Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Semantics for Predicates in Automated Separation Logic Verifiers

Master Thesis

Yushuo Xiao

21st October, 2024

Advisors: Thibault Dardinier, Gaurav Parthasarathy, and Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Contents

Chapter 1

Introduction

Automated program verifiers (or program verifiers in short) are tools that take as input a program, a formal specification, and potential annotations, and try to automatically establish that the program satisfies the specification, or point out the part of the program that may have an error. Many program verifiers are based on *separation logic* to reason about heap-manipulating programs. Separation logic introduces a notion of *resources* that capture ownership of parts of the heap. This enables modular reasoning about heap-manipulating programs. In separation logic, *abstract predicates* (*predicates* in short) are widely used to abstract over resources and logical constraints. Many practical verifiers also support them. Predicates can be used to write modular, readable, and maintainable specifications, and are especially useful for specifying recursive data structures such as linked lists and trees.

Predicates are usually handled differently in theory and in verifier implementations. In separation logic theory, a predicate instance represents the complete unrolling of its definition (i.e. corresponds to the least-fixed point). This is the *equirecursive* interpretation of a predicate. However, it is infeasible for program verifiers to directly implement such an equirecursive interpretation when predicates become unbounded, because it is in general not possible to fully unroll a predicate definition statically (for example, when predicates specify recursive data structures that have statically unbounded recursion depth). Therefore, practical program verifiers differentiate a predicate instance and its corresponding body, and make use of user-provided *fold* and *unfold* operations to manipulate predicate instances explicitly. Informally, a fold operation rolls up the body of a predicate into a predicate instance, and an unfold operation reverses this process, unrolling the predicate body. This is the *isorecursive* interpretation of predicates.

Viper [1] is an intermediate verification language with accompanying program verifiers. Viper supports predicates as separation logic specifications, and provides fold and unfold operations for users to manipulate predicate instances explicitly. However, while Viper generally differentiates a predicate instance from its body as in an isorecursive interpretation, the isorecursive interpretation cannot capture the way Viper's verifiers handle predicates fully. The reason is that the verifiers may potentially look into the body of a predicate to gain more information without fully unrolling predicate definitions to enable more powerful reasoning. On the other hand, the equirecursive interpretation does not overapproximate all of Viper's features either, because there are features that fundamentally require some sort of distinction between predicates and their bodies. To the best of our knowledge, no existing formalization of Viper's semantics fully explains the behavior of predicates while supporting all other features of the language.

In this work, we close the gap by defining a novel semantics for Viper, which is based on the isorecursive model but has an equirecursive flavor. This approach provides a more precise understanding of how Viper manages predicates, addressing limitations in existing formalizations.

Our technical contributions can be summarized below.

- We propose and formalize a novel semantics for Viper. To our knowledge, this is the first semantics that is able to precisely explain Viper predicates in combination with all of Viper's features. The semantics is fully mechanized in Isabelle/HOL.
- We prove some critical properties on the new semantics, which establish connections between various components of the semantics and enhance our confidence in its correctness. Additionally, these properties form a basis for validating Viper verifier implementations (see the next contribution). Some of these properties are mechanized in Isabelle/HOL.
- We conduct some preliminary experiments on the validation of a Viper verifier (specifically, the Viper verifier based on verification condition generation [2]). These experiments explore the soundness of the verifier: if a Viper program is verified by the verifier, the program is also correct according to

our semantics. This demonstrates that the semantics we define can serve as a foundation for validating Viper verifier implementations.

The thesis is organized as follows. In Chapter 2, we provide a brief introduction to the Viper language, focusing on the aspects relevant to this thesis. In Chapter 3, we show why neither the equirecursive nor the isorecursive interpretation of predicates can capture all Viper programs. In Chapter 4 and 5, we formally develop our new semantics, starting with our novel state model and then defining the operational semantics. Chapter 6 shows the properties of our semantics and documents the experiments we did on validating Viper's verification condition generation verifier. Chapter 7 concludes the thesis and points out future directions.

Chapter 2

An Introduction to Viper

Viper [1] is a verification infrastructure that simplifies the development of program verifiers. It comes with an *intermediate verification language*, the Viper intermediate language. Viper has strong support for permission logics such as separation logic and implicit dynamic frames. It supports permissions natively and uses them to express ownership of heap locations. The language is a simple sequential, object-based, imperative programming language and allows users to write specifications for programs. Viper has two *back-ends*, which are program verifiers that check the absence of errors in a Viper program and ensure the specifications are met. It also enables simpler development of verifiers for real programming languages by implementing *front-ends*, which translate programs and specifications written in other languages into Viper and let Viper check the correctness of these programs. Such verifiers include Prusti [3] for Rust, Gobra [4] for Go, Nagini [5] for Python, and Hypra [6] for Hyper Hoare Logic, etc.

Since the thesis is about designing a formal semantics for Viper, in this chapter we give a brief introduction to the Viper intermediate language, putting emphasis on important features that are relevant to this thesis.

In Figure 2.1, we list the syntax of the Viper subset we are concerned with in this thesis. A real Viper program consists of global declarations and method definitions. A method includes a statement as its body. However, this work does not consider method calls explicitly[1], so we omit them in the presented Viper syntax, and only

[1] Method calls can be desugared into inhale and exhale.

Expression

$e ::= x \mid \mathsf{lit}(v) \mid e.f \mid e \ \mathsf{bop}\ e \mid \mathsf{uop}\ e \mid \texttt{perm}(e.f) \mid \texttt{perm}(P(\vec{e}))$

Assertion

$A ::= \texttt{acc}(e.f, e) \mid \texttt{acc}(e.f, \texttt{wildcard}) \mid \texttt{acc}(P(\vec{e}), e) \mid \texttt{acc}(P(\vec{e}), \texttt{wildcard}) \mid$
$\quad e \mid A * A \mid e \Rightarrow A \mid e \ ?\ A : A$

Statement

$s ::= s; s \mid x := e \mid e.f := v \mid \texttt{var}\ x : \tau \mid \texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\} \mid \texttt{inhale}\ A \mid \texttt{exhale}\ A \mid$
$\quad \texttt{fold}\ \texttt{acc}(P(\vec{e}), e) \mid \texttt{fold}\ \texttt{acc}(P(\vec{e}), \texttt{wildcard})$
$\quad \texttt{unfold}\ \texttt{acc}(P(\vec{e}), e) \mid \texttt{unfold}\ \texttt{acc}(P(\vec{e}), \texttt{wildcard})$

Field Declaration

$F_{\text{decl}} ::= \texttt{field}\ f : \tau$

Predicate Declaration

$P_{\text{decl}} ::= \texttt{predicate}\ P(\overline{x : \tau})\ \{\ A\ \}$

Program

$Prog ::= \overline{F_{\text{decl}}}\ \overline{P_{\text{decl}}}\ s$

Figure 2.1: Syntax of a Viper subset relevant to this thesis.

include statements that represent the only executable part of the Viper program (one can see it as the body of the only method in the program). The global declarations include fields and predicates. Field declarations describe the structure of an object. There is only one type of object, i.e. Viper does not support defining multiple classes. Only objects are stored in the heap. Predicates will be introduced later in this chapter.

Viper is based on a flavor of separation logic called *implicit dynamic frame* (IDF) [7], which provides native support for permission-based reasoning. Permissions are used to describe ownership of heap locations. Viper supports *fractional permissions* [8, 9]. Each heap location has a fractional permission amount ranged between `0` and `1`. A program must hold *full permission* (permission amount of `1`) to a heap location to modify its value, and must hold non-zero permission to read its value. The permissions to heap locations can be explicitly manipulated by statements in a Viper program.

Viper assertions can describe permissions. For example, the accessibility predicate `acc(e.f,p)` stands for `p` amount of permission to the field `e.f`, where `e` evaluates to a reference which points to an object in the heap. Viper assertions can also describe logical (value) constraints. In IDF, value constraints are expressed via *heap-dependent* expressions such as `x.f == 1`, whose evaluation is partial. In an assertion, such constraint is usually complemented by an accessibility predicate to provide permission to read the value, such as `acc(x.f,1) && x.f == 1`. This assertion is equivalent to the “point-to” assertion `x.f` $\mapsto 1$ in separation logic.

The statements **`inhale`** A and **`exhale`** A are used to explicitly manipulate permissions, where A is an assertion. The statement **`inhale`** A adds the permissions specified in A and assumes the logical constraints in A. The statement **`exhale`** A removes the permissions specified in A, and fails if there are not sufficient permissions in the state or a logical constraint does not hold. After this step, for any heap location to which all permission was removed, **`exhale`** also forgets the value stored on that location in the heap. This models the fact that other parts of the code might hold full permission to it and change its value.

Predicates are used in separation logic to abstract over resources and constraints. Viper allows users to write recursively-defined predicates. For example, we can define a classic predicate for a linked list.

```
predicate List(x: Ref) {
    x != null ==> acc(x.next) && (x.value) && List(x.next)
}
```

In this example, `next` and `value` are two fields defined in global declarations, representing the next element in the linked list and the value stored in the current node, respectively.

Predicates are also a type of resource that can be inhaled or exhaled. In Viper, a predicate instance is treated differently as its body. Users need to provide explicit ghost operations `fold` and `unfold` to roll up or unroll predicate definitions. Predicates also support fractional permissions [10]. Viper back-ends treat fractional predicates with *syntactic multiplication*, which multiplies every accessibility predicate with the fraction on the syntax level. They are discussed in detail and formally defined in Section 4.4.1.

Both normal heap locations and predicates support *wildcard permission*. A wildcard permission is an unspecified positive amount of permission. It is often used when users want read permission to some heap location but do not care about the precise permission amount.

Chapter 3

The Need for a New Semantics

Formalizing the Viper semantics has been an ongoing effort in the research community [11, 12, 13]. A correct formalization of the semantics gives the precise meaning of a Viper program, helps understand the language better, and make it possible to reason formally about front-end and back-end implementations.

Roughly speaking, two fundamental ideas of how to define the semantics have already been proposed, namely the *equirecursive* semantics and the *isorecursive* semantics. They differ in the way they treat predicates.

- **Equirecursive semantics**. This is the usual way of interpreting a predicate in separation logic. The meaning of a predicate is seen as the complete unrolling of its definition. To deal with recursive predicates, we use the *least fixed point* interpretation of a predicate. In equirecursive semantics, the state model does not track the amount of permission to each predicate, and inhaling a predicate (adding a predicate instance resource to the state) directly adds the predicate body to the state. Therefore, the effects of `fold` and `unfold` operations are completely nullified.

- **Isorecursive semantics**. The isorecursive view of predicates distinguishes predicate instances from their definitions. Users need to provide ghost `fold` and `unfold` statements to explicitly roll up or unroll a predicate definition. This is the typical way most Viper users view predicates, and it is also how back-ends handle them.

Both semantics have their strengths, but also come with fundamental weaknesses. Equirecursive semantics does not support the full set of Viper features, in particular permission introspection, and its limitations are inherent due to the equirecursive nature. On the other hand, purely isorecursive semantics cannot account for all back-end behaviors, as it is too weak to verify some correct assertions.

In the following two sections, we explain the problems with isorecursive and equirecursive semantics in detail, respectively. These problems are then solved using a novel semantics in the next chapter.

3.1 Purely Isorecursive Semantics

In this section, we discuss a *purely isorecursive semantics*[1] and its limitations.

The key idea of isorecursive semantics is treating predicates as first-class resources, just like normal heap locations. We need to keep track of predicates as a separate resource type rather than the unrolling of their bodies. In isorecursive semantics, the amount of permission of a predicate is tracked in the state, and fold and unfold manipulate the state directly. Concretely, the semantics maintains two masks, the *heap mask* and the *predicate mask*. These masks store the direct permissions to heap locations[2] and predicate instances the state holds. When a predicate instance is unfolded, it is removed from the predicate mask and the permissions specified in its body are added to the heap mask.

This semantics evokes some important questions to answer. Consider the program pair in Figure 3.1. The program on the left inhales full permission to `x.f` twice, and the program on the right defines a predicate containing one permission to `x.f` and inhales a full permission to `x.f` directly and then an instance of `P(x)`. First, consider the program on the left. Since the permission amount of each heap location cannot exceed 1, the state after two inhale statements should not be considered valid. If we allowed this state, we could use one full permission to change the value of `x.f`, and at the same time be able to assert that the value of `x.f` had not changed, since we kept the other full permission (the heap information `x.f` is *framed* around the

[1] We call the semantics purely isorecursive semantics because it can be improved to avoid some limitations of the purely isorecursive semantics to some degree. However, it is still not an ideal semantics. We document the attempt in Appendix A.

[2] A heap location is like $x.f$, where x is a reference and f is a field.

```
field f: Int


method m(x: Ref) {
    ∅
    inhale acc(x.f)
    {acc(x.f)}
    inhale acc(x.f)
    {acc(x.f, 2)} → inconsistent
}
```

```
field f: Int
predicate P(x: Ref) { acc(x.f) }

method m(x: Ref) {
    ∅
    inhale acc(x.f)
    {acc(x.f)}
    inhale P(x)
    {acc(x.f), P(x)}
    unfold P(x)
    {acc(x.f, 2)} → inconsistent
}
```

Figure 3.1: A program pair in isorecursive semantics. After inhaling `acc(x.f)` (same in both programs), the left one inhales `acc(x.f)` directly, and the right one inhales the same permission via `P(x)`.

assignment, in separation logic term). This means we have reached an *inconsistent* state, and in theory any assertion can be proven. In practice, Viper back-ends detect this inconsistency. This means, that to be able to capture this back-end behavior, the semantics must also detect the inconsistency. In the left program, the inconsistency can be easily detected since the semantics already sees that there is more than full permission to `x.f` in the program state.

However, in the program on the right, the question is more subtle. Since the permission to `x.f` is wrapped inside the predicate `P(x)`, it is not possible to detect the inconsistency without fully unfolding the predicates. But the question is, do back-ends detect the inconsistency? If a back-end does not detect it, the semantics has the option to ignore the inconsistency and continue the execution until the predicate is unfolded and two full permissions of `x.f` are exposed. More generally, if back-ends never look into predicates except in the case of statements that require doing so (such as fold and fold), then one could easily conceive a naive "purely" isorecursive semantics. This semantics only discovers inconsistency based on the heap mask. In other words, it only checks direct permissions to heap locations for consistency. For the inhale-unfold sequence shown in Figure 3.1, the execution succeeds after the two inhale statements, and detects the inconsistency only after the unfold.

```
field f: Int

predicate P(x: Ref) { acc(x.f) }

function getVal(x: Ref)
    requires P(x)
{
    unfolding P(x) in x.f
}

method m(x: Ref)
    requires P(x) && acc(x.f)
    ensures false
{
    x.f := x.f + 1
    assert getVal(x) == old(getVal(x))
    assert getVal(x) == old(getVal(x)) + 1
}
```

Figure 3.2: A Viper program showing that the inconsistency exists in the back-end implementation even when the predicate is not unfolded.

