ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Improving User-Defined Permission Models in Viper

Practical Work

Anqi Li

March 15, 2023

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

Abstract

Viper is a powerful toolchain and infrastructure for program verification. It automates and provides native support for permission-based reasoning by using separation logic. Currently, the fractional permission model is the only permission model available in Viper. A permission model is a partial commutative monoid useful for reasoning about shared data structures. Despite the merits of fractional permissions such as infinite splittability (i.e. any non-empty share can be split into two non-empty shares), alternative permission models can be useful depending on the use cases. This project aims at assessing and improving a prototype that introduces user-defined permission models to Viper. The goal is achieved by first conducting several case studies where multiple permission models are encoded using the prototype, and then adding to the prototype a new feature that allows users to specify a permission model for a field so that multiple permission models can exist in the same program.

Contents

Chapter 1

Introduction

1.1 Motivation

Program verification is the process of formally proving that a program satisfies its specification. It is useful for proving the absence of errors in various software systems. Nowadays, many software tools have been developed to automate program verification, such as Viper [10]. Dissimilar to other verification tools like Boogie [9] and Why3 [8], Viper uses separation logic [11] to natively support permission-based reasoning via permission models.

A permission model is, typically, a partial commutative monoid [4, 7]. Each permission model has its own set of permission shares that are used to express heap location ownership. Permission shares can be split and recombined. By using a permission model, one can reason about method calls with the frame rule, and also efficiently verify concurrent programs that manipulate shared data structures.

For the time being, Viper supports only one permission model – fractional permissions [2, 3]. In the fractional permission model, permission amounts are rational values in the range of [0, 1]. A permission amount of 1 allows one to write to a heap location, while any non-zero permission amount allows one to read from a heap location. The fractional permission model can be used to efficiently reason about programs involving concurrent reads, divide-and-conquer algorithms and so on, but it is not suitable for every situation. For instance, consider the code snippet in Listing 1.1 that attempts to formulate an inductive predicate for a binary tree data structure. When $p > \frac{1}{2}$, the predicate always describes a binary tree, while when $p \leq \frac{1}{2}$, it can describe a directed acyclic graph (DAG) rather than a binary tree. To understand why this is the case, consider a DAG with 4 distinct nodes T, L, R, B as shown in Figure 1.1.

```
field leftChild: Ref
field rightChild: Ref
field elem: Int
predicate Tree (this: Ref, p: Perm) {
    this != null && p > 0/1 ==>
        acc(this.elem, p) & &
        acc(this.leftChild, p) & &
        acc(this. rightChild, p) &&
        Tree(this.leftChild, P) &&
        Tree(this.rightChild, p)
}
```

Listing 1.1: Tree predicate written in Viper

Assume that `Tree(T, p)` holds. The definition of the predicate implies that `Tree(L, p)`, `Tree(R, p)` and `Tree(B, p+p)` must also hold. When $p > \frac{1}{2}$, $p + p > 1$ is an undefined permission amount, which invalidates such DAG. In contrast, when $p \leq \frac{1}{2}$, $p + p \leq 1$ is a valid permission amount, so this DAG can be "mistakenly" considered as a tree. As a result, if a method has some positive permission amount to a binary tree and would like to read it, it needs to know that the left and right subtrees of the root node are disjoint to be sure that it is indeed reading a binary tree instead of a DAG. This example demonstrates the limitations of fractional permissions in some situations. Consequently, it is desirable to allow users to define alternative permission models and switch between them depending on the use cases.

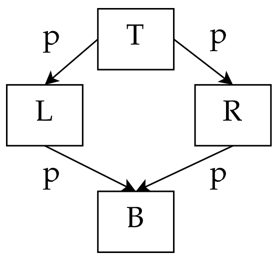
$$\begin{tikzcd} & T \arrow[dl, "p"'] \arrow[dr, "p"] & \\ L \arrow[dr, "p"'] & & R \arrow[dl, "p"] \\ & B & \end{tikzcd}$$

Figure 1.1: A DAG described by the tree predicate in Listing 1.1

1.2 Related Work

User-defined permission models were introduced to Viper for the first time as an experimental feature in a master's thesis project. The thesis [12] documents the theoretical foundations, methodology and implementation details for this feature. As an extension of the thesis, this project builds upon the existing prototype that implements such feature.

1.3 Report Outline

The content of the report is organized as follows. In Chapter 2, multiple case studies are presented to explore the limits of the existing prototype.

Next, in Chapter 3, we describe how a new feature, which enables different permission models in the same Viper program depending on the field, has been implemented based on the prototype. Finally, we draw a conclusion and discuss future work in Chapter 4.

Chapter 2

Case Studies

In this chapter, we present the case studies done with multiple permission models, including counting permissions [2], tree shares [7] and duplicable permissions. The main purpose of the case studies is to test the limits of the existing prototype that enables users to define alternative permission models in Viper programs. To achieve this goal, for each permission model, we study its semantics, encode it using the prototype and test the encoded model. The details of each case study are given in the following sections.

2.1 Background

2.1.1 Architecture of Viper

The prototype and all implementation done in this project are integrated into Carbon – one of the two backends of Viper. It is essentially a compiler that takes a program written in the Viper intermediate language as input and produces a Boogie program as output. Figure 2.1 gives an overview of the architecture of the Viper verification toolchain.

2.1.2 Separation Algebra

A separation algebra (SA) [4] is defined as a tuple $\langle A, \oplus, u \rangle$, where A is a set of elements, $\oplus$ is a partial operation of joining two elements, and u is the unit element in A. Every SA satisfies the following axioms [7]:

- Commutative: $a \oplus b = b \oplus a$
- Associative: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
- Identity: $a \oplus u = a$
- Cancellative: $a_1 \oplus b = a_2 \oplus b \implies a_1 = a_2$

A permission model is a SA used for permission accounting.

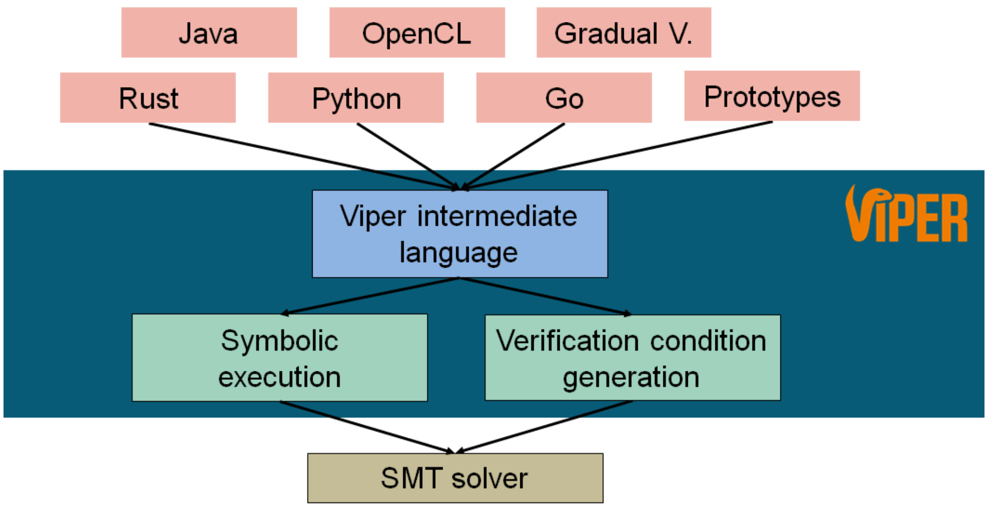

Figure 2.1: Viper architecture overview

2.2 Counting Permissions

The counting permission model allows one to count the number of permissions given out. An intuitive explanation of this permission model can be found in Section 2.1.7 of Roshardt [12]'s thesis, while a more formal definition and extensive details are available in the paper authored by Bornat et al. [2].

2.2.1 Permission Shares

In the model of counting permissions, permission shares are represented using integers. More specifically, 0 means a full permission, -1 means a read permission, and $+k$ means a source permission from which k read permissions have been taken. The unit share is represented by a symbol $u \notin \mathbb{Z}$.

As an example, consider three threads A, B and C. Initially, thread A has a full permission to some resource r, and the other threads have no permissions to access r. In other words, thread A has permission share 0, and threads B and C both have unit permissions represented by u. Thread A later gives out read permissions to threads B and C respectively. As a consequence, thread A now has permission share $+2$, while each of threads B and C has permission share -1.

2.2.2 Joining Permission Shares

To join two shares a and b, the following rules apply:

$$a \oplus b = \begin{cases} \perp & a \geq 0 \land b \geq 0 \\ \perp & (a \geq 0 \lor b \geq 0) \land (a + b < 0) \\ a + b & \text{Otherwise} \end{cases} \tag{2.1}$$

2.2.3 Result and Application

By using the prototype, we encoded counting permissions in Viper as shown in Listing 2.1. The function `share(x:Int)` returns the permission share represented by integer `x`, so the full share, represented by the return value of `full()`, is `share(0)`. Based on Rules 2.1, we can easily formulate the axiom for `plus(x:Perm,y:Perm)` by computing the arithmetic sum of the two input shares, and we may also derive the joinable axiom `axJoinable` by negating the disjunction of all conditions under which two shares $a \oplus b = \perp$. Lastly, to understand the encoding of `minus(x:Perm,y:Perm)` and `geq(x:Perm,y:Perm)`, one should refer to Section 5.3.3 of Roshardt [12]'s thesis.

```
domain _Perm {
    function unit(): Perm
    function full(): 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

    function share(x: Int): Perm

    axiom axFullShare { full() == share(0) }

    axiom axJoinable {
        forall a: Int, b: Int ::
        joinable(share(a), share(b)) <==>
            ((a < 0 && b < 0) ||
             (a < 0 && b > 0 && a + b >= 0) ||
             (a > 0 && b < 0 && a + b >= 0))
    }

    axiom axJoinableUnit { forall t: Perm ::
        joinable(unit(), t) && joinable(t, unit())
    }

    axiom axPlus { forall a: Int, b: Int ::
        joinable(share(a), share(b)) ==>
        plus(share(a), share(b)) == share(a+b)
    }

    axiom axUnitPlus { forall t:Perm ::
        plus(unit(), t) == t
        && plus(t, unit()) == t
    }

    axiom axMinus { forall a: Int, b: Int ::
        geq(share(a), share(b)) && a != b ==>
        minus(share(a), share(b)) == share(a-b)
    }

    axiom axUnitMinus { forall p: Perm ::
        minus(p, p) == unit()
    }

    axiom axGeq {
        forall a: Int, b: Int ::
        geq(share(a), share(b)) <==>
        ((a < 0 && b < 0 && a < b) ||
         (a >= 0 && (b < 0 || (b >= 0 && a < b))) ||
         (a == b))
    }

    axiom axUnitGeq { forall p: Perm ::
        geq(p, unit())
    }
}
```

Listing 2.1: Counting permissions encoded using the prototype

The encoded counting permission model has been used to verify a program with a readers-and-writers pattern where multiple threads concurrently read from and write to some shared resource in the critical section. It is useful in this scenario due to its ability to count the permissions given out. Listing 2.2 shows how we modeled this scenario in Viper by referring to Bornat et al. [2]'s work.

In the following code, the variable `count` records the number of readers that are currently accessing the shared resource. To enter the critical section, a reader first needs to gain a read permission. If it is the first reader that attempts to read the resource, it sets the variable `m` to `0` to prevent any writers from writing to the shared resource. Before the reader leaves the critical section, if it is the last reader to leave, i.e. if `count == 0`, it sets `m` to 1 to unblock the writers. When `m == 1`, one writer will gain a full permission to the shared resource and therefore become able to write.

```
field f: Int

define I_write(m)
    m != 0 ==> acc(res.f, full())

define I_read(num)
    num != 0 ==> acc(res.f, share(num))

define P() {
    var m: Int
    m := havoc_int_binary()
    assume m != 0
    inhale I_write(m)
    m := 0
    exhale I_write(m)
}

define V() {
    var m: Int
    m := havoc_int_binary()
    assume true
    inhale I_write(m)
    m := 1
    exhale I_write(m)
}

method havoc_int() returns (res: Int)
ensures res >= 0

method havoc_int_binary() returns (res: Int)
ensures res == 0 || res == 1

method reader(res: Ref) {
    var count: Int
    count := havoc_int()
    assume true
    inhale I_read(count)
    if (count == 0) {
        P()
    }
    count := count + 1
    exhale I_read(count)

    var content: Int
    content := res.f     // Read

    count := havoc_int()
    assume count > 0
    inhale I_read(count)
    count := count - 1
    if (count == 0) {
        V()
    }
    exhale I_read(count)
}

method writer(res: Ref) {
    P()
    res.f := 2   // Write
    V()
}
```

Listing 2.2: Viper program modeling the readers-and-writers problem

2.3 Tree Shares

The tree share permission model uses binary trees to represent permission shares. We are interested in this model because of its desirable properties: infinite splittability and disjointness. The former property means that any non-empty shares can be split into two non-empty shares, whilst the latter one means that no non-empty share can join with itself. A more extensive description of this model can be found in Section 2.1.7 of Roshardt [12]'s thesis.

2.3.1 Permission Shares

Permission shares are represented using binary trees defined as $\tau = \circ | \bullet | \langle \tau, \tau \rangle$ where

- $\circ$ is a leaf with boolean value *False*, representing an empty permission. It is also the unit share.
- $\bullet$ is a leaf with boolean value *True*, representing a full permission.

- $\langle \tau, \tau \rangle$ is a binary tree with unlabeled internal nodes and boolean-valued leaves, which can represent any permission amount.

Note that, under such definition, trees of different shapes can represent the same permission amount. For instance, both ● and $\langle$●, ●$\rangle$ represent a full permission. We say that such trees are equivalent to one another.

2.3.2 Joining Permission Shares

To join two shares represented by trees of the same shape, one should apply the following rules leaf-wise:

$$\begin{cases} \circ \oplus \circ = \circ \\ \circ \oplus \bullet = \bullet \\ \bullet \oplus \circ = \bullet \\ \langle \tau_1, \tau_2 \rangle \oplus \langle \tau_3, \tau_4 \rangle = \langle \tau_1 \oplus \tau_3, \tau_2 \oplus \tau_4 \rangle \end{cases} \tag{2.2}$$

In the case where two shares are represented by trees τ_1, τ_2 of different shapes, one needs to perform the following steps to convert them to equivalent trees of the same shape before joining:

1. Let h_1, h_2 be the height of τ_1, τ_2 respectively. Compute $h = max(h_1, h_2)$.
2. For each leaf of τ_1 (resp. τ_2), keep replacing it with two children of the same value as itself until τ_1 (resp. τ_2) becomes a full binary tree [1] of height h.

As an example, Figure 2.2 shows how a binary tree of height 1 is unfolded to an equivalent full binary tree of height 2.

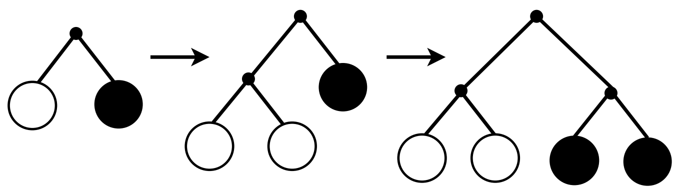

Figure 2.2: Tree unfolding example

2.3.3 Result

In the initial attempt to encode tree shares, permission shares were encoded directly as trees, and equality of two shares was expressed using a function `eq(p1:Perm, p2:Perm)` whose semantics were specified in an axiom. This

[1] A full binary tree is a binary tree in which every node except for the leaves has exactly 2 children.

is because two different trees can represent the same permission amount, which cannot be directly expressed with the equality operator ==. Upon detecting the declaration of `eq(p1:Perm, p2:Perm)` in the permission model, the prototype syntactically replaced all occurrences of the equality operator with calls to this function when translating the Viper program.

However, the tree shares encoded in this way did not have the expected behavior because the equality relation did not propagate to functions. In other words, we do not automatically have `eq(a, b)` $\not\Longrightarrow$ `eq(f(a), f(b))` for some function `f`. This can happen for many reasons such as missing axioms and bad triggers. Without such implication, one has to perform more complex steps to arrive at the same conclusion as when such implication is available. Therefore, in the final encoding of tree shares, we decided to use a domain to define the binary tree data structure and another one to define the permission shares represented by the trees. Equality of two tree shares is expressed with == and the axiom in Listing 2.3. The complete Viper encoding of tree shares is available in Appendix A.1.

```
function eqTree(Tree t1, Tree t2): Bool
function share(Tree t): _Perm_TreeShare
axiom axShareEq { forall t1: Tree, t2: Tree ::
    eqTree(t1, t2) ==> share(t1) == share(t2)
}
```

Listing 2.3: Equality axiom for tree shares

2.4 Duplicable Permissions

Duplicable permissions model the behavior of `wildcard` in Viper, which is an unspecified positive permission amount useful for implementing duplicable read-only resources [1].

2.4.1 Permission Shares

Duplicable permissions have 3 kinds of permission shares:

- ○: empty permission, also the unit share
- ●: full permission
- ◐: some positive permission amount that allows read-only access

2.4.2 Joining Permission Shares

To join two permission shares, the following rules apply:

$$\begin{cases} \Circle \oplus \Circle = \Circle \\ \Circle \oplus \CIRCLE = \CIRCLE \oplus \Circle = \CIRCLE \\ \Circle \oplus \LEFTcircle = \LEFTcircle \oplus \Circle = \LEFTcircle \\ \LEFTcircle \oplus \LEFTcircle = \LEFTcircle \end{cases} \tag{2.3}$$

Among these rules, the last one is the most important one. It allows any read-only permissions to be duplicated.

2.4.3 Result and Application

The encoding of duplicable permissions is shown in Listing 2.4.

```
domain _Perm {
    function unit(): Perm
    function full(): 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

    function share(): Perm

    axiom axUnitAndFull {
        full() != share() && unit() != share()
    }

    axiom axPlus { forall p1: Perm, p2: Perm ::
        joinable(p1, p2) &&
        p1 != unit() && p2 != unit() ==>
            plus(p1, p2) == share()
    }

    axiom axPlusUnit { forall p: Perm ::
        plus(unit(), p) == plus(p, unit()) &&
        plus(unit(), p) == p
    }

    axiom axMinus { forall p1: Perm, p2: Perm ::
        geq(p1, p2) && p1 != unit() &&
        p2 != unit() && p1 != full() &&
        p2 != full() ==>
            minus(p1, p2) == share()
    }

    axiom axMinusFullFull {
       minus(full(), full()) == unit()
    }

    axiom axMinusUnit { forall p: Perm ::
        minus(p, unit()) == p
    }

    axiom axMinusFull { forall p: Perm ::
        p != unit() && p != full()
        ==> minus(full(), p) == share()
    }

    axiom axJoinable { forall p1: Perm, p2: Perm ::
        joinable(p1, p2) <==>
            (p1 != full() && p2 != full()) ||
            (p1 == full() && p2 == unit()) ||
            (p1 == unit() && p2 == full())
    }

    axiom axGeq { forall p1: Perm, p2: Perm  ::
        geq(p1, p2) <==>
            (p1 == full() && p2 != full()) ||
            (p1 != unit() && p2 == unit()) ||
            (p1 != unit() && p2 != unit()
            && p1 != full() && p2 != full())
    }

    axiom axGeUnit { forall p: Perm ::
        geq(p, unit())
    }
}
```

Listing 2.4: Duplicable permissions encoded using the prototype

The encoded duplicable permission model was later used in a quick runtime comparison experiment with `wildcard`. In the experiment, we created a Viper program with references `ref0, ref1, ref2, ref3, ref4, ref5` and a block of code, shown in Listing 2.5, where a `wildcard` permission amount was inhaled and exhaled once for each of these references.

```
inhale acc(ref0.f, wildcard)
        ...
inhale acc(ref5.f, wildcard)
exhale acc(ref0.f, wildcard)
        ...
exhale acc(ref5.f, wildcard)
```

Listing 2.5: Test code for runtime comparison between wildcard and duplicable permissions

Such code block was then repeated for a different number of times to form a set of test programs. We ran the programs with `wildcard` and recorded the runtime of Viper, and then replaced `wildcard` with `share()` from duplicable permissions, which is `wildcard` in practice, in all programs and measured the runtime again. Table 2.1 shows that duplicable permissions always yield a shorter runtime than `wildcard`. The reduction in runtime is more and more significant as the number of code blocks increases.

Number of code blocks	Runtime of `wildcard`	Runtime of duplicable permissions
1	2.34s	2.55s
2	3.35s	2.80s
3	152s	3.04s
4	366s	9.91s

Table 2.1: Runtime comparison between wildcard and duplicable permissions

2.5 Summary

The case studies show that the prototype could encode various permission models and that different permission models have different properties. The properties that users might want in a permission model are as follows:

- Infinite splittability: each non-empty share can be split into two non-empty shares.
- Disjointness: no non-empty share joins with itself.
- Countability: the permission model can count the number of permissions given out.
- Recombinability: shares split from a full share can recombine to form a full share.
- Complexity: defined with respect to the workload for SMT solvers to work with the permission model.

Table 2.2 summarizes the properties of the aforementioned permission models.

Properties	Fractional permissions	Counting permissions	Tree shares	Duplicable permissions
Infinite splittability	✓		✓	✓
Disjointness			✓	
Countability		✓		
Recombinability	✓	✓	✓	
Complexity	+++	++	++++	+

Table 2.2: Properties of permission models

Consequently, users might prefer one permission model to another based on the use cases, or they might need multiple permission models in one Viper program, which leads to the development of a new feature, namely permission-model-per-field, that we are going to discuss in the next chapter.

Chapter 3

Permission-Model-per-Field Feature

This chapter describes a new feature, permission-model-per-field, that allows users to specify a permission model for a field in a Viper program. This feature enables multiple permission models to coexist in the same program. In the following sections, we first show the syntax of this feature, and then illustrate how it has been implemented.

3.1 Syntax

It is obvious that this new feature requires that multiple permission models could be defined in one Viper program. This leads to some changes in the syntax of permission model declaration. To define a permission model, users should declare a Viper `domain` with identifier `_Perm_<id>` and a list of mandatory function declarations as follows:

- `function unit_<id>():_Perm_<id>`
  This function returns a unit share.
- `function full_<id>():_Perm_<id>`
  This function returns a full share.
- `function plus_<id>(x:_Perm_<id>, y:_Perm_<id>):_Perm_<id>`
  This function returns the result of joining two shares `x` and `y`.
- `function minus_<id>(x:_Perm_<id>, y:_Perm_<id>):_Perm_<id>`
  This function returns the result of inversely joining two shares `x` and `y`. It is used to compute `z` such that $\texttt{x} = \texttt{y} \oplus \texttt{z}$.
- `function joinable_<id>(x:_Perm_<id>, y:_Perm_<id>):Bool`
  This function checks if two shares `x` and `y` can join with each other.
- `function geq_<id>(x:_Perm_<id>, y:_Perm_<id>):Bool`
  This function checks if share `x` is greater than or equal to share `y`.

An error message will be thrown if any of the functions above is missing or does not have the expected signature. The semantics of these functions depend on the encoded permission model and should be specified using axioms.

To specify a permission model for a field, the following syntax should be used for field declarations:

```
field ([_Perm_<id>])? <field_id>: <field_type>
```

Note that when the permission model of a field is left unspecified, the default permission model, i.e. fractional permissions, is used.

3.2 Implementation

This section gives implementation details of the permission-model-per-field feature. We first show the changes made in the parser and typechecker of Silver, and then list the updates in the permission module of Carbon.

3.2.1 Parser

The grammar rule of field declarations has been updated to allow users to specify `_Perm_<id>` for the declared field. This leads to the addition of one argument, i.e. `permId`, to the constructor of `PField` with a default value `Fractions`.

3.2.2 Typechecker

The typechecker checks that `_Perm_<id>` in every field declaration is indeed the identifier of a `domain` declared in the program. It also ensures that strings starting with `unit`, `full`, `plus`, `minus`, `geq` and `joinable` are only used as the identifier of functions declared in permission models, i.e. any `domain` whose identifier has the prefix `_Perm_`. Moreover, the typechecking rules are updated such that `domain` types are allowed wherever only the built-in `Perm` type was expected in the prototype. This has caused changes in the typechecking of access predicates, `perm()` function calls, and wherever `wildcard` can appear. Furthermore, the typechecker makes sure that the permission model used to access a field matches with the permission model in the declaration of the field. For instance, if `x` has type `Ref` and field `f` is declared to use counting permissions, the typechecker rejects the statement `inhale acc(x.f, 1/2)`.

3.2.3 Permission Module

The permission module specializes in the handling of permission models. It checks that the declaration of permission models has the expected syntax, and then creates and maintains a `Map` object that allows us to access everything in a permission model by providing its identifier.

To enable the use of multiple permission models, the module makes use of polymorphism. Recall that every permission model is required to have its own `unit`, `full`, `plus`, `minus`, `geq` and `joinable` functions. Instead of emitting Boogie functions corresponding to these functions for every different permission model encountered, the module emits the following polymorphic Boogie functions only once:

```
function  full<P>(): P;
function  unit<P>(): P;
function  joinable<P>(x: P, y: P): bool;
function  plus<P>(x: P, y: P): P;
function  geq<P>(x: P, y: P): bool;
function  minus<P>(x: P, y: P): P;
```

Listing 3.1: Polymorphic Boogie functions emitted by the permission module

The semantics of these functions from each different permission model are defined via axioms with the type parameter `P` specified. For example, Listing 3.2 shows an axiom that describes the semantics of the `plus` function for fractional permissions, while the axiom in Listing 3.3 formulates the semantics of the same function for duplicable permissions. The type parameter `P` is instantiated as `FracPerm` in the former axiom and `_Perm_wildcardDomainType` in the latter one.

```
axiom (forall x: FracPerm, y: FracPerm ::
  joinable(x, y) ==> plus(x, y) == x + y
);
```

Listing 3.2: Axiom of `plus` function from fractional permissions

```
axiom (forall p1: _Perm_wildcardDomainType, p2:
    _Perm_wildcardDomainType ::
  joinable(p1, p2) && p1 != unit() && p2 != unit()
  ==> (plus(p1, p2): _Perm_wildcardDomainType) == (
      share(): _Perm_wildcardDomainType)
);
```

Listing 3.3: Axiom of `plus` function from duplicable permissions

By using polymorphism, code duplication is avoided since functions shared by permission models are emitted only once in the Boogie program. Moreover, existing code can be reused without adaptation. For example, the code emitting the axiom that initializes the zero masks to unit shares, shown in Listing 3.4, does not need to be adapted, because there is no need to specify the permission model associated with each unit share thanks to polymorphism. More importantly, the modularity of Carbon code base is maintained: other modules in Carbon do not need to know the existence of multiple permission models in the permission module.

```
axiom (forall <A, B, P> o_2: Ref, f_5: (Field A B P)
    :: { ZeroMask[o_2, f_5] }
  !IsPredicateField(f_5) ==>
  ZeroMask[o_2, f_5] == (unit(): P)
);
```

Listing 3.4: Zero mask initialization

Chapter 4

Conclusion and Future Work

4.1 Conclusion

In this project, we have carried out case studies involving multiple permission models to assess the functionality of the prototype that allows users to define alternative permission models in Viper. Several bug fixes have also been made to the prototype at the same time. Additionally, we have added a new feature to the prototype to enable users to specify a permission model for a field so that multiple permission models can exist in one Viper program.

4.2 Future Work

There are several aspects that can be improved in the future to extend our work. Firstly, more extensive testing of the permission-model-per-field feature is needed. One possible way to do so is to encode real-world scenarios that use multiple permission models. Secondly, the automation of magic wands [6] needs to be adapted to the changes made in the permission module of Carbon. Last but not least, due to the introduction of the permission-model-per-field feature, there can exist multiple permission models in one predicate. Thereby, the multiplication of predicates [5] needs to be generalized.

