

```
    "no_inter_single x l \longleftrightarrow < x \subseteq set l"
definition no_inter :: "'a set }=>\mathrm{ 'b list }=>\mathrm{ bool" where
    "no_inter A l \longleftrightarrow(\foralla\inA. no_inter_single a l)"
lemma inter_sum_sets:
    assumes "no_inter A 1"
            and "no_inter B l"
        shows "no_inter (A \oplus\oplusB) l"
    <proof>
definition SP :: "('a, 'b, 'c) program # nat }=>\mathrm{ ' 'b list }=>\mathrm{ ('a, 'b, 'c)
stmt }=>\mathrm{ bool" where
    "SP Pr n l s \longleftrightarrow ( \forallA A'. ver_program Pr ^ wf_program Pr ^ wf_stmt
Pr s ^ A' >> A ^ SC Pr n l s ^ ver Pr A s ^ no_inter A l ^ no_inter
A’ l ^ set (modif s) \subseteq set l \longrightarrow ver Pr A' (inline Pr n l s) ^ sem Pr
A' (inline Pr n l s) >> sem Pr A s)"
lemma havoc_int_squared_single:
    assumes "Some a = Some v \oplus Some r"
            and "set x \cap\sigma r = {}"
            and "ver Pr {v} (Havoc x)"
        shows "sem Pr {a} (Havoc x) = sem Pr {v} (Havoc x) \oplus\oplus {r}" (is "?a
= ?b")
<proof>
lemma havoc_invertible:
    assumes "set V\subseteq\sigma a"
            and "Some b = Some (h_comp a V) \oplus Some hv"
            and "hv \inh V"
            shows "\existshv'\inh V. Some a = Some (h_comp b V) \oplus Some hv'"
<proof\rangle
lemma decompo_sigma:
    assumes "\sigma p=a\cupb"
            and "a\capb = {}"
            and "pure p"
            shows "\existspa pb. Some p = Some pa \oplus Some pb ^\sigma pa = a ^\sigma pb = b"
\langleproof\rangle
lemma smaller_defined:
    assumes "a << b"
            and "b ## r"
            shows "a ## r"
    \langleproof\rangle
lemma pure_defined_sum:
    assumes "Some x = Some xx \oplus Some p"
        and "pure p"
```

```
        and "xx ## r"
        and "p ## r"
    shows "x ## r"
<proof>
lemma havoc_int_squared:
    assumes "set x \cap\sigmar={}"
            and "ver Pr A (Havoc x)"
            and "}\\varphi.\varphi\inA\Longrightarrow set x\subseteq\sigma\varphi
            shows "sem Pr (A \oplus\oplus {r}) (Havoc x) = sem Pr A (Havoc x) \oplus\oplus {r}"
(is "?a = ?b")
<proof\rangle
lemma simple_sem_exhale:
    assumes "wf_assert P"
            and "Some \varphi = Some i }\oplus\mathrm{ Some r"
            and "i \in Inh P"
            and "Some \varphi' = s_core i }\oplus\mathrm{ Some r"
            shows "sem Pr {\varphi} (Exhale P) = {\varphi'}" (is "?a = ?b")
<proof\rangle
lemma h_set_pure:
    assumes "A >> B"
            and "A >> h x"
            shows "A >> B \oplus\oplus h x"
<proof>
lemma bigger_h_single:
    assumes "set x \subseteq\sigma a"
    shows "{a} >> {h_comp a x} \oplus\oplus h x"
<proof\rangle
lemma core_i_phi_exhale:
    assumes "Some i_phi = Some i }\oplus\mathrm{ Some x"
            and "Some core_i_phi = s_core i }\oplus\mathrm{ Some x"
            and "i \in Inh P"
            and "supported P"
            and "P i_phi"
            shows "sem Pr {i_phi} (Exhale P) = {core_i_phi}" (is "?a = ?b")
<proof>
lemma get_method_inlined:
    assumes "get_method Pr m = Some (m, args, ret, P, Q, s)"
            and "new_s = rename s (args @ ret, x @ y, l, read s)"
            and "new_l = l @ read new_s"
            shows "inline Pr (Suc n) l (MethodCall y m x) = inline Pr n new_l
new_s"
```