The discussion above is hypothetical. In fact, the Viper program in Figure 3.2 shows that the purely isorecursive semantics is not powerful enough to capture the behavior of Viper back-end implementations, in particular the verification condition generation (VCG) back-end.

Figure 3.2 shows a simple Viper program with functions.[3] The program declares one field `f`, a predicate `P` which only contains full permission to `x.f`, and a function `getVal` that requires the predicate instance `P(x)` as precondition and returns the value of `x.f` by temporarily unfolding the predicate. Functions in Viper have no side effects. The function only looks into the predicate to read its value without changing the state. In the method `m`, it requires both full permission to `x.f` and a predicate instance `P(x)`. Here, the state is already inconsistent from the equirecursive point of view. In the body of the method, the value of the heap location `x.f` is incremented by 1. Then, we assert two things: (1) the current value of `x.f` is the

[3] Functions are in general not considered in the work, but provide a motivation for detecting consistency.

same as the value before the assignment (`old` expressions return the value of the expression at the beginning of the method); (2) the current value of `x.f` is larger than the value before the assignment by 1. From the equirecursive point of view, the pre-condition is already inconsistent since it contains two full permissions of `x.f`. As a result, an equirecursive semantics could verify both assertions plus `false` as the post-condition.

In the VCG back-end which handles predicates isorecursively, the inconsistency is still detected because of the existence of the function `getVal`. In Viper, function bodies can only refer to the values of heap locations whose permissions are included in the pre-condition of the function. Therefore, the value of a function only depends on its pre-condition. The VCG back-end encodes this property as an axiom: if the values of the heap locations specified by the pre-condition do not change, the value of the function does not change, either. In case the precondition of the function is a predicate, as in the example in Figure 3.2 where the precondition of the function `getVal` is `P(x)`, the VCG back-end uses the *snapshot* of the predicate as the representation of the "value" of the predicate. The snapshot of a predicate is all the values included in the predicate organized in an unstructured way. Snapshots are updated when changes are made to the predicate, e.g., when the predicate is folded, inhaled, etc. In method `m`, `P(x)` is used as a pre-condition. In this case, the concrete values of the heap locations included in `P(x)` are unknown to the verifier, and the VCG encoding just uses some unconstrained value as the snapshot of `P(x)`. The important point is, the assignment `x.f := x.f + 1` does not affect the snapshot of `P(x)`, so the VCG back-end is able to assert that the value of `getVal(x)` is the same as that at the beginning of the method. On the other hand, the VCG back-end can also detect that the value of `x.f` has changed by the assignment, so the second assertion is also verified. The successful verification of these two assertions implies that the state is inconsistent and the post-condition can in turn be proven.

Even if we remove the two assertions in the Viper program, the verification conditions generated by the back-end are still satisfied.[4] This suggests that the semantic must be powerful enough to detect the inconsistency after the assignment, in order to justify the back-end behavior. One way of achieving this is by pruning states that have at most 1 permission after unfolding all predicates. A naive, but unsuc-

[4] However, the program does not verify in practice because of SMT solver limitations.

cessful attempt of doing so is documented in Appendix A. The better solution is our main contribution of the thesis, described in Chapter 4 and 5.

Before we start the development of our new semantics, we first briefly discuss the equirecursive semantics in the remainder of this chapter. The equirecursive semantics detects inconsistencies naturally, but has other fundamental limitations.

3.2 Equirecursive Semantics

In this section, we explain equirecursive semantics on a high level. The semantics is different from the purely isorecursive semantics described in the last section in the sense that it treats predicates in an equirecursive way—instead of explicitly storing the permission for predicates in a separate predicate permission mask, the semantics always expands the predicate fully and stores the permissions to heap locations that the predicate guards directly in the state. More concretely, equirecursive semantics interprets predicates using their *least fixed points*, and uses the interpretation whenever a predicate instance is specified in the program or specification. For example, when inhaling a predicate instance, the semantics constructs the least fixed point and adds the direct permissions to heap locations to the state.

The notion of state consistency is natural to equirecursive semantics—it only has one heap mask, and consistency can be defined as "the permission to each heap location does not exceed 1". For example, in the program from Figure 3.2, the equirecursive semantics detects the inconsistency after inhaling the pre-condition.[5]

```
∅
inhale P(x) && acc(x.f)
{x.f, x.f}  (too much permission to x.f, inconsistent)
```

The pre-condition `P(x)` is directly replaced with its least fixed point, in this case the body of `P(x)`, `acc(x.f)`. Therefore, the pre-condition is equivalent to two full permissions to `x.f`. Equirecursive semantics can detect the inconsistency by just looking at the heap mask, and prune the inconsistency state.

However, it is not a perfect, well-rounded semantics that does everything because of the following reasons.

[5] In Viper, the semantics for methods is inhaling pre-conditions to an empty state, and then executing the body.

Equirecursive semantics does not support all Viper features fundamentally. Viper supports *permission introspection*, which allows users to explicitly query the permission held by the current state. Permission introspection of a normal heap location should only return the direct permission of the location (excluding any permission folded in a predicate). Besides, permission introspection allows users to explicitly query the amount of permission of a predicate stored in the current state. For example, back-ends verify the assertion in the following Viper program.

```
predicate P(x: Ref) { acc(x.f, 1/2) }

method m(x: Ref) {
    inhale P(x) && acc(x.f, 1/2)
    assert perm(P(x)) == 1  // verifies
}
```

To reason about the value of `perm(...)`, the semantics must be able to preserve the information about predicate instances in its state. However, in equirecursive semantics, the state only knows it owns full permission to `x.f`, but is not able to tell if the permission is from a direct permission or a predicate, let alone the permission amount of `P(x)`. To this end, equirecursive semantics has a fundamental flaw because it loses the information about predicates in an irreversible way.

This shortcoming of the equirecursive semantics implies that we have to at least modify upon the semantics to capture all Viper features, and in turn formally connect Viper to front-ends and back-ends.

Back-end verification implementations treat predicate isorecursively. The back-end implementations of separation logic verifiers with predicates usually have an isorecursive flavor, since static verification cannot reason about unbounded recursive predicates efficiently. This requires the users to manually fold and unfold predicates via statements where necessary. An equirecursive semantics might not correspond to back-end implementations closely and connecting them directly poses a significant challenge.

To demonstrate why an isorecursive-like semantics is preferred when connecting Viper to a back-end implementation, consider the following Viper code snippet.

```
field f: Int
predicate P(x: Ref) { acc(x.f) }
```

```
method m(x: Ref) {
    inhale P(x)
    unfold P(x)
}
```

In the verification condition generation back-end, the unfold is encoded as looking up the permission amount in the predicate mask and deciding if it has at least 1 permission. However, if we choose the equirecursive semantics, we lose the information about predicate instances irreversibly. As a result, the connection between the back-end implementation and the semantics cannot be easily made.

The discussion in this chapter shows that both equirecursive and purely isorecursive ways of designing the Viper semantics do not work because of their own reasons. This motivates us to formalize another semantics in order to capture all Viper features and at the same time make connection to back-end implementations straightforward. Below, we characterize three properties that we would like the semantics to have.

- **Captures all back-end behaviors on predicates**. This means if a back-end can verify a program, the semantics must also be able to prove its correctness (except for those that arise from implementation errors). Otherwise, the semantics is not powerful enough and does not qualify to justify back-end implementations.
- **Supports (or has the potential to support) all Viper features**. Ideally, all Viper features will eventually be formalized within the semantics. To achieve this, the proposed semantics must not have inherent limitations that prevent the support of certain features.
- **Reflects how people understand a Viper program**. In Viper, predicates are a type of resource that is treated the same as regular permissions to heap locations. Users need to explicitly fold and unfold predicates to create predicate instances or obtain their contents. This is not only how back-ends treat predicate, but also the mental model of Viper users. Our new semantics should reflect this characteristic of Viper.

- **Easy to connect to both back-ends and front-ends**. One important goal of defining the Viper semantics is to establish an end-to-end soundness guarantee between a front-end and a back-end. Therefore, we are looking for a semantics that makes this connection feasible.

In the next two chapters, we will propose a novel semantics that stands between the isorecursive and the equirecursive semantics, which preserves as much information as possible in the state model and facilitates the validation of back-end implementations.

Chapter 4

A New State Model

This chapter includes the core contribution of this work—a novel state model for predicates which will be used to define a new Viper semantics. The state model borrows ideas from both the equirecursive and the isorecursive state model and combines the strength of both semantics discussed in the previous chapter, while avoiding their limitations. This chapter starts with a high level key idea of the new state model, and then develops the semantics in a more formal manner.

4.1 Key Idea

We build our new state model based on the purely isorecursive semantics because it has more similarities.

As we pointed out, a purely isorecursive semantics without consistency is incorrect because it does not reflect how back-end implementations handle predicates, and defining consistency with the naive approach is cumbersome to work with. On the other hand, defining consistency in equirecursive semantics is straightforward, but we cannot use an equirecursive state model due to the irreversible loss of predicate information. The main idea of the new state model is, keeping the isorecursive state model, but adding more information for predicates so we can obtain the the same permission information as in the equirecursive state while still preserving the predicate structures.

We use a recursive structure called *nested mask* to achieve this. The nested mask contains a heap mask and a predicate mask, which stores the same information as the ones introduced for the purely isorecursive semantics. Additionally, to store the predicate structures in the state model, for each predicate the state owns, it also stores another nested mask that contains all permission information folded inside the predicate. In short, a nested mask plays two roles: it represents the top-level state—permissions to all heap locations and predicates owned by the state; it represents the concrete permissions inside an individual predicate in a structural and recursive way. It will be made clear that the new state model is a refinement of both the equirecursive state (in the sense that it preserves all information stored in the equirecursive state but has a more refined data structure to organize this information structurally) and the isorecursive state (in the sense that one isorecursive state could have multiple representations in the nested mask model).

In the new state model, defining state consistency is straightforward, and keeping the predicates structurally makes it easy to connect to an isorecursive implementation of Viper back-ends. Next, we develop the semantics formally, starting with the definition of the nested mask.

4.2 Nested Mask

The Viper language gives users the ability to manipulate permissions of heap locations explicitly. As a result, we need to keep track of a heap (value information) as well as its permission information in the program state. In our semantics, the component that stores the permission information is a *nested mask*.

A nested mask is a tree-like inductive data structure, which has the following type.

$$\mathsf{NestedMask} := \underbrace{(L \rightharpoonup \mathbb{R}^+)}_{\text{heap mask}} \times \underbrace{(P \rightharpoonup \mathbb{R}^+ \times \mathsf{NestedMask})}_{\text{predicate mask and nested masks}}$$

In the definition, $\mathbb{R}^+$ denotes the set of (strictly) positive real numbers. The nested mask has two components, which store the permissions to heap locations and predicates, respectively. The first component is the heap mask. It is a partial function[1] that only stores the information of direct heap permission. In other words, if we inhale a predicate instance, this component does not change. If a heap location has

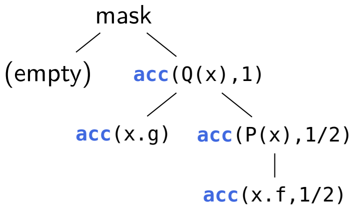
[1] Here, instead of restricting the permission to be no more than 1, we allow any positive real number to be the permission amount of a heap location, but having more than 1 permission is still not allowed in the final semantics and is rather an intermediate step. It will be explained in Section 4.4.2.

zero permission, the partial function is undefined at that location. Otherwise, it is defined to be the permission owned by the state. The other component only stores information about predicates. It is a partial function from predicate locations[2] to a pair of positive real numbers and a (recursive) nested mask. If a predicate instance $p \in P$ is present in the state, then the partial function maps p to its permission amount and a nested mask representing the predicate. This permission has no upper bound because we can have an unbounded number of the same predicate, unlike normal heap locations. If we omit the nested mask in this partial function, the second component is just the same as the predicate mask in the purely isorecursive semantics described in Section 3.1.

The key in our new state model is a nested mask associated with each predicate present in the state. In addition to the normal permission amount of a predicate instance, we also store a nested mask, representing all the permissions (heap locations and nested predicates) in this predicate. This is better understood with an example. Suppose we have the following predicate definitions.

```
field f: Int
field g: Int
predicate P(x: Ref) { acc(x.f) }
predicate Q(x: Ref) { acc(x.g) && acc(P(x), 1/2) }
```

Now, starting from the empty state (a state without any permission), we try to inhale `Q(x)`. We show the mask after the inhale directly below.



We will use some conventions when representing a nested mask using a tree. Each tree or subtree (excluding leaves) is a nested mask. The entire tree starting from

[2] Here, we do not define a concrete set of predicate locations, and parameterize it using a set P instead. In the concrete Viper semantics, one could imagine P as a pair of predicate name and value list (represents the arguments of the predicate).

the root represents the complete mask that corresponds to the Viper state. For each node, the first (leftmost) branch is the heap mask. Each other branch represents a predicate instance.[3] We use `acc`$(P(\vec{a}), q)$ to represent the predicate instance with its permission amount, where $P(\vec{a})$ is the predicate name with its evaluated argument list, and q is the permission amount owned by the state. For example, `acc(Q(x),1)` means the state owns permission amount `1` to the predicate instance `Q(x)`. For each branch that represents a predicate instance, it must have children to represent the nested mask associated with the predicate instance (see the type of the nested mask).

Let us return to the example and see the state after the inhale. Since we have inhaled a predicate instance (`Q(x)`), the direct heap mask does not change and is still the empty mask (the left branch of the root). The mask for predicates should be the partial function that maps `P(x)` to `1` and all other locations to `0`.

The subtree with the root `acc(Q(x),1)` should represent the permissions in this predicate instance. To build this state, we obtain each heap location and predicate in the body of `Q(x)`. In this example, there is one heap permission access predicate `acc(x.g)` and another predicate `P(x)`. These two permissions belong to the heap mask and the predicate mask, respectively.

