Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Extending the Viper Verification Language with User-Defined Permission Models

Matthias Roshardt

September 16, 2021

Advisors: Thibault Dardinier, Prof. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

A number of formal methods can be used to verify the adherence of a program to its specification, and thus the absence of bugs. In the wake of the introduction of separation logic into the field, a new generation of automated verification tools were developed. A common feature of such verifiers is the presence of permission models, allowing program locations to hold differing quantities, called shares, of heap-based resources. Permission models are a core part of the verifier's internal logic, placing constraints on what can and cannot be inferred about a program. Viper is an intermediate language and verification toolchain, comprising a number of front-ends for various high-level programming languages. It uses an early model involving fractional permissions, which has a number of undesirable limitations. This thesis provides a survey of currently studied permission models and generalises Viper's implementation, so as to allow for the introduction of arbitrary permission models. The language is extended to include a user interface through which such models can be axiomatically specified. We explore Viper's requirements towards such axiomatisations, and provide axiom sets for selected permission models.

Acknowledgments

I would like to warmly thank my supervisor, Thibault Dardinier, for his consistent support. It was truly wonderful to be able to rely on his sense of direction when things were going well and his patience when progress was slow. His infectious enthusiasm for the project gave me the motivation and drive I needed to put together the thesis you are now reading.

Special thanks must also be extended to Gaurav Parthasarathy and Marco Eilers, who were always quick to help me out with their extensive knowledge of the Carbon codebase. Without them, these last months would have been a lot more confusing than they already were.

Additionally, I would like to thank Professor Peter Müller for leading such a welcoming and mutually supportive research team and allowing me to be a part of it for these six months. I am also grateful to have had him as a teacher; after all it was his undergraduate course on formal methods that has sparked my enduring interest in the field.

Lastly, but with particular emphasis, I would like to thank my fiancée for the innumerable ways in which she supported me during what was both academically and personally a challenging period of my life.

Contents

Chapter 1

Introduction

1.1 Motivation

Software bugs are a well-known plight of computer programmers. Typically, testing is employed to stem the tide of unintended program behaviour. However, as Edsger W. Dijkstra famously put it, testing can only show the presence of bugs, but never their absence. This caveat may be unproblematic for most software, but safety-critical software in aviation, public infrastructure or finance dictate a higher degree of correctness. Formal verification techniques may be employed to prove the absence of errors by providing a specification for the program which is then verified. While the formal techniques have been around since the 70s, there was for a long time a gap between the kind of program that could feasibly be encoded in a rigorous logical framework and the kind of software one would encounter in real world applications. Before the advent of separation logic [1] at the turn of the millennium, reasoning about programs operating on mutable, potentially unbounded datastructures such as trees or linked lists, particularly in the presence of reference aliasing and concurrency, required extremely complex and lengthy proofs. In a nutshell, separation logic enables *local reasoning*, making it possible to ignore parts of the global program state not relevant to a particular program section. Since that pioneering theoretical work, a number of automated verification tools for just such situations have been developed [2][3][4][5].

Modern verification frameworks such as Viper [2] provide light-weight syntax for encoding the desired program specifications in the form of pre- and postconditions for methods and a number of auxiliary constructs such as loop invariants and assertions. An essential feature of program specifications in separation logic based verifiers is the handling of *permissions*. In Viper, reading a value from the heap requires one level of permission, whereas writing requires another. Using this form of permission accounting, we can

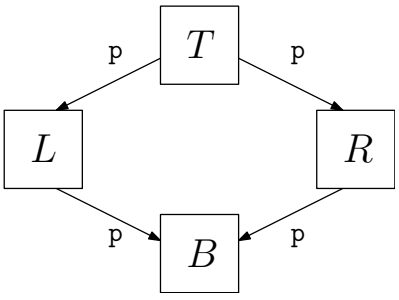
```
0  field left: Ref
1  field right: Ref
2  field elem: Int
3
4  predicate Tree(this: Ref) {
5      this != null ==> acc(this.elem) && acc(this.left) &&
6      acc(this.right) && Tree(this.left) && Tree(this.right)
7  }
8
9  method m(x: Ref, p: Perm)
10     requires p > none && acc(Tree(x), p)
11 {
12     unfold acc(Tree(x), p)
13     if (x != null && x.left != null && x.right != null) {
14         unfold acc(Tree(x.left), p)
15         unfold acc(Tree(x.right), p)
16         assert x.left.right != x.right.left//Why does this fail?
17     }
18 }
```

Listing 1.1: Tree predicates exhibiting unexpected behaviour

directly prove the absence of data races, resource leaks and all manners of unintended side effects.

Needless to say, the way permissions are implemented in a verifier has a large influence over how a specification must be written. Viper uses fractional permissions, that is, a program thread can hold a value between 0 and 1 to a particular heap location. A full permission of 1 is required to assign to the location, but only a non-zero value is required for read access. This model is attractive because of its theoretical simplicity and straightforward embedding into the SMT solver powering the verification process. However, it is also completely unsuitable for other scenarios. Essential concepts from concurrent programming such as semaphores and spin-locks are extremely cumbersome to verify under this model. It has been shown to be possible [6], but it required the use of inelegant and essentially unsound workarounds to achieve.

Readers familiar with Viper will also recognise the example (Listing 1.1) of tree predicates, which unexpectedly describe DAGs but only if a permission of less than $1/2$ is held. To see why this is true, consider the scenario on the right, where we have four distinct tree nodes starting with T, whose left and right children are L and R, and B which is the right child of L and the left child of R. Now, if we have permission p to



the resource `Tree`(T), then we can infer that
we have permission `p` to both L and R, and the same permission to their respective children. However, if `p` $\leq 1/2$, L and R might actually share a child (B in our example), to which we would have permission `p + p`. Thus, if we actually wanted to describe trees, we would have to either ensure that the predicate is always used with a permission of more than $1/2$, or make explicit the requirement that descendants of a node be disjoint, severly complicating the predicate's definition.

What this discussion boils down to is the recognition that different permission models are required in different situations, and one solution can never cover every possible use case. For this reason, it would be desirable to have the option of switching out permission models according to the user's needs. Viper provides an interesting target for such a development because of its wide array of front-ends for modern programming languages and its extensive feature set which this new solution must fit into. While much of this thesis draws on theoretical work laid out by Dockins et al. [7], and multiple verifiers, such as HIP/SLEEK [3] or Caper [4], have automated a number of the permission models discussed in this thesis, it is the unique adaptations required by the Viper infrastructure and the requirement for a user interface that define the contributions presented hereafter.

1.2 Thesis Outline

The structure of this thesis is as follows: Chapter 2 introduces the necessary theoretical background on separation algebras and provides a brief overview of the Viper infrastructure and how its permission model is implemented. Chapter 3 explores necessary and sufficient criteria for correctly axiomatizing new permission models in Boogie. It also addresses Viper features which are incompatible with generalised permission models, such as predicates and wildcards, and proposes alternative implementation mechanisms. Chapter 5 presents axiom sets for selected separation algebras including soundness proof outlines. Finally, in Chapter 6 we introduce and discuss the implementation of an interface which enables users to specify custom permission models.

Chapter 2

Background

The purpose of this chapter is to provide the reader with the necessary theoretical and technical background for the main contributions of this thesis described in later chapters. We begin with a cursory review of separation logic with particular emphasis on the kinds of separation algebras that will be implemented. We then provide a basic overview of the Viper verification toolchain which will be the target for the automation of new permission models. Finally, we take a brief look at some relevant implementation details of the Carbon back-end as well as Boogie, the intermediate verification language it targets.

2.1 Separation Logic

Much can be said about separation logic, certainly too much for the scope of this thesis. The point of this section is to provide the reader with the gist of how separation logic works, and how permissions and separation algebras fit into the picture. Our discussion remains informal on purpose; Viper's semantics are as of yet not rigorously formalized, so there is little point in introducing a bunch of formalisms in this section that have no follow-up in later parts. To the interested reader we recommend the excellent introductory text by O'Hearn [8].

2.1.1 Intuition and History

Hoare's logic [9][10] is perhaps the most commonly known formal system for reasoning about imperative programs. It is a logical calculus that operates over statements of the form $\{P\}C\{Q\}$ commonly called *Hoare triples*, which express the notion that executing a program C in a state satisfying the assertion P will result in a state which satisfies the assertion Q. States in Hoare's logic are assignments of values to program variables, which we will call a *store*. As useful as this logic is for verifying the correctness of

simple programs, its formalism rapidly becomes unwieldy when dealing with programming languages that operate not only on local variables but values in heap memory addressable by references, which includes essentially all modern imperative programming languages. The issue in a nutshell is what is known in AI research [11] as the frame problem: given a procedure which operates on the heap, it is very difficult to formally express the notion that all heap values not 'touched' by a procedure remain the same in the post-state as in the pre-state. What we would ideally want is the ability to 'split off' the part of the state which the procedure needs and prove that it fulfils its specification on that partition of the state. From that, we can conclude that it also fulfils its specification in the presence of additional heap values (which it does not interact with).

What seems like a blatantly obvious notion now was formally captured only in the last 20 years. Based on earlier work on the logic of bunched implications [12][13][14], Reynolds, O'Hearn, Yang and Ishtiaq developed separation logic at the turn of the millenium [1][15]. Separation logic has since spurred a host of interesting developments [16][17][18][19] in the field of program verification including numerous automated verifiers [20][21][22], one of which is Viper [2].

2.1.2 Basic Definitions

At its core, separation logic is an extension of Hoare's logic comprising three essential additions: points-to assertions (which allow us to make statements about heap values), separating conjunctions (which compose points-to assertions) and the frame inference rule (which is the solution to the problem posed in the previous section), each of which we introduce in turn.

Heaps and Points-to Assertions

Compared to Hoare's logic, where the state is comprised of a variable store (which maps variables to values), the state in separation logic is a (store, heap)-pair $\langle s, h \rangle$. We define heaps as follows:

Definition 2.1.1 (Heap)**.** Let L be a set of heap locations and V be a set of values. A heap is a partial map $h : L \hookrightarrow V$. Furthermore, let $dom(h) \subseteq L$ define the set of locations mapped by h. For a location $l \in \mathrm{dom}(h)$, we denote the associated value using the familiar function syntax $h(l)$.

One particular heap is so common, we will dedicate a symbol to it:

Definition 2.1.2 (Empty Heap)**.** The empty heap $\square$ is the unique heap h satisfying $\mathrm{dom}(h) = \emptyset$.

We leave the definition of the set of locations L and values V deliberately vague, as there are many possibilities: in Viper for instance, locations are pairs of references and field identifiers (glossing over predicates and wands), whereas in the original paper [1], locations were natural numbers, enabling the use of C-like address arithmetic.

We can now bring points-to assertions into the picture:

Definition 2.1.3 (Points-to Assertion)**.** Let $l \in L$ be a location and $v \in V$ be a value. A state (s, h) satisfies the points-to assertion $l \mapsto v$, denoted $(s, h) \models l \mapsto v$, if and only if dom$(h) = \{l\}$ and h(l) = v.

Put prosaically, $l \mapsto v$ holds if and only if the value v is at the address l and *nothing else* is in 'memory'. We again define a special symbol for the empty heap assertion:

Definition 2.1.4 (Empty Heap Assertion)**.** We define **emp** to be the unique assertion where $(s, h) \models$ **emp** if and only if $h = \square$.

Of course, we must provide axioms that allow us to introduce and eliminate these assertions. Formulations of separation logic usually comprise new and dispose operations, with definitions like the following[1]:

$$\{\textbf{emp}\}x := \mathsf{new}\ (E)\{x \mapsto E\}$$
$$\{x \mapsto \_\}\mathsf{dispose}\ x\{\textbf{emp}\}$$

So far, so good; we have heaps and assertions that can only be satisfied by singleton heaps. These are the basic blocks which we can use to build more complex assertions. Towards this end, we introduce *composition* of two heaps, which is possible if and only if their domains (mapped locations) are disjoint.

Definition 2.1.5 (Heap Composition)**.** Let h_1, h_2 be heaps. The symbol $\star$ denotes heap composition and is defined as:

$$(h_1 \star h_2)(l) = \begin{cases} \textbf{undefined}, & \text{if } dom(h_1) \cap dom(h_2) \neq \emptyset \\ h_1(l), & \text{if } l \in dom(h_1) \\ h_2(l), & \text{if } l \in dom(h_2) \end{cases}$$

Corollary 1. *From the above definition immediately follows that heap composition $\star$ is associative and commutative:*

$$h \star g = g \star h \tag{2.1}$$
$$f \star (g \star h) = (f \star g) \star h \tag{2.2}$$

[1]The expression $x \mapsto \_$ matches any assertion with location x.

Furthermore, we observe that $\star$ exhibits the cancellative property:

$$h_1 \star g = h_2 \star g \to h_1 = h_2 \tag{2.3}$$

Or, in plain English, we see that given a composed heap and one of its constituent heaps, we can uniquely determine the other constituent heap (a procedure to achieve this is as simple as set subtraction).

Corollary 2. *We observe that the* empty heap $\square$ *acts as the unique*[2] *identity element with respect to the algebra $\langle L \hookrightarrow V, \star \rangle$; that is, for all heaps h we have*

$$h \star \square = \square \star h = h. \tag{2.4}$$

Thus, heaps under composition form a partial (commutative) monoid.

One may rightly wonder why we zero in on such seemingly inconsequential properties of the composition operator. For now, let us put a pin in this discussion and return to it later.

Separating Conjunction

We will now use the above definition to manifest the second component of separation logic:

Definition 2.1.6 (Separation Conjunction)**.** Given formulas P and Q, the separating conjunction $P{*}Q$ is a formula that is satisfied by a state (s, h) if and only if there exist heaps h_1, h_2 such that all of the following holds:

1. $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$
2. $h = h_1 \star h_2$
3. $(s, h_1) \models P$
4. $(s, h_2) \models Q$

Again in plain English, this definition says that $P * Q$ is satisfied if there exist *separate* heap portions that can individually satisfy the left and right component of the conjunction. To induce a complete logic, more components are needed of course. One approach is to add an assertion false that is by definition insatisfiable, as well as the connective $\Rightarrow$, with the semantics that $P \Rightarrow Q$ is satisfied by a state (s, h) if when (s, h) satisfies P then it also satisfies Q.

The rest then falls into place:

$$\neg P \stackrel{\text{def}}{=} P \Rightarrow \mathsf{false}$$

[2]This is ensured by the cancellation property.

$$P \wedge Q \stackrel{\text{def}}{=} \neg(\neg P \Rightarrow Q)$$

$$P \vee Q \stackrel{\text{def}}{=} P \Rightarrow \neg Q$$

Without getting too lost in the weeds, we would like to point out some of the similarities and differences between the separating conjunction $*$ and logical conjunction $\wedge$. For one, recognise that for heap-independent statements, $*$ and $\wedge$ are interchangeable. But with points-to assertions, where $(l \mapsto v) \wedge (l \mapsto v)$ is the same as $l \mapsto v$, $(l \mapsto v) * (l \mapsto v)$ is insatisfiable. On the other hand, $(l \mapsto v) * \neg(l \mapsto v)$ is satisfiable while $(l \mapsto v) \wedge \neg(l \mapsto v)$ is not.

Magic Wands As a final remark, consider the deduction theorem which states that[3]:

$$A \wedge B \vdash C \text{ if and only if } A \vdash B \Rightarrow C \tag{2.5}$$

In some sense, this commonly known theorem relates implication with conjunction. In the logic of bunched implications [13], there is an analogue of implication for separating conjunctions. It is defined as follows:

Definition 2.1.7 (Separating Implication)**.** Let P, Q be assertions. Then their *separating implication* $P \mathrel{-\!\!*} Q$ is satisfied by a state (s, h) if and only if for all heaps h' where $h \cap h' = \emptyset$, $\langle s, h' \rangle \models P$ implies $\langle s, h' \star h \rangle \models Q$.

On its own, this definition seems somewhat opaque, but the point is that whereas $\Rightarrow$ is the adjoint of $\wedge$, the separating conjunction $\mathrel{-\!\!*}$ is the adjoint of $*$, i.e.

$$A * B \vdash C \text{ if and only if } A \vdash B \mathrel{-\!\!*} C \tag{2.6}$$

The $\mathrel{-\!\!*}$ operator is commonly referred to as the *magic wand* in the literature. Its semantics and applications go beyond the scope of this short introduction; we refer to Ishtiaq and O'Hearn's paper [13] for details.

Frame Rule

Let us for a moment recall what one of the motivations behind separation logic was in the first place. Suppose we have some procedure `foo`, which has precondition P and postcondition Q and which takes a single reference as an argument. Suppose now that we are trying to verify a specification for some procedure which in turn calls `foo`, which might take the form of the Hoare triplet $\{P * R\}\texttt{foo(x)}\{Q * R\}$ for some variable x. The point is: we might have more information about the heap (in the form of R) than

[3]The symbol $\vdash$ represents syntactic consequence, $A \vdash B$ holds when B can be derived from A in the logical system.

the procedure `foo` needs, which means its precondition cannot be matched against the pre-state. The frame rule states that this is indeed possible:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\mathsf{mod}(C) \cap \mathsf{fv}(R) = \emptyset \tag{2.7}$$

The side condition states that C must not modify any free variable in R. There are a few different readings of this inference rule. One is to say that if a program C executes correctly in a state satisfying P, then C also executes correctly in a 'larger' state satisfying $P * R$. Or inversely, if a program C operates in an isolated (separate) region of the heap, then the other values remain unchanged.

The frame rule allows for much more succinct proofs compared to using basic Hoare logic. It is also of particular interest in the presence of concurrent execution, usually denoted $C_1||C_2$. Applying the same notion, we can posit what is known as the *concurrency rule*:

$$\frac{\{P_1\}C_1\{Q_1\} \ \ldots \ \{P_n\}C_n\{Q_n\}}{\{P_1 * \ldots * P_n\}C_1|| \ldots ||C_n\{Q_1 * \ldots * Q_n\}} i \neq j \rightarrow \mathsf{mod}(C_i) \cap \mathsf{fv}(P_j \vee Q_j) = \emptyset \tag{2.8}$$

That is, if different threads operate on separate parts of the heap, they can be verified independently as they cannot interfere with one another.

2.1.3 Ownership Semantics

The next historical evolution of separation logic [23][16][17] is based on the following observation: If the assertion $l \mapsto v$ holds, we can be sure that there is no other part of the program that can access or even change $l \mapsto v$. Thus, the points-to assertion can be read as a declaration of *ownership* of the location l. This change of perspective leads to more interesting questions: If we consider a location l to be a resource which can be owned, is there some reasonable semantics for safely sharing the resource? This question is of particular interest in the context of concurrency. We encounter many situations where it is desirable to have multiple threads read from the same locations, however, this is only safe if no other location writes to it during that time. We will call such systems of ownership management *permission models*. In the following, we will formally introduce this notion, and then discuss desirable properties for those models as in the context of program verification.

Basic Definitions

First, we would like to draw an important distinction. What we want is to somehow 'split up' or *separate* an assertion $l \mapsto v$ into parts. The part we give to a program depends on what kinds of operations it can perform on the heap

location l. We use two terms in the following discussion: a *share* is the word for a part of the resource l, and a *permission* is the class of privileges that are associated with owning a particular share. The map assigning permissions to shares is called a *policy*. In the following, if nothing else is stated, $\mathbb{P}$ will denote the set of permissions in a model, whereas $\mathbb{S}$ denotes the set of shares. Finally, $\Pi : \mathbb{S} \to \mathbb{P}$ will denote a model's permission policy.

Permission Sets Let us first consider what kinds of permissions we want to be able to issue. In basic separation logic, we functionally have two permissions: we either have the assertion $l \mapsto v$, giving us the freedom to read from or assign to l, or we don't, and thus don't even know about the value v. A common extension, and the one employed by Viper, is to add a read-only permission to this set, allowing the use of the value stored at l, but not overwriting it. Because we will frequently refer to these permissions, we assign symbols to them: $\bullet$ to denote the full permission, $\mathbin{\rotatebox{90}{◐}}$ to denote the read-only permission and $\circ$ to denote the empty permission. However, additional permissions and context-dependent policies have been explored in the literature [24][25][26]. We will try to keep the discussion as general as possible, but do assume a static permission policy, where we use the symbols $\sigma_\bullet, \sigma_{\mathbin{\rotatebox{90}{◐}}}, \sigma_\circ$ to refer to an arbitrary share that is mapped to a full, read-only or empty permission respectively.

Share Sets Next, we define how the resource l can be split into shares. What we want is some set $\mathbb{S}$ with some operation $\texttt{split} : \mathbb{S} \to \mathbb{S} \times \mathbb{S}$. Of course, it must be possible to recombine the 'pieces' of the original resource, which is the inverse operation of the split. We call this operation the *join* and denote it with the symbol $\oplus$. We call the pair $\langle \mathbb{S}, \oplus \rangle$ a *share algebra*. A share algebra $\langle \mathbb{S}, \oplus \rangle$ combined with a set of permissions $\mathbb{P}$ and a policy $\Pi : \mathbb{S} \to \mathbb{P}$ is called a *permission model*.

2.1.4 Introducing permissions to the logic

As a first step, we augment the definition of a heap: a heap is now not just a finite partial map from locations to values, each value now has an associated share, that is heaps have the signature $L \hookrightarrow V \times \mathbb{S}$. We overhaul the points-to assertion also: let $(s, h) \models l \mapsto_\sigma v$ if and only if $\mathrm{dom}(h) = \{l\}$ and $h(l) = (v, \sigma)$.

Now, we introduce the following postulate allowing the combination of shares (either as an axiom or inference rule):

$$l \mapsto_{\sigma_1} v_1 * l \mapsto_{\sigma_2} v_2 \iff l \mapsto_\sigma v \text{ iff. } \sigma_1 \oplus \sigma_2 = \sigma \text{ and } v = v_1 = v_2 \tag{2.9}$$

That is, points-to assertions can be split and combined according to the share algebra $\langle \mathbb{S}, \oplus \rangle$ as long as both the locations and values of the two assertions

are identical (or at least provably equal). Of course, an empty permission is akin to having no assertion at all, thus we have for all shares $\sigma_{\bigcirc}$ with $\Pi(\sigma_{\bigcirc}) = \bigcirc$:

$$l \mapsto_{\sigma_{\bigcirc}} v \iff \mathbf{emp} \tag{2.10}$$

There are a few properties we would like to hold in the permission model:

1. The share join operation $\oplus$ should be associative and commutative. This agrees with our natural understanding of how a resource behaves: If a pile of things is divided into parts, it should not matter in which order the pieces are recombined.
2. Adding any empty share to another share[4] should not give a different permission, i.e. the empty permission is the identity element. This again follows from a natural viewpoint: if I add an empty pile to an existing pile, I should get that same pile.
3. The cancellative property should hold in the share algebra. This again motivated by the resource analogy: if we take away something from a pile, the amount remaining is uniquely defined by what we started with and what we took away. This property implies that the empty share must be unique.
4. It must not be possible to add any non-empty permission to a full permission, i.e.

   $$\forall \sigma_{\bullet}, \sigma_1, \sigma_2. \Pi(\sigma_{\bullet}) = \bullet \longrightarrow \sigma_{\bullet} \oplus \sigma_1 = \sigma_2 \longrightarrow \Pi(\sigma_1) = \bigcirc.$$

   The reasoning for this is as follows: Suppose σ_1 maps to a read permission. Then we would get some share σ_2 which can be split into a read and a write permission. Since we never want to have a read and a write permission at the same time (because this leads to data races), such an object should not exist. Analogous reasoning can be employed for the case where $\Pi(\sigma_1) = \bullet$.