$\langle p r o o f\rangle$
lemma no＿inter＿inline＿general：
＂wf＿program Pr $\wedge$ wf＿stmt Pr $s \Longrightarrow$ set $1 \cap$ set（modif（inline Pr $n 1$
s））$\subseteq$ set（modif $s$ ）＂
〈proof〉
lemma ver＿program＿method＿verif＿aux：
＂ver＿program＿aux Pr Pr＿bis $\wedge$ Some（m，args，ret，P，Q，s）＝get＿method Pr＿bis $m \Longrightarrow$ ver $\operatorname{Pr}\{u\}$（Var（args＠ret）；Inhale $P$ ；s ；Exhale Q）＂〈proof〉
lemma ver＿program＿method＿verif：
assumes＂ver＿program Pr＂
and＂Some（ $m$ ，args，ret，$P, Q, s$ ）＝get＿method Pr m＂
shows＂ver $\operatorname{Pr}\{u\}$（Var（args＠ret）；Inhale $P$ ；s ；Exhale Q）＂ $\langle p r o o f\rangle$
lemma verif＿inhale＿var＿alone：
assumes＂ver Pr \｛u\} (Var (args @ ret) ; Inhale P)"
shows＂sem Pr \｛u\} (Var (args @ ret) ; Inhale $P$ ）$=h$ args $\oplus \oplus h$ ret
$\oplus \oplus$ Inh $P^{\prime \prime}$
$\langle p r o o f\rangle$
lemma verif＿inhale＿var：
assumes＂ver $\operatorname{Pr}\{u\}$（Var（args＠ret）；Inhale $P$ ；s）＂
shows＂sem $\operatorname{Pr}\{u\}$（Var（args＠ret）；Inhale $P$ ；s）＝sem Pr（h args
$\oplus \oplus$ h ret $\oplus \oplus$ Inh P）s＂
$\langle p r o o f\rangle$
lemma ver＿asso：
＂ver $\operatorname{Pr} A(s 1 ;(s 2 ; s 3)) \longleftrightarrow \operatorname{ver} \operatorname{Pr} A((s 1 ; s 2) ; s 3) "$
〈proof〉
lemma sem＿asso：
assumes＂ver $\operatorname{Pr} A(s 1 ; s 2$ ；s3）＂
shows＂sem $\operatorname{Pr} A(s 1 ;(s 2 ; s 3))=s e m \operatorname{Pr} A((s 1 ; s 2) ; s 3) "$
〈proof〉

## A．4．1 Method case

lemma method＿inlining＿induct：
assumes＂$S P$ Pr $n$ new＿l new＿s＂
and＂Some（ $m$ ，args，ret，$P, Q, s$ ）＝get＿method Pr m＂
and＂new＿s＝rename s（args＠ret，x＠y，l，read s）＂
and＂new＿l＝ 1 ＠read new＿s＂
shows＂SP Pr（Suc n）l（MethodCall y m x）＂
〈proof〉

```
lemma bigger_set_singleton:
    assumes "\a. a \inA\Longrightarrow(\existsb\inB. sem Pr {a} s1 >> sem Pr {b} s2)"
    shows "sem Pr A s1 >> sem Pr B s2"
\langleproof\rangle
lemma instantiate_SP:
    assumes "ver_program Pr ^ wf_program Pr ^ wf_stmt Pr s ^ {\varphi'} >> {\varphi}
\SC Pr n l s ^ ver Pr {\varphi} s ^ no_inter {\varphi'} l ^ no_inter {\varphi} l ^ set
(modif s) \subseteq set l"
            and "SP Pr n l s"
            shows "ver Pr {\varphi'} (inline Pr n l s) ^ sem Pr {\varphi'} (inline Pr n l
s) >> sem Pr {\varphi} s"
    \langleproof>
definition f :: "'a set }=>\mathrm{ 'a assertion }=>\mathrm{ 'a set" where
    "f A b = Set.filter b A"
lemma f_singleton:
    "(\cupa\inA. f {a} b) = f A b" \langleproof\rangle
lemma sem_assume_filter:
    assumes "ver Pr A (Assume b)"
    shows "sem Pr A (Assume b) = f A b"
<proof>
lemma f_inclusion:
    "fAb\subseteqA"\langleproof\rangle
lemma wfm_f:
    assumes "wfm Pr b"
        and "A >> B"
        and "ver Pr B (Assume b)"
        shows "f A b >> f B b"
    <proof>
lemma if_assume_then_true:
    assumes "x\in sem Pr A (Assume b)"
        and "ver Pr A (Assume b)"
    shows "b x"
    \langleproof>
lemma after_while:
    assumes "ver Pr A (While b I s)"
        and "sa \in sem Pr A (While b I s)"
        shows "lnot b sa"
<proof>
lemma general_same_f:
    assumes "\a. a \in A\Longrightarrowb a"
```

```
    shows "f A b = A"
    <proof>
lemma after_while_same_f:
    assumes "ver Pr A (While b I s)"
    shows "f (sem Pr A (While b I s)) (lnot b) = sem Pr A (While b I s)"
    <proof>
lemma after_inline_while:
    "ver Pr A (inline Pr n l (While b I s)) \Longrightarrow sa \in sem Pr A (inline Pr
n l (While b I s)) \Longrightarrow (lnot b sa)"
<proof\rangle
lemma after_inline_while_same_f:
    assumes "ver Pr A (inline Pr n l (While b I s))"
    shows "f (sem Pr A (inline Pr n l (While b I s))) (lnot b) = sem Pr
A (inline Pr n l (While b I s))"
    <proof>
lemma wfm_bigger:
    assumes "c << a"
        and "wfm Pr b"
        and "bc"
        and "ver Pr {c} (Assume b)"
    shows "b a"
<proof\rangle
lemma simple_SP:
    assumes " }\AA\mathrm{ '. ver_program Pr ^ wf_program Pr ^ wf_stmt Pr s ^A,
>> A ^ SC Pr n l s ^ ver Pr A s ^ no_inter A' l ^ no_inter A l ^ set
(modif s) \subseteq set l \Longrightarrow ver Pr A' (inline Pr n l s) ^ sem Pr A' (inline
Pr n l s) >> sem Pr A s"
    shows "SP Pr n l s"
    <proof>
lemma assume_false_sem:
    "sem Pr A (Assume lfalse) = {}" \langleproof\rangle
lemma verif_exhale_exists_decompo:
    assumes "ver Pr {\varphi} (Exhale I)"
            and "wf_assert I"
        shows "\existsi r. i \in Inh I ^ Some \varphi = Some i }\oplus\mathrm{ Some r"
<proof>
lemma assume_not_bigger:
    assumes "ver Pr {\varphi} (While b I s)"
            and "wf_assert I"
        shows "{\varphi} >> sem Pr {\varphi} (Exhale I ; Havoc (modif s) ; Inhale I)"
<proof\rangle
```

```
lemma assume_not_bigger_real:
    assumes "ver Pr A (While b I s)"
        and "wf_assert I"
        shows "A >> sem Pr A (Exhale I ; Havoc (modif s) ; Inhale I)"
<proof\rangle
lemma add_h_comp_core:
    assumes "Some x = Some a }\oplus\mathrm{ Some b"
        and "set V\subseteq\sigmax"
        shows "Some x = Some a \oplus Some (h_comp b V) \oplus s_core x"
    \langleproof\rangle
lemma h_comp_set_add_two:
    assumes "Some x = Some a }\oplus\mathrm{ Some b"
    shows "{h_comp x V} = {h_comp a V} \oplus\oplus {h_comp b V}"
\langleproof\rangle
lemma h_comp_set_add:
    assumes "Some x = Some a }\oplus\mathrm{ Some b }\oplus\mathrm{ Some c }\oplus\mathrm{ Some d }\oplus\mathrm{ Some e"
    shows "{h_comp x V} = {h_comp a V} \oplus\oplus {h_comp b V} \oplus\oplus {h_comp c V}
\oplus\oplus{h_comp d V} \oplus\oplus {h_comp e V}" (is "?a = ?b")
\langleproof>
lemma h_comp_sum_set_smaller_pure:
    assumes "pure a"
            and "a << b"
        shows "{h_comp a V} \oplus\oplus {h_comp b V} = {h_comp b V}"
    \langleproof\rangle
lemma h_comp_smaller_elem:
    assumes "h_comp |a| V << x"
    shows "{h_comp |a| V} \oplus\oplus {h_comp |x| V} = {h_comp |x| V}"
    \langleproof\rangle
lemma core_four:
    assumes "Some a = Some b \oplus Some c \oplus Some d \oplus Some e"
    shows "s_core a = s_core b \oplus s_core c \oplus s_core d \oplus s_core e"
<proof>
lemma h_comp_set_add_four:
    assumes "Some x = Some a \oplus Some b \oplus Some c \oplus Some d"
    shows "{h_comp x V} = {h_comp a V} \oplus\oplus {h_comp b V} \oplus\oplus {h_comp c V}
\oplus\oplus{h_comp d V}" (is "?a = ?b")
<proof\rangle
lemma framing_set_singleton:
    assumes "\sigmar\cap set (modif s) = {}"
    and "ver Pr A s"
```

```
            and "framing Pr s"
    shows "sem Pr (A\oplus\oplus{r}) s >> sem Pr A s \oplus\oplus{r}"
\langleproof\rangle
lemma sigma_sem_havoc:
    assumes "ver Pr {a} (Havoc V)"
        and "x < sem Pr {a} (Havoc V)"
    shows "\sigmax=\sigma a\cup set V"
\langleproof\rangle
lemma bigger_and_ver_assume:
    assumes "ver Pr B (Assume b)"
        and "wfm Pr b"
        and "A >> B"
        shows "f A b >> f B b"
            and "f A (lnot b) >> f B (lnot b)"
    \langleproof\rangle
lemma wfm_f_and_sum:
    assumes "wfm Pr b"
        and "ver Pr A (Assume b)"
        shows "f (A \oplus\oplusD) b = f A b \oplus\oplus D" (is "?a = ?b")
<proof>
lemma not_empty_exhale:
    assumes "ver Pr {\varphi} (Exhale P)"
        and "wf_assert P"
    shows "sem Pr {\varphi} (Exhale P) \not= {}"
    \langleproof\rangle
lemma not_empty_havoc:
    assumes "ver Pr {\varphi} (Havoc V)"
    shows "sem Pr {\varphi} (Havoc V) \not= {}"
\langleproof\rangle
lemma not_empty_comp:
    assumes "\a. sem Pr {a} s1 f= {}"
        and "\b. sem Pr {b} s2 }\not={}
        and "ver Pr {c} (s1; s2)"
    shows "sem Pr {c} (s1 ; s2) }=={}
<proof>
lemma h_comp_add_single:
    assumes "{a} = {b} \oplus\oplus {c}"
    shows "{h_comp a x} = {h_comp b x} \oplus\oplus {h_comp c x}"
    <proof>
```


## A．4．2 Loop case

lemma loop＿inlining＿induct：
＂SP Pr $n l($ While $b I s) \wedge S P \operatorname{Pr} n l s \Longrightarrow S P \operatorname{Pr}(S u c n) l$（While $b$ I s）＂
〈proof〉

```
lemma sem_loop_set:
    assumes "ver Pr A (While b I s)"
; Inhale I ; Assume (lnot b))" (is "?a = ?b")
<proof\rangle
lemma wf_none_impossible:
    assumes "wf_stmt Pr (MethodCall y m x)"
        and "wf_program s"
        shows "get_method Pr m f= None"
    \langleproof\rangle
```

    shows "sem Pr A (While b I s) = sem Pr A (Exhale \(I\); Havoc (modif s)
    
## A．4．3 Soundness proof

```
lemma soundness_invariant:
    "SP Pr n l s"
<proof>
```

theorem soundness:
assumes "wf_program Pr"
and "wf_stmt Pr s"
and "SC Pr $n$ (modif s) s"
and "ver_program Pr"
and "ver $\operatorname{Pr}\{u\} s$ "
shows "ver $\operatorname{Pr}\{u\}$ (inline $\operatorname{Pr} n(m o d i f s) s) "$
〈proof〉
end
end

## A．Appendix：Isabelle Formalization

## A． 5 Renaming

```
theory Renaming
    imports Main
begin
fun sum :: "nat list => nat" where
    "sum [] = 1"
| "sum (t # q) = t + (sum q)"
lemma member_smaller:
    "x set l \Longrightarrowx\leqsum l + 1"
    \langleproof\rangle
```

type_synonym rename_t $=$ "nat list $\times$ nat list $\times$ nat list $\times$ nat list"
fun wf_renaming :: "rename_t $\Rightarrow$ bool" where
"wf_renaming (11, 12, 13, 14) $\longleftrightarrow ~(l e n g t h ~ 11 ~=~ l e n g t h ~ 12 ~ \wedge ~ d i s t i n c t ~$
11 ^ distinct 12)"
fun get_value :: "nat $\Rightarrow$ nat list $\Rightarrow$ nat list $\Rightarrow$ nat option" where
"get_value $x$ (ta \# qa) (tb \# qb) = (if $x=$ ta then Some tb else get_value
$x$ qa qb)"
| "get_value x _ _ = None"
lemma not_in_list_none:
" $x \notin$ set $a \Longrightarrow$ get_value $x$ a $b=$ None"
〈proof〉
lemma elem_get_value:
" $i<$ length $a \wedge$ distinct $a \wedge$ distinct $b \wedge$ length $a=$ length $b \Longrightarrow$ get_value
(a ! i) a $b=$ Some (b ! i)"
〈proof〉
definition sum_all :: "nat list $\Rightarrow$ nat list $\Rightarrow$ nat list $\Rightarrow$ nat list $\Rightarrow$ nat"
where
"sum_all abcd=sum $a+\operatorname{sum} b+\operatorname{sum} c+\operatorname{sum} d "$
lemma sum_greater_one:
"sum $1 \geq 1$ "
〈proof〉
lemma sum_all_commut:
"sum_all a b c d = sum_all b c d a"
〈proof〉
lemma x_greater_not_in:
assumes " $x \geq$ sum_all a b c $d$ "
shows " $x \notin$ set $a$ "
〈proof〉

```
lemma x_greater_not_in_all:
    assumes "x \geq sum_all a b c d"
    shows "x & set a \cup set b \cup set c \cup set d"
\langleproof\rangle
definition injective :: "(nat }=>\mathrm{ nat) # bool" where
    "injective f}\longleftrightarrow(\forallab.fa=fb\longrightarrowa=b)
definition surjective :: "(nat => nat) => bool" where
    "surjective f}\longleftrightarrow(\forally.\existsx. fx = y)"
definition bijective :: "(nat => nat) => bool" where
    "bijective f}\longleftrightarrow\mathrm{ injective f}^\mathrm{ surjective f"
definition id_above :: "(nat }=>\mathrm{ nat) }=>\mathrm{ bool" where
    "id_above f}\longleftrightarrow(\existsm.\forallx\geqm.fx=x)
definition same_lists :: "(nat => nat) => nat list }=>\mathrm{ nat list }=>\mathrm{ bool"
where
    "same_lists f a b \longleftrightarrow (\foralli < length a. f (a ! i) = b ! i)"
lemma easy_append_same_lists:
    assumes "same_lists f a b"
        and " }fx=y\mathrm{ "
        shows "same_lists f (x # a) (y # b)" (is "same_lists f ?a ?b")
<proof>
lemma induction_create:
    "length a = length b ^ distinct a ^ distinct b \Longrightarrow (\existsf. bijective
f ^ id_above f ^ same_lists f a b)"
<proof\rangle
fun get_f :: "nat list }=>\mathrm{ nat list }=>\mathrm{ (nat }=>\mathrm{ nat)" where
    "get_f a b = (SOME f. bijective f ^ id_above f ^ same_lists f a b)"
lemma get_f_works_well:
    assumes "length a = length b"
            and "distinct a"
            and "distinct b"
            and "f = get_f a b"
            shows "bijective f}^\mathrm{ id_above f}\wedge\mathrm{ same_lists fa b"
    \langleproof\rangle
fun first_rename :: "rename_t }=>\mathrm{ nat }=>\mathrm{ nat" where
    "first_rename (a, b, c, d) x = (let r = get_value x a b in
    if r = None then x + sum_all a b c d
    else the r)"
```

```
lemma first_rename_in_a:
    assumes "wf_renaming ( \(a, b, c, d\) )"
            and "i < length a"
    shows "first_rename ( \(a, b, c, d\) ) (a ! i) = b ! i"
    〈proof〉
lemma first_rename_in_d:
    assumes "wf_renaming ( \(a, b, c, d\) )"
        and \(" x \in \operatorname{set} d\) "
        and "x \(\neq\) set \(a "\)
        shows "first_rename ( \(a, b, c, d\) ) \(x=x+s u m \_a l l ~ a b c d "\)
    〈proof〉
fun distinctify :: "nat list \(\Rightarrow\) nat list" where
    "distinctify [] = []"
| "distinctify \((t \# q)=(i f t \in \operatorname{set} q\) then distinctify \(q\) else \(t\) \# distinctify
q)"
lemma same_elems_distinctify:
    "set 1 = set (distinctify 1 )"
    〈proof〉
lemma distinctify_is_distinct:
    "distinct (distinctify l)"
    〈proof〉
lemma f_rename_b:
    assumes "wf_renaming ( \(a, b, c, d\) )"
            and "domain = distinctify (a a d)"
            and "images = map (first_rename ( \(a, b, c, d\) ) domain"
            and " \(f=\) get_f domain images"
            shows "bijective \(f \wedge\) id_above \(f \wedge\) same_lists \(f\) domain images"
\(\langle p r o o f\rangle\)
fun rename_b :: "nat \(\Rightarrow\) rename_t \(\Rightarrow\) nat" where
```



```
    get_f domain (map (first_rename ( \(a, b, c, d\) )) domain) x)"
lemma rename_b_in_a:
    assumes "wf_renaming \((a, b, c, d)\) "
        and "i < length a"
    shows "rename_b (a ! i) ( \(a, b, c, d\) ) \(=b\) ! i"
〈proof〉
lemma rename_b_in_d:
    assumes "wf_renaming ( \(a, b, c, d\) )"
            and " \(x \in \operatorname{set} d\) - set \(a\) "
            shows "rename_b \(x(a, b, c, d)=x+s u m \_a l l a b c d "\)
〈proof〉
```

```
fun gen_list :: "nat }=>\mathrm{ nat list" where
    "gen_list 0 = []"
    | "gen_list (Suc n) = gen_list n @ [n]"
lemma gen_list_length:
    "length (gen_list n) = n"
    <proof>
lemma gen_list_def:
    "\i. i < n \Longrightarrow gen_list n ! i = i"
    \langleproof\rangle
lemma gen_list_member:
    "x < set (gen_list n) \longleftrightarrow x < n"
    \langleproof\rangle
lemma distinct_gen_list:
    "distinct (gen_list n)"
    <proof>
fun invert_f :: "(nat }=>\mathrm{ nat) }=>\mathrm{ (nat }=>\mathrm{ nat)" where
    "invert_f f = (\lambdax. THE y. f y = x)"
lemma the_y_works:
    assumes "bijective f"
    shows "f (THE y. f y = x) = x"
<proof>
lemma bijective_then_works:
    assumes "bijective f"
    shows "invert_f f (fx) = x"
    \langleproof>
lemma bijective_inverse_other_dir:
    assumes "bijective f"
    shows "f (invert_f fx
    \langleproof\rangle
lemma bijective_inv:
    assumes "bijective f"
    shows "bijective (invert_f f)"
    \langleproof>
fun rename_inv_b :: "rename_t }=>\mathrm{ rename_t" where
    "rename_inv_b (a, b, c, d) = (let domain = distinctify (a @ d) in
    let f = get_f domain (map (first_rename (a, b, c, d)) domain) in
    let m = (SOME m. \forallx \geqm. fx = x) in
    let ante = gen_list m in
```

```
    (ante, map (invert_f f) ante, [], ante))"
lemma wf_renaming_rename_inv_b:
    assumes "wf_renaming (a, b, c, d)"
    shows "wf_renaming (rename_inv_b (a, b, c, d))"
\langleproof\rangle
fun rename_a :: "nat }=>\mathrm{ rename_t }=>\mathrm{ nat" where
    "rename_a x (a, b, c, d) = (let r = get_value x a b in
    if r = None then x else the r)"
fun rename_elem :: "nat }=>\mathrm{ rename_t }=>\mathrm{ nat" where
    "rename_elem x (a,b,c,d) = (if c = [] ^ set a = set b then rename_a
x (a,b, c, d)
    else rename_b x (a, b, c, d))"
lemma gen_list_set:
    "set (gen_list m) = {i. i < m}"
<proof>
lemma rename_elem_rename_inv_b:
    assumes "wf_renaming t"
    shows "rename_elem x (rename_inv_b t) = rename_a x (rename_inv_b t)"
<proof>
lemma id_above_then_m_works:
    assumes "id_above f"
            and "m = (SOME m.\forallx\geqm. f x = x)"
            shows "}\forallx\geqm.fx=x
<proof\rangle
lemma rename_inv_b_is_invert_f:
    assumes "wf_renaming (a, b, c, d)"
            and "domain = distinctify (a a d)"
            and "images = map (first_rename (a, b, c, d)) domain"
            and "f = get_f domain images"
            shows "rename_a x (rename_inv_b (a, b, c, d)) = invert_f f x"
<proof\rangle
lemma rename_inv_b_is_inv:
    assumes "wf_renaming ( }a,b,c,d)
    shows "rename_elem (rename_b x (a, b, c, d)) (rename_inv_b (a, b, c,
d)) = x"
<proof>
fun rename_inv_a :: "rename_t }=>\mathrm{ rename_t" where
    "rename_inv_a (a, b, c, d) = (b, a, [], map (\lambdaelem. rename_elem elem
(a, b, c, d)) d)"
```

```
lemma wf_rename_inv_b:
    assumes "wf_renaming ( }a,b,c,d)
    shows "wf_renaming (rename_inv_a (a, b, c, d))"
    \langleproof\rangle
fun rename_inv :: "rename_t m rename_t" where
    "rename_inv (a, b, c, d) = (if c = [] ^ set a = set b then rename_inv_a
(a,b, c, d)
    else rename_inv_b (a, b, c, d))"
lemma wf_rename_inv:
    assumes "wf_renaming t"
    shows "wf_renaming (rename_inv t)"
    <proof>
lemma rename_inv_a_is_inv:
    assumes "wf_renaming (a, b, [], d)" (is "wf_renaming ?t")
        and "set a = set b"
    shows "rename_elem (rename_a x ?t) (rename_inv_a ?t) = x"
<proof\rangle
lemma rename_removes_vars_b:
    assumes "wf_renaming ( }a,b,c,d)
        and "var \in set d"
        and "rename_b var ( }a,b,c,d)\in\operatorname{set a \cup set c"
        shows "rename_b var ( }a,b,c,d)\in\operatorname{set b"
<proof\rangle
lemma rename_inv_works:
    assumes "wf_renaming t"
    shows "rename_elem (rename_elem x t) (rename_inv t) = x"
    \langleproof>
lemma rename_removes_vars:
    assumes "wf_renaming ( }a,b,c,d)
            and "var \in set d"
            and "rename_elem var (a, b, c, d) \in set a \cup set c"
            shows "rename_elem var (a, b, c, d) \in set b"
<proof>
lemma rename_elem_in_a_b:
    assumes "wf_renaming (a, b, c, d)"
            and "i < length a"
            shows "rename_elem (a ! i) (a, b, c, d) = b ! i"
<proof>
end
```


## EH

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):
BEYOND THE FRAME RULE: STATIC INLINING IN SEPARATION LOGIC

Authored by (in block letters):
For papers written by groups the names of all authors are required.
Name (s)
First name (s):
DARDINIER
THIBAULT

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

MIONS (FRANCE), 07.04.2020

$\qquad$
$\qquad$

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


[^0]:    ${ }^{1}$ More precisely, checking if an assertion is actually an invariant for a loop is decidable if the underlying theory is decidable.
    ${ }^{2}$ It is decidable if the underlying theory is decidable.
    ${ }^{3}$ This verification only guarantees partial correctness, namely the property that the program is correct when it terminates.

[^1]:    ${ }^{4}$ That is, verification after small modifications of a first verified version.

[^2]:    ${ }^{1}$ We describe here the intuitionistic version of separation logic, and not the classical one.

[^3]:    ${ }^{2}$ Even if loops and method calls are encoded using havoc in Viper, havoc is not a Viper statement.

[^4]:    ${ }^{3} P[x / \arg ]$ substitutes $\arg$ by $x$ in $P$.

[^5]:    ${ }^{1}$ This order relation on sets of states is not antisymmetric. As an example, take $\varphi$ a state with full permission to $x . f$, and $\varphi^{\prime}$ a state with half permission to $x . f$. Then $\left\{\varphi, \varphi^{\prime}\right\}$ is both stronger and weaker than $\left\{\varphi^{\prime}\right\}$.

[^6]:    ${ }^{1}$ The $\Sigma$ symbol refers to a sum using $\oplus$.

[^7]:    ${ }^{1}$ The set of states models the non-determinism of the semantics, while the error signifies a failure (and ignores states which may not fail).

[^8]:    ${ }^{1}$ The function preserves validity for stronger states.
    ${ }^{2}$ The function returns a stronger (or equally strong) set of output states for a set of stronger input states.

[^9]:    ${ }^{3}$ The sum of a set of states and a singleton $\{r\}$ does not exactly results in the set where all states have been added to $r$. States which cannot be added to $r$ are simply removed.