The final layer of the nested mask is built by unfolding the definition of `P(x)` and multiplying each permission inside by $\frac{1}{2}$ syntactically. This results in a heap mask with `acc(x.f,1/2)` and an empty predicate heap. Since there is no predicate instance at this layer, the nested mask does not expand further.

This concludes the example of constructing the nested mask for a simple predicate. The example shows the most important feature of the nested mask state model: it contains the full information of the permission held within a predicate along with its nested structure. Its benefits will become clear later when we develop the semantics. Note that the construction is done in an informal manner. How to build a nested mask for a given predicate is described formally in Section 4.4.

[3] We represent each predicate instance owned by the state with a new branch for the sake of clarity. In reality, there might be infinitely many (possibly uncountable) predicate instances and should be represented by a partial function properly.

4.3 Complete State Model

The nested mask introduced in the previous section stores only permission information. The missing part of the state model is the actual value information of the heap locations. These two parts together constitute the complete state model of our semantics. This section gives an intuition on this state model.

Our semantics uses a *total heap* to store the values of the heap. The idea is not new and already used in an existing isorecursive semantics without predicates [11] and is typical for IDF. With a total heap, every heap location, whether the state holds permission to it or not, stores a concrete value. If a state is reached, where a heap location l has value v in the heap but no corresponding permission, then the semantics ensures that any execution can reach the same state where l has value v' instead. This property models the fact that values for heap locations without permissions do not affect whether a program verifies or not.

Now we give the formal definition of the state model.

Definition 4.1 (state model) *A Viper state is a 2-tuple (H, M), where*

$$H : L \to V$$

is a total mapping from heap locations L to Viper values V, and

$$M : \mathsf{NestedMask}$$

is the nested mask representing permissions owned by the state.

Next, we define some notations used in the development of the semantics.

Notation 4.2 *We use $x.1$ and $x.2$ to represent the first and the second component of a pair, respectively. Therefore, given a nested mask M,*

$$M.1 : L \rightharpoonup \mathbb{R}^+$$

denotes the heap mask, and

$$M.2 : L \rightharpoonup \mathbb{R}^+ \times \mathsf{NestedMask}$$

denotes the map from predicate locations to their permissions and nested masks.

These notations can be nested to obtain deeper information of a nested mask. For example, if we want to obtain the permission of the predicate instance $P(x)$ owned by the mask M, we could use $M.2(P(x)).1$.

Despite having a multi-layer way of representing permissions, there is only one heap that is shared among all nested masks. This makes the state model simpler since we do not need to deal with inconsistent heap values, compared to having a separate heap in each nested mask.

With the total heap, a *total evaluation* of expressions can be defined. This contrasts the normal evaluation of a Viper expression, where *well-definedness* must be checked before evaluation. Well-definedness means every heap location needed to be evaluated must have a non-zero permission in the state since a heap location without permission should not be accessed. The normal evaluation is used in most parts of the semantics (see Section 5.2), since reading a heap location with zero permission is not allowed and makes the execution fail. On the other hand, the total evaluation is used in defining consistency in Section 4.4.1. In the total evaluation, well-definedness is not checked, and the values of heap locations are just obtained by consulting the total heap H without error. Partial operations, such as integer division, are also made total by assigning an unknown but fixed value to inputs that otherwise yield undefined results. This gives us a completely total evaluation, in the sense that the evaluation of all type-correct expressions does not fail. Defining this evaluation is straightforward, and we omit its formal definition. We refer to the big-step total evaluation as $e \Downarrow_t v$ and use it in the development of the semantics.

4.4 Defining State Consistency

In Section 3.1, we showed in the example (Figure 3.2) that the semantics must detect the inconsistencies that back-ends are able to detect. One way of doing so is by pruning states with more than one full permission to some heap location. We decide to take this design choice, and consider any state with more than full permission to some heap location to be inconsistent. To define this precisely, we need to answer two questions.

1. **How to acquire the nested mask for a predicate?** In Section 4.2, we showed the structure of the nested mask for a simple predicate informally. How to construct the nested mask formally for any predicate must be defined formally. It is used when the heap contains a predicate instance, and a nested mask must be associated with the predicate instance in the state.

2. **How to check if the permission to each heap location is no more than** 1**?** The nested mask is a recursive data structure, and permissions to the same heap location might scatter across the nested mask. We need a way to formally define this property over the nested mask.

We answer these two questions by giving two types of consistency, *external consistency* and *internal consistency*, respectively.

1. **External Consistency**. External consistency defines what the nested mask should be for a given predicate instance. It was informally demonstrated by the example in Section 4.2. Intuitively, the nested mask for a given predicate instance should contain exactly what is specified in the body of a predicate and nothing more. External consistency disregards the restriction of not exceeding full permission to any heap location. For example, if the predicate body specified two full permissions to some heap location, the nested mask should also contain exactly two full permissions to that heap location in its heap mask. Note that external consistency must construct the state recursively since the predicate must be recursive. External consistency is formally defined in Section 4.4.1.

2. **Internal Consistency**. Internal consistency depends only on the state model, and it intuitively means the permission to every heap location should not exceed 1. Internal consistency can be defined via summing over the nested mask for each heap location recursively. However, since the nested mask might have infinite branches, how to sum over the data structure is not obvious. It is defined formally in Section 4.4.2.

To see how these two types of consistency are used together. We use the inhale operation as an example. The inhale of a predicate instance is achieved in two steps, which correspond to external consistency and internal consistency. In the first step, we construct the state using external consistency. For readers familiar with *unbounded separation logic* [10], it is equivalent to constructing the state in unbounded separation logic. In other words, we do not care how much permission each heap location has as long as it satisfies the predicate body. This is also why the signature of the heap mask is $L \rightharpoonup \mathbb{R}^+$. However, to make important separation logic rules hold (such as the frame rule), we must impose upper bounds on the permissions of each heap location *at statement boundaries*. Therefore, the second step is to use internal consistency to prune any state with more than full permission to some heap location. Suppose we are to inhale `P(x)` to a state $\varphi = (H, M)$. The first step is to construct a nested mask M_{inh} that is *externally consistent* with `P(x)`. M_{inh} is the nested mask representation of `P(x)`. Then we add M_{inh} to $M.2(\texttt{P(x)}).2$ (defined as a location-wise addition), and increase $M.2(\texttt{P(x)}).1$ by 1. The resulting state might have more than one permission for some heap locations. This is where we need to impose upper bounds on heap locations and use internal consistency to prune any inconsistent states.

Separating the two types of consistency gives us clean and modular ways of dealing with the semantics and proving properties on the semantics. In particular, internal consistency can be defined entirely on the state (decoupled from predicate definitions), and external consistency isolates precisely the property required on the state with respect to the predicate definitions. The benefits of doing such a split were made clear by Dardinier *et al.* [10] and re-observed while developing the nested mask semantics and extending the approach to isorecursive predicates settings.

4.4.1 External Consistency

External consistency defines what a nested mask should be like for a given predicate. Predicates might be recursive, and the nested mask cannot be constructed in practice using an algorithm since the depth of a predicate is potentially unbounded. However, the semantics does not have to be operational. Instead, we use inductive propositions to define it.

The first step towards defining external consistency is defining a "one-level" consistency, which we call *satisfiability*, denoted by the inductively defined proposition sat. The proposition sat only takes an assertion, a heap mask, and a predicate mask into account. Notably, it does not restrict what the nested masks should be. By recursively restricting nested masks on all layers, we easily obtain the final definition of external consistency. We begin with the definition of sat, which is at the core of external consistency.

Figure 4.1 gives the formal inductive definition of sat. Since sat depends only on the heap mask and the predicate mask, and leaves the total heap alone, we omit H from the arguments of sat. All expression evaluation is done by looking up the heap H implicitly. For naming conventions, we use M_h and M_p for the heap mask

$$\textsc{SatAcc}\quad \frac{e_r \Downarrow_t r \qquad e_p \Downarrow_t p \qquad M_h = [\![(r,f) \mapsto p]\!] \qquad M_p = \varnothing}{\mathsf{sat}\ \ M_h\ \ M_p\ \ \texttt{acc}(e_r.f, e_p)}$$

$$\textsc{SatAccWildcard}\quad \frac{e_r \Downarrow_t r \qquad p > 0 \qquad M_h = [\![(r,f) \mapsto p]\!] \qquad M_p = \varnothing}{\mathsf{sat}\ \ M_h\ \ M_p\ \ \texttt{acc}(e_r.f, \texttt{wildcard})}$$

$$\textsc{SatAccPredicate}\quad \frac{\vec{e} \Downarrow_t \vec{a} \qquad e_p \Downarrow_t p \qquad M_h = \varnothing \qquad M_p = [\![(P,\vec{a}) \mapsto p]\!]}{\mathsf{sat}\ \ M_h\ \ M_p\ \ \texttt{acc}(P(\vec{e}), e_p)}$$

$$\textsc{SatAccPredicateWildcard}\quad \frac{\vec{e} \Downarrow_t \vec{a} \qquad p > 0 \qquad M_h = \varnothing \qquad M_p = [\![(P,\vec{a}) \mapsto p]\!]}{\mathsf{sat}\ \ M_h\ \ M_p\ \ \texttt{acc}(P(\vec{e}), \texttt{wildcard})}$$

$$\textsc{SatPure}\quad \frac{e \Downarrow_t \texttt{true} \qquad M_h = \varnothing \qquad M_p = \varnothing}{\mathsf{sat}\ \ M_h\ \ M_p\ \ e} \qquad\qquad \textsc{SatStar}\quad \frac{\mathsf{sat}\ \ M_{hA}\ \ M_{pA}\ \ A \qquad \mathsf{sat}\ \ M_{hB}\ \ M_{pB}\ \ B}{\mathsf{sat}\ \ (M_{hA} \oplus M_{hB})\ \ (M_{pA} \oplus M_{pB})\ \ A\ \&\&\ B}$$

Figure 4.1: The definition of satisfiability (sat).

and the predicate mask, respectively. The precise type of M_h and M_p is as follows.

$$M_h : L \rightharpoonup \mathbb{R}^+ \qquad M_p : P \rightharpoonup \mathbb{R}^+$$

We use the following notation to construct partial functions from scratch.

Notation 4.3 *The notation*

$$[\![x_1 \mapsto v_1, x_2 \mapsto v_2, \cdots]\!]$$

denotes the partial function where x_1 is mapped to v_1, and x_2 is mapped to v_2, etc. All other unspecified inputs are undefined. Besides, we use the empty set symbol $\varnothing$ to denote the partial function that is undefined everywhere.

The definition of sat in Figure 4.1 is standard in separation logic. Note that we require the mask constructed for an assertion to be "precise", in the sense that it cannot have more permissions than what the assertion specifies. Using sat, we

$$\textsc{ExtCons}\quad \frac{\forall\, p.\ M.2(p) \text{ defined} \implies \mathsf{extcons}_{M.2(p).1,p}\ (M.2(p).2)}{\mathsf{extcons}\ (M)}$$

$$\textsc{ExtConsPredLoc}\quad \frac{\mathsf{sat}\ \ M.1\ \ (\mathsf{fst} \circ M.2)\ \ \mathsf{synmult}\ (q, \mathsf{body}(P(\vec{v}))) \qquad \mathsf{extcons}\ (M)}{\mathsf{extcons}_{q,P(\vec{v})}\ (M)}$$

Figure 4.2: The definition of external consistency (extcons). The heap H is implicitly used in sat. If ambiguity arises, we use $H \models \mathsf{extcons}\,(M)$ to specify the heap where the external consistency is satisfied.

$$\begin{aligned}
&\mathsf{synmult}\ (q, \texttt{acc}(e_1.f, e_2)) &&= \texttt{acc}(e_1.f, e_2 * q)\\
&\mathsf{synmult}\ (q, \texttt{acc}(e_1.f, \texttt{wildcard})) &&= \texttt{acc}(e_1.f, \texttt{wildcard})\\
&\mathsf{synmult}\ (q, \texttt{acc}(P(\vec{e}), e_2)) &&= \texttt{acc}(P(\vec{e}), e_2 * q)\\
&\mathsf{synmult}\ (q, \texttt{acc}(P(\vec{e}), \texttt{wildcard})) &&= \texttt{acc}(P(\vec{e}), \texttt{wildcard})\\
&\mathsf{synmult}\ (q, e) &&= e\\
&\mathsf{synmult}\ (q, A\ \&\&\ B) &&= \mathsf{synmult}\ (q, A)\ \ \&\&\ \mathsf{synmult}\ (q, B)
\end{aligned}$$

Figure 4.3: The definition of syntactic multiplication.

define external consistency by applying it inductively. The definition of external consistency is shown in Figure 4.2.

The definition of extcons consists of only two mutual recursive rules. The rules check every existing predicate location recursively. Intuitively, to derive that a nested mask M is externally consistent, one needs to consider every predicate location p which M holds at least some permission to. A predicate location p is of shape $P(\vec{v})$, where P is the predicate name and $\vec{v}$ is a list of evaluated values. Each predicate declaration has a body, and $\mathsf{body}(P(\vec{v}))$ means taking the body of P and substituting its variable references with the corresponding ones specified in $\vec{v}$. Our definition says, for each predicate location $P(\vec{v})$ that exists in the state, $\mathsf{body}(P(\vec{v}))$ *syntactically multiplied* (defined in Figure 4.3) by the permission of that predicate location should be satisfied by the direct masks ($M.1$ and $(\mathsf{fst} \circ M.2)$ in the definition) of the nested mask associated with $P(\vec{v})$. Since the body of $P(\vec{v})$ might contain other predicates, its nested mask also has to be themselves externally consistent.

4.4.2 Internal Consistency

As we motivated in the previous chapter, the semantics should prune any state that contains more than full permission to a heap location, including the permissions folded inside a predicate. In our semantics, this restriction is handled by internal consistency. Achieving this on the nested mask model is intuitive. The nested mask is merely an algebraic structure that stores positive real numbers as permissions, and all permissions to heap locations (including those inside a predicate) are exposed somewhere in the nested mask. Intuitively, we must calculate the sum over the entire nested mask for each heap location and check if they are no more than 1. However, in practice, this task is non-trivial because a state can have an infinite number of predicates that own the same heap location.[4]

We define the permission amount of a heap location based on an infinite sum of an uncountable set. We express the infinite sum using a function over an index set (in our setting, the set P of all predicate locations). The sum of a function $f : P \to \mathbb{R}$ over its domain is defined as the limit of the sum of finite subsets of P, formalized as below.