It does not take long to see that the first three properties are identical with the ones laid out for the algebra over heaps. This similarity has first been stated by Calcagno, O'Hearn and Yang [27], who also proposed a unified heap and permission model based on the notion of a *separation algebra*, which we will present in the following.

2.1.5 Separation Algebra

The following definitions are courtesy of Dockins, Hobor and Appel [7]. We choose to reprint their formulation due to its theoretical elegance and clear

[4] By 'empty share' we mean 'share which maps to the empty permission'.

exposition. Though the authors of that paper propose a separation logic entirely induced by a separation algebra, we only pick out the parts that are relevant for permission models, as the logic encoded by Viper is already given.

Definition 2.1.8. A separation algebra $\langle A, \oplus, u \rangle$ consists of a set A, a *join operation* $\oplus$, and the identity element (often called the *unit element*) $u \in A$, such that:[5]

$$a \oplus b = b \oplus a \qquad \text{(commutative)}$$

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \qquad \text{(associative)}$$

$$a \oplus u = a \qquad \text{(identity)}$$

$$a_1 \oplus b = a_2 \oplus b \longrightarrow a_1 = a_2 \qquad \text{(cancellative)}$$

That is to say, a separation algebra is a partial, cancellative, commutative monoid.

Auxiliary Definitions

Definition 2.1.9 (Definedness Relation)**.** Since the join operation $\oplus$ is partial, it will be useful to have a shorthand for stating that the result of a join operation is defined. W will thus use $a \oplus_{\downarrow} b$ to denote the fact that $a \oplus b$ is defined.

Definition 2.1.10 (Inverse Join)**.** We define for each separation algebra the (partial) inverse join operation $\ominus$, defined using the following axiom:

$$a \oplus b = c \longrightarrow c \ominus b = a \tag{2.11}$$

Definition 2.1.11 (Ordering Relation)**.** For each separation algebra we define a preorder (i.e. a reflexive and transitive relation) $\preceq$, defined as:

$$a \preceq b \iff \exists x.\, a \oplus x = b. \tag{2.12}$$

This ordering becomes important later on, because it is the analogue of $\oplus_{\downarrow}$ with respect to the inverse join operation.

2.1.6 Desirable Properties of Separation Algebras

Apart from the definitions provided in the previous section, there are a number of additional properties that an 'ideal' separation algebra should have. These definitions are again due to Dockins et al. [7]. We also refer to that paper for detailed justifications of the properties outlined below.

[5] Free variables are $\forall$-quantified unless stated otherwise.

Positivity

A separation algebra $\langle A, \oplus, \mathsf{u} \rangle$ satisfies positivity if and only if:

$$a \oplus b = \mathsf{u} \longrightarrow a = \mathsf{u} \tag{2.13}$$

That is, the unit cannot be split into non-unit elements. This is an essential property, as without it, we could produce a derivation:

$$\{\mathbf{emp}\} \vdash \{l \mapsto_{\mathsf{u}} v\} \vdash \{l \mapsto_{\sigma} v * l \mapsto_{\mathsf{u} \ominus \sigma}\}$$

Whereby we obtain a non-empty permission from an empty permission for a location l and value v of our choosing. We could then use the frame rule to get rid of the separating conjunction and prove unintuitive results.

Infinite Split

A separation algebra $\langle A, \oplus, \mathsf{u} \rangle$ is infinitely splittable if and only if:

$$\forall a \exists x, y.\, x \oplus y = a \wedge (x = \mathsf{u} \longrightarrow a = \mathsf{u}) \wedge (y = \mathsf{u} \longrightarrow a = \mathsf{u}) \tag{2.14}$$

That is, each non-unit element can be split into two non-empty elements. This property is very useful (in many cases even necessary) when dealing with divide-and-conquer computations.

Disjointness

A separation algebra $\langle A, \oplus, \mathsf{u} \rangle$ satisfies disjointness if and only if:

$$a \oplus a = b \longrightarrow a = \mathsf{u} \tag{2.15}$$

That is, non-unit elements cannot be joined with themselves. This seemingly arbitrary property is actually what prevents the malformed tree predicates presented in the introduction of this thesis.

Cross-Split

A separation algebra $\langle A, \oplus, \mathsf{u} \rangle$ satisfies the cross-split property if and only if:

$$\begin{aligned} &a \oplus b = z \wedge c \oplus d = z \longrightarrow \exists ac, ad, bc, bd. \\ &ac \oplus ad = a \wedge bc \oplus bd = b \wedge ac \oplus bc = c \wedge ad \oplus bd = d \end{aligned} \tag{2.16}$$

That is, if an element can be split two different ways, it should also be possible to split it into *four* pieces that respect the original decomposition. This property is important in some scenarios involving ownership transfer between concurrent threads.

2.1.7 Survey of Permission Models

In this section, we review the most relevant permission models encountered in the literature and evaluate them with regard to the properties introduced in the previous section.

Trivial Algebra

The simplest possible separation algebra contains only the unit. It is named here for completeness, but otherwise uninteresting.

Boolean Algebra

The boolean permission model is not a practical model (in the sense that using it is equivalent to separation logic without permissions), but it provides a toy example for later discussions. It is an algebra with two elements $\langle\{\bullet, \circ\}, \oplus, \circ\rangle$, where:

$$\circ \oplus \circ = \circ \tag{2.17}$$

$$\circ \oplus \bullet = \bullet \tag{2.18}$$

$$\bullet \oplus \circ = \bullet \tag{2.19}$$

The permission policy is straightforward: $\bullet$ is mapped to a full permission, $\circ$ means no permission. The model satisfies positivity, disjointness and is (trivially) cross-splittable.

Fractional Permissions

Fractional permissions constituted one of the early non-trivial extensions of separation logic [17]. They are the model currently used by Viper, as well as other tools such as VeriFast [28]. It is the algebra $\langle[0,1] \cap \mathbb{Q}, \oplus, 0\rangle$, where:[6]

$$a \oplus b = \begin{cases} a + b, & \text{if } a + b \leq 1 \\ \textbf{undefined}, & \text{otherwise} \end{cases} \tag{2.20}$$

Its usual policy is:

$$\Pi(a) = \begin{cases} \circ, & \text{if } a = 0 \\ \bullet, & \text{if } a = 1 \\ \text{◐}, & \text{otherwise} \end{cases} \tag{2.21}$$

The model satisfies positivity, infinite and cross-splittability, but not disjointness. However, a disjoint variant has been proposed [4], where we additionally condition the join on $a \neq b$.

[6] The symbol $+$ denotes regular addition over rationals, and $\leq$ is the familiar ordering of rationals.

Counting Permissions

Counting permissions [16] are represented by the algebra $\langle \mathbb{Z} \cup \{\mathsf{u}\}, \oplus, \mathsf{u} \rangle$, where:

$$a \oplus b = \begin{cases} a + b, & \text{if } a \neq \mathsf{u} \land b \neq \mathsf{u} \land a < 0 \land b < 0 \\ a + b, & \text{if } a \neq \mathsf{u} \land b \neq \mathsf{u} \land (a < 0 \land b \geq 0 \lor a \geq 0 \land b < 0) \land a + b \geq 0 \\ a, & \text{if } b = \mathsf{u} \\ b, & \text{if } a = \mathsf{u} \\ \textbf{undef}, & \text{otherwise} \end{cases} \tag{2.22}$$

With the policy:

$$\Pi(a) = \begin{cases} \bigcirc, & \text{if } a = \mathsf{u} \\ \bullet, & \text{if } a = 0 \\ \mathbin{\rotatebox{90}{◐}}, & \text{otherwise} \end{cases} \tag{2.23}$$

The intuition behind the model is simpler than its definition: there are two kinds of shares, token bundles (negative integers) and token factories (non-negative integers). Token factories can be split into another token factory and a token bundle, like so:

$$n \geq 0 \land l \mapsto_n v \iff n \geq 0 \land m > 0 \land (l \mapsto_{n+m} v) * (l \mapsto_{-m} v)$$

Looking at it from the other end, we could say that a factory can absorb at most as many tokens as it has issued (which we can simply read off as its value) after which point we have the full permission 0 (with 0 outstanding tokens). On the other hand, token bundles can be split up to the point where they are singleton bundles, e.g.:

$$l \mapsto_{-2} v \iff (l \mapsto_{-1} v) * (l \mapsto_{-1} v)$$

Because the integers under $\oplus$ do not possess an identity element, u is added. Counting permissions satisfy positivity, but neither disjointness, cross-splittability nor infinite splittability. Nevertheless, it is a very relevant model because it is the archetypal definition of a token counting scheme, which many scenarios call for.

Fractional Extension The above definition of $\oplus$ also yields a separation algebra over the set of rational numbers $\mathbb{Q}$ (with added unit) [16]. What we gain from this is infinite splittability, but especially with counting permissions, we might not want read permissions to be infinitely splittable. The reason for this is that many token counting schemes employed in concurrent settings rely on the fact that a read token cannot be further divided. Consequently, this model is less often seen in practice.

Simplified Version Because the original definition is quite complex, we also consider a simpler model to serve as a baseline implementation. In this model, tokens cannot be bundled. It is the algebra $\langle \mathbb{N} \cup \{\mathsf{u}, \tau\}, \oplus, \mathsf{u} \rangle$ where:

$$a \oplus b = \begin{cases} a - 1, & \text{if } a > 0 \wedge b = \tau \\ b - 1, & \text{if } b > 0 \wedge a = \tau \\ a, & \text{if } b = \mathsf{u} \\ b, & \text{if } a = \mathsf{u} \end{cases} \tag{2.24}$$

This version has the added benefit of satisfying disjointness.

Integer Sets

Parkinson [29] proposed *named shares*, which refers to the algebra $\langle \mathcal{P}(\mathbb{N}), \uplus, \emptyset \rangle$, where $\uplus$ is the union of disjoint sets, that is:

$$a \uplus b = \begin{cases} a \cup b, & \text{if } a \cap b = \emptyset \\ \textbf{undefined}, & \text{otherwise} \end{cases} \tag{2.25}$$

The policy of this model is:

$$\Pi(a) = \begin{cases} \bigcirc, & \text{if } a = \emptyset \\ \bullet, & \text{if } a = \mathbb{N} \\ \mathbin{\rotatebox{90}{◐}}, & \text{otherwise} \end{cases} \tag{2.26}$$

There are a couple of issues to discuss with this model. First, depending on the implementation, queries over this algebra might be undecidable. If for instance the sets a and b are given by predicates $P_a, P_b : \mathbb{N} \to \mathsf{Bool}$, proving

$$\forall n. \neg(P_a(n) \wedge P_b(n)),$$

which we need in order to determine whether the join of a and b is even defined, may be impossible. One variant of the model addresses this by only considering finite and co-finite integer sets[7], which allow for more concrete representations (list of elements or list of missing elements).

But, while we obviously have positivity and disjointness, and cross-splitting is realisable via set intersections, we do not have infinite splittability (because singleton sets cannot be split into two non-empty sets). Workarounds (such as restricting to infinite sets) are either hard to automate [30] or come at the cost of cross-splittability [7].

[7] Using the cardinalities of the sets we can also draw a bridge to the simple token counting scheme of counting permissions.

Tree Shares

Tree shares are an interesting model because they satisfy all of the outlined desired properties. On top of that, the model of finite and co-finite integer sets described above can be embedded within this model. We refer to the algebra $\langle \mathbb{T}, \oplus, \bigcirc \rangle$, where $\mathbb{T}$ is the set of binary trees with boolean-valued leaves and unlabeled internal nodes, i.e. $\tau := \bigcirc \mid \bullet \mid \langle \tau, \tau \rangle$. There are equivalence classes over trees, according to the following congruence definitions:

$$\bigcirc \cong \langle \bigcirc, \bigcirc \rangle \tag{2.27}$$

$$\bullet \cong \langle \bullet, \bullet \rangle \tag{2.28}$$

$$a_1 \cong b_1 \wedge a_2 \cong b_2 \longrightarrow \langle a_1, a_2 \rangle \cong \langle b_1, b_2 \rangle \tag{2.29}$$

$$a = b \longrightarrow a \cong b \tag{2.30}$$

In plain English, we can unfold leaves into an internal node that has as children two copies of its own value, and fold internal nodes with identical leaves as children into a leaf with the value of its children. The resulting trees belong to the same equivalence class as the originals. We use this definition for the join, which assumes that both arguments have the same shape. The congruence relation satisfies this requirement: let a, b be two trees we want to join, and let h_a, h_b be the height[8] of each tree. Then we can simply unfold both a and b to the full tree[9] of height $\max(h_a, h_b)$, and necessarily have the same shape while respecting the equivalence classes.

The join operation is defined as:[10]

$$\bigcirc \oplus \bigcirc = \bigcirc \tag{2.31}$$

$$\bigcirc \oplus \bullet = \bullet \tag{2.32}$$

$$\bullet \oplus \bigcirc = \bullet \tag{2.33}$$

$$\langle a_1, a_2 \rangle \oplus \langle b_1, b_2 \rangle = \langle a_1 \oplus b_1, a_2 \oplus b_2 \rangle \tag{2.34}$$

Which essentially is performing the Boolean separation algebra join leaf-wise.

Tree shares actually support a number of additional operations owing to the fact that they are both a boolean algebra as well as a distributive lattice. Notably, they support a complement operation $\neg$ satisfying $\tau \oplus \neg\tau \cong \bullet$, as well as a quasi-multiplicative operation which we will revisit later. We refer to [7] for a detailed treatment of the algebraic properties of tree shares. Tree shares also have decidable semantics [31] and have previously been automated in the HIP/SLEEK verifier [3].

[8] We have $\text{height}(\bullet) = \text{height}(\bigcirc) = 0$ and $\text{height}(\langle \tau_1, \tau_1 \rangle) = 1 + \max(\text{height}(\tau_1), \text{height}(\tau_2))$

[9] The full tree of height n has 2^n leaves, all at the same height.

[10] Or rather, can be defined as. A number of equivalent formulations is possible.

Composite Algebras

Dockins et al. [7] also propose a number of operators over separation algebras. For instance we can build the product of two algebras like so:

Definition 2.1.12 (Product Separation Algebra)**.** Let $\langle A, \oplus_A, \mathsf{u}_A \rangle$ and $\langle B, \oplus_B, \mathsf{u}_B \rangle$ be separation algebras. Their *product separation algebra* is $\langle A \times B, \oplus, (\mathsf{u}_A, \mathsf{u}_B) \rangle$ where $\oplus$ is defined componentwise:

$$(a_1, b_1) \oplus (a_2, b_2) = (a_1 \oplus_A a_2, b_1 \oplus_B b_2) \tag{2.35}$$

Another interesting construction is the function operator:

Definition 2.1.13 (Function Separation Algebra)**.** Let $\langle A, \oplus_A, \mathsf{u}_A \rangle$ be separation algebra and D some set. The the *function separation algebra* of D and A is $\langle D \to A, \oplus, (\mathsf{u}_A, \mathsf{u}_B) \rangle$ where $\oplus$ is defined pointwise:

$$f \oplus g = \begin{cases} \lambda x.\, f(x) \oplus_A g(x), & \text{if } \forall x : D.\, f(x) \oplus_\downarrow g(x) \\ \textbf{undefined}, & \text{otherwise} \end{cases} \tag{2.36}$$

The sum operator is more involved as the resulting algebra would have two units (and thus not be a separation algebra). There is an elegant theoretical solution for this in Dockins' paper, but this is beyond the scope of this introduction. The point is to alert the reader of the possibility that separation algebras can be composed in interesting ways, which we should keep in mind when approaching the implementation.

2.2 Viper Verification Infrastructure

In the following, we will provide a brief overview of the Viper verification toolchain and highlight various features of its intermediate language which will be important for understanding the contributions in this thesis. Most details are sourced from the excellent Viper online tutorial [32], which the reader is encouraged to consult in case of confusion.

2.2.1 Architecture

Viper currently has two supported back-ends: one is a symbolic execution tool called Silicon [33], the other revolves around the generation of verification conditions and is called Carbon. This thesis' implementation is carried out in Carbon. Both back-ends ultimately use the SMT solver Z3 [34] as the base layer of the tech stack. Whereas Silicon directly generates files for Z3 to check, Carbon uses another research verification tool called Boogie [35] (which in turn is built on top of Z3). Thus, Carbon is in essence a compiler turning viper files into Boogie files. For more details on Boogie, refer to Section 2.3.

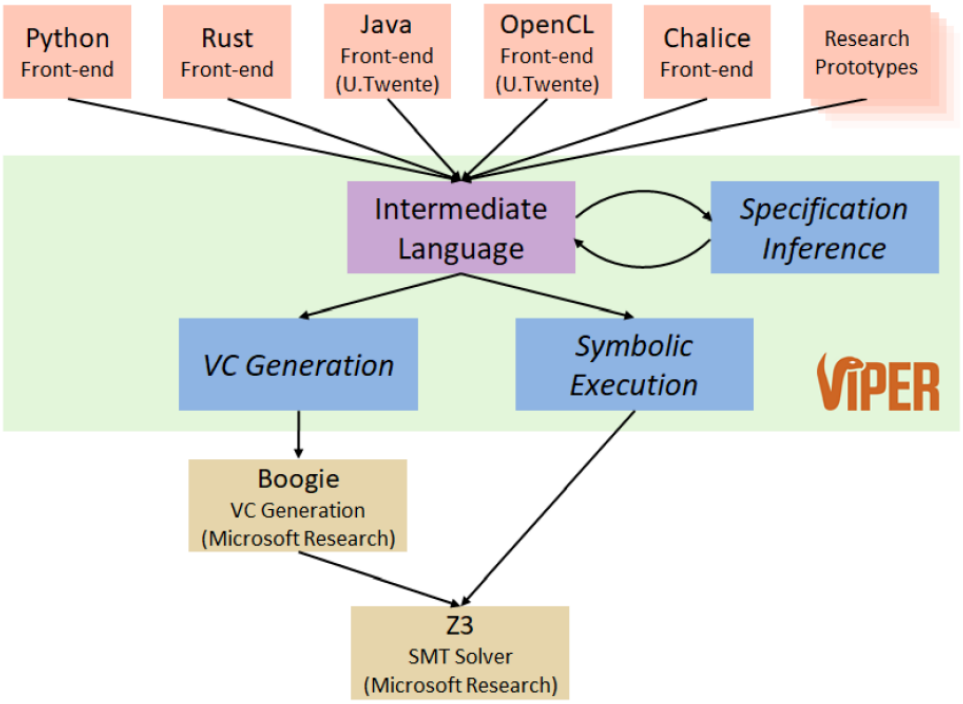


Figure 2.1: The Viper verification infrastructure

On the front end, there are a number of language specific verifiers, e.g. for Rust [36], Python [37], Go [38] and others. The guiding philosophy behind the Viper architecture (Fig. 2.1) is modularity; the Viper intermediate language provides a powerful interface through which various high-level language features can be expressed.

2.2.2 Silver Language Overview

For historical reasons, the Viper intermediate language is called Silver. Its syntax, an example of which is provided in Listing 2.1, is loosely based on the syntax of popular imperative languages such as Java, or C++. Viper programs are structured around *methods* – imperative-style procedure declarations along with a specification – which operate on a heap of objects (as well as local variables) and are verified independently of one another. All expressions are (strongly) typed, and there is no notion of subtyping (or, god forbid, type casting), so all expressions belong to one of the following types:

- `Ref`: references (to objects, except for the built-in Ref constant null)
- `Bool` for Boolean values
- `Int`: mathematical (unbounded) integers

```
0  field f: Int
1
2  method client(a: Ref)
3          requires acc(a.f)
4          ensures acc(a.f)
5  {
6          var oldf : Int
7          oldf := a.f
8          increment(a)
9          assert a.f == oldf + 1
10 }
11
12 method increment(x: Ref)
13         requires acc(x.f)
14         ensures acc(x.f) && x.f == old(x.f) + 1
15 {
16         x.f := x.f + 1
17 }
18
19 method print(x: Ref)
20     requires acc(x.f, 1/2)
21     ensures acc(x.f, 1/2)
```

Listing 2.1: Viper code example

- `Rational`: mathematical (unbounded) rationals
- `Perm`: permission amounts
- `Seq[T]`, `Set[T]`, `Multiset[T]`: Sequences, Sets and Multisets of members of some type `T`
- A type declared by the user via domains

In the following we will briefly examine Viper's heap model, method declarations as well as the predicates and domain features.

2.2.3 Objects and Fields

All heap accesses happen via *references* (type `Ref`) to objects on the heap. Objects have *fields* which can be declared at the top level of the program (e.g. the one on line 1 of Listing 2.1). All objects have the same fields. Unlike in object oriented programs however, objects do not have methods, so they can also be thought of as records or structs. Objects do not have explicit values (e.g. a memory address), and can only be compared through equality. There is a special `Ref` value named `null`, which indicates that the reference does not point to an object. Viper enforces the semantics of null references by disallowing inhalations of permissions to such references. This introduces the

pleasant effect that if we have a permission to a heap location, we do not need to check if that location is `null`. Objects are not created or deallocated in an imperative sense – viper code has no execution model after all – but instead references are assumed to be non-`null` (creation) or set to `null` (deallocation), the rest is handled via permissions, as we will soon see.

2.2.4 Methods

Silver methods comprise on one hand commonly known imperative constructs such as variable declarations and assignments, if-else-branches, loops and method calls and on the other hand, logical constructs that enable the formulation of a specification for the imperative code. The latter takes the form of pre- and postconditions (`requires...`, `ensures...`), as well as loop invariants, `assert` statements, and the `inhale` and `exhale` statements that we will discuss in more detail in a moment. As seen in the above listing, methods are not required to have a body.

