Automatic Verification of Heap-Dependent Folds in Viper

by

Peeranat Tokaeo

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

December 2023

© Peeranat Tokaeo, 2023
The following individuals certify that they have read, and recommend to
the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis
entitled:

**Automatic Verification of Heap-Dependent Folds in Viper**

submitted by **Peeranat Tokaeo** in partial fulfillment of the requirements
for the degree of **Master of Science** in **Computer Science**.

**Examinining Committee:**

Alexander Summers, Associate Professor, Computer Science, UBC
*Supervisor*

Ronald Garcia, Associate Professor, Computer Science, UBC
*Supervisory Committee Member*
Abstract

Data structures such as lists and arrays are commonly used in real programs to store and manipulate data. To verify such programs, a verification language needs to provide a representation of data structures and tools to reason about them. The Viper verification language enables users to model recursive data structures using predicates and random-access data structures using quantified permissions. While Viper supports heap-dependent functions to express aggregate properties on recursive data structures, it cannot express the same properties for random-access data structures without involving recursion.

We introduce a novel technique to support reasoning about aggregate properties on random-access data structures; we call such properties heap-dependent folds. We provide an encoding of this technique using the built-in constructs of Viper, allowing the verifier to prove fold-like properties without recursion or induction. Finally, we implement our design into a Viper plugin. The plugin introduces a new syntax for expressing folds on user-defined data structures and automatically generates relevant axioms to automate proofs.
Lay Summary

Programmers may want to verify the properties of the programs they write to ensure the programs work as intended. They can model their program inside a verification programming language to do so. Viper is one such verification language that provides a program verifier for automatically proving properties about programs written in Viper. Often, it may be useful to verify properties of the data structures in the program, such as the sum of integers in an array. Such properties are called heap-dependent folds. However, Viper does not support representing these properties.

We present a technique for reasoning about these folds. We encode the technique into Viper and implement it as a plugin to the Viper language. The plugin adds new syntax and generates the necessary assumptions to support automatic proofs about heap-dependent folds. Programmers can now express heap-dependent folds in Viper and verify more properties about data structures automatically.
Preface

This thesis is original, unpublished, independent work by Peeranat Tokaeo. Alexander Summers was the supervisor and helped advise the project throughout.
# Table of Contents

Abstract .................................................. iii

Lay Summary ........................................ iv

Preface ................................................... v

Table of Contents .................................... vi

List of Figures ........................................ x

List of Listings ........................................ xi

Acknowledgments ...................................... xii

1 Motivation .......................................... 1
   1.1 Practical Example: Array Elements Preserved After Sorting . 3
   1.2 Related Work ...................................... 5

2 Background .......................................... 6
   2.1 Program Verification and Viper ...................... 6
   2.2 Viper Framing and Permissions ..................... 7
      2.2.1 The Frame Problem ............................ 7
      2.2.2 Permissions ................................... 8
   2.3 Viper Basic Features and Syntax ................ 10
      2.3.1 Methods ........................................ 10
      2.3.2 Statements ................................... 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Viper State</td>
<td>12</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Gaining and Losing Permissions Through Accessibility Predicates</td>
<td>12</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Quantified Permissions</td>
<td>14</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Domains</td>
<td>14</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Fields</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>SMT Solving and Triggers</td>
<td>16</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Satisfiability Modulo Theories (SMT)</td>
<td>16</td>
</tr>
<tr>
<td>2.4.2</td>
<td>SMT Solving in Viper</td>
<td>17</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Triggers in Quantifiers</td>
<td>17</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Choosing Triggers</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Other Relevant Viper Features</td>
<td>19</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Predicates and Functions</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Design</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Fold Intuition</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Heap-dependent Fold Definition</td>
<td>23</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Well-definedness Requirements</td>
<td>27</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Connection to Viper Quantified Permissions</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Singleton Decomposition</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Combinatorial Explosion from Chained Singleton Decomposition</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Possible Optimization</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Local and Intermediate Decomposition</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Modular Strategy: Nodes and Edges</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Example Combining Exhales and Inhales</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Exhale Case</td>
<td>41</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Modular Design to Handle Exhales</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Inhale Case</td>
<td>45</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Finding the Output Fold Filter</td>
<td>46</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Folds Flowing Backwards with Lookahead</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>Modelling in Viper</td>
<td>51</td>
</tr>
</tbody>
</table>
4.1 Heap-dependent Fold Definition In Viper ........................................ 51
  4.1.1 Defunctionalization ......................................................... 52
  4.1.2 Fold Object ................................................................. 53
  4.1.3 Snapshot Introduction ..................................................... 54
  4.1.4 Encoding FoldApply ....................................................... 57
  4.1.5 Example: Array Encoding ................................................ 60

4.2 Well-definedness Checks .......................................................... 64
  4.2.1 Operator associativity, commutativity, and identity .................. 64
  4.2.2 Injectivity Of Receiver on the Filter .................................. 65

4.3 Encoding the Modular Strategies ................................................ 70
  4.3.1 Encoding the Reassignment Strategy .................................... 70
  4.3.2 Inlined Axiom for Singleton Decomposition .......................... 74
  4.3.3 Encoding the Exhale Strategy ........................................... 76
  4.3.4 Inlined Axiom for Exhales Decomposition ............................. 78
  4.3.5 Encoding the Inhale Strategy ........................................... 82
  4.3.6 Inlined Axiom For Inhale Decomposition ............................... 82

5 Implementation in Scala .............................................................. 86
  5.1 Viper Plugin Architecture .................................................... 86
  5.2 Before Parsing ........................................................................ 87
    5.2.1 Specifying a User Syntax ................................................. 87
    5.2.2 Custom PAst Nodes ......................................................... 89
  5.3 Before Type-checking and Resolving ......................................... 89
  5.4 During Type-checking and Resolution ....................................... 90
  5.5 Translation to AST ................................................................ 91
    5.5.1 Translating the Fold Component Declarations ..................... 91
    5.5.2 Translating the Fold Application ..................................... 92
  5.6 Adding Well-definedness Checks and Inlined Axioms ..................... 93
    5.6.1 Desugaring Method Call .................................................. 93
    5.6.2 Adding Inlined Axioms .................................................... 94

6 Evaluation ....................................................................................... 96
  6.1 Test Cases ................................................................................. 96
List of Figures

Figure 6.1 Six Consecutive Assignments ......................... 98
Figure 6.2 Reassignment Enclosed in Exhale and Inhale ........ 100
Figure 6.3 Multiset of Array Elements After Swap ............. 101
Figure 6.4 Bubble Sort ........................................... 102
Figure 6.5 Graph Sum ............................................. 104
**List of Listings**

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>modifyState Example with Permissions</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Partial Domain for Array</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>hfold Object Constructor</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Snapshot Application Pseudocode</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of a Snapshot Function Declaration (Partial)</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Fold Apply Functions</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Decomposition By Removing 1 Element</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Domain Axiom for General Decomposition</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>Full Array Encoding</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Inverse Axioms for filterReceiverGood</td>
<td>69</td>
</tr>
<tr>
<td>4.9</td>
<td>Singleton Decomposition Example</td>
<td>71</td>
</tr>
<tr>
<td>4.10</td>
<td>Encoding of mapDelete</td>
<td>72</td>
</tr>
<tr>
<td>4.11</td>
<td>Singleton Decomposition Axiom</td>
<td>75</td>
</tr>
<tr>
<td>4.12</td>
<td>Array Exhale Example</td>
<td>77</td>
</tr>
<tr>
<td>4.13</td>
<td>Exhale Decomposition Axiom</td>
<td>79</td>
</tr>
<tr>
<td>4.14</td>
<td>Encoding of mapSubmap</td>
<td>80</td>
</tr>
</tbody>
</table>
Acknowledgments

To my supervisor Alex, thank you for your guidance and teaching throughout the past two and a half years. Our weekly research meetings kept this whole project together, and I don’t know where I would be without them. Thank you for always promptly replying to my Slack messages and emails, whether it be for writing feedback, technical questions, or even casual small talk.

To Ron Garcia, thank you for reading my thesis and giving me feedback despite all the chaos, exams, and holidays in late December. Also, I thoroughly enjoyed taking your course, and CPSC 509 is probably the best course I have taken at UBC. 100% would recommend and take it again.

To all my lab members at the Software Practices Lab, thank you for all the good times and for the many free snacks everyone brought to the lunch table. They really kept me going. Your enthusiasm and curiosity for research were truly inspiring, and I enjoyed all of our social gatherings.

To my family, thank you for all the emotional and financial support. I wouldn’t be here without you, and I hope I have made you proud.

To all my friends in Vancouver, thank you for spending time with me. You are the light that supported me throughout all the rainy days in Vancouver. I treasure all the time I spent with you.

To all my friends elsewhere, thank you for your support behind the scenes. Although we have probably not talked much lately, I hope we can all catch up soon.
Chapter 1

Motivation

Many useful programs rely on data structures stored on the program heap, like arrays and lists, for data storage and manipulation. As such, to prove many interesting properties of a program, a program verifier like Viper should be capable of modelling these data structures.

There are two main ways to describe data structures on the heap in Viper: predicates and quantified permissions. Viper predicates can represent recursive data structures, while quantified permissions allow users to write assertions flatly about a random-access data structure. Together with predicates, Viper functions can denote properties of a recursive data structure. However, no such companion feature exists for quantified permissions to express aggregate properties on a random-access data structure, such as the sum of all elements in an array, or the multiset of all integers stored in a graph. From here on, we refer to these aggregate properties as folds.

Consider the following example with arrays.

Suppose we want to model a random-access array in Viper and describe some of its properties. Firstly, Viper models mutable data as data on the heap. Each unit of data is stored at a field location, also known as a heap location. In Viper, access to memory is restricted, and one must have permission to read or write to it. To express having access to an entire array \( a \), one may write an assertion similar to the following:

\[
\text{assert for all } i: \text{Int} :: i \geq 0 \land i < \text{len}(a) \Rightarrow \text{acc}(a[i])
\]
Here, we use an abbreviation $a[i]$ to denote the heap location of array $a$ at index $i$. This assertion uses a feature called quantified permissions, quantifying over all integer indices $i$ and expressing that the program has access to the array element corresponding to each index. By design, this description of an array with quantified permissions is flat and does not involve recursion. We can express all the indices in the array together at once, following the data structure’s random-access nature.

However, the proofs would still require induction if we try writing cumulative properties over the entire array in Viper. Suppose we want to express the sum of all elements in an array, starting from some arbitrary index $\text{start}$ up to the array length. We could attempt this in Viper by writing the following function:

```viper
function arraySum(a: Array, start: Int) : Int

    requires forall i: Int :: i >= start && i < len(a) => acc(a[i])
    requires start >= 0 && start <= len(a)
    {
        start >= len(a) ? 0 : a[start] + arraySum(a, start + 1)
    }
```

This sum function requires access to the array from the indices $\text{start}$ to the array length. Note that the sum function is defined recursively on the start index. All proofs involving changes in the sum would now require induction.

However, this is in contrast to the random-access nature of arrays. Suppose an array element has changed at index $i$. To show the change to the total sum of all elements expressed in our function, we would need to unroll the body of $\text{arraySum}$ up to index $i$ in a proof by induction.

Instead, we can ideally observe that the sum property can be seen as an unordered fold, and it could be decomposed into

$$\text{sumArray}(a) = \text{sumArray}_{\text{excluding_index_i}}(a) + a[i]$$

i.e., the sum of an array is the $i$th element plus the sum of all other elements. We call such a splitting a decomposition.

Unfortunately, this kind of reasoning is not supported in Viper. In this master’s thesis, we design a reasoning technique and its encoding using
Viper features to enable expressing and proving these properties as folds on random-access data structures. We implement the encoding in a Viper plugin that introduces a new syntax for expressing folds on any user-defined data structure and generates relevant axioms in the background. In addition, the plugin automatically decomposes the relevant folds in response to program heap changes; this allows for reasoning without recursion, in scenarios like the `sumArray` example above. Altogether, this plugin should help the user prove fold-like properties more succinctly, without requiring recursion or induction.

1.1 Practical Example: Array Elements Preserved After Sorting

Folds are generalizable to other practical use cases beyond the sum of an array. For instance, the multiset of elements stored in an array is also a fold. Consider verifying a sorting algorithm on an integer array. There are two properties one may care about:

1. The array is sorted after running the algorithm. Without loss of generality, let us assume the sort is in ascending order.

2. All the original array elements are preserved, possibly arranged in a different order in the resulting array. No new elements are added or removed.

The first sortedness property can be expressed with a quantifier: an array \(a\) is sorted iff for all index \(i\), if \(i > 0\) and \(i < \text{length}(a)\) then \(a[i] \geq a[i-1]\).

The second property can be restated using a fold; the multiset of elements in the array prior to sorting is equivalent to the multiset of elements in the array after sorting. This multiset of array elements can be defined as a fold on an array because it is unordered and can be decomposed. We describe the precise definition of a fold in Chapter 3. Reasoning about the same property is much more difficult with Viper’s built-in function construct.

Most sorting algorithms would call a `swap` method (or inline the same code) to swap elements from potentially disparate portions of an array until
the array is sorted. This swap method could contain modular specifications stating that it preserves the multiset of elements of the relevant subarray, i.e., the result of the fold is the same before and after the swap on that subarray. Then, inside the sorting algorithm, calls to the swap method can trigger a fold decomposition, stating that the elements multiset of the entire array is equal to the multiset-union of (the elements multiset of the indices affected by swap) and (the elements multiset of the indices unaffected). The specifications of the swap function along with framing (discussed in Section 2.2) would automatically prove that the second property holds for a sorting algorithm.

Without support for folds, reasoning about the second property becomes much more difficult. One might define a recursive function on an array to express its multiset of elements, which may look like the following:

```plaintext
1 function elementsInArray(a: Array, start: Int, end: Int):
    Multiset[Int]
2     requires start >= 0 && end <= len(a) && start <= end
3     requires access_array(a, start, end) // denotes access to array
4 {
5     start >= end ? Multiset[Int]():
6         (Multiset(loc(a,start).val) union
7             elementsInArray(a, start + 1, end))
8 }  
```

Reasoning about this function would require induction, like in the previously mentioned sumArray example. If the swap method’s specification depends on this function, reasoning about calling the swap method would be analogously more restrictive, as the user would need to manually unroll the recursive definition until the relevant indices appear. In contrast, with our techniques the user would only need to specify the swap method pre/postconditions and the loop invariants of the sorting algorithm using the appropriate folds, and the proof would go through. We demonstrate the usage of folds in verifying bubble sort more concretely in Section 6.1.3.
1.2 Related Work

This section discusses the high-level overview of the related works to motivate our project. For a more in-depth discussion about technical design decisions within the related works, see Chapter 7.

There are a few past attempts to support reasoning about folds. In his Ph.D. thesis [14], Ter-Gabrielyan describes possible axiomatizations and Viper representations of folds as a way to express heap reachability properties. They introduce the idea of heap snapshots, which we utilize in our own project. However, their technique only serves as a demonstration and does not support automatic decompositions for the proofs, which we do.

Tierry Hörmann’s bachelor’s thesis [9] also provides specifications and encodings of folds in Viper, but similarly, their proofs require decompositions that are not fully automatic. Their axioms involve universal quantifiers that quantify over heaps, which can only be done in Boogie [10], a language which Viper compiles to when using the Carbon verifier. Unfortunately, implementation details were largely omitted, and the project code on Bitbucket is no longer accessible; we are unable to reproduce their results.

Outside of Viper, the Spec# program verifier [2] provides a number of predefined fold expressions, namely sum, count, product, min and max [11]. Each type of fold is supported by Boogie axioms expressed inductively on integers. A user can potentially decompose the integer range into two subranges to circumvent inductive reasoning, but this must be done manually. Otherwise, the user would still tend to rely on induction for the proofs involving folds. In contrast to Spec#, our tool allows the user to define their own custom fold operation, which is not limited to sums, products, etc. We also allow the iterator variable to be of arbitrary type, not restricted to integers. The axioms we generate are general to all of these types and operations, allowing more flexible use cases. We also support automatic decompositions with no additional user inputs required.
Chapter 2

Background

This chapter explains the background knowledge required to understand the rest of this thesis.

2.1 Program Verification and Viper

Program verification is the process of proving the correctness of a program according to its formal specifications. In this process, the programmer first annotates the source program with assertions written in a specification language. A program verifier then checks the annotated program statically, i.e., without executing the program, to verify the annotated assertions. Before the verification, certain verifiers would transform the source program into a verification language program through a verification frontend. For example, the Prusti [1] and Gobra [15] verifiers transform Rust and Go programs, respectively, into the Viper verification language.

Viper [13] is a verification infrastructure which provides an intermediate verification language and two built-in verifier backends. The first backend, Silicon, relies on symbolic execution, and the second backend, Carbon, generates verification conditions in Boogie, another intermediate verification language. Both ultimately use the SMT solver Z3 to check the validity of the input Viper program. Typically, we use the word Viper to refer to the intermediate verification language and Viper backends to refer to the two
backends as described. A Viper frontend, such as Gobra or Prusti, extends a general-purpose programming language with a specification language for annotations and ultimately compiles the program from the source language into Viper for verification.

2.2 Viper Framing and Permissions

A key feature of Viper is its support for permissions logic, which is a solution to the frame problem of heap-modifying programs.

2.2.1 The Frame Problem

The frame problem is a problem that arises in modular program verification when it becomes too verbose to describe how a method or function changes the state of a program. In modular program verification, a call to a method is summarized by preconditions and postconditions rather than explicitly duplicating the method’s body. This way, the method body can be abstracted and described modularly by its specifications. Concretely, a verifier translates a method call into an assertion of the preconditions, i.e., the requirements of the methods, and an assumption of the postcondition, i.e., the effects of the method.

However, in the presence of a global state like the heap, any method body could modify that state. Without any additional specification, one would need to conservatively assume that the entire global state has changed once a method call occurs.

For example, suppose the method `modifyState` modifies an array `a` at index `i`, and suppose that the array elements are stored in the heap, making them global. `a[i]` refers to the array element indexed `i`.

```plaintext
1 method modifyState(a: Array, i: Int)
2   requires length(a) > i
3   ensures a[i] == 10
4   {
5     a[i] := 10
6     a[0] := 0
7   }
```
The method’s postcondition ensures that the array element at index $i$ has a value of 10, but the method body also modifies the element at index 0. The method specifications (without the body) do not specify the values at other array indices, so for soundness, the method caller has to assume that all elements could have changed.

Alternatively, the programmer would have to write more verbose pre/post-conditions to describe the unchanged portion of the global state. For example, the modifyState method specifications must be rewritten to include a framing axiom, i.e., an axiom that prescribes the states that remained unchanged.

```plaintext
1 method modifyState(a: Array, i: Int)
2     requires length(a) > i
3     ensures forall j: Int :: j != i && j >= 0 && j < length(a) ==> a[j] == old(a[j])
4     // old means the state before the method execution
5     ensures a[i] == 10
6     {
7         a[i] := 10
8         a[0] := 0
9     }
```

This becomes excessive quickly because additional assumptions must be added for all heap locations, and every modular heap-modifying component must contain these axioms. In fact, we cannot write assumes describing all other heap data in a modular way. The example method only concerns array indices 0 and $i$, but the postconditions must describe the values at every other index. This is not modular.

### 2.2.2 Permissions

To mitigate the frame problem, Viper uses a *permission logic* to describe heap location permissions that a method may have. A permission is a key to access a heap location, and each heap location has exactly 1 permission. Modular program components in Viper, such as methods, functions, and loops, require permission for specific locations to read and/or modify them.
They can also have fractional permissions between 0 and 1, which may allow components to read the heap locations but not modify them. To write, the method needs to hold full permissions, i.e., 1/1. Fractional permissions are particularly useful for modelling concurrent programs with methods accessing the same global state concurrently.

Framing is achieved when a method has permission to heap locations across two states. Those heap locations are considered unchanged between two states unless modified explicitly by the method itself. If the current method holds some permission to a heap location, other methods would not have full permission, so they cannot possibly modify that state. Returning to the `modifyState` example, we can include the permissions in the method’s pre/postconditions, as shown in Listing 2.1.

```java
1 method modifyState(a: Array, i: Int)
2     requires length(a) > i
3     requires acc(a[i]) && acc(a[0]) // takes permissions
4     ensures acc(a[i]) && acc(a[0]) // gives permissions
5     ensures a[i] == 10
6     {
7         a[i] := 10
8         a[0] := 0
9     }
```

**Listing 2.1:** modifyState Example with Permissions

Suppose that another method calls `modifyState`. The method call only takes permissions to array indices 0 and i. Any other index can be assumed unchanged, provided the caller has these permissions. The specification does not need to include an explicit framing axiom denoting that all other indices remain the same. Unrelated heap locations, relevant only to the caller, do not need to be described in the `modifyState`; this makes the method specifications modular.

From a modular component’s perspective, giving permissions away, such as by calling a method, means losing knowledge about a heap location. Likewise, if a method holds permissions for a heap location, it knows that the location could not have been modified while it currently holds permissions.
There is only 1 permission per heap location, and any other method would have needed permissions to modify it.

Notice also that the `modifyState` method takes permissions in the `requires` but also gives those permissions back in the `ensures`. From the caller’s perspective, it gives `modifyState` permissions and gets the permissions back after that method is finished.

The kind of permission logic Viper employs is more precisely called implicit dynamic frames. “Frames” refers to framing modular program components, like method calls, function calls, and loops. “Implicit” refers to how the frames are implicitly determined by the permissions given to a method, and methods are framed to the portion of the global state to which they have permissions. No explicit framing axioms are required. The frames are “dynamic” because the permissions may depend on values in the program, so permission checks must be done with calls to the SMT solver.

### 2.3 Viper Basic Features and Syntax

Next, we discuss some of the fundamental Viper constructs needed for future reference when demonstrating our design with Viper code. The code examples that we have described so far are in pseudocode. So, in this section, we introduce the correct Viper constructs to transform our pseudocode example to real Viper. Let us use the previous pseudocode snippet in Listing 2.1 as the leading example.