Definition 4.4 (infinite sum of a positive real-value function) *A function*

$$f : P \rightharpoonup \mathbb{R}$$

has a sum of s, denoted as $f \to s$, if and only if for all $\varepsilon > 0$, there exists a finite subset A of P, such that for all finite subset $B \supset A$ of P, $s - \varepsilon \leq \mathsf{sum}(B) \leq s + \varepsilon$, where $\mathsf{sum}(B)$ is the sum of all elements in the set B which is always well-defined on finite sets.

The theory behind the definition is rooted in topology [14, 15] which is beyond the scope of this thesis. Our definition for summation over a function is just a special case and enjoys several key properties that have already been proven in the literature. An Isabelle/HOL library also exists, which is enough to prove properties we need in this work. Several properties are listed in Appendix B.

[4] An infinite number of predicates can be specified with an *iterated separating conjunction* (called *quantified permission* in Viper's terminology). We do not elaborate on this feature further in this work.

Definition 4.4 is used to define the permission amount of a heap location over the entire nested mask. We give an intuition on how it is defined before showing the formal definition. Now let us fix a heap location l. Looking at the top level of the nested mask), the permission to l might exist in the heap mask and any of the predicate instances stored in the predicate mask. There is only one heap mask, but a potentially infinite number of predicate instances. Handling the first part is straightforward. Describing the sum of the latter part needs the infinite sum. In fact, if the total amount of permission to l from all predicate subtrees is s, then there must exist a function $f : P \longrightarrow \mathbb{R}$, where $f(p)$ is the amount of permission to l in the subtree p for all predicate instance p, and the sum of f is s (i.e. $f \longrightarrow s$). The value s plus the permission amount stored in the heap mask is the total permission amount to l in the entire state. Recursively, we use the same way to describe that $f(p)$ is the total permission amount to l in the subtree for p.

Now we formally define the permission amount to a heap location over the nested mask.

Definition 4.5 *Given a nested mask M and a heap location $l \in L$, we say M owns s permission of l, denoted as $M \xrightarrow{l} s$, if it can be derived using the following rule.*

HeapLocSum

$$\frac{\exists f : P \longrightarrow \mathbb{R}_0^+.\ f \longrightarrow (s - M.1(l)) \land \\ \left(\forall p \in P.\ \textsf{if } M.2(p) \textsf{ undefined then } f(p) = 0 \textsf{ else } M.2(p).2 \xrightarrow{l} f(p)\right)}{M \xrightarrow{l} s}$$

With this definition, we can finally give the definition of internal consistency.

Definition 4.6 (internal consistency) *A nested mask M is considered internally consistent, denoted as* $\textsf{intcons}\,(M)$*, if and only if*

$$\forall l \in L.\ \exists s \leq 1.\ M \xrightarrow{l} s.$$

Chapter 5

Semantics

In the last chapter, we introduced nested masks, the novel state model for Viper semantics which is suitable for representing separation logic predicates. This chapter defines an operational semantics for Viper programs based on this state model. We will start with an overview of the design of the semantics, then proceed with a more formal definition of expression evaluation with well-definedness checks, and finally define the semantics for each statement individually.

This chapter is mainly about the formal semantics of the Viper language. Properties of the semantics will be discussed in Chapter 6 to validate back-end implementations. However, some properties are briefly mentioned in this chapter when they come naturally. All definitions in this chapter are mechanized in Isabelle/HOL.

5.1 Overview of the Semantics

A Viper program consists of global declarations (e.g. fields and predicates) and methods. A method has a statement as its body, and is executed sequentially. There are three possible outcomes of executing a statement.

1. **It might** *fail*. A statement fails when an assertion does not always hold, because of a division by zero, or when there is not enough permission to access a field, etc. In the semantics, we use the symbol $\lightning$ to represent a failure.
2. **It might** *stop*. A statement stops if no failure occurs, but the execution does not proceed further due to an inconsistency. After an execution stops, we

say it is in the *magic* state. We use the symbol $\circledcirc$ to represent a magic state. One example is `assume false`. It makes the assumptions inconsistent and all following code irrelevant. Therefore, the semantics prunes this execution and the rest of the method is trivially correct. Another example is inhaling more than full permission to a heap location. The resultant state is inconsistent and thus pruned by the semantics. Note that a program stops does not imply it has errors. It stops quietly and simply does not proceed.

3. **It might** *succeed.* A statement succeeds if no failure occurs while executing the statement and it does not stop (i.e. the execution continues).

The semantics is non-deterministic, which means that the execution of a statement might have multiple outcomes. For example, the semantics can go to either a success state a the magic state after inhaling a wildcard permission, depending on the concrete permission value the semantics chooses for the wildcard. A method is considered correct if and only if no possible execution of its body results in a failure state.

The following two sections introduce the big-step expression evaluation and statement semantics. The result of an expression evaluation might be a value (success), or $\lightning$ (failure). If during the reduction of a statement, some expression evaluation fails, the whole statement also results in a failure. The outcome of executing a statement has the three said possibilities, represented by a concrete state, $\lightning$, or $\circledcirc$.

5.2 Expression Evaluation

In this section, we formally define the expression evaluation. Expression evaluation is an essential part of the semantics, and will be used in Section 5.3 when we define statement semantics. Moreover, expression evaluation also demonstrates the advantages of the new state model over the equirecursive one. We will discuss its benefits in this section.

Note that we already mentioned a total evaluation ($\Downarrow_t$) in Section 4.3 without giving its formal definition. The evaluation we define in this section is the default one used by all parts of the semantics except external consistency. We call this evaluation the *partial evaluation*. Partial evaluation is different as it checks the well-definedness of the expression. For example, accessing a heap location with zero

$$\textsc{EvalVar}\quad \frac{S(x) = v}{V, H, M \models x \Downarrow v} \qquad\qquad \textsc{EvalLit}\quad \frac{}{V, H, M \models \mathsf{lit}(v) \Downarrow v}$$

$$\textsc{EvalFieldAccess}\quad \frac{V, H, M \models e \Downarrow r \qquad M.1(r, f) > 0 \qquad H(r, f) = v}{V, H, M \models e.f \Downarrow v}$$

$$\textsc{EvalFieldAccessFailure}\quad \frac{V, H, M \models e \Downarrow r \qquad M.1(r, f)\ \mathsf{undefined}}{V, H, M \models e.f \Downarrow \lightning}$$

$$\textsc{EvalBinaryOp}\quad \frac{V, H, M \models e_1 \Downarrow v_1 \qquad V, H, M \models e_2 \Downarrow v_2}{V, H, M \models e_1\ \mathsf{bop}\ e_2 \Downarrow \mathsf{bop}(v_1, v_2)} \qquad \textsc{EvalUnaryOp}\quad \frac{V, H, M \models e \Downarrow v}{V, H, M \models \mathsf{uop}\ e \Downarrow \mathsf{uop}(v)}$$

$$\textsc{EvalPermAcc}\quad \frac{V, H, M \models e \Downarrow r}{V, H, M \models \mathtt{perm}(e.f) \Downarrow \mathsf{if}\ M.1(r, f)\ \mathsf{undefined\ then}\ 0\ \mathsf{else}\ M.1(r, f)}$$

$$\textsc{EvalPermPredicate}\quad \frac{V, H, M \models \vec{e} \Downarrow \vec{v}}{V, H, M \models \mathtt{perm}(P(\vec{e})) \Downarrow \mathsf{if}\ M.2(P(\vec{v}))\ \mathsf{undefined\ then}\ 0\ \mathsf{else}\ M.2(P(\vec{v})).1}$$

Figure 5.1: Big-step evaluation of Viper expressions.

permission succeeds with the total evaluation, but results in a failure in the partial evaluation.

The big-step expression evaluation is formally defined in Figure 5.1. We use the judgement $V, H, M \models e \Downarrow v$ to denote the evaluation of a Viper expression e under the store V (a mapping from local variables to values), the total heap H, and the nested mask M. The evaluation might fail, represented by the $\lightning$ symbol. As an example, in the EvalFieldAccessFailure case, if there is no permission of the heap location (r, f), the evaluation fails. A binary operation may also fail, for example, because of a division by zero. This is reflected by v_1 bop v_2 returning $\lightning$

on undefined inputs.

The total evaluation $\Downarrow_t$ is defined very similarly, except that permissions are not checked for field access, and binary operations always return a fixed value on normally undefined inputs.

From the rule EvalPermAcc and EvalPermPredicate, we can see how permission introspection works in the new state model. A permission introspection expression should return the exact amount of direct permission of the specified location owned by the state. It is straightforward to do so in an isorecursive semantics, but impossible in an equirecursive setting. Our state model keeps this information in its direct masks, and as a result, it supports permission introspection in a straightforward manner.

5.3 Statement Semantics

In this section, we define a big-step operational semantics for a Viper program. We only present the rules for a subset of the Viper language, given in Figure 2.1. In particular, it shows the rules for all four important operations on predicates: fold, unfold, inhale, and exhale. Subsections 5.3.1–5.3.4 represent these four statements each. Besides, we omit the statements that are irrelevant to predicates. The semantics for these statements can be defined in a straightforward way.

A successful execution in the semantics makes a state transition into a new state. We use the judgement

$$V, H, M \xrightarrow{s} V', H', M'$$

to denote the big-step execution of the statement s, where the state composed of the store V, the heap H, and mask M transitions into the new state V', H', M'. Moreover, we use the notation

$$V, H, M \xrightarrow{s} \lightning \qquad V, H, M \xrightarrow{s} ◎$$

to indicate a failure and going to the magic state, respectively. The store V is only changed by variable assignments, which we do not present here. For all statements we consider, V never changes and is only used to evaluate expressions. Readers may safely ignore V. A change in H only appears in field assignments and sometimes in exhale. The mask change exists in all four operations.

5.3.1 Inhale

We define the semantics of inhale in this section. A Viper inhale statement is written as

$$\texttt{inhale}\ A$$

where A is an assertion. The assertion syntax is presented in Figure 2.1. Intuitively, the behavior of `inhale` A can be expressed by the Hoare triple

$$\{R\}\ \texttt{inhale}\ A\ \{R * A\}$$

The inhale adds resources specified by A, and assumes all logical constraints. The semantics can be informally expressed by constructing a separate state where A is satisfied and adding the state to the current state.

$$\frac{V, H, M_A \models A}{V, H, M \xrightarrow{\texttt{inhale}\ A} V, H, M \oplus M_A}$$

This rule explains the intuitive behavior of inhale. However, we also want an operational semantics of inhale which reflects back-end implementations. To meet this requirement, we define an auxiliary inhale semantics, shown in Figure 5.2.

The following definition is used in the semantics of inhale, which adds permissions to some location (heap location or predicate location) to the nested mask in four of the rules (inhaling permission to a heap location or a predicate instance).

Definition 5.1 (location addition) *The notation*

$$M\left[\!\!\left[(r, f) \overset{+}{=} q\right]\!\!\right]$$

denotes adding q permission to the heap location (r, f). Namely,

$$\begin{aligned} M\left[\!\!\left[(r, f) \overset{+}{=} q\right]\!\!\right].1(l) &= \text{if } [l = (r, f)] \text{ then } q + M.1(l) \text{ else } M.1(l) \\ M\left[\!\!\left[(r, f) \overset{+}{=} q\right]\!\!\right].2 &= M.2 \end{aligned}$$

Similarly, the notation

$$M\left[\!\!\left[P(\vec{v}) \overset{+}{=} (q, M')\right]\!\!\right] \quad (q > 0)$$

denotes adding q permission and the nested M' to the predicate location $P(\vec{v})$ which is

$$\textsc{InhaleAcc}\quad \frac{\begin{array}{c} V,H,M \models e_1 \Downarrow r \\ V,H,M \models e_2 \Downarrow p \qquad \text{if intcons}\left(M\left\llbracket (r,f) \overset{\pm}{=} p \right\rrbracket\right) \text{ then } \mathcal{R} = V,H,M\left\llbracket (r,f) \overset{\pm}{=} p \right\rrbracket \text{ else } \mathcal{R} = \circledcirc \end{array}}{V,H,M \xrightarrow{\texttt{inhale acc}(e_1.f,e_2)} \mathcal{R}}$$

$$\textsc{InhaleAccWildcard}\quad \frac{\begin{array}{c} V,H,M \models e \Downarrow r \\ p > 0 \qquad \text{if intcons}\left(M\left\llbracket (r,f) \overset{\pm}{=} p \right\rrbracket\right) \text{ then } \mathcal{R} = V,H,M\left\llbracket (r,f) \overset{\pm}{=} p \right\rrbracket \text{ else } \mathcal{R} = \circledcirc \end{array}}{V,H,M \xrightarrow{\texttt{inhale acc}(e.f,\texttt{wildcard})} \mathcal{R}}$$

$$\textsc{InhalePredicate}\quad \frac{\begin{array}{c} V,H,M \models \vec{e} \Downarrow \vec{v} \qquad V,H,M \models e_2 \Downarrow p \qquad \text{extcons}_{P(\vec{v}),p}\left(M_{P(\vec{v})}\right) \\ \text{if intcons}\left(M\left\llbracket P(\vec{v}) \overset{\pm}{=} (p, M_{P(\vec{v})}) \right\rrbracket\right) \text{ then } \mathcal{R} = V,H,M\left\llbracket P(\vec{v}) \overset{\pm}{=} (p, M_{P(\vec{v})}) \right\rrbracket \text{ else } \mathcal{R} = \circledcirc \end{array}}{V,H,M \xrightarrow{\texttt{inhale acc}(P(\vec{e}),e_2)} \mathcal{R}}$$

$$\textsc{InhalePredicateWildcard}\quad \frac{\begin{array}{c} V,H,M \models \vec{e} \Downarrow \vec{v} \qquad p > 0 \qquad \text{extcons}_{P(\vec{v}),p}\left(M_{P(\vec{v})}\right) \\ \text{if intcons}\left(M\left\llbracket P(\vec{v}) \overset{\pm}{=} (p, M_{P(\vec{v})}) \right\rrbracket\right) \text{ then } \mathcal{R} = V,H,M\left\llbracket P(\vec{v}) \overset{\pm}{=} (p, M_{P(\vec{v})}) \right\rrbracket \text{ else } \mathcal{R} = \circledcirc \end{array}}{V,H,M \xrightarrow{\texttt{inhale acc}(P(\vec{e}),\texttt{wildcard})} \mathcal{R}}$$

$$\textsc{InhalePure}\quad \frac{V,H,M \models e \Downarrow \text{true}}{V,H,M \xrightarrow{\texttt{inhale}\ e} V,H,M} \qquad\qquad \textsc{InhalePureMagic}\quad \frac{V,H,M \models e \Downarrow \text{false}}{V,H,M \xrightarrow{\texttt{inhale}\ e} \circledcirc}$$

$$\textsc{InhaleStar}\quad \frac{V,H,M \xrightarrow{\texttt{inhale}\ A} V,H,M_A \qquad V,H,M_A \xrightarrow{\texttt{inhale}\ B} \mathcal{R}}{V,H,M \xrightarrow{\texttt{inhale}\ A\ \&\&\ B} \mathcal{R}} \qquad \textsc{InhaleStarFailureFirst}\quad \frac{V,H,M \xrightarrow{\texttt{inhale}\ A} \mathcal{R} \qquad \mathcal{R} = \lightning \lor \mathcal{R} = \circledcirc}{V,H,M \xrightarrow{\texttt{inhale}\ A\ \&\&\ B} \mathcal{R}}$$

$$\textsc{InhaleFailure}\quad \frac{e \text{ is a direct sub-expression of } A \ \land\ V,H,M \models e \Downarrow \lightning}{V,H,M \xrightarrow{\texttt{inhale}\ A} \lightning}$$