Some other logical constructs worth highlighting are assumptions (keyword `assume`) and assertions (keyword `assert`). At any location within a method there is a set of assumptions. While some are automatically managed (e.g. the statement `x := 3` modifies earlier assumptions about the variable `x`), it is possible to use the keyword `assume` on a boolean expression to add that expression to the set of assumptions for all subsequent program locations. On the other hand, `assert` behaves as expected; the verifier tries to prove the given expression and issues an error message if it fails.

```
0 method m()
1 {
2     var a : Int
3     var b : Int
4     assume a >= 0 && b != 0
5     assert a / b <= a
6 }
```

Listing 2.2: Example uses of `assume` and `assert`

At the beginning of every method, we start with an empty heap **emp**, that is we do not have access to any objects' fields. We will examine how permissions are managed in the next section.

2.2.5 Permissions

Separation logic makes an appearance in Viper in the form of *resource assertions*, such as the commonly seen *access predicate*. Instead of the formal $l \mapsto_\pi v$ seen in the theory section, we write `acc(x.f, π)` in Viper (where `x` is some reference and `f` is some field identifier and π is some value of type `Perm`). The latter form is called an *access predicate*. There is also a semantical difference

between the theoretical formulation and Viper's access predicates. In the classical separation logic calculus, we generally have $\{l \mapsto_{\pi_1} v\} \vdash \{l \mapsto_{\pi_2} v\}$ only if $\pi_1 = \pi_2$. In Viper on the other hand, that side condition is relaxed to $\pi_1 \geq \pi_2$, that is if we have `acc(x.f,`π_1`)` for some permission π_1, then we also have `acc(x.f,`π_2`)` for all permissions π_2 that are less than or equal to[11] π_1. This is due to the fact that Viper does actually not model the classical separation logic introduced earlier, but is built on the notion of *implicit dynamic frames* [39][40], which can be seen as an intuitionistic analogue.

Viper has three permission levels: `none`, `read` and `write`. As the names suggest, reading from a heap location `x.f` (i.e. evaluating any expression that `x.f` is a part of) requires at least a read permission to that location. Assigning to the location requires a write permission. Writing `acc(x.f)` is short for `acc(x.f, write)`.

There is more to it than that, though. Viper's permission model is based on fractional permissions, that is permission amounts π are rational numbers in the interval $[0, 1]$.[12] Fractional permissions are constructed using integer division (e.g. `1/2`) as opposed to decimal point notation (e.g. `0.5`). The fraction `0/1` represents a `none` permission, `1/1` represents a `write` permission, and any number in $(0, 1)$ is considered a `read` permission.

Viper's fractional permissions support the following operators:

- `+` : $(\texttt{Perm}, \texttt{Perm}) \rightarrow \texttt{Perm}$ Rational number addition (results may extend beyond $[0, 1]$)
- `-` : $(\texttt{Perm}, \texttt{Perm}) \rightarrow \texttt{Perm}$ Rational number subtraction (with potentially negative results)
- `*` : $(\texttt{Perm}, \texttt{Perm}) \rightarrow \texttt{Perm}$ Rational number multiplication
- `*` : $(\texttt{Perm}, \texttt{Int}) \rightarrow \texttt{Perm}$ Multiplication by integer scalar
- `/` : $(\texttt{Perm}, \texttt{Int}) \rightarrow \texttt{Perm}$ Division by integer scalar

Considering the theory outlined in Section 2.1.5, it is surprising to see the last three operators here. We will just put a pin in this for now, as this will be very relevant in later parts of the thesis (cf. Section 4.8). The plain reason for now is that the type `Perm` and the rational number type `Rational` are just aliases for one another in the implementation. It is even legal to assign a `Perm` typed value to a `Rational` typed left hand side, and vice versa.

Pure Expressions

In separation logic, we can have assertions over the heap, and statements over local variables, constants and literals. These are fundamentally different,

[11] W.r.t. the order $\geq$

[12] Viper's permissions are actually a bit more relaxed than that, see section 2.2.6

as explained in Section 2.1 – resource assertions are predicates over heaps, whereas the other type of expressions are boolean expressions. To illustrate this, consider the expression `a == 3 && a == 3` vis-a-vis `acc(x.f) && acc(x.f)`. The former is equivalent to `a == 3` whereas the latter is decidedly not equivalent to `acc(x.f)`. In Viper, any expression that contains no resource assertions is called *pure*. Pure expressions may be combined with any logical operator, whereas non-pure expressions may not use disjunction (`||`-operator) or negation (`!`-operator). In the same vein, the logical implication operator `==>` may not be used with non-pure expressions as the left hand side (as it is essentially a combination of disjunction and negation). An implication $\mathcal{E}_p$ `==>` $\mathcal{E}_r$ with a pure expression $\mathcal{E}_p$ and a non-pure right hand-side $\mathcal{E}_r$ gets reduced either to `true` or $\mathcal{E}$ in a given state, depending on the value of $\mathcal{E}_p$.

Inhale and Exhale

Permissions in Viper methods are managed using so-called *inhale* and *exhale* statements. Used with a *pure* boolean expression (i.e. one which does not contain any resource assertions), `inhale` behaves identically to `assume`. However, a statement such as `assume acc(x.f, 1/2)` will generally lead to a contradiction, as we keep track of all permissions to heap locations. If we know that the permission to `x.f` is less than `1/2` (either at the current or any subsequent program location), then the above statement leads to an inconsistent state. This is in contrast with `inhale acc(x.f, 1/2)`, which adds (joins) the permission `1/2` to the existing permission. This can lead to an inconsistent state if the sum of the two is greater than 1, i.e. the result of the join is undefined. We will discuss this behaviour in greater detail in Section 4.6.

On the converse, we have `exhale`, which behaves identically to `assert` on pure expressions, and subtracts the specified permission amount from the existing permission to the given location. If that is not possible (because the existing permission is too small), an error message is issued. We will examine how this works under the hood in Section 2.3.2.

As a final remark on this topic: the keywords `requires` and `ensures`, which are used to specify pre- and postconditions respectively, behave the same as an `inhale` at the beginning and an `exhale` at the end of the method. The only difference is that pre- and postconditions are visible to other methods, whereas inhales and exhales in the method's body are not. When one method calls another, e.g. in Listing 2.1 where `client` calls `increment`, the caller first exhales the callee's precondition and then inhales the callee's postcondition.

Wildcards

Wildcards are special permission amounts that can be used in situations where we don't care exactly *how* much of a permission we have, just that

we have *some* non-empty permission. In other words, the assertion `acc(x.f, wildcard)` can be viewed as an existential statement, analogous to something like $\exists p.$ `acc(x.f,` p`)`. Internally, a `wildcard` is a permission amount that is unknown, but assumed to be nonzero. Every instance of the `wildcard` keyword is independent of all others – in general, one wildcard does not equal another. Of course, wildcards being these 'black boxes' means they have some very unusual behaviour. For one, as long as we do not know *for sure* that we have a full permission to a location, we can always inhale another wildcard instance. Conversely, as long as we know we have *some* permission, we can always exhale another wildcard.

```
0  field f: Int
1
2  method m(x: Ref)
3  ensures acc(x.f, wildcard)
4  {
5      var a : Perm
6      var b : Perm
7      inhale acc(x.f, wildcard)
8      a := perm(x.f)
9      exhale acc(x.f, a)
10     inhale acc(x.f, wildcard)
11     b := perm(x.f)
12     //inhale acc(x.f, wildcard - wildcard) //evil, no can do
13     inhale acc(x.f, a - b) //this is fine
14 }
```

Listing 2.3: Cruel abuse of wildcards

The type checker allows wildcards to occur only in access predicates, and disallows any sort of operation on them (e.g. `assert wildcard + wildcard == wildcard` would be rejected for all of these reasons). However, using permission introspection (the `perm` operator), we can extract a permission amount from a heap location and perform all the arithmetic we want on it, as illustrated in Listing 2.3. Of course, this example was written in complete disregard for the expected use of wildcards, but nonetheless foreshadows some of the interesting problems that they may pose in the future (cf. Section 4.7).

2.2.6 Predicates

Abstract predicates are another type of resource assertion in Viper. They are top-level declarations with a unique name, arguments and an optional body. While in principle any boolean expression could be used as their body, their usefulness is showcased when dealing with structures of objects on the heap.

For instance, consider linked lists, which have no bounds on the number of members. Exhaustively inhaling access to every node individually is

```
0  field elem: Int
1  field next: Ref
2
3  predicate list(this: Ref) {
4    acc(this.elem) && acc(this.next) &&
5    (this.next != null ==> list(this.next))
6  }
7
8  method append(this: Ref, e: Int)
9    requires list(this)
10   ensures  list(this)
11 {
12   unfold list(this)
13
14   if (this.next == null) {
15     var n: Ref
16
17     n := new(elem, next)
18     n.elem := e
19     n.next := null
20     this.next := n
21
22     fold list(n)
23   } else {
24     append(this.next, e)
25   }
26   fold list(this)
27 }
```

Listing 2.4: Use of list predicate

therefore not possible. However, the recursive nature of predicates allows us to do just that. In the example given in Listing 2.4, if we have the resource `list(this)`, we can always apply an operation called *unfolding* to receive the instance of the `list` predicate for the next node. On the converse, as seen in the method `append`, we can obtain the resource `list(n)` for the new node `n` by proving the body of the predicate in using the `fold` operation. Roughly speaking, the `unfold` operation exhales the predicate instance and inhales its body, while `fold` does the inverse (exhaling the body, followed by inhaling the instance).

Like with heap locations, it is possible to have a fractional permission to a predicate. For instance, having `acc(list(this), 1/3)` would give us a read-only permission to all elements in the list. As with access predicates of heap locations, `list(this)` is just a shorthand for `acc(list(this), 1/1)`. Unlike with heap locations, it is possible to inhale a fraction greater than `1/1` to a predicate without leading to a logical contradiction. For instance, we could

have a bodyless `predicate token()`, and inhale whole number amounts of this predicate to verify some system in which tokens are passed around, and performing certain actions requires certain amounts of tokens. Of course, this use case is at odds with the more general permission models which do not have this kind of 'extension' above the full permission. This issue is discussed in a bit more detail in Section 4.8.

2.2.7 Magic Wands

The magic wand operator `--*` introduces the third kind of resource assertion in Viper (besides heap access assertions and predicates). If `A` and `B` are resources, then `A --* B` is also a resource, which, when held together with resource `A`, yields resource `B` (consuming both the wand and `A`). We cover this operator here for completeness' sake; its use and implementation are beyond the scope of this thesis.

2.2.8 Domains

In Viper, we can formulate arbitrary mathematical theories via the domain module. This feature will see heavy use in Chapter 6, where we discuss a general-purpose interface for the Viper permission model. Each domain must have a unique name; this name is then introduced to the program as a new type (in Listing 2.5, this type is `PA`). Within the body of the domain, the user may declare functions and axioms. Functions are purely symbols, their semantics are not provided in a body, but rather by the axioms. For instance, in the given example, the behaviour of the `plus` function is given solely by the last two functions. As we can see, axioms support the use of universal quantifiers, along with existential quantifiers. We can also see in each axiom an expression in braces. This is called a trigger, embodying a technique known as *e-matching*: each axiom is only instantiated when an expression can be matched against its trigger(s). We will examine triggers in more detail once we talk about Boogie (cf. Section 2.3.3). Domain types can also accept type parameters; this way it is possible to build algebraic datatypes such as lists or trees of other types.

2.2.9 Carbon Back-End

This section provides a brief overview of Carbon's software architecture. As mentioned before, Carbon compiles Silver (`.vpr`) files into Boogie files (`.bpl`). Because the early phases (broadly speaking: lexing, parsing, type checking) are shared between Carbon and the other back-end, Silicon, they are organized into a separate code repository called Silver. This detail becomes relevant when we discuss changes to the Silver language, as compatibility with Silicon must be ensured before release. What the Silver package delivers to the back-ends is a type-checked abstract syntax tree of the input program.

```
0  domain PA {
1  function _0(): PA
2  function S(PA): PA
3
4  axiom axSInj { forall n: PA, m: PA:: {S(n), S(m)}
5      S(n) == S(m) <==> m == n}
6  axiom axSno0 { forall n: PA :: {S(n)} S(n) != 0}
7
8  function plus(PA, PA): PA
9  axiom axPlus0 { forall a: PA :: {plus(a, 0)} plus(a, 0) == a}
10 axiom axPlusB { forall a: PA, b: PA {plus(a, S(b))}
11     plus(a, S(b)) == S(plus(a, b))}
12 }
```

Listing 2.5: Peano arithmetic via a Viper domain

The phases remaining for the Carbon code base are therefore optimization and generation of Boogie code. The main functional unit of Carbon is the *Module*. The modules can be thematically organized into four categories:

- Translation modules for generic types, expressions and statements.
- Modules for built-in datastructures (`set`, `seq`, `map`).
- Modules for user-defined functions (not methods) and domains.
- Modules dealing with separation logic (permissions, `inhale` and `exhale` statements, magic wands, heaps and program state).

The top level modules are then given the relevant sections of the program and the emitted Boogie statements are assembled into the output program. Another pillar of the software architecture are *Components* for different kinds of purposes, which are Scala traits that can be inherited by the relevant modules. For instance, an `inhale` statement both involves operations on permissions (which is the responsibility of the `PermissionModule`) and changes to the heap (which is the responsibility of the `HeapModule`). For this scenario, the trait `InhaleComponent` (which comprises a single method, `inhaleExp`) is inherited by both of these modules.

For the purposes of this thesis, the most noteworthy modules are the permission module (obviously), and the domain module, both of which will be discussed in more detail when the need arises.

2.3 Boogie

Boogie [41][35] is an intermediate language and verifier developed by Microsoft Research. The Boogie language bears many syntactic similarities to

Silver. Put somewhat crudely, Boogie is Silver sans separation logic components. What we will therefore mostly be examining are the differences, as well as the ways in which some of the separation logic aspects of Viper are modeled in Boogie. The information here should be taken with a grain of salt, as the official documentation [42] is lackluster and the only somewhat comprehensive language tour [41] is outdated in important ways.

2.3.1 Similarities and Differences

Overall, Boogie's syntax is very similar to Silver's, if a bit more rigid, in particular with regards to the placement of variable declarations and use of whitespace. Just as in Viper, the verification 'units' in Boogie are methods, called `procedures` here. Again we have the familiar keywords `requires` and `ensures` used to specify pre- and postconditions. Boogie only supports a small number of built-in types: `Int`, `Bool`, `Real` (numbers) and maps (which we will explain in more detail when discussing the translation of heap accesses).

The procedural building blocks of Viper can be found here as well. The familiar `assume` and `assert` statements make an appearance too. The syntax for axiom and function declarations is identical; however in Boogie they are top-level declarations and not wrapped in a domain. In fact, domains do not exist in Boogie. If we want to declare an additional type, we can do so at the top level.

Unlike Viper, Boogie supports global mutable variables as well as constants (constants, to be fair, can be modeled using 0-ary functions). Procedures must declare which global variables they modify. This is so that if one procedure calls another, the caller has the right to remember the assignments of all global variables not modified by the callee. Forgetting assignments can be carried out explicitly, using the syntax `havoc v;` for some variable `v`. We will see heavy use of this feature once we discuss how wildcards are implemented.

2.3.2 Representing Permissions in Boogie

Since there is too much to say about how Carbon maps Viper's separation logic constructs to Boogie – that discussion would have to cover most of the Carbon implementation – we will limit our discussion simply on the representation of permissions in Boogie, since that is the most relevant for this thesis, and also the only part that is subject to substantial changes as a consequence of the work presented. Notably absent from our discussion here will be the modeling of and transitioning between heap states.

Permission Mask

We keep track of all currently held permissions in a global variable called `Mask`. Its type is:

```
MaskType = <A, B> [Ref, Field A B]Perm,
```

that is, it is a map from heap locations (`Ref, Field`-pairs) to Permissions). The type `Perm` is simply an alias for `Real` (numbers) in the current implementation. Notably, this definition has two type parameters, `A` and `B`, which are used to further specify the `Field` type. In Boogie, most declarations can have type parameters, including type declarations, function declarations and axioms. The purpose of the type parameters of `Field` is twofold: on one hand, it allows us to use the same Mask datastructure for all types of fields while still remembering their type. An `Int`-typed field would have type `Field NormalField int`, a boolean field type `Field NormalField bool` and so on. The type `NormalField` implies the existence of non-normal fields, and that is indeed the case: predicates and magic wands are managed by the same mask. Those have the type `Field` A `FrameType`, where A is either a type indicating a specific predicate or a magic wand. Their implementation is much more involved and are thus left omitted in the following discussion.

At the start of each method, the permission mask is assigned the constant `ZeroMask`, which is defined by the following, unsurprising axiom:

```
forall <A, B> o: Ref, f: (Field A B) :: ZeroMask[o, f] == NoPerm
```

That is, a method starts out with an empty heap (**emp**), after which permissions are added to or removed from the mask using the `inhale` and `exhale` statements discussed in the following.

Consistency Assumptions

It is possible that some sequence of operations leaves the permission mask (or heap) in an inconsistent state. For instance, inhaling a permission to `x.f` if we already have a write permission would lead to an invalid entry in the permission mask (i.e. a value greater than `1/1`). For this purpose, the following predicate is defined:

```
function state(Heap: HeapType, Mask: MaskType): bool
```

For our purposes, we can ignore what `state` says about the heap (in the current implementation, this argument is ignored completely). For the mask argument, the `state` implies the following (slightly simplified):

1. All mask entries must be greater than or equal to the empty permission (`0/1`).

2. All mask entries that do not refer to permissions or magic wands must be less than or equal to the full permission (`1/1`).

Carbon implements this consistency criterion by interleaving the statement `assume state(Heap, Mask);` with statements that may invalidate the mask. This of course leads to a contradiction if the mask is actually in an inconsistent state. After this point, verification essentially stops, or rather succeeds trivially. The `assume state(Heap, Mask)` also serves as a help to make certain inferences. For instance, if we have a method that `requires acc(x.f) && acc(y.f)`, we can assert in the method's body that `x != y`. This is because the only kind of verification trace that is compatible with the assumption `state(Heap, Mask)` is the one where a full permission did not get added twice to the same location.

The astute reader may have a hunch that this kind of consistency assumption does not translate well to other kinds of permission models. This is correct; the `state`-predicate only works because we have 'large' type (`Real`) of whose members valid permission values are a subset ($[0,1] \subset \mathbb{R}$). We will discuss this issue in more detail in Section 4.6.

Inhale

Since the disjunction operator $\vee$ is not defined over resource assertions, we can assume that the argument of the inhale operation is a boolean expression in what is roughly speaking conjunctive normal form, that is, the statement is of the form[13]:

$$\texttt{inhale } \mathcal{E}_1 * \mathcal{E}_2 * \ldots * \mathcal{E}_n.$$

Where $\mathcal{E}_i$ is either a pure expression, a resource assertion or a conditional resource assertion (use of `==>` operator with resource assertion) for $i \in \{1, \ldots, n\}$. The expressions are inhaled in order, from left to right. We proceed by cases:

Pure Expressions are compiled into the statement `assume` $\mathcal{E}_i$

Access Predicates of the form `acc(x.f,`π`)` are translated into the following[14]:

```
perm := π;
assert perm >= NoPerm;
assume perm > NoPerm ==> x != null;
Mask[x, f] := Mask[x, f] + perm;
```

[13] Even if it is not exactly in this form, transforming it is as easy as rearranging the parentheses.

[14] The variable `perm : Perm` is defined at the beginning of the procedure.

Conditional Resource Assertions of the form $\mathcal{E}_1$ `==>` $\mathcal{E}_2$ are translated into the following:

`if (`$\mathcal{E}_1$`) {` ⟨translate $\mathcal{E}_2$⟩ `}`

In all three cases, the output is followed by the consistency assumption `assume state(Heap, Mask);`.

Exhale

Exhales function analogously (very roughly speaking) to inhales, subtracting permissions where an `inhale` would be adding them. Unlike `inhale` statements, `exhale`s may reorder expressions; concrete values (like `1/2`) get subtracted first, followed by abstract permissions (i.e. unknown values, like wildcards or uninitialized variables). The subtraction is preceded by an assertion of the following kind: `assert {:msg ...} Mask[x, f] >= perm;` where the message, which gets issued when the assertion fails, relays information about the source code location that produced the assertion. Exhale statements also delete values from the heap to which we no longer have permission.

Wildcards

For uses of wildcards, Carbon compiles a variable declaration `var wildcard : Perm where wildcard > NoPerm`. Whenever we use use a wildcard, we first reset our knowledge about its value (wildcard instances are not identical to one another) using the statement `havoc wildcard`. The `where` clause defines an assumption over the `wildcard` variable that is maintained throughout uses of `havoc` (as opposed to independent `assume` statements about the variable, which are forgotten upon `havoc`).

Wildcards are inhaled just like other permissions. Exhales are slightly different. One could be tempted to just `assume Mask[x, f] > wildcard` and then `assert Mask[x, f] >= wildcard` just like with regular permissions. However, this leads to a contradiction if `Mask[x, f] == NoPerm`. Thus, the implementation actually first checks `assert Mask[x, f] > NoPerm` to ensure a wildcard can be exhaled and then `assume Mask[x, f] > wildcard` before subtracting the wildcard.

Permission Policy Enforcement

Enforcing Viper's permission policy is straightforward. When encountering a field access, i.e. an expression of the form `x.f`, we place before the translation of whatever statement the expression appears in, the statement: `assert hasDirectPerm(Mask, x, f);` which is a roundabout (but necessarily so) way of asserting `Mask[x, f] > NoPerm`. Lastly, before any field assignment `x.f := ...` we assert `Mask[x, f] == FullPerm`. `NoPerm` and `FullPerm` are simply aliases for the values `1.0` and `0.0`.

2.3.3 Guarantees and limitations

Boogie and Z3 are (hopefully) sound. This means if Boogie verifies an assertion, that assertion should be guaranteed to hold. However, since undecidable theories can be formulated in Boogie, an assertion failure can mean two things: either the assertion is actually false, or the verifier just did not manage to prove it. This incompleteness will be a topic of discussion in the next chapter (see Section 3.5).

Sometimes it is possible to salvage a valid but unproven assertion by either adding appropriate axioms or by adding a 'helper'-assertion before it as an intermediate proof goal. Unlike with proof assistants like Coq or Isabelle/HOL, there is little feedback on why the proof failed or what the remaining proof goals are, so the process of supporting the verifier with auxiliary assumptions and proof goals is inherently somewhat experimental.

We would be remiss to not mention the third kind of observable behaviour: non-termination. This reason for this behaviour is usually a matching loop caused by inappropriate axiom triggers, a topic we will discuss now.