Appendix A

Appendix

A.1 Viper encoding of tree shares

```

domain Tree {
    function getRight(this: Tree): Tree
    function getLeft(this: Tree): Tree
    function node(t1: Tree, t2: Tree): Tree

    function eqTree(t1: Tree, t2: Tree): Bool
    function _eqTree(t1: Tree, t2: Tree): Bool
    function geqTree(t1: Tree, t2: Tree): Bool
    function _geqTree(t1: Tree, t2: Tree): Bool
    function joinableTree(t1: Tree, t2: Tree): Bool
    function _joinableTree(t1: Tree, t2: Tree):
       Bool
    function plusTree(t1: Tree, t2: Tree): Tree
    function _plusTree(t1: Tree, t2: Tree): Tree
    function minusTree(t1: Tree, t2: Tree): Tree
    function _minusTree(t1: Tree, t2: Tree): Tree
    function isLeaf(t1: Tree): Bool


    function toFullTree(t: Tree, h: Int): Tree
    function sameShape(t1: Tree, t2: Tree): Bool
    function getHeight(t: Tree): Int
    function max(a: Int, b: Int): Int

    function dummy(t: Tree): Bool

    // Leaves
    unique function zero(): Tree     // White circle
    unique function one(): Tree      // Black circle

    axiom axConst {
        zero() != one()
    }

    axiom axMax { forall a: Int, b: Int ::
    {max(a, b)}
        (a >= b ==> max(a, b) == a) &&
        (a < b ==> max(a, b) == b)
    }

    axiom axIsLeaf { forall t: Tree :: {isLeaf(t)}
        isLeaf(t) <==> (t == zero() || t == one())
    }

    axiom axLeaf { forall t: Tree ::
        (t == zero() || t == one()) ==>
        (getRight(t) == t && getLeft(t) == t)
    }

    axiom axNonLeaf { forall t1: Tree, t2: Tree ::
        getLeft(node(t1, t2)) == t1 &&
        getRight(node(t1, t2)) == t2 &&
        !isLeaf(node(t1, t2))
    }

    axiom axSameShape { forall t1: Tree, t2: Tree :: {dummy(t1), dummy(t2)}
        sameShape(t1, t2) <==>
            (isLeaf(t1) && isLeaf(t2)) ||
            (!isLeaf(t1) && !isLeaf(t2) &&
             sameShape(getLeft(t1), getLeft(t2)) &&
             sameShape(getRight(t1), getRight(t2))
            )
    }

    axiom axGetHeight { forall t: Tree ::
        {dummy(t), getHeight(t)}
        (isLeaf(t) ==> getHeight(t) == 0) &&
        (!isLeaf(t) ==> getHeight(t) == 1 + max(getHeight(getLeft(t)), getHeight(getRight(t))))
    }

    axiom axToFullTree { forall t: Tree, h: Int :: {dummy(t), toFullTree(t, h)}
        (h == 0 ==> toFullTree(t, h) == t) &&
        (h != 0 ==> toFullTree(t, h) == node(toFullTree(getLeft(t), h-1), toFullTree(getRight(t), h-1)))
    }

    axiom axEqTree { forall t1: Tree, t2: Tree :: {eqTree(t1, t2)}
        (sameShape(t1, t2) ==> eqTree(t1, t2) == _eqTree(t1, t2)) &&
        (!sameShape(t1, t2) ==> eqTree(t1, t2) == _eqTree(toFullTree(t1, max(getHeight(t1), getHeight(t2))), toFullTree(t2, max(getHeight(t1), getHeight(t2)))))
    }

    // Assume that t1 & t2 have the same shape
    axiom axEqTreeHelper { forall t1: Tree, t2: Tree :: {dummy(t1), dummy(t2)}
        _eqTree(t1, t2) <==>
            ((isLeaf(t1) && isLeaf(t2) && t1 == t2) ||
            (!isLeaf(t1) && !isLeaf(t2) && _eqTree(getRight(t1), getRight(t2)) &&
            _eqTree(getLeft(t1), getLeft(t2))))
    }

    axiom axJoinableTree { forall t1: Tree, t2: Tree :: {joinableTree(t1, t2)}
        (sameShape(t1, t2) ==> joinableTree(t1, t2) == _joinableTree(t1, t2)) &&
        (!sameShape(t1, t2) ==> joinableTree(t1, t2) == _joinableTree(toFullTree(t1, max(getHeight(t1), getHeight(t2))), toFullTree(t2, max(getHeight(t1), getHeight(t2)))))
    }

    // Assume that t1 & t2 have the same shape
    axiom axJoinableTreeHelper { forall t1: Tree, t2: Tree :: {dummy(t1), dummy(t2)}
        _joinableTree(t1, t2) <==>
            (
                (isLeaf(t1) && isLeaf(t2) &&
                    ((t1 == zero() && t2 == one()) ||
                    (t1 == one() && t2 == zero()) ||
                    (t1 == zero() && t2 == zero()))
                ) ||
                (!isLeaf(t1) && !isLeaf(t2) &&
                  _joinableTree(getRight(t1), getRight(t2)) &&
                  _joinableTree(getLeft(t1), getLeft(t2))
                )
            )
    }

    axiom axGeqTree { forall t1: Tree, t2: Tree :: {geqTree(t1, t2)}
        (sameShape(t1, t2) ==> geqTree(t1, t2) == _geqTree(t1, t2)) &&
        (!sameShape(t1, t2) ==> geqTree(t1, t2) == _geqTree(toFullTree(t1, max(getHeight(t1), getHeight(t2))), toFullTree(t2, max(getHeight(t1), getHeight(t2)))))
    }

    // Assume that t1 & t2 have the same shape
    axiom axGeqTreeHelper { forall t1: Tree, t2: Tree :: {dummy(t1), dummy(t2)}
        _geqTree(t1, t2) <==>
            ((isLeaf(t1) && isLeaf(t2) &&
                (t1 == t2) || (t1 == one() && t2 == zero())) ||
            (!isLeaf(t1) && !isLeaf(t2) &&
                _geqTree(getLeft(t1), getLeft(t2)) && _geqTree(getRight(t1), getRight(t2))))
    }

    axiom axPlusTree { forall t1: Tree, t2: Tree ::
        {plusTree(t1, t2)}
        (sameShape(t1, t2) ==> plusTree(t1, t2) ==
            _plusTree(t1, t2)) &&
        (!sameShape(t1, t2) ==> plusTree(t1, t2) ==
            _plusTree(toFullTree(t1, max(getHeight(
            t1), getHeight(t2))), toFullTree(t2, max
            (getHeight(t1), getHeight(t2)))))
    }

    // Assume that t1 & t2 have the same shape
    axiom axPlusTreeHelper { forall t1: Tree, t2:
       Tree :: {dummy(t1), dummy(t2)}
        _joinableTree(t1, t2) ==>
            (
                (isLeaf(t1) && isLeaf(t2) &&
                    ((((t1 == zero() && t2 == one()
                        ) || (t1 == one() && t2 ==
                        zero()))
                            ==> _plusTree(t1, t2)
                               == one()) &&
                     ((t1 == zero() && t2 == zero()
                        ) ==> _plusTree(t1, t2) ==
                        zero()))
                ) ||
                (!isLeaf(t1) && !isLeaf(t2) &&
                    _plusTree(t1, t2) == node(
                        _plusTree(getLeft(t1),
                        getLeft(t2)), _plusTree(
                        getRight(t1), getRight(t2)))
                        )
            )
    }

    axiom axMinusTree { forall t1: Tree, t2: Tree
       :: {minusTree(t1, t2)}
        (sameShape(t1, t2) ==> minusTree(t1, t2) ==
            _minusTree(t1, t2)) &&
        (!sameShape(t1, t2) ==> minusTree(t1, t2)
           == _minusTree(toFullTree(t1, max(
           getHeight(t1), getHeight(t2))),
           toFullTree(t2, max(getHeight(t1),
           getHeight(t2)))))
    }

    // Assume that t1 & t2 have the same shape
    axiom axMinusTreeHelper { forall t1: Tree, t2:
       Tree :: {dummy(t1), dummy(t2)}
        _geqTree(t1, t2) ==>
        (
            (isLeaf(t1) && isLeaf(t2) &&
                ((t1 == t2 ==> _minusTree(t1, t2)
                   == zero()) &&
                 ((t1 == one() && t2 == zero()) ==>
                     _minusTree(t1, t2) == one()))
            )  ||
            (!isLeaf(t1) && !isLeaf(t2) &&
                _minusTree(t1, t2) == node(
                   _minusTree(getLeft(t1), getLeft(
                   t2)), _minusTree(getRight(t1),
                   getRight(t2))))
        )
    }
}

domain _Perm {
    function unit(): Perm
    function full(): 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
    function share(tree: Tree): Perm

    axiom axUnit { unit() == share(zero()) }
    axiom axFull { full() == share(one()) }

    axiom axJoinable { forall t1: Tree, t2: Tree ::
        joinable(share(t1), share(t2)) ==
           joinableTree(t1, t2)
    }

    axiom axPlus { forall t1: Tree, t2: Tree ::
        joinable(share(t1), share(t2)) ==>
            plus(share(t1), share(t2)) ==
            share(plusTree(t1, t2))
    }

    axiom axGeq { forall t1: Tree, t2: Tree ::
        geq(share(t1), share(t2)) ==
        geqTree(t1, t2)
    }

    axiom axMinus { forall t1: Tree, t2: Tree ::
        geq(share(t1), share(t2)) ==>
            minus(share(t1), share(t2)) ==
            share(minusTree(t1, t2))
    }

    axiom axEq { forall a: Tree, b: Tree ::
        eqTree(a, b) ==> share(a) == share(b)
    }

    axiom axUnitPlus { forall p: Perm ::
        plus(unit(), p) == p &&
        plus(p, unit()) == p
    }
}
```

Listing A.1: Equality axiom for tree shares

Bibliography

[1] Viper tutorial. https://viper.ethz.ch/tutorial/.

[2] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40(1): 259–270, jan 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040327. URL https://doi.org/10.1145/1047659.1040327.

[3] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44898-3.

[4] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, 2007. doi: 10.1109/LICS.2007.30.

[5] Thibault Dardinier, Peter Müller, and Alexander J. Summers. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.*, 6 (OOPSLA2), oct 2022. doi: 10.1145/3563326. URL https://doi.org/10.1145/3563326.

[6] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound automation of magic wands. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 130–151, Cham, 2022. Springer International Publishing. ISBN 978-3-031-13188-2.

[7] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 161–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10672-9.

[8] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.

[9] K. Rustan M. Leino. This is boogie 2. June 2008. URL `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`.

[10] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49122-5.

[11] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

[12] Matthias Roshardt. Extending the viper verification language with user-defined permission models. Master's thesis, 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):

Improving User-Defined Permission Models in Viper

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Li	Anqi

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, Switzerland March 15, 2023	[illegible]

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