Figure 5.2: Operational inhale semantics.

formally defined as (the symbol $\oplus$ is defined later in the definition)

$$\begin{aligned}
M\left\llbracket P(\vec{v}) \overset{\pm}{=} (q, M') \right\rrbracket.1 &= M.1 \\
M\left\llbracket P(\vec{v}) \overset{\pm}{=} (q, M') \right\rrbracket.2(p).1 &= \text{if } \ [p = P(\vec{v})] \ \text{ then } q + M.2(p).1 \text{ else } M.2(p).1 \\
M\left\llbracket P(\vec{v}) \overset{\pm}{=} (q, M') \right\rrbracket.2(p).2 &= \text{if } \ [p = P(\vec{v})] \ \text{ then } M' \oplus M.2(p).2 \text{ else } M.2(p).2
\end{aligned}$$

If $M.2(p)$ is undefined, $M.2(p).1$ and $M.2(p).2$ both return undefined*. Thus, the real number addition is extended with the undefined value as follows.*

$$\begin{aligned} x + \mathsf{undefined} &= x \\ \mathsf{undefined} + y &= y \end{aligned}$$

The addition of two nested masks, $\oplus$, is recursively defined as follows.

$$\begin{aligned} (M_1 \oplus M_2).1(l) &= M_1.1(l) + M_2.1(l) \\ (M_1 \oplus M_2).2(p).1 &= M_1.2(p).1 + M_2.2(p).1 \\ (M_1 \oplus M_2).2(p).2 &= M_1.2(p).2 \oplus M_2.2(p).2 \\ M_1 \oplus \mathsf{undefined} &= M_1 \\ \mathsf{undefined} \oplus M_2 &= M_2 \end{aligned}$$

Intuitively, the statement `inhale` A adds permissions specified in the assertion A, and goes to magic if the resultant state is not internally consistent (see rules InhaleAcc, InhaleAccWildcard, InhalePredicate, and InhalePredicateWildcard). The statement goes to magic if a logical constraint in A is not satisfied (InhalePureMagic) and does nothing if the constraint holds (InhalePure). Besides, it might also fail if evaluating an expression inside A results in a failure (InhaleFailure). For separating conjunction (A && B in Viper's syntax), the semantics processes the assertion from left to right (see InhaleStar). If inhaling the left component fails or stops, then inhaling the whole separation conjunction also fails or stops. Otherwise, the inhale of the second component is done in the state after inhaling the first component (M_A in InhaleStar).

Among all the inhale rules, InhalePredicate is the most interesting one which is also new to the semantics. Most other rules already exist in a similar form in the existing formalization. By definition, all predicates stored in the state must have a nested mask associated with them to represent all permissions included in that predicate recursively. The construction of such a nested mask was already formally defined in Section 4.4.1. Here, we just use the inductively defined proposition extcons to constrain the nested mask for the predicate. Therefore, $M_{P(\vec{v})}$ is a nested mask, which contains the structure and permissions of p permission of the predicate $P(\vec{v})$. Then, the permission p together with the constructed nested mask $M_{P(\vec{v})}$ is added to $M.2(P(\vec{v}))$.

One question readers might wonder is, if say, there is already q permission to $P(\vec{v})$ in the mask M, and we try to inhale another p permission to the same predicate location, will the sum of the two mask satisfies $q + p$ of $P(\vec{v})$? In other words, does inhaling `acc`$(P(\vec{v}), p + q)$ have the same behavior as inhaling `acc`$(P(\vec{v}), p)$ and `acc`$(P(\vec{v}), q)$ consecutively? The short answer is yes. And this is one benefit of working in the unbounded separation logic without considering internal consistency, where *combinability* holds. The property is formally given in Section 6.3.2.

Given these good properties, the semantics for inhaling a predicate is rather straightforward. First, we construct the externally consistent state for the predicate. Then we add this state to the current state. This step is done without pruning any inconsistent state. Finally, we use internal consistency to prune any state with more than full permission to some heap location. If the state gets pruned, the execution goes to magic. This step considers permissions to heap locations altogether including the direct ones and the ones folded in predicate. This precisely solves the problem demonstrated in Section 3.1 which the purely isorecursive semantics suffers from.

5.3.2 Exhale

Exhale is the dual of inhale. Intuitively, `exhale` A removes permissions specified in A, and asserts logical constraints in A. If some logical constraint does not hold, the exhale statement results in a failure.

In contrast to inhale, exhale does not prune executions; instead, it performs non-deterministic assignments to the heap locations where *all* permissions are removed. For example, starting from a state where `x.f` stores the integer 1 with full permission, the statement

```
exhale acc(x.f)
```

removes the permission from the state, and non-deterministically assigns a value to `x.f`. This can be seen as a branching of the execution, where one execution forks into potentially infinitely many executions, where each execution holds a different value of `x.f`.

Figure 5.3 shows two operational rules for exhale[1]. Similar to inhale, the semantics for exhale also employs an operational semantics, to reflects how back-ends treat them. We omit most exhale rules and only show the most interesting ones.

[1] We omit other straightforward cases.

$$\textsc{ExhaleAcc}\quad \frac{\begin{array}{c} V, H, M \models e_1 \Downarrow r \\ V, H, M \models e_2 \Downarrow p \qquad M.1(r, f) \geq p \qquad M' = M[\![(r, f) := M.1(r, f) - p]\!] \\ \forall\, l \neq (r, f). H(l) = H'(l) \qquad \forall\, l \in L, s > 0.\ M \xrightarrow{l} 0 \lor M' \xrightarrow{l} s \implies H(l) = H'(l) \end{array}}{V, H, M \xrightarrow{\texttt{exhale acc}(e_1.f, e_2)} V, H', M'}$$

$$\textsc{ExhalePredicate}\quad \frac{\begin{array}{c} V, H, M \models \vec{e} \Downarrow \vec{v} \\ V, H, M \models e_2 \Downarrow p \qquad M.2(P(\vec{v})) \geq p \qquad M' = M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} 1 - \dfrac{p}{M.2(P(\vec{v})).1}\right]\!\!\right] \\ \forall\, l \in L, s > 0.\ M \xrightarrow{l} 0 \lor M' \xrightarrow{l} s \implies H(l) = H'(l) \end{array}}{V, H, M \xrightarrow{\texttt{exhale acc}(P(\vec{e}), e_2)} V, H', M'}$$

Figure 5.3: Operational exhale semantics.

Let us take a close look at the rule ExhaleAcc. This rule removes permission to some heap location. After evaluating the reference r and the permission amount p, the rule checks if there is sufficient permission stored in the state, and removes the specified amount if there is. Otherwise, the execution fails (the rule is not shown). Finally, the rule ExhaleAcc potentially makes a non-deterministic assignment. If this process removes all permission to (r, f), a non-deterministic assignment takes place and assigns an unconstrained value to (r, f). The last premise of the rule makes sure the values for heap locations that are not affected by the inhale or with a non-zero permission do not change and other locations (can only be (r, f) in this case) unconstrained.

The rule ExhalePredicate shows the removal of a predicate instance. We remove the nested mask proportionally by multiplying all permission values on each layer by a fraction, using Definition 5.2. For example, if the state owns full permission to $P(\vec{v})$, and the inhale operation removes $\frac{3}{4}$ permission to it, the resultant mask will be $M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} \frac{1}{4}\right]\!\!\right]$.

Definition 5.2 (fractional multiplication) *The fractional multiplication,*

$$M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right] \qquad (q > 0)$$

is defined as

$$
\begin{aligned}
M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right].1 &= M.1 \\
M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right].2(p).1 &= \text{if } \ [p = P(\vec{v})] \ \text{ then } q \cdot M.2(p).1 \text{ else } M.2(p).1 \\
M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right].2(p).2 &= \text{if } \ [p = P(\vec{v})] \ \text{ then } q \odot M.2(p).2 \text{ else } M.2(p).2
\end{aligned}
$$

The fractional multiplication of a nested mask, $\odot$, is recursively defined as follows.

$$
\begin{aligned}
(q \odot M).1(l) &= q \cdot M.1(l) \\
(q \odot M).2(p).1 &= q \cdot M.2(p).1 \\
(q \odot M).2(p).2 &= q \odot M.2(p).2 \\
q \odot \text{undefined} &= \text{undefined}
\end{aligned}
$$

Similar to the addition, we extend the positive real number multiplication with an undefined value as follows.

$$q \cdot \text{undefined} = \text{undefined}$$

Furthermore,

$$M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} 0\right]\!\!\right]$$

is a special case that is defined by

$$
\begin{aligned}
M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right].1 &= M.1 \\
M\left[\!\!\left[P(\vec{v}) \overset{\times}{=} q\right]\!\!\right].2(p) &= \text{if } \ [p = P(\vec{v})] \ \text{ then undefined else } M.2(p)
\end{aligned}
$$

In the nested mask, each predicate location is mapped to a pair of positive real number (the permission) and a nested mask. Multiplying the predicate location $P(\vec{v})$ of the mask M by q applies to both fields. The multiplication of a nested mask is just defined as multiplying every permission by q recursively.

In the rule ExhalePredicate, the removal of a predicate is defined as removing the nested mask associated with the predicate instance proportionally at all levels, using the $\overset{\times}{=}$ operation. If the removal results in some heap locations having zero permission, they are also non-deterministically assigned with unconstrained values

$$\dfrac{\begin{array}{c}\textsc{Unfold}\\ V,H,M \models \vec{e} \Downarrow \vec{v} \qquad V,H,M \models e_2 \Downarrow p \qquad M.2(P(\vec{v})).1 \geq p \\ M' = M\left[\!\!\left[P(\vec{v}) \stackrel{\times}{=} 1 - \dfrac{p}{M.2(P(\vec{v})).1}\right]\!\!\right] \oplus \left(\dfrac{p}{M.2(P(\vec{v})).1} \odot M.2(P(\vec{v})).2\right)\end{array}}{V,H,M \xrightarrow{\texttt{unfold acc}(e_1.f,e_2)} V,H',M'}$$

Figure 5.4: Unfold semantics.

(similar to the ExhaleAcc case).

A similar question that arises in the semantics of exhaling a predicate, is the following: after we remove a fraction of the nested mask, is the remaining part still externally consistent? The answer is again yes. The property is called *factorizability*: if we remove a proportional fraction of the nested mask of a predicate location in an externally consistent state, the resulting state is still externally consistent. This property is formally stated in Section 6.3.2.

5.3.3 Unfold

An unfold statement unboxes one folded predicate, and exposes its content to the top-level mask. Intuitively, an unfold does not create or destroy permissions, but only shuffles the already owned permissions between direct masks and nested masks. In our semantics, the unfold semantics reflects exactly this.

An unfold operation in the semantics only depends on the nested mask of the state. Because the nested mask already has all information about folded predicates, including all non-deterministic choices in the predicate body (e.g. wildcard permissions), the unfold operation does not need information about the predicate body. Therefore, the unfold operation is merely an algebraic operation on the nested mask.

The formal semantics of unfold is shown in Figure 5.4. In the rule, we first need to make sure there is enough permission to unfold ($M.2(P(\vec{v})).1 \geq p$). Then, we multiply the permission and the nested mask of the predicate location $P(\vec{v})$ by $1 - p/M.2(P(\vec{v})).1$ (the remaining fraction of the predicate that is not unfolded).

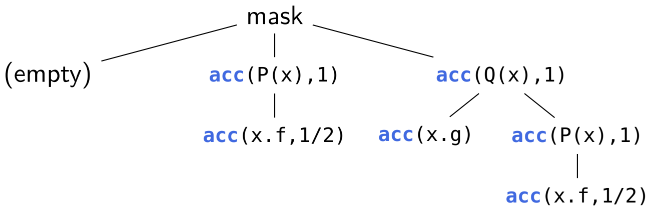
```
field f: Int
field g: Int
predicate P(x: Ref) { acc(x.f, 1/2) }
predicate Q(x: Ref) { acc(x.g) && P(x) }

method m(x: Ref)
    requires P(x) && Q(x)
{
    unfold acc(Q(x), 2/3)
}
```

Figure 5.5: A Viper example to illustrate the semantics of unfold.

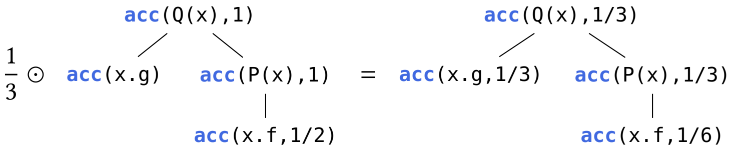
This removes p permission of the predicate from the state. Finally, we add this part back to the top-level masks (the $\oplus$ operation in the rule).

It is intuitive to understand that the unfold operation does not create or destroy permissions, but rather rearranges the permissions between the two masks. As a result, for each heap location, the total amount of permission also does not change (see Section 6.3.1).