Axiom Triggers and Matching Loops

A commonly used feature of axioms is universal quantification. For instance, if we recall the encoding of Peano arithmetic (cf. Listing 2.5), we have an axiom that says $\forall n : \mathsf{PA}.S(n) \neq 0$. The only way Boogie can use an axiom is if it is instantiated with some values. Since there are infinitely many possible instances of the above axiom, it is impossible to have every instance at the ready. Instead, the axiom is instantiated whenever its trigger can be matched against a newly derived expression. The trigger in the above case is simply $S(n)$, so if we for example wrote `assert S(S(a)) != 0` for some Peano integer `a`, `S(S(a))` would be matched against `S(n)` with the substitution `n` $\rightarrow$ `S(a)`, and the assertion could be proven with the generated instance of the axiom.

Trigger Form A trigger is an expression or comma-separated sequence of expressions in braces $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ after variable quantification. Each of the trigger expressions $\mathcal{E}_i$ for i from 1 to n must be available for the axioms to be instantiated. Trigger expressions may not be variable literals, nor a comparison between expressions (because of the problem of infinite instantiations that we started with). Trigger expressions are not required to appear in the axiom's body, but all quantified variables must be mentioned in a trigger. Finally, it is possible to have multiple triggers per axiom by simply inserting another trigger $\{\ldots\}$ after the previous one. The effect is that each trigger functions independently.

Matching Loops To illustrate how easy it is to accidentally create a matching loop, suppose we wanted to introduce the predecessor function P (inverse of S) to Peano arithmetic with the following axiom:

```
forall n: PA :: {P(a)} S(P(a)) == a && P(S(a)) == a
```

Now, if we have a matching expression `P(`$\mathcal{E}$`)` for the trigger, then the right half of the conjunction in the axiom produces the instance `P(S(`$\mathcal{E}$`))` – which is another match for the trigger! The next instantiation produces the expression `P(S(S(`$\mathcal{E}$`)))` and so on.

Consequences for Implementation

There are a few points to keep in mind as we move on to the discussion of the implementation in the next chapter:

- Formulating very general triggers for axioms can lead to matching loops. Similar with failed but valid assertions, the axioms may contain a potential matching loop, but may only be found if an unlucky sequence of assertions is assembled.
- Formulating very restrictive triggers (or no triggers) can have the same effect as not having any axioms in the first place. Therefore, setting triggers, like many other processes when working with Boogie, is a delicate affair and one can not rely on the verifier to prove or disprove that appropriate triggers have been chosen.
- It is very easy to accidentally formulate inconsistent axioms, however, depending on their instantiation, this unsoundness may not be revealed until – again – an unlucky sequence of assertions is encountered. The fact that the inconsistency isn't revealed until the wrong (or right) axioms are instantiated has even been exploited to encode alternative permission models [6]. Thus it is not enough to `assert false` and assume that Boogie will find the inconsistency – axioms must be verified externally.
- Lastly, even if the right axioms and triggers have been chosen, there is no completeness guarantee – one has to verify through testing that Boogie is actually capable of reaching the desired proof goals. We discuss the details of this in Section 3.5.

Chapter 3

Axiomatization

In the following chapters we document the techniques and rationales used to conceive of the models documented in the following chapter. The problem for this chapter is, in essence, to find ways to embed the mathematical models presented in naive set theory in Chapter 2 into Boogie's proof theory. Our principal concerns are soundness on one hand, as well as a degree of completeness that allows proper functioning of the Carbon verifier relying on the embedded permission model.

3.0.1 Design Goals and Tradeoffs

On a practical note, we must consider our axioms not just as abstract mathematical theories, but as objects which a program (ultimately, the SMT solver) operates on. On one hand, this tightens our soundness goal, as matching loops (see also Section 2.3.3) are another way our verifier can become practically useless. Additionally, each axiom comes at a cost – the more axioms are in the model, the longer the SMT solver will generally run. To some degree, our goals are in conflict with one another. Generally speaking, the fewer axioms we define, the lower the chance that there is some latent contradiction between them. It is also easier to prove consistency for a smaller set of axioms than a larger one. On the other hand, since the power of Boogie only goes so far, we are forced to introduce new axioms that help the verification along. Identifying every possible kind of assertion that Carbon might emit and ensuring that the axioms suffice in proving them is a time-consuming and ultimately open-ended process since aspects of the implementation may change over time. While we can provide no theoretical guarantee that the axioms discussed in the following satisfy the completeness requirement, we do provide a rationale according to which at least a subset of assertions are provable. Additionally, testing has been performed on each model to verify that the most common uses are covered (see Section 3.5).

3.0.2 Set Notation vs. Type Notation

In the previous chapter, where separation algebras were introduced, we used set notation (e.g. $a \in A$)[15] as is common in mathematical treatises. However, when formulating axioms for the aforementioned algebras, we will stick to the type theory and the associated notation ($a : A$) used by the back-end (Boogie). Since there is no notion of subtyping in Boogie, this means there are subtle differences when denoting subsets vis-a-vis 'subtypes'.

3.1 Modeling Sum Types

In the following sections, we will frequently need to extend a type A with additional members to define some useful extension of an existing algebra, be it to introduce an identity element to algebras that do not have one or to build product or sum algebras from existing algebras. In set notation, we can simply write: $A \cup \{e\}$ to denote the set A extended with an additional element $\{e\}$. Operations over A can be extended with straightforward notation. We will denote the equivalent construction in type theory by $A + \{e\}$, which is a shorthand for the sum type $A + \mathsf{Unit}$, where $\mathsf{inl} : A \to A + \mathsf{Unit}$ and $\mathsf{inr} : \mathsf{Unit} \to A + \mathsf{Unit}$ denote the left and right introduction respectively and finally, $e = \mathsf{inr}(())$. Operations over this new type have different types of signatures than the old ones, and thus to maintain a semblance of coherent notation we must use the introduction function inl whenever members of the old type A are used. To make this notation a bit easier on the eyes, we will use the shorthand $[a] = \mathsf{inl}(a)$.

Boogie does not provide these constructions as primitives, so axioms must be added for the introduction function inl[16]:

$$\forall a : A, b : A.\, [a] = [b] \longleftrightarrow a = b \tag{3.1}$$

$$\forall a : A.\, [a] \neq e \tag{3.2}$$

3.1.1 Finite Types

Unfortunately, the above axioms are not powerful enough to prove arbitrary properties over the sum type (e.g. associativity of a join), since we have not yet encoded the fact that the sum type contains *only* the elements defined by left and right introduction. If there exist other latent elements in the sum

[15] As before, if nothing else is stated, A is the symbol used to denote the carrier set of the separation algebra under discussion, whereas u denotes its identity element.

[16] In Axiom 3.1, the inverse direction of the implication ($a = b \longrightarrow [a] = [b]$) is redundant as function semantics are built into Boogie.

type then proofs by case distinctions will obviously not work. Fixing this should be simple enough in theory. Let $\Sigma = A + \{e\}$ be the sum type. Then:

$$\forall \sigma : \Sigma. \, (\exists a : A. [a] = \sigma) \vee \sigma = e \tag{3.3}$$

In practice, the problem is that we cannot define triggers for this axiom (c.f. Section 2.3.3). This can be worked around in the following way: we define the self-explanatory function $\mathsf{id} : \Sigma \to \Sigma$ with the axiom

$$\forall \sigma : \Sigma. \, \mathsf{id}(\sigma) = \sigma \tag{3.4}$$

and use the fact that $\mathsf{id}(\sigma)$ is a valid axiom trigger which we then use for Axiom 3.3. If we then try to prove some property about elements of $\sigma : \Sigma$, we use the expression $\mathsf{id}(\sigma)$ instead of σ or the axiom won't be instantiated and the proof may fail. Note that a similar situation arises when we try to introduce constructors for elements of an arbitrary type A. For instance, let us say we want to model fractional shares. Let F be the type of these shares. We want to be able to construct these shares using rational numbers, so we define the constructor $\mathsf{frac} : \mathbb{Q} \to F$. Of course, the following must hold, if we want to prove properties of F via the constructor:

$$\forall a : F. \, \exists q : \mathbb{Q}. \, \mathsf{frac}(q) = a$$

Here again, the above workaround would be needed.

3.2 Share Constructors

In the remaining sections of this chapter, there will be many references to the term 'constructor'. For the purposes of this thesis, a share constructor for a share type A is a function with A as the result type. These functions usually have no definition they can be substituted for, i.e., if $f : T \to A$ is a constructor, there is no axiom of the form

$$\forall t : T. \, \mathcal{P} \longrightarrow f(t) = \mathcal{E}$$

with some precondition[17] $\mathcal{P}$ and some expression $\mathcal{E}$.

There may be multiple constructors. A desirable property of a set of constructors is that every share in the separation algebra can be described using one of the constructors as per the following somewhat informal definition [18]:

[17] Preconditions are commonly encountered when defining more complex functions, see Section 3.7

[18] We use the unconventional quantifiers $\mathbb{\forall}, \mathbb{\exists}$ to set meta-axioms like this one typographically apart from the actual axioms discussed in this chapter.

Definition 3.2.1 (Covering Constructors)**.** Let A be the type of shares, and C be a set of constructors of A. We say that the constructors cover the type A if

$$\forall a : A.\, \exists f \in C, t.\, f(t) = a$$

holds in the model.

The point of the above definition is to introduce the assumption that constructor expressions are the 'primitives' of share expressions and ensure that every kind of share value has some primitive representation. This assumption is essential for the completeness argument presented in Section 3.5.

Additionally, if a share $a : A$ is doubly covered, i.e. there exist constructors f_1, f_2 and expressions t_1, t_2 such that $f_1(t_1) = f_2(t_2) = a$ holds in the model, we demand that the equivalence is provable from the axioms. The most straightforward way to ensure this is to make the ranges of the constructors disjoint.

3.3 Common Axioms

There are a number of assumptions the implementation makes about the permission model. As discussed in the background chapter, separation algebras are partial cancellative commutative monoids, and all useful ones are positive, which gives rise to the following axioms:

$$\forall a : A, b : A.\, a \oplus b = b \oplus a \tag{3.5}$$

$$\forall a : A, b : A, c : A.\, a \oplus (b \oplus c) = (a \oplus b) \oplus c \tag{3.6}$$

$$\forall a : A, b : A, c : A.\, a \oplus c = b \oplus c \longrightarrow a = b \tag{3.7}$$

$$\forall a : A.\, a \oplus \mathsf{u} = a \tag{3.8}$$

$$\forall a : A, b : A.\, a \oplus b = \mathsf{u} \longrightarrow a = \mathsf{u} \tag{3.9}$$

However, when encoding a concrete permission model, we may wish to omit one or more of them. The explanation is given in the following section.

3.4 Redundant Axioms

While in this chapter we outline a number of generally valid axioms, it is also true that oftentimes, these general axioms are made redundant when a concrete permission model is formulated. To illustrate this, consider the following encoding of the extremely simple boolean permission model (introduced in Section 2.1.7) with just two shares, empty ○ and full ●:

$$\bigcirc \oplus \bigcirc = \bigcirc \tag{A1}$$

$$\bigcirc \oplus \bullet = \bullet \tag{A2}$$

$$\bullet \oplus \bigcirc = \bullet \tag{A3}$$

It comprises just three axioms, yet almost all general axioms presented in this chapter are easily derived from them. This raises two questions: is this a problem and if yes, how does this inform the implementation?

On the first question, the answer is an unsatisfying "it depends". Generally, having redundant axioms increases the likelihood of inconsistencies, or at least increases the complexity of proving the consistency of the encoding. Having more axioms also means the search space for the SMT-solver is larger, as there are more axioms which can be expanded at any step. On the other hand, if a proof is too long, Boogie will simply fail to verify the corresponding statement.

Consequences for Implementation The unfortunate realisation stemming from the above discussion is that devising axioms for permission models for Boogie is an inherently manual process with weak (practical) completeness guarantees on the result. The approach we chose when encoding new permission models in carbon was to

1. rigorously check for inconsistencies between axioms and to
2. use tests to determine if the axioms are strong enough to prove predetermined proof goals (see also Section 3.5.2).

Using this approach, many general axioms did not make it into the encodings. However, there is still a small reason to discuss them: reuse. In the above example algebra, we could have replaced the last axiom with the axiom of commutativity (Equation 3.5). Of course, writing $\bullet \oplus \bigcirc = \bullet$ is easier than $\forall a : A, b : A.\ a \oplus b = b \oplus a$, but if we have the latter ready-made somewhere in the implementation, we can just reference it. This is arguably helpful, as every time a user or developer types something themselves, fresh errors can sneak in. In fact, just as we were first typesetting the simple equations above, we switched up one of the symbols by accident.

However, shorthands for boilerplate axioms is a quality of life feature at best, and remains an unexplored possibility in the implementation.

3.4.1 Fine Print: Axioms Include Undefined Results

The common axioms presented in this section are usually stronger than what can be derived from a concrete encoding. Consider for instance the following

axiom, which might be used to define the join over fractional shares[19] F:

$$\forall q : \mathbb{Q}.\, \mathsf{valid}(q) \longleftrightarrow 0 \leq q \wedge q \leq 1$$

$$\forall p : \mathbb{Q}, q : \mathbb{Q}.\, (\mathsf{valid}(p) \wedge \mathsf{valid}(q) \wedge \mathsf{valid}(p+q)) \longrightarrow [p] \oplus [q] = [p+q]$$

From this definition, it is possible to prove $a \oplus b = b \oplus a$ for arbitrary a, b *only if* there exist rationals p, q such that $[p] = a \wedge [q] = b$ (provable from constructor rules) *and* the precondition $\mathsf{valid}(p) \wedge \mathsf{valid}(q) \wedge \mathsf{valid}(p+q)$ is fulfilled. We could *not*, for instance, prove $2/3 \oplus 3/4 = 3/4 \oplus 2/3$ with these axioms. On the other hand, the commutativity axioms is strong enough to prove this. This fact is relevant for our discussion of unknown values in the next section.

3.5 Managing Incompleteness

As previously discussed, we want to keep our set of axioms as lean as possible. As we know from Gödel's famous incompleteness theorem [43], in axioms capable of modeling integer arithmetic, there are always proof goals that are not reachable from the axioms, and that is before we account for the limited power of Boogie. Instead of obsessing over an air-tight axiomatisation of the permission models, we might want to flip the problem around and exercise some reverse mathematics: what proof goals involving permissions does Carbon emit and which axioms are needed to prove them?

3.5.1 Consequences of Incompleteness

First, a word on what the practical implications are if the axioms are not powerful enough. For the user, this issue will take the appearance of an assertion that really should pass, but does not. Such an assertion is presented to the user as a potential error. What happens next depends on the user's confidence in her abilities: if it is high, she might recognise that nothing is wrong with her Viper program and be frustrated with the limited capabilities of the verifier. If it is low, she will invest more time trying to find the error in the encoding that isn't there and ultimately progress to the high confidence case. Needless to say, the question of addressing weak axiom sets is well motivated.

3.5.2 Types of Proof Goals

Of course, since Viper provides the option to encode arbitrary mathematical theories through the domain feature, there is no theoretical limit on the kinds of statements that the verifier might be asked to prove. However, these use

[19] Here, $[a]$ is shorthand for $\mathsf{share}(a)$ where $\mathsf{share} : \mathbb{Q} \to F$ is the share constructor.

cases are presumably uncommon, and the problem is simultaneously the solution. If the power of the axioms does not suffice for the user's needs, it is possible to add any necessary axioms via domains. Therefore, we restrict this discussion to everything encountered outside of domains.

We divide the remaining goal types into two categories: permission arithmetic with known quantities and arithmetic with unknown quantities. The top level proof goals are the same in both cases. Since a proof goal is always some boolean-typed expression, the kinds of questions we will have to answer are comparisons. These can be explicit (e.g. `assert perm(x.f) >= 1/2`) or implicit (e.g. `exhale acc(x.f, 1/2)`). Of course, these comparisons can involve permission-typed expressions of arbitrary complexity. To help this discussion, we introduce a few terms and a simple grammar for talking about these expressions:

$$\mathsf{Atom} \;:=\; \mathsf{Concrete} \mid \mathsf{Unknown}$$

$$\mathsf{PExp} := \mathsf{Atom} \mid \mathsf{PExp} \oplus \mathsf{PExp} \mid \mathsf{PExp} \ominus \mathsf{PExp}$$

$$\mathsf{PCmp} := \;\mathsf{PExp} \;=\; \mathsf{PExp} \mid \mathsf{PExp} \;\succeq\; \mathsf{PExp} \;\ldots$$

We use PExp as an abbreviation for permission-typed expression. An atom is an expression that cannot be further decomposed. What the two atom cases refer to is explained in separate sections below. Before we get to that, we make one more definition:

Definition 3.5.1 (Concrete Permission Expressions)**.** A permission expression $\mathcal{E}$ is concrete if all its atoms are concrete. Otherwise we say the expression is unknown.

The purpose of listing this grammar is the recognition that we can break the problem down over the structure of the expressions. If we can assemble a list of axioms that allow proving comparisons between atoms and comparisons between join and inverse join composite expressions, we can be reasonably assured (by the principle of structural induction), that the axioms are strong enough for any way the verifier might use permissions, without having to scour the code base for every possible use.

Notation For the following discussion, we will borrow some notation from model theory: we denote the set of axioms by $\mathbf{\Phi}$. For an arbitrary statement (boolean-typed expression) A, we say that $\mathbf{\Phi} \vdash A$ if A is provable from $\mathbf{\Phi}$. Provability in this context has a stricter meaning, as it refers to provability *in Boogie*. On the other hand, we write $\mathbf{\Phi} \models A$ if A is true in every model of $\mathbf{\Phi}$.

3.5.3 Concrete Expression Arithmetic

A concrete atom takes the form of a constructor[20] application (e.g. `share(1)` in our axiomatization of counting shares, cf. Section 5.3), or, in the case of fractional permissions, a division of two concrete integers, e.g. `1/2`. Let $\mathcal{E}_1, \mathcal{E}_2$ be arbitrary constructor applications. We require:

- Either $\mathbf{\Phi} \vdash \mathcal{E}_1 = \mathcal{E}_2$ or $\mathbf{\Phi} \vdash \mathcal{E}_1 \neq \mathcal{E}_2$.
- Either $\mathbf{\Phi} \vdash \mathcal{E}_1 \succeq \mathcal{E}_2$ or $\mathbf{\Phi} \vdash \neg(\mathcal{E}_1 \succeq \mathcal{E}_2)$.[21]
- Either $\mathbf{\Phi} \vdash \mathcal{E}_1 \oplus_{\downarrow} \mathcal{E}_2$ or $\mathbf{\Phi} \vdash \neg(\mathcal{E}_1 \oplus_{\downarrow} \mathcal{E}_2)$. [22]
- If $\mathbf{\Phi} \vdash \mathcal{E}_1 \oplus_{\downarrow} \mathcal{E}_2$, then there must exist a constructor expression $\mathcal{E}_3$ such that $\mathbf{\Phi} \vdash \mathcal{E}_1 \oplus \mathcal{E}_2 = \mathcal{E}_3$. The existence of $\mathcal{E}_3$ rests on the assumption outlined in section 3.2.
- If $\mathbf{\Phi} \vdash \mathcal{E}_1 \succeq \mathcal{E}_2$, then there must exist a constructor expression $\mathcal{E}_3$ such that $\mathbf{\Phi} \vdash \mathcal{E}_1 \ominus \mathcal{E}_2 = \mathcal{E}_3$. The existence of $\mathcal{E}_3$ rests on the assumption outlined in section 3.2.

In plain English, these requirements guarantee on one hand that primitive permission expressions (i.e. constructor applications) are comparable and that if we start with permission primitives (i.e. constructor applications), we can reduce the result of a join or inverse join to another primitive expression, which is in turn comparable. Thus we have a guarantee that any comparison of concrete values can be proven or disproven from the axioms. For almost all use cases, this guarantee is sufficient.

3.5.4 Unknown Atom Arithmetic

Unknown atoms come in three flavours:

1. Uses of uninitialized permission-typed variables
2. Applications of abstract or inductive functions whose return type are permissions
3. Uses of the keyword `wildcard`

Assumptions These atoms may come with some sort of assumption. For instance, wildcards come with the assumption `wildcard` $\neq \bigcirc$ (depending on the context, other assumptions added, cf. Section 4.7). We will thus denote unknown expressions as a tuple $\langle \mathcal{E}, \mathcal{A} \rangle$, subject to the following rule:

$$\langle \mathcal{E}_1, \mathcal{A}_1 \rangle \oplus \langle \mathcal{E}_2, \mathcal{A}_2 \rangle = \langle \mathcal{E}_1 \oplus \mathcal{E}_2, \mathcal{A}_1 \wedge \mathcal{A}_2 \rangle$$

[20] See Section 3.2 for the definition of constructors.

[21] A weaker version actually suffices. Requiring that if $\mathbf{\Phi} \models \mathcal{E}_1 \succeq \mathcal{E}_2$, then $\mathbf{\Phi} \vdash \mathcal{E}_1 \succeq \mathcal{E}_2$ must hold, is enough.

[22] Analogous to the previous footnote.

and an analogous rule for the inverse join.

Power of Unknown Expressions Drawing on the well known universal generalisation inference rule

$$\frac{P(x)}{\forall x.\, P(x)}$$

it is easy to see that assertions involving unknown quantities are very powerful indeed, and it has to be expected that we eventually hit the wall in terms of what Boogie can prove. Therefore, we require that the basic separation algebra properties can be derived, and rely on the user to fill in the gaps, should they occur. Wildcards have special semantics that are explored in more detail in Section 4.7

Guarding Against Undefined Values One thing we can do is exclude potentially undefined values from the program; again, drawing on division by zero as an analogy, it does not make sense to make any assertions over an expression like `x/y` (for uninitialized variables `x, y`), nor any expression that contains it if we do not know if `y` is zero. Note that this is different from the case where an `inhale` statement induces a join with an undefined result. What we mean here are expressions generated by the user which are invalid permissions, i.e. the expresseion `1/2 - 10/2`. Viper currently enforces this behaviour for zero-divisions, issuing an error to the user. We require:

1. An unknown expression $\langle \mathcal{E}_1 \oplus \mathcal{E}_2, \mathcal{A} \rangle$ must be rejected if $\mathbf{\Phi} \cup \mathcal{A} \nvdash \mathcal{E}_1 \oplus_{\downarrow} \mathcal{E}_2$.
2. Analogously, unknown expressions of the form $\langle \mathcal{E}_1 \ominus \mathcal{E}_2, \mathcal{A} \rangle$ must be rejected if $\mathbf{\Phi} \cup \mathcal{A} \nvdash \mathcal{E}_1 \succeq \mathcal{E}_2$.

The implementation of these rules is discussed in Section 4.6.

3.6 Definedness and Comparison Predicates

3.6.1 Definedness

There are many instances in constructing the axioms and when working with permissions in the verifier where it is essential to have a binary predicate $\oplus_{\downarrow} : A \times A \to \mathsf{Bool}$ which determines whether the expression $a \oplus b$ is defined for a particular choice of a and b. One could try to express this in a general way, like so:

$$\forall a : A, b : A.\, a \oplus_{\downarrow} b \longleftrightarrow \exists x.\, a \oplus b = x.$$

However, again for trigger-related reasons, it is not possible to encode this axiom in Boogie. We must therefore construct a predicate that is more straightforward to evaluate. Its exact formulation will depend on the particular separation algebra. Some examples can be seen in the appendix.

When specifying the relation, particular attention must be paid that

$$\forall a : A, b : A.\, a \oplus_{\downarrow} b \longleftrightarrow b \oplus_{\downarrow} a \tag{3.10}$$

holds. The relation must conform to other properties also, but speaking from experience the above one is the easiest to get wrong, which leads to inconsistent axioms for reasons that should become obvious over the next few sections.

3.6.2 Inverse Join

As we will see later, the verifier requires a kind of 'subtraction' over permissions, i.e. an inverse join $\ominus$. Defining this is pretty straightforward:

$$\forall a : A, b : A, c : A.\, a \oplus b = c \longleftrightarrow c \ominus b = a.$$

But, since it is impossible to set appropriate triggers for this formulation, we must use an alternate version:

$$\forall a : A, b : A.\, (a \oplus b) \ominus b = a \tag{3.11}$$

However, if you think that this is enough to prove the converse identity

$$\forall a : A, b : A.\, (a \ominus b) \oplus b = a$$

you would be mistaken. For that we need to axiomatise how the inverse join 'distributes' with the join:

$$\forall a : A, b : A, c : A.\, (a \oplus b) \ominus c = a \oplus (b \ominus c) \tag{3.12}$$

3.6.3 Preorder

Again, defining the preorder $\succeq$ over A is easy in principle:

$$\forall a : A, b : A.\, a \succeq b \longleftrightarrow \exists x : A.\, a = b \oplus x.$$

As with the definedness predicate, a more straightforward definition is needed in practice, which depends on the algebra at hand. Note that the preorder $\succeq$ acts as the analogue of $\oplus_{\downarrow}$ with respect to $\ominus$, that is $a \ominus b$ is defined if and only if $a \succeq b$.

3.6.4 Equality

Most separation algebras have rather straightforward equality semantics. Boogie provides a built-in equality operator that checks for syntactic equality, which is enough for most models. Nonetheless, we might want to define our own equality predicate (e.g. on trees, which is a model with equivalence classes). Together with the $\succeq$ predicate, we can define a number of dependent comparisons for convenience (namely: $\succ, \prec, \preceq, \neq$).

3.7 Modeling Case Distinctions

A common feature of many permission models is a join function that is defined by case distinctions (at the very least, there is a case where the function is undefined and one where a result is given). There is nothing particularly complicated about axiomatising these, but in favor of completeness we will discuss the approach used in this thesis.

Suppose for the sake of generality that $f : A_1 \times \ldots \times A_n \to B$ is an n-ary function. We will write $\bar{a}$ as a shorthand for the n-tuple of arguments $(a_1, \ldots, a_n)$. Then, suppose a definition of the following form is provided:

$$f(\bar{a}) = \begin{cases} \mathcal{R}_1, & \text{if } \mathcal{P}_1(\bar{a}) \\ \vdots & \\ \mathcal{R}_k, & \text{if } \mathcal{P}_k(\bar{a}) \\ \textbf{undefined}, & \text{otherwise} \end{cases}$$

where $\mathcal{R}_1, \ldots, \mathcal{R}_k$ are expressions of type B with free variables in $\bar{a}$, and $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are predicates (which may also simply be boolean expressions) over $\bar{a}$. We can simply turn this into a set of k axioms of the form:[23]

$$\forall \bar{a} : \bar{A}.\, \mathcal{P}_i(\bar{a}) \longrightarrow f(\bar{a}) = \mathcal{R}_i$$

for $i \in \{1, \ldots, k\}$. To avoid soundness issues via contradicting definitions, we must ensure at least the following meta-property:

$$\forall i, j \in \{1, \ldots, k\}, \bar{a} : \bar{A}.\, i \neq j \longrightarrow \mathcal{P}_i(\bar{a}) \longrightarrow \neg\mathcal{P}_j(\bar{a}) \vee \mathcal{R}_i(\bar{a}) = \mathcal{R}_j(\bar{a}) \quad (3.13)$$

that is, either the predicates identify disjoint subsets of $\bar{A}$ or the right hand sides of the definition agree under the given variable assignment. We (or at a later stage, the user) would have to manually check this, but an automated approach is possible.

Automated Conflict Avoidance Let $\mathcal{P}_0(\bar{a}) = \bigwedge_{i=1\ldots k} \neg\mathcal{P}_i(\bar{a})$ be the implicitly defined predicate for the undefined case. We construct new predicates $\hat{\mathcal{P}}_i(\bar{a}) = \mathcal{P}_i(\bar{a}) \wedge \bigwedge_{j=0\ldots i-1, i+1\ldots k} \neg\mathcal{P}_i(\bar{a})$ for all $i \in \{0, \ldots, k\}$. The new predicates are by construction mutually exclusive[24] and can be used in the axioms instead of the original predicates to ensure soundness. However, the construction obviously complicates the predicates, making verification more computationally intensive. For this reason, we did not implement this procedure.

[23] Note that in the actual Boogie encoding, the arguments would need to be individually spelled out, as we are dealing with syntactic function application.

[24] Care must be taken that none of the original predicates are identical, otherwise the constructed predicate would be unsatisfiable by construction.

Automated Definedness Predicate As a nice bonus of the above construction, we can get the definedness relation $f_\downarrow$ for free. We have $f_\downarrow(\bar{a}) \longleftrightarrow \neg\hat{\mathcal{P}}_0(\bar{a})$. As with the above though, finding a more succinct formulation of the relation might pay dividends in terms of performance.

3.8 Turning Semigroups into Monoids

Not all permission models in the literature strictly adhere to the way separation algebras are defined in this thesis (see Section 2.1.5). For example, the previously described *counting shares* do not have an identity element, and are thus not a monoid, but a semigroup. However, for the implementation it is much more pleasant to be able to assume that all permission models are separation algebras. Consider for instance the exhaling of some permission p to a given location. Under separation algebras, we can simply assert $p_0 \succeq p$, where p_0 is the currently held permission. However, if the algebra has no identity element, then generally, $\neg(a \succeq a)$, and thus we would have to implement a special case checking if the exhaled permission is equal to the currently held one. The good news are that it is relatively simple to lift a semigroup to a monoid by 'tacking on' an identity element.

Definition 3.8.1 (Identity-Extension)**.** Given a commutative, cancellative, (partial) semigroup $\langle A, \oplus\rangle$, its identity-extension is the separation algebra $\langle A + \{\mathsf{u}\}, \widehat{\oplus}, \mathsf{u}\rangle$, where:

$$\forall a : A, b : A.\ [a] \mathbin{\widehat{\oplus}} [b] = [a \oplus b] \tag{3.14}$$

$$\forall x : A + \{\mathsf{u}\}.\ x \mathbin{\widehat{\oplus}} \mathsf{u} = x \wedge \mathsf{u} \mathbin{\widehat{\oplus}} x = x \tag{3.15}$$

As corollaries, we get:

$$\forall a : A.\ [a] \mathbin{\widehat{\ominus}} [a] = \mathsf{u} \tag{3.16}$$

$$\forall a : A, b : A.\ a \neq b \longrightarrow [a] \mathbin{\widehat{\ominus}} [b] = [a \ominus b] \tag{3.17}$$

Finally, the predicates are defined as follows:

$$\forall x : A + \{\mathsf{u}\}.\ x \mathbin{\widehat{\oplus_\downarrow}} \mathsf{u} \wedge \mathsf{u} \oplus_\downarrow x \tag{3.18}$$

$$\forall a : A, b : A.\ [a] \mathbin{\widehat{\oplus_\downarrow}} [b] \longleftrightarrow a \oplus_\downarrow b \tag{3.19}$$

$$\forall x : A + \{\mathsf{u}\}.\ x \mathbin{\widehat{\succeq}} \mathsf{u} \tag{3.20}$$

$$\forall a : A.\ \neg(u \mathbin{\widehat{\succeq}} [a]) \tag{3.21}$$

$$\forall a : A, b : A.\ [a] \mathbin{\widehat{\succeq}} [b] \longleftrightarrow a \sqsupset b \tag{3.22}$$

Note The order $\sqsupset$ on the semigroup $\langle A, \oplus\rangle$ is irreflexive (because no identity element exists), asymmetric ($a \sqsupset b \longrightarrow \neg(b \sqsupset a)$) and transitive.

3.9 Dealing with Undefined Values

3.9.1 The Origin of Undefined Values

As has been discussed in the background chapter, separation algebras are partial monoids, and as such the result of the join operation $\oplus$ is undefined for some elements. In particular, for any reasonable permission model, the value $\sigma_\bullet \oplus \sigma_\bullet$ is undefined (denoted as $\bot$). In theory, we may simply ignore the existence of such a value by ensuring that no result for $\sigma_\bullet \oplus \sigma_\bullet$ is derivable from the axioms[25]. The semantics of an undefined value are also readily modeled by the SMT solver: For all values $a : A$, neither $a = \bot$ nor $a \neq \bot$ can be proven.

3.9.2 Expressing Undefinedness

However, trouble arises when we try to formulate a predicate $\mathsf{defined} : A \to \mathsf{Bool}$, which determines whether a given value a is defined. From a theoretical standpoint, such a predicate is nonsensical: 'definedness' is not a property of elements of A, it is a property of terms in the formal language. Thus, the predicate defined is inherently a higher-order statement and thus almost impossible to model using Viper's back-end tools. This realization leaves us with two options: either we shoehorn the semantics of an undefined value into the algebra using an explicit value $\bot$, or we check which part of the implementation demands a defined predicate and change it so that this requirement is obviated. We will describe the former approach below, while the latter (and more elegant) approach will be discussed in Section 4.6.

3.9.3 Extending Separation Algebras with Undefined Values

The problems described in the previous section stem from the fact that the join operation $\oplus$ is partial. In the following, we describe an approach to 'completing' the operation using an explicit undefined value $\bot$.

Definition 3.9.1 (Completion)**.** Given a separation algebra $\langle A, \oplus, u \rangle$, we define the *completion* $\langle A + \{\bot\}, \widehat{\oplus}, u \rangle$ as follows: Let $\oplus_\downarrow$ be the minimal binary relation on A which includes all pairs of elements whose join is defined. The (total) function $\widehat{\oplus}$ is defined using the following axioms:

$$\forall a : A, b : A.\ a \oplus_\downarrow b \longrightarrow [a] \mathbin{\widehat{\oplus}} [b] = [a \oplus b] \tag{3.23}$$

$$\forall a : A, b : A.\ \neg(a \oplus_\downarrow b) \longrightarrow [a] \mathbin{\widehat{\oplus}} [b] = \bot \tag{3.24}$$

$$\forall x : A + \{\bot\}.\ x \mathbin{\widehat{\oplus}} \bot = \bot \wedge \bot \mathbin{\widehat{\oplus}} x = \bot \tag{3.25}$$

[25] In order to enable reasoning about aliased references in Viper, it important that we can determine from the axioms whether an expression $a \oplus b$ is defined, even if a result cannot be derived. This will be done using the joinability predicate $\oplus_\downarrow$ discussed later.

Using the completion of the original algebra, it becomes trivial to define the above definedness predicate. However, this is not a free lunch, as we will discuss next.

3.9.4 Algebraic Properties of $\bot$

The semantics of $\bot$ must be defined with care to avoid inconsistencies. Things truly get difficult once we consider what the introduction of an undefined constant means for the algebra. Here is an overview of the more troubling implications:

1. By the reflexivity of equality, we get $\bot = \bot$. If this were to hold, we have changed what it means for a value to be undefined. In particular, we have $\mathbf{\Phi} \vdash \mathcal{E}_1 = \mathcal{E}_2$ for all undefined expressions $\mathcal{E}_1, \mathcal{E}_2$. This is consequential for the implementation as unprovable assertions can be relied on to generate error messages.
2. As opposed to values which are unknown (e.g. uninitialized variables) and thus undefined, the quantity $\bot$ is explicitly undefined. To be consistent, either all variables should be automatically initialized to $\bot$, or the assumption should be made that the variable is not equal to $\bot$.
3. We must have $\forall a : A + \{\bot\}.\, a \oplus \bot = \bot$. To see why, consider the implications of having permission $b = a \oplus \bot$ to a heap location for some $a, b \neq \bot$ and being able to split off an undefined share from b to get another defined share a.
4. As a consequence of point 3, the algebra loses its cancellative property.
5. As a consequence of point 3, in the order $\succeq$, $\bot$ is now the maximal element instead of the full share.
6. As a consequence of point 3, the algebra no longer has an identity element.

Of course, there are ways to work around most of these issues. For practical purposes, it is enough that the 'separation algebra' is cancellative*, that $=$ is an equality relation*, that $\succeq$ is a partial order* etc, where * points to the terms and conditions saying that $\bot$ is exempt. However, this does not change the following two issues: the semantics of the undefined element are very murky and encoding all the axioms significantly complicates the formulation of a satisfactory set of axioms (see Appendix A for an example).

3.9.5 Implementation to the Rescue

If we take one thing away from the previous discussion, it is that we want to pretend that $\bot$ is not a part of the separation algebra as much as possible. Remember: the only reason we introduced this element is to be able to decide

for any value whether that value represents a valid share or whether it is undefined. Whenever we have identified such a value in a user-generated expression, this should be be handled similar to a division by zero. Meaning, we stop everything that we are doing, and report it to the user, since contingent results might be affected by the undefined value and will be meaningless.

Thus, in the implementation, we would like to assume that all values are defined until we encounter one that is decidedly not. If we work under this assumption, it is possible to eliminate many of the edge cases in the definitions (for instance, $a \oplus \perp$ can be left undefined, because we will never proceed to compute $a \oplus \perp$ if we have identified $\perp$). However, this assumption needs to be upheld by the implementation at all times, otherwise strange assertion failures will be presented to the user, who presumably knows nothing about the implementation detail that is $\perp$. For the basic `inhale` and `exhale` statements, the assumption is preserved , but it is unclear whether that is true for the more complicated constructs in the Viper language, which is a subject for future work.

3.9.6 Evaluation

Ultimately, there are many reasons to avoid the undefined extension if at all possible. Here is the shortlist:

1. The axioms become more convoluted, complicating soundness proofs.
2. The semantics of the $\perp$-element are questionable, allowing for unintuitive proofs.
3. The verifier must uphold rigid invariants to avoid incomprehensible error messages.

This is ultimately why this approach was abandoned in favour of a more labour-intensive, yet ultimately more principled and elegant approach as detailed in Section 4.6.

Chapter 4

Carbon Implementation

In the following chapter we discuss the practical implementation of the axiom sets introduced in Chapter 5. We detail the relevant parts of Carbon's architecture, document our proposed general permission module and explore the pitfalls encountered along the way. We round off the discussion with potential approaches for supporting wildcards and abstract predicates.

4.1 Permission Module Overview

The interface for permission modules is given by a trait named `PermissionModule`. Its default implementation is called `QuantifiedPermModule` and is a rather large piece of code (at over 1'700 lines) which does a lot of heavy lifting for Carbon. The essential methods it implements are listed below[26]:

These definitions are accompanied by a number of useful wrapper methods for other scenarios, as well as methods for magic wands which this thesis ignores. An overview of the most important methods can be found in the following table:

4.2 Departing from Rational Number Arithmetic

As previously discussed, the `Perm` type in Silver refers to fractional permissions, and is actually equivalent to the `Rational` type under the hood (this was actually very consequential for this work, as further discussed in Section 6.4). In Boogie, this type gets mapped to the built-in `Real` datatype. As one would expect from an implementation of rational numbers, addition, multiplication, subtraction, division as well as additive inverses (unary minus operator) are provided. However, as we know, separation algebras are

[26]The type annotation `Exp` refers to the type of Boogie expressions, whereas `sil.Exp` refers to Silver expressions.

Name/Type	Description
`permType : Type`	The permission type (reference for other modules).
`preamble` `: Seq[Decl]`	A sequence of Boogie declarations prepended to the output program. All permission-related axioms are defined here.
`translatePerm` `: sil.Exp` $\rightarrow$ `Exp`	Translates a silver expression of type `Perm` into a Boogie expression of type `permType`.
`permissionPositive` `: sil.Exp` $\rightarrow$ `Exp`	Generates a Boogie expression that evaluates to true if and only if the given permission is non-empty.
`hasDirectPerm` `: sil.LocationAccess` $\rightarrow$ `Exp`	Analogous to unpacking the location access (which can be a predicate or a field access) and calling `permissionPositive`, but uses the predicate `hasDirectPerm`, which is explained in section **??**
`inhaleExp` `: sil.Exp` $\rightarrow$ `Stmt`	Translation of an inhale statement.
`exhaleExp` `: sil.Exp` $\rightarrow$ `Stmt`	Translation of an exhale statement.

in general not endowed with a multiplication (and thus neither division) operation (although efforts in that direction have been made [44]), and they also don't have well-defined additive inverses (in fact, almost all useful models satisfy positivity, which means no share except the empty share has an additive inverse, see Section 2.1.6). On top of that, Boogie's `Reals` require no axioms as their semantics are built into the system. They can also be used in conjunction with Boogie's built-in ordering operators ($>, \geq, <, \leq$). Many of these 'luxurious' features of rational numbers are baked into the existing code base and had to be addressed. The process of creating a new module with different permission semantics can thus be broken down into the following parts:

1. Copying the parts of the default implementation that are independent of the permission model.

2. Generating errors in `translatePerm` for permission expressions that make use of the multiplication, scalar (integers) multiplication, division, and unary minus operations. Removing at least the latter from the Silver language would be the most principled approach, but would break backward compatibility in substantial ways.

3. Replacing all uses of comparison operators on permissions. This needs to be done as Boogie does not support operator overloading. Instead, the operand expressions are passed to specialized functions in the permission module (`shareGt`, `shareGe` etc.), which wrap the arguments into the appropriate axiomatically-defined predicates. These functions already existed in the codebase, but were not consistently used.
4. Replacing all uses of arithmetic operators on permissions analogously to the above.
5. Axiomatizing the permission model in the preamble.

4.3 Permission Policy

As discussed in the background (Section 2.1.3), a permission model is a separation algebra plus a policy $\pi : \mathsf{Share} \to \mathsf{Perm}$. The verifier then checks if the rules associated with the different permissions have been observed. Viper historically uses just three permissions: None, Read and Write, with the straightforward policy:

$$\pi(\sigma) = \begin{cases} \mathsf{None}, & \text{if } \sigma = 0 \\ \mathsf{Write}, & \text{if } \sigma = 1 \\ \mathsf{Read}, & \text{otherwise} \end{cases}$$

The semantics of the permissions are self-explanatory: assigning to a heap location requires a Write and reading from a location requires a Read or Write permission (see also Section 2.2.5). While other permissions such as a write only permission are conceivable, we estimated the effort in overhauling the Viper infrastructure exceeds the need for such a feature. The three permission levels where thus left as-is. The same goes for the policy. While it is feasible to separate the share and permission types in the Boogie encoding and provide a policy alongside the separation algebra, it is more convenient and presumably faster during verification to simply designate two elements of the separation algebra as the empty (None) and full (Write) permission respectively, and consider all other elements as read permissions. This decision is well-motivated: the aspects of simplicity and performance have already been mentioned, but in practice, the need for multiple distinct full (resp. empty) permissions rarely manifests.

4.4 General Permission Module

Based on the above outline, we created an alternative permission module based on counting permissions as a proof of concept and an intermediate step for a more general permission module. What quickly became apparent

was that a large portion of the permission module's code could be carried over to the new one. The obvious solution was to separate out all of the separation algebra related methods into an interface, which comprises the following definitions:

- A list of axioms (`axioms`), which are arbitrary Boogie declarations which shall define all necessary types, functions, and properties of the separation algebra.
- The share type (`shareType`). This type can either be used by the permission module directly or be used by compositional separation algebras to define their types.
- Constant expressions for the full and empty share (`fullShare`, `emptyShare`). These implicitly define the permission model's policy, as outlined in the previous section.
- Functions for join operation $\oplus$ and inverse join (`shareJoin`, `shareInvJoin`). These functions have the signature $(\texttt{Exp}, \texttt{Exp}) \rightarrow \texttt{Exp}$, which is general enough to completely maintain the behaviour of the previous implementation of fractional permissions: `shareJoin(a, b)` can return `a + b` for fractions and `join(a, b)` for other algebras (where `join` is a function defined by the share algebras axioms; its name can be chosen arbitrarily by the separation algebra implementation).
- Definedness predicates for the join and inverse join (`joinable`, `shareGe`), where the latter doubles as a comparison operator. These functions have the same signature as the join and inverse join, with the same benefits as outlined above.
- Equality predicate (`shareEq`).
- Derived comparison predicates (`shareNe`, `shareGt`, `shareLe`, `shareLt`) for convenience. Default implementations are provided.
- The previously discussed `permissionPositive` function, which is simply a wrapper around `shareNe` that performs some static checks in attempt to simplify the resulting expression (like turning `shareNe(fullShare, emptyShare)` into `True`).
- A share constructor (`shareConstr`), which is used to create user-defined share amounts (e.g. `2/3` in fractional permissions). The constructor takes two integer expressions and returns a (`shareType`) expression. The reason why the arguments are integers and there are two of them has to do with Silver's syntactic rules. The separation algebra is free to define how these arguments are interpreted (for instance, the counting permission implementation simply discards the second argument and uses the first to create a counting share with that value). This solution

is not the most elegant, but is hard to improve as further explained in Section 4.5.

The general permission module (named `AlgebraicPermModule`) is provided with a separation algebra implementation during initialization and delegates all share operations to it. With this separation into permission module and separation algebra, implementing new permission modules becomes as straightforward as possible.

4.5 Limitations

There are two notable shortcomings of the generalized permission module. One is due to architectural reasons, another due to over-reliance on rational number arithmetic by the implementation. They are explained in the following paragraphs.

4.5.1 Share Constructors

As we know, the Viper infrastructure comprises another back-end besides Carbon and they both share the Silver language specification. We were therefore very reluctant about changing the latter, as this would break compatibility with the other back-end, Silicon. As it stands, fractional permissions are deeply baked into Silver's specification. Besides the use of multiplication, division and unary minus operations on permissions, which are undefined in the context of separation algebras, it also employs a simple way of instantiating permission values as a fraction of two integers (e.g. `1/1` for a full permission). This is the most primitive representation of a permission-typed value in a Silver program's AST. For the back-end, this means that a division of two integers is the only way the user can convey any information about what kind of share value she would like at this program location (besides using constants, such as `write, read, wildcard`).

We therefore have two choices, either figure out some sensible mapping from pairs of integers into shares (which is what `shareConstr` is used for in the separation algebra interface), or redefine the language syntax to enable arbitrary construction of shares. Apart from compatibility with Silicon, the latter approach has another problem: every new permission model potentially requires a new variant of the language specification (and therefore its own Silver parser). Take tree shares for instance: expressions specifying trees have a much different structure than arithmetic expressions involving rational numbers. This means introducing tree shares is not just about writing down a few axioms, but also involves a grammatically non-trivial change in the Silver language; a scenario that does not scream user-friendliness. This limitation is a key motivation behind the front-facing permission interface presented in the next chapter.

4.5.2 Lack of Predicate Support

It is no coincidence that Viper supports multiplication and division over permissions. Predicates rely on these operations, and we have not yet found a reasonable workaround. As a more subtle point, it is possible to hold permissions greater than 1 to a predicate, that is predicates essentially use an extended permission model compared to other resources.

For the uninitiated reader, this problem is elaborated in the following. Consider Listing 4.1, where we define a predicate `onlyThis` over (infinite, for simplicity) linked lists of objects. Given an object where this predicate holds, we get write access to the value of that element, but only read access (`1/2`) to objects further on in the list. The multiplication comes in when we repeatedly unfold the predicate, as seen in the method `foo` (which Viper verifies successfully), where we start with a fractional amount of `1/3` that predicate. After one unfolding, we end up with a share of `1/3 * 1/2 = 1/6` of the predicate to the next element. The further we unfold, the more the permission decays. When the predicate is folded again, the permission amounts are restored to the original state. Possible solutions to this limitation are discussed in Section 4.8.

```
0  field value : Int
1  field next : Ref
2
3  predicate onlyThis(this: Ref) {
4      acc(this.value) && acc(this.next)
5      && acc(onlyThis(this.next), 1/2)
6  }
7
8  method foo(this: Ref)
9  requires acc(onlyThis(this), 1/3)
10 {
11     unfold acc(onlyThis(this), 1/3)
12     assert perm(onlyThis(this.next)) == 1/6
13 }
```

Listing 4.1: A predicate requiring share multiplication

4.6 Eliminating Undefined Values

Due to all the issues outlined in section 3.9.6, it was clear that the problem of undefined values had to be dealt with at the root, not the ugly leaves. The root is the way Boogie handles consistency checks around `inhale` statements (described in detail in Section 2.3.2). To briefly recap, an inhale statement comprises a join operation; namely, `inhale acc(x.f, σ)` joins the share σ onto the share representing the currently held permission to the location `x.f`. This join can of course produce an undefined result. On one hand, if for instance

we already hold a full permission, and σ is also a full permission, then we hold an undefined permission to `x.f`. If that's not enough, it is even possible for σ itself to be undefined by being larger than 1 (the implementation guards against division by zero and inhaling negative fractions). The reason the latter is allowed has to do with a scenario involving predicates in which permissions are allowed to surpass a full share, as previously discussed.

Be that as it may, the range of legal permissions to heap locations is still the range $[0, 1]$, and the only reason these shenanigans is even possible is that we perform these operations in rational number arithmetic as opposed to a more restricted model. After the inhaled permission is joined onto the already held permission, the implementation assumes that the result is defined, i.e. that the result is in the range $[0, 1]$. If that is not the case (i.e. the permission to the heap location is undefined), we reach a contradiction. This way, no nonsensical verification errors will be generated. The bad news: no errors will be generated at all, since *ex falso sequitur quodlibet* – everything can be proven from contradictions, up to and including `assert false`.

Whether the behaviour described above is sensible is something we will go into a bit further on. For now, we would like to discuss the implementation detail that spurred the development of the theoretical work outlined in Section 3.9. That detail is how the verifier checks whether the new permission is defined. This was clearly done with fractions in mind: first we compute the result, which for the sake of argument comes out as $3/2$, then we assume that it is in the range $[0, 1]$ and arrive at our desired contradiction. Note however, that this is generally only possible when you have a 'larger' algebra, in this case the rationals, in which the result of the computation is guaranteed to be defined, and then check if it is in the subset of defined values (for the 'smaller' algebra) afterwards. How can this be done if there is *no* such larger algebra that our permission model is embedded in? The short answer is: it cannot – there is no way to formulate a predicate that could identify an undefined share resulting from an ill-formed join (the detailed reason is explained in Section 3.9). In the general case, there is no subset of 'defined shares' in a larger set of possible share values. Either we know the value because we have some axiomatically grounded expansion of it, or we do not, and it remains a black box. If we want to retain the original semantics of the verifier, there are two options:

1. Make it so that that subset exists, e.g., by extending the separation algebra with a special undefined element which all undefined join results are mapped to, or
2. insert the `assume` statement before the join is assigned to the heap location's mask entry, because at this point we know the arguments of the join and can simply use the definedness predicate $\oplus_{\downarrow}$ to determine whether the result will be defined.

For a detailed presentation of the first approach, see Section 3.9. In summary, that solution not only necessarily changes the verifier's semantics in questionable ways, it also comes at the cost of almost all useful theoretical properties of separation algebras. Therefore, our attention is turned to the second option, which required more changes to the code base but is conceptually cleaner and maintains the previous implementation's semantics.

4.7 Wildcard Semantics

In this section, we will explore how wildcards (cf. Section 2.2.5 for an introduction, and Section 2.3.2 for implementation details) should behave in generalized permission models. Recall that an assertion $l \mapsto_\omega v$ can be understood as a statement $\exists\, \omega.\, \omega \neq \sigma_\bigcirc \wedge l \mapsto_\omega v$. However, one could take a different viewpoint and consider wildcards as elements of the separation algebra. Towards this end, let ω denote a wildcard instance. Then the following four properties hold:

1. Wildcards are unknown non-empty permission quantities. It follows that for arbitrary permissions σ:
   - $\sigma \oplus_\downarrow \omega \longrightarrow \sigma \oplus \omega \succ \sigma$
   - $\sigma \succeq \omega \longrightarrow \sigma \ominus \omega \prec \sigma$
   - Consequently, inhaling ω while already holding a full permission $\sigma_\bullet$ is not possible, and neither is exhaling ω while holding the empty permission $\sigma_\bigcirc$.
2. When holding an arbitrary read non-full permission σ to a heap location and inhaling ω, we assume $\sigma \oplus_\downarrow \omega$ and $\sigma \oplus \omega \prec \sigma_\bullet$, i.e. adding a wildcard to a non-full permission yields another non-full permission[27].
3. When holding an arbitrary non-empty permission σ to a heap location and exhaling ω, we assume $\sigma \succeq \omega$ and $\sigma \ominus \omega \succ \sigma_\bigcirc$, i.e. subtracting a wildcard to a non-empty permission yields another non-empty permission. Note that this includes the case where σ is itself a wildcard.
4. Wildcard instances are generally not equivalent to each other. That is, if ω_1 and ω_2 denote distinct uses of the `wildcard` keyword, then $\mathbf{\Phi} \nvdash \omega_1 = \omega_2$.

A consequence of this behaviour is that it is possible to inhale (starting with a non-full permission) and exhale (starting with a non-empty permission) an

[27] This is slightly misleading. The implementation does not assume $\sigma \oplus \omega \neq \sigma_\bullet$ until we try to add another wildcard. But since it is always possible to introduce this assumption, it is sensible to accept it as given. On the converse, it is not possible, without explicitly doing so, to introduce the assumption $\sigma \oplus \omega = \sigma_\bullet$.

arbitrarily large number of wildcard instances. Note also that adding and then subtracting a wildcard does not restore the original permission amount, rather we just end up with another unknown non-empty permission, akin to exhaling the current permission and inhaling a fresh wildcard.

4.7.1 Compatibility with General Permission Models

Our first intuition is that wildcards lead to possible unsoundness. Consider counting permissions C, where the following holds:

$$\forall a : C.\, [-1] \succ a \longrightarrow a = \mathsf{u} \tag{4.1}$$

That is, there are no permissions 'between' -1 and the empty permission. Now, let $\sigma = [-1] \ominus \omega$. We know -1 is non-empty, so by property 3, we may assume $[-1] \succeq \omega$, so σ is well-defined. But then, by properties 1 and 3 we get $[-1] \succ \sigma$ and also $\sigma \succ \mathsf{u}$, which implies $\sigma \neq \mathsf{u}$, leading to a contradiction.

In fact, any permission model that does not possess the property of infinite splittability (cf. Section 2.1.6) is incompatible with this interpretation of wildcards. However, the axioms can give us some breathing room; the above statement is generally not provable from the lean set of axioms we commonly define. Thus, analogous to non-standard Peano arithmetic [45], we can find a non-standard model for the axioms that does allow for wildcards. We will discuss one such approach in the following section.

4.7.2 An Algebraic Solution

It is actually possible to model almost all properties of wildcards using an augmented separation algebra, as presented below:

Definition 4.7.1 (Wildcard-Extension)**.** Let $\langle A, \oplus, \mathsf{u}\rangle$ be a separation algebra. The wildcard-extension of $\langle A, \oplus, \mathsf{u}\rangle$ is a separation algebra $\langle A + \{\omega^+, \omega^-\}, [\mathsf{u}]\rangle$, where:

$$\forall a : A, b : A.\, [a] \mathbin{\widehat{\oplus}} [b] = [a \oplus b] \tag{4.2}$$

$$\forall a : A, b : A.\, [a] \mathbin{\widehat{\oplus_\downarrow}} [b] \longleftrightarrow a \oplus_\downarrow b \tag{4.3}$$

$$\forall a : A, b : A.\, [a] \mathbin{\widehat{\ominus}} [b] = [a \mathbin{\widehat{\ominus}} b] \tag{4.4}$$

$$\forall a : A, b : A.\, [a] \mathbin{\widehat{\succeq}} [b] \longleftrightarrow a \succeq b \tag{4.5}$$

$$\forall a : A.\, a \mathbin{\widehat{\oplus_\downarrow}} \omega^+ \longrightarrow a \oplus \omega^+ \neq [\sigma_\bullet] \wedge a \oplus \omega^+ \succ a \tag{4.6}$$

$$\forall a : A + \{\omega^+, \omega^-\}.\, a \neq [\sigma_\bullet] \longrightarrow a \mathbin{\widehat{\oplus_\downarrow}} \omega^+ \wedge \omega^+ \mathbin{\widehat{\oplus_\downarrow}} a \tag{4.7}$$

$$\forall a : A + \{\omega^+, \omega^-\}.\, a \mathbin{\widehat{\succeq}} \omega^- \longrightarrow a \ominus \omega^- \neq [\mathsf{u}] \wedge a \succ a \ominus \omega^- \tag{4.8}$$

$$\forall a : A + \{\omega^+, \omega^-\}.\, a \neq \mathsf{u} \longrightarrow a \succeq \omega^- \tag{4.9}$$

In plain English, these axioms lift all existing join semantics and predicates into the new algebra, and encode properties 1-3 from the introductory paragraph. Note that we have to use two distinct wildcard elements: one with only the join defined (which is used for wildcard inhales) and one with only inverse joins defined (used for exhales). This is to avoid reductions like $(\sigma \oplus \omega) \ominus \omega = \sigma$. Property 4 however is violated, specifically, $\sigma \oplus \omega^+$ will provably equal $\sigma \oplus \omega^+$ for all σ. Additionally, we can also always prove $\sigma \oplus \omega^+ \neq \sigma_{\bullet}$, as previously alluded to in a footnote.

4.7.3 Evaluation

Similar to the question of how to handle undefined joins, the approach to implementing wildcards is a design choice. One can either go for the slightly modified semantics of algebraic wildcards, or one could leave it up to the implementation to ensure the functionality of wildcards, potentially at the cost of soundness (remember, the user could add something like equation 4.1 to the axioms which the implementation cannot guard against). Due to time constraints, this thesis does not fully address the question, mostly porting the behaviour of the previous implementation. Potential issues with soundness therefore persist at this point in time.

4.8 Solutions to Predicate Multiplication

We propose two potential solutions to facilitate predicate support. One, there are permission models, which have a quasi-multiplicative operation. One such model which we will examine are tree shares. A second approach takes the form of special function-valued permissions exclusively used in predicates.

4.8.1 Tree Multiplication

Le and Hobor [44] have proposed a multiplication-like operation. They define $\tau_1 \otimes \tau_2$ to be the tree obtained by replacing every full leaf $\bullet$ in τ_1 with τ_2. This operation is right-associative, has identity $\bullet$ and null point $\circ$, and is right-distributive over the join, i.e. $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$. The authors argue that this operation can be used to handle Viper-style abstract predicates. However, while we have successfully added this operation to our axiomatisation of the tree share model, we do not as of yet know if the results from Le and Hobor's paper translate directly to our implementation. The reason for this concern is the fact that $\otimes$ does not respect tree congruence in its left argument, that is even if $\tau_1 \cong \tau_2$, we generally do not have $\tau_1 \otimes a = \tau_2 \otimes a$. Investigating this interaction is left for future work.

4.8.2 Split Functions

We do not currently know whether the full algebraic richness of multiplication (commutativity, distribution over addition, etc), over rationals is required for Viper's predicates to function. However, if the following minimal requirements prove sufficient, the approach proposed in the following might be applicable to a wide range of permission models. The requirements on a potential multiplication operator $\otimes$ are as follows:

1. There exists an identity element e_1, such that for all permissions p, $p \otimes e_1 = p$.
2. There exists a null point e_0, such that for all permissions p, $p \otimes e_0 = \mathsf{u}$.
3. If $p \succ \mathsf{u}$, then there exists some element e and share p', such that $p \otimes e = p'$ and $p' \succ p$.
4. If $p \otimes e = p'$ for some element e and shares p, p', then there exists an element $\bar{e}$ such that $(p \otimes e) \oplus (p \otimes \bar{e}) = p$.

Together, these requirements ensure that any element p can be 'split apart' into two parts (by multiplying with appropriate elements) which recombine to p through the join operation. At the trivial end, we can at least split each element into itself and the unit. Non-trivial splits are compatible only with infinitely splittable separation algebras. Note that we do not require that the right-hand operands e belong to the set of shares. We propose a minimal set $\{e_0, e_1, e_l, e_r\}$ of such operands, with which we define the functions $\mathsf{id}(p) \stackrel{\text{def}}{=} p \otimes e_1 \stackrel{\text{def}}{=} p$, $\mathsf{kill}(p) \stackrel{\text{def}}{=} p \otimes e_0 \stackrel{\text{def}}{=} \mathsf{u}$, $\mathsf{split}_l(p) \stackrel{\text{def}}{=} p \otimes e_l$, $\mathsf{split}_r(p) \stackrel{\text{def}}{=} p \otimes e_r$, with the attached axiom $\mathsf{split}_l(p) \oplus \mathsf{split}_r(p) = p$.

We see that the pairs e_1, e_0 and e_l, e_r fulfil requirement 4. The only loose end is the definition of split_l and split_r. However, the requirements we have stated are lax enough that finding such 'canonical splits' is possible in many models. For instance, in fractional permissions, the canonical split could be $\mathsf{split}(p) \stackrel{\text{def}}{=} (\mathsf{split}_l(p), \mathsf{split}_r(p)) = (p * 1/2, p * 1/2)$, whereas in tree shares the split[28] $\mathsf{split}(\tau) = (\tau \otimes_{\mathbb{T}} \langle \bullet, \circ \rangle, \tau \otimes_{\mathbb{T}} \langle \circ, \bullet \rangle)$ would satisfy the requirements.

The way such splits would be employed would be as a replacement for the multiplication operator in the body of predicates. To illustrate this, suppose `P(x: Ref)` is the name of some predicate over a reference, with some field f. In the body of `P`, we would have:

- `P(x)` is short for `acc(P(x), id)`
- `acc(P(x), splitL)` is used instead of e.g. `acc(P(x), 1/2)`

[28] Here, the operator $\otimes_{\mathbb{T}}$ refers to the tree multiplication operation discussed above.

Conclusion The proposed model is highly speculative at this point. More investigation into the exact nature and requirements on a sound implementation of abstract predicates in general permission models is needed.

Chapter 5

Concrete Models

In this chapter, we will discuss the concrete axioms for counting shares and tree shares that were used for the integration with Viper. We provide a soundness argument for each axiom by referring to the underlying model, and proving that the axiom is true therein.

5.1 Logical System

The logic used to express the axioms and correctness proofs is based on the proof theory of the SMT-solver underlying the implementation. It can be thought of as a many-sorted logic, with sorts Int, Bool, Real which have built-in semantics. Additional sorts may be declared to instantiate arbitrary theories. Each n-ary function symbol F has an associated signature $\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \to \mathcal{T}$ declaring the sorts of each argument as well as the result. Any n-ary relation $\mathcal{R}$ between sorts $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is actually a function with signature $\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \to \mathsf{Bool}$, where the result is True if and only if the given tuple is in the relation.

We have built-in arithmetic operators $+, -, \cdot, \div$ which have signatures $\mathsf{Num} \times \mathsf{Num} \to \mathsf{Num}$ where Num is either Real or Int. They have the usual semantics, including commutativity and associativity of addition and multiplication, as well as distributivity of multiplication over addition. There is also a unary negation operation which uses the same symbol as subtraction.

Furthermore, we have comparison operators $\leq, <, \geq, >$ which are relations over the numeric types with the usual semantics. Finally, we have a polymorphic (signature $\mathcal{T} \times \mathcal{T} \to \mathsf{Bool}$ for any sort $\mathcal{T}$) equality relation $=$.

We also have the familiar boolean operators $\wedge, \vee, \to, \neg$ which have the signatures $\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$ and $\mathsf{Bool} \to \mathsf{Bool}$ (for $\neg$).

The last primitive we have are variables (in the following denoted by lowercase letters), which each have a sort annotation. Expressions (terms) and

statements are built up in the expected fashion:

- Variables are expressions belonging to the sort identified by the variable annotation, denoted $v : \mathcal{T}$.
- Literal symbols such as $0, 1, -1, 2, -2, \ldots, 0.0, 0.5, \ldots$, as well as True and False are expressions belonging to Int, Real and Bool respectively.
- If $\mathcal{E}_1 : \mathcal{T}_1, \ldots, \mathcal{E}_n : \mathcal{T}_n$ are expressions and F is an n-ary function symbol with signature $\mathcal{T}_1 \times \cdots \times \mathcal{T}_n \to \mathcal{T}$, then $F(\mathcal{E}_1, \ldots, \mathcal{E}_n) : \mathcal{T}$ is an expression.
- Statements are expressions belonging to the sort Bool.
- If $\mathcal{E}$ is a statement, then $\forall v : \mathcal{T}. \mathcal{E}$ and $\exists v : \mathcal{T}. \mathcal{E}$ are statements for each sort $\mathcal{T}$ and any variable v if and only if each free occurrence of v in $\mathcal{E}$ belongs to the sort $\mathcal{T}$.

The takeaway here is that we only consider well-formed statements with respect to the given function signatures. The derivation rules are the same as in classical logic. For the following sections, all free variable occurrences are assumed to be appropriately $\forall$-quantified in the interest of visual clarity.

5.2 Identity Extension

We recall that the unit extension demands that the extended algebra is a cancellative semigroup. Our model for the extension is the disjoint sum of the original carrier set A and the singleton set $\{\mathsf{u}\}$. We assume that $\langle A, \oplus_A \rangle$ provides a model for the function symbols $\oplus_A, \oplus_{\downarrow}^{A}, \ominus_A$ and $\succeq_A$. We declare a new sort $\mathbb{S}$, as well as the unary function symbol $\mathsf{inl} : A \to \mathbb{S}$, where again we use the shorthand $[a]$ instead of $\mathsf{inl}(a)$.

The introduction function inl is injective, and its image disjoint from $\{\mathsf{u}\}$ justifying the axioms:

$$\forall a : A, b : A. [a] = [b] \longrightarrow a = b \tag{5.1}$$

$$\forall a : A. [a] \neq \mathsf{u} \tag{5.2}$$

Under these assumptions, we are expecting the lifted $\oplus_A$ operation to behave as in the original algebra, justifying:

$$\forall a : A, b : A. [a] \oplus [b] = [a \oplus_A b] \tag{5.3}$$

Analogous justifications are valid for the functions $\oplus_{\downarrow}^{A}, \ominus_A$ and $\succeq_A$. Because A and $\{\mathsf{u}\}$ are necessarily disjoint, we can be sure that defining a behaviour for u under $\oplus$ will not clash with the previous axiom, that is:

$$\forall x : \mathbb{S}. x \mathbin{\widehat{\oplus}} \mathsf{u} = x \wedge \mathsf{u} \mathbin{\widehat{\oplus}} x = x \tag{5.4}$$

The same holds for the dependent axioms for $\succeq$ and $\oplus_{\downarrow}$.

One restriction we must observe is that while A has no identity element for all elements, there may still be $a : A, b : A$ with $a \oplus_A b = a$. Thus the following axiom can introduce a contradiction:

$$\forall a : A.\, [a] \ominus [a] = \mathsf{u} \tag{5.5}$$

But because we want the extended model to be a separation algebra, the above must hold. The only solution is to demand that there may exist no $a : A, b : A$ such that $a \oplus b = a$.

5.3 Counting Shares

5.3.1 Model

Our model for counting permissions is the one described in the background chapter of this thesis, with one difference. We will only prove soundness for the algebra without the added unit element. This algebra, as previously stated, is a semigroup and thus not a separation algebra. We then rely on the correctness of the unit extension to ensure the soundness of the full axiom set. What this means for our axiomatisation is that we will use the built-in sort Int for counting shares.

5.3.2 Join

We recall that there are two cases for the join: we can join token bundles with token bundles and token bundles with factories (i.e. the factory reabsorbs the tokens). For the second case, we have the side condition that the result must be another token factory, i.e. a factory cannot absorb more tokens than it has issued. We thus declare two auxiliary function symbols, case_T and case_F, which are binary predicates over shares, meaning they have signature $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$. Their interpretation is that case_T is true if and only if both operands are token bundles, and case_F is true if and only if its left operand is a token bundle and its right operand is a token factory capable of reabsorbing its left argument. We therefore state:

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\, \mathsf{case}_T(m, n) \longleftrightarrow m < 0 \wedge n < 0 \tag{5.6}$$

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\, \mathsf{case}_F(m, n) \longleftrightarrow m < 0 \wedge n > 0 \wedge m + n \geq 0 \tag{5.7}$$

The definition of the join is now a straightforward matter. It is required that one of the two cases holds, and then joining is a matter of simple addition.

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\, m \oplus_{\downarrow} n \longleftrightarrow \mathsf{case}_F(m, n) \vee \mathsf{case}_F(n, m) \vee \mathsf{case}_T(m, n) \tag{5.8}$$

Note that we have to state both argument permutations of case_F to ensure the function commutes.

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ m \oplus_{\downarrow} n \longrightarrow m \oplus n = m + n \tag{5.9}$$

5.3.3 Inverse Join

The tricky part with the inverse join is the ordering $\succeq$, not the inverse itself, which is as easy as stating:

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ m \succeq n \longrightarrow m \ominus n = m - n \tag{5.10}$$

For the definition of the ordering, we think back to our two join cases. We know that if l in $l \succeq r$ must either be a token bundle or a token factory. In the first case, r must also be a token bundle, and must contain strictly fewer tokens than l, because there are no empty token bundles. This translates into $l < r$ since token bundles are represented by negative integers. In the second case (l being a factory), there are two possibilities:

1. r is a token bundle. This case has no other side conditions, because a factory can issue an arbitrary amount of tokens.
2. r is a token factory. In this case there must exist a token bundle x, such that $l = r + x$, or in other terms, $x = r - l$. Because token bundles are negative, we get the requirement $r - l < 0$, or $l < r$ for short.

For convenience, we again define two predicates, case_T^{-1} and case_F^{-1}. We can convince ourselves that their behaviour is modeled by the following axioms:

$$\forall r : \mathsf{Int}, l : \mathsf{Int}.\ \mathsf{case}_T^{-1}(l, r) \longleftrightarrow l < 0 \wedge r < 0 \wedge l < r \tag{5.11}$$

$$\forall r : \mathsf{Int}, l : \mathsf{Int}.\ \mathsf{case}_F^{-1}(l, r) \longleftrightarrow l \geq 0 \wedge (r < 0 \vee r \geq 0 \wedge r < l) \tag{5.12}$$

Finally, we define:

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ m \succeq n \longleftrightarrow \mathsf{case}_F^{-1}(m, n) \vee \mathsf{case}_T(m, n) \tag{5.13}$$

5.4 Tree shares

5.4.1 Model

The model we use to prove the soundness of our axiomatization are the tree shares described in the background chapter of this thesis, augmented with the joinability relation, which contains all pairs of trees that have a defined join, and the inverse join operation. We will denote this model as $\mathbf{M}$.

5.4.2 Basics

Type Declaration We start the axiomatization by declaring a type $\mathbb{T}$.

Common Function Symbols We declare the function symbols $\cong, \succeq, \oplus_{\downarrow}$ of type $\mathbb{T} \times \mathbb{T} \to \text{Bool}$, as well as $\oplus, \ominus$ of type $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$, which have the familiar interpretations.

Atom Constants We declare the constant symbols for the full $\bullet$ and empty $\circ$ node. Using the keyword `unique`, we implicitly state:

$$\neg(\bullet = \circ) \tag{5.14}$$

Using the interpretation of equality as

$$\mathbf{M} \models A = B \text{ if and only if } A \text{ is the same object as } B$$

we see that this axiom indeed does hold in $\mathbf{M}$, since the full and empty node are not the same.

Node Constructor We define the function symbol `node` of type $\mathbb{T} \times \mathbb{T} \to \mathbb{T}$. We will use $\langle A, B \rangle$ as a notational shorthand for $\texttt{node}(A, B)$. The interpretation of this symbol is the function that takes trees τ_1, τ_2 and produces the tree that has τ_1 as is left child and τ_2 as its right child.

5.4.3 Equality and Congruence Axioms

Equality Relation

Boogie implicitly provides an inductive equality relation over trees. By reflexivity of $=$ we get $\bullet = \bullet$ and $\circ = \circ$, as well as $a_1 = b_1 \wedge a_2 = b_2 \longleftrightarrow \langle a_1, a_2 \rangle = \langle b_1, b_2 \rangle$ (the reverse direction holds because the node constructing function is invertible in the model, yielding its constituent subtrees).

Congruence Relation

Stand-alone Definition We can define the congruence relation just like originally presented in Chapter 2. We recall that the model for this relation is the following: two trees are congruent to one another, if there is some sequence of leaf unfolding (or folding) for both trees which results in a common tree. To get to a common tree from $\langle a_1, a_2 \rangle$, we simply apply the operations for a_1 on the left subtree and the ones for a_2 on the right subtree (and the analogue for $\langle b_1, b_2 \rangle$). The resulting trees must be equal by definition of the node constructor.

$$a = b \longrightarrow a \cong b \tag{5.15}$$

{**TODO:**These axioms are made redundant by H1-H4}

$$\circ \cong \langle \circ, \circ \rangle \tag{5.16}$$

$$\bullet \cong \langle \bullet, \bullet \rangle \tag{5.17}$$

$$a_1 \cong b_1 \wedge a_2 \cong b_2 \longrightarrow \langle a_1, a_2 \rangle \cong \langle b_1, b_2 \rangle \tag{5.18}$$

Axiom 5.15 holds, as we can simply apply no transformations on both sides to reach a common tree a. Axioms 5.16 and 5.17 can be satisfied by applying a single leaf unfolding to the left hand side. For Axiom 5.18, assume the left-hand side to be true. This means there exist transformations sequences for transforming a_1 and b_1 into a common tree (ditto for a_2 and b_2). Thus if we can construct a sequence of transformations for $\langle a_1, a_2 \rangle$ by applying a_1's transformations and then a_2's. This creates the same tree as the analogous transformation on $\langle b_1, b_2 \rangle$.

Dependent Definition We can define congruence by referring to $\succeq$, as follows:

$$a \cong b \longleftrightarrow a \succeq b \wedge b \succeq a \tag{5.19}$$

Which is true by definition as presented by Dockins et al. [7].

Abstract Properties The following axioms declare that $\cong$ is an equality relation. Reflexivity holds trivially (no operations required), symmetry holds because we can still find a common subtree for b and a if we have one for a and b. Transitivity requires a longer argument: let τ_1 be the common tree reached from a and b, and let τ_2 be the common tree reached from b and c. Let σ_a be the sequence of transformations turning a into τ_1, and let $\bar{\sigma_b}$ be the reverse of the transformations turning b into τ_1. The reverse can be found by reversing the order of transformations and changing each unfolding into a folding and vice versa. The transformation $\bar{\sigma_b}$ thus turns τ_1 into b. Now consider the transformation formed by first applying σ_a and then $\bar{\sigma_b}$, which turns a into b. Thus, b is a common tree reached from a and b. An analogous argument gives us a transformation turning c into b. Thus b is a common tree reachable from both a and c and thus $a \cong c$.

$$a \cong a \tag{5.20}$$

$$a \cong b \longleftrightarrow b \cong a \tag{5.21}$$

$$a \cong b \longrightarrow b \cong c \longrightarrow a \cong c \tag{5.22}$$

5.4.4 Join Axioms

Join Function

The model for the join operator $\oplus$ is the operation which (under the assumptions that both operands have the same shape) performs a leaf-wise logical or-operation. Joining a 'true' leaf $\bullet$ with another $\bullet$ is not possible. By

structural induction, this means the join is undefined over trees which both have a ●-leaf at the same position in their structure. We thus postulate:

$$\circ \oplus \circ = \circ \tag{5.23}$$

$$\circ \oplus \bullet = \bullet \tag{5.24}$$

$$\bullet \oplus \circ = \bullet \tag{5.25}$$

Which mirrors our model. For the recursive case, we have:

$$\forall \tau_a : \mathbb{T}, \tau_b : \mathbb{T}, \tau_c : \mathbb{T}, \tau_d : \mathbb{T}.\ \langle \tau_a, \tau_b \rangle \oplus \langle \tau_c, \tau_d \rangle = \langle \tau_a \oplus \tau_c, \tau_b \oplus \tau_d \rangle \tag{5.26}$$

Joinability Predicate

From our model, we can directly see that the following axioms hold:

$$\circ \oplus_{\downarrow} \circ \tag{5.27}$$

$$\circ \oplus_{\downarrow} \bullet \tag{5.28}$$

$$\bullet \oplus_{\downarrow} \circ \tag{5.29}$$

$$\neg(\bullet \oplus_{\downarrow} \bullet) \tag{5.30}$$

$$\forall \tau_a : \mathbb{T}, \tau_b : \mathbb{T}, \tau_c : \mathbb{T}, \tau_d : \mathbb{T}.\ \langle \tau_a, \tau_b \rangle \oplus_{\downarrow} \langle \tau_c, \tau_d \rangle \longleftrightarrow \langle \tau_a \oplus_{\downarrow} \tau_c, \tau_b \oplus_{\downarrow} \tau_d \rangle \tag{5.31}$$

5.4.5 Inverse Join Axioms

Inverse Join Function

The inverse join can be seen as a leaf-wise subtraction, if ● represented 1 and ○ represented 0. Of course, we cannot subtract ● from ○. We thus state:

$$\circ \ominus \circ = \circ \tag{5.32}$$

$$\bullet \ominus \circ = \bullet \tag{5.33}$$

$$\bullet \ominus \bullet = \circ \tag{5.34}$$

With the expected recursive case:

$$\forall \tau_a : \mathbb{T}, \tau_b : \mathbb{T}, \tau_c : \mathbb{T}, \tau_d : \mathbb{T}.\ \langle \tau_a, \tau_b \rangle \ominus \langle \tau_c, \tau_d \rangle = \langle \tau_a \ominus \tau_c, \tau_b \ominus \tau_d \rangle \tag{5.35}$$

Ordering Relation

In our model, the ordering relation more or less says that a tree τ_1 is greater than τ_2, if τ_1 has $\bullet$ at least at all positions where τ_1 has them (again assuming the same shape for both operands). We thus have:

$$\circ \succeq \circ \tag{5.36}$$

$$\neg(\circ \succeq \bullet) \tag{5.37}$$

$$\bullet \succeq \circ \tag{5.38}$$

$$\bullet \succeq \bullet \tag{5.39}$$

$$\forall \tau_a : \mathbb{T}, \tau_b : \mathbb{T}, \tau_c : \mathbb{T}, \tau_d : \mathbb{T}.\ \langle \tau_a, \tau_b \rangle \succeq \langle \tau_c, \tau_d \rangle \longleftrightarrow \langle \tau_a \succeq \tau_c, \tau_b \succeq \tau_d \rangle \tag{5.40}$$

5.4.6 Multiplication Axioms

We recall from section 4.8 that in the tree share model, the operation $\tau_1 \otimes \tau_2$ is defined as replacing each occurrence of $\bullet$ in τ_1 with τ_2. The following axioms model this operation:

$$\forall \tau : \mathbb{T}. \circ \otimes \tau = \circ \tag{5.41}$$

$$\forall \tau : \mathbb{T}. \bullet \otimes \tau = \tau \tag{5.42}$$

$$\forall \tau : \mathbb{T}, a : \mathbb{T}, b : \mathbb{T}.\ \langle a, b \rangle \otimes \tau = \langle a \otimes \tau, b \otimes \tau \rangle \tag{5.43}$$

5.4.7 Operations Modulo Congruence

Essentially all operations over trees (multiplication being the only exception) that we consider here have assumed that all operands have the same shape. We have skated over what exactly this means, and will now justify this convention. What it means exactly for two trees to have the same shape is the following:

1. If L_1 and L_2 are leaves, then L_1 has the same shape as L_2.
2. If L is a leaf, then it does not have the same shape as $\langle \tau_1, \tau_2 \rangle$ for all trees τ_1, τ_2.
3. The trees $\langle \tau_a, \tau_b \rangle$ and $\langle \tau_c, \tau_d \rangle$ have the same shape if and only if τ_a has the same shape as τ_c and τ_b has the same shape as τ_d.

To see why we can require operands with the same shape, consider the following informal function definition:

1. $\mathsf{common}(L_1, L_2)$ returns (L_1, L_2) for leafs L_1, L_2.
2. $\mathsf{common}(L, \langle \tau_1, \tau_2 \rangle)$ returns $\mathsf{common}(\langle L, L \rangle, \langle \tau_1, \tau_2 \rangle)$ when L is a leaf.

3. $\mathsf{common}(\langle \tau_a, \tau_b \rangle, \langle \tau_c, \tau_d \rangle)$ returns $(\langle \tau'_a, \tau'_b \rangle, \langle \tau'_c, \tau'_d \rangle)$ where $(\tau'_a, \tau'_c) = \mathsf{common}(\tau_a, \tau_c)$ and $(\tau'_b, \tau'_d) = \mathsf{common}(\tau_b, \tau_d)$.

Employing an inductive argument, we quickly see that if $\mathsf{common}(\tau_1, \tau_2)$ returns (τ'_1, τ'_2), then τ'_1 has the same shape as τ'_2, but also $\tau_1 \cong \tau'_1$ and $\tau_2 \cong \tau'_2$.

With that out of the way, we can get more precise about what it means that a function 'expects arguments to have the same shape'. If F is such a function[29], then we have:

$$\text{If } \mathsf{common}(\tau_1, \tau_2) \text{ returns } (\tau'_1, \tau'_2), \text{ then } F(\tau_1, \tau_2) \text{ equals } F(\tau'_1, \tau'_2) \tag{5.44}$$

This requirement in turn demands that F indeed does not discriminate over $\cong$, that is:

$$\text{If } \tau_1 \cong \tau'_1 \text{ and } \tau_2 \cong \tau'_2 \text{ then } F(\tau_1, \tau_2) \text{ equals } F(\tau'_1, \tau'_2) \tag{5.45}$$

The question we want to answer next, is how we can axiomatise these meta-theoretic assumptions. The way we do this is by implicitly applying the common procedure to all uses of the function symbol F. Due to the structure of trees, we usually have a number of axioms covering tree atoms, i.e. axioms of the form $F(L_1, L_2) = \mathcal{E}$ for leafs L_1, L_2 and some right hand side expression $\mathcal{E}$. The other case is recursive, producing axioms of the form $\forall \tau_a : \mathbb{T}, \tau_b : \mathbb{T}, \tau_c : \mathbb{T}, \tau_d : \mathbb{T}. F(\langle \tau_a, \tau_b \rangle, \langle \tau_c, \tau_d \rangle) = \mathcal{E}$ for some expression $\mathcal{E}$ which typically contains the sub-expressions $F(\tau_a, \tau_c)$ and $F(\tau_b, \tau_d)$. We simply add to this system of axioms the following:

$$\forall \tau_1 : \mathbb{T}, \tau_2 : \mathbb{T}. F(\bullet, \langle \tau_1, \tau_2 \rangle) = F(\langle \bullet, \bullet \rangle, \langle \tau_1, \tau_2 \rangle) \tag{H1}$$

$$\forall \tau_1 : \mathbb{T}, \tau_2 : \mathbb{T}. F(\circ, \langle \tau_1, \tau_2 \rangle) = F(\langle \circ, \circ \rangle, \langle \tau_1, \tau_2 \rangle) \tag{H2}$$

$$\forall \tau_1 : \mathbb{T}, \tau_2 : \mathbb{T}. F(\langle \tau_1, \tau_2 \rangle, \bullet) = F(\langle \tau_1, \tau_2 \rangle, \langle \bullet, \bullet \rangle) \tag{H3}$$

$$\forall \tau_1 : \mathbb{T}, \tau_2 : \mathbb{T}. F(\langle \tau_1, \tau_2 \rangle, \circ) = F(\langle \tau_1, \tau_2 \rangle, \langle \circ, \circ \rangle) \tag{H4}$$

We argue that these axioms are sound, as they are modeled by functions which are equal under congruent arguments. Furthermore, they are necessary as otherwise no result could be derived for such functions if the arguments do not have the same shape.

Instantiations Axioms H1-H4 are instantiated for the following functions: $\cong, \oplus, \oplus_{\downarrow}, \ominus, \succeq$. We have omitted them in the relevant sections in the interest of conciseness. After all, the soundness argument is analogous for all of them.

[29] Here we only consider binary function symbols, but the common procedure can readily be extended to an arbitrary number of operands.

Chapter 6

Viper Interface

6.1 Overview

One of the stated goals at the outset of this thesis was to not only bring alternative permission models to the Carbon back-end, but to provide an interface for user-specified permission models in the Silver language. The building blocks were already there: through the domain module (see Section 2.2.8), we can specify all the definitions and axioms we need. The challenge was to integrate the provided definitions with the permission module. For this purpose, we defined a set of required definitions, explained in Section 6.2, and extracted the axioms from keyworded special domain to replace the default permission module's axioms.

6.2 Interface Specification

Taking the separation algebra described in Section 4.4 as a starting point, the definition of the interface is simple; the user must provide:

- A function declaration `plus(Perm, Perm) : Perm`.
- A function declaration `joinable(Perm, Perm) : Bool`.
- A function declaration `minus(Perm, Perm) : Perm`.
- A function declaration `geq(Perm, Perm) : Bool`.
- Optionally, a predicate `eq(Perm, Perm) : Bool`. If it is not found, then Boogie's default equality operator is used.
- Optionally, share constructors and other auxiliary functions.
- A set of axioms to define the behaviour of the above functions.

If any of the mandatory signatures are missing (or use the wrong types), the domain is not a valid permission model. Additionally, the permission domain

must have the name '_Perm'. An example use of the proposed interface can be seen in the following listing, which verifies under the new implementation (see Listing 6.1).

6.3 Implementation

6.3.1 Domain Module Overview

From a functional perspective, the domain module is relatively uninteresting: it essentially just takes the parsed axioms and functions declarations (Viper domains do not have constant and type definitions) and emits them more or less directly into the Boogie file. It helps that the Silver syntax for functions and axioms is essentially identical with Boogie's, except that Silver axioms can be named. The domain module also maintains a separate namespace, something we will have to change as the permission module needs to have the same namespace as the domain of user-specified permissions, such that the former can use the definitions of the latter.

6.3.2 Enabling Custom Permissions

We added a command line option to Carbon named `permModel` for ease of testing the different models. The following arguments are accepted:

- `fractions`: the default fractional permission model is used.
- `domain_specified`: the verifier looks for a domain named `_Perm`, checks if it fulfils the requirements and uses the new `DomainPermModule`.
- `newfractions`: fractional permissions via the general permission module.
- `counting`: counting permissions via the general permission module.
- `trees`: tree permissions via the general permission module.
- If no argument is provided, the default (fractional) permission module is used.

6.3.3 Switching the Permission Module

Depending on the command line arguments provided, a different permission module is used by the verifier. This 'live switching' requirement was not considered in the original software architecture, which assumes that all modules are statically known. The main reason for this choice are the numerous dependencies between modules that exist despite valiant efforts to separate the code. One such example is the permission type used at various points in the verifier. A solution was found by treating the verifier class'

```
field f : Int

method main(x: Ref)
requires acc(x.f, fullShare())
ensures acc(x.f)
{
x.f := square(x)
}

function square(x: Ref) : Int
requires acc(x.f, share(1/2))
{
x.f * x.f
}

domain _Perm {
    function unitShare(): Perm
    function fullShare(): Perm
    function plus(x: Perm, y: Perm): Perm
    function minus(x: Perm, y: Perm): Perm
    function geq(x: Perm, y: Perm): Bool
    function joinable(x: Perm, y: Perm): Bool

    //Optional Functions
    function share(x: Rational): Perm
    function valid_share(x: Rational): Bool

    //Axioms
    axiom axValid { forall x: Rational ::
        valid_share(x) <==> (x >= 0/1 && x <= 1/1)
    }
    axiom axUnit { unitShare() == share(0/1) }
    axiom axFull { fullShare() == share(1/1) }
    axiom axInj { forall a: Rational, b: Rational ::
        share(a) == share(b) ==> (a == b)
    }
    axiom axPlus { forall a: Rational, b: Rational ::
        joinable(share(a), share(b)) ==>
        plus(share(a), share(b)) == share(a + b)
    }
    axiom axMinus { forall a: Rational, b: Rational ::
        geq(share(a), share(b)) ==>
        minus(share(a), share(b)) == share(a - b)
    }
    axiom axCanJoin { forall a: Rational, b: Rational ::
        joinable(share(a), share(b)) <==>
        valid_share(a) && valid_share(b) && valid_share(a + b)
    }
    axiom axGeq { forall a: Rational, b: Rational ::
        geq(share(a), share(b)) <==>
        valid_share(a) && valid_share(b) && valid_share(a - b)
    }
}
```

Listing 6.1: Simplified encoding of fractional permissions via domain module

permission module field as a variable, and making extensive use of Scala's `lazy` keyword to avoid null-references.

Automatic Module Switching

In the current implementation, switching the permission module to the domain-specified variant requires both a domain with the keyworded name `_Perm` and the appropriate command option to be provided. This is to facilitate easier testing, as the proposed extension is still experimental. In principle, using the new module could be as simple as checking for existence of an appropriately named domain.

6.4 Untangling Rationals from Permissions

Although the general approach used in this thesis was to leave a small fingerprint on the Silver language specification as possible, there is one major change that was essentially unavoidable. Though the Viper documentation says that rational numbers are a built-in type, this is only true insofar as `Rational` is a type synonym for `Perm`. As mentioned before, the permission interface leaves Silver's grammar as it is, and simply changes the semantics of the `Perm` type through the back-end. However, a consequence of the aliasing between `Rational` and `Perm` is that if we change the semantics of `Perm`, then rationals are also affected. Therefore it was necessary to introduce `Rational` as an actual type to the parser. A consequence of this change is that Viper programs that assigned `Perm` l-values to `Rational` typed r-values or vice versa will now be rejected by the type checker. Such assignments would have been legal under the previous implementation.

6.5 Possible Extensions

6.5.1 Quality of Life Features

While the domain-specified permission feature provides all necessary functionality to encode arbitrary permission models, there are a number of reusable extensions that could be provided in the implementation, such as:

- Product separation algebras,
- function separation algebras,
- unit extensions,
- disjoint sum separation algebras and
- boilerplate axioms such as commutativity, cancellation, etc.

Building these compositional features by hand is feasible, but tedious. Given some indication by the user, the back-end could automatically construct these extensions. If the extension is verified, the user must worry only about the correctness of the core encoding instead of the entire construction. However, it is not clear what the interface for these constructions looks like. The most straightforward way would be to declare a keyworded function in the `_Perm` domain. For instance, a function `Product(`A_1`,` A_2`)` could be declared, where A_1, A_2 are the names of the two component separation algebras which the user has previously defined. The benefit of such a solution is that no changes to the syntax would have to be introduced, at the cost of overloading an existing concept in the language. For now, we leave this as a possibility for future work.

6.5.2 Automated Checks

As we have seen in Chapter 3, ensuring that a set of axioms is sound yet complete enough is a somewhat involved task. It might therefore be worthwhile to explore ways in which the implementation can reduce the burden on the user. One such way are automated soundness and completeness checks. While soundness is not something the implementation can generally prove, we can nonetheless ensure that some basic contradictions are absent. This could take the form of an auto-generated method with the postcondition `false`, which could perform some share operations to give the SMT-solver a chance at deriving a contradiction, if it exists. Given the proper annotations, additional semantic analysis could be performed on the axioms to find possible contradictions, e.g. in the definitions of functions with case distinctions as outlined in Section 3.7.