#### 2.3.1 Methods

Our `modifyState` snippet above is a method declaration. A Viper method is a sequence of statements where much of Viper’s verification happens. Viper handles a method call modularly by only using the method’s preconditions and postconditions, denoted in the method declaration with keywords `requires` and `ensures`, respectively. Viper verifies that the method satisfies its specifications once, and any calls to the method get abstracted by its specifications.
In verifying a method, Viper first inhales\(^1\) the preconditions, verifies the body and then verifies that the postcondition holds. Verifying and calling a method have opposite behaviours. Verifying a method means inhaling its preconditions and exhaling its postconditions. Calling a method means exhaling the preconditions and inhaling the postconditions. \texttt{assert} checks the condition written in the statement, and \texttt{assume} assumes the conditions as true. If another method, \texttt{caller}, calls \texttt{modifyState}, the call essentially gets translated into exhales and inhales, as in the following example.

```
method caller(a: Array, i: Int)
  requires length(a) > i
  requires acc(a[i]) && acc(a[0])
  ensures acc(a[i]) && acc(a[0])
  {
    // modifyState(a,i) desugars into exhales and inhales of its specifications.
    exhale length(a) > i
    exhale acc(a[i]) && acc(a[0])
    inhale acc(a[i]) && acc(a[0]) // gives permissions
    inhale a[i] == 10
  }
```

### 2.3.2 Statements

Inhales, exhales, and method calls are examples of Viper statements. Viper statements include common programming operations such as variable declarations, assignments, if-statements, and while-loops.

Here are some Viper statements that are not common programming operations.

- \texttt{assert} A: Check the permissions and propositions in A.
- \texttt{assume} A: Assume the permissions and propositions in A.
- \texttt{exhale} A: Remove the permissions and assume the propositions in A.

\(^1\)For now, think of exhales as asserts and inhales as assumes. We discuss their differences later.
• **inhale A**: Add the permissions and assume the propositions in A.

Asserts and assumes are statements that do not modify permissions, whereas inhales and exhales do. When the statements do not contain accessibility predicates, asserts and exhales can be used interchangeably, as with assumes and inhales.

### 2.3.3 Viper State

According to the Viper tutorial, the Viper program state includes

- All variables in scope, such as the local variables, input parameters, and return variables.
- Permissions to heap locations
- Values stored in heap locations

One can use old expressions like `old[a](...)` to denote the value at a particular state, defined by some label `a`. Old expressions, however, only work for heap values, not general variables.

Statements that modify state are impure. As such, variable and heap reassignment with `:=` are impure, as are **inhale** and **exhale** statements that modify permissions.

### 2.3.4 Gaining and Losing Permissions Through Accessibility Predicates

Differences between asserts and exhales appear when accessibility predicates are used, as denoted by `acc(A)` where A mentions a heap location. For example, the caller method body has the statement `exhale acc(a[i]) && acc(a[0])`. An accessibility predicate describes access to a specific heap location and is considered an impure expression, because inhaling or exhaling the predicate may modify the current permissions. Inhaling an accessibility predicate means gaining permissions to the heap location, and exhaling an accessibility predicate means losing permissions.
Perm expressions

A perm expression, written as perm(A), represents permissions to a heap location as a fraction. Unlike accessibility predicates, perm expressions inside inhale or exhale statements do not cause the state to lose or gain any permissions. So, they are useful for querying permissions inside an inhale or exhale statement without gaining or losing any permissions. Additionally, they can be used in general Viper expressions, while accessibility predicates are restricted to statements. Also, perm expressions are fractions, and accessibility predicates are boolean values, so their usages are different. For instance, one could write an inequality with perm expressions denoting permissions greater than 1/2, which would not be easy to express with accessibility predicates.

Separating Conjunctions

Accessibility predicates can be conjoined together using the && operator, as in exhale acc(a[i]) && acc(a[0]). This represents a separating conjunction in Viper. The && operator behaves like the ordinary logical & operator for other expressions. But on accessibility predicates, the separating conjunction requires that the permissions must be separate, i.e., we have the sum of the conjoined permissions.

For example, exhale acc(a[i]) && acc(a[i]) translates to checking that the state contains 1 permission for heap location a[i] and also another 1 permission for a[i]. This would require 2 permissions to the same location, a[i], which is not possible by definition. So, the exhale would always fail and raise a verification error.

This separating property of the && operator helps prevent complications from aliasing. If x and y are aliases of the same reference, exhale acc(x.F) && acc(y.F) would fail. Likewise, inhale acc(x.F) && acc(y.F) implies that x and y are not aliases.
2.3.5 Quantified Permissions

Although multiple accessibility predicates can be conjoined with the \&\& operator, sometimes we need to use them inside a universal quantifier. For example, we may want to express having permissions from array indices 0 up to an unbounded array length. Explicitly writing out each index is impossible, so we want to write the accessibility predicates inside a quantifier, like in the following code snippet.

```viper
method hasAllArray(a: Array, i: Int)
    requires forall i: Int :: i >= 0 && i < len(a) ==> acc(a[i])
    // This precondition expresses having permissions to all array locations.
```

Accessibility predicates that appear inside a universal quantifier are called quantified permissions. The usual form of quantified permissions is \texttt{forall x : T :: c(x) => acc(r(x).F, p(x))}, where \texttt{T} is an arbitrary type, \texttt{c(x)} is some boolean expression \texttt{x}, \texttt{r(x)} is a receiver expression, \texttt{F} is a field, and \texttt{p} is a expression denoting the fractional permissions at each \texttt{x}. \texttt{c} can be thought of as a filter limiting the quantified permissions over the set of input \texttt{x} that passes the filter, and \texttt{r(x).F} represents the heap location.

In our \texttt{hasAllArray} example, \texttt{forall x : T} is \texttt{forall i: Int}. The boolean expression \texttt{c(x)} is \texttt{i >= 0 && i < len(a)}, and the access predicate \texttt{acc(r(x).F, p(x))} is \texttt{acc(a[i])}; here, \texttt{p(x)} is omitted, which gets interpreted as the full (1/1) permissions in Viper.

Now, we have been using the pseudocode \texttt{a[i]} to represent the array element at index \texttt{i}, but arrays do not actually have native support in Viper. To model arrays properly, we need to use Viper domains.

2.3.6 Domains

An array type can be declared as a Viper domain. Viper domains allow the definitions of new types, functions, and axioms. A Viper domain partially modelling arrays is shown in Listing 2.2. We define a new type \texttt{domain Array} and two new functions. These functions are uninterpreted functions without bodies, so we must write axioms to describe their desired behaviour. In
domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int

  axiom len_nonneg {
    forall a: Array :: { len(a) }
    len(a) >= 0
  }
}

Listing 2.2: Partial Domain for Array

domain Example[A] {
  function f(a:A) : A
}

The function f is polymorphic; it takes an input of any type A and outputs the same type. Viper also allows declaring functions outside of domains. Those functions can depend on the heap and have bodies, but notably, they cannot use type variables like domain functions. We discuss more about these non-domain functions later.
2.3.7 Fields

In Viper, global memory is represented by fields inside objects. Objects are represented with values of type \texttt{Ref}, and all objects contain all fields declared in the program. Objects and fields paired together make up \textit{heap locations}, also called field locations. Intuitively, one can conceptualize the heap as a mutable map of type $(\texttt{Ref}, \texttt{Field} \ A) \rightarrow A$. In fact, the Viper backends would model the heap as exactly this map!

Fields can be declared at the top level using the keyword \texttt{field}. For example, \texttt{field val: Int} declares a field of type \texttt{Int} called \texttt{val}. The field \texttt{val} of some reference \texttt{r} is represented by \texttt{r.val}, which can then be read and/or written to, given that we have permissions to that heap location.

So, the pseudocode representing an array element \texttt{a[i]} should instead be written as \texttt{loc(a,i).val} in Viper.

2.4 SMT Solving and Triggers

To check the \texttt{exhale} and \texttt{assert} statements inside methods, Viper relies on generating verification conditions for an SMT solver.

2.4.1 Satisfiability Modulo Theories (SMT)

An SMT solver is a tool that solves satisfiability modulo theories problems or SMT for short. SMT problems are a generalization of the Boolean satisfiability problem, SAT for short. A SAT problem involves finding a solution to a Boolean formula by assigning a value to each variable. A model is a map from all variables to their respective truth value. The formula is considered \textit{satisfiable} if a solution exists. For example, $A \land B$ is satisfiable with the model mapping $A = \text{true}$ and $B = \text{true}$. A formula is considered \textit{valid} if it always holds, regardless of the model. For example, $A \lor \neg A$ is valid because the formula evaluates to true regardless of $A$’s truth value. A formula is considered \textit{unsatisfiable} if no satisfying model exists.

SMT extends the SAT problem by adding theories, allowing for more complex expressions such as addition, equality, and inequality. Numbers and functions are also allowed in addition to Boolean variables. For example,
\[ f(x) + x > 1 \] is an SMT formula, and solving it would require finding the assignments to number \( x \) and function \( f \) such that the formula holds. Satisfiability, unsatisfiability, and validity also mean the same in SMT solving.

### 2.4.2 SMT Solving in Viper

In Viper, we want to check each `assert` statement given some set of assumptions, either from `assume` or explicit axioms. To do so, Viper generates the required verification conditions and gives them to Z3, an SMT solver from Microsoft Research [4]. Generally, we want to check whether assumption \( A \) entails assertion \( A \), i.e., \( A \models B \). **Entailment** means that for all models \( M \) such that \( M \) satisfies \( A \), then \( M \) also satisfies \( B \). Consider the following Viper program.

```viper
method smt_example(x: Int) {
    assume x > 10
    assert x != 0
}
```

We want to solve the entailment \( x > 10 \models x \neq 0 \). It is known that checking entailment \( A \models B \) is equivalent to checking the validity of implication \( A \rightarrow B \), which is also equivalent to checking the unsatisfiability of \( A \land \neg B \). So, for this `smt_example` method, the Viper verifier would give the conditions \( x > 10 \land \neg(x \neq 0) \) to the Z3 solver to check for unsatisfiability.

### 2.4.3 Triggers in Quantifiers

Viper also allows writing universal quantifiers and existential quantifiers. Universal quantifiers (also called forall or for-all quantifiers) are written as \( \forall x: \text{Int}.A \) denoting that \( A \) holds for all possible integer \( x \). Existential quantifiers are written as \( \exists x: \text{Int}.A \), denoting that an integer \( x \) exists with property \( A \).

Universal quantifiers are particularly useful in Viper for writing domain axioms, as in the array encoding domain example in Listing 2.2. Technically, a for-all quantifier represents a possibly infinite conjunction over the quantified type. So, the `len_nonneg` axiom is saying
len(a_1) >= 0 && len(a_2) >= 0 && len(a_3) >= 0 && ...

including all possible arrays \texttt{a_i: Array}. The SMT solver cannot instantiate every possible \texttt{a_i} from the quantifier, so it instead relies on the trigger, which is enclosed inside the curly braces \{\ldots\}. The term quantifier will only be instantiated to yield \texttt{len(a_i) >= 0} as an assumption if elsewhere \texttt{len(a_i)} is mentioned for the same \texttt{a_i}. The triggering term could be written in a statement by the user or could be learned from the instantiation of a different axiom.

The SMT solver will also match the trigger using known equivalence. For example, suppose an axiom uses \{g(f(x))\} as a trigger. Then, suppose there is an assumption that \texttt{f(x) = 5}. Then, the term \texttt{g(5)} will match the trigger because the SMT solver can replace \texttt{5} with \texttt{f(x)}, knowing they are equal. This is called \textit{e-matching} \cite{note}, where equivalence is also used for trigger matching. Moreover, the SMT solver’s backend uses a data structure called an \textit{e-graph} that helps track equivalence between terms. Terms that exist as an assumption are considered added to the e-graph.

\section{2.4.4 Choosing Triggers}

Triggers must be chosen carefully to ensure that the axioms are instantiated when necessary. Consider the following example.

```
method not_triggered() {
    assume forall x: Int :: {f(x)}
    f(x)
    assume forall x: Int :: {f(x)}
    f(x) => false
    assert false
}
```

In the method \texttt{not_triggered}, there are two axioms with universal quantifiers; one says that \texttt{f(x)} is always true and the other says that \texttt{f(x)} implies false, for any integer \texttt{x}. Together, this should prove false. However, the assertion of false fails because neither axiom ever instantiates; the term \texttt{f(x)} does not appear anywhere else. Explicitly writing out any instance \texttt{f(x)} would make the proof go through, like in the following example.
method triggered() {
  assume forall x: Int :: {f(x)}
  f(x) => false
  assume forall x: Int :: {f(x)}
  f(x)
  var y: Int // added
  assert f(y) // added
  assert false
}

Though we want the quantifier to instantiate whenever necessary, we also want to avoid matching loops [6], which occur when a choice of triggers allows axioms to be instantiated indefinitely.

### 2.5 Other Relevant Viper Features

#### 2.5.1 Predicates and Functions

A predicate in Viper is a top-level declaration that represents parameterized assertions. Its definition could contain recursive calls (calls to itself) and resource assertions, such as access to a heap location, i.e., field. A predicate instance could represent access to a heap data structure. For example, we could model a linked list with a predicate as follows:

```viper
field elem: Int
field next: Ref

predicate list(this: Ref) {
  acc(this.elem) && acc(this.next) &&
  (this.next != null => list(this.next))
}
```

The `elem` field models the integer stored at each list node, and the `next` field models the pointer to the next list node. The list instance with input `this` means that we have access to fields `elem` and next of `this`, and if the `next` reference points to a node, we also have the list instance with that node as input. In short, if the predicate `list(this)` holds, then we have access to
the whole list, starting with node \texttt{this}.

Next, we may also be interested in certain properties of the list, such as the sum of all the elements of the list. To express that sum, we use the following function definition:

\begin{verbatim}
function listSum(l: Ref) : Int
  requires list(l)
  
  unfolding list(l) in l.next == null ? l.elem :
  l.elem + listSum(l.next)

\end{verbatim}

A function defined outside of a domain can have its own body and pre and postconditions. However, its body must be an expression, unlike a method body, which may contain a list of statements. A non-domain function in Viper is also called a heap-dependent function because it can access heap elements via fields and permissions.

In the above code snippet, the \texttt{unfolding} keyword allows the program to access terms inside the predicate instance. In the \texttt{listSum} function body, we read \texttt{l}'s next field, whose permissions we acquire from unfolding the predicate \texttt{list(l)}. It is worth noting that the function is defined recursively to match the recursive structure of the list defined in the \texttt{list} predicate. Predicates and functions allow us to model recursive data structures and describe their properties. However, for random-access data structures, it is more appropriate to use quantified permissions.
Chapter 3

Design

After providing the necessary background, we now discuss the main concern of our project: heap-dependent folds. This chapter formally defines heap-dependent folds and discusses our developed axiomatizations to allow reasoning about them in Viper. Although the design is for Viper, we avoid going too deeply into the implementation details in this chapter, keeping things more mathematical. The next chapter will discuss the Viper axioms we designed in more detail, including selecting triggers, controlling instantiations, etc.

3.1 Fold Intuition

Consider once more our goal of describing a comprehensive property for a random-access data structure, for example, the sum of an integer array. What are some components or steps to calculate the sum of all integers in an array? First, we start with a set of indices of interest, which could be of any type. For arrays, it is a set of integer indices. We call this the filter. Then, given an index, we need some way to extract the element corresponding to that index. For an array $a$, we use $a[\text{index}]$ to access that element at the index. This mapping from an index to a data point is called a receiver expression. Finally, we combine these data points by adding them up, i.e., applying addition.
Addition is an example of an *operator*, a binary function describing how to combine each pair of values. Addition is special because it is commutative and associative; we can add array elements in any arbitrary order and get the same sum. One key observation is that an array sum can be decomposed. One could calculate the sum of half of the array and the sum of the other half; combining the two sums with addition still gives the total array sum. An array sum is considered an unordered fold, meaning the same two-way split can be made even into non-contiguous parts. These properties derive from the commutative and associative properties of the operator, which enable flexible decompositions of the sum. We focus our support on corresponding unordered folds in this thesis. Finally, if the array is empty, i.e., there are no indices, we consider the sum to be 0, which is the *identity* of the addition operator.

Two additional components to calculating the sum of all elements in an array are the *field* and *mapping*. In Viper, each mutable array element is stored on the heap, which can be accessed through a field and a reference. A receiver expression maps an index to a reference, and combining the reference with a field accesses the data point. A mapping function is applied to each data point before being fed to the operator. For example, one may wish to sum up all integers in the array, each squared. After accessing each array element through the receiver expression, we want to square the element before adding them. We call this additional operation on individual elements a mapping.

These components described earlier can be generalized beyond a sum of array elements. The set of indices could have elements of an arbitrary type, and we can customize the receiver expression and field to map from that type to some data. The operator could also be any binary operator that is commutative and associative and has an identity. After all these generalizations, one can still perform decomposition and recombine each portion using the operator.

A fold is the evaluation of all these components combined, namely the filter, receiver, field, mapping, and operator. Specifically, we are concerned with *heap-dependent folds* because data points of heap data structures are
Definition 1 (Heap-dependent Fold). A heap-dependent fold is defined by the following 5 components:

1. Filter: a set of indices that are relevant, typically defined by some filtering condition over some specified type.

2. Receiver: an expression mapping the set of indices to a set of references. Each index must map to a different reference; the receiver is required to be injective (more on this in the well-definedness section).

3. Field: this combines with the references to get the relevant heap locations

4. Mapping: a function to be applied to the values stored in the relevant heap locations.

5. Operator: an operator that combines the mapped values together using the defined binary operator. The values are folded together.

3.1.1 Heap-dependent Fold Definition

We now provide a formal notation for heap-dependent folds. Let

$$\text{fold}_\sigma[o](m(r.F) \mid f)$$

be the fold over all heap elements defined by filter $f$, receiver $r$, and field $F$, where heap $\sigma$ is the current heap, and operator $o$ is the operator object containing a commutative and associative binary operation $(B, B) \to B$ and an identity element $i_0 : A$ for that operation. Filter $f$ is a set of unordered elements, each of type $A$. The receiver $r$ is a function $A \to \text{Ref}$. The mapping $m$ is a function from the field type $V$ to output $B$, i.e., $V \to B$. The fold value depends on the current heap $\sigma$.

At different program points, a heap-dependent fold typically varies only in the heap $\sigma$ and filter $f$. For example, suppose the programmer is interested in the sum of an integer array in the method. The array elements could be stored on the heap.
modified, changing $\sigma$, or perhaps we can take a subarray, changing filter $f$.
So, in contexts where other arguments obviously remain constant or are not relevant to the discussion, we omit $r, F, m, o$, leaving only

$$\text{fold}_{\sigma}(f).$$

**Evaluation of Heap-dependent Fold**

Intuitively, a heap-dependent fold evaluation simply aggregates all values on the heap defined by the filter, receiver expression, and field; each of the five components contributes a computation to the process. The filter produces a set of relevant indices, and the receiver function is applied to those indices, giving a set of references. These references, paired with a field, point to some heap locations, and reading them would give a set of values stored on the heap. Two such values can then be chosen arbitrarily to be combined using the binary operator. The values are combined repeatedly until only one value remains. If the filter is empty, the output is the identity $i_0$ of the operator.

The only addition to this description is the presence of a mapping function. Often, it may be useful to apply a function to each heap value before performing the combine prescribed by the binary operator. For example, suppose we want to collect all the integers stored in a heap chunk into a set. Then, the binary operator should output an integer set, which would require each operator input to be an integer set, as enforced by the operator’s type signature $(A, A) \to A$. However, each heap location stores an integer, which causes a mismatch of types. We can circumvent this by introducing a mapping function that constructs a singleton integer set from an integer.

The evaluation of a heap-dependent fold proceeds as follows:

- If the filter $f$ is empty, the fold evaluates to $i_0$, where $i_0$ is the identity element.

- If the filter $f$ is a singleton set $\{x\}$, the fold evaluates to $m(r(x).F)$, where $r$ is the receiver function, $F$ is the field, and $m$ is the mapping
function.

- If the filter \( f \) can be deconstructed such that \( f = f_1 \uplus f_2 \), where \( f_1, f_2 \) are filters and \( \uplus \) represents the disjoint union (\( f_1 \) and \( f_2 \) are disjoint sets), then

\[
\text{fold}_\sigma [o](m(r.F) \mid f) = \text{fold}_\sigma [o](m(r.F) \mid f_1) \uplus_o \text{fold}_\sigma [o](m(r.F) \mid f_2),
\]

where \( \uplus_o \) is the infix notation for the binary operator defined by \( o \).

This rule splits the fold into two and combines them using the binary operator \( o \).

These rules do not prescribe the exact way to calculate a fold. For example, one could repeatedly use the third rule to split up a filter with the empty set, i.e., \( f = f \uplus \{\} \), and there would be no computational progress.

The third rule above describes a decomposition, which splits a filter into two smaller filters, \( f_1 \) and \( f_2 \). The challenge in reasoning about the fold values is finding the two smaller filters suitable for the decomposition. In practice, we only find one smaller filter \( f_1 \). If we have an existing filter \( f \) and find a subset \( f_1 \) of \( f \), we can derive an \( f_2 \) as a set-minus, i.e., \( f_2 = f - f_1 \).

So, when we say decompose \( f \) with \( f_1 \), we mean \( f = f_1 \uplus (f - f_1) \).

However, there is an unbounded number of indices for an unbounded data structure, i.e., a data structure that can have an indeterminate size. Exploring every possible decomposition given an unbounded set of indices is impractical. This is one key challenge to our design in Viper.

**Viper’s Heap Structure**

Let us discuss heap \( \sigma \) in more detail. In Viper, there are two components to the heap in the program state:

- The heap value map: a total map representing values stored on the heap, typed \( \text{(Ref, Field A)} \rightarrow A \)

- The permission mask: a total map representing permissions to each heap location, typed \( \text{(Ref, Field A)} \rightarrow \text{Perm} \)
We model these two using just one partial map $\sigma$, typed $(\text{Ref}, \text{Field } A) \to A$. When permissions are lost to a heap location, we delete the appropriate keys in $\sigma$. When permissions are gained, we add the new keys to $\sigma$, with unspecified default values.