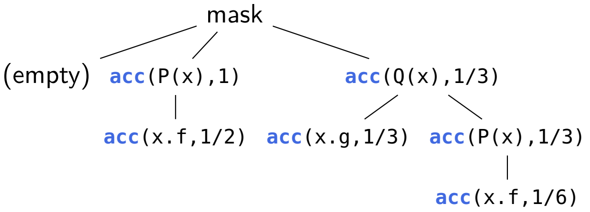
Example. We now show an example to illustrate the execution of an unfold statement. Take the Viper program in Figure 5.5 as an example. We first construct the state where the pre-condition is satisfied (we omit the total heap since all values are unconstrained).



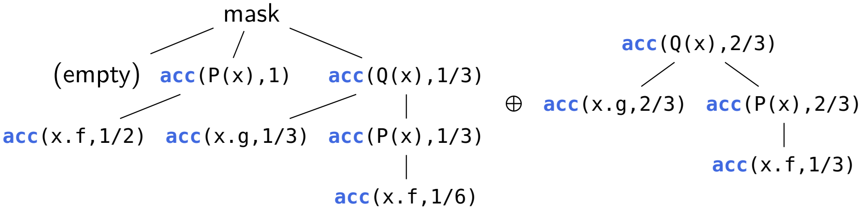
We are unfolding $\frac{2}{3}$ permission of `Q(x)` from the state. However, we have 1 permission to that predicate location. Hence, we need to calculate one-third of the nested mask of Q(x), which is the remaining fraction of the predicate staying in the same place in the tree.



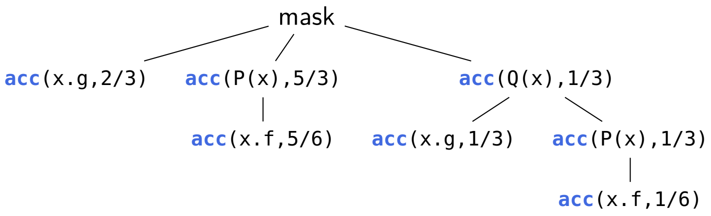
The subtree of acc(Q(x),1) is replaced with the result:



This completes the removal of the predicate. Next, two-thirds (the unfolded part) of the initial nested mask for Q(x) need to be added to the top-level masks:



We merge these two trees recursively and obtain the following result.



As we can see, the total amount of permission of `x.f` and `x.g` do not change after unfolding. The permission only moves inside the tree and is not created or destroyed.

Benefits. The new semantics of unfold enjoys a number of benefits on unfold compared to the purely isorecursive semantics.

- **It only relies on the state**. In the purely isorecursive semantics, predicate instance information is only stored as real numbers representing the permission amount, and no structure of the predicate instance is remembered. As a result, the predicate body must be consulted in order to unfold a predicate. In our semantics, the permission and structure information is stored in state, so unfold becomes a pure algebraic operation on the nested mask. It reflects the intuition that the unfolding of a predicate only changes the representation of existing permissions inside a state and should not rely on external information.

- **It is deterministic**. A given predicate instance might have multiple nested mask representations (e.g. if the predicate body contains a wildcard permission). However, the nested mask already fixes all non-deterministic choices at the creation of the permission (when inhaling), and an unfold is therefore deterministic.

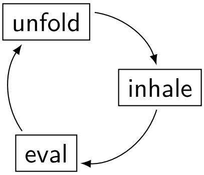
- **It never fails**. The unfold operation does not create or destroy permissions, so the resultant state is always internally consistent, and later in Section 6.3.2 we will show that external consistency is also preserved by unfold. Therefore, the semantics of unfold naturally succeeds and it is not necessary to check if the resultant state is consistent explicitly. In contrast, in purely isorecursive semantics, if a predicate body contains a wildcard permission, there are infinitely many unfoldings of a given predicate. Some of the unfoldings might be internally inconsistent (because they choose a permission too large). This means we need to choose only those unfoldings that result in an internally consistent state. This adds another layer of complexity to the semantics. In our semantics, this non-deterministic choice is made when the predicate instance is inhaled. The inhale semantics guarantees that the state is consistent.

- **It supports `unfolding` expressions better.** Viper supports `unfolding` expressions. For example,

  ```
  unfolding P(x) in x.f
  ```

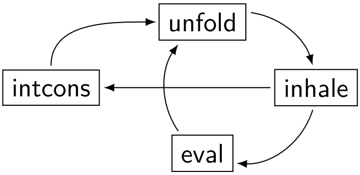
  unfolds `P(x)` temporarily, and reads the value of `x.f`. The state that the expression is evaluated in might not have read permission to `x.f` but the body of `P(x)` might have it. The unfolding expression will also consider the permissions specified inside `P(x)` when it determines the well-definedness of `x.f`, without affecting the current state. Alternatively, one could think of this expression as first unfolding `P(x)` and reading the value of `x.f`, but restoring the previous state before finishing the evaluation.

  In the purely isorecursive semantics, it causes a dependency between seemingly separate parts of the semantics, as shown in the following diagram.

  First, if the state model does not store predicate body information, the most natural way (as implemented in the back-ends) of defining unfold is via inhale, i.e. removing the predicate permission and inhaling its body. Second, inhale needs expression evaluation (recall Figure 5.2). Third, evaluation needs unfold because of `unfolding` expressions. These thus form a cyclic dependency.

  Things get worse if we add internal consistency to the picture, as shown below.

  As a result, different components of the semantics need to be defined in a mutually recursive way, and doing any induction proof on one of the components could be cumbersome.

$$\textsc{Fold}\quad \frac{V,H,M \models \vec{e} \Downarrow \vec{v} \qquad V,H,M \models e_2 \Downarrow p \qquad V,H,M \xrightarrow{\texttt{exhale}\ \mathsf{synmult}(p, P(\vec{v}))} V,H,M'}{V,H,M \xrightarrow{\texttt{unfold}\ \texttt{acc}(e_1.f, e_2)} V,H,M' \left[\!\left[P(\vec{v}) \overset{+}{=} (p, M \ominus M') \right]\!\right]}$$

Figure 5.6: Fold semantics

5.3.4 Fold

The final component of the predicate semantics is fold. A fold statement gathers resources specified in the predicate body, and boxes them into a predicate instance. On the nested mask model, it is reflected by moving permissions on the top-level mask one layer down the tree.

However, the situation for fold is more complicated than for unfold. Unfold is merely an algebraic operation on the nested mask, a "shift up" operation on the corresponding nested mask. In contrast, fold cannot be expressed just as a "shift down" operation, since it is not clear which permissions in the top-level masks belong to the predicate being folded. Besides, a fold operation might be non-deterministic, if, for example, the predicate body contains a wildcard permission. Therefore, the non-deterministic choice must be made when a predicate is being folded.

To calculate which permissions belong to the predicate body, we use an exhale operation. The predicate body is first exhaled, and then the removed permissions are added to the nested mask of the folded predicate location.

The formal definition of fold semantics is given in Figure 5.6. The subtraction of nested masks is defined as the reverse of addition:

$$M_1 \ominus M_2 = M \iff M \oplus M_2 = M_1$$

We only use the subtraction where M_2 is the resultant mask of an exhale starting from M_1. Since exhale only removes permissions from the mask, M_2 contains no more than the permission in M_1 on every location. Therefore, it is easy to show that the subtraction is always well-defined and is unique.

The semantics of fold takes the following steps. After evaluating the predicate location $P(\vec{v})$, the semantics exhales its body syntactically multiplied by p. The syntactic multiplication is similar to what we did in the definition of external consistency. Note that exhale performs non-deterministic assignments, if all permissions are exhaled for some heap location. However, this never happens in fold since the permissions are eventually added back to the mask. Therefore, we restrict the resultant heap of exhale to be unchanged. Finally, we add the removed part $(M \ominus M')$ to the predicate location $P(\vec{v})$. This concludes the semantics of fold.

Chapter 6

Validating the Semantics

One important goal of designing the correct Viper semantics is to validate back-end implementations. The two existing Viper back-ends are implemented in Scala with a very large codebase. Verifying their correctness is highly non-trivial, and bugs in the implementation are continually discovered and fixed. Formally connecting the implementations to the semantics would make these verifiers substantially more trustworthy.

To make this formal connection, we adapt the methodology from the work of Parthasarathy *et al.* [11], which uses *forward simulation* to describe the soundness of a back-end implementation. The work focuses on the translational back-end of Viper, which translates a Viper program into the intermediate verification language Boogie [16]. By using forward simulation, we show that the translation of a Viper statement (in Boogie) *simulates* the source Viper statement (formally defined in Section 6.1). We extend the methodology to support predicates and show how statements related to predicates are simulated by the translation. In this chapter, the idea is explained using the translational back-end of Viper, but similar ideas can also be used to validate the symbolic execution back-end.

In this chapter, we first define forward simulation, and explain how we use forward simulation to express the soundness of a back-end translation (Section 6.1), and then apply this idea to predicate translation (we place emphasis on the unfold statement). This motivates some properties that must hold on the semantics itself (Section 6.2). We formalize these properties and show that the semantics we

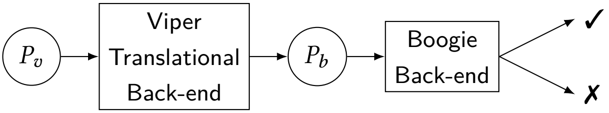


Figure 6.1: The overall architecture of the Viper translational back-end. The Viper program P_v is translated into a Boogie program P_b by Viper's translational back-end. The Boogie back-end tries to verify P_b, reporting either that the program is correct or that it may contain potential errors.

defined satisfies these (Section 6.3).

6.1 Forward Simulation

In the translational back-end of Viper, a Viper program is translated into a Boogie program. Boogie is a lower-level intermediate verification language compared to Viper, without native support for separation logic concepts, such as permissions and predicates. Therefore, the translational backend must translate Viper statements into constructs that Boogie supports. Boogie has its own back-end, and ultimately uses an SMT solver to check the correctness of a Boogie program.

The overall architecture of the translational back-end of Viper is shown in Figure 6.1. In the diagram, P_v is an input Viper program. The translational back-end translates this program into a Boogie program, P_b, whose correctness is checked by the Boogie back-end. The Boogie back-end reports either correct, indicating the absence of error in the encoded program, or a failure, indicating potential errors in the program. Viper directly uses this result as the verification result of the source Viper program. To make the result meaningful, the translation from P_v to P_b must be *sound*:

> *If the translated Boogie program is correct (with respect to the Boogie semantics), the source Viper program must also be correct (with respect to the Viper semantics).*

Here, we do not consider the correctness of Boogie's implementation[1], but only consider the formal semantics of a Boogie program defined by Parthasarathy *et al.* [11].

[1] Existing work already explored the soundness of Boogie's backend [17].

Typically, in the context of a transpiler (source-to-source compiler), we want the translated program to have the exact behavior as the source program. However, in verification, this preservation is not required or even feasible. For practical reasons, we allow the translation to be *incomplete*: the translated program might fail as well, even if the source program is correct with respect to the semantics. Instead, we focus on the soundness of the translation.

To describe the soundness property, we use the notion of *forward simulation*. Intuitively, a forward simulation between a Viper and a Boogie statement shows that for any execution of the Viper statement, there exists a Boogie execution that *simulates* it. Suppose a Viper statement s_v is translated to a Boogie statement (or a sequence of statements) s_b. The general simulation judgement between these two programs is as follows.

$$\begin{aligned}
&\mathsf{sim}(s_v, s_b) \triangleq \forall\, \sigma_v\ \sigma_b.\ R(\sigma_v, \sigma_b) \implies \\
&\quad \left(\forall\, \sigma_v'.\ \sigma_v \xrightarrow[v]{s_v} \sigma_v' \implies \exists\, \sigma_b'.\ \sigma_b \xrightarrow[b]{s_b} \sigma_b' \land R(\sigma_v', \sigma_b')\right) && \text{(success case)} \\
&\quad \left(\sigma_v \xrightarrow[v]{s_v} \lightning \implies \sigma_b \xrightarrow[b]{s_b} \lightning\right) && \text{(failure case)}
\end{aligned}$$

The definition has three main components.

1. **A state relation**. The Viper semantics and the Boogie semantics are designed differently, and their state models are different. Therefore, we use a state relation R to describe the connection between the two state models. For example, in the translation, Viper variables are translated into Boogie variables with potentially different names, Viper permissions are translated into a Boogie map. Intuitively, the state relation says that those variables and the map (among other elements) must agree between the Viper state and Boogie state.

2. **Viper reduction**. In the definition,

   $$\sigma_v \xrightarrow[v]{s_v} \sigma_v'$$

   corresponds to the semantics we defined in Section 5.3. We add v under the arrow to distinguish the Viper semantics from the Boogie one.

3. **Boogie reduction**. The Boogie semantics is already defined in previous work

[17]. The execution of the Boogie statement s_b is denoted as

$$\sigma_b \xrightarrow[b]{s_b} \sigma'_b$$

where the Boogie state σ_b transitions into the state σ'_b. Similar to the Viper semantics, a Boogie execution might also result in a failure (↯) or magic (◎).

The simulation judgement $\mathsf{sim}(s_v, s_b)$ says, starting from any Viper state σ_v and Boogie state σ_b that are related by R, (1) if after the execution of the Viper statement s_v, the state successfully transitions into another state σ'_v, there also exists a Boogie execution of the translated statement s_b that starts from the initial Boogie state σ_b and transitions into the new state σ'_b, which is still related to the final Viper state σ'_v by R, and (2) if the Viper execution fails, there must exist a Boogie execution that fails.

The failure case directly entails that the translation is sound. One equivalent way to express the failure case is that, if a Viper program is incorrect (i.e. if it has at least one failing execution), the encoded Boogie program must have at least one failing execution as well (which is also the incorrectness criteria of a Boogie program). Otherwise, Boogie might (unsoundly) report success which forgoes a potential error in the Viper program. The success case does not imply soundness, but enables to prove the simulation rule compositionally. It prevents the case that all Boogie executions go to magic when a Viper program succeeds.

The following diagram demonstrates forward simulation $\mathsf{sim}(s_v, s_b)$ visually.

$$\begin{tikzcd}
\sigma_v \arrow[r, "s_v"] \arrow[d, dash, dotted, "R"'] & \sigma'_v \arrow[d, dash, dotted, "R"] \\
\sigma_b \arrow[r, "{[\![s_v]\!] = s_b}"] & \sigma'_b
\end{tikzcd}$$

In the diagram, s_v is a Viper statement, and $[\![s_v]\!]$ denotes the translation of s_v to Boogie by Viper's translational back-end. The diagram illustrates that if Viper and Boogie begin in the states σ_v and σ_b, respectively, with their states related by R, then their final states, σ'_v and σ'_b, will also be related by R. (The failure case is not depicted.)