Second, a number of checks may be performed to ensure that some basic identities (e.g. $a \oplus b \longleftrightarrow b \oplus a$) can be derived. This also could take the form of an automatically generated method with a number of assertions that produce appropriate error messages if necessary properties of the algebras are not provable.

Chapter 7

Conclusion

7.1 Conclusion

We have surveyed the landscape of permission models in the literature and identified interesting alternatives to extend the Viper verification language with. We have developed practical techniques for embedding permission models in the logic of the Boogie verifier, and have identified necessary completeness conditions on the embeddings which ensure the functionality of Viper's essential separation logic constructs. We have proposed permission model extensions aimed at supporting Viper's predicate and wildcard features. We provided soundness proof sketches for selected permission models and extensions.

Furthermore, we have implemented a generalized permission module for the Carbon back-end, as well as an interface for direct use with the Viper language, enabling users to specify arbitrary permission models alongside a program specification, and have provided embeddings of select permission models. In doing this, we have separated the handling of types `Perm` and `Rational` by the language parser, allowing these types to have different semantics. As of this moment, the proposed changes to the implementation should be considered experimental, with rigorous testing still outstanding, as further explained below.

7.2 Future Work

Over the course of working on this project, it seemed as if for every explored possibility, three new ones pop up. Time constraints have prevented us from examining the following avenues.

7.2.1 Correctness Testing

While a lot of effort went into ensuring that the implementation of the generalised permission module is correct, we have not passed the full suite of Carbon tests. This is mostly due to the fact that a lot of old viper test files would simply not work with new permissions. Consider tree shares for instance, which have to be specified in entirely different terms than simple fractions (e.g. `tree(full(), empty())` as opposed to `1/2`). Additionally, test files that use advanced Viper features not yet supported by the generalised permission module obviously cannot be expected to work. Combing through the test suite, identifying which tests should be expected to pass, rewriting the test files and then finally looking for and fixing bugs is a time-consuming process that we deprioritized in favour of additional exploration of support for advanced features. Before the proposed changes can become a part of the official Viper infrastructure, this work will have to be done.

7.2.2 Performance Evaluation

One concern immediately raised is the issue of verification performance. One benefit enjoyed by fractional permissions as currently implemented is its simplicity: permission operations usually only require a few rational number additions, subtractions or comparisons, whereas models with more pleasing theoretical properties typically place a higher burden on the SMT-solver owing to their complexity. Tree shares in particular have the potential to serve as an alternative or even replacement for the default fractional permissions. However, such a major change can only be supported if verification speeds are comparable to the current implementation. Over the course of this thesis, testing was only performed on small programs, where performance is not a concern. Rigorous testing will help evaluate the permission models proposed in this thesis.

7.2.3 Predicate Support

Support for Viper's `predicates` is at best experimental at this point. Some separation algebras beside fractions have the potential to provide the necessary multiplication operations, but the exact requirements and assumptions of the implementations are as of yet unknown. To have confidence in full functionality and soundness, additional exploration of this area is necessary.

7.2.4 Magic Wand Support

This thesis has more or less completely ignored magic wands, due to aspects of their implementation which would require substantial modifications to the code base that would not have been feasible to carry out in the allotted

time. Needless to say, magic wands are not supported by the generalised permission module as of yet.

7.2.5 Silicon Compatibility

As stated many times throughout this thesis, we have targeted Viper's Carbon back-end for this project. This means that the symbolic execution back-end, Silicon, still can only accept fractional permissions. Additionally, amongst the changes proposed in this thesis is a separation of the `Perm` and `Rational` types, breaking compatibility with Silicon, which still assumes that the two types are aliases. Even if full generalised permission support is off the table, some work will have to go into ensuring that the back-ends agree on the language specification, which is a software package shared between the two.

7.2.6 Formal Verification of Axioms

The soundness proofs of the extensions and axiom sets presented in this thesis, while making an attempt at formal rigour, fall short of machine-checked Coq or Isabelle/HOL proofs. The premium on soundness is especially high in a verification tool like Carbon, and hand-written proofs and procedures are known to be lacking in this regard: Le et al. [31] report finding a significant number of errors in their treatment of tree shares after subjecting their procedures to a machine proof in Coq.

Appendix A

Counting Permissions with Undefined Extension

This model was the first that was implemented and gave the author quite a bit of trouble, on one hand in the form of contradictions in the axioms, and in the form of too weak axioms. This is noteworthy, as a potential user of the final permission module will have to wrestle with these difficulties too, unless a simplifying interface is given. In the following, we describe our systematic approach to formulating the axioms. The idea is to not miss any potential result, by dividing every possible input to the share operations into equality classes w.r.t the result.

Tables Let, $m > 0, n > 0$. The join conditions look as follows (conditions below diagonal are symmetrical due to commutativity of the join). The results of the join are as follows:

$\oplus$	u	n	0	$-n$	$\bot$
u	u	n	0	$-n$	$\bot$
m		$\bot$	$\bot$	$m+(-n)$ if $m+(-n) \geq 0$ else $\bot$	$\bot$
0			$\bot$	$\bot$	$\bot$
$-m$				$(-m)+(-n)$	$\bot$
$\bot$					$\bot$

The table for $m \succeq n$ is as follows:

$\succeq$	u	n	0	$-n$	$\bot$
u	$\top$	$\bot$	$\bot$	$\bot$	$*$
m	$\top$	$m-n<0 \vee m=n$	$\bot$	$\top$	$*$
0	$\top$	$\top$	$\top$	$\top$	$*$
$-m$	$\top$	$\bot$	$\bot$	$(-m)-(-n)<0 \vee m=n$	$*$
$\bot$	$*$	$*$	$*$	$*$	$*$

The additional $m = n$ clauses stem from the fact that thanks to the unit, $m \geq m$ for all m. The $*$ symbol defines an undefined result.Thus, the inverses are defined as follows:

$\ominus$	u	n	0	$-n$	$\bot$
u	u	$\bot$	$\bot$	$\bot$	$\bot$
m	m	u if $m = n$ else $m - n$	$\bot$	$m - (-n)$	$\bot$
0	0	$-n$	u	n	$\bot$
$-m$	$-m$	$\bot$	$\bot$	u if $m = n$ else $(-m) - (-n)$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Writing the Axioms - Join Looking at the tables above, our challenge is twofold: On one hand, we may not leave any of the defined results undefined, as each one may be needed while on the other hand, we may not have any contradictory definitions for the same inputs, nor define any of the fields marked with $*$. Thus I will mark which entries have been defined by each axiom.

Starting off, we define:

$$P_A(m,n) \longleftrightarrow m < 0 \wedge n < 0$$

$$P_B(m,n) \longleftrightarrow m > 0 \wedge n < 0 \wedge m + n \geq 0$$

so we can postulate:

$$\forall m : \texttt{int}, n : \texttt{int}.\, P_A(m,n) \vee P_B(m,n) \vee P_B(n,m) \longrightarrow \mathsf{inl}(m) \oplus \mathsf{inl}(n) = \mathsf{inl}(m+n)$$

Looking at our tables, we can cross off the following entries:

$\oplus$	u	n	0	$-n$	$\bot$
u					
m				$\times$ if $m + (-n) \geq 0$	
0					
$-m$		$\times$ if $(-m) + n \geq 0$		$\times$	
$\bot$					

We then declare:

$$\forall m : \texttt{int}, n : \texttt{int}.\, \neg(P_A(m,n) \vee P_B(m,n) \vee P_B(n,m)) \longrightarrow \mathsf{inl}(m) \oplus \mathsf{inl}(n) = \bot$$

And glance down at our table of checkboxes:

$\oplus$	u	n	0	$-n$	$\bot$
u					
m		$\times$	$\times$	$\times$	
0		$\times$	$\times$	$\times$	
$-m$		$\times$	$\times$	$\times$	
$\bot$					

To address the bottom and right side of the table, we declare:

$$\forall a : S.\, a \oplus \bot = \bot \text{ and}$$

$$\forall a : S.\, \bot \oplus a = \bot.$$

This does define $\bot \oplus \bot$ twice, but that is fine because the result is the same both ways. We get:

$\oplus$	u	n	0	$-n$	$\bot$
u					$\times$
m		$\times$	$\times$	$\times$	$\times$
0		$\times$	$\times$	$\times$	$\times$
$-m$		$\times$	$\times$	$\times$	$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

We finally declare:

$$\forall a : S.\, a \oplus \mathsf{u} = a \text{ and}$$

$$\forall a : S.\, \mathsf{u} \oplus a = a.$$

This gives us redundant definitions for $\bot \oplus \mathsf{u}$ and $\mathsf{u} \oplus \bot$, which is again no problem. This completes our join table.

Writing the Axioms - Inverse We start with the axiom:

$$\forall a : S.\, a \neq \bot \longrightarrow a \succeq \mathsf{u}$$

Thus giving us an initial table:

$\succeq$	u	n	0	$-n$	$\bot$
u	$\times$				$*$
m	$\times$				$*$
0	$\times$				$*$
$-m$	$\times$				$*$
$\bot$	$*$	$*$	$*$	$*$	$*$

Note that I deliberately did not define $\bot \succeq \mathsf{u}$. Moving on, we declare:

$$\forall a : S.\, a \neq \bot \longrightarrow a \neq \mathsf{u} \longrightarrow \neg(\mathsf{u} \succeq a)$$

Filling in the remainder of the top row:

$\succeq$	u	n	0	$-n$	$\bot$
u	$\times$	$\times$	$\times$	$\times$	$*$
m	$\times$				$*$
0	$\times$				$*$
$-m$	$\times$				$*$
$\bot$	$*$	$*$	$*$	$*$	$*$

The next axiom is:

$$\forall n : \mathtt{int}.\, \mathsf{inl}(0) \succeq \mathsf{inl}(n)$$

We are of course relying on the prior axiom $\forall n : \mathtt{int}.\, \mathsf{inl}(n) \neq \bot \wedge \mathsf{inl}(n) \neq \mathsf{u}$. We now have the center row filled:

$\succeq$	u	n	0	$-n$	$\bot$
u	$\times$	$\times$	$\times$	$\times$	$*$
m	$\times$				$*$
0	$\times$	$\times$	$\times$	$\times$	$*$
$-m$	$\times$				$*$
$\bot$	$*$	$*$	$*$	$*$	$*$

Looking at the table, we see that just six entries are missing, all between integer-derived shares. Three of those are always false, so we simply define:

$$\begin{aligned}\forall m : \mathtt{int}, n : \mathtt{int}.\, n \neq 0 \longrightarrow \mathsf{inl}(m) \succeq \mathsf{inl}(n) \leftrightarrow \\ m > 0 \wedge n > 0 \wedge (m - n < 0 \vee m = n) \vee \\ m > 0 \wedge n < 0 \vee \\ m < 0 \wedge n < 0 \wedge (m - n < 0 \vee m = n)\end{aligned}$$

which can be shortened to:

$$\begin{aligned}\forall m : \mathtt{int}, n : \mathtt{int}.\, n \neq 0 \longrightarrow \mathsf{inl}(m) \succeq \mathsf{inl}(n) \leftrightarrow \\ mn > 0 \wedge (m - n < 0 \vee m = n) \vee \\ m > 0 \wedge n < 0 \vee\end{aligned}$$

Note that I excluded the pre-existing entries in the center row to avoid conflicting definitions. This completes our table. All that remains is defining the inverse (which we denote by $\ominus$. We again start off with the unit axiom:

$$\forall a : S.\, a \ominus \mathsf{u} = a$$

Giving us the following checklist:

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$				
m	$\times$				
0	$\times$				
$-m$	$\times$				
$\bot$	$\times$				

We move on to the $\bot$ axioms:

$$\forall a : S.\, a \ominus \bot = \bot$$

$$\forall a : S.\, \bot \ominus a = \bot$$

Which entails inoffensive double definitions for $\bot \ominus \mathsf{u}$ and $\bot \ominus \bot$. We now have:

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$				$\times$
m	$\times$				$\times$
0	$\times$				$\times$
$-m$	$\times$				$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

We now handle the case where a share gets subtracted from itself:

$$\forall n : \mathtt{int}.\, \mathsf{inl}(n) \ominus \mathsf{inl}(n) = \mathsf{u}$$

Looking at our checklist, the following argument combinations are now defined:

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$				$\times$
m	$\times$	$\times$ if $m = n$			$\times$
0	$\times$		$\times$		$\times$
$-m$	$\times$			$\times$ if $m = n$	$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

Next, we target the inverses involving 0:

$$\forall n : \mathtt{int}.\, n \neq 0 \longrightarrow \mathsf{inl}(0) \ominus \mathsf{inl}(n) = \mathsf{inl}(-n)$$

Two more slots are now filled in:

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$				$\times$
m	$\times$	$\times$ if $m = n$			$\times$
0	$\times$	$\times$	$\times$	$\times$	$\times$
$-m$	$\times$			$\times$ if $m = n$	$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

Next let us fill in the remainder of the top row:

$$\forall a : S.\, a \neq u \longrightarrow u \ominus a = \bot$$

Again, we have a double definition in the case $a = \bot$, and again we see that this is matches the other definition, so all is good.

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$	$\times$	$\times$	$\times$	$\times$
m	$\times$	$\times$ if $m = n$			$\times$
0	$\times$	$\times$	$\times$	$\times$	$\times$
$-m$	$\times$			$\times$ if $m = n$	$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

Next let's take care of the remaining share subtractions that yield $\bot$:

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, m < 0 \wedge n > 0 \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(n) = \bot$$

$$\forall m : \mathtt{int}.\, m \neq 0 \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(0) = \bot$$

This is our progress:

$\ominus$	u	n	0	$-n$	$\bot$
u	$\times$	$\times$	$\times$	$\times$	$\times$
m	$\times$	$\times$ if $m = n$	$\times$		$\times$
0	$\times$	$\times$	$\times$	$\times$	$\times$
$-m$	$\times$	$\times$	$\times$	$\times$ if $m = n$	$\times$
$\bot$	$\times$	$\times$	$\times$	$\times$	$\times$

Now, we just need a final axiom:

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, m \neq n \longrightarrow$$
$$(m > 0 \wedge n \neq 0 \vee m < 0 \wedge n < 0) \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(n) = m - n$$

Which takes care of the remaining three cases, giving us the following complete list of axioms:

$$\forall n : \mathtt{int}.\, \mathsf{inl}(n) \neq \bot \wedge \mathsf{inl}(n) \neq \mathsf{u} \tag{C.1}$$

$$\forall n : \mathtt{int}, m : \mathtt{int}.\, \mathsf{inl}(m) = \mathsf{inl}(n) \longleftrightarrow m = n \tag{C.2}$$

$$P_A(m,n) \longleftrightarrow m < 0 \wedge n < 0 \tag{C.3}$$

$$P_B(m,n) \longleftrightarrow m > 0 \wedge n < 0 \wedge m + n \geq 0 \tag{C.4}$$

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, P_A(m,n) \vee P_B(m,n) \vee P_B(n,m) \longrightarrow \mathsf{inl}(m) \oplus \mathsf{inl}(n) = \mathsf{inl}(m+n) \tag{C.5}$$

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, \neg(P_A(m,n) \vee P_B(m,n) \vee P_B(n,m)) \longrightarrow \mathsf{inl}(m) \oplus \mathsf{inl}(n) = \bot \tag{C.6}$$

$$\forall a : S.\, a \oplus \bot = \bot \tag{C.7}$$

$$\forall a : S.\, \bot \oplus a = \bot. \tag{C.8}$$

$$\forall a : S.\, a \oplus \mathsf{u} = a \tag{C.9}$$

$$\forall a : S.\, \mathsf{u} \oplus a = a. \tag{C.10}$$

$$\forall a : S.\, a \neq \bot \longrightarrow a \succeq \mathsf{u} \tag{C.11}$$

$$\forall a : S.\, a \neq \bot \longrightarrow a \neq \mathsf{u} \longrightarrow \neg(\mathsf{u} \succeq a) \tag{C.12}$$

$$\forall n : \mathtt{int}.\, \mathsf{inl}(0) \succeq \mathsf{inl}(n) \tag{C.13}$$

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, n \neq 0 \longrightarrow \mathsf{inl}(m) \succeq \mathsf{inl}(n) \leftrightarrow mn > 0 \wedge (m - n < 0 \vee m = n) \vee m > 0 \wedge n < 0 \tag{C.14}$$

$$\forall a : S.\, a \ominus \mathsf{u} = a \tag{C.15}$$

$$\forall a : S.\, a \neq u \longrightarrow u \ominus a = \bot \tag{C.16}$$

$$\forall a : S.\, a \ominus \bot = \bot \tag{C.17}$$

$$\forall a : S.\, \bot \ominus a = \bot \tag{C.18}$$

$$\forall n : \mathtt{int}.\, \mathsf{inl}(n) \ominus \mathsf{inl}(n) = \mathsf{u} \tag{C.19}$$

$$\forall n : \mathtt{int}.\, n \neq 0 \longrightarrow \mathsf{inl}(0) \ominus \mathsf{inl}(n) = \mathsf{inl}(-n) \tag{C.20}$$

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, m < 0 \wedge n > 0 \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(n) = \bot \tag{C.21}$$

$$\forall m : \mathtt{int}.\, m \neq 0 \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(0) = \bot \tag{C.22}$$

$$\forall m : \mathtt{int}, n : \mathtt{int}.\, m \neq n \longrightarrow (m > 0 \wedge n \neq 0 \vee m < 0 \wedge n < 0) \longrightarrow \mathsf{inl}(m) \ominus \mathsf{inl}(n) = \mathsf{inl}(m - n) \tag{C.23}$$

Bibliography

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

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

[3] W.-N. Chin, C. David, and C. Gherghina, "A hip and sleek verification system," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 9–10. [Online]. Available: https://doi.org/10.1145/2048147.2048152

[4] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal, "Caper: Automatic verification for fine-grained concurrency," in *Proceedings of the 26th European Symposium on Programming (ESOP'17)*, Apr. 2017, pp. 420–447.

[5] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, "Gillian, part i: A multi-language platform for symbolic execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 927–942. [Online]. Available: https://doi.org/10.1145/3385412.3386014

[6] A. J. Summers and P. Müller, "Automating deductive verification for weak-memory programs (extended version)," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 6, pp. 709–728, Dec 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00559-y

[7] R. Dockins, A. Hobor, and A. W. Appel, "A fresh look at separation algebras and share accounting," in *Programming Languages and Systems*, Z. Hu, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 161–177.

[8] P. W. O'Hearn, "A primer on separation logic (and automatic program verification and analysis)." *Software Safety and Security*, vol. 33, pp. 286–318, 2012.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[10] C. A. R. Hoare, *Procedures and Parameters: An Axiomatic Approach*. USA: Prentice-Hall, Inc., 1989.

[11] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 1969, pp. 463–502, reprinted in McC90.

[12] J. Brotherston and C. Calcagno, "Classical bi: a logic for reasoning about dualising resources," *ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 328–339, 2009.

[13] S. S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 14–26. [Online]. Available: https://doi.org/10.1145/360204.375719

[14] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Computer Science Logic*, L. Fribourg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19.

[15] J. C. Reynolds, "Intuitionistic reasoning about shared mutable data structure," in *Millennial Perspectives in Computer Science*. Palgrave, 2000, pp. 303–321.

[16] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005, pp. 259–270.

[17] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis*, R. Cousot, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 55–72.

[18] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.

[19] P. W. O'Hearn, H. Yang, and J. C. Reynolds, "Separation and information hiding," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 3, pp. 1–50, 2009.

[20] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 23–42.

[21] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[22] J. Berdine, C. Calcagno, and P. W. O'hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 115–137.

[23] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *CONCUR 2004 - Concurrency Theory*, P. Gardner and N. Yoshida, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 49–67.

[24] B. Jacobs and F. Piessens, "Expressive modular fine-grained concurrency specification," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011, pp. 271–282.

[25] M. Parkinson, R. Bornat, and P. O'Hearn, "Modular verification of a non-blocking stack," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007, pp. 297–302.

[26] A. Charguéraud and F. Pottier, "Temporary read-only permissions for separation logic," in *European Symposium on Programming*. Springer, 2017, pp. 260–286.

[27] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local action and abstract separation logic," in *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE, 2007, pp. 366–378.

[28] B. Jacobs and F. Piessens, "The verifast program verifier," Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Tech. Rep., 2008.

[29] M. J. Parkinson, "Local reasoning for Java," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-654, Nov. 2005. [Online].

Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-654.pdf

[30] A. Hobor, *Oracle semantics*. Princeton University, 2008.

[31] X.-B. Le, T.-T. Nguyen, W.-N. Chin, and A. Hobor, "A certified decision procedure for tree shares," in *Formal Methods and Software Engineering*, Z. Duan and L. Ong, Eds. Cham: Springer International Publishing, 2017, pp. 226–242.

[32] Programming Methodology Group, ETH Zürich, "The Viper tutorial," accessed: September 5, 2021. [Online]. Available: viper.ethz.ch/tutorial

[33] M. H. Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution," Ph.D. dissertation, ETH Zurich, 2016.

[34] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[35] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[36] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360573

[37] 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.

[38] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification (CAV)*, A. Silva and K. R. M. Leino, Eds. Springer International Publishing, 2021, pp. 367–379.

[39] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, May 2012. [Online]. Available: https://doi.org/10.1145/2160910.2160911

[40] M. Parkinson and A. Summers, "The relationship between separation logic and implicit dynamic frames," *Logical Methods in Computer Science*, vol. 8, no. 3, Jul 2012. [Online]. Available: http://dx.doi.org/10.2168/LMCS-8(3:1)2012

[41] K. R. M. Leino, "This is boogie 2," June 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[42] D. Liew, "The boogie intermediate verification language," accessed: September 5, 2021. [Online]. Available: https://boogie-docs.readthedocs.io/en/latest/

[43] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i," *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, Dec 1931. [Online]. Available: https://doi.org/10.1007/BF01700692

[44] X.-B. Le and A. Hobor, "Logical reasoning for disjoint permissions," in *Programming Languages and Systems*, A. Ahmed, Ed. Cham: Springer International Publishing, 2018, pp. 385–414.

[45] L. Halbeisen and R. Krapf, *Gödels Theorems and Zermelo's Axioms: a firm foundation of mathematics*. Springer Nature, 2021.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Extending the Viper Verification Language with User-Defined Permission Models

Authored by (in block letters):

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

Name(s):	**First name(s):**
Rosshardt	Matthias René

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Zürich, 16.9.2021	

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