Two folds with two different heap arguments could still be equal if the relevant heap portions are equal. Consider the case when two heap-dependent folds have the same arguments other than the heap i.e., one has heap $\sigma_0$, the other $\sigma_1$. If

$$\forall e \in f. \sigma_0[r(e), F] = \sigma_1[r(e), F],$$

then $\text{fold}_{\sigma_0}(f) = \text{fold}_{\sigma_1}(f)$. That is, if two heap-dependent folds differ only in their heap argument and their respective heap portions defined by the filter, receiver, and field are equal, then the two folds are equal.

We define a heap snapshot as a portion of the heap limited by the filter, receiver, and field. Formally,

$$\text{snap}_{r.F}^f(\sigma) = \{ e \to \text{value} \mid ([r(e), F] \to \text{value}) \in \sigma \land e \in f \}.$$  

For convenience, we have changed the key from a $[\text{Ref}, \text{Field}]$ to an element of the filter $e \in f$, but it expresses the same thing considering the receiver $r$ is injective with respect to the filter $f$. Also, we removed the field from the key because we specified the field $F$ as the only relevant field. These little details are not completely relevant now, but this snapshot definition will be crucial in the implementation detail of the next chapter.

We write

$$\text{snap}_{r.F}^f(\sigma_0) = \text{snap}_{r.F}^f(\sigma_1)$$

to denote two heaps equal with respect to receiver $r$, field $F$, and filter $f$, i.e., their snapshots are equal. So,

$$\forall \sigma_i, \sigma_j. \text{snap}_{r.F}^f(\sigma_i) = \text{snap}_{r.F}^f(\sigma_j) \implies \text{fold}_{\sigma_i}(f) = \text{fold}_{\sigma_j}(f).$$

Technically, one could modify the fold notation to include the snapshot,
combining receiver $r$, field $F$, filter $f$, and heap $\sigma$ into one expression:

$$\text{fold}[\sigma]\{m(\text{snap}_f^{r,F}(\sigma))\}.$$ 

We use the previous notation in this chapter, but when discussing Viper encoding in the next chapter, we will use this snapshot representation, as it aligns more closely with the Viper code.

### 3.1.2 Well-definedness Requirements

There are several requirements to make the heap-dependent fold have a well-defined mathematical meaning. Firstly, the types have to match. The filter $f$ must have type $\text{Set}[A]$, and the receiver has type $A \rightarrow \text{Ref}$, and given that the field has type $V$, the mapping has type $V \rightarrow B$. The binary operator then has type $(B, B) \rightarrow B$ and has an identity typed $B$. Implementation-wise, this all gets encoded as part of the Viper language type-checker, which we discuss in Section 5.4.

There are other requirements that need to be checked separately by the SMT solver:

**Operator must be associative, commutative, and have the given identity**

In an unordered fold, each element in the input set can be combined in any arbitrary order, mimicking the random access nature of a data structure. So, the operator must be provably associative and commutative.

Following the third rule of a fold evaluation, i.e., the decomposition rule, we can always decompose a filter into itself and the empty set. So, when the input set is empty, the fold must still evaluate to a value for soundness. So,

$$\text{fold}_\sigma(f) = \text{fold}_\sigma(f) \oplus \text{fold}_\sigma(\{\})$$

would hold for all $f$ if the fold on the empty set evaluates to the identity of the operator.
**Receiver must be injective on the filter**

Each element of the filter set must map to a different reference. This greatly simplifies the implementation because each heap location maps to one element in the filter. Due to the bijection between them, we can interchange quantifying over the elements or references. Moreover, if one heap location gets reassigned, the decomposition would involve extracting that one element from the original filter.

We designed our feature to work mostly with quantified permissions, which already require the injectivity of the receiver in Viper. So, this requirement is not really an additional restriction, provided the user intends to use quantified permissions to describe the permissions to the underlying data structure.

**Current state must have enough permissions to all heap locations defined by the filter and receiver**

A Viper method cannot read values from heap locations it does not have permission to. So, a fold can only be evaluated at a program point if it has permissions to all the heap locations defined by filter $f$ and receiver $r$.

Technically, in Viper, the heap and permissions are separate maps. However, we have overloaded our heap notation $\sigma$ as a partial heap representing both the heap and the permissions. If permissions have changed, then we consider the heap $\sigma$ to have changed too. One can think of $\sigma$ as representing the heap portion with permissions.

### 3.1.3 Connection to Viper Quantified Permissions

Recall that quantified permissions in Viper have the canonical form $\forall x: T:: c(x) \Rightarrow acc(r(x).F, p(x))$. Some design choices were adopted from this to heap-dependent folds.

1. The filter set $f$ is analogous to the $\forall x: T:: c(x)$ portion of quantified permissions, which defines a set of elements typed $T$.

2. The receiver function $r$ is an abstraction of the receiver expression $r(x)$.
in quantified permissions.

3. Field $F$ denotes the relevant field.

4. Receivers $r$ must be injective in both quantified permissions and heap-dependent folds, with respect to filter $f$ representing boolean expression $c(x)$.

### 3.2 Singleton Decomposition

This section discusses our design for decomposing heap-dependent folds in response to a heap-modifying operation. Suppose we have a heap-dependent fold

$$\text{fold}_{\sigma_0}[o](m(r.F) \mid f)$$

at a program point with heap $\sigma_0$. Given that we do not modify any other argument in this discussion, we abbreviate this as

$$\text{fold}_{\sigma_0}(f).$$

Suppose the heap has been modified, and at a new program point, we have heap $\sigma_1$. What are some facts that we can deduce about $\text{fold}_{\sigma_0}(f)$ and $\text{fold}_{\sigma_1}(f)$?

To answer this, we consider scenarios where the heap can be changed in Viper. There are only three such scenarios.

1. A heap location gets explicitly reassigned with the $\texttt{:=}$ operator, e.g.,
   $$\texttt{r.val := 0}$$

2. The program state exhales permissions to a portion of the heap.

3. The program state inhales permissions to a portion of the heap.

A method call is considered a combination of scenarios 2 and 3 because the Viper verifier desugars a method call as exhales and inhales, something that we will do manually. Losing and gaining permissions technically does not always modify the heap, but instead the permission mask of the heap.
However, in our formalization, we use $\sigma$ to denote the partial map of the heap portion with permissions, which will have changed after inhales and exhales that modify permissions.

Consider an example of the first scenario, which will be the focus of this section. Suppose we are interested in a fold with the sum operator $o_{\text{sum}}$, which is a binary operator with identity $0$. Let the mapping $m$ be the identity function, so $m(A) = A$. Then, suppose we have an arbitrary $i \in f$ and encounter a reassignment $r(i).F := r(i).F + 1$. Suppose the current method also has permissions to all heap locations defined by filter $f$. Let the heap prior to the reassignment be $\sigma_0$, and the heap after is $\sigma_1$. Intuitively, we can see that the sum over filter $f$ and receiver $r$ must now be $1$ larger than previously.

Now, suppose the user writes an assertion to check:

$$\text{fold}_{\sigma_1}[o_{\text{sum}}](f) = \text{fold}_{\sigma_0}[o_{\text{sum}}](f) + 1.$$ 

We can construct a new filter by removing element $i$ from the original filter $f$ to prove this assertion. This gives us the decomposition of $f$:

$$f = (f - \{i\}) \cup \{i\}$$

With this decomposition, we can use the third rule of evaluating a heap-dependent fold, which gives, for $\sigma_0$

$$\text{fold}_{\sigma_0}(f) = \text{fold}_{\sigma_0}(f - \{i\}) \oplus_{\text{sum}} \text{fold}_{\sigma_0}(\{i\})$$

and similarly for $\sigma_1$,

$$\text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_1}(f - \{i\}) \oplus_{\text{sum}} \text{fold}_{\sigma_1}(\{i\}).$$

Notice, however, that $\text{fold}_{\sigma_0}(f - \{i\})$ and $\text{fold}_{\sigma_1}(f - \{i\})$ are equal, because only heap location $r(i).F$ has been modified. Filter set $f - \{i\}$ does not include element $i$, so the heaps $\sigma_0$ and $\sigma_1$ are identical with respect to this
filter. Formally, we know that

\[ \forall e \in (f - \{i\}).\sigma_0[r(e), F] = \sigma_1[r(e), F], \]

i.e., the heap portions defined by the filter are equal. Or in other words,

\[ \text{snap}_{r,F}^i(\sigma_0) = \text{snap}_{r,F}^i(\sigma_1). \]

We need injectivity to derive this equality. \( r(i) \) maps to a unique reference, and injectivity guarantees that there is no other \( j \in f \land j \neq i \) such that \( r(i) = r(j) \). So, removing \( i \) from \( f \) means nothing else points to the same reference.

Furthermore, Viper permissions also guarantee that no other state change has occurred during the reassignment and that \( \sigma_0 \) and \( \sigma_1 \) differ only at \( r(i).F \). Remember that we assumed the current method holds permissions to all heap locations defined by the filter, which means nothing else could modify these locations other than the method. These locations are guaranteed to have stable values unless the method makes the changes itself.

Next, consider the second fold with the second filter in our decomposition, \( \text{fold}_{\sigma_0}({\{i\}}) \). The filter is a singleton, so by definition of a heap-dependent fold, this evaluates to \( \sigma_0[r(i), F] \) (the mapping is the identity function). Similarly, we consider \( \text{fold}_{\sigma_1}({\{i\}}) \), which evaluates to \( \sigma_1[r(i), F] \). Now, recall that the two heaps occur before and after the reassignment \( r(i).F := r(i).F + 1 \).

By definition of the reassignment, we know

\[ \sigma_1[r(i), F] = \sigma_0[r(i), F] + 1. \]

So, \( \text{fold}_{\sigma_1}({\{i\}}) = \text{fold}_{\sigma_0}({\{i\}}) + 1 \). Substituting these equalities into the decomposition of \( \text{fold}_{\sigma_1}(f) \), we get

\[ \text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_1}(f - \{i\}) \oplus_{\text{sum}} \text{fold}_{\sigma_1}({\{i\}}) \]

\[ \text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_0}(f - \{i\}) \oplus_{\text{sum}} (\text{fold}_{\sigma_0}({\{i\}}) + 1) \]

\[ \text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_0}(f) + 1 \]
We use the associativity of addition in that last step.

So, we have proven what we wanted. Let us recall the steps of our proof to relate the fold prior to the heap change to the fold after:

1. Start with some fold$_{\sigma_0}(f)$

2. The heap $\sigma_0$ encounters a modification to some location $r(i).F$, which produces a new heap $\sigma_1$

3. Perform a decomposition on the filter $f$ to extract two filters. The first filter denotes the portion of the heap unchanged. The second filter denotes the portion changed. In our example, we split $f$ into $(f - \{i\}) \cup \{i\}$.

4. Use the third evaluation rule of fold, which gives $\text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_1}(f - \{i\}) \oplus \text{sum} \text{fold}_{\sigma_1}(\{i\})$, and the same for $\sigma_0$.

5. Prove that $\text{fold}_{\sigma_1}(f - \{i\}) = \text{fold}_{\sigma_0}(f - \{i\})$ using injectivity and Viper permissions.

6. Prove that $\text{fold}_{\sigma_1}(\{i\}) = \text{fold}_{\sigma_0}(\{i\}) + 1$ using the second evaluation rule of fold, i.e., the singleton rule.

7. Substitute equalities to deduce

$$\text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_0}(f) + 1$$

Notice that we did not need to use induction in this argument. The argument is independent of the choice of the operator or any other component in the fold, meaning the same argument can be made for any fold. We can automate this decomposition strategy, which we discuss in the next chapter.

In practice, we should also make an extra check to confirm that the update is relevant to the fold. In our example, this amounts to checking $i \in f$ before performing the decomposition.
3.2.1 Combinatorial Explosion from Chained Singleton Decomposition

Although the simple strategy described previously can handle a singleton heap reassignment, it can cause a combinatorial explosion when many reassignments are chained together. Consider a situation where repeated reassignments are made. Suppose we have a filter \( f \) and two arbitrary integers \( i, j \in f \). Then, consider a method which performs two consecutive reassignments to the heap locations of each index.

\[
\begin{align*}
\{ & \\
\ldots & \\
r(i).F := r(i).F + 1 & \\
r(j).F := r(j).F + 1 & \\
\}
\]

Again, we want to prove that

\[
\text{fold}_{\sigma_2}(f) = \text{fold}_{\sigma_0}(f) + 2
\]

where \( \sigma_0 \) is the heap at the start, and \( \sigma_2 \) is the heap after the reassignments. Assume that the operator is addition and the mapping is the identity function.

Employing the same strategy, we could split the filter \( f \) by removing each of \( i \) and \( j \).

\[
\begin{align*}
f &= (f - \{i\}) \cup \{i\} \\
f &= (f - \{j\}) \cup \{j\}
\end{align*}
\]

However, this is insufficient, because neither

\[
\text{snap}_{f-\{i\}}^{r.F}(\sigma_0) = \text{snap}_{f-\{i\}}^{r.F}(\sigma_2)
\]

nor

\[
\text{snap}_{f-\{j\}}^{r.F}(\sigma_0) = \text{snap}_{f-\{j\}}^{r.F}(\sigma_2).
\]

The new filters do not create heap portions that are equal. We must extract both \( i \) and \( j \) from the filter. So, a naive approach is to allow repeated
decomposition of filters that were themselves generated from a decomposition.

Repeated decomposition is very costly because the number of filter decompositions to consider grows exponentially. If we take the generated filters \((f - \{i\}), \{i\}, (f - \{j\}), \{j\}\) and decompose them again, we get eight new filters.

\[
\begin{align*}
(f - \{i\}) &= (f - \{i\} - \{i\}) \cup \{i\} \\
(f - \{i\}) &= (f - \{i\} - \{j\}) \cup \{j\} \\
(f - \{j\}) &= (f - \{j\} - \{j\}) \cup \{\} \\
(f - \{j\}) &= (f - \{j\} - \{i\}) \cup \{i\} \\
(\{i\}) &= (\{i\} - \{i\}) \cup \{i\} \\
(\{i\}) &= (\{i\} - \{j\}) \cup \{j\} \\
(\{j\}) &= (\{j\} - \{j\}) \cup \{j\} \\
(\{j\}) &= (\{j\} - \{i\}) \cup \{i\}
\end{align*}
\]

Some of the above equations may be ill-defined. Recall that to decompose filter \(f\) correctly, we need \(f_1\) such that \(f_1 \subseteq f\). Then we can write \(f = (f - f_1) \cup f_1\). Sometimes, it is unclear whether \(f_1\) is a subset, so in practice, the SMT solver will case-split and try both options. One case might assume \(f_1 \not\subseteq f\). In such cases, the ill-defined decomposition equations are not generated. For example, \(i\) and \(j\) may or may not be the same integer. So whether the subset property \(\{j\} \subseteq f - \{i\}\) holds is unclear. Nevertheless, these decompositions may happen in one case, so we write them explicitly above.

Some of the equations are well-defined but do not introduce any new information. For example, we can see \((f - \{i\} - \{i\})\) should be equal to \((f - \{i\})\), because \(i\) has already been subtracted from \(f\). However, the SMT solver still performs a case split here. The case assuming \(\{i\} \subseteq f - \{i\} - \{i\}\) will lead to a contradiction eventually, but this decomposition is still temporarily considered.

We can repeatedly decompose further until we have 16 new filters in
Recall that for each of these filters, there are still two folds, each with heap $\sigma_0$ and $\sigma_2$. So, there are at least 32 new folds created from the decomposition.

This is a very large number, and we already removed some of the filters. For example, $f - \{i\} - \{i\} - \{i\}$ will not be a generated filter because we assume the SMT solver could prove $f - \{i\} - \{i\} = f - \{i\}$, discarding potential decompositions from that filter.

In practice, the SMT solver case-splitting makes this filter decomposition performance even worse. As we have explained briefly, the SMT solver has to check if $f_1 \subseteq f$ before producing $f = f_1 \cup (f - f_1)$. This creates a case split in the proof, and each case may have its own filter decompositions.
For simplicity, we can assume the number of filters doubles at each branch. Nested case splits can also happen, so the total number of folds could increase by a factor of $2^n$ for $n$ case splits.

The number of case splits will also increase by the number of reassignments. At least, there must be one case split on the equality between each index of the reassignment. For our example, we have two indices, $i$ and $j$. The SMT solver needs to check $i = j$, giving one case split. If we have three indices $i, j, k$, the solver checks $i = j$, $i = k$, and $k = j$, giving three case splits. So, the number of case splits is equal to $m \choose 2$, where $m$ is the number of indices in the reassignments.

Though we have not formally proven the total number of filters, it becomes very large in practice. This example illustrates the problem with allowing repeated decomposition. We have tried running examples with three reassignments with the operator $\Rightarrow$ using this naive strategy; Viper fails its verification, timing out at 100 seconds. Given three reassignments already timeout the verifier, it’s clear that we will need to consider alternative approaches.

### 3.2.2 Possible Optimization

We first tried adding optimizations to Viper’s built-in set axioms. For example, we can observe that for any set $f$, $f - \{i\} - \{i\} = f - \{i\}$, or that $f - \{j\} - \{i\} = f - \{i\} - \{j\}$ (right commutativity of set minus). After we add these as axioms, Viper can verify folds inside methods with 3 reassignments but still timeouts again at 4.

### 3.3 Local and Intermediate Decomposition

In this subsection, we present our solution to the combinatorial explosion using local and intermediate decompositions, allowing modular handling of each heap modification. There were two main causes of the explosion.

1. Chained decomposition: a filter generated from a decomposition gets decomposed further.
2. SMT solver case splitting: the SMT solver case splits on the pairwise
equality of each index getting reassigned. Each case split doubles the
number of filters.

Once again, we consider the example with two reassignments, where the
fold operator is addition, and the mapping is the identity function.

```
{ ...
  \sigma_0
  r(i).F := r(i).F + 1
  r(j).F := r(j).F + 1
  ...
  \sigma_2
}
```

We want to prove that

\[
\text{fold}_{\sigma_2}(f) = \text{fold}_{\sigma_0}(f) + 2.
\]

The key observation is that this is simply two copies of the example with one
reassignment appended together. So, let us declare an intermediate heap \(\sigma_1\),
representing the heap state after the first assignment \(r(i).F := r(i).F + 1\).
Then, the problem can be split up into proving

\[
\text{fold}_{\sigma_1}(f) = \text{fold}_{\sigma_0}(f) + 1
\]

and

\[
\text{fold}_{\sigma_2}(f) = \text{fold}_{\sigma_1}(f) + 1.
\]

For the first equation, we need to perform only one decomposition of the
filter \(f\), i.e., \(f = (f - \{i\}) \cup \{i\}\) which gives one decomposition for each of \(\sigma_1\)
and \(\sigma_0\). For the second equation, we again perform only one decomposition
of the filter \(f\), i.e., \(f = (f - \{j\}) \cup \{j\}\), which gives one decomposition for
each of \(\sigma_2\) and \(\sigma_1\). Note that the decomposition removing \(\{j\}\) does not
happen with respect to \(\text{fold}_{\sigma_0}(f)\), and the decomposition removing \(\{i\}\) does
not happen with respect to \(\text{fold}_{\sigma_2}(f)\).

We call this strategy the *local and intermediate decomposition*. First,
after each heap reassignment to some index \(i\), we introduce an intermediate
heap $\sigma_j$ and a heap-dependent fold with argument $\sigma_j$. So, at each atomic heap modification, we have $\sigma_i$ and $\sigma_j$ representing the heap before and after the modification. Then, we locally decompose the two folds by subtracting $i$ from filter $f$, giving

$$\text{fold}_{\sigma_i}(f) = \text{fold}'_{\sigma_i}(f - \{i\}) \oplus \text{fold}'_{\sigma_i}(\{i\})$$

and the same for $\sigma_j$. Finally, for each reassignment, we can deduce an intermediate equality, such as

$$\text{fold}_{\sigma_j}(f) = \text{fold}_{\sigma_i}(f) + 1$$

in our example. These intermediate equalities could chain together to prove properties comparing folds after arbitrarily many reassignments.

There is a reason why we describe the strategy as intermediate and local decompositions.

1. Intermediate: we introduce intermediate heaps in addition to the heaps we initially wrote.

2. Local: the decomposition applies only to the folds with heap arguments representing the state prior and after the assignment, i.e., only two states.

The folds generated from a decomposition should not be further decomposed for performance reasons explained previously. As such, we distinguish between a primary and a secondary fold. In our notation, a fold' (read fold prime), denotes a secondary fold; the fold without the prime is primary. A primary fold is a fold that we allow to be decomposed; these folds are generally written explicitly by the user or generated as an intermediate fold by our strategy. A secondary fold is a fold generated by a decomposition, so the user will not directly mention it. Secondary folds do not get decomposed, while primary folds do. Both folds have the same evaluation meaning. This greatly lowers the performance cost of our implementation. In our example, fold$_{\sigma_i}(f)$ is primary, and the generated fold fold'$_{\sigma_i}(f - \{i\})$ is secondary.
From here onwards, when talking about decomposing a fold, we implicitly mean these primary folds.

### 3.3.1 Modular Strategy: Nodes and Edges

Intermediate and local decompositions allow each heap modification to be handled modularly. Only the heaps directly before and after the modification, i.e., \( \sigma_i \) and \( \sigma_j \), are relevant at each step.

With this modular strategy, the number of decompositions grows linearly with the number of heap assignments, which is a huge improvement. We no longer need to repeatedly decompose a filter generated from a decomposition, and there are no case splits on the equality of indices. Indices are instead handled separately and locally for each reassignment.

However, there are some requirements to make this work. Think of each primary fold as trickling down through each node in the method, where each node is a heap-modifying statement. So, to relate the (primary) fold at the start of the method \( \sigma_a \) to some fold at the end \( \sigma_z \), there must be a path linking \( \sigma_a \) to \( \sigma_z \). Each node is responsible for generating its own local path.

For example, we can relate \( \sigma_0 \) to \( \sigma_2 \) in the earlier example because at each reassignment, we related \( \sigma_0 \) to \( \sigma_1 \) and \( \sigma_1 \) to \( \sigma_2 \), so there is a path between the start and end. Relating \( \sigma_0 \) to \( \sigma_1 \) amounts to decomposing each fold by removing index \( i \) for their respective filters, which allows deducing \( \sigma_1 = \sigma_0 + 1 \). Likewise, for the next reassignment, we decompose each fold \( \sigma_1 \) and \( \sigma_2 \) by removing index \( j \) from their respective filters, which allows deducing \( \sigma_2 = \sigma_1 + 1 \).

So, the rules of this modular strategy can be thought of more generally as follows:

1. Each heap-modifying operation is a node
2. Each (primary) fold in the heap state before the heap-modifying operation is a directed edge into the node, a *fold input*. For example, if a statement \( A \) mutates the heap from \( \sigma_i \) to \( \sigma_j \), then \( \sigma_i \) is an edge into the node.
3. Each heap-modifying operation generates a new intermediate fold, considered a directed edge out of the node, or a fold output. This fold is also considered primary. An exception is if the user writes the fold directly at the heap state $\sigma_j$. Then, this is not an out-edge, as it is not an intermediate fold generated by our strategy.

4. The folds in and folds out are both decomposed with some node-specific rule. This allows us to establish some relation between the folds in and out. The decomposed folds with smaller filters are secondary folds, written as fold'. Secondary folds do not get decomposed further.

We will employ this strategy for the rest of the thesis. Next, we extend it further to handle exhale and inhale, the two other heap-modifying operations. The challenge will be defining exactly what goes in and out of these two nodes.

So far, the strategy for each heap-modifying node are:

1. Reassignment $r(i).F := \ldots$, mutating heap from $\sigma_i$ to $\sigma_j$: If fold input is fold$_{\sigma_i}(f)$, then fold output is fold$_{\sigma_j}(f)$. Decompose by removing $i$ from both folds.

2. Exhale: $\ldots$

3. Inhale: $\ldots$

We will fill in the blanks in the following sections.

3.3.2 Example Combining Exhales and Inhales

Before we explain the handling of inhale and exhale statements, consider the following example. Suppose we have a filter $f$ of type $\text{Set[Int]}$ and a subset $f_1 \subset f$. Suppose also we have an integer $a$ in $f$ but is not part of $f_1$, i.e., $a \in f \land a \not\in f_1$. Then, consider the next lines of code.
method exhale_reassign_inhale() {
  ...
  // Exhale, changes heap state from \(\sigma_0\) to \(\sigma_1\)
  exhale forall i: Int :: i in f1 => acc(r(i).val)
  // Reassignment, changes heap state from \(\sigma_1\) to \(\sigma_2\)
  r(a).val := r(a).val + 1
  // Inhale and assume nothing changed
  // , changes heap state from \(\sigma_2\) to \(\sigma_3\)
  inhale forall i: Int :: i in f1 =>
      acc(r(i).val) && r(i).val == old(r(i).val)
  // Pseudocode asserting change in the fold
  assert fold[sum](f) = old(fold[sum](f)) + 1
}

The assertion at the end aims to prove

\[
fold_{\sigma_3}(f) = fold_{\sigma_0}(f) + 1.
\]

Note how the users only explicitly mention heap states \(\sigma_0\) and \(\sigma_3\), i.e., the heap states at the start and end. All other heap states are considered intermediate.

Following our local and intermediate decomposition plan, we need a fold input in the state right before each heap modifying operation. So, for the reassignment, we need a fold input \(fold_{\sigma_1}(f)\) to generate fold output \(fold_{\sigma_2}(f)\). The input fold should be generated as an output fold of the exhale statement. However, things get tricky because \(fold_{\sigma_1}(f)\) might not even be well-defined with regard to permissions.

### 3.4 Exhale Case

The second way that the heap could be modified is through an exhale statement. An exhale statement containing an accessibility predicate \(acc(A)\) would result in losing the permissions to \(A\). In addition, an exhale statement could also contain quantified permissions, which would mean losing an
unbounded number of permissions to the locations under the quantifier.

Consider the statement

```java
{  // σ₀
    exhale P
    // σ₁
}
```

where P may contain accessibility predicates and quantified permissions. Let the heap immediately prior to this statement be σ₀ and the heap immediately after be σ₁. Suppose, we are interested in the folds fold_{σ₀}[o](m(r.F) | f) with the addition operator o and the identity mapping m. After the exhale, we want to know how the value of the fold relates to fold_{σ₁}[o](m(r.F) | f).

### 3.4.1 Modular Design to Handle Exhales

Before we can even discuss these folds, there is a potential well-definedness problem. Even if fold_{σ₀}[o](m(r.F) | f) is well-defined, the second fold might not be. σ₁ may not have all the permissions to all the heap locations defined by filter f, receiver r, and field F, because the exhale might have given permissions away. For example, suppose the statement is `exhale acc(r(i).val)` for some i ∈ f. Then, by the semantics of exhales, the heap σ₁ does not contain permissions to key (r(i), val), and fold_{σ₁}[o](m(r.F) | f) would be ill-defined!

Remember that we still want to consider each heap modification modularly. As discussed in the previous section, we consider a heap-modifying line of code as a node, and for every fold coming in, there must be a fold going out. Additionally, the node should perform a decomposition on each fold, both in and out, to establish a relationship between them. Altogether, this allows us to link two folds from the start of the method to the end. So, if fold_{σ₀}[o](m(r.F) | f) enters the `exhale` node, what should come out? And what decompositions does the node apply to the folds?

To answer the first question, we need to define a new f′ such that fold_{σ₁}[f′] is well-defined. We can do that by removing all the heap locations without permission as a result of the exhale.
Defining a Well-Defined Output Fold

Let $\sigma_0.\text{keys}(F)$ represent the set of $\text{Ref}$ keys in heap $\sigma_0$ with some field $F$. Recall that we represent the heap as a partial map typed $(\text{Ref}, \text{Field A}) \rightarrow A$, which has tuple keys of the form $(\text{Ref}, \text{Field A})$. So, think of this $\text{keys}(F)$ function as extracting the tuple keys from the heap, filtering the tuple such that the second element is equal to $F$, and outputting the first element typed $\text{Ref}$ of the remaining tuples. Recall that each key on $\sigma_0$ exists if and only if the method has permissions to that heap location at the program point.

Then, we define

**Definition 2** (Lost References).

$$L := \sigma_0.\text{keys}(F) - \sigma_1.\text{keys}(F),$$

i.e., let $L$ be the set of references that the exhale gave away permissions to, with respect to field $F$.

Then, we define the filter of remaining elements $f'$ as

**Definition 3** (Filter Not Lost).

$$f' := \{ e \in f | r(e) \not\in L \},$$

where $r$ is the receiver function.

By definition, $f'$ guarantees that $\text{fold}_{\sigma_1}(f')$ is well-defined with regard to permissions.

Decomposition of In and Out Folds

Next, we decide the decomposition of the folds in and out of the exhale node. For the fold input, i.e., $\text{fold}_{\sigma_0}(f)$, there is one obvious contender. We can decompose $f$ using $f'$! By definition, $f'$ is a subset of $f$, which gives us the decomposition $f = f' \uplus (f - f')$. Applying this to the fold evaluation, we get the fold decomposition

$$\text{fold}_{\sigma_0}(f) = \text{fold}_{\sigma_0}'(f') \oplus \text{fold}_{\sigma_0}'(f - f').$$
Keep in mind that the folds on the RHS of that equation are secondary folds, so they do not get decomposed any further and are not considered as a directed edge for future heap-modifying nodes.

For the fold output fold$_{\sigma_1}(f')$, there is no decomposition needed. Assuming we model folds as a heap-dependent notion that Viper can handle (see next chapter), by permission framing, we know that fold$_{\sigma_1}(f') = \text{fold}_0^{'}(f')$ because the method holds permissions to all elements defined by $f'$ across the \textbf{exhale} statement, and the exhale will not have modified any of those elements unlike the explicit reassignment $:=$. However, by explicitly declaring fold$_{\sigma_1}(f')$ as a fold output, we have elevated it from being a secondary fold in fold$_0^{'}(f')$ to a primary fold fold$_{\sigma_1}(f')$. That means the filter $f'$ could become a candidate for decomposition in future heaps.

So, we update the strategy for heap-modifying nodes.

1. Reassignment $r(i).F := ...$, mutating heap from $\sigma_i$ to $\sigma_j$: If fold input is fold$_{\sigma_i}(f)$, then fold output is fold$_{\sigma_j}(f)$. Decompose by removing $i$ from both folds.

2. Exhale, mutating heap from $\sigma_i$ to $\sigma_j$ via permission loss. If fold input is fold$_{\sigma_i}(f)$, then fold output is fold$_{\sigma_j}(f')$ where $f'$ is the subset of $f$ where permissions were not lost. Decompose by subtracting $f'$ from $f$ for the fold inputs. Fold outputs do not need a decomposition.

3. Inhale: ...

This strategy creates the flow of folds as part of the modular strategy design. If there is a heap assignment after an exhale statement, the exhale strategy will output a fold as input to the corresponding node. For example,

```plaintext
1 exhale acc(r(i).val)
2 r(j).val := r(j).val + 5
```

This code snippet has two heap-modifying nodes. If the fold input to the first \textbf{exhale} node is fold$_{\sigma_0}(f)$, then the fold output to the same node is fold$_{\sigma_1}(f')$, which itself is a fold input to the $:=$ node.

In addition to continuing the flow of folds, this exhale strategy can also prove some interesting examples in isolation. Suppose we have an arbitrary

44
subset \( f'' \subseteq f \). Then, suppose the user assumes, through some assume statement or requires clause, \( \text{fold}_{\sigma_0}[\text{sum}](f'') = 7 \) and \( \text{fold}_{\sigma_0}[\text{sum}](f) = 10 \). Then, there is the statement \text{exhale}\forall i: A :: i \in f'' \Rightarrow \text{acc}(r(i).F)\), which translates to exhaling all elements of filter \( f'' \).

The user now wants to prove \( \text{fold}_{\sigma_1}[\text{sum}](f - f'') = 3 \) in an assert statement. This works because, from our strategy, we will have generated the equations

\[
\text{fold}_{\sigma_0}[\text{sum}](f) = \text{fold}'_{\sigma_0}[\text{sum}](f') + \text{fold}'_{\sigma_0}[\text{sum}](f - f'),
\]

and

\[
\text{fold}_{\sigma_1}[\text{sum}](f') = \text{fold}'_{\sigma_0}[\text{sum}](f').
\]

We also know that \( f' = f - f'' \) and \( f - f' = f'' \). From equational reasoning, we deduce

\[
\text{fold}_{\sigma_0}[\text{sum}](f) = \text{fold}'_{\sigma_0}[\text{sum}](f - f'') + \text{fold}'_{\sigma_0}[\text{sum}](f'')
\]

\[
10 = \text{fold}'_{\sigma_0}[\text{sum}](f - f'') + 7
\]

\[
\text{fold}'_{\sigma_0}[\text{sum}](f - f'') = \text{fold}_{\sigma_1}[\text{sum}](f - f'') = 3.
\]

In practice, however, the equalities \( f' = f - f'' \) and \( f - f' = f'' \) are not so obvious to the SMT solver, and they require some explicit extensional equality checks. We discuss this more in the next chapter.

The exhale strategy allows for reasoning through its decomposition, while also generating intermediate folds as inputs for further heap mutation nodes.

### 3.5 Inhale Case

Now, we consider the last scenario of heap mutation, the inhale statement, which may add permissions and new keys to the heap \( \sigma \).

Intuitively, an inhale is the opposite of an exhale. In our exhale strategy, we started with input \( \text{fold}_{\sigma_{in}}(f) \), lost some permissions, constructed a new filter \( f' \subseteq f \) where permissions were retained, decomposed the initial fold by subtracting the new filter \( f' \), and finally, generated an intermediate fold out
fold$\sigma_{out}(f)$.

In an inhale strategy, we could imagine simply doing it reversed. We could start with some input fold$\sigma_{in}(f)$, gain some permissions, find a larger filter $f^* \supseteq f$ where $f^*$ includes items from the gained permissions, generate an output fold$\sigma_{out}(f^*)$, and then decompose this new output fold by subtracting the input filter $f$.

There are two main questions regarding this.

1. How do we know what $f^*$ to construct?

2. Is this enough in practice?

### 3.5.1 Finding the Output Fold Filter

Picking which $f^*$ to construct from a smaller filter $f$ is not so obvious. One viable idea is to construct the set of all heap locations gained from the inhale, extract a filter set out of that, and union it to $f$.

However, this idea is problematic when the amount of permissions gained exceeds the filter of interest. For example, consider the following snippet.

```java
method test()
  requires forall i: Int:: i in f => acc(r(i).F)
{
  exhale forall i: Int:: i in f_1 => acc(r(i).F)
  label a // after exhale
  assume e in (f - f1)
  r(e).F := r(e).F + 1
  label b // after reassignment
  inhale forall i: Int:: i in f_1 || i in f_2 => acc(r(i).F)
  inhale forall i: Int:: i in f_1 => r(i).F == old(r(i.F))
  label c // after inhales
  assert fold[sum](f) = old(fold[sum](f) + 1)
}
```

Suppose that $r$ is an injective receiver, and the filters $f$ and $f_1 \subset f$. Also, suppose $f_2$ is disjoint from filter $f$. So, in the code snippet, we start with
permission to elements defined by filter $f$. Then, an exhale happens, removing permissions to filter $f_1$. Then, a reassignment happens to some arbitrary heap location defined by $e \in (f - f_1)$. Then, an inhale happens, where the method gets permissions to elements defined by filters $f_1$ and $f_2$. Another inhale assumes that all the heap elements defined by $f_1$ remain unchanged throughout this whole process (framing axiom). We now want to prove that

$$\text{fold}_{\sigma_c}(f) = \text{fold}_{\sigma_{\text{old}}}(f) + 1$$

If we use the modular strategy, we can think of folds as inputs and outputs to heap modifying nodes. In our example, the first exhale node has input $\text{fold}_{\sigma_{\text{old}}}(f)$, and output $\text{fold}_{\sigma_a}(f')$ where $f' = f - f_1$. Then, the reassignment node has input $\text{fold}_{\sigma_a}(f')$ and output $\text{fold}_{\sigma_b}(f')$. The node that inhales access then has input $\text{fold}_{\sigma_b}(f')$. If we simply take all the permissions and put it in a filter, the inhale node would have output $\text{fold}_{\sigma_c}(f \cup f_2)$.

However, for that final assert, we need some knowledge about $\text{fold}_{\sigma_c}(f)$, but we do not get it with this strategy! Instead, we have $\text{fold}_{\sigma_c}(f \cup f_2)$, which has a bigger filter than we want.

A more useful idea is to construct the filters that were there before the exhale. If we start with a fold on some filter $f$, lose some permissions to it, and then gain permissions back, it is reasonable to try to reconstruct $f$ again in practice. Concretely, if we start from some fold before an exhale, we should aim to construct the same fold after an inhale. To decide the output fold of an inhale node, we look back at the input folds of a previous exhale node. In our case, the exhale node has input $\text{fold}_{\sigma_{\text{old}}}(f)$, with filter $f$. Then, we can output the fold with the same filter $f$ at the inhale node.

To explain the strategy in more detail:

1. Look at the input folds to all previous exhales. Extract the filters.

2. Check that the inhale has retrieved new permissions to the filter for well-definedness.

3. Check that the filter is new, i.e., the method did not have full permissions to the filter before the inhale.
4. For each extracted filter \( f^* \), construct \( \sigma_{\text{new}}(f^*) \) as an output fold to the inhale node.

Then, similar to the exhale strategy, we decompose this output fold \( \sigma_{\text{new}}(f^*) \) by subtracting the filter \( f \), which is part of the input fold. At the exhale node, the input fold \( \sigma_{\text{old}}(f^*) \) should also be decomposed by subtracting the filter \( f \) for symmetry.

Finally, we summarize the strategies for each node in the following definition.

**Definition 4 (Final Strategies).** These are the final strategies for each heap-modifying statement.

1. **Reassignment** \( r(i).F := \ldots \), mutating heap from \( \sigma_i \) to \( \sigma_j \): If input fold is \( \sigma_i(f) \), then output fold is \( \sigma_j(f) \). Decompose by removing \( i \) from both folds.

2. **Exhale**, mutating heap from \( \sigma_i \) to \( \sigma_j \) via permission loss: If input fold is \( \sigma_i(f) \), then output fold is \( \sigma_j(f') \) where \( f' \) is the subset of \( f \) where permissions were not lost. Decompose by subtracting \( f' \) from \( f \) for the fold inputs. Fold outputs do not need a decomposition.

3. **Inhale**, mutating heap from \( \sigma_i \) to \( \sigma_j \) via permission gain: If the input fold is \( \sigma_i(f) \), search the input folds to previous exhales for a filter \( f^* \supset f \). If new heap state has permissions to \( f^* \), construct the output fold \( \sigma_j(f^*) \). Decompose the output fold by subtracting \( f \) from \( f^* \). The input fold to the corresponding exhale node should be retroactively decomposed the same way. Fold inputs to the inhale node do not need a decomposition.

### 3.5.2 Folds Flowing Backwards with Lookahead

There are certain limitations to this inhale strategy. If there are inhale statements without prior exhales, then our strategy does not generate any output fold. We become reliant on the user writing the fold manually.

Consider the following example, with \( f \) and \( f'' \) as disjoint filters.
Here, the inhale node does not generate any new folds because there is no prior exhale statement. At the point of the inhale, no input fold has any information about the filter \( f \) \( \cup \) \( f'' \), so the assert would not verify.

An approach we considered to solve this is to use some lookahead. Our inhale strategy looks back at input folds to previous exhales, so why not look forward at future relevant folds? Perhaps those future folds can be fed as input folds, like data flowing backwards. In the mentioned example, if the inhale node could somehow see

\[
\text{fold}_\sigma(f \cup f'')
\]

at the end of the method, then we can perform the decomposition and generate an output fold in the previous heap state.

If this is possible, we can treat an inhale exactly like an exhale; it just needs to be read backwards. At the line before an exhale statement, the method has permissions and maybe some fold with a filter \( f \). At the line after, the method has fewer permissions and a fold with a smaller filter \( f' \). For inhales, it is the same read backwards. At the line after an inhale, the method has permissions and maybe some fold with a filter \( f \). At the line before, the method has fewer permissions and a fold with a smaller filter \( f' \). If we allow data to flow forward and backward, we can treat inhales and exhales in the same way.

We are unable to do this due to practical limitations. As we will explain in the upcoming chapter, generating intermediate folds from folds in a previous heap state relies crucially on quantifier triggering. Folds in the previous heap state generate folds in the following heap state. If folds in the following heap state also generate folds in the previous heap state, we potentially end up in
a matching loop. New generates old, which generates new, which generates old, etc.

The other practical limitation comes from lookahead triggering. Our experiments show that including state-dependent expressions in quantifier triggers has some undocumented behaviour in Viper, especially with triggers containing terms from a future state, which is not clearly a supported feature. The triggers simply do not work as one would expect, so we could not achieve this look-ahead idea and kept the present design.
Chapter 4

Modelling in Viper

Now that we have explained the strategies for supporting folds formally, in this chapter, we present our encoding of them into Viper, designed to implement our strategies automatically.

4.1 Heap-dependent Fold Definition In Viper

Consider our fold notation as defined in the previous chapter,

\[
\text{fold}_\sigma[o](m(r.F) \mid f),
\]

with filter \(f\), receiver \(r\), field \(F\), and mapping \(m\), where heap \(\sigma\) is the current heap, and operator \(o\) is the operator object containing a binary operation \((B, B) \rightarrow B\) and an identity element \(i_0 : A\) for that operation.

There are several challenges to translating this into Viper. Firstly, Viper does not allow first-class functions. Viper supports first-order logic, so we cannot quantify over functions, and function arguments cannot be functions. The second challenge comes from the limitations in interacting with the heap. Although the heap is part of the Viper program state, the heap is not first class; it cannot be bound to an expression. Like functions, the heap is not a valid function argument and cannot be a quantified variable.
4.1.1 Defunctionalization

To solve the first challenge, we apply defunctionalization to convert each input function to a combination of domain declarations. First, we declare the domain types for each of the function arguments to a fold. We declare \texttt{Receiver[A]}, \texttt{Operator[B]}, and \texttt{Mapping[V, B]} as (polymorphic) domain types with type arguments \(A, V, \) and \(B\). The filters can be represented with Viper's built-in \texttt{Set}, so we do not define new types for them.

For each of these domain types, we define an \texttt{apply} function that takes an instance of the newly defined types as the first argument, followed by the intended arguments. For example, the receiver domain has a function \texttt{recApply(r: Receiver[A], a: A): Ref}. Then, we can declare axioms to define the results of these functions. These axioms depend on the function definitions written by the user; we describe the handling of user declarations in the next chapter. For now, here is the \texttt{Receiver} domain declaration with the appropriate domain functions.

\begin{verbatim}
1 domain Receiver[A] {
2  function recApply(r: Receiver[A], a: A): Ref
3  function recInv(rec: Receiver[A], ref: Ref): A
4 }
\end{verbatim}

The receiver inverse \texttt{recInv} is something we get from the injectivity of the receiver over the filter (discussed later in the well-definedness checks section). A receiver maps an element of a filter to a reference. Sometimes, given an arbitrary reference, we want to extract the original element in the filter. We do so using the \texttt{recInv}. For example, if there is a reassignment \(r(i).\text{val} \ := \ldots\), we need to be able to extract \(i\) with \texttt{recInv(r(i))} to perform the appropriate fold decomposition. We show this explicitly in the listing on page 75.

The \texttt{Operator[B]} domain has an \texttt{opApply} function that takes two additional arguments, each representing an input to a binary operator. In addition, each operator should have an identity value, so we define another function, \texttt{opGetIden(o: Operator[B]): B}, that outputs the specified identity.
domain Operator[B] {
  function opApply(op: Operator[B], val1: B, val2: B): B
  function opGetIden(op: Operator[B]): B
}

Likewise, the Mapping[V,B] domain has a mapApply function for applying the mapping.

domain Mapping[V, B] {
  function mapApply(m: Mapping[V, B], _mInput: V): B
}

This is similar to classes in object-oriented programming languages like Java or Python. Each domain type is a class, and the apply functions are methods that each takes self as the first argument.

4.1.2 Fold Object

Like the other types we have defined, we can also defunctionalize the fold function, representing it as a domain type or a class. From a class point of view, we define fold as a class that takes three constructor arguments: a receiver Receiver[A], a mapping object Mapping[V,B], and an operator Operator[B]. Unfortunately, fold is already a keyword in Viper (for folding predicates), so we use hfold for heap-dependent fold as the name of our constructor instead:

domain Fold[A,V,B] {
  function hfold(r: Receiver[A], m: Mapping[V,B],
         op: Operator[B]): Fold[A,V,B]
}

Listing 4.1: hfold Object Constructor

Given a hfold object, we should be able to extract the components from it, namely the receiver, operator, and mapping. We declare getter functions for them, as shown in the following snippet.
domain Fold[A,V,B] {
    function hfold(r: Receiver[A], m: Mapping[V,B],
        op: Operator[B]): Fold[A,V,B]

    function getreceiver(c: Fold[A, V, B]): Receiver[A]
    function getoperator(c: Fold[A, V, B]): Operator[B]
    function getmapping(c: Fold[A, V, B]): Mapping[V, B]
}

Some components are still missing from this object representation: the filter, the field, and the heap. One can think of a hfold instance as a partially applied fold. The remaining components appear as arguments to another function that generates the snapshot.

4.1.3 Snapshot Introduction

As we explained earlier, a fold evaluation only depends on a portion of the heap defined by the filter, the receiver, and the field. We refer to this portion as a snapshot of the heap, which we have defined as

\[
\text{snap}^r F^f (\sigma) = \{ e \rightarrow \text{value} \mid ([r(e), F] \rightarrow \text{value}) \in \sigma \land e \in f \}.
\]

In our Viper encoding, we modify the values in the snapshot map to include the application of the mapping function. As written earlier, an alternative representation of a fold using a snapshot is

\[
\text{fold}[o]\{m(\text{snap}^r F^f (\sigma))\}.
\]

Notice the \(m\) applied to the snap. In our Viper encoding, we decide to distribute that \(m\) application into the values of the snap map, which we can express as

\[
\text{fold}[o]\{\text{snap}^m (r,F) (\sigma)\}.
\]

Concretely, the snapshot map with the mapping function distributed is

\[
\text{snap}^m (r,F) (\sigma) = \{ e \rightarrow m(\text{value}) \mid ([r(e), F] \rightarrow \text{value}) \in \sigma \land e \in f \}.
\]
Recall that the keys to this snapshot map should be an element of the filter, i.e., they have type \( A \). Applying the receiver \( r \) to this \( A \) should give a \( \text{Ref} \), which would be a key to the heap map. Also, the snapshot function takes a field argument, which limits the keys of the heap map only to the tuples with the specified field.

A first attempt in pseudocode at encoding this snap function might look like the following.

```viper
1 domain Fold[A,V,B] {
2     function hfold(r: Receiver[A], m: Mapping[V,B], op: Operator[B]): Fold[A,V,B]
3
4     function snap(fo: Fold[A,V,B], filter: Set[A],
5         field: V, heap: Map[(Ref, Field V), V]): Map[A,B]
6 }
```

**Listing 4.2: Snapshot Application Pseudocode**

Since the snapshot function depends on the receiver and mapping function, our function takes a \( \text{Fold} \) object that already encapsulates a \( \text{Receiver} \) and a \( \text{Mapping} \).

The snap function above is pseudocode because of a few practical limitations. For one, the field and the heap cannot be included as function arguments. More importantly, the snap function is a heap-dependent function, which cannot be declared inside a Viper domain definition.

Functions defined outside a domain declaration in Viper take the heap as an implicit argument. However, these non-domain functions cannot utilize type arguments, so the argument \( \text{fo}: \text{Fold}[A, V, B] \) would not be valid. So, in practice, we generate actual snap functions in the style of Listing 4.3.
function snap_Int_Int_Int_val(fo: Fold[Int, Int, Int],
  filter: Set[Int]): Map[Int, Int]
  requires forall i: Int :: {i in filter} i in filter =>
    acc(recApply(getreceiver(fo), i).val)
  ensures domain(result) == filter
  ensures (forall i: Int :: { result[i] } (i in filter) =>
    result[i] == (mapApply((getmapping(fo)),
      (recApply((getreceiver(fo)), i)).val)))
Listing 4.3: Example of a Snapshot Function Declaration (Partial)

In this example, the snap function is intended for a Fold[Int, Int,
Int] instance and the val field. The field val is inlined into the precondition,
so it is not an argument. Altogether, this monomorphism means we must
define a separate snap function declaration for each field and each
Fold[A, V, B]. Because Viper does not allow type arguments for heap-dependent
functions, we need to perform monomorphization manually in our plugin; this
amounts to declaring a new function for each instance of the type variables
A, V, B and the relevant field.

Note that the preconditions contain quantified permissions, denoting
access to all indices defined by receiver(i).val, where i is an element of
the filter. To create a snapshot of the heap elements, we need permission
to read the heap elements. Technically, this can be fractional permissions
since we only need read access. Also, using quantified permissions requires
injectivity of the receiver expression with respect to the filter. This is a
well-definedness check that we include as another function precondition. We
discuss this in Section 4.2.

For now, we have omitted other specifications of the snap functions from
this example; there is an injectivity check of the receiver and post-conditions
to aid with fold decomposition, filter equality checks, and connection to a
secondary snap function, snap_prime. We explain these things gradually.

The ensures clause of the snap function defines the output map. The
keyword result in Viper denotes the output of a function, and function
postconditions typically refer to the result extensively. The first post-
condition above ensures that the output’s keys, also called domain, are equal to the filter set. The second post-condition ensures that the output map’s value for key \( i \) is equal to \( \text{mapping}(\text{receiver}(i).\text{val}) \) with the appropriate defunctionalized apply functions.

With this snap function defined, we can declare an apply function for Fold, with inputs of the fold object and the snapshot output.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Fold([A, V, B])</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>function hfold(r: Receiver([A]), m: Mapping([V, B]), op: Operator([B])): Fold([A, V, B])</td>
</tr>
<tr>
<td></td>
<td>function hfoldApply(fo: Fold([A, V, B]), snap: Map([A, B])): B</td>
</tr>
<tr>
<td></td>
<td>function hfoldApply1(fo: Fold([A, V, B]), snap: Map([A, B])): B</td>
</tr>
</tbody>
</table>

**Listing 4.4:** Fold Apply Functions

This mimics the earlier definition,

\[
\text{fold}[\sigma]\{\text{snap}_{m(r.F)}(\sigma)\}.
\]

The \( \text{hfoldApply1} \) is the secondary fold, as denoted with \( \text{fold}' \). Secondary folds do not get decomposed, preventing the combinatorial explosion as discussed in Section 3.2.1. Otherwise, the value of a secondary fold is the same as that of a primary fold.

Notice that despite the snap function being heap-dependent, its map output is considered pure, allowing us to use it as an argument \( \text{snap}: \text{Map}[A, B] \) to a pure domain function \( \text{hfoldApply} \). Fortunately, this means we can use type arguments and declare a polymorphic \( \text{hfoldApply} \). Given the choice, we generally prefer to define functions as pure domain functions instead of heap-dependent ones whenever possible.

### 4.1.4 Encoding FoldApply

Now that we have defined an apply function for fold, we need to write axioms to describe the function’s behaviour.

As described in the previous chapter, the rules for a heap-dependent
fold’s evaluation have 3 different cases. In the first case, when the input filter is empty, the fold should output the identity value as defined by the operator.

```plaintext
axiom _emptyFold {
  (forall c: Fold[A, V, B], snap: Map[A,B] ::
    { (hfoldApply(c, snap)) }
  domain(snap) == Set[A]() ==>
    (hfoldApply(c, snap)) ==
      (opGetIden((getoperator(c))))
}
```

The trigger for this axiom is just an instance of a fold application.

The second case is when the input filter is a singleton set. The output should be the mapping function applied to an element on the heap. In our case, the mapping function is already included in the snapshot map values, so we simply need to read the snapshot at the key equal to the single element in the filter.

```plaintext
axiom _singleton {
  (forall c: Fold[A, V, B], snap: Map[A,B], elem: A ::
    { (hfoldApply(c, snap)), Set(elem) }
  domain(snap) == Set(elem) ==>
    (hfoldApply(c, snap)) == snap[elem])
}
```

The trigger is the fold application and a mention of a singleton set `Set(elem)`. We construct a singleton set during singleton decomposition, so this axiom would instantiate, giving the value of the fold over that set.

The third case is the decomposition case. Recall that we perform two kinds of decompositions in our strategies for handling inhales, exhales, and heap reassignment as described in Definition 4 on page 48. For exhales (and some parts of inhale strategy), we start from a filter \( f \), and then setminus a subset \( f' \). For heap reassignment, we subtract a singleton set from the filter \( f \). We distinguish these two cases into two scenarios: subtracting a singleton filter or subtracting a whole filter.
Singleton Decomposition Axiom

The case for singleton decomposition can be axiomatized by the following domain axiom:

```plaintext
axiom _dropOne {
  forall fo: Fold[A, V, B], snap: Map[A,B], key: A ::
    { triggerDeleteKey(hfoldApply(fo, snap), key) }
    (key in domain(snap)) ==>
    hfoldApply(fo, snap) ==
    opApply(getoperator(fo),
    hfoldApply1(fo, mapDelete(snap, Set(key))),
    snap[key])
}
```

Listing 4.5: Decomposition By Removing 1 Element

In other words, if the element `key` is in the filter, then `hfold(snap) = op(hfold1(snap - key), snap[key])` (omitting the defunctionalization for brevity). There are a few things to observe with this axiom. Firstly, the trigger has a function `triggerDeleteKey(...)`, which we have not discussed. This is simply a dummy Boolean function used to trigger the axiom when necessary. The plugin assumes an instance of this dummy trigger function when we identify that we need to perform a decomposition, as described in the reassignment strategy in Definition 4 on page 48.

The other interesting point is the operations on the snap function. As described earlier, `domain(snap)` refers to the filter. In the previous chapter, a singleton decomposition involved removing a key from the filter, which we encode here as `mapDelete(snap, Set(key))`. Viper’s built-in `Map` type does not have a function that deletes keys, so our plugin generates those manually. For now, think of `mapDelete` as deleting a set of keys from the snapshot, which is equivalent to removing keys from the filters. We discuss further details about the snapshot function later.

This axiom generates only one new secondary fold. The second fold from the decomposition is a fold with a singleton filter, so we inline its evaluation, i.e., `snap[key]` explicitly.
Set Decomposition Axiom

The next type of decomposition involves subtracting a whole filter.

1 axiom loseMany {
2  forall c: Fold[A, V, B], snap1: Map[A,B], keys: Set[A] ::
3  { triggerDeleteBlock((hfoldApply(c, snap1)), keys) }
4  (keys subset domain(snap1)) ==>
5  (hfoldApply(c, snap1)) ==
6  opApply(getoperator(c),
7  hfoldApply1(c, mapDelete(snap1, keys)),
8  hfoldApply1(c, mapSubmap(snap1, keys)))
9 }  

Listing 4.6: Domain Axiom for General Decomposition

The axiom states that if filter keys is a subset of the current filter, i.e., the domain of the snapshot, then the fold can be decomposed as

\[ \text{hfold}(\text{snap}) = \text{op}(\text{hfold1}(\text{snap} - \text{keys}), \text{hfold1}(\text{snap}|\text{keys})) \]

The notation snap|keys refers to a projection of the snapshot map onto the specified keys, i.e., restricting the map onto the filter keys. Viper builtin Map type lacks this operation, so we generate another function, mapSubmap, to model this operation.

Notice again that this axiom relies on another dummy boolean function, triggerDeleteBlock. The plugin generates an instance of the triggering function when implementing the inhale and exhale strategies described in Definition 4 on page 48.

4.1.5 Example: Array Encoding

We present an example encoding an array sum, i.e., its receiver, mapping, operator, etc. Recall that the standard array encoding uses a field and an Array domain. We use an array of integers to declare a field of type Int.
field val: Int
domain Array {
    function loc(a: Array, i: Int): Ref
    function len(a: Array): Int
    axiom len_nonneg {
        forall a: Array :: { len(a) }
        len(a) >= 0
    }
    // functions and axioms to ensure loc() injectivity
    function first(r: Ref): Array
    function second(r: Ref): Int
    axiom all_diff {
        forall a: Array, i: Int :: {loc(a,i)}
        first(loc(a,i)) == a && second(loc(a,i)) == i
    }
}

Listing 4.7: Full Array Encoding

The last all_diff axiom ensures that the loc function has inverses first and second. This also implies that loc, the Array receiver function, is injective.

For an array, the receiver function should be from an integer index (Int) to a Ref. So, we declare a function to generate a Receiver[Int] object, and an axiom describing the recApply function on this object.

function arrayRec(a: Array): Receiver[Int]

axiom {
    (forall a: Array, i: Int ::
        { (recApply(arrayRec(a), i)) }
        { loc(a, i) }
        recApply(arrayRec(a), i) == loc(a, i))
}

The axiom relates the Receiver[Int]'s application to the Array domain's loc function. One thing to note is that the loc function takes two arguments: an array and an index. But a receiver function, by definition, only takes
one input. We circumvent this issue by having \texttt{arrayRec} take an array argument to output a \texttt{Receiver[Int]} instance. That array argument can now be considered like a field in the object and can be used in the axiom. So technically, \texttt{recApply} still only takes one argument in addition to self, i.e., \texttt{arrayRec(a)}.

The triggers for the axioms are chosen to be \texttt{recApply(arrayRec(a), i)} or \texttt{loc(a, i)}. Triggers written in two curly braces are considered disjunctive triggers; only one must match to instantiate the axiom. We added both the LHS and RHS equations to ensure bidirectional triggering. Bidirectional triggering is a heuristic for choosing a trigger by ensuring that a universal quantifier involving an implication or equality can be triggered both by the left or right-hand terms, improving completeness.

Next, we create a mapping function instance, i.e., \texttt{Mapping[Int,Int]}. For an array sum, this should just be the identity function. We can define a general identity mapping function because it is quite common.

```plaintext
function mapIdentity(): Mapping[V, V]

axiom {
  (forall v: V ::
    { (mapApply((mapIdentity()), v))
      (mapApply((mapIdentity()), v)) == v)
  }
}
```

Applying the identity mapping function changes nothing to the input \(v\).

Next, we define the addition operator, \texttt{Operator[Int]}, which should also have 0 as the identity.

```plaintext
function add(): Operator[Int]

axiom {
  (forall a: Int, b: Int :: { opApply(add(), a, b) }
    opApply(add(), a, b) == a + b)
}

axiom {
  opGetIden(add()) == 0
}
```
The first axiom defines the \texttt{apply} function for the \texttt{add()} instance of 
\texttt{Operator[Int]}; since it is a binary operator, the apply function takes two 
additional arguments. Notice that the trigger only contains the LHS of the 
equation, because \( a + b \) is not a valid trigger. SMT solvers typically disallow 
interpreted symbols like plus + or less than \(<\) from triggers, so \( a + b \) would 
not be allowed. For practical use cases, this does not make a difference in 
completeness.

The second axiom defines the identity of addition, 0. The axiom does not 
use a quantifier, but it is possible if the instance \texttt{add} contains any arguments.

Lastly, we define a filter as a set of integer indices. Typically, one would 
specify the array sum over a range of indices starting from \texttt{start} to \texttt{end}, 
exclusive of the end.

```plaintext
function intRange(start: Int, end: Int): Set[Int]

axiom {
  (forall start: Int, end: Int, i: Int ::
    { (i in intRange(start, end)) }
    (i in intRange(start, end)) == (i >= start & i < end))
}
```

Filters can be defined as a set represented by a Boolean expression, like 
\( i \geq start \land i < end \). Any index \( i \) that makes the expression true would 
be considered part of the set. Similar to the operator axiom, the filter axiom 
triggers only on the LHS of the equation because \( \geq, \land, < \) on the RHS are 
all interpreted symbols forbidden in triggers.

Together with the \texttt{snap_Int_Int_Int_val} declaration of the snapshot 
function, we can declare a fold object and the fold application. Suppose we 
want to write the sum of the array from indices 0 up to the array length. Given 
an array \( a \), the fold instance can be instantiated with \texttt{hfold(arrayRec(a), 
mapIdentity(), add())}, and a fold application is

```plaintext
hfoldApply(hfold(arrayRec(a), mapIdentity(), add()),
  snap_Int_Int_Int_val(
    hfold(arrayRec(a), mapIdentity(), add()),
    intRange(0, len(a)))).
```
The fold object \texttt{hfold(arrayRec(a), mapIdentity(), add())} is repeated, but this does not cause any overhead because it is not really an object. No additional memory space is allocated for every application of the \texttt{hfold}. Furthermore, Viper understands congruence, so if two applications of \texttt{hfold} have the same arguments, they are considered equal.

### 4.2 Well-definedness Checks

There are four main well-definedness requirements that need to be checked for soundness.

1. Types have to match for all parameters
2. Operators must be associative, commutative, and have the specified identity
3. Receivers must be injective on the filter
4. The current state must have enough permissions to all heap locations defined by the filter and receiver

Correct typing is ensured by Viper’s backend type-checker. The next chapter discusses how we use the type-checker in our plugin.

#### 4.2.1 Operator associativity, commutativity, and identity

The operator’s well-definedness checks can be made in a separate method. Putting these checks in a method means they are verified only once, avoiding unnecessary repeated checks.

For example, the plugin would generate the following method to check the \texttt{add()} operator. The first assert checks commutativity. The second assert checks associativity. The final assert checks that the identity is actually the identity.
method operator_add_welldef_check()
{
assert (forall _i1: Int, _i2: Int ::
  { notTriggered(opApply((add())), _i1, _i2) }
  (opApply((add())), _i1, _i2) ==
  (opApply((add())), _i2, _i1))
assert (forall _i1: Int, _i2: Int, _i3: Int ::
  { notTriggered(opApply((opApply((add())), _i1, _i2)), _i3) }
  (opApply((opApply((add())), _i1, _i2)), _i3) ==
  (opApply(_i1, (opApply((add())), _i2, _i3)))
assert (forall _i1: Int ::
  { notTriggered(opApply(_i1, (opGetIden((add()))))) }
  (opApply(_i1, (opGetIden((add()))))) == _i1)
}

Although these properties are important for soundness, we do not need them in any of the proofs. They do not need to appear as axioms, so the separate method checks are sufficient. Also, the triggers in the assertions are essentially mute because we do not intend the quantifiers to instantiate anywhere else. In the final implementation, we use the dummy function notTriggered to ensure that the quantifiers do not instantiate.

4.2.2 Injectivity Of Receiver on the Filter

Injectivity of the receiver with respect to the filter is required by quantified permissions. Since we use quantified permissions in the snapshot function specifications, Viper must be able to prove injectivity to avoid well-definedness errors in using the snapshots.

In addition, we also rely on having an inverse of the receiver in some of our strategies. For example, in the inlined axiom for singleton decomposition (explained in the upcoming sections), we use recInv(loc(a,i)) to extract the index i. This inverse would not exist without the injectivity of the receiver.

There are a few challenges and requirements for implementing the injectivity well-definedness checks.
• Injectivity checks should throw an error if it fails.

• Injectivity is required for quantified permissions, so an injectivity axiom must exist after the check. Unlike the techniques for checking operator well-definedness, we cannot have an assertion in a separate method that we discard.

• We should avoid repeated injectivity checks. A receiver and filter pair should only be checked once.

By the first requirement, the injectivity check cannot be done in a domain axiom. If the checks fail, there should be an assertion error, and domain axioms do not generate those errors. So, it must be some kind of assertion or precondition check.

An alternative we previously considered to check injectivity is simply adding `assert` statements prior to the user-mentioned fold expressions. However, this technique would limit the usage of fold expressions outside a method body, where assert statements cannot be added in. For example, if a fold expression appears inside a heap-dependent function body, there is nowhere to add an assert statement because a function body can only have an expression!

**Injectivity Check in Snapshot Precondition**

We chose to encode an injectivity check as an additional precondition to our snapshot function, as shown in the following example:
function snap_Int_Int_Int_val(fo: Fold[Int, Int, Int],
    filter: Set[Int]): Map[Int, Int]
requires forall ind1: Int, ind2: Int :
    { ind1 in filter, ind2 in filter }
    ind1 in filter && ind2 in filter
    && ind1 != ind2 ==>
    recApply((getreceiver(fo)),ind1) !=
    recApply((getreceiver(fo)),ind2)
    // the next precondition has quantified permissions
requires forall i: Int :: {i in filter} i in filter ==>
    acc(recApply(getreceiver(fo), i).val)

The check is done according to the standard definition of injectivity. The receiver r is injective with respect to filter if

\[ \forall i, j \in \text{filter}. i \neq j \implies r(i) \neq r(j). \]

Given two distinct indices in the filter, the receiver applied to each index should point to different references.

Having this as a precondition ensures that we check for injectivity before a snapshot is taken. Should the check fail, there would be a precondition check failure error. Notice that we place this precondition above the quantified permissions precondition; quantified permissions in Viper require injectivity of the receiver, so having the precondition earlier means we prove it first.

**filterReceiverGood Flag**

We also want to save this knowledge of injectivity for later, so we do not need to prove it again. We declare an additional domain function

\[ \text{function filterReceiverGood}(f: \text{Set}[A], r: \text{Receiver}[A]) \]

to achieve this. If an instance of filterReceiverGood(f,r) is true, then receiver r is injective over some filter f. Once we prove injectivity fully with the precondition, the snap function can have a postcondition ensuring filterReceiverGood(fo,r).
If we know \( \text{filterReceiverGood}(f, r) \), we should be able to skip the snapshot precondition check. To encode this skipping, we add a short-circuiting disjunction in the extra precondition with \( \text{filterReceiverGood}(f, r) \) as the first term:

\[
\begin{align*}
\text{function } \text{snap\_Int\_Int\_Int\_val}(fo: \text{Fold}[\text{Int}, \text{Int}, \text{Int}], \\
\quad \text{filter: } \text{Set}[\text{Int}]): \text{Map}[\text{Int},\text{Int}] \\
\quad \ldots. \\
\quad \text{ensures } \text{filterReceiverGood}(\text{filter}, \text{getreceiver}(fo)) \\
\end{align*}
\]

This way, if a filter and receiver pair is already known to be injective through \( \text{filterReceiverGood}(f, r) \), the disjunction evaluates to true without needing to verify the second term.

We can define additional axioms for \( \text{filterReceiverGood} \) in a domain declaration. Specifically, we add to the \text{Receiver} domain the axioms stating that injective receiver and filter pairing have an inverse. The axioms are shown in Listing 4.8.

There are two inverse axioms. The first states that given an element \( a \) in the filter, \( \text{recInv}(r(a)) = a \). The second axiom goes the other way; for some arbitrary reference, \( ref \), given that \( \text{recInv}(ref) \) is in the filter, then \( r(\text{recInv}(ref)) = ref \). The two distinct axioms are chosen for triggering completeness. Applying a \text{recApply} function explicitly would instantiate the first inverse axiom. Typically, in our receiver definitions, a receiver expression like \( \text{loc}(a,i) \) can be used as a trigger to instantiate the corresponding \( \text{recApply}(r,i) \), (see our receiver definition for array sum before). However, there are some receiver expressions that are not valid triggers, such as
the identity receiver from Ref to Ref, so an explicit recApply may not be instantiated as a term. The second axiom ensures that if we apply a recInv function to that Ref, such as in our inlining strategy, we still instantiate a recApply term and gain some knowledge about the inverse.

We also generate other useful axioms that can infer further injectivity properties. Some other properties are

- If receiver \( r \) is injective over a filter \( f \), then it is still injective over filter \( f \ setminus f_1 \) for any filter \( f_1 \). This is particularly useful when we decompose folds by taking subsets of the filter.

- If receiver \( r \) is injective over a filter \( f_1 \ union f_2 \), then it is still injective over filter \( f_1 \) and over filter \( f_2 \) separately.

Current state must have enough permissions to all heap locations defined by the filter and receiver

The well-definedness condition of having permissions to all heap locations is checked automatically in the snapshot precondition. Recall that the snapshot precondition uses quantified permissions to check access to all heap locations defined by the receiver, filter, and field. This check is done at the hfold call site.
4.3 Encoding the Modular Strategies

Suppose the user writes an expression with a fold application denoting an array sum over all elements:

\[
\text{hfoldApply}(\text{hfold}(\text{arrayRec}(a), \text{mapIdentity}(), \text{add}()), \\
\quad \text{snap}_{\text{Int}_{\text{Int}_{\text{Int}_{\text{val}}}}}( \\
\quad \quad \text{hfold}(\text{arrayRec}(a), \text{mapIdentity}(), \text{add}()), \\
\quad \quad \text{intRange}(0, \text{len}(a))).)
\]

We disregard the user-facing syntax for now, but the user would essentially write something similar that gets compiled down to that expression. For brevity, we write \(\text{fold}(\text{snap}(\text{intRange}(0, \text{len}(a))))\) as a short notation for the aforementioned fold application. This is analogous to the previous formal notation \(\text{fold}_\sigma(\text{filter})\), with \(\text{snap}\) representing the snapshot of heap \(\sigma\). We omit the receiver, mapping, and operator unless they are directly relevant.

Recall that our strategies for reasoning about heap-dependent folds rely on the appropriate response to each heap-modifying operation. The final strategies for heap-modifying nodes are in the definition on page 48. We go through each one of these and discuss the axioms needed to encode these strategies. As we proceed, we also discuss the trigger choices for each of the axioms.

4.3.1 Encoding the Reassignment Strategy

Considering the following program where two elements of the array get incremented by 1 each, we want to prove that the fold representing the sum of the array has increased by 2.
method arrayReassignment(a: Array, i: Int, j: Int)

requires // access to the entire array
requires len(a) > 0
requires i >= 0 && i < len(a)
requires j >= 0 && j < len(a)

ensures // access to the entire array

{
loc(a,i).val := loc(a,i).val + 1
label afterI

loc(a,j).val := loc(a,j).val + 1
label afterJ

assert fold(snap(intRange(0, len(a)))) ==
old(fold(snap(intRange(0, len(a)))) + 2
}

We omit some of the quantified permissions for brevity; the labels are there to aid the explanation.

Following the input and output fold model we previously discussed, an input fold to the first reassignment node is old(fold(snap(intRange(0, len(a))))). The output fold should then be the same, but in a different heap state, i.e., old[afterI](fold(snap(intRange(0, len(a))))), Recall that the old keyword can take an optional label denoting the specific heap state. This generated output fold is considered an intermediate fold because the user only explicitly writes the fold expression in the heap state at the start and end of the method; now we have generated a fold in the state afterI.

The second thing to do for a reassignment node is to perform a decomposition on both input and output folds by removing the index reassigned. In this case, it is index i, or in practice recInv(loc(a, i)). Recall we had a domain axiom, especially for such a decomposition at Listing 4.5. That axiom has the trigger triggerDeleteKey((hfoldApply(c, snap)), key) where the first argument is the fold application and the second argument is the key to delete, i in our example.

The decomposition generated would look like the following.
assume hfoldApply(fo, snap) ==
   opApply(getoperator(fo),
      hfoldApply1(fo, mapDelete(snap, Set(key))),
      snap[key])

Listing 4.9: Singleton Decomposition Example

This is technically not a generated assume statement but an assumption to be instantiated by an axiom (we show this in the next subsection). Previously, we explained that mapDelete deletes a set of keys from a map. In practice, this is implemented as a combination of a domain function with no axiom and a postcondition of the snapshot, as shown in Listing 4.10.

function mapDelete(m: Map[A,B], e: Set[A]): Map[A,B]
  ...
function snap_Int_Int_Int_val(fo: Fold[Int, Int, Int], filter: Set[Int]): Map[Int,Int]
  ...
  ensures (forall s: Set[Int] ::
    { (mapDelete(result, s): Map[Int,Int]) }
    s != Set[Int]() =>
    snap_Int_Int_Int_val_prime(fo, (filter setminus s)) ==
    (mapDelete(result, s): Map[Int,Int]))

Listing 4.10: Encoding of mapDelete

Inside the snapshot function postcondition, we have an axiom which triggers an application of mapDelete on the function output and some set s. This expresses that if the set is not empty, instantiate another snapshot with the same arguments except with filter setminus s. Since the filter input to the snap function determines the keys available in the snapshot map, this essentially removes all keys that are part of s.

Another thing to note is that the precondition generates a snap_Int_Int_Int_val_prime, the primed version of the original snap function. We did this to prevent Viper’s termination plugin from complaining and to stay consistent with the secondary folds. The folds generated in a decomposition are secondary folds annotated with a prime symbol or 1 in its Viper name. Similarly, we consider snapshots created from a decomposition
to be prime versions. The full specifications of the prime snapshots are as follows.

```plaintext
function snap_Int_Int_Int_val_prime(fo: Fold[Int, Int, Int], f: Set[Int]): Map[Int,Int]
requires (filterReceiverGood(f, (getreceiver(fo))))
requires (forall ind: Int ::
{ (ind in f) }
 (ind in f) =>
acc((recApply((getreceiver(fo)), ind)).val))
ensures (filterReceiverGood(f, (getreceiver(fo))))
```

For snapshot primes, we remove most of the pre/post conditions of ordinary snapshots, leaving only the injectivity check via filterReceiverGood and the quantified permissions. The permissions used in the snap functions are important for Viper’s function framing axioms, which we use to prove the equality of folds across states. We discuss the relevance of function framing further in the upcoming paragraphs.

Technically, we could axiomatize mapDelete using domain axioms to handle general maps, but this caused poor performance in practice. Ultimately, we only use mapDelete with snapshots, so we do not need to generalize it.

Going back to Listing 4.9, applying the decomposition to the input fold `old(fold(snap(intRange(0, len(a))))), we get

```plaintext
old(fold(snap(intRange(0, len(a)))) ==
old(fold'(snap(intRange(0, len(a)) setminus i)))) +
old(snap[i]).
```

Doing the same to the output fold `old[afterI](fold(snap(intRange(0, len(a))))), we get

```plaintext
old[afterI](fold(snap(intRange(0, len(a)))) ==
old[afterI](fold'(snap(intRange(0, len(a)) setminus i)))) +
old[afterI](snap[i]).
```

Consider the second term of the decomposition for both the input and output folds. From the semantics of reassignment, Viper knows `old[afterI](snap[i]) = old(snap[i]) + 1. Then, to show that the fold has increased by 1, we need to prove that the first terms of the decomposition for both input
and output folds are equal; Viper derives this automatically from function framing.

Recall that the snap function requires permissions to all heap elements defined by the filter, receiver, and field. If at two program points, those heap elements stay the same, then the function output is the same. In our example method, we assume that the method holds permissions to the entire array, so between the start of the method and label afterI, no other entity could have modified the array. Altogether, Viper can derive

1. \( \text{snap}(\text{intRange}(0, \text{len}(a)) \setminus i) == \text{old[afterI]}(\text{snap}(\text{intRange}(0, \text{len}(a)) \setminus i)) \)

because only the heap location at index \( i \) has been changed, and this snap instance does not need permissions to it. Keep in mind that the \textbf{old} keyword distributes inwards to the heap-dependent components in an expression. Because all its arguments are provably equal,

1. \( \text{old}(\text{fold}(\text{snap}(\text{intRange}(0, \text{len}(a)) \setminus i))) == \text{old[afterI]}(\text{fold}(\text{snap}(\text{intRange}(0, \text{len}(a)) \setminus i))) \)

We can repeat the same exact steps for the reassignment to index \( j \), generate the decompositions, and deduce that the fold has increased by 1 again after the second reassignment. The final assertion would verify, as the fold has increased by 2.

4.3.2 Inlined Axiom for Singleton Decomposition

To implement this strategy, we inline an assume statement with an axiom immediately after each relevant reassignment. For reassignments, the axiom is as follows.
label previousLabel
loc(a,i).val := loc(a,i).val + 1
assume forall fo: Fold[Int, Int, Int], f: Set[Int] ::
{old[previousLabel]((hfoldApply(fo,
    snap_Int_Int_Int_val(fo, f))))}
filterReceiverGood(f, (getreceiver(fo))) &&
(forall __ind: Int ::
{ (__ind in f) }
(__ind in f) =>
perm((recApply((getreceiver(fo)), __ind)).val) ==
write) =>
triggerDeleteKey(
    hfoldApply(fo, snap_Int_Int_Int_val(fo, f)),
    (recInv((getreceiver(fo)), loc(a,i))))
&&
triggerDeleteKey(old[previousLabel](
    (hfoldApply(fo, snap_Int_Int_Int_val(fo, f)))),
    (recInv((getreceiver(fo)), loc(a,i))))

Listing 4.11: Singleton Decomposition Axiom

The axiom quantifies on fold objects and filters. It triggers on an instance of a fold in the state directly before the reassignment, in our case. Notice that the axiom repeatedly contains the snapshot function application snap_Int_Int_Int_val(fo, f) in the body and the trigger. One might consider replacing it everywhere with a quantified variable m: Map[Int, Int] instead. However, this would not work with old expressions the way we intended. The old[previousLabel] expression in the trigger annotation means the axiom should instantiate only on snapshots in the previousLabel state. However, if we replace the snapshot function in this axiom with a quantified variable m, the trigger old[previousLabel](m) does not instantiate the axiom on a map in the state of the previous label. It would instantiate for all maps in all states! Hence, we kept the snap function in the axiom to achieve local triggering, i.e., this axiom would only instantiate for folds exactly in the state previousLabel. This is analogous to the local decomposition idea, which limits decomposition to fold in a specific state, local to the relevant heap-modifying node.
For well-definedness, the LHS of the axiom requires injectivity of the receiver over the filter $f$ and the correct permissions. Then, the RHS of the axiom simultaneously generates two fold applications and decompositions. The first $\text{hfoldApply}(\text{fo}, \text{snap}_{\text{Int}_{\text{Int}_{\text{Int}}}}(\text{fo}, f))$ is the output intermediate fold, and the second is the input fold. Both of them get decomposed via the $\text{triggerDeleteKey}$ function, and the appropriate key is obtained from the receiver inverse function $\text{recInv}$.

An axiom should be generated for each combination of type variables in $\text{Fold}[A,V,B]$ for existing fold instances. In summary, the generated inlined axiom implements our designed strategy by doing the following:

1. The trigger $\{\text{old}[\text{previousLabel}](\text{fold}(\ldots))\}$ encodes the concept of input folds to the reassignment node. An input fold is a fold in the state immediately before the reassignment. Every such fold would match the trigger, instantiating the axiom.

2. The LHS encodes the well-definedness checks on the input fold.

3. The first term on the RHS generates a $\text{fold}(\ldots)$ without the $\text{old}$ expression, denoting an output fold in the state after the reassignment.

4. On the RHS of the axiom, the dummy function $\text{triggerDeleteKey}$ is applied to both the input and output folds. This function instantiates an axiom, shown in Listing 4.5, that performs a decomposition of the fold if the reassigned heap location belongs in the fold. No decomposition happens if the designated key is irrelevant to the fold.

### 4.3.3 Encoding the Exhale Strategy

Encoding the exhale strategy can be done in almost the same manner as the singleton strategy. If there is an exhale statement in the method that causes it to lose permissions, then we have to consider the input and output folds.

An output fold to an exhale node, as described in the definition on page 48, is $\text{fold}_{\sigma_j}(f')$ where $f'$ is the subset of $f$ where permissions were not lost. We previously described constructing a set of references that are lost in
Definition 2 of page 43 and then using that to construct the subset of $f$ with permissions not lost in Definition 3. Let us encode these definitions in Viper.

Let us consider the following working example.

```viper
method arrayExhale(a: Array, i: Int)
    requires // access to the entire array
    requires len(a) > 0
    requires i >= 0 && i < len(a)
    ensures // access to the entire array
{
    label 10
    assume fold(snap(intRange(0, len(a)))) == 10
    assume fold(snap(intRange(0, i))) == 4

    exhale forall e: Int:: i in intRange(0, i) ==> acc(loc(a,i).val)
    label 11

    assert fold(snap(intRange(i, len(a)))) == 6
}
```

Listing 4.12: Array Exhale Example

To encode our exhale strategy on our example, we need to declare a new variable `lostP_val_l1`, denoting the references with permission lost to field `val`.

```viper
var lostP_val_l1: Set[Ref]
assume forall pElem: Ref ::
    { (pElem in lostP_val_l1) }
    (pElem in lostP_val_l1) ==
    (perm(pElem.val) == none &&
    old[10](perm(pElem.val)) > none)
```

The axiom in the `assume` statement says that an element `pElem` is in this set if and only if the method holds some permission to `pElem.val` in state 10 but no permissions (`none`) in the new state, i.e., the method lost permissions to this reference’s `val` field. We generate one such variable and axiom pair for every relevant field in the exhale.
Next, we want to construct the filter $f'$ of remaining elements from the original filter $f$, which is \texttt{intRange(0, len(a))} in the above example. To do so, we define a domain function \texttt{filterNotLost()}, denoting the remaining filter after some permissions were lost, as in Definition 3. We display the function and its axioms below.

\begin{verbatim}
function filterNotLost(f1: Set[A], r: Receiver[A], lostR: Set[Ref]): Set[A]

axiom _filterNotLostAxiom {
  (forall a: A, fs: Set[A], r: Receiver[A], lostR: Set[Ref] ::
    (a in filterNotLost(fs, r, lostR)) ==
    ((a in fs) && !((recApply(r, a) in lostR))))
}

axiom _filterNotLostSubset {
  (forall fs: Set[A], r: Receiver[A], lostR: Set[Ref] ::
    (filterNotLost(fs, r, lostR)) subset fs)
}
\end{verbatim}

The first axiom says element $a$ is in the new filter if it is in the original filter, and \texttt{receiver(a)} is not a reference in the lost references set. The second axiom states explicitly that this new filter is a subset of the original filter. The trigger choices in these axioms are standard.

### 4.3.4 Inlined Axiom for Exhales Decomposition

With all of these sets constructed, we implement the exhale strategy by inlining another axiom after the exhale statement. The axiom is shown in Listing 4.13. The LHS of the axiom encodes the well-definedness checks of the strategy. The checks require injectivity of the receiver on the filter, old state permissions to the heap locations prescribed by the filter, receiver, and field, and current state permissions to heap locations prescribed by the newly created \texttt{filterNotLost} filter.

The RHS of the axiom first generates an \texttt{exhaleFoldSnap} instance that
```plaintext
label 10
exhale forall e: Int:: i in intRange(0, i) ==> acc(loc(a,i).val)

label 11
var lostP_val_l1: Set[Ref]
// omitted the axiom for lostP

assume forall fo: Fold[Int, Int, Int], f: Set[Int] ::
{ old[10]((hfoldApply(fo, snap_Int_Int_Int_val(fo, f))))}
(filterReceiverGood(f, (getreceiver(fo)))) &&
((forall __ind: Int ::
{ (__ind in f) } 
(__ind in f) ==> 
old[10](perm((recApply((getreceiver(fo)),
  __ind): Ref).val) ==
write)) &&
(forall __ind: Int ::
{ (__ind in
(filterNotLost(f, (getreceiver(fo)), lostP_val_l1))) } 
(__ind in
(filterNotLost(f, (getreceiver(fo)), lostP_val_l1))) ==> 
perm((recApply((getreceiver(fo)), __ind): Ref).val) ==
write)) ==> 
exhaleFoldSnap(fo, old[10](snap_Int_Int_Int_val(fo, f)), 0) &&
dummy1(hfoldApply(fo, snap_Int_Int_Int_val(fo, (filterNotLost(f,
  (getreceiver(fo)), lostP_val_l1)))))) &&
triggerDeleteBlock(old[10]((hfoldApply(fo,
  snap_Int_Int_Int_val(fo,
  f))),(filterNotLost(f, (getreceiver(fo)),
  lostP_val_l1))))
```

Listing 4.13: Exhale Decomposition Axiom

takes the fold object, the old snapshot, and a fieldID integer as input. This is used to trigger an axiom from the inhale strategy, where we try to reconstruct a filter that existed before the exhale. Think of the exhaleFoldSnap as saving this fold and snapshot pair for the later inhale strategy axiom.

The next term generates the output fold using a dummy function. dummy1 simply generates the fold without giving any additional information about it; think of it as constructing the output fold. Then, the second conjunct uses a triggering function triggerDeleteBlock to decompose the input fold (the
old one), as described by the listing on page 60. The decomposition looks something like this.

\begin{verbatim}
1 hfoldApply(fo, snap) ==
2     opApply(getoperator(fo),
3         hfoldApply1(fo, mapDelete(snap, keys)),
4         hfoldApply1(fo, mapSubmap(snap, keys)))
\end{verbatim}

We have previously explained \texttt{mapDelete} as a function implemented by a post-condition of the snap function; \texttt{mapSubmap} is the same except the opposite. Instead of deleting keys from the map, it keeps only these keys on the map.

The following code snippet summarizes the encoding for \texttt{mapSubmap}.

\begin{verbatim}
1 function mapSubmap(m: Map[A,B], e: Set[A]): Map[A,B]
2     ...
3 function snap_Int_Int_Int_val(fo: Fold[Int, Int, Int], filter: Set[Int]): Map[Int,Int]
4     ...
5     ensures (forall s: Set[Int] ::
6         (mapSubmap(result, s))
7         (s subset filter) ==> (mapSubmap(result, s)))
8     ...
\end{verbatim}

\textbf{Listing 4.14: Encoding of mapSubmap}

Again, the output is a prime instance of the snap function for the same reasons as the \texttt{mapDelete} axiomatization.

This decomposes the input \texttt{fold(snap(intRange(0,len(a))))} as

\begin{verbatim}
old(fold(snap(intRange(0,len(a)))) =
     old(fold(snap(intRange(0, i)))) +
     old(fold(snap(intRange(i, len(a))))).
\end{verbatim}

Some of these terms would technically contain \texttt{setminus} and \texttt{filterNotLost}, but we have simplified that for readability. The output fold generated would be \texttt{fold(snap(intRange(i, len(a))))}, which by Viper framing is equal to \texttt{old(fold(snap(intRange(i, len(a))))).}

From the original program, we know
\[\text{old}(\text{fold}(\text{snap}(\text{intRange}(0, \text{len}(a)))))) == 10\]

and
\[\text{old}(\text{fold}(\text{snap}(\text{intRange}(0, i)))) == 4,\]

so from the decomposition, we can deduce
\[\text{old}(\text{fold}(\text{snap}(\text{intRange}(i, \text{len}(a)))))) == 6,\]

making the final assertion verify. The assertion would go through because
\[10 = 4 + 6,\]

and permission framing, which guarantees
\[\text{old}(\text{fold}(\text{snap}(\text{intRange}(i, \text{len}(a)))))) == \text{fold}(\text{snap}(\text{intRange}(i, \text{len}(a))))\].

In summary, we perform the following steps at each exhale statement where permissions are lost.

1. Declare a new variable \texttt{lostRef} of type \texttt{Set[Ref]}

2. Generate an \texttt{assume} statement, denoting that the \texttt{lostRef} variable stores all the references to which the method lost permissions (relative to some specified field).

3. Generate another \texttt{assume} statement with a quantifier (our inlined axiom) that encapsulates the following:

   (a) The trigger is \{\texttt{old[10]}(\texttt{fold(...))}\} representing the input folds, i.e., the folds in the state immediately prior to the exhale (at label 10). Such folds would match the trigger and instantiate the axiom. Each input fold has a corresponding filter \texttt{f}.

   (b) The LHS of the axiom does the well-definedness checks.

   (c) The first term in the RHS records the input folds, its snapshot, and its \texttt{fieldID} using the \texttt{exhaleFoldSnap} function. This is used for the inhale strategies, which we explain in the coming subsection.

   (d) The second term on the RHS generates a new \texttt{fold(filterNotLost(f))}, without the \texttt{old} expression, meaning it is the output fold in the state
after the exhale. The filter $\text{filterNotLost}(f)$ is the subset of the original input fold’s filter $f$ that the method retains permissions to (relative to the receiver, field, and $\text{lostRef}$).

(e) The third term on the RHS applies the function $\text{triggerDeleteBlock}$ to the input fold with $\text{filterNotLost}(f)$ as an additional argument. This dummy function instantiates the axiom shown in Listing 4.6, which performs a decomposition on the input fold by subtracting $\text{filterNotLost}(f)$ from the input fold’s filter $f$. Since $\text{filterNotLost}(f)$ is always a subset of $f$, this decomposition always happens.

### 4.3.5 Encoding the Inhale Strategy

Finally, we encode the inhale strategy as described in the definition on page 48 and the list above the definition. Starting from some input $\text{fold}($snap$(f))$, we want to construct an output $\text{fold}($snap$(f^*))$ where $f^*$ is a bigger filter. We derive this bigger filter from an input fold to exhale nodes. Recall that we introduce the function $\text{exhaleFoldSnap}(fo, m, id)$ for saving the input folds to exhale nodes and their snapshot $m$. We use this as a trigger in the inlined axiom for inhale.

### 4.3.6 Inlined Axiom For Inhale Decomposition

Our plugin inlines the following assume statement after a relevant inhale where permissions are gained. Suppose the state before the inhale has the label $l0$, and the inhale node has the input filter $\text{fold}($snap$(f))$. 


The trigger for this axiom is a fold application immediately before the inhale statement (in state 10), and \text{exhaleFoldSnap}(fo, m, 0), where 0 is the fieldID of field \text{val}. Our plugin keeps track of these IDs internally; each field has a unique ID. The \text{exhaleFoldSnap} would be generated for all folds and snapshots that are input folds to exhale nodes, and each instance of the function application would match the trigger. We obtain the big filter \(f^*\) by taking the domain of map \(m\) as matched in the trigger \text{exhaleFoldSnap}(fo, m, 0). Recall that the domain of a snapshot is its filter argument.

The LHS of the axioms denotes the well-definedness checks, such as the injectivity of the receiver and the usual permissions. There are two separate injectivity conditions; the first is the injectivity of the input fold to the inhale. The second is the injectivity of the output fold, which we generate. The third condition on the LHS (\(f \text{ subset domain}(m)\)) requires that the input fold’s filter \(f\) is actually a subset of \(f^*\). The final condition on the LHS uses quantified permissions to check that the method has permissions to the heap locations denoted by the filter \(f^*\).

On the RHS of the axiom, the first \text{triggerDeleteBlock} generates the output fold\(\text{snap}(f^*)\) and decomposes it by subtracting the filter \(f\). The
second `triggerDeleteBlock` decomposes the exhale node’s input fold in
the same way for symmetry, as previously described in the last chapter on
page 48.

To summarize, for encoding the inhale strategy, we only generate one
assume statement with a forall quantifier, i.e., our inlined axiom. The axiom
encodes the following operations:

1. The first trigger is `old[10](fold(...))` representing the input folds,
i.e., the folds in the state immediately prior to the inhale (at `label 10`).
Such folds would match the trigger and instantiate the axiom. Each
input fold has a corresponding filter \( f \).

2. The second trigger is `exhaleFoldSnap(fold, snap, fieldID)`, which
represents the folds and snapshots that we input folds to previous exhale
nodes. We wish to reconstruct the filter encoded in those snapshots.
For this axiom to instantiate, there must be `exhaleFoldSnap(fold, snap, fieldID)`
where the `fold` and `fieldID` arguments match those
of the input fold. Note that both the first and second trigger must
match for our inlined axiom to instantiate.

3. The first term on the LHS checks that the input fold’s filter \( f \) is a
subset of the `snap`’s filter, i.e., the larger filter \( f^* \).

4. The next terms on the LHS perform the other necessary well-definedness
checks. Most importantly, there is a check that after the inhale, the
method has permissions to all heap locations defined by \( f^* \) (and the
receiver and field).

5. If all the checks pass, the RHS of the axiom generates an output fold
with the larger filter \( f^* \), in the new state after the inhale.

6. Decompose the output fold by subtracting \( f \) from \( f^* \), via instantiating
the `triggerDeleteBlock` function.

7. From the `exhaleFoldSnap(fold, snap, fieldID)`, decompose the
relevant input fold to the corresponding exhale by subtracting \( f \) using
the same `triggerDeleteBlock` function.
And we have finished encoding all our strategies into Viper.
Chapter 5

Implementation in Scala

In this chapter, we present the implementation of our encodings as a Viper plugin, written in Scala.

5.1 Viper Plugin Architecture

The source code for the Viper programming language [7] contains a built-in framework for building plugins, allowing us to interact with the internals of Viper. The internal Viper process starts from a source file written by the user and ends at verification. This process goes through the following stages:

1. Parsing: parse the strings read from the file to generate a parse tree, also called a parser abstract syntax tree (PAST).

2. Resolution and type-checking: resolve the identifiers and type-check the parse tree.

3. Translation: translate the parse tree to an abstract syntax tree (AST).

4. Verification: feed the generated AST to the specified Viper verifier, i.e., either Silicon or Carbon.

The plugin framework allows us to insert hooks between these stages, particularly at the following steps:

1. Before parsing
2. Before resolution and type-checking
3. Before translation
4. Before verification

We implement our plugin to support heap-dependent folds by implementing the appropriate hook at each of these steps. In the following sections, we describe each of these hooks.

5.2 Before Parsing

The Viper framework allows us to implement a hook before a Viper program gets parsed. In this hook, we modified the parser, adding syntax for expressing a heap-dependent fold, which would be considered a Viper expression. In addition, we add syntax for defining each of the individual fold components, namely the receiver function, the filter, the mapping, and the operator.

5.2.1 Specifying a User Syntax

First, we translate from the following formal notation of a fold,

\[ \text{fold}_r[o](m(r.F) | f) \],

to syntax for the user in Viper.

We define the fold expression syntax to be

\[ \text{hfold}[o](m(r.F) | f) \]

with expressions \( f: \text{Set}[A], r: \text{Receiver}[A], m: \text{Mapping}[V,B], o: \text{Operator}[B] \) and some field \( F \) of type \( V \). The type variables \( A, V, B \) correspond to the type variables of our \text{Fold} domain encoding, as seen in Listing 4.1 on page 53. Again, we chose \text{hfold} for heap-dependent fold as the keyword because \text{fold} is already a keyword for folding Viper predicates. The variable \( m \) can be omitted, which makes the mapping function default to the identity function.

After all the appropriate processing, this expression internally translates to an application of \text{hfoldApply} in our encoding, as shown below.
We explained the meaning of this encoding on page 57.

Note that the `hfold` in the user-facing syntax is different from the `hfold` we use internally in our encoding. Internally, `hfold` is just a constructor for a `Fold` object, which is an abstracted internal component that the user cannot interact with directly. The `hfold` that the user writes would evaluate the fold, i.e., it translates to a `hfoldApply` as shown in the above listing.

Likewise, users do not have direct access to our encodings’ constructors for the fold expression inputs, like `Receiver[A]` or `Operator[B]`. Instead, we add user syntax for declaring each fold component. The following example is a collection of declarations for components in a fold, which collects all integers inside an array into a multi-set.

```plaintext
receiver arrayRec(a: Array) (fun i: Int :: loc(a,i))
filter intRange(start: Int, end: Int)
  (fun i: Int :: i >= start && i < end )
mapping singleM() (fun i: Int :: Multiset(i))
identityOp mUnion() (Multiset[Int](),
  fun a: Multiset[Int], b: Multiset[Int] :: a union b )
```

We call these *fold component declarations*. These declarations must be made at the top level, so they cannot be inside a method or domain. These declarations cannot simply contain variables that one would use in a method.

So, our design allows for two sets of variables/arguments, and these declarations declare functions that output functions. There are *outer variables*, such as the `a: Array` in `arrayRec(a: Array)`, and there are *inner variables*, such as the `fun i: Int`. The number of inner variables is already fixed by our definitions. For example, a receiver function has type \( A \rightarrow \text{Ref} \), so it must take only one argument, which we bound in the above as `i: Int`. However, the number of outer variables can be arbitrary. The above example has declarations with one, two, or no outer variables. Then, when writing the fold syntax inside a method, the user can apply the declared functions, bounding the outer variables to the relevant method-local variable.

The `receiver` keyword describes a syntax to declare a receiver function \( A \rightarrow \text{Ref} \), modulo some outer variables. The `filter` keyword declares a
Set[A] via a function returning a Boolean. The mapping keyword declares a function to be applied after accessing the relevant heap location. The identityOp keyword declares an operator and its identity; the user must input the identity and the binary operator as a tuple. Since the identity can typically be expressed more briefly, we designed it as the first element of the user-input tuple. The field can be defined separately using Viper’s built-in syntax.

These declarations need to be type-checked, and we discuss our interaction with the type-checker in more detail in Section 5.4.

5.2.2 Custom PAst Nodes

In addition to adding new syntax, we modified the parser to generate custom parse nodes for the fold expression and each type of fold component declaration.

The custom parse nodes for the fold components declarations serve two purposes. First, they must ensure that the nodes are resolved as function declarations so the user can write the declared elements inside the fold expression syntax or even interact with it outside a fold, for the case of the filter. For example, the filter defined in the example declarations on page 88 can be queried inside a method with an assertion like assert 4 in intRange(0,5).

The second purpose of the custom parse nodes is to define their typing-checking behaviour. In the Viper Scala implementation, parse nodes have a shared trait, where each node must define a type-checking method. We discuss this more in the upcoming Section 5.4.

5.3 Before Type-checking and Resolving

After the parser runs and generates a parse tree, including custom nodes, we can interact with the parse tree through another hook before identifier resolving and type-checking happens. At this stage, we add the domain declarations described in the last chapter, such as the Fold domain, as parser nodes into the generated parse tree. Internally, we generate these declarations.
as strings and feed them to the parser again to generate a collection of parser nodes. Then, we simply add this collection to the previously generated parse tree.

An alternative design choice we briefly considered was appending these domain declarations directly into the source file, so the parser needs to run only once on the source. However, this creates incorrect line numbers in error messages to the users since these domains are not written in the original file. Generating parse nodes on these domains separately allows us to maintain the original line numbers in the source file.

5.4 During Type-checking and Resolution

We define the appropriate type-checking functions for each of the custom parse nodes.

Type-checking for Component Declarations

One goal of type-checking is inferring the instantiations for the type variables in the declarations. For example, a receiver function has type $A \rightarrow \text{Ref}$, so for each receiver declaration, we need to infer a concrete type for variable $A$. Most of the time, this amounts to reading the function argument annotations, like \texttt{fun i: Int}, but sometimes the inference has to occur inside the body of the function. For example, a mapping is a function $V \rightarrow B$; we can deduce type variable $V$ from the argument annotations, but to deduce $B$, we have to infer the type of the expression inside the declaration body.

In addition, the type-checker needs to verify that the body has the intended type and that the number of inner arguments is correct. For example, the receiver function’s body must have the type \texttt{Ref}, and the function must have one (inner) argument.

Type-checking for Fold Expressions

As we have explained, a fold expression must contain $f: \text{Set}[A]$, $r: \text{Receiver}[A]$, $m: \text{Mapping}[V, B]$, $o: \text{Operator}[B]$ as inputs, along with field $F$ of type $V$. Each component of the fold should have types that match.
For example, if the filter \( f \) is a set of integers, the receiver must be a \( \text{Receiver[Int]} \). If the specified field is typed \( \text{Int} \), and the type variable \( V \) must be \( \text{Int} \) in mapping. If the mapping has the type \( \text{Mapping[Int, Multiset[Int]]} \), then the operator must have type \( \text{Operator[Multiset[Int]]} \). All of this is implemented with the appropriate type-checking and inference for each fold expression.

5.5 Translation to AST

After type-checking and resolution, the parse tree gets translated into an abstract syntax tree. Each custom parse node requires its own translation rule.

5.5.1 Translating the Fold Component Declarations

Each component declaration parse node gets converted into a domain AST node. Each domain has a function that returns the defunctionalized representation of the declared component and a corresponding axiom describing its behaviour. For example, the declaration \( \text{receiver arrayRec(a: Array)} \) (\( \text{fun i: Int:: loc(a, i)} \)) gets translated into a domain, which would look like the following if printed.

```
1 domain arrayRec_Receiver_Domain { 
2   function arrayRec(a: Array): Receiver[Int]
3
4   axiom { 
5       (forall a: Array, i: Int :: 
6           { (recApply(arrayRec(a), i): Ref} 
7           { loc(a, i) 
8             (recApply(arrayRec(a), i): Ref) == loc(a, i)) 
9       }
10   }
```

The axiom contains a forall quantifier over both the outer and inner variables, as written in the top-level declaration. With the defunctionalization technique and using the \( \text{recApply} \) function, we can specify the intended behaviour of the receiver.
A tricky part is automatically picking triggers for the axiom. Though we could leave the quantifier with no triggers, letting the SMT solver choose for us, we generally want to define our own explicitly. In this case, we define two triggers disjunctively, one representing the application of the receiver and the second representing the body of the receiver function. The same idea applies to other fold components.

Yet, there are some instances where the body of the function is not a valid trigger. For example, for the declaration \texttt{filter intRange(start: Int, end: Int) \{fun i: Int:: i >= start \&\& i < end \}}, the body of the function is \( i \geq \text{start} \&\& i < \text{end} \); this expression contains interpreted symbols, which are not allowed in triggers. Luckily, the Viper implementation has a built-in method for identifying invalid triggers, which we use to detect and filter them out. This could lead to some incompleteness if the user expects the term \( i \geq \text{start} \&\& i < \text{end} \) to instantiate a new term \( i \) in \texttt{in intRange(start, end)}. In realistic usages of our plugin, however, this incompleteness does not appear in practice.

5.5.2 Translating the Fold Application

We translate each fold expression parse node into its proper encoding described in the previous chapter. We define 3 custom classes of AST nodes for this purpose. Firstly, we declare a custom AST node called \texttt{AFold3Tuple} representing the fold 3-tuple, similar to the \texttt{hfold} object as explained in the listing on page 53. This encapsulates the three arguments: receiver, mapping, and operator. Next, we declare a custom AST node called \texttt{ASnapApp} representing the snapshot application as explained in the listing on page 55. Lastly, we have a custom AST node called \texttt{AFoldApply}, representing the fold application to the snap.

These custom AST nodes correspond directly to our internal Viper encoding. A tricky part is handling the snapshot applications and extracting the correct snapshot declaration. Recall that the snapshot function is a heap-dependent function that cannot be declared as a domain function. We need to use monomorphization to declare separate instances of a snapshot
declaration corresponding to each type variable contained in \texttt{Fold[A, V, B]} and the specified field. We generate a function declaration AST node representing the snapshot declaration for each unique combination of these.

5.6 Adding Well-definedness Checks and Inlined Axioms

After the program translation to AST, we have the last hook of the plugin framework: the hook before verification. At this stage, we add the operator well-definedness checks and the inlined axioms described in the last chapter. In addition, we also internally convert method calls into a combination of inhales, exhales, and reassignment, to be explained in the upcoming Section 5.6.1.

We generate a new method AST node containing the corresponding commutativity, associativity, and identity well-definedness checks for each of the operators used in a fold expression. We do not add the check method if an operator is defined but unused in a fold expression.

5.6.1 Desugaring Method Call

Our strategies assume there are only 3 heap modifying operations: reassignment, inhales, and exhales. So, for our strategies to work with method calls, the calls must be converted into these three operations. Usually, the verifier handles each method call modularly, but we implement this directly into our plugin.

A method call can be translated into exhales of its preconditions and inhales of its postconditions. The pre/postconditions of a method can be extracted from the method declaration, but we must manually replace the variables with the call site’s inputs.

Furthermore, a method can have a return value, which may also appear in the postcondition. To support this, we declare a unique temporary output variable for each method call and replace every instance of the return value in the postcondition with this temporary variable. If the method return value is used as input to a reassignment, we perform the reassignment using
our temporary variable instead.

Finally, we reverse the order of the preconditions because some assertions must be done before losing permissions. For example, consider the following method preconditions.

1. \texttt{requires acc(r.val)}
2. \texttt{requires r.val > 0}

These should be reversed and converted into the following exhales:

1. \texttt{exhale r.val > 0}
2. \texttt{exhale acc(r.val)}

Without the reversal, the exhale would fail because the method would lose the permissions to \texttt{r.val} and then try to read it afterwards. Error messages may appear in reversed order but could be fixed by modifying how the plugin handles errors.

5.6.2 Adding Inlined Axioms

Finally, we add the inlined axioms explained in the previous chapter at the appropriate inhales, exhales, and reassignments. We only want to add these axioms to the statements that modify the state for inhales and exhales. For inhales and exhales to modify state, they must contain accessibility predicates \texttt{acc(A)} either independently or in quantified permissions. We detect these accessibility predicates and the corresponding field. Then, we generate the inlined axioms after the relevant statements for each fold expression with the specified field.

After each exhale, we also declare a variable representing the set of lost references, as described in Definition 2 on page 43. We insert the corresponding assume describing the set after the variable declaration.

Furthermore, we generate and track labels corresponding to the states before and after each heap-modifying statement. These labels are used in \texttt{old} expressions inside the inlined axioms.
Heap Reads

During our implementation, we observed that an inlined axiom could be useful when a heap element is read, not just when it is modified. Hence, we also decided to generate an inlined axiom for each heap read. For example, a read of \texttt{loc(a, i).val} would generate a singleton decomposition to all folds in the state of the read. This is like the strategy for handling heap reassignment, except the inlining does not generate any intermediate fold because the heap state does not change from a read.

There are a few challenges we encountered when implementing this addition. For one, a heap read can occur inside a quantifier, and the read expression can contain a quantified variable. These quantified variables cannot simply be mentioned outside the quantifier, so our plugin does not generate axioms for these tricky reads. Future work could investigate how to support decomposition guided by a quantified read.

Another challenge comes from reads that occur together with a heap-modifying statement. To ensure that the decomposition is in the right state, we are careful to perform the decomposition before the reassignment. If the read and write happen to the same heap location, we also detect that to avoid generating redundant axioms for reassignment and read.

Finally, after all the inlined axioms are inserted, we transform our custom AST nodes into the vanilla Viper nodes. The transformations are straightforward because our custom AST nodes are defined with a similar structure to our Viper encodings. Each \texttt{AFold3Tuple} becomes a proper Viper function call node represented by \texttt{hfold(receiver, mapping, operator)}, each \texttt{ASnapApp} becomes the function call node of the monomorphized snapshot function, and each \texttt{AFoldApply} becomes a call node printed as \texttt{hfoldApply(...)}. After the transformations, the Viper AST no longer has any custom nodes and can be fed to the specified verifier.
Chapter 6

Evaluation

In this chapter, we discuss some of the tests we conducted with our plugin and some challenges we encountered during the design and implementation of the plugin. Finally, we finish with some possible limitations from the user’s perspective.

6.1 Test Cases

We conducted some tests to demonstrate the main features of our Viper plugin. These tests are not designed to be rigorous performance benchmarks; we are more focused on illustrating our plugin’s features and possible use cases. However, we briefly discuss the observed runtime and some factors that may have affected performance in each example. In some tests, we also inspect the results using the Axiom Profiler [3], a tool that visualizes quantifier instantiations. We describe our findings in each subsection.

All tests are run using the Carbon backend because the Silicon backend does not support heap-dependent triggers, which we rely on for our encoding. Silicon also does not allow quantifier triggers that do not appear in the quantifier body. For instance, our dummy function triggerDeleteKey is not a valid trigger in our inlined axioms shown in Listing 4.5 on page 59 because the function does not appear in the quantifier body. Hence, we only use the Carbon backend for our tests.
6.1.1 Chained Singleton Reassignment

Firstly, we consider a program that performs six consecutive heap reassignments, as shown in Figure 6.1. We previously described a combinatorial explosion that occurs when many heap reassignments are chained together, so this example tests the actual performance of our plugin in such a scenario.
Figure 6.1: Six Consecutive Assignments

```plaintext
field val: Int
receiver arrayRec(a: Array)
  (fun i: Int :: loc(a,i))
filter allInt(start: Int, end: Int)
  (fun i: Int :: i >= start && i < end )
identityOp add()
  (0, fun a: Int, b: Int :: a + b )
define access_array(a, i1, i2)
  (forall j: Int :: {loc(a,j)} i1 <= j && j < i2 ==>
   acc(loc(a,j).val))

method test1(i0: Int, a:Array)
  requires access_array(a, 0, len(a))
  requires len(a) > 10
  requires i0 in allInt(0,len(a))
{
  var i1 : Int
  assume i1 in allInt(0,len(a))
  var i2 : Int
  assume i2 in allInt(0,len(a))
  var i3 : Int
  assume i3 in allInt(0,len(a))
  var i4 : Int
  assume i4 in allInt(0,len(a))
  var i5 : Int
  assume i5 in allInt(0,len(a))
  loc(a,i0).val := loc(a,i0).val + 1
  loc(a,i1).val := loc(a,i1).val + 1
  loc(a,i2).val := loc(a,i2).val + 1
  loc(a,i3).val := loc(a,i3).val + 1
  loc(a,i4).val := loc(a,i4).val + 1
  loc(a,i5).val := loc(a,i5).val + 1
  assert hfold[add()](arrayRec(a).val | allInt(0,len(a))) ==
    old(hfold[add()](arrayRec(a).val | allInt(0,len(a)))) + 6
}
```

98
As explained in the Design chapter, multiple reassignments chained together could lead to a combinatorial explosion of decompositions. Without our local and intermediate decompositions strategy, verification using the naive strategy would exceed 100 seconds even after just 3 consecutive reassignments.

Our example program has an assertion with a fold expression describing the sum of the integer array from indices 0 to the array length. Our plugin verifies the program within 60 to 80 seconds, which is an improvement from the naive strategy. There are usually fewer than 6 consecutive reassignments in practical programs, so such programs would verify even faster.

Yet, the runtime is still longer than we expected with our technique. Technically, there should only be 12 decompositions on folds, one before and one after each reassignment. To investigate, we measured the verification time with various numbers of reassignments: 3 reassignments take 7 seconds, 4 reassignments take 15 seconds, 5 reassignments take 32 seconds, and 6 reassignments take about 70 seconds. We observed that the time seems to double per a single increase in the number of reassignments.

This exponential behaviour is surprising because the number of folds and their decompositions should theoretically increase linearly per the number of reassignments, as explained in the design chapter. We hypothesize that this results from some unknown case splitting. The SMT solver may be case-splitting in its proof search on an additional term for every new reassignment.

In a case split, the SMT solver would search two paths, potentially duplicating all terms in one branch to the second, doubling the verification time. Upon using the Axiom Profiler, we observed that our inlined axioms repeatedly instantiate on the same term, which suggests that each instantiation appears in a different case/branch. However, the Axiom Profiler does not give specific information about the SMT solver’s case splitting. Future work into a tool or debugger for visualizing the SMT solver’s proof search and case splits may help us troubleshoot this issue.
6.1.2 Reassignment Enclosed In Inhales and Exhales

The next test method in Figure 6.2 has a reassignment operation enclosed between an exhale statement and an inhale statement, both with quantified permissions. The assertion at the end checks that the sum of the integer array has increased by 1. This case tests whether our encoding’s inlined axioms appropriately generate the flow of folds as described in the design. The exhale statement (node) takes an input fold and produces an output fold. That output fold is then an input fold to the assignment, which also produces an output fold. Finally, that output fold is also input to the inhale node, which produces an output fold in the state at the end of the method.

Figure 6.2: Reassignment Enclosed in Exhale and Inhale

```plaintext
method testExhaleInhale(i0: Int, a:Array)
  requires access_array(a, 0, 10)
  requires len(a) > 10
  requires i0 in allInt(0,5)
  {
    exhale forall i: Int :: i in allInt(5,10) =>
      acc(loc(a,i).val)
    loc(a,i0).val := loc(a,i0).val + 1
    inhale forall i: Int :: i in allInt(5,10) =>
      acc(loc(a,i).val)
    inhale forall i: Int :: i in allInt(5,10) =>
      loc(a,i).val == old(loc(a,i).val)
    assert hfold[add()](arrayRec(a).val | allInt(0,10)) ==
      old(hfold[add()](arrayRec(a).val | allInt(0,10))) + 1
  }
```

Our plugin verifies this example within 25 seconds, suggesting that our implementation works as intended. An intermediate fold is generated after each exhale, reassignment, and inhale statement. Ultimately, there is a path
linking the old fold to the fold in the final state of the method.

6.1.3 Multiset of Array Elements After Swap

The next test case utilizes a different fold that collects all array elements into a multiset, which is a different use case from our usual array sum example. Notably, the mapping function converts an integer stored in a heap location to a singleton multiset, and the operator is the multiset union. The Figure 6.3 has a swap method and a postcondition that the multiset of elements in the array remains the same after the swap. The statements inside the method lead to inlined axioms for both heap writes and heap reads. With these axioms, the plugin verifies this example in around 5 seconds.

```plaintext
receiver arrayRec(a: Array)
  (fun i: Int :: loc(a,i))
filter twoInts(i1: Int, i2: Int)
  (fun i: Int :: i == i1 || i == i2 )
mapping singleM()
  (fun i: Int :: Multiset(i))
identityOp mUnion()
  (Multiset[Int](),
  fun a: Multiset[Int], b: Multiset[Int] :: a union b )
define access_array(a, i1, i2)
  (forall j: Int :: {loc(a,j)} i1 <= j && j < i2 ==>
    acc(loc(a,j).val))
method swap(a:Array, i1: Int, i2: Int)
  requires acc(loc(a,i1).val) && acc(loc(a,i2).val)
  requires i1 >= 0 && i1 < len(a) && i2 >= 0 && i2 < len(a)
  ensures acc(loc(a,i1).val) && acc(loc(a,i2).val)
  ensures
    hfold[mUnion()](singleM(arrayRec(a).val)|twoInts(i1,i2)) ==
    old(hfold[mUnion()](singleM(arrayRec(a).val)|twoInts(i1,i2)))
{ var temp: Int := loc(a,i1).val
  loc(a,i1).val := loc(a,i2).val
  loc(a,i2).val := temp
}
```
Figure 6.4: Bubble Sort

```plaintext
filter allInt(start: Int, end: Int)
  (fun i: Int :: i >= start && i < end )
filter twoInts(i1: Int, i2: Int)
  (fun i: Int :: i == i1 || i == i2 )
define access_array(a, i1, i2)
  (forall j: Int :: {j in allInt(i1,i2)}
    j in allInt(i1,i2) ==> acc(loc(a,j).val))
define mUnionfold(a, f)
  (hfold[mUnion()](singleM(arrayRec(a).val) | f))

method bubbleSort(a: Array)
  requires access_array(a, 0, len(a))
  ensures access_array(a, 0, len(a))
  ensures mUnionfold(a, allInt(0,len(a))) ==
    old(mUnionfold(a, allInt(0,len(a))))
{
  var n : Int := len(a)
  var swapped : Bool := true
  while(swapped)
    invariant access_array(a, 0, len(a))
    invariant mUnionfold(a, allInt(0,len(a))) ==
      old(mUnionfold(a, allInt(0,len(a))))
    {
      swapped := false
      var i : Int := 1
      while(i < len(a))
        invariant access_array(a, 0, len(a))
        invariant mUnionfold(a, allInt(0,len(a))) ==
          old(mUnionfold(a, allInt(0,len(a))))
        {
          if (loc(a,i - 1).val > loc(a,i).val) {
            swap(a, i-1, i)
            swapped := true
          }
        }
    }
}
```
6.1.4 Bubble Sort Using Swap

The next example in Figure 6.4 involves a bubble sort method making calls to the \texttt{swap} method (as declared in Figure 6.3) to sort an integer array. The method is inspired by the pseudocode example on Wikipedia \cite{12}. The final assertion checks that the multiset of elements in the array remains the same. This test case checks our plugin’s handling of method calls and usage of fold expressions in loop conditions. In addition, we demonstrate these features in the bubble sort algorithm as a realistic use case. To focus on the fold, we omit the specifications for the sortedness property and the ghost code to verify it.

For brevity, we included the \texttt{hfold} expression inside a macro \texttt{mUnionfold} that takes an array and a filter as inputs. Note that the filter functions are different in the two methods. The first \texttt{swap} method has a fold with a filter \texttt{twoInts(i1,i2)}, while the second method \texttt{sortArray} has a fold with a filter \texttt{allInt(0,len(a))}, which is an entirely different function. Yet, our generated axioms are still able to reason about both filters together.

With modifications to the Boogie file to remove lingering axioms from well-definedness checks, the example verifies in around 20 seconds. Without these modifications, the example timeouts at 200 seconds. Recall that Viper’s Carbon verifier generates a Boogie file before invoking Z3, the SMT solver. We discovered that Carbon generates some assertions containing quantifiers without trigger annotations, especially for well-definedness checks. The quantifiers without triggers can potentially lead to matching loops, as the SMT solver may automatically pick the wrong triggers. This is a bug, and the correct implementation would use a dummy function \texttt{neverTriggered(...)} as a trigger to prevent the quantifier from instantiating after the assertion finishes.

We suspect that the nested while-loops and a method call in the \texttt{bubbleSort} method generates more well-definedness checks than usual, exacerbating the performance effects of this bug. However, upon removing these well-definedness checks, the method verifies quite quickly.
**Figure 6.5: Graph Sum**

```plaintext
1  field val: Int
2  field left: Ref
3  field right: Ref

4  define INV(nodes)
5      !(null in nodes)
6      && (forall n: Ref :: n in nodes => acc(n.val))
7      && (forall n: Ref :: n in nodes => acc(n.left))
8      && (forall n: Ref :: n in nodes => acc(n.right))
9      && (forall n: Ref :: {n.left in nodes}{n in nodes, n.left}
10         n in nodes && n.left != null =>
11            n.left in nodes)
12      && (forall n: Ref :: {n.right in nodes}{n in nodes, n.right}
13         n in nodes && n.right != null =>
14            n.right in nodes)

15  receiver idenRec() (fun r: Ref :: r)
16  identityOp add() (0 ,fun a: Int, b: Int :: a + b )

17  method sumNodes(nodes : Set[Ref], node: Ref)
18     requires node in nodes
19     requires INV(nodes)
20     ensures INV(nodes)
21     ensures hfold[add()](idenRec().val | nodes) ==
22        old(hfold[add()](idenRec().val | nodes)) + 2
23     {
24         node.val := node.val + 1
25         var node2: Ref
26         assume node2 in nodes
27         node2.val := node2.val + 1
28         var node3: Ref
29         assume !(node3 in nodes)
30         inhale acc(node3.val)
31         node3.val := node2.val + 1
32     }
104
```
6.1.5 Graph Data Structure Sum

Our final example illustrates how our design can also model data structures other than arrays. Figure 6.5, inspired by an example on the Viper page [8], models a graph where each node stores an integer and has pointers to two other nodes: left and right. The graph is modelled as a set of these nodes, i.e., \texttt{Set[Ref]}. Suppose we want to use the fold to express the sum of all integers stored in the graph. Unlike the previously discussed set of integers, the filter here is a set of references. The receiver is the identity function.

The method body in the program contains reassignments to two nodes. The first two reassignments increment the node’s stored integers by 1. The last reassignment is a decoy, incrementing the integer of a node not on the graph. The method post-condition states that the sum of integers on the graph becomes larger by 2. With our plugin loaded, this example verifies within 5 seconds.

6.2 Challenges in Implementation

We faced some challenges during the design and implementation of our plugin, which may have impacted the completeness or performance of our encoding.

6.2.1 Reliance on Extensional Equality Checks

The SMT solver requires costly extensional equality checks to deduce equality between certain filters. For example, consider the following filter declaration.

\begin{verbatim}
filter intRange(start: Int, end: Int) (fun i: Int :: i >= start && i < end )
\end{verbatim}

Then, suppose that we have the following fold decompositions:

\begin{verbatim}
assume fold(intRange(0,10)) ==
    fold(intRange(0,10) setminus intRange(5,10)) +_{op}
    fold(intRange(5,10))
assert fold(intRange(0,5)) == ... //assert something about this fold
\end{verbatim}

The final assert checks some property of the fold over the filter \texttt{intRange(0,5)}. From a user perspective, it may be obvious that this filter is equal to the
filter \texttt{intRange}(0, 10) \texttt{setminus} \texttt{intRange}(5, 10). However, this equality is not immediately obvious to the SMT solver, and we must trigger an extensional equality check, i.e., asserting or assuming an explicit \texttt{intRange}(0, 10) \texttt{setminus} \texttt{intRange}(5, 10) == \texttt{intRange}(0, 5). Internally, we have axioms that check all filters against each other for extensional equality.

Using the Axiom Profiler, we discovered that this extensional equality axiom has a high cost. In the Axiom Profiler, the cost of an axiom refers to the total number of instances generated by it, along with the instances generated by those instances, and so on. Generally, a high cost suggests that the axiom led to poor performance.

Nevertheless, we are currently dependent on the extensional equality checks. Without them, the typical use cases would not be verified because the SMT solver would not automatically deduce the equality of syntactically different filters.

### 6.2.2 Reliance on Heap-dependent Triggers

Our generated inlined axioms decompose folds in states local to the axiom; this can only be achieved using old expressions inside the axiom body and inside the trigger. Unfortunately, the behaviour of heap-dependent triggers is not well-documented, and we observed that the axioms may instantiate unintentionally. This generates extraneous output fold terms, which may instantiate other inlined axioms in other states. Although these do not generate matching loops, they could lead to poor performance.

### 6.2.3 Performance Issues and Lack of Debugging Options

Our plugin suffers from poor performance in programs with many exhales and inhales. Despite our best efforts, the verification times remain quite slow. The causes are unclear, and we lack the tools to properly debug. For instance, we cannot see the SMT solver’s proof steps to verify an assertion. If we could see the steps, perhaps we can investigate the instantiations generated for each assertion and identify the extraneous ones to be deleted. The next best thing is the Axiom Profiler, which shows a tree of axiom instantiations but
does not visualize key details such as the proof steps or the SMT case splits.

6.3 Limitations and Discussion

From the user’s perspective, the tool may have some limitations. Firstly, our plugin supports heap-dependent folds, so the user would not be able to express general folds or folds with a combination of heap-dependent and non-heap-dependent portions.

Secondly, users may find the syntax to be quite verbose. Each fold component must be declared separately at the top level and combined in a rather large expression. A possible extension could be allowing inlined anonymous functions as input to the fold expression. In the current design, the user could declare a macro if certain combinations of fold components are used repeatedly.

Finally, there are some completeness limitations. Arbitrary folds are not decomposed automatically without the appropriate inhales, exhales, or reassignments. For example, suppose we are considering a fold representing the sum of elements inside an array. Given assumptions

- \( \text{fold(intRange}(0,10)) == 10 \)
- \( \text{fold(intRange}(5,10)) == 6 \)

the user may want to derive \( \text{fold(intRange}(0,5)) == 4 \). However, this would require a decomposition, which would not happen automatically without a heap-modifying statement. As a workaround, we included the function \( \text{disjUnionEq(s1, s2, s3)} \) in our plugin, which says that sets \( s1 \) and \( s2 \) are disjoint, and that \( s1 \cup s2 == s3 \). An instance of this function would instantiate the appropriate decomposition \( \text{fold(s3)} = \text{fold(s2)} \oplus \text{fold(s1)} \). However, this is not automatic; the user must manually write that function in an assert or assume statement.

Despite these limitations, our work has many positives. We have defined a novel technique for reasoning about heap-dependent folds that are fully customizable. For example, the user could define new types for the indices and design any operator they wish, provided everything is well-defined. We
introduced the local and intermediate decomposition techniques to allow reasoning about folds modularly at each state change. This improves performance significantly compared to the naive strategy explored in some previous work (discussed in the final chapter). Our test results demonstrate good performance and the technique’s flexibility in realistic use cases.

Furthermore, we provided an encoding of these techniques using Viper’s built-in features and an implementation that works at the Viper level, meaning it is potentially verifier agnostic. Users can verify fold-like properties about data structures with minimal ghost code, as our plugin automatically performs the necessary decompositions. To our knowledge, no previous work has achieved such an implementation in Viper or any other verification tool.
Chapter 7

Related Work

7.1 Spec#

Spec# [2] is a specification language on top of C# developed by Microsoft Research. It provides predefined fold operations [11], such as sum, count, product, min, and max. In the original paper, these folds were referred to as “comprehensions,” which may explain why similar work has used the same term. However, our project uses the term heap-dependent fold to avoid confusion with list comprehension and set comprehension, which acts like set-builder notation that constructs a new collection from an input collection.

The folds in Spec# are limited only to integer indices, whereas our plugin allows any arbitrary types. The axioms in Spec# also assume some ordering in its support for decompositions. A fold can be decomposed by removing an element at the lowest index or at the highest index. This works for integer ranges but is not scalable to arbitrary types, such as references. This also means Spec# does not directly support arbitrarily picking out a random integer \( i \) in a decomposition, which we do. This is a crucial feature in our implementation because we want to support reasoning for random access data structures, which, by definition, can be accessed at any arbitrary point.

Spec# does have an axiom for decomposing a fold into two folds with two smaller filters. However, this requires manual triggering from the user. Two folds must exist with one high index equal to the other low index,
i.e., they define contiguous parts. The two can then be combined via some preset operator. Our tool instead detects relevant filters according to heap modifications and generates the decomposition manually.

Although the examples presented in the Spec# paper all included arrays, technically, the approach also supports folds that are not heap-dependent. The folds can be on any arbitrary expression that may utilize an integer index. Our tool only supports heap-dependent folds, but we can also rely on heap modifications to pick the appropriate decompositions. Spec# does not use heap reads or writes to axiomatize or decompose its folds.

## 7.2 Hörmann’s Thesis

Tierry Hörmann’s bachelor’s thesis [9] also attempted to implement folds (also called comprehensions, likely because of Spec#) in Viper. Their implementation was in Boogie [10], which allows them to write quantifiers on the heap but also means the support is only available in one of the Viper backend verifiers (that uses Boogie). Instead, our plugin adds code to the original Viper program. Although this should make our plugin more general, in practice the alternative Viper backend, Silicon, is missing some features that our plugin requires, such as heap-dependent quantifier triggers.

One standout feature of Hörmann’s axioms is allowing arbitrary combinations of filters used in folds. Their axiom checks each filter against every other filter for subset or disjoint relations. If the subset relation holds, the two filters subtract to generate another filter. If they are disjoint, the axioms generate a new filter, which is the union of the two disjoint filters. Although the newly formed filters are prevented from generating more filters, similar to our secondary fold idea, the checks are still very costly. Each of these checks happens for each pair of filters, which leads to at least a polynomial complexity with respect to the number of filters.

Hörmann does perform automatic decomposition based on heap reassignments with the operator :=. However, their analysis shows that the time complexity for singleton decomposition is $O(n^n)$, with $n$ as the number of heap locations modified. We have reduced this to linear by using our local
and intermediate strategy, allowing for modular handling at each heap access.

Unfortunately, his thesis only partially touched on implementation details, and the project code on Bitbucket has been removed. We are unable to reproduce his results for a more detailed comparison.

7.3 Ter-Gabrielyan’s Dissertation

In his Ph.D. dissertation [14], Ter-Gabrielyan formalizes folds, which he referred to as comprehensions. In doing so, he defines snapshot maps, which we adapted in our implementation.

The input to their folds is a set of references, in contrast to a filter and a receiver function in our implementation. This allows them to circumvent the injectivity problem because each reference in a set of references is unique by properties of sets. In practice, however, this makes defining the input set more complicated and unnatural. The user would often need to declare a set and write additional axioms using some base indices and a receiver anyway. Furthermore, this design does not align with the design for quantified permissions, which we use extensively.

Ter-Gabrielyan’s folds are not exclusively heap-dependent, allowing a more general body in the fold expressions. In an interesting example, Ter-Gabrielyan proposes nested folds to reason about matrices. However, they do not discuss the automatic decomposition of the folds in response to heap updates. It is unclear how to reconcile the fold’s heap-dependent and non-heap-dependent components, especially when certain components may interact with each other.
Chapter 8

Conclusion and Future Work

We have designed a plugin for supporting heap-dependent folds in Viper. This allows the user to reason about properties of random access data structures without relying on induction. Folds are decomposed automatically in response to heap reads and updates, namely reassignment, exhales and inhales.

In addition, we introduce a modular strategy to reason about each heap change locally, reducing coupling between folds at different states. We solved the combinatorial explosion resulting from repeated decomposition and reduced the number of generated folds to $O(n)$, with $n$ as the number of heap updates. We implemented this as a Viper plugin, only modifying the Viper input. This should make the plugin compatible with any front-end and verifier, though only the Carbon verifier works with it in practice due to limitations in the Silicon verifier.

8.1 Future Work

8.1.1 Implementation in the Verifier

Further research could attempt to implement fold supports at the verifier level instead of the Viper level. More fine-grain control of the calls to the SMT solver could lead to more flexibility. It may be possible to optimize the
generated axioms or quantifiers that have high-performance costs.

Furthermore, we relied heavily on interactions with the heap, which could be done more explicitly in Boogie. Implementing folds using the Carbon verifier, which translates Viper into Boogie, could make the heap-dependent components of the fold easier to handle.

8.1.2 Heap-dependent Triggers

Our axioms rely heavily on heap-dependent triggers despite a scarcity of examples illustrating their behaviour. From our own experiments, much of the undocumented behaviour is unexpected, i.e., some axioms trigger when they are not supposed to, or some axioms do not trigger when they should. For example, triggers that mention a later state never instantiate the axiom. Further research into state-dependent triggers and refinements to the Boogie encoding of the heap-dependent functions may help.

8.1.3 Optimization

Some of our experiments showed surprisingly poor performance, but we lack the tools to properly investigate the causes of these performance drops. Though the Axiom Profiler [3] exists, we are still missing tools to visualize the proofs for individual assertions. Research into better debugging tools could help us examine the relevant axioms and instances for each assertion; this way, we can also isolate the performance hogs and remove them if possible.

8.1.4 Syntax Improvements

With our plugin, a component of a fold has to be individually declared explicitly at the top level. Ideally, a fold can be defined with inlined anonymous functions, as commonly found in general programming languages. Further research could extend the techniques of defunctionalization to support inlined function declarations, which could be useful to Viper overall, not just for expressing folds.
8.1.5 Support for General Folds

Our plugin supports heap-dependent folds, not arbitrary folds. It cannot express the sum of an arbitrary set of integers. Further research could examine the axiomatizations to support general folds. In particular, it is unclear how to reconcile the techniques for non-heap-dependent and heap-dependent folds. Further research can investigate the appropriate decompositions in response to heap reads, inhales, exhales, and reassignment for a fold containing both heap-dependent and independent portions.
Bibliography