The only work to validate a translation from Viper program s_v to Boogie program s_b is to prove $\mathsf{sim}(s_v, s_b)$. Previous work [11] already employs this method to validate the translation of a subset of Viper. Specifically, rather than proving the implementation of the verifier correct once and for all—which is impractical due to the complexity of the codebase—they developed an approach that automatically generates an Isabelle proof of $\mathsf{sim}(s_v, s_b)$ during every execution of the verifier. This certifies the Viper verifier on a per-run basis. We refer to this as the *proof generation* approach to certifying the Viper-to-Boogie translation.

In our work, we try to extend the method to support predicates. The next section shows an example of predicate translation and explains what additional properties need to be proven to make the simulation work.

6.2 An Example of Forward Simulation

Figure 6.2 shows a Viper source program and the translation of its method body (we omit the encoding of the pre-condition, which is encoded via inhale). The Viper program has two fields, `f` and `g`. The predicate `P` specifies full permission to `x.f` and its value to be 1. The method `m` requires `P(x)` and full permission to `x.g` as its pre-condition. In the Viper semantics, the precondition is expressed with an inhale at the beginning of the method body. In particular, the semantics of the method is equivalent to inhaling the precondition and then executing the body. The example only shows the translation of the method body.

In the Boogie encoding, the map `M` reflects the (direct) masks in the state model, i.e. the heap mask and the predicate mask. The map `H` corresponds to the total heap. Nested masks are not reflected in the encoding. The first statement in the Viper program is a field assignment. To encode this, the Boogie program first checks if the current state owns write (full) permission to the heap location. If there is insufficient permission, the program (both the Viper program and the Boogie program) results in a failure. Otherwise, the assignment is reflected in the total heap. This statement is itself irrelevant to predicates, but we will see how this statement affects the simulation of statements that follow it.

The second statement in the body of `m` is an unfold. To encode this, the Boogie program first checks if there is at least permission amount 1 to the predicate location `P(x)`. Then, it removes 1 at that location from the predicate mask, and *inhales* the body of `P(x)`. This is where the Viper semantics and the Boogie encoding differ substantially. (Recall that, in the unfold semantics defined in 5.3.3, we only

```
field f: Int
field g: Int
predicate P(x: Ref) { acc(x.f) && x.f == 1 }

method m(x: Ref)
    requires P(x) && acc(x.g)
{
    x.g := 2
    unfold P(x)
}
```

⇓

```
⟦ x.g := 2; unfold P(x) ⟧ =
```

```
// Translation of the body of method m:
    // x.g := 2
        assert M[x, g] == 1
        H[x, g] := 2
    // unfold P(x)
        assert M[null, P(x)] >= 1
        M[null, P(x)] := M[null, P(x)] - 1
        ⟦ inhale acc(x.f) && x.f == 1 ⟧
```

Figure 6.2: An example of back-end translation (simplified). On the last line of the Boogie program, instead of writing down the concrete Boogie encoding of inhale, we use a pair of brackets to denote the encoding of the corresponding Viper statement.

move the nested mask associated with `P(x)` one layer up instead of inhaling the predicate body.)

Proving the simulation of the statement "`x.g := 2`" is rather straightforward[2], since the encoding mimics the Viper semantics. In contrast, simulating the unfold operation defined in the semantics with an inhale is much less straightforward. Below, step by step, we explain what properties are needed in order to justify the

[2] To actually write the proof is still highly non-trivial, since it needs to deal with the differences between the state models where numerous low-level details need to be considered. Instead, we only focus on the high-level ideas here.

encoding of unfold.

1. **The nested mask of `P(x)` must contains what is added via the statement `inhale acc(x.f) && x.f == 1`.** In the semantics, the nested mask of `P(x)` is shifted up to acquire the new state where `P(x)` is unfolded. However, the translational Viper back-end uses inhale to encode unfold, so we must be able to show that, there exists an execution of inhale, such that the resulting state is exactly the same as the "shift up" operation we define for unfold. Using the diagram below, we can see how the simulation is achieved through an intermediate step.

$$\begin{tikzcd}
\sigma_v \arrow[r, "\texttt{unfold P(x)}"] \arrow[d, equal] & \sigma'_v \arrow[d, equal] \\
\sigma_v \arrow[r, "M := M\left\llbracket P(x) \overset{\times}{=} 1 - \frac{1}{M.2(P(x)).1}\right\rrbracket;\ \texttt{inhale body(P(x))}"] \arrow[d, dotted, no head, "R"'] & \sigma'_v \arrow[d, dotted, no head, "R"] \\
\sigma_b \arrow[r, "\texttt{M[null, P(x)] := M[null, P(x)]-1;}\ \llbracket\ \texttt{inhale body(P(x))}\ \rrbracket"] & \sigma'_b
\end{tikzcd}$$

The top line of the diagram is the actual unfold semantics defined by a shift-up operation. The bottom line is the Boogie encoding for unfold. The middle line can be seen as an intermediate step for the proof (decreasing the permission amount in the predicate mask and then inhaling the body). One contribution of the thesis is proving in the Viper semantics that the middle line has the same behavior as the first line. Then we do the normal simulation proof for inhale between the middle and the bottom line. This simulation should be straightforward since the Boogie encoding corresponds to the Viper code more closely. By combining these two steps, we establish a connection between the Viper semantics and the Boogie encoding.

To prove the connection between the top and the middle line, we need to relate what we store in the nested mask for a predicate to what is added to the state via inhale. The key to this connection is external consistency. Recall that external consistency intuitively means the nested mask stores what is specified in the predicate body. The nested mask for every predicate is externally consistent. In Section 6.3.3 we will show that, if a nested mask is externally consistent with respect to an assertion, then an inhale of the same assertion must be able to reach the same state. This property exactly corresponds to the property we need to prove the simulation of unfold statements.

2. **External consistency must hold at the point where unfold is executed**. Now, we know that if external consistency holds before unfold, we could use an inhale to simulate the behavior of unfold. The next question is, does external consistency hold for all nested masks at any point an unfold might occur?

   In the example program in Figure 6.2, we know that after inhaling the precondition of the method, the nested mask of `P(x)` is externally consistent (by the definition of inhale). However, between the inhale and the unfold, there might be arbitrary statements. These statements need to preserve state consistency. In fact, we are able to give an affirmative answer to this question: external consistency is preserved by all Viper statements. To prove this, we further require the following property.

3. **Internal consistency must hold between each statement**. In the example from Figure 6.2, the statement between the inhale and unfold is a field assignment. As motivated above, this assignment statement needs to preserve external consistency. However, since the assignment changes the heap, and external consistency depends on the heap to make sure the logical constraints of the predicate are satisfied, it is not obvious why external consistency is preserved. We thus must show that the bodies of all folded predicates do not depend on the value of the field we just modified.

   Actually, we need the property that statements preserve internal consistency to prove the property above. Now we provide an intuition as to why preservation of internal consistency is needed. In the field assignment statement, the state must own full permission to the heap location to make the execution go through. Otherwise, the execution fails without proceeding. Therefore, if we assume that the state before the assignment is internally consistent, permission to the same heap location cannot exist anywhere else. The other ingredient to prove the preservation of external consistency is the *self-framing* property of predicate bodies. An assertion is self-framing if it can be inhaled in any state without failure. This implies that if the assertion refers to the value of a heap location, the permission to this location must also be contained in the same assertion. For example, `acc(x.f) && x.f == 1` is self-framing, while `x.f == 1` is not. In a Viper program, we require all predicate bodies to be self-framing. Therefore, if a predicate mentions the value of `x.g`, it must contain some permission to `x.g`. Such a predicate cannot exist in an internally consistent state where the execution of an assignment to `x.g` succeeds, since the assignment already requires full permission to `x.g`.

Combining internal consistency with the self-framing restriction of Viper predicates, we are able to prove the preservation of external consistency. To guarantee internal consistency at any point of the execution, we need to ensure internal consistency is preserved by any statement. The property is formalized in Section 6.3.1.

These three properties (simulation of unfold with inhale, preservation of external consistency, and preservation of external consistency) are some fundamental properties that our semantics must satisfy in order to establish the simulation for unfold. It turns out that they are also sufficient to simulate other statements. Formally stating and proving these properties not only takes a solid step towards predicate translation validation, but also gives us confidence in the newly defined semantics.

6.3 Properties of the Semantics

This section formally states the three properties described in the last section, and gives a proof outline or intuition for each of them.

6.3.1 Preservation of Internal Consistency

This section discusses one property of the semantics: internal consistency is preserved by all statements. We formalize the property as follows.

Theorem 6.1 (preservation of internal consistency) *Let s be any Viper statement,*

$$\mathsf{intcons}\,(M) \implies S, H, M \xrightarrow{s} S', H', M' \implies \mathsf{intcons}\,(M')$$

Proof Outline The proof is done via an induction over s. Below, we only focus on proving for a subset of primitive statements (assignment, inhale, exhale, fold, and unfold).

- **Assignment** ($s = x := e$ **or** $s = e_1.f := e_2$). An assignment statement does not change the nested mask. Therefore, internal consistency is preserved.

- **Inhale** ($s =$ `inhale` A). The proof follows the definition of inhale. We explicitly use internal consistency to prune executions of inhale (see the rules InhaleAcc, InhaleAccWildcard, InhalePredicate, and InhalePredicateWildcard in Figure 5.2). Therefore, any state resulting from a successful transition must be internally consistent.

- **Exhale** ($s =$ `exhale` A). An exhale operation only makes the permissions stored in the nested mask smaller. By definition of the infinite sum, the state will own no more than the permission amount before the exhale for each heap location. Therefore, by definition, internal consistency is preserved.

- **Fold** ($s =$ `fold acc`$(P(\vec{v}), e)$). The semantics of fold first employs an exhale operation, and add the exhaled part to the nested mask of the predicate location. In the first step, suppose the nested mask of the state is M_e after the exhale. By definition of state subtraction (Section 5.3.4), we have

  $$(M \ominus M_e) \oplus M_e = M$$

  Using the property on infinite sum addition, we can prove

  $$\forall l.\ \exists s_p\ s_e.\ M \ominus M_e \xrightarrow{l} s_p \wedge M_e \xrightarrow{l} s_e \wedge M \xrightarrow{l} s_p + s_e \wedge s_p + s_e \leq 1$$

  The final nested mask of the fold operation, M', is given by

  $$M' \triangleq M_e \left[\!\!\left[P(\vec{v}) \stackrel{+}{=} (p, M \ominus M_e) \right]\!\!\right]$$

  where $P(\vec{v})$ and p are the evaluated predicate location and the permission, respectively, which are not relevant to internal consistency. Again, using the addition rule of infinite sum, we can prove that

  $$\forall l\ s_p\ s_e.\ M \ominus M_e \xrightarrow{l} s_p \implies M_e \xrightarrow{l} s_e \implies M' \xrightarrow{l} s_p + s_e$$

  As $s_p + s_e \leq 1$, the equation implies that M' is internally consistent.

- **Unfold** ($s =$ `unfold acc`$(P(\vec{v}), e)$). The proof of the unfold case is very similar to the fold case. We first subtract part of the nested mask from the whole nested mask, and then add it back to the nested mask. The sum of each heap location does not change during this procedure, according to similar reasoning as above. □

6.3.2 Preservation of External Consistency

All statements must preserve external consistency, as it is required to justify unfold. The property requires all predicates to be *self-framing*, which is defined below.

Definition 6.2 (self-framing) *An assertion A is self-framing, if and only if*

$$\forall S\ H\ M.\ \neg \left(S, H, M \xrightarrow{\texttt{inhale}\ A} \lightning \right)$$

The precise property of the preservation of external consistency is stated below.

Theorem 6.3 (preservation of external consistency) *Let s be any Viper statement. If all predicates are self-framing,*

$$H \models \mathsf{extcons}\,(M) \implies \mathsf{intcons}\,(M) \implies$$
$$S, H, M \xrightarrow{s} S', H', M' \implies H' \models \mathsf{extcons}\,(M')$$

Proof Outline Similar to the proof for internal consistency, we do a case split over the type of statement.

- **Field Assignment** ($s = e_1.f := e_2$). Note that in the theorem, we also require the initial state to be internally consistent. This is needed to prove the assignment case. If an assignment statement is successful, the state must own a write (full) permission to the heap location. Combined with the fact that the state is internally consistent, no nested mask may contain any permission to the same location. Since all predicate bodies are self-framing, we can prove that external consistency does not rely on the heap values of the locations with zero permission. Therefore, the modification of this heap location does not affect the external consistency of nested masks.

- **Inhale** ($s = \texttt{inhale}\ \texttt{acc}(P(\vec{v}), e)$). We only consider inhaling a predicate. Other cases are trivial.

  In the semantics for inhale, it constructs an externally consistent nested mask for the inhaled predicate location and adds it to the nested mask. This procedure only affects one predicate location, namely the inhaled one. Other nested masks are left untouched, so external consistency is preserved trivially for them. For the predicate location being inhaled, there are two cases to be considered.

1. **There was initially zero permission to this predicate location.** Then the inhale semantics guarantees that the final state is externally consistent by definition.

2. **There was already another nested mask stored in the state, representing the existing fraction of the predicate.** For example, the state might already own 1 permission of the predicate location p before we inhale another 0.5 permission of p. The semantics adds these two parts together. Now, we have that the existing nested mask M satisfies $\mathsf{extcons}_{1,p}\ (M)$, and the part we add to it satisfies $\mathsf{extcons}_{0.5,p}\ (M')$. We need to prove $\mathsf{extcons}_{1.5,p}\ (M \oplus M')$.

   We call this property the *combinability* [10] of external consistency:

   Lemma 6.4 (combinability of external consistency)

   $$\mathsf{extcons}_{p,P(\vec{v})}\ (M_1) \implies \mathsf{extcons}_{q,P(\vec{v})}\ (M_2) \implies \mathsf{extcons}_{p+q,P(\vec{v})}\ (M_1 \oplus M_2)$$

   It says that, if M_1 satisfies p amount of predicate $P(\vec{v})$ and M_2 satisfies q amount of predicate $P(\vec{v})$, the sum of the two nested masks satisfies $p + q$ amount of predicate $P(\vec{v})$. The proof has some low-level details, but otherwise quite straightforward. We omit its proof here.

