Optimisation of a Deductive Program Verifier

Bachelor’s Thesis

Philippe Voinov
voinovp@ethz.ch

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

Supervisors:
Prof. Dr. Peter Müller
Alexander J. Summers
Malte Schwerhoff

April 25, 2019
Abstract

Verifying software often requires the use of tools which support many advanced features. To efficiently develop such verification tools, it is common to build them in layers on top of existing software. For example, many verification tools utilize SMT solvers as a lower layer to discharge proof obligations. Carbon, part of a suite of verification tools developed at ETH Zürich, is built in such a layered way.

A verification task undergoes many transformations as it gets passed through all of the layers inside and before Carbon. Sometimes a small change in an input program can cause a very large increase in total verification time, because of the way that the transformations interact with each other.

In this thesis we make changes to some of the core transformations in Carbon and measure their effect on verification time. We also present a simple input program which takes surprisingly long to verify in Carbon and discuss the possible cause. Additionally, we present the tools which we developed to help understand such performance issues.
Abstract

1 Introduction

2 Background
  2.1 Viper
  2.2 Boogie
  2.3 Boogie program generated by Carbon

3 Alternate shape for heaps and masks

4 Per-field heaps and masks
  4.1 “Legacy” concept
  4.2 Changes to axioms
    4.2.1 Legacy and generic copies
    4.2.2 Legacy and concrete split
  4.3 Changes to method bodies
  4.4 Unsupported features
    4.4.1 Allocation encoding
    4.4.2 Successor heaps

5 Prusti issues
  5.1 Motivation
  5.2 Isolating an issue
  5.3 Variations of the synthetic issue
  5.4 Understanding the synthetic issue
  5.5 Possible solutions
    5.5.1 Annotating predicates
    5.5.2 Store permissions differently
CONTENTS

A  Blacklisted files  A-1
B  Prusti-related Viper files  B-1
C  Prusti synthetic benchmark prototypes  C-1
Chapter 1

Introduction

Viper [1] is a set of tools for software verification developed at ETH Zürich. It supports permission-based reasoning, which is useful for verifying concurrent and heap-manipulating programs. Input is given using Viper’s own intermediate language.

Users of Viper occasionally create inputs which take unusually long to verify. The reasons for bad performance are often not immediately clear to the user. Though performance issues definitely exist with certain kinds of inputs, there hasn’t been a centralized survey and classification of these inputs yet.

Viper currently has two backends to perform verification: Silicon - a verifier based on symbolic execution, and Carbon - based on verification condition generation. This project will focus on investigating and improving verification time in Carbon.

In chapter 2 we give some background on the structure of the Viper project. We also introduce Boogie [2], the verification tool underlying Carbon. We then present some of the transformations which Carbon performs.

One of the core transformations performed by Carbon is encoding Viper’s built-in heap in Boogie. In chapters 3 and 4 we discuss the significance of this transformation and present possible alternative encodings. We evaluate the performance of these encodings in section 8.5 of the benchmarking chapter.

Chapter 5 explores the structure and behavior of a certain class of Viper input programs. Specifically, these are the programs generated by Prusti [3], a verifier for the Rust language. These programs are interesting since many of them take too long to verify with Carbon to be practical, although the other Viper backend (Silicon) verifies them quickly. We present an interesting synthetic example inspired by Prusti. We then present prototype improvements to Carbon and evaluate them in section 8.6 of the benchmarking chapter.

Chapters 6 and 7 develop some of the tools we developed and used for understanding performance issues in Carbon. The tool presented in chapter 6 is a new benchmark runner, which we configured to automatically measure the performance of new versions of viper. In chapter 7 we present how we instrumented
1. INTRODUCTION

Viper to extract detailed timing traces which show how much time is spent at various levels of the verification stack.

In the benchmarking chapter (8), we present all of the benchmarks which we performed and how we evaluated the results.
2.1 Viper

Viper is a set of tools for software verification. Figure 2.1 shows the components of Viper, as well as their dependencies. It also shows some of the available Viper frontends, which are discussed further below.

![Figure 2.1: Components of Viper and related software (from the Viper tutorial [4])] (image)

Most tools in the Viper ecosystem use the Viper intermediate language, either as their input or output format. We assume that the reader has written and verified some small Viper programs. They should be familiar with methods, references, permissions and predicates in the Viper language, as covered by the Viper tutorial [4].
Programs in the Viper language are often not written by hand, but are instead generated by one of the frontends. Frontends usually convert programs from a different programming language with extra annotations into Viper programs. We will not discuss frontends in detail. It is sufficient to note that each frontend generates code which uses Viper features in different amounts and combinations.

Once a Viper program has been parsed and typechecked, it is given to one of the backends, which perform all further verification. For example, backends check the correctness of pre-conditions, post-conditions and invariants. They also ensure that sufficient permission is available where required. Viper has two back-ends: Silicon and Carbon. Silicon performs symbolic execution and uses the Z3 SMT solver [5] to discharge proof obligations. Carbon converts a Viper program into a Boogie program, which is then verified by Boogie. Boogie generates multiple verification conditions, which are then checked by Z3. We focused only on the performance of Carbon, not Silicon. The following sections describe Carbon and Boogie in more detail.

2.2 Boogie

Boogie is the program verification tool which is used by Carbon. Boogie takes programs written in the Boogie intermediate verification language (previously known as BoogiePL) as input. The language is described in detail in its own paper [6].

Though there are many differences between the Boogie and Viper languages, the most important differences are that Boogie does not have a built-in concept of a heap or permissions.

2.3 Boogie program generated by Carbon

The Boogie programs generated by Carbon all have the same general structure:

- Preamble sections - global variables, functions and axioms
- Translated Viper fields - Boogie constant for each Viper field
- Translated predicates - types and helper functions for each Viper predicate
- Translated methods - Boogie procedure for each Viper method

The preamble sections generated by Carbon generally do not depend on the input Viper program. Since Boogie does not have the concept of a heap built in, the main task of the preambles is to define the heap and related structures. Carbon defines the heap as a map-typed Boogie global variable. This is a map
with two keys (an object reference and a field name) and some type of value (depending on the field accessed). The relevant section of the heap preamble is as follows:

```boogie
1 type Ref;
2 const null: Ref;
3 type Field A B;
4 type NormalField;
5 type HeapType = <A, B> [Ref, Field A B]B;
6 var Heap: HeapType;
```

Note that references are not a built-in concept in Boogie, but are declared by Carbon. As mentioned, the type of a value stored in the heap-map depends on the field being accessed. This is expressed by the type parameter "B" which is part the type of each field constant. For example a Viper field called "count" with type "Int" will be translated by Carbon as follows:

```boogie
1 const unique count: Field NormalField int;
```

This constant represents the Viper field at the Boogie level and is used as the second key to access the heap (along with an object reference). Note that the type parameter "A" is always "NormalField" for fields translated from Viper. Different values for "A" are used for saving things other than values of object fields on the heap-map (for example information about predicates). Some of those other uses are covered later in this section.

Most of the generated Boogie program (except for the preamble) depends on the Viper program being translated. We will use the following example from the Viper tutorial [4] to illustrate the details of this translation:

```boogie
1 field left: Int
2 field right: Int
3
4 predicate tuple(this: Ref) {
5   acc(this.left) && acc(this.right)
6 }
7
8 method setTuple(this: Ref, l: Int, r: Int)
9 requires tuple(this)
10 ensures tuple(this)
11 {
12   unfold tuple(this)
13   this.left := l
14   this.right := r
15   fold tuple(this)
16 }
```

The "setTuple" method is translated as a Boogie procedure with the following signature:

```boogie
1 procedure setTuple(this: Ref, l: int, r: int) returns ()
2 modifies Heap, Mask;
```
2. Background

The arguments and return values of this procedure correspond to the ones in the Viper method. It is explicitly specified which global variables are modified by the method. "Heap" is the global variable that was declared in the heap preamble. We will explain the purpose of the mask later in this section. In the body of a Boogie procedure Carbon generates multiple Boogiestatements for each Viper statement. For example, here is the Boogie translation of the assignment "this.left := 1."

```boogie
assert {:msg * Assignment might fail. There might be insufficient
    permission to access this.left. (tutorial-tuple.vpr@13.3) [14]*}
  FullPerm == Mask[this, left];
2 Heap[this, left] := 1;
3 assume state(Heap, Mask);
```

The heap access itself is on line 2 of the Boogie snippet, where "this" is a Boogie local variable of type "Ref", "left" is a field constant (described above) of type "Field NormalField int", and "1" is a local variable (argument of the Boogie procedure) of type "int". The assertion on line 1 corresponds to a verification error which could possibly be reported by Carbon. Carbon parses the assertion failures from Boogie and reports them as Viper errors. This specific assertion checks that write permission for the field "this.left" is present at this point in the Viper method. Carbon stores the current permission amount for each field in the program in a Boogie map called the "mask". The mask is defined in the preamble generated by Carbon:

```boogie
type Perm = real;
type MaskType = <A, B> [Ref, Field A B]Perm;
var Mask : MaskType;
const NoPerm : Perm;
axiom NoPerm == 0.000000000;
const FullPerm : Perm;
axiom FullPerm == 1.000000000;
```

Unlike the heap, which stores a different types of value for each field, the mask always stores "Perm" values. These are real numbers which encode a permission amount (where 1 is write permission to a field). Almost every translated Viper statement reads or writes the mask.

A heap, mask and set of local variables describe the whole state of a Viper method. The fact that a specific version of a heap and mask pair are part of the same state is captured by the "assume state(...)" statement on line 3 of the translated Viper assignment. This is used for defining certain axioms, as we will show later.

Carbon currently generates Boogie programs in a similar way as its predecessor [7]. Many details in that paper are relevant for understanding the encoding used by Carbon.
Alternate shape for heaps and masks

Two of the most important differences between the Viper language and Boogie are the heap and the concept of permissions. Boogie has no built-in heap. Carbon generates a Boogie map to store the values of object fields. It also generates a second map with similar structure - the mask - which stores the current amount of direct permissions to a field.

The following is a short Viper program and the relevant pieces of it’s Boogie translation.

```viper
field f_int : Int
field f_ref : Ref

method example (x : Ref) {
  inhale acc(x.f_int)
  inhale acc(x.f_ref)
  x.f_int := 2
}

var Heap : HeapType;
type HeapType = <A, B> [ Ref , Field A B]B;
var Mask : MaskType;
type MaskType = <A, B> [ Ref , Field A B]Perm;

procedure example () returns () modifies Heap, Mask {
  Heap[x, f_int] := 2;
}
```

There has been some work comparing different heap encodings [8] using maps in Boogie, which showed significant performance differences between different encodings. That paper compared the performance of the following types of heaps in Boogie:
3. Alternate shape for heaps and masks

- Linear heap: $h[dot(o, f)]$
- State-based linear heap: $h[dot2(o, f)]$
- Synchronous heap: $h[o, f]$
- Two dimensional heaps: $h[o][f]$ and $h[f][o]$
- Field heaps: $H_f[o]$

The notation on the right corresponds closely to how an access of field f in object o could look in our implementation. We do not consider the linear heap and state-based linear heaps. The synchronous heap is the style currently used by Carbon and shown in the Boogie program above. Field heaps are treated separately in the section “Per-field heaps and masks”.

Changing the heap shape to $h[o][f]$ in the Boogie example above would change the following lines:

```boogie
1 type HeapType = <A, B> [Ref] [Field A B]B;
2 Heap[x][f_int] := 2;
```

Similar changes have to be made anywhere the heap or mask is used in the generated Boogie program, including in axioms.

Most heap accesses generated by Carbon are created by the “lookup” method (now “heapLookup”) in the heap module. A similar method in the perm module handles mask accesses. We adjusted both of these methods to use the desired heap and mask shapes. We also added similar methods for accessing predicate masks and wand masks. Since all of these methods perform similar tasks, we added a “HeapLikeShapeHelper” class to prevent code duplication. This helper class also generates the Boogie type definitions for these maps. We made the heap shape configurable via a command line argument. Our changes to Carbon were quite minor, as its modular design worked well in this case. Except for a few locations where Boogie “MapSelect” expressions were created manually (instead of through the lookup methods), these were all the changes that were required to get Carbon working correctly with all tested examples.
Chapter 4

Per-field heaps and masks

As mentioned in “Alternate shape for heaps and masks”, it is possible to split the heap and store it using one map per field. This is possible in Viper since fields are declared statically. The example Viper program from that chapter could be encoded as:

```viper
var LegacyHeap : LegacyHeapType;
type LegacyHeapType = <A, B> [LegacyField A B]B;
type HeapType T = [Ref]T;
var LegacyMask : LegacyMaskType;
type LegacyMaskType = <A, B> [LegacyField A B] Perm;
type MaskType = [Ref] Perm;
var Heap_f_int : HeapType int;
var Mask_f_int : MaskType;
var Heap_f_ref : HeapType Ref;
var Mask_f_ref : MaskType;
```

4.1 “Legacy” concept

The heap generated by Carbon is mainly used to store the values of object fields. However other things, for example predicate masks, are also stored in the heap map. This is done for convenience, since everything in the heap map is often used in similar ways in axioms. These non-object-field things are usually stored “on the null object” and with a “fake” field. For example this is how permissions for a predicate “p(self)” would be stored on the mask:

```viper
type PredicateType_p;
function p(self: Ref, next: Ref): Field PredicateType_p FrameType;
Mask[null, p(x, x)] := Mask[null, p(x, x)] + FullPerm;
```

Since the mask is used with something which is not a normal Viper field in this example, it is not immediately obvious how this example should be adjusted to support per-field heaps and masks. One option would be to also create a
separate mask for each of these things which may be stored on the heap, for example:

```plaintext
var Mask_p = MaskType_p;
type MaskType_p = [Ref , Ref] Perm;
Mask_p[x , x] := Mask_p[x , x] + FullPerm;
```

Note that each argument of the predicate becomes a key of its new mask. Also, each mask in the Boogie program could potentially have a different type (not just a different instantiation of a generic type).

Another approach is to create per-field heaps for normal Viper fields, but to also leave the “old” heap in place. This approach requires significantly less complicated changes to the generated Boogie program, since since any axioms or heap usages in procedure bodies remain the unchanged, unless they work with normal Viper fields. We called this heap the “legacy heap” and prefixed all the functions, types, etc. which work with it in Boogie with “legacy”. Since no normal fields are stored on the legacy heap after our changes, the map key corresponding to a Viper object reference will always be “null”. We removed the object key, making the legacy heap a single-keyed map. The same changes were made to the mask.

### 4.2 Changes to axioms

Unlike changing the shape of the heap and masks, splitting the heap per field requires more significant changes to the axioms generated by Carbon. In this section we show examples of the most representative or interesting kinds of changes.

#### 4.2.1 Legacy and generic copies

One of the most common changes was to make two copies of an axiom: a legacy version and a generic per-field copy. We take “HasDirectPerm” as an example. It checks if there is direct permission (not through a folded predicate) to access a given object field in a given state. The original axiom is as follows:

```plaintext
function HasDirectPerm< A, B>(Mask : MaskType , o : Ref , f : (Field A B)) : bool;
axiom (forall <A, B> Mask : MaskType, o : Ref , f : (Field A B) ::
{ HasDirectPerm(Mask, o , f) } 
HasDirectPerm(Mask, o , f) <==> Mask[o , f] > NoPerm
);
```

We replace it with a legacy copy of the same axiom by adding “legacy” prefixes and removing the object reference (since it is always “null”):
4. Per-field heaps and masks

```plaintext
function LegacyHasDirectPerm< A, B>({LegacyMask : LegacyMaskType , f : LegacyField A B}) : bool;
axiom (forall < A, B > LegacyMask : LegacyMaskType , f : LegacyField A B ::
   start { LegacyHasDirectPerm (LegacyMask , f ) }
   end LegacyHasDirectPerm (LegacyMask , f ) <-> LegacyMask[f] > NoPerm
);
```

We define the per-field copy of this axiom only once. It can be used for any Viper field.

```plaintext
function HasDirectPerm (Mask : MaskType , o_1 : Ref ) : bool ;
axiom (forall Mask : MaskType , o_1 : Ref ::
   start { HasDirectPerm (Mask , o_1) }
   end HasDirectPerm (Mask , o_1) <-> Mask[o_1] > NoPerm
);
```

Note that we arrive at this copy of the axiom by removing the field variable from the original axiom, since it is determined by the given mask.

It is very straightforward to add support for per-field heaps for these kinds of axioms. However it is not possible to convert all axioms in this way, as demonstrated in the next section.

### 4.2.2 Legacy and concrete split

One of the most important functions generated by Carbon is “IdenticalOnKnown-Locations”. It ensures that two heaps are identical on every location for which direct or indirect permission exists in a given mask. There are multiple axioms which define the function, which we will not list here. The original signature of the function is as follows:

```plaintext
function IdenticalOnKnownLocations( Heap : HeapType , ExhaleHeap : HeapType , Mask : MaskType ) : bool ;
```

The natural way to extends this function to per-field heaps is to pass all the heaps as consecutive arguments. The same applies to masks. Specifically, assuming fields “f_1” (Int), “f_2” (Ref), etc.:

```plaintext
function IdenticalOnKnownLocationsAllFields( LegacyHeap : LegacyHeapType , Heap_f_1 : HeapType int , Heap_f_2 : HeapType Ref , // ... LegacyExhaleHeap : LegacyHeapType , ExhaleHeap_f_1 : HeapType int , ExhaleHeap_f_2 : HeapType Ref , // ... LegacyMask : LegacyMaskType , Mask_f_1 : MaskType , Mask_f_2 : MaskType , // ... ) : bool ;
```

When defining axioms for this function, one would have to mention the heap and mask for every field as a quantified variable. Additionally, any axiom that
applies independently to each field would have to be defined multiple times (once for each field). To avoid this and to simplify the conversion of axioms, we create additional functions and delegate the individual fields to them:

```plaintext
IdenticalOnKnownLocationsAllFields( /* ... */ ) =>>

LegacyIdenticalOnKnownLocations(
    LegacyHeap, LegacyExhaleHeap, LegacyMask) &&

IdenticalOnKnownLocationsOneField(
    Heap_f_1, ExhaleHeap_f_1, Mask_f_1) &&

IdenticalOnKnownLocationsOneField(
    Heap_f_2, ExhaleHeap_f_2, Mask_f_2) &&

IdenticalOnFoldedLocationsOneField(
    LegacyHeap, LegacyExhaleHeap, LegacyMask,
    Heap_f_1, ExhaleHeap_f_1, Legacy_f_1) &&

IdenticalOnFoldedLocationsOneField(
    LegacyHeap, LegacyExhaleHeap, LegacyMask,
    Heap_f_2, ExhaleHeap_f_2, Legacy_f_2)
```

The axioms for many of these functions can now be converted using the procedure described in “Legacy and generic copies”. An interesting exception is “IdenticalOnFoldedLocationsOneField”. It ensures that for each location in given per-field heaps, the value at that location is equal if a folded predicate gives permission to that location. Permissions which are folded inside predicates are stored on the predicate mask. The predicate mask is in turn stored on the legacy heap. As previously described, the predicate mask still has two keys (object reference and legacy field constant). It now becomes necessary to know which per-field mask belongs to which legacy field constant. This association happens during the split of “IdenticalOnKnownLocationsAllFields”, as shown above. This avoids having to declare a separate (non-generic) axiom for each field.

### 4.3 Changes to method bodies

The necessary adjustments for method bodies are quite straightforward. Heap and mask lookups are adjusted in a similar way to what is described in “Alternate shape for heaps and masks”. Also, any statement which uses a heap or mask without a lookup is typically duplicated once for every field, as in the following example:

```plaintext
Mask := ZeroMask;
```

In the per-field version of Carbon this becomes:

```plaintext
LegacyMask := LegacyZeroMask;
Mask_f_1 := ZeroMask;
Mask_f_2 := ZeroMask;
```
4.4 Unsupported features

4.4.1 Allocation encoding

Allocation encoding lets Carbon track whether an object is currently allocated. This information is stored on the heap as follows:

```plaintext
const unique $allocated: Field NormalField bool;
axiom (forall o: Ref, f: Field NormalField Ref, Heap: HeapType ::
   { Heap[o, f] } 
   Heap[o, $allocated] => Heap[Heap[o, f], $allocated]
```

This feature allows Carbon to determine, for example, that a reference passed as a method argument can not be equal to a new reference created within that method. One way to support this feature in the modified version of Carbon would be to keep the "object" key in the legacy heap and mask, instead of removing it as described at the beginning of this chapter. Allocation encoding would then be the only remaining feature which uses the second key.

4.4.2 Successor heaps

The original Carbon contained the following functions:

```plaintext
function succHeap(Heap0: HeapType, Heap1: HeapType): bool;
function succHeapTrans(Heap0: HeapType, Heap1: HeapType): bool;
```

This “successor heap” relation defines an order for heaps in Carbon. A heap HB is a successor of a heap HA if HB was created by updating one entry in HB, or if HB was created by copying HA using “IdenticalOnKnownLocations”.

We attempted to convert these functions to support per-filed heaps by passing the legacy heap and each per-field heap as consecutive arguments, as for “IdenticalOnKnownLocationsAllFields”. While this could generally work, a problem arises when trying to convert the following axiom:

```plaintext
axiom (forall <A, B> Heap: HeapType, o: Ref, f: Field A B, v: B ::
   { Heap[o, f := v] } 
   succHeap(Heap, Heap[o, f := v])
)
```

When converting this axiom to per-field heaps as multiple arguments of “succHeap”, each heap would be mentioned as a quantified variable. This means that each heap must also be mentioned in the trigger. However there is nothing restricting which heaps are used to instantiate the quantifier (other than their type). This means that the quantifier may be instantiated with a heap for field “f_1” from early in a method body, combined with a heap for field “f_2” from much later in the method body. This not only does not make sense for the definition of successor heaps, but would also perform very badly.
4. **Per-field heaps and masks**

Another way to convert this axiom is to require that the heaps in the quantified variables occur together in a application of “state”. This is a Boogie function generated by Carbon which is applied after some operation changes the heap or mask. This would slightly change the semantics of “succHeap”, since if a heap is modified twice before calling “state”, only the second version of that heap will be considered a successor of the original.

A third possibility to convert “succHeap” is to modify it as described in “Legacy and generic copies”. This would make it impossible to relate the heaps for one field with those for another field using “succHeap”.

Successor heaps are only used in Carbon in one axiom to optimize the performance of quantified permissions. We chose to not support successor heaps in the per-field version of Carbon, and to disable them in the other versions of Carbon during benchmarking.
5.1 Motivation

Prusti [3] is a Viper frontend for the Rust programming language. It generates Viper programs by combining explicit annotations (e.g., preconditions) added by the user with information from the Rust type system. The developers of Prusti normally only verify the Viper programs generated by Prusti with Silicon, since verification with Carbon is only fast with very small examples. We attempted to understand what was causing performance issues with Viper programs generated by Prusti.

5.2 Isolating an issue

We first tried to verify one of the smallest non-empty Rust programs:

```rust
extern crate prusti_contracts;

fn main() {
    assert!(true);
}
```

This example takes around 1 second to verify on our benchmark machine. The Viper program generated by Prusti for this example contains around 128 lines, 1 method, 8 predicate definitions, 6 fields, 12 local variables, 2 assertions, 10 inhale statements and 1 exhale statement. We will omit the "extern crate" line from now on, since it is in all Prusti examples.

We then looked for an example which would be slow to verify. We found the following program from the Prusti test suite:

```rust
fn main() {
    let x1 = 7;
    let x2 = 7;
    let y = 9;
    assert!(x1 == x2);
    assert!(x1 != y);
}
```
This example takes around 250 seconds to verify on the same machine. For comparison, Silicon takes around 1 second to verify this example. Clearly using Carbon together with Prusti for interactive verification is currently not practical.

The generated Viper program contained 247 lines, 1 method, 9 predicate definitions, 6 fields, 31 local variables, 3 assertions, 26 inhale statements and 1 exhale statement. The Viper program for this example is not extremely large compared to the first example.

We tried modifying the slow Prusti example above in various ways to try to identify which Rust code causes slow Viper code to be generated. The following Rust program takes 23 seconds to verify with Carbon:

```rust
def main() {
    let x = 7;
    assert!(x == 7);
    assert!(x == 7);
    assert!(x == 7);
}
```

It is similar to the previous example, but uses less variables. We observed that adding more assertions to this example significantly increased the verification time.

The most similar fast example that we found was the following:

```rust
def main() {
    assert!(true);
    assert!(true);
    assert!(true);
}
```

This takes around 1 second to verify. Increasing the amount of assertions also slightly increases the verification time, but much less than with the example with a variable.

We then proceeded to examine the generated Viper program for the example with a single local variable. The full Viper program can be found in the appendix B (assert-after-set-same-3x.rs.vpr). Prusti generates Viper methods based on the control flow graph of that method, which is provided by the Rust compiler. The main part of the Viper method body consists of labeled blocks of code, which are linked together using goto statements. At the start of the Viper method body, the low-level storage locations used in the CFG representation of the method are allocated. An example of an allocation and assignment to such a location follows.

```viper
var _1: Ref
inhale acc(i32(_1), 1 / 1)
unfold acc(i32(_1), 1 / 1)
_1.val_int := 7
```
5. Prusti issues

These lines were extracted from the full program generated by Prusti. They appear in this order in the method body, but with other lines in between.

We modified the Viper file for this example by hand in various ways. We applied each modification in order, building on the results of the previous modification. We list the approximate verification time for the full modified program after each modification in brackets. As mentioned, the verification time for this example was initially 23 seconds. The modifications were as follows:

- We removed any empty blocks which were connected by goto statements, and changed the targets of the remaining goto statements to maintain the same control flow. This did not significantly change verification time (23s).

- We split each block of code into separate methods, instead of connecting them using goto. We declared all Viper variables only in the methods where they were used. Shared variables were passed between methods as arguments and return values. We manually added pre- and postconditions, which likely makes it easier for Carbon to verify the program. These changes significantly reduced verification time (7s).

- We removed chains of assignments to local Viper variables ("b := a; c := b;") of type Ref. We also removed the declarations of these now-unused variables, as well as any inhale and unfold statements with the corresponding predicates (but not the predicate definitions). The verification time was slightly reduced (6s).

- We removed all temporary boolean Viper variables. Instead we combined any boolean expressions and moved them to the locations where they were used (conditions of if-statements). We also removed all calls to now-empty methods. The verification time was significantly reduced (2s).

- We removed all now-unused predicate declarations and definitions, since the temporary variables which used them were previously removed. The verification time was not significantly reduced (2s).

- We recombined all methods back into one single method using goto statements. We removed all explicit pre- and post-conditions which we added to the methods in a previous step. The verification time was reduced (less than 1s).

- We added back all unused temporary variables, including the corresponding predicates and inhale statements, without using them. None of these added variables are ever read or assigned to. None of the inhaled predicates are ever unfolded. The verification time was slightly increased (1s).

The final version of the example with all modifications can be found in the appendix B (assert-after-set-same-3x.rs-equiv-aggressive-goto-useless-inhales.vpr).
It contains as many local variable declarations, predicate definitions and inhale statements as the original version. However, it contains significantly less unfold statements, assignments and goto statements. By inspecting the differences between these two versions, we created the following Viper program for synthetic benchmarking:

```viper
type val_int: int

predicate i32(self: Ref) {
  acc(self.val_int)
}

method example() {
  var _1: Ref
  var _2: Ref
  var _3: Ref
  inhale i32(_1)
  inhale i32(_2)
  inhale i32(_3)
  unfold i32(_1)
  unfold i32(_2)
  unfold i32(_3)
  _1.val_int := 7
  _2.val_int := _1.val_int
  _3.val_int := _2.val_int
  assert _3.val_int == 7
}
```

This synthetic example uses multiple reference-typed local variables. It gains access to a field through these references indirectly, by inhaling and unfolding a simple predicate. A constant value is copied in a chain through the field of each object.

The example can be naturally extended to any size, where \( n \) is the number of variables, inhales, unfolds and assignments. By this definition, the version shown above is for \( n=3 \). This synthetic benchmark becomes very slow to verify in Carbon with relatively small \( n \) (107s for \( n=10 \)). The verification time grows very quickly (possibly exponentially in \( n \)). Silicon does not exhibit the same slowdown. For full benchmarks, see section 8.6 in the benchmarking chapter.

### 5.3 Variations of the synthetic issue

To better understand what causes the long verification time of the above synthetic example in Carbon, we created multiple variations of it and compared
their verification time. The modifications listed below were all applied to the unmodified version of the example, not to the result of the previous modification.

- Gain field access directly instead of using predicates
  \((\text{ref} - \text{int} - 1 \times n, \text{much faster})\)
- Interleave in different ways
  - \((n \times (\text{var}, \text{inhale}, \text{unfold}, \text{assign})), \text{assert } (\neg a, \text{much faster})\)
  - \((n \times \text{var}), (n \times (\text{inhale}, \text{unfold}, \text{assign})), \text{assert } (\neg b, \text{much faster})\)
  - \((n \times \text{var}), (n \times \text{inhale}), (n \times (\text{unfold}, \text{assign})), \text{assert } (\neg c, \text{similar to baseline})\)
  - \((n \times \text{var}), (n \times (\text{reversed inhal}e), (n \times (\text{unfold}, \text{assign})), \text{assert } (\neg c - \text{reverse inhal}e, \sim 2 \times \text{as fast as baseline})\)
  - \((n \times \text{var}), (n \times (\text{inhale}, \text{unfold})), (n \times \text{assign}), \text{assert } (\neg d, \text{much faster, but slightly slower than interleave } (\neg a)\)
  - Only use 2 of the objects for assignment chains and still inhale all predicates, but don't unfold them \((\sim \text{less unfolded, much faster})\)
  - Inhale and unfold everything, but only assign to the first object and assert on that \((\text{no assignment chain}) (\sim \text{one assign, similar to baseline})\)
  - Don't do any assignments at all \((\sim \text{no assign}, 3 \times \text{slower than baseline})\)

Of the interleavings presented above, the ones where each unfold was directly followed the corresponding inhale were significantly faster than those where this wasn’t the case. Additionally the amount of assignments did not seem to be important, except when there were no assignments at all.

5.4 Understanding the synthetic issue

Together with Alex Summers we discussed the synthetic benchmark and some of the variations presented here. We arrived at the following possible explanation for the verification time of the synthetic benchmark. We will present it as a step-by-step explanation of the evaluation of a Viper program at the Boogie level. Boogie does not actually verify programs in the way we present, but instead generates verification conditions which are evaluated by Z3. However, this explanation should help illustrate the issue as we understand it, and likely has at least some relation to the challenges which Z3 faces while evaluating the VCs generated by Boogie.

We assume that Boogie uses a strategy similar to the following common axiomatization to evaluate maps. Given a map \(m\), keys \(k_1\) and \(k_2\), and a value \(v\), then one of the following holds:

- If \(k_1 = k_2\), then \(m[k_1 := v][k_2] = v\)
- If \(k_2 \neq k_2\), then \(m[k_1 := v][k_2] = m[k_2]\)
In the following text we use $p$ for the predicate "i32", $x_1$ for the variable "_1", $M$ for the mask generated by Carbon. Before performing the first unfold (line 16 of the Viper program for $n = 3$), it must be established that there is enough permission to unfold the predicate. We will write this condition as $M_n[p(x_1)] >= 1$ (note that the actual mask generated by Carbon has a pair of keys, but when used to store permission to predicates, the first key is always null, so we omit it in this explanation). We use the notation $M_n$, since the current version of the mask at this execution point is the empty mask with $n$ mask updates applied (one for each inhale). Specifically, let $M_0$ be the initial empty mask, and let $M_i$ (the mask after the $i$-th inhale) equal $M_{i-1}[p(x_i)] := M_{i-1}[p(x_i)] + 1$. We assume that to check whether $M_n[p(x_1)] >= 1$ holds, the left side of the expression has to be evaluated. By definition of $M_i$, we have $M_n[p(x_1)] = M_{n-1}[p(x_n)] := M_{n-1}[p(x_n)] + 1[p(x_1)]$. We then apply the map axiom from above to get one of the following expressions:

- If $k_1 = k_2$, then $M_{n-1}[p(x_n)] + 1 >= 1$
- If $k_2 \neq k_2$, then $M_{n-1}[p(x_1)] >= 1$

In the first case, we can conclude that the expression is true, since $M_{n-1}[p(x_n)] >= 0$ is given by the “GoodMask” axiom. In the second case, we must use the same evaluation process $n - 1$ more times. At each of these evaluation steps will have to consider two cases similar to those above (except for the last step, since $p(x_1) = p(x_1)$).

After we have considered all possible evaluations, we now have $2^{n-1}$ different cases. In each of these cases, our assertion has the same result (there is enough permission to unfold the predicate). However these $2^{n-1}$ cases can not necessarily be merged, since the permission for $p(x_1)$ is different in some cases. For example at the first evaluation step we have either $M_n[p(x_1)] = M_{n-1}[p(x_n)] + 1 = M_{n-1}[p(x_1)] + 1$, or $M_n[p(x_1)] = M_{n-1}[p(x_1)]$.

We then proceed to evaluate the second unfold statement in a similar way to the first. We will also have many different cases for the choices that can be made while evaluating the mask expression. Each of these can be combined with any of the cases from the previous unfold. Every case where $p(x_1) = p(x_2)$ is assumed leads to a contradiction, since it implies that more than 1 unit of permission is held to the “val_int” field of $x_1$.

After all $n$ unfold statements have been evaluated, only one valid case remains which does not lead to a contradiction. This is the case where $i! = j => p(x_i)! = p(x_j)$ for every $i$ and $j$.

In summary, since it is not initially clear which references are disjoint, it is possible to assume any combination of equality between predicates while checking the assertions required for unfolding. There are many possible ways to make
these assumptions, which can not necessarily be combined, since the amount of 
permission held to each predicate may differ. These various cases can only be 
ignored once the inequality of each pair of references is shown, by considering 
further unfold statements in each case and reaching a contradiction.

Although our theory of what causes the slowdown with our synthetic example 
is not necessarily correct, it seems consistent with the performance of different 
variations of our synthetic example.

5.5 Possible solutions

In this section we present some possible ways to improve the performance of your 
synthetic benchmark. The general idea behind all of these approaches is to make 
it easier to establish the disjointness of references. The first two approaches help 
to recognize the disjointness of predicates, which implies that the references they 
use are disjoint (if the two predicates have the same type).

We did not implement these solutions in Carbon. However, we tested each 
of these solutions with our synthetic example by hand-modifying the Boogie 
program which was generated by Carbon for our synthetic example. The results 
are in section 8.6 of the benchmarking chapter. The modified Boogie programs 
for $n = 3$ can be found in appendix C.

5.5.1 Annotating predicates

In Viper it is not possible to hold more than 1 unit of permission to a single 
object field. The same limitation does not apply to folded predicates, since in 
some cases it would unnecessarily limit how predicates can be used. The following 
example would not verify if permission to predicates were limited to 1:

```
field value : Int

predicate half(x : Ref) {
  acc(x.value , 1 / 2)
}

method example() {
  inhale half(x)
  inhale half(x)
  unfold acc(half(x) , 2)
  x.value := 9
}
```

Unlike in this example, unfolding more than 1 unit of most predicates is not 
possible, since that would grant more than 1 permission to a field. Because it 
would not be possible to unfold them, it does not make sense to hold more than 
1 unit of such predicates at all. One approach, which was suggested by Alex
5. Prusti issues

Summers, is to prevent holding too much of such a predicate through explicit annotations:

```plaintext
field left : Int
field right : Int

predicate tuple (this : Ref) limit 1 / 1 {
    acc (this . left) && acc (this . right)
}
```

This would allow Carbon to generate a Boogie axiom which specifies that the mask always contains less than 1 unit of permission to the tuple predicate, similar to the axiom which applies to fields. In our synthetic example, this would make it possible to determine when inhaling a predicate, that it is not equal to any of the already inhaled predicates.

These annotations would also make new uses for predicates possible, for example to model a resource of which there are a finite number of copies known before verification. Adding support for annotations would require a change to the Viper language.

5.5.2 Store permissions differently

In the current Boogie programs generated by Carbon, the amount of information about permissions which are folded inside of predicates is very limited. Carbon keeps a "predicate mask" for each predicate. Predicate masks are shaped like the main mask, but store boolean values instead of real permission values. The relevant Boogie definitions generated by Carbon for the predicate mask of the tuple predicate in the previous section are as follows:

```plaintext
// Preamble of Permission module.
type PMaskType = <A, B> [Ref, Field A B] bool;

// Translation of predicate tuple
type PredicateType_tuple;
function tuple#sm (this : Ref) : Field PredicateType_tuple PMaskType;
```

The "tuple#sm" function is used for accessing the predicate mask for the tuple predicate, which is stored on the heap. "sm" stands for "secondary mask", which is another name for the predicate mask. An empty predicate mask, which contains no permissions, contains false for every field. If an entry in the predicate mask contains true, it means that there is some amount of permission to the given field (at least read permission, but possibly write). If a predicate mask contains false, it does not necessarily mean that there is no permission to the corresponding field. Initially the predicate mask for each predicate is empty. When a predicate is folded, the permissions which get folded into the predicate are added to the predicate mask. For example, when folding an instance of the tuple predicate Carbon will generate the following code after folding:
5. Prusti issues

```java
    Heap[null, tuple#sm(this)]{this, left} := true;
    Heap[null, tuple#sm(this)]{this, right} := true;
```

Carbon uses predicate masks to help define the “IdenticalOnKnownLocations” function (see section 4.2.2). If permission to a predicate is held, and the corresponding predicate mask contains true for some object field, then that field is considered a known location.

Predicate masks do not help to establish the disjointness of references with fields with folded write permissions:

```java
    method example(x: Ref, y: Ref) {
        // Mask { }
        // Predicate mask tuple(x) { }
        // Predicate mask tuple(y) { }
        inhale acc(x.left) && acc(x.right)
        // Mask { x.left: 1, x.right: 1 }
        // Predicate mask tuple(x) { }
        // Predicate mask tuple(y) { }
        fold tuple(x)
        // Mask { tuple(x): 1 }
        // Predicate mask tuple(x) { x.left: true, x.right: true }
        // Predicate mask tuple(y) { }
        inhale acc(y.left) && acc(y.right)
        // Mask { tuple(x): 1, y.left: true, y.right: true }
        // Predicate mask tuple(x) { x.left: true, x.right: true }
        // Predicate mask tuple(y) { }
        fold tuple(y)
        // Mask { tuple(x): 1, tuple(y): 1 }
        // Predicate mask tuple(x) { x.left: true, x.right: true }
        // Predicate mask tuple(y) { y.left: true, y.right: true }
        assert x != y // this will fail
    }
```

The comments in the above Viper method show what is stored in the main mask and each predicate mask. Since the predicate mask does not store whether a folded permission to a field is a read or write permission, there is not enough information for the assertion to determine that x != y.

We propose to store the combined information about known permissions (direct and folded) in one location: the total mask. The total mask would have the same type as the mask. It would store real permission amounts (not booleans). For each object field, the corresponding entry in the total mask contains the sum of all direct and known folded permissions. The previous example would look as follows with a total mask:

```java
    method example(x: Ref, y: Ref) {
        // Mask { }
        // TotalMask { }
        inhale acc(x.left) && acc(x.right)
        // Mask { x.left: 1, x.right: 1 }
        // TotalMask { x.left: 1, x.right: 1 }
    }
```
5. Prusti issues

```
fold tuple(x)
// Mask { tuple(x): 1 }
// TotalMask { x.left: 1, x.right: 1 }
inhere acc(y.left) && acc(y.right)
// Mask { tuple(x): 1, y.left: 1, y.right: 1 }
// TotalMask { x.left: 1, x.right: 1, y.left: 1, y.right: 1 }
fold tuple(y)
// Mask { tuple(x): 1, tuple(y): 1 }
// TotalMask { x.left: 1, x.right: 1, y.left: 1, y.right: 1 }
assert x != y // this will not fail
```

The assertion would now succeed, since if x were equal to y, we would have more than 1 unit of permission to x.left, which is not possible.

When a predicate gets folded, the current version of Carbon updates the relevant predicate mask. For supporting total masks, Carbon would be modified to instead generate code which updates the total mask to add any field permissions which are being folded into the predicate.

If a predicate instance is directly inhaled, instead of being created by folding, then the total mask will not contain any field permissions which are contained inside this predicate. This is because calculating the permissions contained in a predicate is not possible in general (specifically for recursive predicates). One could calculate the permissions which would be gained if the predicate were unfolded once, or recursively up to some constant amount of levels deep. However this would likely lead to more confusing performance for Viper users. We will call this optional feature the “unfolding hack” in the rest of this text, since calculating the permissions stored in a predicate in this non-general way is equivalent to unfolding the predicate after it is inhaled and immediately folding it again.

Though having a total mask can provide extra information in programs like the one above, it would only be helpful in our synthetic example if the unfolding hack is used. This is because the predicate used in the synthetic example is never explicitly folded. As mentioned above, this is not a general approach.

5.5.3 Explicitly specify disjointness

The issues with our synthetic example stem from the fact that the disjointness of the references used can only be determined at the end of the program. In some cases when writing or generating a Viper program, it may be known to the author that certain references are disjoint.

Specifically, this seems to be the case in Prusti. Each reference variable created by Prusti is used together with a predicate to model a low-level storage location. These storage locations are disjoint by definition. By encoding this knowledge more explicitly in the generated Viper program, we could prevent the verifier from performing unnecessary case splits. We can do this by modifying
Prusti to assign a fresh value to each reference variable after their declaration. For example, the start of a method with 3 storage locations would look as follows:

```plaintext
var _1 : Ref
var _2 : Ref
var _3 : Ref

_1 := new()
_2 := new()
_3 := new()
```

The Carbon “allocation encoding” feature must be enabled, otherwise the verifier will not recognize that the variables are disjoint, even though they have each been assigned a fresh value.
As part of this thesis we developed a new system for benchmarking Viper. Our system is designed both to periodically measure the performance of the latest versions of Viper from version control, and for manually conducting experiments. We named this new system “Viper Runner 2”, after the existing “Viper Runner” benchmarking tool [9], which was developed as part of a previous master’s thesis [10]. Our system is significantly different from the original Viper Runner, which makes it better suited for some use cases.

6.1 Motivation and design decisions

6.1.1 Manual use

We used Viper Runner 2 to run most of the benchmarks in this thesis, and to evaluate their results. This motivated many of the features and design decisions in Viper Runner 2. In this section, we will present some of those decisions.

Since Carbon is written in Scala, it runs inside the Java Virtual Machine (JVM). The JVM has a significant startup time (around half a second). This startup time can also be quite inconsistent. The original Viper Runner used Nailgun [11] to avoid the JVM startup time. Nailgun is a client-server system which runs programs in a shared persistent JVM instance. After the original Viper Runner was developed, Viper Server was created. Viper Server is a standalone program which contains Carbon and Silicon as libraries, and provides an API to start and monitor verification jobs over HTTP. Viper Server is currently used by the Viper IDE plugin [12] for Visual Studio Code. We wanted Viper Runner 2 to use Viper Server so that performance measurements would be as similar as possible to those in the Viper IDE. Additionally, using the HTTP API of Viper Server allows us to obtain more structured information from the verification backends, which we used for our detailed timing implementation (see chapter 7).

Most of the benchmarks which we needed to perform involved comparing
multiple versions of Carbon, or the same version with different command line options. Like the original Viper Runner, our system has built-in support for multiple configurations. For our benchmarks we modified not only Carbon, but also Silver, Viper Server and Boogie. This motivated our decision to treat the combination of all Viper components and dependencies as a single versionable unit. In the original Viper Runner, this versioning was not explicit.

We obtained results from around 120000 verifications of Viper programs while working on this thesis. To manage and evaluate this data in a convenient way, we decided to store all verification results in an SQL database. The original Viper Runner writes individual results to a CSV file instead.

6.1.2 Automated use

One of the goals of this thesis was to create a semi-automatic setup for testing Viper verification time. As part of that goal, we specified that our tool should run on a server. Since Viper Runner 2 would run on a shared machine, we developed a browser based UI, which allows multiple people to view results and start jobs simultaneously.

In our goals, we also specified that there should be a mechanism for automatically submitting Viper versions from version control for testing. We integrated Viper Runner 2 with Jenkins to achieve this. We save the versions submitted by Jenkins using the same versioning system as described in the “Manual use” section above.

6.2 Concepts and structures

In this section, we will describe various concepts from Viper Runner 2. These concepts may be helpful to users of the system. Additionally, they help to understand the detailed design of the runner, which is presented in the following section.

Viper Runner 2 has a global collection of Viper files, each with a name and specific content. They are immutable and stored in the main SQL database, not as actual files.

As mentioned in the previous section, Viper Runner 2 has the concept of Viper versions. A Viper version in Viper Runner 2 is defined by a name and a URL of a Docker image. Docker images effectively contain the full filesystem of a Linux system, including all installed applications. Docker images for Viper contain Viper Server with all of its dependencies. For more details on how Docker is used, see the next section.

To start benchmarking in Viper Runner 2, it is necessary to create a “job”.
This is done by specifying which Viper configurations and Viper files to test. A Viper configuration is a specific Viper version, along with a choice of backend (Carbon or Silicon) and any extra command line arguments for the backend.

Jobs in Viper Runner 2 consist of “chunks”. A chunk consists of a single Viper configuration along with an ordered list of Viper files. Typically each chunk contains a small amount of different files (for example 5), which are each repeated multiple times consecutively within the chunk. Repeating files consecutively can allow various systems to warm up (for example the just-in-time compiler in the JVM). Splitting jobs into chunks also makes it possible to interleave the execution of multiple jobs. Each chunk can be attempted multiple times. Chunks are reattempted in case some part of the runner crashes or is shut down.

As Viper Runner 2 executes chunks within a job, it saves verification results. A verification result is the whole result of running a single Viper file once with a single Viper configuration. These verification times contain the output received from Viper Server, including verification errors (for example an error about insufficient permission with details), verification time, and Java or Scala stack traces in case of crashes. If a file occurs multiple times in a chunk, or a chunk is attempted multiple times, then multiple verification results are created.

6.3 Detailed design

Job management and execution are separated in Viper Runner 2. There is a “master” component, which tracks jobs and provides the UI, and a “worker-agent” component, which manages Viper Server via Docker and forwards verification requests. The actual verification and timing is performed by Viper Server.

As previously mentioned, verification results are stored in an SQL database. The master component connects to a PostgreSQL database. All other components of Viper Runner 2 do not have any persistent state.

The worker-agent component starts the required version of Viper in a Docker container. Docker containers use a form of process isolation. Using Docker containers allows us to run any version of Viper with any dependencies, independent of the OS version of the host machine on which Viper Runner 2 runs. It also allows us to quickly switch between different versions of Viper. Running a process inside a container does not incur a significant performance overhead, unlike using virtual machines, so it is suitable for benchmarking.

The master and worker-agent components communicate over gRPC [13] (a standard for remote procedure calls). This allows them to run on separate machines. The master and worker-agent are both written in Go [14]. We chose to use Go since it has a native interface to the Docker daemon, which we use to start a specific version of Viper.
The typical flow for benchmarking verification in Viper Runner 2 is as follows: A user submits the “Create job” form in their browser. The master component finds the appropriate files and creates multiple chunks for the job in the database. A background process in the master finds one of the new unattempted chunks. It sends that chunk, including the content of the Viper files, to the worker-agent. The worker-agent starts the version of Viper which is necessary for this chunk as a Docker container. Once the worker-agent successfully connects to the Viper Server instance which it just started, the worker-agent starts to verify the first file. The worker-agent writes the content of the Viper file to a temporary file on disk, and sends a verification request to Viper Server with the path to this file. Viper Server verifies the file and streams the verification results back to the worker-agent. The worker-agent converts these results and streams them to the master, which saves them to the database. Once all the files in the chunk have been verified, the worker-agent notifies the master. The master marks the chunk attempt as successful. The master then picks the next unattempted chunk and continues verification.

Timeouts are enforced by the worker-agent together with Viper Server. Before stating the verification of a file with Viper Server, the worker-agent starts a timer. Once this timer crosses a certain threshold, the worker-agent asks Viper Server to discard the current verification. We made some adjustments to Viper Server and Carbon to ensure that discarding a job would always stop verification as intended. We also adjusted the start script of Boogie, to ensure that all child processes are stopped (Boogie and Z3) when requested.

For automatic benchmarking of versions of Viper from version control, we integrate with the existing Jenkins CI system of the research group. A periodic Jenkins job builds all of the components of Viper and packages them and their dependencies into a Docker image. The Jenkins job then notifies the Viper Runner 2 master via an HTTP request that a new Viper version is available. The master saves this new Viper version in its database and creates a Viper Runner 2 job to compare it against a reference version of Viper.

6.4 Deployment

We deployed Viper Runner 2 on two virtual machines and one physical machine, which were provided by the research group. We would like to thank Vytautas Astrauskas for providing us with all the required resources and support.

We managed the deployment using Ansible (a configuration management framework). This allows us to very quickly provision new machines if needed.

We used Ubuntu 18.04 as the OS on every machine. Each machine also runs Docker.

One of the VMs runs the Viper Runner 2 master. Specifically, the Viper
Runner 2 master and PostgreSQL both run in Docker containers inside this VM.

The other VM runs a private Docker registry. A Docker registry is a storage server for Docker images. We configured Jenkins to build the Viper Docker image directly on this VM. It is possible to perform the build elsewhere if desired in the future. Once Jenkins has built the image, it uploads it to the Docker registry under a unique URL. This is the URL which Jenkins then sends to the Viper Runner 2 master, which sends it to the worker-agent, which downloads the Docker image from this private Docker registry. This is necessary, since it is not typically possible to directly send or store Docker images without a registry.

The only physical machine runs the Viper Runner 2 worker-agent. It also runs at most one Viper Docker container, which is started by the worker-agent.

The web UI served by the Viper Runner 2 master uses HTTPS, with certificates provided by the research group. The Docker registry also uses a similar certificate. The master and worker-agent have self-signed certificates from the same CA, which they use for mutual authentication and encryption when communicating.
Carbon primarily encodes an input for Boogie and then interprets the results, so ideally only a small portion of the total verification time for a file would be spent in Carbon itself. We wanted to check whether this was the case, and to better understand which parts of Carbon or Silver could be optimized. We built a solution for collecting and viewing detailed timing information from Viper.

7.1 Example

We were inspired by the tracing infrastructure used in the Google Chrome browser, and used some of their tooling in our solution. Chrome is a significantly more complicated project than Viper, and is heavily optimized for performance. It contains two tracing systems: one for developers of web pages and one for the developers of Chrome itself. The one which is mainly intended for Chrome developers [15] captures and displays the time spans of many kinds of events in the various processes and threads which Chrome is composed of. The trace viewer UI which is part of that system is available as a separate open-source project [16].

We used the Chrome trace viewer UI to show timing information from Viper. The screenshot in figure 7.1 shows a trace from Carbon in the Chrome trace viewer.

The top section (“Process 1”) is Viper Server. The grey or white blocks in the background indicate separate threads, with their thread IDs (here 40 for the main thread of Viper Server) shown on the left. Each colored bar shows the time span of an individual event. The colored bars within a thread are automatically stacked so that each bar has a time span which contains all bars below. The coloring of bars is chosen by the trace viewer and has no significance for Viper traces.

The bottom section (“Process 301”) contains events from Boogie. For Boogie, we display the timespan in which each method or function was verified. This
7. DETAILED TIMING

Figure 7.1: Example trace from Carbon in the Chrome trace viewer

allows us to see how long each individual method takes to verify, as well as how Boogie chooses to parallelize verification.

7.2 Implementation

To view a trace from Viper in the Chrome trace viewer, we had to generate a trace file in the appropriate format. The subset of the Chrome trace format which we use requires us to create a JSON file containing a list of objects. Each object marks either the beginning or the end of a time span. Each object must contain a display name, process ID, thread ID and timestamp. Timestamps can be in any timebase (for example time since start of Viper), but they must be consistent across threads and processes.

We modified Viper Server to send these JSON traces together with the verification results that it normally sends. Since they types for these results are defined in Silver, we had to make modifications to Silver and Carbon. We added a helper class which stores timing events to Silver. An instance of this helper class is created before starting verification, and is passed down through all parts of Viper where tracing is necessary, including to Carbon. Each part of Viper can easily add its own tracing events to this helper object, without knowledge of the Chrome tracing format. This helper class handles serialization of the traces, which is done after trace is complete, so that the impact of the tracing system on performance is minimal.
The trace in the above example contains events from Viper Server, Silver and Carbon in the Viper Server process, but also events from Boogie in the other process. Within Viper we can use our helper class to save timing events, and have a shared timebase from the high resolution timing facilities in Scala. To also integrate timing events from Boogie, we enabled the tracing feature built into Boogie, which prints certain events with timestamps as part of the Boogie output. We parse this Boogie output in Carbon and add the individual trace prints to the tracing helper class.

Since Boogie is written in C# and runs in Mono on non-Windows systems, it has different timing facilities than Viper which uses Scala in a JVM. Also, the built-in timing prints in Boogie format time as seconds since Boogie startup. We wanted to avoid changing the Boogie timestamp. We convert from the Boogie timebase to the Viper timebase by assuming that the last timing print from Boogie occurs exactly when Scala stops waiting for the Boogie process. For this purpose, we added an extra timing print to Boogie immediately before it exits. Our assumption synchronizes timebases when stopping Boogie, instead of when starting it, since starting Boogie can take a significant amount of time due to Mono startup. Although our assumption is not completely correct, this only affects the alignment of timespans from Boogie and from Viper in the trace viewer. Specifically, we show the Mono startup time as slightly longer than it actually is, and the Mono shutdown time as instant.
8.1 Tested Viper programs

The Viper programs we used for most benchmarks came from the following sources:

- Silver test suite (Silver)
- Viper “examples” repository (VPE)
- Nagini (Python frontend) test suite (Nagini)
- Prusti (Rust frontend) test suite (only smallest examples, see below) (Prusti)
- Graph reachability examples (Graph)
- Own examples (Own)

We use the short names in the last parentheses to refer to these sources in tables.

All benchmarks used all of these examples, unless stated otherwise. Viper programs from each source generally had quite different characteristics. For example, the Silver test suite uses many special Viper features, but mostly consists of very short running examples with many methods.

Most examples generated by Prusti [3] (the Rust frontend for Viper) are unusably slow to verify with Carbon. This was a previously-known issue. Because of this, we only included a few very small Prusti examples in the test suite. We investigate the performance of Prusti examples further in chapter 5.

8.2 Methodology

We ran all benchmarks presented in this report with the following setup: We used a dedicated workstation machine with an Intel Core i7-4770 CPU (3.40
8. Benchmarking

GHz base frequency) and 16 GiB of RAM. We performed a clean installation of Ubuntu Server 18.04.2 LTS (Linux kernel 4.15.0). The system had no GUI and had automatic updates disabled. We ran the Viper Runner 2 worker-agent and Docker on this machine (see section 6.4). We ran the version of Viper under test as a Docker container with Viper Server inside. Chapter 6 contains more details to about how Viper Runner 2 runs benchmarks.

The Viper Docker image was built by first freshly cloning the repositories of Carbon, Silicon, Silver, Viper Server and Boogie at fixed Mercurial revisions or Git commits. We then performed all necessary build steps in Docker using Ubuntu 19.04, Mono 5 and OpenJDK 11. We used Ubuntu 19.04 since earlier versions only ship Mono 4, which is not new enough to compile the latest versions of Boogie. The Z3 SMT solver was also built from source from a fixed commit.

We ran all benchmarks using viper-runner-2. Details of how it runs examples are in chapter 6. Each example was run 7 times consecutively. We discarded the fastest and slowest times for each example, as suggested in an internal Viper wiki page [17]. All results in this document are based on the 5 remaining times.

We exclude some Viper files from our results, either due to timeouts or missing timings. Excluded files are mentioned separately before other results. We enforced a timeout of 2000s per single run of each example. A Viper file for which timed out during any of the 5 runs (longest run of 7 is discarded, see above) is considered to have issues with timeouts. When Carbon raises a Java runtime exception, Viper Server will not report timing information. If a there is missing timing information for a file for any of the 5 runs, it is considered to have issues with missing timings.

8.3 Baseline

To evaluate our modifications to Viper and to understand our example Viper programs we used a baseline configuration of Viper.

8.3.1 Versions

The following Mercurial revisions or Git commits were used for each component in the baseline configuration:

- Carbon - 8747f34da108389f3c05df6dfb08d6f6d17be90314
- Silicon - daa10897c21db08e7dae0331dfce29c4a996c8f
- Silver - a4d98b94f58df4e80620d22059210942df61df29
- Viper Server - 2d3f6c173b3529c13bb69da78e7ab50c32d7e27e
8. Benchmarking

- Boogie - 32cb0b9398bd5fac91717ca820d56bd391322434
- Z3 - d6df51951f4cedc95f0d9d3b1297d04da465d8f2ca

The versions of Silicon and Silver which we used were the unmodified latest revisions on 11.03.2019.

The above version of Viper Server was based on the latest revision on 11.03.2019, but with an added command line option to change the bind interface for its HTTP server. This option was necessary for the deployment of Viper Runner 2 to work correctly (see section 6.4). This change should not affect performance or correctness at all.

The above version of Carbon was based on a revision from 10.03.2019, just before the “successor heap” feature was added. We chose this base revision since successor heaps are not supported (or are disabled via a command line argument) in the versions of Carbon which we modified for later benchmarks (section 4.4). We back-ported a bug fix (exhaling unfolding expressions) onto this base revision to arrive at the revision listed above. This bug fix was also present in our other modified versions of Carbon, and was necessary for them to work correctly.

The above Boogie commit was the latest commit as of running our benchmarks (committed on 05.04.2019). We used this commit for the baseline configuration, since we use modified versions of Boogie in other measurements to support collecting detailed timing information (see chapter 7). Those modified versions were based on the unmodified commit listed above.

The above Z3 commit is for release 4.8.4. This was the latest release with a version number (but not the latest commit) as of 11.03.2019 (released on 20.12.2018).

8.3.2 Arguments

We used the Carbon command line argument “–disableAllocEncoding”, since the “allocation encoding” feature affects performance and correctness, but is not supported in the per-field version of Carbon (see section 4.4).

We also used the “–disableCaching” argument, so the Viper Server caching mechanism was correctly disabled. For this reason the warning in section 8.4 does not apply to the baseline benchmarks.

8.3.3 Results

We evaluated the examples from the sources described in section 8.1 using the baseline Viper configuration. We tested 1528 Viper files in total. We excluded 75
files from further evaluation. The reasons why each individual file was blacklisted are listed in appendix A.

As previously described, we calculated the mean verification time for each Viper file over 5 runs. The next table contains summary statistics for this mean verification time in seconds for the 1453 Viper files with no issues.

<table>
<thead>
<tr>
<th>Source</th>
<th>Total</th>
<th>Slow</th>
<th>Mean</th>
<th>Min</th>
<th>P10</th>
<th>P50</th>
<th>P90</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>1453</td>
<td>51</td>
<td>3.573</td>
<td>0.002</td>
<td>0.014</td>
<td>1.005</td>
<td>7.682</td>
<td>199.125</td>
</tr>
<tr>
<td>Silver</td>
<td>1048</td>
<td>5</td>
<td>2.034</td>
<td>0.002</td>
<td>0.008</td>
<td>0.942</td>
<td>6.538</td>
<td>150.636</td>
</tr>
<tr>
<td>Nagini</td>
<td>260</td>
<td>28</td>
<td>7.876</td>
<td>0.167</td>
<td>6.638</td>
<td>7.341</td>
<td>10.006</td>
<td>25.954</td>
</tr>
<tr>
<td>Own</td>
<td>85</td>
<td>11</td>
<td>8.223</td>
<td>0.872</td>
<td>0.896</td>
<td>1.226</td>
<td>14.979</td>
<td>199.125</td>
</tr>
<tr>
<td>VPE</td>
<td>35</td>
<td>5</td>
<td>6.333</td>
<td>0.473</td>
<td>0.966</td>
<td>4.439</td>
<td>15.694</td>
<td>26.507</td>
</tr>
<tr>
<td>Graph</td>
<td>14</td>
<td>0</td>
<td>3.630</td>
<td>0.013</td>
<td>0.033</td>
<td>1.857</td>
<td>8.029</td>
<td>8.219</td>
</tr>
<tr>
<td>Prusti</td>
<td>11</td>
<td>2</td>
<td>3.688</td>
<td>1.002</td>
<td>1.007</td>
<td>1.240</td>
<td>13.923</td>
<td>15.251</td>
</tr>
</tbody>
</table>

The min and max columns show the minimum and maximum mean verification time. This means that individual verifications of some files may have taken longer than the time listed in the max column. The P10 column contains the 10th percentile of the mean verification times. The P50 and P90 columns contain the 50th and 90th percentile respectively.

To help understand the impact of our changes to Carbon on “slow” examples, we chose to define slow Viper files as ones which have a mean verification time of at least 10 seconds with a given configuration. As with all other results, the slow category does not include blacklisted files.

The Viper programs which we collected from various sources perform quite differently from each other. The table is split by source to illustrate this. While most examples we tested with come from the Silver test suite, most of the slow examples were generated from the Nagini test suite.

8.4 Important warning: Incorrect caching

Viper Server has a built in system for caching verification results for parts of Viper programs. Of course we planned to disable this cache for benchmarking. The latest versions of Viper Runner 2 do this by automatically adding the “–disableCaching” command line argument when creating a verification job, which correctly disables caching.

However, when testing alternate heap shapes and per-field heaps, we had accidentally not disabled this cache correctly. Once we discovered the issue, it was already too late to re-run the affected benchmarks. This means that all the results from the benchmark runs in section 8.5 are not comparable to the results from other sections. More importantly, all results within that section are possibly incorrect, including the relationships between verification times for
8. Benchmarking

different heap shapes. Benchmark runs in other sections are not affected by this issue.

We believe that our incorrect results likely still have a useful relationship to the results which we would have if we had disabled the cache correctly. We will now explain how we think our data was affected. Viper Server performs caching by taking the full input program and looking up all methods from this program in a cache. It then creates a “fake program”, by copying all non-method pieces of the input program, as well as all methods for which no up-to-date results were found in the cache. For any methods with up-to-date results in the cache, Viper Server directly outputs the cached errors and copies the method into the fake program, but removes the method body. After verifying the fake program, Viper Server updates its cache as necessary. The whole process described in this paragraph definitely occurred in our incorrectly configured benchmark runs.

However, we believe that no method lookups ever hit in the cache, meaning that the work performed by Carbon would be the same as if caching were correctly disabled. The reason we think that no cache lookups hit is due to the way that Viper Runner 2 works internally (see section 6.3). When it verifies a Viper file, the content of that file is first copied to a temporary file with a name containing a pseudo-random 32 bit integer. This temporary file is then passed to Viper Server as the input file. Since the cache inside Viper Server has the file name as part of the key, it is very likely that no cache hit will occur. It is important to note that even if the cache was never hit, the overhead of performing lookups and updates to the cache still occurred. This means that our results are definitely still not identical to those which we would have collected if the cache was directly disabled.

8.5 Alternate shape and per-field heaps

We tested the performance of Carbon with the modifications described in the chapters “Alternate shape for heaps and masks” (chapter 3) and “Per-field heaps and masks” (chapter 4). As described in section 8.4, the data in this section is possibly incorrect, since we failed to correctly disable caching in Viper Server.

8.5.1 Versions

We used the same setup as for the baseline tests, but with the following Mercurial revisions or Git commits:

- Carbon (alternate heap shapes) - f3e4ecf2a977
- Carbon (per-field heap) - 508b6539af3d
8. Benchmarking

- Silicon - 8d45012ea230
- Silver - 43b9bc10dd8a
- Viper Server - 6745b31cf938
- Boogie - a79deace0ba6cac28853a52ae0ef8e8937fe1115a54
- Z3 - d6df51951f4cd95f0d9f3b1297d04a465d8f2ca

In general, the above software versions are based on the ones in the baseline configuration. Except for the modifications which we tested (heap shapes), they also contain support for collecting detailed timing information (see chapter 7) and some fixes to allow all of the components to be compiled and used together. Below we explain in detail which versions of each component were used.

The above versions of Carbon are based on the one used in the baseline configuration. We implemented the changes described in the per-field heap and alternate heap shape chapters (chapters 4 and 3), and added detailed timing support (chapter 7) to arrive at the above revisions.

The above version of Silver is based on the version used in the baseline configuration. It also includes an extra feature (CSV reporter) to work with an up-to-date version of Prusti. This should not affect our measurements. Additionally, we added detailed timing support to Silver to arrive at the above revision.

The version of Viper Server which we used is based on the version used in the baseline configuration. It also includes later build-related fixes from the project’s authors, so that it would build correctly with the versions of the other Viper components which we used. Additionally, we added detailed timing support to arrive at the above revision.

The version of Silicon which we used was an unmodified version created by Silicon’s authors. It was chosen so that it would build correctly with the other Viper components. This is not the same version of Silicon as in the baseline configuration, but we generally did not perform any benchmarks with Silicon for this report.

The version of Boogie which we used is based on the one used in the baseline configuration. We added detailed timing support to arrive at the above commit.

8.5.2 Arguments

When testing alternate heap shapes, Carbon was run with “–disableAllocEncoding” and “–disableSuccessorHeaps”. Note that we did not pass “–disableSuccessorHeaps” to Carbon in our baseline configuration, since the successor heap feature was not implemented in that version of Carbon. We also passed the appropriate
heap shape (for example via “-heapShape=h[f][o]”, “-maskShape=h[f][o]” and “-predicateMaskShape=h[f][o]” for the “h[f][o]” heap shape). When testing per-field heaps, Carbon was run with no extra arguments, since allocation encoding and successor heaps are not supported in the per-field version of Carbon.

8.5.3 Results

We tested using the same Viper files as the baseline configuration. As previously mentioned, we blacklisted the Viper files which gave different output than the baseline for alternate heap shapes or per-field heaps (see appendix A). That means that our results are based on the timings of the 1453 examples which had no issues in the baseline benchmark and gave the same output for all tested configurations of Carbon. The overall results for these example were as follows:

<table>
<thead>
<tr>
<th>Source</th>
<th>Total</th>
<th>Slow</th>
<th>Mean</th>
<th>Min</th>
<th>P10</th>
<th>P50</th>
<th>P90</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>h[o][f]</td>
<td>1453</td>
<td>86</td>
<td>3.838</td>
<td>0.000</td>
<td>0.003</td>
<td>0.965</td>
<td>8.050</td>
<td>197.466</td>
</tr>
<tr>
<td>h[f][o]</td>
<td>1453</td>
<td>83</td>
<td>4.024</td>
<td>0.000</td>
<td>0.003</td>
<td>0.967</td>
<td>8.110</td>
<td>336.528</td>
</tr>
<tr>
<td>h[o][f]</td>
<td>1453</td>
<td>92</td>
<td>5.468</td>
<td>0.000</td>
<td>0.003</td>
<td>0.973</td>
<td>8.425</td>
<td>1156.891</td>
</tr>
<tr>
<td>h[f][o]</td>
<td>1453</td>
<td>77</td>
<td>3.521</td>
<td>0.000</td>
<td>0.003</td>
<td>0.971</td>
<td>8.029</td>
<td>109.426</td>
</tr>
<tr>
<td>h[f][o]</td>
<td>1453</td>
<td>111</td>
<td>3.630</td>
<td>0.000</td>
<td>0.003</td>
<td>0.977</td>
<td>8.942</td>
<td>84.419</td>
</tr>
</tbody>
</table>

By overall mean verification time the h[o][f] heap shape was the slowest configuration and h[f][o] was the fastest. The h[f][o] configuration had the second-fastest mean verification time, but interestingly also had more slow cases than with any other configuration. The maximum verification time varied significantly between configurations. In the rest of this section, we will attempt to illustrate how the different configurations compare with various groups of examples.

Splitting the above table by the source of our Viper files gives the following:
Examples from different sources seem to react quite differently to changes in heap shape.

The graph-reachability examples perform best with the $h[f, o]$ heap shape by every metric shown, with other configurations being slightly slower.

With Nagini examples, the $h[o, f]$, $h[f, o]$ and $h[f][o]$ heaps perform similarly to each other, while the $h[o][f]$ heap is very slightly slower. The per-field heap configuration is slow on around 40 more examples than the other configurations, which reflects strongly on the overall statistics for this configuration.

The metrics from the Prusti examples show very large differences between configurations, but this is likely due to the fact that all the examples in this category are generated from tiny Rust programs which are very similar to each
other. This makes the results from this category less useful in isolation. The $h[f,o]$ heap shape performs much slower than all other configurations in this category. Interestingly, it also performs significantly differently than $h[o,f]$.

In the category with our own examples the $h[o][f]$ heap shape has the slowest mean verification times. The per-field heap configuration performs much faster than all other configurations on our own examples. Like the Prusti examples, the examples in this category are likely not helpful for understanding “typical” Viper performance. They are mostly synthetic and many of the examples are variations of each other.

The mean verification time of Silver test cases with the $h[o][f]$ heap is significantly higher than other configurations. This may be caused by the slowest examples, since the 90th percentile verification times are similar for all configurations. The per-field heap configuration has faster mean verification times than all other configurations in this category, closely followed by the $h[f][o]$ heap shape.

The VPE examples (Viper project examples) show similar verification times for all configurations except for the $h[o][f]$ configuration, which is significantly slower.

In summary of the results from the above table: Splitting by the source of our Viper files suggests that the relationship between the performance of different configurations may significantly depend on the chosen Viper files.

From the above table it is not clear how individual files are affected by changing the heap shape. We now attempt to show this using the following strategy: For each Viper file, we try to determine which configuration is the fastest. We consider a configuration the fastest if the average verification time with that configuration is more than $n$ seconds faster than with any other configuration. By this definition, there is not always a fastest configuration. The following tables show the results for the values of $n$ 0, 0.1, 0.5, 1, 2, 10 and 50 (seconds):

<table>
<thead>
<tr>
<th>Fastest config</th>
<th>$n = 0$</th>
<th>$n = 0.1$</th>
<th>$n = 0.5$</th>
<th>$n = 1$</th>
<th>$n = 2$</th>
<th>$n = 10$</th>
<th>$n = 50$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclear</td>
<td>108</td>
<td>1194</td>
<td>1400</td>
<td>1426</td>
<td>1433</td>
<td>1447</td>
<td>1453</td>
</tr>
<tr>
<td>$h[o,f]$</td>
<td>635</td>
<td>82</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$h[f,o]$</td>
<td>280</td>
<td>36</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$h[o][f]$</td>
<td>132</td>
<td>40</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$h[f][o]$</td>
<td>182</td>
<td>50</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$h_f[o]$</td>
<td>116</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>14</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

For $n=0$, $h[o,f]$ was the fastest configuration for around 40% of Viper files. The “unclear” case is only possible for $n=0$ if the two fastest configurations have the same average verification time for a given Viper file. This occurs for small Viper files for which parsing fails (failed parsing takes a few milliseconds independent of configuration and is rounded to whole milliseconds by Viper Server).

Already for $n=0.1$ we see that for around 80% of Viper files there is no clear
8. Benchmarking

fastest heap shape. For the files which fall into the “unclear” category, this means that there is no single configuration that is 0.1 seconds faster than all others. In these cases, there could still be multiple configurations which are faster than the others, but not than each other. For Viper files where the heap shape makes no difference in verification time, random timing variation could still cause a configuration to falsely be counted as fastest, since n is quite small.

For 0.1 <= n <= 0.5, between 50% and 80% of the files where a configuration was categorized the fastest come from the Nagini test suite.

For n >= 1, the per-field heap configuration is clearly the fastest more often than any other configuration. There are also a few files for which the h[o][f] heap shape seems to have a significant advantage.

The next table is similar to the last, but instead shows for how many files a configuration was the slowest by a margin of n seconds:

<table>
<thead>
<tr>
<th>Slowest config</th>
<th>n = 0</th>
<th>n = 0.1</th>
<th>n = 0.5</th>
<th>n = 1</th>
<th>n = 2</th>
<th>n = 10</th>
<th>n = 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclear</td>
<td>47</td>
<td>1106</td>
<td>1210</td>
<td>1288</td>
<td>1384</td>
<td>1435</td>
<td>1447</td>
</tr>
<tr>
<td>h[o,f]</td>
<td>68</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h[f,o]</td>
<td>77</td>
<td>11</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>h[o][f]</td>
<td>273</td>
<td>66</td>
<td>40</td>
<td>35</td>
<td>31</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>h[f][o]</td>
<td>125</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h[o]</td>
<td>863</td>
<td>250</td>
<td>194</td>
<td>123</td>
<td>32</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Although the per-field heap configuration was the fastest more often than any other configuration for n = 1, it was also most often the slowest. However, almost all of the 123 affected files are Nagini test cases. This is consistent with the significantly higher mean verification time for Nagini test cases with per-field heaps.

For n >= 0.1, roughly half of the files where the h[o][f] configuration was the slowest come from the Silver test suite. This is also consistent with the mean verification times presented above.

As the two tables above have shown, there are some cases one configuration is significantly faster or slower than all of the others. More generally, there are sometimes significant differences between configurations. The next table gives concrete examples of Viper files with such differences. We chose the files to list as follows: For each Viper file, we took the mean verification times with each configuration. From these 4 times we calculated the standard deviation and divided it by the mean. This metric has no deep statistical meaning, but we found it to be effective in finding Viper files with large differences between configurations. The table contains the 10 Viper files for which this term was the largest, sorted from largest to smallest.
<table>
<thead>
<tr>
<th>Source</th>
<th>File</th>
<th>Config</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Own</td>
<td>functions-with-predicates/normal-100x.vpr</td>
<td>h[0, f]</td>
<td>4.324</td>
</tr>
<tr>
<td>Own</td>
<td>functions-with-predicates/normal-100x.vpr</td>
<td>h[f, o]</td>
<td>4.373</td>
</tr>
<tr>
<td>Own</td>
<td>functions-with-predicates/normal-100x.vpr</td>
<td>h[0][f]</td>
<td>334.129</td>
</tr>
<tr>
<td>Own</td>
<td>functions-with-predicates/normal-100x.vpr</td>
<td>h[f][o]</td>
<td>5.470</td>
</tr>
<tr>
<td>Own</td>
<td>functions-with-predicates/normal-100x.vpr</td>
<td>h[f][o]</td>
<td>5.843</td>
</tr>
<tr>
<td>Own</td>
<td>assignment-chains/ref-pre...s-with-unfolds-1x7.vpr</td>
<td>h[0, f]</td>
<td>4.379</td>
</tr>
<tr>
<td>Own</td>
<td>assignment-chains/ref-pre...s-with-unfolds-1x7.vpr</td>
<td>h[f, o]</td>
<td>4.426</td>
</tr>
<tr>
<td>Own</td>
<td>assignment-chains/ref-pre...s-with-unfolds-1x7.vpr</td>
<td>h[0][f]</td>
<td>55.145</td>
</tr>
<tr>
<td>Own</td>
<td>assignment-chains/ref-pre...s-with-unfolds-1x7.vpr</td>
<td>h[f][o]</td>
<td>4.356</td>
</tr>
<tr>
<td>Own</td>
<td>assignment-chains/ref-pre...s-with-unfolds-1x7.vpr</td>
<td>h[f][o]</td>
<td>2.077</td>
</tr>
<tr>
<td>Silver</td>
<td>all/issues/carbon/0056.sil</td>
<td>h[0, f]</td>
<td>2.800</td>
</tr>
<tr>
<td>Silver</td>
<td>all/issues/carbon/0056.sil</td>
<td>h[f, o]</td>
<td>2.848</td>
</tr>
<tr>
<td>Silver</td>
<td>all/issues/carbon/0056.sil</td>
<td>h[0][f]</td>
<td>33.519</td>
</tr>
<tr>
<td>Silver</td>
<td>all/issues/carbon/0056.sil</td>
<td>h[f][o]</td>
<td>2.854</td>
</tr>
<tr>
<td>Silver</td>
<td>all/issues/carbon/0056.sil</td>
<td>h[f][o]</td>
<td>2.468</td>
</tr>
<tr>
<td>Own</td>
<td>many-folds/field-100x.vpr</td>
<td>h[0, f]</td>
<td>1.740</td>
</tr>
<tr>
<td>Own</td>
<td>many-folds/field-100x.vpr</td>
<td>h[f, o]</td>
<td>1.767</td>
</tr>
<tr>
<td>Own</td>
<td>many-folds/field-100x.vpr</td>
<td>h[0][f]</td>
<td>17.899</td>
</tr>
<tr>
<td>Own</td>
<td>many-folds/field-100x.vpr</td>
<td>h[f][o]</td>
<td>2.023</td>
</tr>
<tr>
<td>Own</td>
<td>many-folds/field-100x.vpr</td>
<td>h[f][o]</td>
<td>1.687</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWandE2.sil</td>
<td>h[0, f]</td>
<td>150.469</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWandE2.sil</td>
<td>h[f, o]</td>
<td>336.528</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWandE2.sil</td>
<td>h[0][f]</td>
<td>1156.891</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWandE2.sil</td>
<td>h[f][o]</td>
<td>26.121</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWandE2.sil</td>
<td>h[f][o]</td>
<td>9.608</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.iterative.sil</td>
<td>h[0, f]</td>
<td>8.333</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.iterative.sil</td>
<td>h[0, f]</td>
<td>8.829</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.iterative.sil</td>
<td>h[0][f]</td>
<td>66.703</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.iterative.sil</td>
<td>h[f][o]</td>
<td>12.047</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.iterative.sil</td>
<td>h[f][o]</td>
<td>3.493</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWand.sil</td>
<td>h[0, f]</td>
<td>63.839</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWand.sil</td>
<td>h[0, f]</td>
<td>49.224</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWand.sil</td>
<td>h[0][f]</td>
<td>269.855</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWand.sil</td>
<td>h[f][o]</td>
<td>21.964</td>
</tr>
<tr>
<td>Silver</td>
<td>all/third_party/stefan_recent/testTreeWand.sil</td>
<td>h[f][o]</td>
<td>9.384</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.sil</td>
<td>h[0, f]</td>
<td>74.102</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.sil</td>
<td>h[f, o]</td>
<td>86.987</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.sil</td>
<td>h[f][o]</td>
<td>677.625</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.sil</td>
<td>h[f][o]</td>
<td>109.426</td>
</tr>
<tr>
<td>Silver</td>
<td>all/chalice/AVLTree.sil</td>
<td>h[f][o]</td>
<td>84.419</td>
</tr>
<tr>
<td>Prusti</td>
<td>prusti/assert-after-set-3x.rs.vpr</td>
<td>h[0, f]</td>
<td>13.833</td>
</tr>
<tr>
<td>Prusti</td>
<td>prusti/assert-after-set-3x.rs.vpr</td>
<td>h[f, o]</td>
<td>57.946</td>
</tr>
<tr>
<td>Prusti</td>
<td>prusti/assert-after-set-3x.rs.vpr</td>
<td>h[f][o]</td>
<td>7.008</td>
</tr>
<tr>
<td>Prusti</td>
<td>prusti/assert-after-set-3x.rs.vpr</td>
<td>h[f][o]</td>
<td>9.197</td>
</tr>
<tr>
<td>Prusti</td>
<td>prusti/assert-after-set-3x.rs.vpr</td>
<td>h[f][o]</td>
<td>8.735</td>
</tr>
<tr>
<td>VPE</td>
<td>vmcai2016/linked-list-predicates.sil</td>
<td>h[0, f]</td>
<td>26.064</td>
</tr>
<tr>
<td>VPE</td>
<td>vmcai2016/linked-list-predicates.sil</td>
<td>h[f, o]</td>
<td>26.206</td>
</tr>
<tr>
<td>VPE</td>
<td>vmcai2016/linked-list-predicates.sil</td>
<td>h[0][f]</td>
<td>73.275</td>
</tr>
<tr>
<td>VPE</td>
<td>vmcai2016/linked-list-predicates.sil</td>
<td>h[f][o]</td>
<td>13.127</td>
</tr>
<tr>
<td>VPE</td>
<td>vmcai2016/linked-list-predicates.sil</td>
<td>h[f][o]</td>
<td>7.936</td>
</tr>
</tbody>
</table>
8. Benchmarking

All of the Viper files in the table above show large differences between configurations. The slowest verification times for each file are roughly 10 to 100 times as long as the fastest. In many examples $h[o][f]$ is the slowest configuration, as suggested by the results in the previous tables. This list of examples shows many different kinds of relationships between verification times of different configurations.

One motivation for comparing the performance of different heap shapes is to see if they are a better default choice than the current configuration used by Carbon. The following table shows for how many files a given configuration was slower or faster than the current Carbon configuration. For example, if a specific file had a 13s lower mean verification time with the $h[o][f]$ heap shape than with the current Carbon configuration ($h[o, f]$), this will be counted once in the "1s faster" column and once in the "10s faster" column in the $h[o][f]$ row. This table does not show whether a given configuration was faster than all others.

<table>
<thead>
<tr>
<th>Config</th>
<th>10s slower</th>
<th>1s slower</th>
<th>unclear</th>
<th>1s faster</th>
<th>10s faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h[f, o]$</td>
<td>4</td>
<td>15</td>
<td>1436</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$h[o][f]$</td>
<td>13</td>
<td>45</td>
<td>1396</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>$h[f][o]$</td>
<td>1</td>
<td>12</td>
<td>1409</td>
<td>32</td>
<td>11</td>
</tr>
<tr>
<td>$h_f[o]$</td>
<td>3</td>
<td>174</td>
<td>1243</td>
<td>36</td>
<td>13</td>
</tr>
</tbody>
</table>

The per-field heap configuration was faster than the baseline more often than any other configuration. The $h[f][o]$ shape was faster than the baseline nearly as often as the per-field configuration. However the $h[f][o]$ configuration was also slower than the baseline for significantly fewer files than the per-field configuration.

8.5.4 Conclusion

Our results seem to show a significant difference in verification time between configurations in some cases. However, in most cases there would be no significant change in verification time by changing heap shape. Additionally, there is no single best configuration. The best performing configuration seems to depend on the specific verification task.

We would recommend adopting the implementation for alternate heap shapes which we developed in this thesis. This would allow switching heap shapes to suit the verification task. It also requires relatively few code changes to Carbon (around 100-200 lines). We would not recommend using our per-field heap implementation, since it does not seem to perform significantly better than other heap shapes. Our per-field heap implementation is also much more complex (around 1000-1500 lines changed in Carbon), and does not allow switching to other heap shapes. We also recommend to consider setting the default heap shape to $h[f][o]$ instead of $h[o, f]$, since this will improve mean verification times, while not often
8. Benchmarking

being significantly slower than the current heap shape.

8.6 Prusti issue

In this section, we present benchmark results for the slow Prusti-inspired synthetic example presented in the “Prusti issues” chapter (chapter 5).

8.6.1 Methodology

As mentioned in chapter 5, we tested possible solutions for the issue which we identified by hand-modifying the Boogie program which was generated by Carbon for our synthetic example. Since these modified versions are Boogie programs and not Viper programs, we tested them by running Boogie directly (without running Viper at all). This makes it impossible to compare the measurements from this section with those from other sections. The remaining test methodology also differs from that of other sections.

We used a different machine than for the other benchmarks. Specifically, we used a laptop with an Intel Core i7-6600U CPU (2.60 GHz base frequency) and 16 GiB of RAM. The machine was running Arch Linux with Linux kernel version 5.0.7. The machine was not running any other software at the time of the benchmark, and had automatic updates disabled.

We ran Boogie directly (not in Docker) using a helper script. We made sure to terminate Boogie and Z3 correctly in case of timeouts.

We ran each file 4 times consecutively and did not discard any (slowest or fastest) results. We enforced a timeout of 600s per run of each example. A Boogie program which timed out in any of the 4 runs is marked in the results.

8.6.2 Files

Our Boogie program was based on the synthetic example Viper program from the “Prusti issues” chapter (chapter 5). We denote the number of local variables which that program was created with as \( n \), as we did when we introduced that example. We tested with each of the following values for \( n \): 3, 6, 7, 8, 9, 10, 11, 15, 20, 25, 30, 35, 40. We chose the values 3 and 6 to 11 during initial experimentation, since values of \( n \) smaller than 6 were quite fast, and values larger than 11 are quite slow with an unmodified Boogie program. We chose the evenly spaced values of \( n \) from 15 to 40, since these adequately capture the slowdown which occurs with larger \( n \) for the more efficient modified programs.

For each \( n \), we tested the following versions of the Boogie program:
8. Benchmarking

- Unmodified (original)
- Modified to add total mask, including “unfolding hack” (total-mask)
- Modified to add predicate limit (predicate-limit)
- Modified to assign fresh values to variables (new)

The short names in brackets are used in the rest of this section.

We generated a Boogie program by using Carbon with the arguments “–disableAllocEncoding” and “–disableSuccessorHeaps” with the input file for some fixed $n$. We then used a script to generate the equivalent Boogie program for all required $n$. The output files of this script are considered the unmodified versions of the Boogie program. To generate the other versions, we adjusted our script to generate Boogie programs with the desired changes applied.

An exception to the above process was the program which was modified to assign fresh values to variables (new). The was created in a similar way, but only passing “–disableSuccessorHeaps” and not “–disableAllocEncoding” to Carbon, since this modification requires allocation encoding.

The full unmodified and modified Boogie programs for $n = 3$ can be found in appendix C.

8.6.3 Results

We ran benchmarks as described above and collected the results in the following table:

<table>
<thead>
<tr>
<th>n</th>
<th>original</th>
<th>total-mask</th>
<th>predicate-limit</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.066</td>
<td>1.076</td>
<td>1.065</td>
<td>1.062</td>
</tr>
<tr>
<td>6</td>
<td>1.482</td>
<td>1.251</td>
<td>1.204</td>
<td>1.182</td>
</tr>
<tr>
<td>7</td>
<td>2.602</td>
<td>1.422</td>
<td>1.303</td>
<td>1.331</td>
</tr>
<tr>
<td>8</td>
<td>6.744</td>
<td>1.490</td>
<td>1.404</td>
<td>1.391</td>
</tr>
<tr>
<td>9</td>
<td>27.419</td>
<td>1.619</td>
<td>1.769</td>
<td>1.612</td>
</tr>
<tr>
<td>10</td>
<td>232.099</td>
<td>2.118</td>
<td>1.859</td>
<td>2.042</td>
</tr>
<tr>
<td>11</td>
<td>timeout</td>
<td>2.787</td>
<td>2.160</td>
<td>2.979</td>
</tr>
<tr>
<td>15</td>
<td>timeout</td>
<td>4.765</td>
<td>3.873</td>
<td>7.512</td>
</tr>
<tr>
<td>20</td>
<td>timeout</td>
<td>11.410</td>
<td>9.070</td>
<td>26.863</td>
</tr>
<tr>
<td>25</td>
<td>timeout</td>
<td>30.320</td>
<td>25.304</td>
<td>121.670</td>
</tr>
<tr>
<td>30</td>
<td>timeout</td>
<td>43.310</td>
<td>41.583</td>
<td>287.814</td>
</tr>
<tr>
<td>35</td>
<td>timeout</td>
<td>71.165</td>
<td>99.517</td>
<td>timeout</td>
</tr>
<tr>
<td>40</td>
<td>timeout</td>
<td>111.204</td>
<td>126.721</td>
<td>timeout</td>
</tr>
</tbody>
</table>
8. Benchmarking

Each cell contains the mean verification time in seconds for a given Boogie program. The standard deviation for the verification time of each file was always below 3% of the mean verification time for that file.

Clearly every modified Boogie program performs better than the corresponding unmodified program. Interestingly, all of the modified variants still start becoming slow relatively quickly as $n$ grows.

The total-mask and predicate-limit variants perform roughly similarly to each other, and significantly better than the new or original variants. The differences in verification time between total-mask and predicate-limit variants in larger examples could be caused by randomness in Z3.

Explicitly establishing non-aliasing of variables using new performs significantly worse than the other two modified variants.

8.6.4 Conclusion

The evaluation results of our prototypes of various improvements for our Prusti-inspired synthetic example seem promising. It would be interesting to implement these changes in Viper and Prusti to see how they affect real examples. The new and predicate-limit variants are especially promising, since they should be relatively simple to implement.


Appendix A

Blacklisted files

The following 49 files were blacklisted since they crash with the baseline configuration of Carbon. We consider Java exceptions and Boogie type errors crashes. Normal verification errors are not considered crashes.

- `graph-reachability:trclo_ring-insert.vpr`
- `silver:all/assume/assume10QPpred.sil`
- `silver:all/assume/assume10QPwand.sil`
- `silver:all/heap-dependent_triggers/triggerFoldPackage.sil`
- `silver:all/heap-dependent_triggers/triggerWand.sil`
- `silver:all/invariants/loops1.sil`
- `silver:all/issues/carbon/0061.sil`
- `silver:all/issues/carbon/0062.sil`
- `silver:all/issues/carbon/0071.sil`
- `silver:all/issues/carbon/0092.sil`
- `silver:all/issues/carbon/0203.sil`
- `silver:all/issues/carbon/0219.sil`
- `silver:all/issues/carbon/0223.sil`
- `silver:all/issues/silicon/0292.sil`
- `silver:all/issues/silicon/0310a.sil`
- `silver:all/issues/silicon/0310b.sil`
- `silver:all/issues/silicon/unofficial002.sil`
Blacklisted files

- silver:all/issues/silver/0088.sil
- silver:all/issues/silver/0175.sil
- silver:all/issues/silver/0230.sil
- silver:all/permission_introspection/forpermPredicatesAdvanced.sil
- silver:all/permission_introspection/forpermQP.sil
- silver:all/permission_introspection/forpermWands.sil
- silver:all/permission_introspection/permWandAlias.sil
- silver:all/permission_introspection/permWandApply.sil
- silver:all/permission_introspection/permWandInhale.sil
- silver:all/permission_introspection/permWandQP.sil
- silver:all/permissions/epsilons.sil
- silver:examples/tree-delete-min/tree_delete_min.sil
- silver:graphs/static/examples/ring-insert.vpr
- silver:quantifiedcombinations/multiple_quantifiers.sil
- silver:quantifiedpermissions/issues/issue_0078.sil
- silver:wands/examples/list_insert_tmp.sil
- silver:wands/examples_paper/list_insert_heuristics.sil
- silver:wands/examples_paper/list_insert.sil
- silver:wands/examples_paper/tree_delete_min_heuristics.sil
- silver:wands/examples_paper/tree_delete_min.sil
- silver:wands/examples/tree_delete_min_no_assert.sil
- silver:wands/new_syntax/ApplyingBranching.sil
- silver:wands/new_syntax/QPFields.sil
- silver:wands/new_syntax/QPPredicates.sil
- silver:wands/new_syntax/QPWands.sil
- silver:wands/new_syntax/SnapshotsWithPredicates.sil
- silver:wands/regression/conditionals2.sil
Blacklisted files

- silver:wands/regression/eval_states.sil
- silver:wands/regression/snapshots.sil
- silver:wands/regression/wand_shapes_simple_exhale.sil
- viperproject/examples:tree-delete-min/tree_delete_min.sil
- viperproject/examples:vmcai2016/linked-list-predicates-with-wands.sil

The following 15 files were blacklisted since they give different verification errors with the per-field heap configuration than with the baseline configuration.

- examples:nagini/obligations/verification/chalice2silver/christian/lt_loops.vpr
- examples:nagini/obligations/verification/chalice2silver/loopsAndRelease.vpr
- silver:wands/examples_new_syntax/ListIterator.sil
- silver:wands/new_syntax/ApplyingExpression.sil
- silver:wands/new_syntax/FunctionCall.sil
- silver:wands/regression/conditionals1.sil
- silver:wands/regression/folding_2.sil
- silver:wands/regression/folding_fun_frame_2.sil
- silver:wands/regression/folding_fun_frame.sil
- silver:wands/regression/folding_unfolding_combo.sil
- silver:wands/regression/issue029.sil
- silver:wands/regression/known_folded_1.sil
- silver:wands/regression/let_wands.sil
- silver:wands/regression/lhs.sil
- silver:wands/regression/unfolding.sil

The following 9 files were blacklisted since they timed out after 100s in some configurations, but were not retested with a 2000s timeout. These files could be removed from the blacklist. We did not remove them since we did not have enough time to re-test these files. All non-blacklisted files were run with a 2000s timeout, which they did not hit.

- examples:assignment-chains/ref-pred-int-1x10.vpr
Blacklisted files

- examples:assignment-chains/ref-pred-int-1x11.vpr
- examples:assignment-chains/ref-pred-int-preemptive-unfold-1x11.vpr
- examples:prusti/int-eq.rs.vpr
- graph-reachability:trclo_list_reverse.vpr
- silver:all/third_party/stefan_recent/testTreeWandE1.sil
- silver:graphs/static/examples/list_reverse.vpr
- silver:quantifiedpermissions/sets/unionfind.sil
- silver:transformations/Performance/BinomialHeap.sil

The following 3 files were blacklisted since they give extra verification errors when run with the h|o|f mask shape, compared to running with the baseline configuration. We are unsure why this is the case.

- examples:nagini/functional/verification/test_iterator_list.vpr
- examples:nagini/functional/verification/test_iterator_set.vpr
- examples:nagini/obligations/verification/test_waitlevels.vpr

The following file was blacklisted since we could not correctly load timings for it for one configuration from our SQL database. The reason for this is unclear.

- silver:all/issues/silver/0072.sil

The following file was blacklisted since we are missing the verification results for this file for one configuration. This is likely because it was in a chunk which failed to run 3 times.

- examples:nagini/functional/verification/test_operators.vpr

The following file was blacklisted since this file times out with a 2000s timeout in some configurations.

- examples:prusti/assert-after-set-4x.rs.vpr
assert-after-set-same-3x.rs.vpr:

```rust

field val_bool: Bool
field val_int: Int
field tuple_1: Ref
field val_ref: Ref
field tuple_2: Ref
field tuple_0: Ref

predicate ref$tuple3$ref$str$u32$u32(self: Ref) {
    acc(self.val_ref, 1 / 1) &&
    acc(tuple3$ref$str$u32$u32(self.val_ref), 1 / 1)
}

predicate u32(self: Ref) {
    acc(self.val_int, 1 / 1)
}

predicate str(self: Ref) {
    true
}

predicate tuple0$(self: Ref) {
    true
}
```

B-1
```plaintext
predicate i32 (self: Ref) {
    acc(self.val_int, 1 / 1)
}

predicate bool (self: Ref) {
    acc(self.val_bool, 1 / 1)
}

predicate never (self: Ref) {
    true
}

method m_assert_after_set_same_3x$$main$opensqu$0$closesqu$ ()
    returns (_0: Ref)
    {
        var _0: Bool
        var __t0: Bool
        var __t1: Bool
        var __t2: Bool
        var __t3: Bool
        var __t4: Bool
        var __t5: Bool
        var __t6: Bool
        var __t7: Bool
        var __t8: Bool
        var __t9: Bool
        var _1: Ref
        var _2: Ref
        var _3: Ref
        var _4: Ref
        var _5: Ref
        var _6: Ref
        var _7: Ref
        var _8: Ref
        var _9: Ref
        var _10: Ref
        var _11: Ref
        var _12: Ref
        var _13: Ref
        var _14: Ref
        var _15: Ref
        var _16: Ref
        var _17: Ref
        var _18: Ref
        var _19: Ref
        var _20: Ref
        var _21: Ref
        var _22: Ref
        var _23: Ref
        var _24: Ref
        var _25: Ref
        var _26: Ref
        var _27: Ref
        var _28: Ref
    }
```
```plaintext
label start
// Name: "assert_after_set_same_3x::main"
// Def path: "assert_after_set_same_3x::main[0]"
// Span: 
  ../viper-playground/examples/prusti/assert-after-set-same-3x.rs:3:1:
  8:2
__t0 := false
__t1 := false
__t2 := false
__t3 := false
__t4 := false
__t5 := false
__t6 := false
// Preconditions:
inhale true
inhale true
label pre
// Allocate formal return and local variables
inhale acc(tuple0$(_0), 1 / 1)
inhale acc(i32(_1), 1 / 1)
inhale acc(tuple0$(_2), 1 / 1)
inhale acc(bool(_3), 1 / 1)
inhale acc(i32(_4), 1 / 1)
inhale acc(i32(_5), 1 / 1)
inhale acc(never(_6), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_7), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_8), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(_9), 1 / 1)
inhale acc(tuple0$(_10), 1 / 1)
inhale acc(bool(_11), 1 / 1)
inhale acc(bool(_12), 1 / 1)
inhale acc(i32(_13), 1 / 1)
inhale acc(never(_14), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_15), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_16), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(_17), 1 / 1)
inhale acc(tuple0$(_18), 1 / 1)
inhale acc(bool(_19), 1 / 1)
inhale acc(bool(_20), 1 / 1)
inhale acc(i32(_21), 1 / 1)
inhale acc(never(_22), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_23), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_24), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(_25), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_26), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_27), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_28), 1 / 1)
goto bb0
label bb0
// ----- bb0
_t0 := true
// [mir] StorageLive(_1)
// [mir] _1 = const 7i32
```
unfold acc(i32(_1), 1 / 1)
_1.val_int := 7
// [mir] StorageLive(_3)
// [mir] StorageLive(_4)
// [mir] StorageLive(_5)
// [mir] _5 = _1
unfold acc(i32(_5), 1 / 1)
_5.val_int := _1.val_int
// [mir] _4 = Eq(move _5, const 7i32)
unfold acc(bool(_4), 1 / 1)
_4.val_bool := _5.val_int == 7
// [mir] StorageDead(_5)
// [mir] switchInt(move _3) -> [false: bb3, otherwise: bb2]
if (!_t7) {
  goto bb0_bb3
}
goto bb0_bb2
label bb2
// ================= bb2 =================
_t1 := true
// [mir] StorageLive(_7)
// [mir] StorageLive(_8)
// [mir] _8 = &'12s(*_28)
// [mir] _7 = &'12s(*_8)
// [mir] const std::rt::begin_panic(const "assertion failed: x == 7", move _7) -> bb1
// Rust panic - const "assertion failed: x == 7"
assert false
goto end_of_method
label bb3
// ================= bb3 =================
_t2 := true
// [mir] _2 = ()
// [mir] StorageDead(_3)
// [mir] StorageLive(_11)
// [mir] StorageLive(_12)
// [mir] StorageLive(_13)
// [mir] _13 = 1
unfold acc(i32(_13), 1 / 1)
_13.val_int := _1.val_int
// [mir] _12 = Eq(move _13, const 7i32)
unfold acc(bool(_12), 1 / 1)
_12.val_bool := _13.val_int == 7
// [mir] StorageDead(_13)
// [mir] _11 = Not(move _12)
unfold acc(bool(_11), 1 / 1)
_11.val_bool := !_12.val_bool
// [mir] StorageDead(_12)
```c
// [mir] switchInt(move _11) -> [false: bb5, otherwise: bb4]
__t8 := _11.val_bool
if (!__t8) {
    goto bb3_bb5
}
goto bb3_bb4
label bb4
// =========== bb4 ===========
__t3 := true
// [mir] StorageLive(_15)
// [mir] StorageLive(_16)
// [mir] _27 = promoted[1]
// [mir] _16 = &'30s(*_27)
// [mir] _15 = &'30s(*_16)
// [mir] const std::rt::begin_panic(const "assertion failed: x == 7", move _15) -> bb1
// Rust panic - const "assertion failed: x == 7"
assert false
goto end_of_method
label bb5
// =========== bb5 ===========
__t4 := true
// [mir] _10 = ()
// [mir] StorageDead(_11)
// [mir] StorageLive(_19)
// [mir] StorageLive(_20)
// [mir] StorageLive(_21)
// [mir] _21 = 1
unfold acc(i32(_21), 1 / 1)
_21.val_int := 1.val_int
// [mir] _20 = Eq(move _21, const 7132)
unfold acc(bool(_20), 1 / 1)
_20.val_bool := _21.val_int == 7
// [mir] StorageDead(_21)
// [mir] _19 = Not(move _20)
unfold acc(bool(_19), 1 / 1)
_19.val_bool := !_20.val_bool
// [mir] StorageDead(_20)
// [mir] switchInt(move _19) -> [false: bb7, otherwise: bb6]
__t9 := _19.val_bool
if (!__t9) {
    goto bb5_bb7
}
goto bb5_bb6
label bb6
// =========== bb6 ===========
__t5 := true
// [mir] StorageLive(_23)
// [mir] StorageLive(_24)
// [mir] _26 = promoted[0]
// [mir] _24 = &'48s(*_26)
// [mir] _23 = &'48s(*_24)
// [mir] const std::rt::begin_panic(const "assertion failed: x == 7", move _23) -> bb1
```
// Rust panic - const "assertion failed: x == 7"

assert_false

goto end_of_method

label bb7

// __________ bb7 __________

__t6 := true

// [mir] _18 = ()
// [mir] StorageDead(_19)
// [mir] _0 = ()
// [mir] StorageDead(_1)
// [mir] return

goto return

label bb0_bb3

// __________ bb0 --> bb3 __________

goto bb3

label bb0_bb2

// __________ bb0 --> bb2 __________

goto bb2

label bb3_bb5

// __________ bb3 --> bb5 __________

goto bb5

label bb3_bb4

// __________ bb3 --> bb4 __________

goto bb4

label bb5_bb7

// __________ bb5 --> bb7 __________

goto bb7

label bb5_bb6

// __________ bb5 --> bb6 __________

goto bb6

label return

// __________ return __________

// Target of any 'return' statement.
// Exhale postcondition
// Fold predicates for &mut args and transfer borrow permissions to old

assert true

exhale acc(tuple0$(_0), 1 / 1)
goto end_of_method

label end_of_method

}
field tuple_0 : Ref

predicate ref$tuple3$ref$str$u32$u32(self : Ref) {
    acc(self.val_ref, 1 / 1) &&
    acc(tuple3$ref$str$u32$u32(self.val_ref), 1 / 1)
}

predicate u32(self : Ref) {
    acc(self.val_int, 1 / 1)
}

predicate str(self : Ref) {
    true
}

predicate tuple3$ref$str$u32$u32(self : Ref) {
    acc(self.tuple_0, 1 / 1) &&
    acc(ref$str(self.tuple_0), 1 / 1) &&
    acc(self.tuple_1, 1 / 1) &&
    acc(u32(self.tuple_1), 1 / 1) &&
    acc(self.tuple_2, 1 / 1) &&
    acc(u32(self.tuple_2), 1 / 1)
}

predicate ref$str(self : Ref) {
    acc(self.val_ref, 1 / 1) &&
    acc(str(self.val_ref), 1 / 1)
}

predicate tuple0$(self : Ref) {
    true
}

predicate i32(self : Ref) {
    acc(self.val_int, 1 / 1)
}

predicate bool(self : Ref) {
    acc(self.val_bool, 1 / 1)
}

predicate never(self : Ref) {
    true
}

method example() returns (_0 : Ref) {
    var __t0 : Bool
    var __t1 : Bool
    var __t2 : Bool
    var __t3 : Bool
    var __t4 : Bool
    var __t5 : Bool
    var __t6 : Bool
    var __t7 : Bool
    var __t8 : Bool
    var __t9 : Bool
    var _1 : Ref
    var _2 : Ref
var _3 : Ref
var _4 : Ref
var _5 : Ref
var _6 : Ref
var _7 : Ref
var _8 : Ref
var _9 : Ref
var _10 : Ref
var _11 : Ref
var _12 : Ref
var _13 : Ref
var _14 : Ref
var _15 : Ref
var _16 : Ref
var _17 : Ref
var _18 : Ref
var _19 : Ref
var _20 : Ref
var _21 : Ref
var _22 : Ref
var _23 : Ref
var _24 : Ref
var _25 : Ref
var _26 : Ref
var _27 : Ref
var _28 : Ref

label start
// ------------ start ------------
// Name: "assert_after_set_same_3x::main"
// Def path: "assert_after_set_same_3x::main[0]"
// Span:
// ../viper-playground/examples/prusti/assert-after-set-same-3x.rs:3:1:
// 8:2
__t0 := false
__t1 := false
__t2 := false
__t3 := false
__t4 := false
__t5 := false
__t6 := false
// Preconditions:
inhale true
inhale true
label pre
// Allocate formal return and local variables
inhale acc(tuple0$(_0), 1 / 1)
inhale acc(i32(_1), 1 / 1)
inhale acc(tuple0$(_2), 1 / 1)
inhale acc(bool(_3), 1 / 1)
inhale acc(bool(_4), 1 / 1)
inhale acc(i32(_5), 1 / 1)
inhale acc(never(_6), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(_7), 1 / 1)
```
inhale acc(ref$tuple3$ref$str$u32$u32(8), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(9), 1 / 1)
inhale acc(bool(11), 1 / 1)
inhale acc(bool(12), 1 / 1)
inhale acc(i32(13), 1 / 1)
inhale acc(never(14), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(15), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(16), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(17), 1 / 1)
inhale acc(tuple0$(18), 1 / 1)
inhale acc(bool(19), 1 / 1)
inhale acc(bool(20), 1 / 1)
inhale acc(i32(21), 1 / 1)
inhale acc(never(22), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(23), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(24), 1 / 1)
inhale acc(tuple3$ref$str$u32$u32(25), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(26), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(27), 1 / 1)
inhale acc(ref$tuple3$ref$str$u32$u32(28), 1 / 1)

label bb0
 unfold acc(i32(1), 1 / 1)
  _1.val_int := 7
  if (_1.val_int == 7) {
    goto bb3
  }
  assert false
label bb3
  if (_1.val_int == 7) {
    goto bb5
  }
  assert false
label bb5
  if (_1.val_int == 7) {
    goto return
  }
  assert false
label return
  goto method_end
label method_end
```
Appendix C

Prusti synthetic benchmark prototypes

Unmodified (original):

```plaintext
// Translation of Viper program.
// Based on output from carbon with --disableAllocEncoding and
// --disableSuccessorHeaps.
// Not modified, but converted to Boogie to test against the other
// modifications.

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

function IdenticalOnKnownLocations(Heap: HeapType, ExhaleHeap:
  HeapType, Mask: MaskType): bool;
function IsPredicateField<A, B>(f: (Field A B)): bool;
function IsWandField<A, B>(f: (Field A B)): bool;
function getPredicateId<A, B>(f: (Field A B)): int;

// Frame all locations with direct permissions
axiom (forall <A, B> Heap: HeapType, ExhaleHeap: HeapType, Mask:
  MaskType, o: Ref, f_1: (Field A B) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    ExhaleHeap[o, f_1] })
```
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) ==> HasDirectPerm(Mask, o, f_1) ==> Heap[o, f_1] == ExhaleHeap[o, f_1]

// Frame all predicate mask locations of predicates with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
MaskType, pm_f: (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
IsPredicateField(pm_f), ExhaleHeap[null, PredicateMaskField(pm_f)] }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) ==> HasDirectPerm(Mask, null, pm_f) && IsPredicateField(pm_f) ==> Heap[null, PredicateMaskField(pm_f)] == ExhaleHeap[null, PredicateMaskField(pm_f)]
);

// Frame all locations with known folded permissions
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
MaskType, pm_f: (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
ExhaleHeap[null, pm_f], IsPredicateField(pm_f) }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) ==> HasDirectPerm(Mask, null, pm_f) && IsPredicateField(pm_f) ==> (forall <A, B> o2: Ref, f_1: (Field A B) ::
{ ExhaleHeap[o2, f_1] } 
Heap[null, PredicateMaskField(pm_f)][o2, f_1] ==> Heap[o2, f_1] == ExhaleHeap[o2, f_1]
)
);

// Frame all wand mask locations of wands with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
MaskType, pm_f: (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
IsWandField(pm_f), ExhaleHeap[null, WandMaskField(pm_f)] }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) ==> HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) ==> Heap[null, WandMaskField(pm_f)] == ExhaleHeap[null, WandMaskField(pm_f)]
);

// Frame all locations in the footprint of magic wands
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
MaskType, pm_f: (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
IsWandField(pm_f) }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) ==> HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) ==> (forall <A, B> o2: Ref, f_1: (Field A B) ::
{ ExhaleHeap[o2, f_1] } 
Heap[null, WandMaskField(pm_f)][o2, f_1] ==> Heap[o2, f_1] == ExhaleHeap[o2, f_1]
)
);

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
type Perm = real;

// Preamble of Permission module.

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

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

var Mask : MaskType;

const ZeroMask : MaskType;

axiom (forall <A, B> o_1: Ref, f_2: (Field A B) ::
    { ZeroMask[o_1, f_2] } ZeroMask[o_1, f_2] == NoPerm);

type PMaskType = <A, B> [Ref, Field A B] bool;

const ZeroPMask : PMaskType;

axiom (forall <A, B> o_1: Ref, f_2: (Field A B) ::
    { ZeroPMask[o_1, f_2] } !ZeroPMask[o_1, f_2]);

function PredicateMaskField<A>(f_3: (Field A FrameType)): Field A PMaskType;

function WandMaskField<A>(f_3: (Field A FrameType)): Field A PMaskType;

const NoPerm : Perm;

axiom NoPerm == 0.000000000;

const FullPerm : Perm;

axiom FullPerm == 1.000000000;

function Perm(a: real, b: real): Perm;

function GoodMask(Mask: MaskType): bool;

axiom (forall Heap: HeapType, Mask: MaskType ::
    { state(Heap, Mask) } state(Heap, Mask) => GoodMask(Mask));

axiom (forall <A, B> Mask: MaskType, o_1: Ref, f_2: (Field A B) ::
    { GoodMask(Mask), Mask[o_1, f_2] } GoodMask(Mask) => Mask[o_1, f_2] >= NoPerm && ((GoodMask(Mask) && !IsPredicateField(f_2)) && !IsWandField(f_2) => Mask[o_1, f_2] <= FullPerm);

function HasDirectPerm<A, B>(Mask: MaskType, o_1: Ref, f_2: (Field A B)): bool;

axiom (forall <A, B> Mask: MaskType, o_1: Ref, f_2: (Field A B) ::
    { HasDirectPerm(Mask, o_1, f_2) } HasDirectPerm(Mask, o_1, f_2) <= Mask[o_1, f_2] > NoPerm);

function sumMask(ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType): bool;

axiom (forall <A, B> ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType, o_1: Ref, f_2: (Field A B) ::
    { sumMask(ResultMask, SummandMask1, SummandMask2, ResultMask[o_1, f_2]) } sumMask(ResultMask, SummandMask1, SummandMask2, SummandMask1[o_1, f_2]) => sumMask(ResultMask, SummandMask1, SummandMask2, SummandMask2[o_1, f_2])
    ResultMask[o_1, f_2] == SummandMask1[o_1, f_2] + SummandMask2[o_1, f_2]}
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Preamble of Function and predicate module.

// Declarations for function framing

type FrameType;
const EmptyFrame : FrameType;
function FrameFragment<T>(t : T) : FrameType;
function ConditionalFrame(p : Perm, f_4 : FrameType) : FrameType;
function dummyFunction<T>(t : T) : bool;
function CombineFrames(a_1 : FrameType, b_1 : FrameType) : FrameType;

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Definition of conditional frame fragments

axiom (forall p : Perm, f_4 : FrameType :
  { ConditionalFrame(p, f_4) } => ConditionalFrame(p, f_4) == (if p > 0.000000000 then f_4 else EmptyFrame)
);

// Function for recording enclosure of one predicate instance in another
function InsidePredicate<A, B>(p : (Field A FrameType), v_1 : FrameType, q : (Field B FrameType), w : FrameType) : bool;

// Transitivity of InsidePredicate
axiom (forall <A, B, C> p : (Field A FrameType), v_1 : FrameType, q : (Field B FrameType), w : FrameType, r : (Field C FrameType), u : FrameType :
  { InsidePredicate(p, v_1, q, w), InsidePredicate(q, w, r, u) } => InsidePredicate(p, v_1, q, w) && InsidePredicate(q, w, r, u) ==>
  InsidePredicate(p, v_1, r, u)
);

// Knowledge that two identical instances of the same predicate cannot be inside each other
axiom (forall <A> p : (Field A FrameType), v_1 : FrameType, w : FrameType :
  { InsidePredicate(p, v_1, p, w) } =>
  !InsidePredicate(p, v_1, p, w)
);

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Translation of domain Assume

// The type for domain Assume
type AssumeDomainType;

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Translation of all fields

const unique f_5 : Field NormalField int;
axiom !IsPredicateField(f_5);
axiom !IsWandField(f_5);

// Translation of predicate p

type PredicateType_p;
function p_1(x : Ref) : Field PredicateType_p FrameType;
function p#sm(x : Ref) : Field PredicateType_p PMaskType;
axiom (forall x : Ref ::
  { PredicateMaskField(p_1(x)) }
  PredicateMaskField(p_1(x)) == p#sm(x));
axiom (forall x : Ref ::
  { p_1(x) }
  IsPredicateField(p_1(x)));
axiom (forall x : Ref ::
  { p_1(x) }
  getPredicateId(p_1(x)) == 0);
function p#trigger<A>(Heap : HeapType, pred : (Field A FrameType)) : bool;
function p#everUsed<A>(pred : (Field A FrameType)) : bool;
axiom (forall x : Ref, x2 : Ref ::
  { p_1(x), p_1(x2) }
  p_1(x) == p_1(x2) => x == x2);
axiom (forall x : Ref, x2 : Ref ::
  { p#sm(x), p#sm(x2) }
  p#sm(x) == p#sm(x2) => x == x2);
axiom (forall Heap : HeapType, x : Ref ::
  { p#trigger(Heap, p_1(x)) }
  p#everUsed(p_1(x)));

// Translation of method m

procedure m() returns ()
  modifies Heap, Mask;
  {
    var perm : Perm;
    var _1 : Ref;
    var _2 : Ref;
    var _3 : Ref;
    var newVersion : FrameType;
    var freshObj : Ref;
    // — Initializing the state
Mask := ZeroMask;
assume state(Heap, Mask);

// --- Initializing of old state

// --- Initializing the old state
assume Heap == old(Heap);
assume Mask == old(Mask);

// --- Translating statement: inhale acc(p(_1), write)
perm := FullPerm;
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: inhale acc(p(_2), write)
perm := FullPerm;
Mask[null, p_1(_2)] := Mask[null, p_1(_2)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: inhale acc(p(_3), write)
perm := FullPerm;
Mask[null, p_1(_3)] := Mask[null, p_1(_3)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: unfold acc(p(_1), write)
assume p#trigger(Heap, p_1(_1));
assume Heap[null, p_1(_1)] == FrameFragment(Heap[_1, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert {msg " Unfolding p(_1) might fail. There might be insufficient permission to access p(_1)."}
    perm <= Mask[null, p_1(_1)];
}
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] - perm;

// --- Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_1))) {
    havoc newVersion;
    Heap[null, p_1(_1)] := newVersion;
}
perm := FullPerm;
assume _1 != null;
Mask[_1, f_5] := Mask[_1, f_5] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// — Translating statement: unfold acc(p(_2), write)
assume p#trigger(Heap, p_1(_2));
assume Heap[null, p_1(_2)] == FrameFragment(Heap[_2, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert {:msg "Unfolding p(_2) might fail. There might be insufficient permission to access p(_2)."}
    perm <= Mask[null, p_1(_2)];
}
Mask[null, p_1(_2)] := Mask[null, p_1(_2)] - perm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_2))) {
    havoc newVersion;
    Heap[null, p_1(_2)] := newVersion;
}
perm := FullPerm;
assume _2 != null;
Mask[_2, f_5] := Mask[_2, f_5] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// — Translating statement: unfold acc(p(_3), write)
assume p#trigger(Heap, p_1(_3));
assume Heap[null, p_1(_3)] == FrameFragment(Heap[_3, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert {:msg "Unfolding p(_3) might fail. There might be insufficient permission to access p(_3)."}
    perm <= Mask[null, p_1(_3)];
}
Mask[null, p_1(_3)] := Mask[null, p_1(_3)] - perm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_3))) {
    havoc newVersion;
    Heap[null, p_1(_3)] := newVersion;
}
perm := FullPerm;
assume _3 != null;
// --- Translating statement: _1.f := 7
assert {msg "Assignment might fail. There might be insufficient permission to access _1.val_int."}  

FullPerm == Mask[1, f_5];
Heap[1, f_5] := 7;
assume state(Heap, Mask);

// --- Translating statement: _2.f := _1.f
// --- Check definedness of _1.f
assert {msg "Assignment might fail. There might be insufficient permission to access _1.f."}  
   HasDirectPerm(Mask, _1, f_5);
assume state(Heap, Mask);
assert {msg "Assignment might fail. There might be insufficient permission to access _2.f."}  
   FullPerm == Mask[2, f_5];
Heap[2, f_5] := Heap[1, f_5];
assume state(Heap, Mask);

// --- Translating statement: _3.f := _2.f
// --- Check definedness of _2.f
assert {msg "Assignment might fail. There might be insufficient permission to access _2.f."}  
   HasDirectPerm(Mask, _2, f_5);
assume state(Heap, Mask);
assert {msg "Assignment might fail. There might be insufficient permission to access _3.f."}  
   FullPerm == Mask[3, f_5];
Heap[3, f_5] := Heap[2, f_5];
assume state(Heap, Mask);

// --- Translating statement: assert _3.f == 7
// --- Check definedness of _3.f == 7
assert {msg "Assert might fail. There might be insufficient permission to access _3.f."}  
   HasDirectPerm(Mask, _3, f_5);
assume state(Heap, Mask);
// Phase 1: pure assertions and fixed permissions
assert {msg "Assert might fail. Assertion _3.f == 7 might not hold."}  
   Heap[3, f_5] == 7;
assume state(Heap, Mask);
}

Modified to add total mask, including “unfolding hack” (total-mask):

```plaintext
// Translation of Viper program.

// Based on output from carbon with —disableAllocEncoding and —disableSuccessorHeaps.
// Hand-modified to prototype the total mask concept.

// Preamble of State module.

function state(Heap: HeapType, Mask: MaskType, TotalMask: MaskType): bool;

// Preamble of Alternative heap module.

type Ref;
var Heap: HeapType;
const null: Ref;
type Field A B;
type NormalField;
type HeapType = <A, B> [ Ref , Field A B ] B;

function IdenticalOnKnownLocations(Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, TotalMask: MaskType): boolean;
function IsPredicateField<A, B>(f: (Field A B)): boolean;
function IsWandField<A, B>(f: (Field A B)): boolean;
function getPredicateId<A, B>(f: (Field A B)): int;

// Frame all locations with direct or indirect permissions
axiom (forall <A, B> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, TotalMask: MaskType, o: Ref, f_1: (Field A B)) :
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask), ExhaleHeap[o, f_1] } IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask) => (Mask[o, f_1] > NoPerm || TotalMask[o, f_1] > NoPerm) => Heap[o, f_1] == ExhaleHeap[o, f_1];

// Frame all predicate mask locations of predicates with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, TotalMask: MaskType, pm_f: (Field C FrameType)) :
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask), IsPredicateField(pm_f, ExhaleHeap[null, UnitPredicateMaskField(pm_f)]) } IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask) => HasDirectPerm(Mask, null, pm_f) && IsPredicateField(pm_f) =>
```
Heap[null, UnitPredicateMaskField(pm_f)] == ExhaleHeap[null, UnitPredicateMaskField(pm_f)]

// Frame all wand mask locations of wands with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, TotalMask: MaskType, pm_f: (Field C FrameType)) :
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask), IsWandField(pm_f), ExhaleHeap[null, WandMaskField(pm_f)] }
  IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask) ==> HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) ==> Heap[null, WandMaskField(pm_f)] == ExhaleHeap[null, WandMaskField(pm_f)]

// Frame all locations in the footprint of magic wands
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, TotalMask: MaskType, pm_f: (Field C FrameType)) :
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask), IsWandField(pm_f) }
  IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask, TotalMask) ==> HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) ==> (forall <A, B> o2: Ref, f_1: (Field A B)) :
  { ExhaleHeap[o2, f_1] }
  Heap[null, WandMaskField(pm_f)][o2, f_1] ==> Heap[o2, f_1] == ExhaleHeap[o2, f_1]

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Preamble of Permission module.
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

type Perm = real;
type MaskType = <A, B> [Ref, Field A B] Perm;
var Mask: MaskType;
var TotalMask: MaskType;
const ZeroMask: MaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B)) :
  { ZeroMask[o_1, f_2] }
  ZeroMask[o_1, f_2] == NoPerm
);
type WandMaskType = <A, B> [Ref, Field A B] bool;
const ZeroWandMask: WandMaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B)) :
  { ZeroWandMask[o_1, f_2] }
  ZeroWandMask[o_1, f_2] == NoPerm
);
axiom FullPerm == 1.000000000;
function Perm(a: real, b: real): Perm;
function GoodHeap(Heap: HeapType): bool;
function GoodMask(Mask: MaskType): bool;
function GoodTotalMask(TotalMask: MaskType): bool;
axiom (forall Heap: HeapType, Mask: MaskType, TotalMask: MaskType ::
    { state(Heap, Mask, TotalMask) }
    state(Heap, Mask, TotalMask) ==> GoodHeap(Heap) && GoodMask(Mask)
    && GoodTotalMask(TotalMask) )
axiom (forall <A> Heap: HeapType, o_1: Ref, f_2: (Field A MaskType) ::
    { GoodHeap(Heap), Heap[o_1, f_2] } 
    GoodHeap(Heap) ==> GoodMask(Heap[o_1, f_2])
);
axiom (forall <A, B> TotalMask: MaskType, o_1: Ref, f_2: (Field A B) ::
    { GoodTotalMask(TotalMask), TotalMask[o_1, f_2] } 
    (GoodTotalMask(TotalMask) && !IsPredicateField(f_2) &&
    !IsWandField(f_2)) ==> TotalMask[o_1, f_2] >= NoPerm &&
    TotalMask[o_1, f_2] <= FullPerm) &&
    ((GoodTotalMask(TotalMask) && IsPredicateField(f_2) ||
    IsWandField(f_2))) ==> TotalMask[o_1, f_2] == NoPerm)
axiom (forall <A, B> Mask: MaskType, o_1: Ref, f_2: (Field A B) ::
    { GoodMask(Mask), Mask[o_1, f_2] } 
    GoodMask(Mask) ==> Mask[o_1, f_2] >= NoPerm && ((GoodMask(Mask)
    && !IsPredicateField(f_2)) && !IsWandField(f_2) ==> Mask[o_1,
    f_2] <= FullPerm)
);
function HasDirectPerm<A, B>(Mask: MaskType, o_1: Ref, f_2: (Field A B)): bool;
axiom (forall <A, B> Mask: MaskType, o_1: Ref, f_2: (Field A B) ::
    { HasDirectPerm(Mask, o_1, f_2) } 
    HasDirectPerm(Mask, o_1, f_2) ==> Mask[o_1, f_2] > NoPerm
);
function sumMask(ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType): bool;
axiom (forall <A, B> ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType, o_1: Ref, f_2: (Field A B) ::
    { sumMask(ResultMask, SummandMask1, SummandMask2),
    ResultMask[o_1, f_2] } 
    { sumMask(ResultMask, SummandMask1, SummandMask2),
    SummandMask1[o_1, f_2] } 
    { sumMask(ResultMask, SummandMask1, SummandMask2),
    SummandMask2[o_1, f_2] } 
    { sumMask(ResultMask, SummandMask1, SummandMask2),
    SummandMask1[o_1, f_2] +
    SummandMask2[o_1, f_2] } 
    sumMask(ResultMask, SummandMask1, SummandMask2) ==>
    ResultMask[o_1, f_2] == SummandMask1[o_1, f_2] + SummandMask2[o_1, f_2]
);
type FrameType;
const EmptyFrame : FrameType;
function FrameFragment <T>(t : T) : FrameType;
function ConditionalFrame(p : Perm, f_4 : FrameType) : FrameType;
function dummyFunction <T>(t : T) : bool;
function CombineFrames(a_1 : FrameType, b_1 : FrameType) : FrameType;

// Definition of conditional frame fragments

axiom (forall p : Perm, f_4 : FrameType ::
  ConditionalFrame(p, f_4) == (if p > 0.000000000 then f_4 else EmptyFrame)) ;

// Function for recording enclosure of one predicate instance in another
function InsidePredicate <A, B>(p : (Field A FrameType), v_1 : FrameType, q : (Field B FrameType), w : FrameType) : bool;

// Transitivity of InsidePredicate
axiom (forall <A, B, C> p : (Field A FrameType), v_1 : FrameType, q : (Field B FrameType), w : FrameType, r : (Field C FrameType), u : FrameType :
  InsidePredicate(p, v_1, q, w) && InsidePredicate(q, w, r, u) ==>
  InsidePredicate(p, v_1, r, u)) ;

// Knowledge that two identical instances of the same predicate cannot be inside each other
axiom (forall <A> p : (Field A FrameType), v_1 : FrameType, w : FrameType :
  !InsidePredicate(p, v_1, p, w) && !InsidePredicate(p, v_1, p, w)) ;

// Translation of domain Assume

// The type for domain Assume
type AssumeDomainType;

// Translation of all fields

const unique f_5 : Field NormalField int;
axiom !IsPredicateField(f_5);
axiom !IsWandField(f_5);

// Translation of predicate p
type PredicateType_p;

function p_1(x: Ref): Field PredicateType_p FrameType;
function p#sm(x: Ref): Field PredicateType_p MaskType;

axiom (forall x: Ref ::
  { UnitPredicateMaskField(p_1(x)) }
  UnitPredicateMaskField(p_1(x)) == p#sm(x)
);

axiom (forall x: Ref ::
  { p_1(x) }
  IsPredicateField(p_1(x))
);

axiom (forall x: Ref ::
  { p_1(x) }
  getPredicateId(p_1(x)) == 0
);

function p#trigger<A>(Heap: HeapType, pred: (Field A FrameType)): bool;

function p#everUsed<A>(pred: (Field A FrameType)): bool;

axiom (forall x: Ref, x2: Ref ::
  { p_1(x), p_1(x2) }
  p_1(x) == p_1(x2) ==> x == x2
);

axiom (forall x: Ref, x2: Ref ::
  { p#sm(x), p#sm(x2) }
  p#sm(x) == p#sm(x2) ==> x == x2
);

axiom (forall Heap: HeapType, x: Ref ::
  { p#trigger(Heap, p_1(x)) }
  p#everUsed(p_1(x))
);

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Translation of method m

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure m() returns ()
modifies Heap, Mask, TotalMask;
{
  var perm: Perm;
  var predicatePerm: Perm;
  var _1: Ref;
  var _2: Ref;
  var _3: Ref;
  var newVersion: FrameType;

  // — Initializing the state
  assume (forall <T, A, B> f_2: (Field T MaskType), o_3: Ref, f_3: (Field A B) ::
    { Heap[null, f_2][o_3, f_3] }
  Heap[null, f_2][o_3, f_3] == NoPerm
  );
  Mask := ZeroMask;
  TotalMask := ZeroMask;
assume state(Heap, Mask, TotalMask);

// -- Initializing of old state

// -- Initializing the old state
assume Heap == old(Heap);
assume Mask == old(Mask);
assume TotalMask == old(TotalMask);

// -- Translating statement: inhale acc(p(_1), write)
perm := FullPerm;
TotalMask[1, f_5] := TotalMask[1, f_5] - Heap[null, UnitPredicateMaskField(p_1(_1))][1, f_5] * Mask[null, p_1(_1)];
assert TotalMask[1, f_5] >= NoPerm && TotalMask[1, f_5] <= FullPerm;
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] + perm;
Heap[null, UnitPredicateMaskField(p_1(_1))][1, f_5] := FullPerm;
TotalMask[1, f_5] := TotalMask[1, f_5] + Heap[null, UnitPredicateMaskField(p_1(_1))][1, f_5] * Mask[null, p_1(_1)];
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assert TotalMask[1, f_5] >= NoPerm && TotalMask[1, f_5] <= FullPerm;

// -- Translating statement: inhale acc(p(_2), write)
perm := FullPerm;
TotalMask[2, f_5] := TotalMask[2, f_5] - Heap[null, UnitPredicateMaskField(p_1(_2))][2, f_5] * Mask[null, p_1(_2)];
assert TotalMask[2, f_5] >= NoPerm && TotalMask[2, f_5] <= FullPerm;
Mask[null, p_1(_2)] := Mask[null, p_1(_2)] + perm;
Heap[null, UnitPredicateMaskField(p_1(_2))][2, f_5] := FullPerm;
TotalMask[2, f_5] := TotalMask[2, f_5] + Heap[null, UnitPredicateMaskField(p_1(_2))][2, f_5] * Mask[null, p_1(_2)];
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assert TotalMask[2, f_5] >= NoPerm && TotalMask[2, f_5] <= FullPerm;

// -- Translating statement: inhale acc(p(_3), write)
perm := FullPerm;
TotalMask[3, f_5] := TotalMask[3, f_5] - Heap[null, UnitPredicateMaskField(p_1(_3))][3, f_5] * Mask[null, p_1(_3)];
assert TotalMask[3, f_5] >= NoPerm && TotalMask[3, f_5] <= FullPerm;
Mask[null, p_1(_3)] := Mask[null, p_1(_3)] + perm;
Heap[null, UnitPredicateMaskField(p_1(_3))][3, f_5] := FullPerm;
FullPerm;
UnitPredicateMaskField[p_1(3)][3, 5] * Mask[null, p_1(3)];
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assert TotalMask[3, 5] >= NoPerm && TotalMask[3, 5] <= 
FullPerm;

// — Translating statement: unfold acc(p(_1), write)
assume p#trigger(Heap, p_1(_1));
assume Heap[null, p_1(_1)] == FrameFragment(Heap[1, 5]);
// Phase 1: pure assertions and fixed permissions
predicatePerm := NoPerm;
predicatePerm := predicatePerm + FullPerm;
if (predicatePerm != NoPerm) {
    assert { : msg " Unfolding p(_1) might fail. There might be insufficient permission to access p(_1)." }
predicatePerm <= Mask[null, p_1(_1)];
}
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] - predicatePerm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_1))) {
    havoc newVersion;
    Heap[null, p_1(_1)] := newVersion;
}
perm := FullPerm;
assume _1 != null;
Mask[1, 5] := Mask[1, 5] + perm;
TotalMask[1, 5] := TotalMask[1, 5] - Heap[null, 
UnitPredicateMaskField[p_1(1)][1, 5] * predicatePerm + perm;
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);

// — Translating statement: unfold acc(p(_2), write)
assume p#trigger(Heap, p_1(_2));
assume Heap[null, p_1(_2)] == FrameFragment(Heap[2, 5]);
// Phase 1: pure assertions and fixed permissions
predicatePerm := NoPerm;
predicatePerm := predicatePerm + FullPerm;
if (predicatePerm != NoPerm) {
    assert { : msg " Unfolding p(_2) might fail. There might be insufficient permission to access p(_2)." }
predicatePerm <= Mask[null, p_1(_2)];
}
Mask[null, p_1(_2)] := Mask[null, p_1(_2)] - predicatePerm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_2))) {
    havoc newVersion;
    Heap[null, p_1(_2)] := newVersion;
}
perm := FullPerm;
assume _2 != null;
Mask[_2, f_5] := Mask[_2, f_5] + perm;
TotalMask[_2, f_5] := TotalMask[_2, f_5] - Heap[null, UnitPredicateMaskField(p_1(_2))][_2, f_5] * predicatePerm + perm;
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);

// --- Translating statement: unfold acc(p(_3), write)
assume p#trigger(Heap, p_1(_3));
assume Heap|null, p_1(_3)] == FrameFragment(Heap[_3, f_5]);
// Phase 1: pure assertions and fixed permissions
predicatePerm := NoPerm;
predicatePerm := predicatePerm + FullPerm;
if (predicatePerm != NoPerm) {
    assert {msg " Unfolding p(_3) might fail. There might be insufficient permission to access p(_3)."}
    predicatePerm <= Mask|null, p_1(_3)];
}
Mask|null, p_1(_3)] := Mask|null, p_1(_3)] - predicatePerm;

// --- Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_3))) {
    havoc newVersion;
    Heap[null, p_1(_3)] := newVersion;
}
perm := FullPerm;
assume _3 != null;
Mask[_3, f_5] := Mask[_3, f_5] + perm;
TotalMask[_3, f_5] := TotalMask[_3, f_5] - Heap|null, UnitPredicateMaskField(p_1(_3))][_3, f_5] * predicatePerm + perm;
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);
assume state(Heap, Mask, TotalMask);

// --- Translating statement: _1.f := 7
assert {msg " Assignment might fail. There might be insufficient permission to access _1.val_int."}
FullPerm == Mask[_1, f_5];
Heap[_1, f_5] := 7;
assume state(Heap, Mask, TotalMask);

// --- Translating statement: _2.f := _1.f
// --- Check definedness of _1.f
assert { :msg " Assignment might fail. There might be insufficient permission to access _1.f." }
    HasDirectPerm(Mask, _1, f_5);
assume state(Heap, Mask, TotalMask);
assert { :msg " Assignment might fail. There might be insufficient permission to access _2.f." }
    FullPerm == Mask[ _2, f_5];
    Heap[ _2, f_5] := Heap[ _1, f_5];
assume state(Heap, Mask, TotalMask);

// — Translating statement: _3.f := _2.f

// — Check definedness of _2.f
assert { :msg " Assignment might fail. There might be insufficient permission to access _2.f." }
    HasDirectPerm(Mask, _2, f_5);
assume state(Heap, Mask, TotalMask);
assert { :msg " Assignment might fail. There might be insufficient permission to access _3.f." }
    FullPerm == Mask[ _3, f_5];
    Heap[ _3, f_5] := Heap[ _2, f_5];
assume state(Heap, Mask, TotalMask);

// — Translating statement: assert _3.f == 7

// — Check definedness of _3.f == 7
assert { :msg " Assert might fail. There might be insufficient permission to access _3.f." }
    HasDirectPerm(Mask, _3, f_5);
assume state(Heap, Mask, TotalMask);
// Phase 1: pure assertions and fixed permissions
assert { :msg " Assert might fail. Assertion _3.f == 7 might not hold." }
    Heap[ _3, f_5] == 7;
assume state(Heap, Mask, TotalMask);
}

Modified to add predicate limit (predicate-limit):

// Translation of Viper program.
// Based on output from carbon with —disableAllocEncoding and —disableSuccessorHeaps.
// Hand-modified to prototype the concept of permission limits for predicates.
// Preamble of State module.
function state(Heap: HeapType, Mask: MaskType): bool;
    // ==============================================================
    // Preamble of Alternative heap module.
    // ==============================================================
    type Ref;
    var Heap: HeapType;
    const null: Ref;
    type Field A B;
    type NormalField;
    type HeapType = <A, B> [Ref, Field A B]B;
    function IdenticalOnKnownLocations(Heap: HeapType, ExhaleHeap: 
        HeapType, Mask: MaskType): bool;
    function IsPredicateField<A, B>(f: (Field A B)): bool;
    function IsWandField<A, B>(f: (Field A B)): bool;
    function getPredicateId<A, B>(f: (Field A B)): int;
    // Frame all allocations with direct permissions
    axiom (forall <A, B> Heap: HeapType, ExhaleHeap: HeapType, Mask: 
        MaskType, o: Ref, f_1: (Field A B) :: 
        { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask), 
        ExhaleHeap[o, f_1] } 
        IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) => 
        HasDirectPerm(Mask, o, f_1) => Heap[o, f_1] = ExhaleHeap[o, f_1] 
    );
    // Frame all predicate mask locations of predicates with direct 
    // permission
    axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: 
        MaskType, pm_f: (Field C FrameType) :: 
        { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask), 
        IsPredicateField(pm_f), ExhaleHeap[null, 
        PredicateMaskField(pm_f)] } 
        IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) => 
        HasDirectPerm(Mask, null, pm_f) & & IsPredicateField(pm_f) => 
        Heap[null, PredicateMaskField(pm_f)] = ExhaleHeap[null, 
        PredicateMaskField(pm_f)] 
    );
    // Frame all allocations with known folded permissions
    axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: 
        MaskType, pm_f: (Field C FrameType) :: 
        { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask), 
        ExhaleHeap[null, pm_f], IsPredicateField(pm_f) } 
        IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) => 
        HasDirectPerm(Mask, null, pm_f) & & IsPredicateField(pm_f) => 
        (forall <A, B> o2: Ref, f_1: (Field A B) :: 
        { ExhaleHeap[o2, f_1] } 
        Heap[null, PredicateMaskField(pm_f)][o2, f_1] => 
        Heap[o2, f_1] = ExhaleHeap[o2, f_1] 
    );
    // Frame all wand mask locations of wands with direct permission
    axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: 
    //
\begin{verbatim}
MaskType, pm_f : (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
  IsWandField(pm_f), ExhaleHeap[null, WandMaskField(pm_f)] } 
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) \implies
HasDirectPerm(Mask, null, pm_f) \&\& IsWandField(pm_f) \implies
Heap[null, WandMaskField(pm_f)] \equiv ExhaleHeap[null, 
  WandMaskField(pm_f)]
}

// Frame all locations in the footprint of magic wands
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
  MaskType, pm_f: (Field C FrameType) ::
{ IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
  IsWandField(pm_f) }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) \implies
HasDirectPerm(Mask, null, pm_f) \&\& IsWandField(pm_f) \implies (forall
  <A, B> o2: Ref, f_1: (Field A B) ::
  { ExhaleHeap[o2, f_1] }
Heap[null, WandMaskField(pm_f)][o2, f_1] \implies Heap[o2, f_1] ==
  ExhaleHeap[o2, f_1]
)

// Preamble of Permission module.

type Perm = real;
type MaskType = <A, B>[Ref, Field A B]Perm;
var Mask : MaskType;
const ZeroMask : MaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B) ::
  { ZeroMask[o_1, f_2] }
ZeroMask[o_1, f_2] == NoPerm
);
type PMaskType = <A, B>[Ref, Field A B]bool;
const ZeroPMask : PMaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B) ::
  { ZeroPMask[o_1, f_2] }
!ZeroPMask[o_1, f_2]
);
function PredicateMaskField<A>(f_3: (Field A FrameType)): Field A
  PMaskType;
function WandMaskField<A>(f_3: (Field A FrameType)): Field A
  PMaskType;
const NoPerm: Perm;
axiom NoPerm == 0.000000000;
const FullPerm: Perm;
axiom FullPerm == 1.000000000;
function Perm(a: real, b: real): Perm;
function GoodMask(Mask: MaskType): bool;
axiom (forall Heap: HeapType, Mask: MaskType ::
  { state(Heap, Mask) }
state(Heap, Mask) \implies GoodMask(Mask)
);
\end{verbatim}
Prusti synthetic benchmark prototypes

C-20

axiom (forall <A, B> Mask: MaskType, o_1: Ref, f_2: (Field A B) ::
  { GoodMask(Mask), Mask[o_1, f_2] }
GoodMask(Mask) ==> Mask[o_1, f_2] >= NoPerm && ((GoodMask(Mask) && !IsPredicateField(f_2)) && !IsWandField(f_2) ==> Mask[o_1, f_2] <= FullPerm)
);

// Enforce explicitly annotated predicate limits
axiom (forall Mask: MaskType, f_2: Field PredicateType_p FrameType ::
  { GoodMask(Mask), Mask[null, f_2] }
GoodMask(Mask) ==> Mask[null, f_2] <= FullPerm
);

function HasDirectPerm<A, B>(Mask: MaskType, o_1: Ref, f_2: (Field A B)): bool;

function sumMask(ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType): bool;

axiom (forall <A, B> ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType, o_1: Ref, f_2: (Field A B) ::
  { sumMask(ResultMask, SummandMask1, SummandMask2)
ResultMask[o_1, f_2] } { sumMask(ResultMask, SummandMask1, SummandMask2), SummandMask1[o_1, f_2] } { sumMask(ResultMask, SummandMask1, SummandMask2), SummandMask2[o_1, f_2] }
sumMask(ResultMask, SummandMask1, SummandMask2) ==> ResultMask[o_1, f_2] == SummandMask1[o_1, f_2] + SummandMask2[o_1, f_2]
);

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Preamble of Function and predicate module.
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// Declarations for function framing

type FrameType;
const EmptyFrame: FrameType;
function FrameFragment<T>(t: T): FrameType;
function ConditionalFrame(p: Perm, f_4: FrameType): FrameType;
function dummyFunction<T>(t: T): bool;
function CombineFrames(a_1: FrameType, b_1: FrameType): FrameType;

// Definition of conditional frame fragments

axiom (forall p: Perm, f_4: FrameType ::
  { ConditionalFrame(p, f_4) }
ConditionalFrame(p, f_4) == (if p > 0.000000000 then f_4 else EmptyFrame)
);

// Function for recording enclosure of one predicate instance in
function InsidePredicate<\text{A}, \text{B}> (p: (Field \text{A} FrameType), v_1: FrameType, q: (Field \text{B} FrameType), w: FrameType): bool;

// Transitivity of InsidePredicate
axiom (forall <\text{A}, \text{B}, \text{C}> p: (Field \text{A} FrameType), v_1: FrameType, q: (Field \text{B} FrameType), w: FrameType, r: (Field \text{C} FrameType), u: FrameType ::
{ InsidePredicate(p, v_1, q, w), InsidePredicate(q, w, r, u) }
InsidePredicate(p, v_1, q, w) && InsidePredicate(q, w, r, u) =>
InsidePredicate(p, v_1, r, u));

// Knowledge that two identical instances of the same predicate
cannot be inside each other
axiom (forall <\text{A}> p: (Field \text{A} FrameType), v_1: FrameType, w: FrameType ::
{ InsidePredicate(p, v_1, p, w) }
!InsidePredicate(p, v_1, p, w));

// Translation of domain Assume
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

type AssumeDomainType;

// Translation of all fields
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

const unique f_5: Field NormalField int;
axiom !IsPredicateField(f_5);
axiom !IsWandField(f_5);

// Translation of predicate p
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
type PredicateType_p;
function p_1(x: Ref): Field PredicateType_p FrameType;
function p#sm(x: Ref): Field PredicateType_p PMaskType;
axiom (forall x: Ref ::
{ PredicateMaskField(p_1(x)) }
PredicateMaskField(p_1(x)) == p#sm(x));
axiom (forall x: Ref ::
{ p_1(x) }
IsPredicateField(p_1(x)));
axiom (forall x: Ref ::
{ p_1(x) }
getPredicateId(p_1(x)) == 0);
function p#trigger<\text{A}>(Heap: HeapType, pred: (Field \text{A} FrameType));


```plaintext
bool;
function p#everUsed<A>(pred: (Field A FrameType)): bool;
axiom (forall x: Ref, x2: Ref ::
  { p_1(x), p_1(x2) }
  p_1(x) == p_1(x2) ==> x == x2);
axiom (forall x: Ref, x2: Ref ::
  { p#sm(x), p#sm(x2) }
  p#sm(x) == p#sm(x2) ==> x == x2);

axiom (forall Heap: HeapType, x: Ref ::
  { p#trigger(Heap, p_1(x)) }
  p#everUsed(p_1(x)));

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Translation of method m
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure m() returns ()
 modifies Heap, Mask;
{
  var perm : Perm;
  var _1 : Ref;
  var _2 : Ref;
  var _3 : Ref;
  var newVersion : FrameType;
  // — Initializing the state
  Mask := ZeroMask;
  assume state(Heap, Mask);
  // — Initializing of old state
  assume Heap == old(Heap);
  assume Mask == old(Mask);
  // — Translating statement: inhale acc(p(_1), write)
  perm := FullPerm;
  Mask[null, p_1(_1)] := Mask[null, p_1(_1)] + perm;
  assume state(Heap, Mask);
  assume state(Heap, Mask);
  // — Translating statement: inhale acc(p(_2), write)
  perm := FullPerm;
  Mask[null, p_1(_2)] := Mask[null, p_1(_2)] + perm;
  assume state(Heap, Mask);
  assume state(Heap, Mask);
```
perm := FullPerm;
Mask[null, p_1(-3)] := Mask[null, p_1(-3)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// — Translating statement: unfold acc(p(_1), write)
assume p#trigger(Heap, p_1(-1));
assume Heap[null, p_1(-1)] == FrameFragment(Heap[-1, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
  assert {: msg "Unfolding p(_1) might fail. There might be insufficient permission to access p(_1)."}
  perm <= Mask[null, p_1(-1)];
}
Mask[null, p_1(-1)] := Mask[null, p_1(-1)] - perm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(-1))) {
  havoc newVersion;
  Heap[null, p_1(-1)] := newVersion;
}
perm := FullPerm;
assume _1 != null;
Mask[-1, f_5] := Mask[-1, f_5] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// — Translating statement: unfold acc(p(_2), write)
assume p#trigger(Heap, p_1(-2));
assume Heap[null, p_1(-2)] == FrameFragment(Heap[-2, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
  assert {: msg "Unfolding p(_2) might fail. There might be insufficient permission to access p(_2)."}
  perm <= Mask[null, p_1(-2)];
}
Mask[null, p_1(-2)] := Mask[null, p_1(-2)] - perm;

// — Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(-2))) {
  havoc newVersion;
Heap[null, p_1(_2)] := newVersion;

perm := FullPerm;
assume _2 != null;
Mask[_2, f_5] := Mask[_2, f_5] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: unfold acc(p(_3), write)
assume p#trigger(Heap, p_1(_3));
assume Heap[null, p_1(_3)] == FrameFragment(Heap[_3, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert { : msg " Unfolding p(_3) might fail. There might be insufficient permission to access p(_3)." }
    perm <= Mask[null, p_1(_3)];
}
Mask[null, p_1(_3)] := Mask[null, p_1(_3)] - perm;

// --- Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_3))) {
    havoc newVersion;
    Heap[null, p_1(_3)] := newVersion;
}
perm := FullPerm;
assume _3 != null;
Mask[_3, f_5] := Mask[_3, f_5] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: _1.f := 7
assert { : msg " Assignment might fail. There might be insufficient permission to access _1.val_int." }
FullPerm == Mask[_1, f_5];
Heap[_1, f_5] := 7;
assume state(Heap, Mask);

// --- Translating statement: _2.f := _1.f

// --- Check definedness of _1.f
assert { : msg " Assignment might fail. There might be insufficient permission to access _1.f." }
HasDirectPerm(Mask, _1, f_5);
assume state(Heap, Mask);
assert { : msg " Assignment might fail. There might be insufficient permission to access _2.f." }
FullPerm == Mask[\_2, \_5];
Heap[\_2, \_5] := Heap[\_1, \_5];
assume state(Heap, Mask);

// --- Translating statement: \_3.f := \_2.f
// --- Check definedness of \_2.f
assert { :msg " Assignment might fail. There might be insufficient permission to access \_2.f." }
HasDirectPerm(Mask, \_2, \_5);
assume state(Heap, Mask);
assert { :msg " Assignment might fail. There might be insufficient permission to access \_3.f." }
FullPerm == Mask[\_3, \_5];
Heap[\_3, \_5] := Heap[\_2, \_5];
assume state(Heap, Mask);

// --- Translating statement: assert \_3.f == 7
// --- Check definedness of \_3.f == 7
assert { :msg " Assert might fail. There might be insufficient permission to access \_3.f." }
HasDirectPerm(Mask, \_3, \_5);
assume state(Heap, Mask);
// Phase 1: pure assertions and fixed permissions
assert { :msg " Assert might fail. Assertion \_3.f == 7 might not hold." }
Heap[\_3, \_5] == 7;
assume state(Heap, Mask);

Modified to assign fresh values to variables (new):

// Translation of Viper program.
// Based on output from carbon with --disableSuccessorHeaps (but without --disableAllocEncoding).
// Hand-modified to prototype to explicitly specify disjointness of local variables.
// This could also be generated by modifying the Viper file to add "\_1 := new()", "\_2 := new()", etc.
//
// Preamble of State module.

function state(Heap: HeapType, Mask: MaskType): bool;
type Ref;
var Heap: HeapType;
const null: Ref;
type Field A B;
type NormalField;
type HeapType = <A, B> [Ref, Field A B];
const unique $allocated: Field NormalField bool;
axiom (forall o: Ref, f: (Field NormalField Ref), Heap: HeapType ::
  { Heap[o, f] } 
  Heap[o, $allocated] => Heap[Heap[o, f], $allocated] 
); 
function IdenticalOnKnownLocations(Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType): bool;
function IsPredicateField <A, B>(f: (Field A B)): bool;
function IsWandField <A, B>(f: (Field A B)): bool;
function getPredicateId <A, B>(f: (Field A B)): int;
// Frame all allocations with direct permissions
axiom (forall <A, B> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, o: Ref, f_1: (Field A B) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    ExhaleHeap[o, f_1] } 
  IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) =>
  HasDirectPerm(Mask, o, f_1) => Heap[o, f_1] == ExhaleHeap[o, f_1] 
);
// Frame all predicate mask locations of predicates with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, pm_f: (Field C FrameType) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    IsPredicateField(pm_f), ExhaleHeap>null,
    PredicateMaskField(pm_f) } 
  IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) =>
  HasDirectPerm(Mask, null, pm_f) && IsPredicateField(pm_f) =>
  Heap>null, PredicateMaskField(pm_f) == ExhaleHeap>null,
  PredicateMaskField(pm_f) 
);
// Frame all locations with known folded permissions
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask: MaskType, pm_f: (Field C FrameType) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    ExhaleHeap>null, pm_f, IsPredicateField(pm_f) } 
  IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) =>
  HasDirectPerm(Mask, null, pm_f) && IsPredicateField(pm_f) =>
  (forall <A, B> o2: Ref, f_1: (Field A B) ::
    { ExhaleHeap[o2, f_1] } 
    ExhaleHeap>null, PredicateMaskField(pm_f)[o2, f_1] => Heap[o2, f_1] == ExhaleHeap[o2, f_1] 
  ) 
);
// Frame all wand mask locations of wands with direct permission
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
  MaskType, pm_f: (Field C FrameType)) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    IsWandField(pm_f), ExhaleHeap[null, WandMaskField(pm_f)] }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) =>
HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) =>
Heap[null, WandMaskField(pm_f)] == ExhaleHeap[null, WandMaskField(pm_f)]
)

// Frame all locations in the footprint of magic wands
axiom (forall <C> Heap: HeapType, ExhaleHeap: HeapType, Mask:
  MaskType, pm_f: (Field C FrameType)) ::
  { IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask),
    IsWandField(pm_f) }
IdenticalOnKnownLocations(Heap, ExhaleHeap, Mask) =>
HasDirectPerm(Mask, null, pm_f) && IsWandField(pm_f) => (forall
<A, B> o_2: Ref, f_1: (Field A B)) ::
  { ExhaleHeap[o_2, f_1] }
Heap[null, WandMaskField(pm_f)][o_2, f_1] == > Heap[o_2, f_1] ==
ExhaleHeap[o_2, f_1]
)

// Preamble of Permission module.

type Perm = real;
type MaskType = <A, B> [Ref, Field A B] Perm;
var Mask: MaskType;
const ZeroMask: MaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B)) ::
  { ZeroMask[o_1, f_2] }
ZeroMask[o_1, f_2] == NoPerm
)

type PMaskType = <A, B> [Ref, Field A B] bool;
const ZeroPMask: PMaskType;
axiom (forall <A, B> o_1: Ref, f_2: (Field A B)) ::
  { ZeroPMask[o_1, f_2] }
!ZeroPMask[o_1, f_2]
)

function PredicateMaskField<A>(f_3: (Field A FrameType)): Field A
  PMaskType;

function WandMaskField<A>(f_3: (Field A FrameType)): Field A
  PMaskType;

const NoPerm: Perm;
axiom NoPerm == 0.000000000;
const FullPerm: Perm;
axiom FullPerm == 1.000000000;

function GoodMask(Mask: MaskType): bool;

axiom (forall Heap: HeapType, Mask: MaskType ::
  { state(Heap, Mask) }
state(Heap, Mask) == > GoodMask(Mask)
\[ \begin{array}{l}
\text{GoodMask(Mask)} \implies \text{Mask}[o_1, f_2] \geq \text{NoPerm} \& \& \text{IsPredicateField(f_2)} \& \& \text{IsWandField(f_2)} \implies \text{Mask}[o_1, f_2] \leq \text{FullPerm} \\
\text{HasDirectPerm<(A, B)> (Mask: MaskType, o_1: Ref, f_2: (Field A B)) : bool;}
\end{array} \]

\[ \begin{array}{l}
\text{HasDirectPerm(Mask, o_1, f_2) \iff Mask}[o_1, f_2] > \text{NoPerm} \\
\text{sumMask(ResultMask: MaskType, SummandMask1: MaskType, SummandMask2: MaskType) : bool;}
\end{array} \]

\[ \begin{array}{l}
\text{sumMask(ResultMask, SummandMask1, SummandMask2)} \implies \\
\text{ResultMask}[o_1, f_2] \equiv \text{SummandMask1}[o_1, f_2] + \text{SummandMask2}[o_1, f_2]
\end{array} \]
\[
\{ \text{InsidePredicate}(p, v_1, q, w), \text{InsidePredicate}(q, w, r, u) \}
\]

\[
\text{InsidePredicate}(p, v_1, q, w) \land \text{InsidePredicate}(q, w, r, u) \implies
\text{InsidePredicate}(p, v_1, r, u)
\]

// Knowledge that two identical instances of the same predicate
// cannot be inside each other
axiom (forall \< A \> p: (Field A FrameType), v_1: FrameType, w: FrameType ::
\[
\{ \text{InsidePredicate}(p, v_1, p, w) \}
\]

\[
!\text{InsidePredicate}(p, v_1, p, w)
\]

// Translation of domain Assume
// Translation of all fields
// Translation of predicate p

class (forall x: Ref ::
\[
\{ \text{PredicateField}(p_1(x)) \}
\]

\[
\text{PredicateField}(p_1(x)) \equiv p#sm(x)
\]

axiom (forall x: Ref ::
\[
\{ p_1(x) \}
\]

\[
!\text{PredicateField}(p_1(x))\]

getPredicateId(p_1(x)) == 0
)

function p#trigger<\< A \>>(Heap: HeapType, pred: (Field A FrameType)):
bool;

function p#everUsed<\< A \>>(pred: (Field A FrameType)):
bool;

axiom (forall x: Ref, x2: Ref ::
\[
\{ p_1(x), p_1(x2) \}
\]

\[
p_1(x) == p_1(x2) \implies x == x2
\]

axiom (forall x: Ref, x2: Ref ::
{ p#sm(x), p#sm(x_2) }

p#sm(x) == p#sm(x_2) ==> x == x_2;

axiom (forall Heap: HeapType, x: Ref ::

{ p#trigger(Heap, p_1(x)) }

p#everUsed(p_1(x))
);

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Translation of method m
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

procedure m() returns ()
modifies Heap, Mask;
{
  var perm: Perm;
  var _1: Ref;
  var _2: Ref;
  var _3: Ref;
  var newVersion: FrameType;
  var freshObj: Ref;

  // — Initializing the state
  Mask := ZeroMask;
  assume state(Heap, Mask);

  // — Initializing of old state

  // — Initializing the old state
  assume Heap == old(Heap);
  assume Mask == old(Mask);

  // — Assumptions about local variables
  assume Heap[_1, $allocated];
  assume Heap[_2, $allocated];
  assume Heap[_3, $allocated];

  // — Translating statement: _1 := new()
  havoc freshObj;
  assume freshObj != null && !Heap[freshObj, $allocated];
  Heap[freshObj, $allocated] := true;
  _1 := freshObj;
  assume state(Heap, Mask);

  // — Translating statement: _2 := new()
  havoc freshObj;
  assume freshObj != null && !Heap[freshObj, $allocated];
  Heap[freshObj, $allocated] := true;
  _2 := freshObj;
  assume state(Heap, Mask);
// --- Translating statement: _3 := new()

havoc freshObj;
assume freshObj != null && !Heap[freshObj, $allocated];
Heap[freshObj, $allocated] := true;
_3 := freshObj;
assume state(Heap, Mask);

// --- Translating statement: inhale acc(p(_1), write)

perm := FullPerm;
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: inhale acc(p(_2), write)

perm := FullPerm;
Mask[null, p_1(_2)] := Mask[null, p_1(_2)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: inhale acc(p(_3), write)

perm := FullPerm;
Mask[null, p_1(_3)] := Mask[null, p_1(_3)] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: unfold acc(p(_1), write)

assume p#trigger(Heap, p_1(_1));
assume Heap[null, p_1(_1)] == FrameFragment(Heap[_1, f_5]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert {
        msg "Unfolding p(_1) might fail. There might be insufficient permission to access p(_1)."}
    perm <= Mask[null, p_1(_1)];
}
Mask[null, p_1(_1)] := Mask[null, p_1(_1)] - perm;

// --- Update version of predicate
if (!HasDirectPerm(Mask, null, p_1(_1))) {
    havoc newVersion;
    Heap[null, p_1(_1)] := newVersion;
}
perm := FullPerm;
assume _1 != null;
Mask[ _1, f_5 ] := Mask[ _1, f_5 ] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: unfold acc(p(_2), write)
assume p#trigger(Heap, p_1(_2));
assume Heap[ null , p_1(_2) ] == FrameFragment(Heap[ _2, f_5 ]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert { : msg " Unfolding p(_2) might fail. There might be
    insufficient permission to access p(_2)." }
    perm <= Mask[ null , p_1(_2) ];
}
Mask[ null , p_1(_2) ] := Mask[ null , p_1(_2) ] - perm;

// --- Update version of predicate
if ( ! HasDirectPerm (Mask, null, p_1(_2)) ) {
    havoc newVersion;
    Heap[ null , p_1(_2) ] := newVersion;
}
perm := FullPerm;
assume _2 != null;
Mask[ _2, f_5 ] := Mask[ _2, f_5 ] + perm;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: unfold acc(p(_3), write)
assume p#trigger(Heap, p_1(_3));
assume Heap[ null , p_1(_3) ] == FrameFragment(Heap[ _3, f_5 ]);
// Phase 1: pure assertions and fixed permissions
perm := NoPerm;
perm := perm + FullPerm;
if (perm != NoPerm) {
    assert { : msg " Unfolding p(_3) might fail. There might be
    insufficient permission to access p(_3)." }
    perm <= Mask[ null , p_1(_3) ];
}
Mask[ null , p_1(_3) ] := Mask[ null , p_1(_3) ] - perm;

// --- Update version of predicate
if ( ! HasDirectPerm (Mask, null, p_1(_3)) ) {
    havoc newVersion;
    Heap[ null , p_1(_3) ] := newVersion;
}
perm := FullPerm;
assume _3 != null;
assume state(Heap, Mask);
assume state(Heap, Mask);
assume state(Heap, Mask);

// --- Translating statement: _1.f := 7
assert { :msg " Assignment might fail. There might be insufficient permission to access _1.val_int.*" }
    FullPerm == Mask[1, f_5];
    Heap[1, f_5] := 7;
assume state(Heap, Mask);

// --- Translating statement: _2.f := _1.f
// --- Check definedness of _1.f
assert { :msg " Assignment might fail. There might be insufficient permission to access _1.f." }
    HasDirectPerm(Mask, _1, f_5);
assume state(Heap, Mask);
assert { :msg " Assignment might fail. There might be insufficient permission to access _2.f." }
    FullPerm == Mask[2, f_5];
    Heap[2, f_5] := Heap[1, f_5];
assume state(Heap, Mask);

// --- Translating statement: _3.f := _2.f
// --- Check definedness of _2.f
assert { :msg " Assignment might fail. There might be insufficient permission to access _2.f." }
    HasDirectPerm(Mask, _2, f_5);
assume state(Heap, Mask);
assert { :msg " Assignment might fail. There might be insufficient permission to access _3.f." }
    FullPerm == Mask[3, f_5];
    Heap[3, f_5] := Heap[2, f_5];
assume state(Heap, Mask);

// --- Translating statement: assert _3.f == 7
// --- Check definedness of _3.f == 7
assert { :msg " Assert might fail. There might be insufficient permission to access _3.f." }
    HasDirectPerm(Mask, _3, f_5);
assume state(Heap, Mask);
// Phase 1: pure assertions and fixed permissions
assert { :msg " Assert might fail. Assertion _3.f == 7 might not hold." }
    Heap[3, f_5] == 7;
assume state(Heap, Mask);