- **Exhale** (s = `exhale acc`$(P(\vec{v}), e)$). Similar to inhale, we only need to discuss the exhaling of a predicate. According to the semantics, if there is p permission to the predicate location and the exhale operation removes q ($q \leq p$) permission to the location, then the nested mask is multiplied by $\frac{p-q}{p}$, and does non-deterministic assignment to those heap locations with zero permission.

  To prove that the first step preserves external consistency, we need to prove the *factorizability* [10] property.

  Lemma 6.5 (factorizability of external consistency)

  $$\mathsf{extcons}_{p,P(\vec{v})}\ (M) \implies \mathsf{extcons}_{q,P(\vec{v})}\left(M \overset{\times}{=} \frac{q}{p}\right) \quad (q \leq p)$$

The proof is carried out straightforwardly by an induction over the assertion. Similar to combinability, the proof has a number of low-level details but is otherwise unremarkable. We omit the proof here.

The preservation of the second step follows from the argument in the field assignment case (self-framing assertions cannot refer to heap locations that have zero permission).

- **Unfold** ($s = \texttt{unfold acc}(P(\vec{v}), e)$). According to the semantics, if there is p permission to the predicate location and the unfold operation specifies q ($q \leq p$) amount of permission, then the nested mask of the predicate location is multiplied by $\frac{p-q}{p}$, and $\frac{q}{p}$ of the nested mask is added to the top-level masks.

  The preservation proof for these two steps follows from factorizability and combinability, respectively.

- **Fold** ($s = \texttt{fold acc}(P(\vec{v}), e)$). The fold semantics exhales the predicate body from the state, and then adds the exhaled part to the corresponding nested mask. For the first step, we can prove

$$S, H, M \xrightarrow{\texttt{exhale}\ \mathsf{synmult}(p, P(\vec{v}))} S, H, M' \implies \mathsf{extcons}_{p, P(\vec{v})}\,(M \ominus M')$$

  which asserts that the part we exhaled is externally consistent with respect to p amount of $P(\vec{v})$. In the second step, this part is added to the corresponding nested mask, and the preservation of external consistency follows from combinability. □

6.3.3 Inhale Simulates Unfold

The semantics uses a shift-up operation on the nested mask to carry out an unfold operation. On the other hand, the back-end implementations usually inhale the predicate body to achieve the same operation. To formally connect the semantics and the implementation, we need a lemma to prove their equivalence.

The property is stated below.

Theorem 6.6

$$S, H, M \xrightarrow{\texttt{unfold}\ \texttt{acc}(P(\vec{v}), q)} S, H, M' \implies$$
$$S, H, M \left[\!\!\left[P(\vec{v}) \overset{\times}{=} 1 - \frac{q}{M.2(P(\vec{v})).1} \right]\!\!\right] \xrightarrow{\texttt{inhale}\ \mathsf{synmult}\ (q, \mathsf{body}(P(\vec{v})))} S, H, M'$$

The theorem says that, if the nested mask M transitions into M' through an unfold operation, we could first proportionally remove the nested mask of the predicate location from the state, and then inhale the body while reaching the same final state. Note that the inhale semantics might be non-deterministic and reach multiple other final states as well. This theorem only asserts the existence of the execution where the state M' is reached, and does not prevent the inhale from reaching other states. This is enough to prove the simulation, since back-end implementations will only have more non-determinism by using inhale, and it over-approximates the execution traces in the semantics.

The proof of this theorem uses the fact that the nested masks stored in the state are externally consistent, and thus reflect what predicate bodies mean. This proof essentially establishes the relationship between external consistency and the semantics of inhale. Similarly, the proof for the fold case connects external consistency with the semantics of exhale. By proving these properties, we achieve a stronger guarantee that various components in the semantics, while using different approaches to describe the permissions specified by an assertion, are equivalent to each other. This gives us confidence in the new semantics and serves as a basis for back-end validation.

In conclusion, this chapter introduces an approach to validate the translational back-end of Viper on a high level, and motivates several key properties of the new semantics. We have done some preliminary experiments on the validation of the actual back-end, and gained some evidence that the approach is feasible and the properties are correct. In this chapter, we only focus on the high-level motivations and ideas, and omit the exact way we validate the back-end, since the implementation is not complete at the time of writing. There is more work to do on the implementation side to fully validate the encoding of predicates.

Chapter 7

Conclusion

In this thesis, we present a novel semantics for Viper as our main contribution. This semantics introduces the nested mask state model to manage permissions owned by the state while preserving the structural information of predicate instances, which refines the state model of the purely isorecursive semantics. The semantics enables a distinction between internal and external consistency, making the semantics more intuitive to work with and better suited to explain certain Viper behaviors. Additionally, it supports permission introspection, which is not feasible in an equirecursive model. Our semantics is the first to bridge the gap between Viper and two distinct approaches to predicate interpretation.

Besides, we proved some important properties of this semantics. This gives us evidence that the semantics we define is correct and useful. We also did some preliminary experiments on using the semantics and its properties to validate the translational back-end of Viper, and show the feasibility of doing so by forward simulation. In summary, we believe that the semantics has the potential to validate back-end implementations.

7.1 Future Work

Mechanizing all technical results presented in this thesis. The full semantics is formalized in Isabelle/HOL. However, some properties we discussed in this thesis have not been fully proven in the interactive theorem prover. To eventually validate the back-end implementation, we need everything to be fully formalized and proven in Isabelle.

Certificate generation. As discussed in Chapter 6, the way we validate the translational back-end is by generating a certificate in Isabelle proving the forward simulation. We still need to extend the instrumentation of the verifier implementation to achieve this in an automated manner [11].

Simulation proof for exhale It is not clear how we should prove the simulation between the translation of an exhale statement and the exhale semantics we define. For exhale, the semantics non-deterministically assigns new values to all heap locations with zero permission. However, in practical verifiers, it is impossible to keep track of all locations and do the assignment precisely. Instead, the existing implementation keeps track *known-folded permissions* [18], the set of heap locations that are known to exist somewhere inside a predicate, and does non-deterministic assignment to all other locations. The set of known-folded permissions is a lower bound of the set of heap locations with some permission folded inside a predicate, so the verifier might perform non-deterministic assignments to more locations than the semantics. This behavior is sound, but to justify it we need to establish a formal relation between known-folded permissions and the state model in the semantics.

Validating function translation In the example in Section 3.1, we showed that the translation of functions and predicate is closely related, since predicate snapshots are used to derive information about the results of functions. How to formally connect the translation of functions to the semantics still needs to be explored.

Bibliography

[1] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer, 2016, pp. 41–62, ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2.

[2] *Viperproject/carbon*, Viper Project, Oct. 20, 2024.

[3] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, 147:1–147:30, OOPSLA Oct. 10, 2019. DOI: 10.1145/3360573.

[4] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular Specification and Verification of Go Programs," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 367–379, ISBN: 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_17.

[5] M. Eilers and P. Müller, "Nagini: A Static Verifier for Python," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 596–603, ISBN: 978-3-319-96145-3. DOI: 10.1007/978-3-319-96145-3_33.

[6] T. Dardinier, A. Li, and P. Müller, "Hypra: A Deductive Program Verifier for Hyper Hoare Logic," *Hypra: A Deductive Program Verifier for Hyperproperties (artifact)*, vol. 8, 316:1279–316:1308, OOPSLA2 Oct. 8, 2024. DOI: 10.1145/3689756.

[7] J. Smans, B. Jacobs, and F. Piessens, "Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic," in *ECOOP 2009 Object-Oriented Programming*, S. Drossopoulou, Ed., Berlin, Heidelberg: Springer, 2009, pp. 148–172, ISBN: 978-3-642-03013-0. DOI: 10.1007/978-3-642-03013-0_8.

[8] J. Boyland, "Checking Interference with Fractional Permissions," in *Static Analysis*, R. Cousot, Ed., Berlin, Heidelberg: Springer, 2003, pp. 55–72, ISBN: 978-3-540-44898-3. DOI: 10.1007/3-540-44898-5_4.

[9] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, no. 1, pp. 259–270, Jan. 12, 2005, ISSN: 0362-1340. DOI: 10.1145/1047659.1040327.

[10] T. Dardinier, P. Müller, and A. J. Summers, "Fractional resources in unbounded separation logic," *Proceedings of the ACM on Programming Languages*, vol. 6, 163:1066–163:1092, OOPSLA2 Oct. 31, 2022. DOI: 10.1145/3563326.

[11] G. Parthasarathy, T. Dardinier, B. Bonneau, P. Müller, and A. J. Summers, "Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language," *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language – Artifact*, vol. 8, 208:1510–208:1534, PLDI Jun. 20, 2024. DOI: 10.1145/3656438.

[12] C. Zimmerman, J. DiVincenzo, and J. Aldrich, "Sound Gradual Verification with Symbolic Execution," *Proc. ACM Program. Lang.*, vol. 8, 85:2547–85:2576, POPL Jan. 5, 2024. DOI: 10.1145/3632927.

[13] T. Dardinier, M. Sammler, G. Parthasarathy, A. J. Summers, and P. Müller. "Formal Foundations for Translational Separation Logic Verifiers (extended version)." arXiv: 2407.20002 [cs]. (Jul. 29, 2024), [Online]. Available: http://arxiv.org/abs/2407.20002 (visited on 09/25/2024), pre-published.

[14] S. Willard, *General Topology*. Courier Corporation, Jan. 1, 2004, 384 pp., ISBN: 978-0-486-43479-7.

[15] J. B. Conway, *A Course in Functional Analysis* (Graduate Texts in Mathematics). New York, NY: Springer, 2007, vol. 96, ISBN: 978-1-4757-4383-8. DOI: 10.1007/978-1-4757-4383-8.

[16] K. R. M. Leino, "This is Boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[17] G. Parthasarathy, P. Müller, and A. J. Summers, "Formally Validating a Practical Verification Condition Generator," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 704–727, ISBN: 978-3-030-81688-9. DOI: 10.1007/978-3-030-81688-9_33.

[18] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers, "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions," in *ECOOP 2013 Object-Oriented Programming*, G. Castagna, Ed., Berlin, Heidelberg: Springer, 2013, pp. 451–476, ISBN: 978-3-642-39038-8. DOI: 10.1007/978-3-642-39038-8_19.

Appendix A

Isorecursive Semantics with Naive Consistency

In this section, we try to equip the purely isorecursive semantics (Section 3.1) with consistency.

We still use only two masks, the heap mask and the predicate mask, and define consistency based on them. We do not store any information about the predicate bodies in the state, but rather keep them as additional information that we may access at any time.

The state model in such a semantics is a 3-tuple (H, M_h, M_p), where H is the total heap, M_h is the heap mask, and M_p is the predicate mask with the following types.

$$\begin{aligned} H &: L \rightarrow V \\ M_h &: L \rightarrow [0, 1] \\ M_p &: P \rightarrow [0, +\infty) \end{aligned}$$

The notion of consistency is defined based on the state tuple (H, M_h, M_p), with access to predicate bodies. To define the consistency, we need to account for both permissions in M_h, and any permission folded in M_p. Without defining the formal semantics of unfold, we give the following inductive definition for consistency[1].

$$\textsc{Zero}\quad \frac{\forall\, l.\ M_h(l) \leq 1}{\mathsf{consistent}_0\ (H, M_h, M_p)}$$

$$\textsc{Step}\quad \frac{\begin{array}{c}\forall\, l.\ M_p(l) > 0 \implies \\ \big[\mathsf{unfold}(H, M_h, M_p, l) \rightarrow (M_h', M_p')\big] \land \mathsf{consistent}_n\ (H, M_h', M_p')\end{array}}{\mathsf{consistent}_{n+1}\ (H, M_h, M_p)}$$

$$\mathsf{consistent}\ (H, M_h, M_p) \equiv \forall\, n.\ \mathsf{consistent}_n\ (H, M_h, M_p)$$

The intuition of the definition is that, we can unfold "all" predicates stored in the state, and the resulting state is still consistent. However, since Viper supports quantified permissions, the number of predicates in a state may be infinite, and we can never fully unfold all predicates in a finite sequence. Therefore, we define the consistency as "consistent through any number of unfolding". Since a program can only unfold a finite number of predicates, the definition suffices.

This definition, while correct, can be very difficult to work with. Two primary reasons are listed below.

The consistency definition is unstructured. The definition of consistency unfold the predicates in an arbitrary order. This has an important implication—it is hard to prove even the simplest properties about consistency. One property we might want to prove is that consistency is preserved by fold. This property is intuitively correct, because permissions are not created or destroyed when folding a predicate. All we do is wrapping the permissions into a predicate instance, and the total amount of permissions should not change. However, to prove the consistency of the state after the fold, we do not have to unfold the newly folded predicate first because of the universal quantifier in the definition. As a result, a straightforward induction does not provide a strong enough induction hypothesis for the proof. As a result, proving properties on this definition is extremely challenging.

[1] Here, we assume that unfold is deterministic. In other words, we ignore Viper features such as wildcard permissions. As we will see, defining consistency via unfold is already cumbersome, and non-determinism makes it even harder.

Supporting wildcard permissions becomes harder. In the definition above, we assume that unfold is deterministic, which means a predicate can have exactly one representation. However, if we consider wildcard permissions as well, unfolding a predicate could lead to many different resulting states (since the wildcard amount is not known in the state model). In such a case, consistency would have to be generalized to state that there is at least one possible wildcard amount for each considered unfolding such that one does not exceed full permission for any heap location. This adds another layer of difficulty to the definition and its usage.

Appendix B

Properties of the Infinite Sum

Section 4.4.2 defines the infinite sum to formalize internal consistency. Below, we list some properties of the infinite sum without proof. These lemmas are used to prove the properties of the semantics in Section 6.3. Additionally, there is an Isabelle/HOL library that includes these properties, which we utilize in our Isabelle mechanization.

Lemma B.1 (uniqueness of the infinite sum) *If $f \longrightarrow s_1$ and $f \longrightarrow s_2$, then $s_1 = s_2$.*

Lemma B.2 *If $f \longrightarrow s_1$ and $g \longrightarrow s_2$, then $f + g \longrightarrow s_1 + s_2$, where $f + g$ is defined as $(f + g)(x) = f(x) + g(x)$.*

Lemma B.3 *If $f \longrightarrow s$, then $-f \longrightarrow -s$, where $-f$ is defined as $(-f)(x) = -f(x)$.*

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

● I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

A Semantics for Predicates in Automated Separation Logic Verifiers

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Xiao	Yushuo

With my signature I confirm the following:

- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich, 21.10.2024	Yushuo Xiao

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard