ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair of Programming Methodology
inf Informatik Computer Science

Master Thesis

Chair of Programming Methodology, Department of Computer Science, ETH Zurich

Inference of Pointwise Specifications for Heap Manipulating Programs

Severin Münger

Autumn Semester 2016

ETH student ID:	11-922-531
E-mail address:	muengers@student.ethz.ch
Supervisors:	Dr. Alexander J. Summers
	Dr. Caterina Urban
	Prof. Dr. Peter Müller
Date of submission:	March 20, 2017

Abstract

Research in the field of automated software verification has made tremendous progress over the last decades. Special logics such as separation logic and implicit dynamic frames allow efficient reasoning about general heap manipulating programs. Viper, a software verification infrastructure, is designed based on these techniques and incorporates the concept of permissions. In order for a program to successfully verify, the programmer has to require explicit ownership to each heap location that gets accessed in the program by declaring according permissions.

In this thesis, we propose an elaborate and general approach for the automatic inference of access permissions which unifies and improves existing algorithms. Moreover, we introduce a sound rule for propagating permissions over heap modifications. Integer-dependent heap accesses are handled by including the results of a numerical analysis into our algorithm using quantifier elimination. We extend programs with our inference results using quantified permissions which are natively supported by Viper. The presented algorithm is sound and able to infer permissions for a broad range of real programs efficiently. Finally, with our prototype implementation, we evaluate generated specifications for concrete Viper programs including a sophisticated tree traversal algorithm in terms of precision and how their readability benefits from various simplifications we employ.

Acknowledgements

I would like to offer my special thanks to both my supervisors Dr. Alexander J. Summers and Dr. Caterina Urban for taking their time every week to discuss pending problems and for giving a lot of useful inputs and new ideas. Towards the end of the thesis, they also gave me a lot of constructive feedback for my report. I would also like to thank Jérôme Dohrau for his support with various parts of the Sample implementation. Furthermore, I would like to express my gratitude to Prof. Dr. Peter Müller for giving me the opportunity to work on this interesting project and to everybody working on the Viper project. Finally, I would also like to thank my parents and my brother, who always supported and encouraged me during my studies and while writing this thesis.

Contents

1 Introduction

The field of Software Verification has seen tremendous progress over the last decades. Theoretical frameworks such as abstract interpretation or Hoare logic allow to prove or disprove certain properties of programs statically and have paved the way for automated software verification. In order to prove that a program is correct with respect to the properties in question there first needs to be a precise mathematical specification of the program to prove. The specification, also called contract, describes the properties that the program should have in a very precise mathematical manner that can be understood by the underlying verifier. For some properties this might be a straightforward task. However, specifying complex functional behaviors or heap manipulating programs can quickly result in contracts that are of bigger size than the program itself. This is one of the main reasons why static software verification is still not widely used in industry as opposed to, e.g, testing. Therefore, a lot of research in the field of software verification now focuses on improving the usability and on reducing the verbosity of the needed contracts.
Additional annotations are required especially when dealing with general programs that might access and modify the heap. Several special theories and techniques have been introduced that facilitate the handling of heap accesses and modifications, such as separation logic [22], an extension of Hoare logic, or implicit dynamic frames [23]. These theories introduce the notion of *permissions* which are used to declare ownership to parts of the heap and provide the ability to reason about parts of the program locally. However, on the downside, declaring permissions imposes an additional annotation overhead to the programmer. The Viper framework [20] developed at ETH Zurich is a comprehensive and flexible verification framework which incorporates permission-based reasoning by design.

1.1 Motivation

While allowing to specify heap manipulating programs, on the downside, theories such as separation logic or implicit dynamic frames, which Viper is based on, require explicit declarations of permissions to every heap location that gets accessed or modified at any point in the program. A big advantage of Viper compared to other permission-based frameworks is that it supports declaring universally quantified permissions [19]. Specifying those permissions is usually a rather trivial task compared to the specification of functional behavior but still they can impose considerable effort to the programmer writing the specification. Contracts mainly appear in three places in methods of a program: On method entries as preconditions, at loop heads as invariants and at the end of a method as postconditions. In order to make a heap-manipulating program verify successfully in Viper, method preconditions have to require enough permissions to enable all the field accesses and modifications occurring in the method. Loop invariants should guarantee enough permissions for any number of iterations of the loop body. Automatically inferring permission-related contracts would drastically improve the usability of an automated verification framework for heap manipulating programs such as Viper.

1.2 Goals and Outline

The general goal of this master thesis is to continue previous work on the automatic inference of quantified permission [24]. In particular, we generalize the approach presented in [24] and we address several of its limitations and unsoundness issues.
In the following we give an overview of the goals we set ourselves before starting this thesis.

Design of unified approach The work in [24] describes two specialized and disjoint approaches: one for handling arrays and one for handling general graph-like data structures. It has to be chosen before the analysis which specialized inference to use. We propose a new way of relating these two separate approaches and combine them to an elaborate unified and sound solution.

Quantifier elimination In order to deal with changing variables inside loops, [24] introduced the `forget` operator. A numerical analysis by abstract interpretation was used to infer all the possible values for a changing variable inside a loop. The result of this numerical analysis was used together with the result of a backward permission tracking analysis to generate the quantified permission assertions. Unfortunately, this forget operator in general imposed imprecision and also contained an unsoundness. We propose a more precise approach to project out changing variables using quantifier elimination (cf. Section 3.5).

Support heap manipulations We also support sound inference in case of heap-dependent receivers that potentially get changed themselves. In particular, we present a sound rule for propagating permissions over heap modifications which was not supported in [24]. Furthermore, we incorporate into our approach a heap analysis to facilitate the inference in certain cases (cf. Section 3.4).

Forward analysis to infer post-conditions Until now, only weakest preconditions have been inferred to require *at least* enough permissions for a method to verify. We also develop strongest postcondition rules for quantified permissions and explain how to extend our algorithm with an additional forward analysis using abstract interpretation (cf. Section 3.7).

Simplify generated specifications We will present various simplifications and optimizations of the generated specifications (cf. Section 3.8). All these enhancements improve readability a lot in general.

Furthermore, we will give an overview of the work done in [24] and a brief introduction to abstract interpretation and the Viper framework in Chapter 2. In Chapter 3 we will describe our algorithm in detail. This will lead over to the evaluation in terms of soundness and precision of our algorithm for concrete Viper programs in Chapter 4. Finally, Chapter 5 will cover possible extensions and improvements and especially how the limitations of our algorithm could be addressed in future work. We will conclude this thesis in Chapter 6.

2 Related Work

2.1 Previous Work

This master thesis is based on the work done by Seraiah Walter [24]. In the following we will shortly summarize the most important contributions of that work.
The main goal in [24] was to infer quantified permissions for two different kinds of Viper programs: Programs operating on arrays and programs operating on graph data structures. The developed approach consists of two separate algorithms. The algorithm to infer permissions for array programs employs a separate numerical analysis to infer constraints on integer-valued variables that occur in array accesses. This information is then used to project out changing integer variables in loops. Similarly, the graph algorithm projects out changing reference variables that are receivers of field accesses using sets.
One disadvantage of the approach in [24] is that it has to be decided manually which of the two algorithms to use. That also means that the approach is not able to handle programs that contain both array accesses and graph-like data structure accesses. Another shortcoming is that heap modifications in the graph algorithm are not supported. This led to the main motivations for our work.

2.2 The Viper Framework

Viper [20] is a verification framework that has its own intermediate verification language and provides flexibility to write front-ends for multiple high-level programming languages as well as interfaces for various back-ends. The modular structure of Viper is shown in Figure 2.1. The core component of Viper is its own intermediate verification language. The Viper language is sequential, imperative and object-based. A Viper program can consist of global field declarations, abstract and non-abstract functions, predicates, domains, and methods. Domains in turn consist of a sequence of abstract functions, so called domain functions, and axioms. Viper natively provides only a limited number of data types: `Int`, `Bool`, `Rational`, `Perm` (data type for permissions), `Ref` (data type for references), `Set[T]`, and `Seq[T]`. It is important to note that fields are only declared once per program and can be accessed on any `Ref`-typed receiver. The Viper language was designed with the balance between simplicity and expressiveness in mind. It can be used to handle modern high-level programming languages while still providing only a limited number of language features. Still, it is not overloaded with language features which makes it easier to develop efficient back-ends.
The modular structure of Viper minimizes the effort for a programmer to adapt a new programming language to the Viper framework. As soon as there is a complete front-end, i.e., a compiler that translates programs written in the original programming language to equivalent Viper programs, it is possible to verify any program written in this language. There already exist various front-ends, e.g., for Chalice [14], Java [2], or a subset of Scala.

On the back-end, there currently exist two verifiers that are part of Viper. Carbon, which basically translates Viper programs into Boogie [1] programs and Silicon, which is based on symbolic execution. Both back-ends rely on Microsoft's Z3 SMT solver [10].
There exists a specification inference framework for the Viper language, called Sample. It is based on abstract interpretation and we will introduce it in Section 2.3.1.

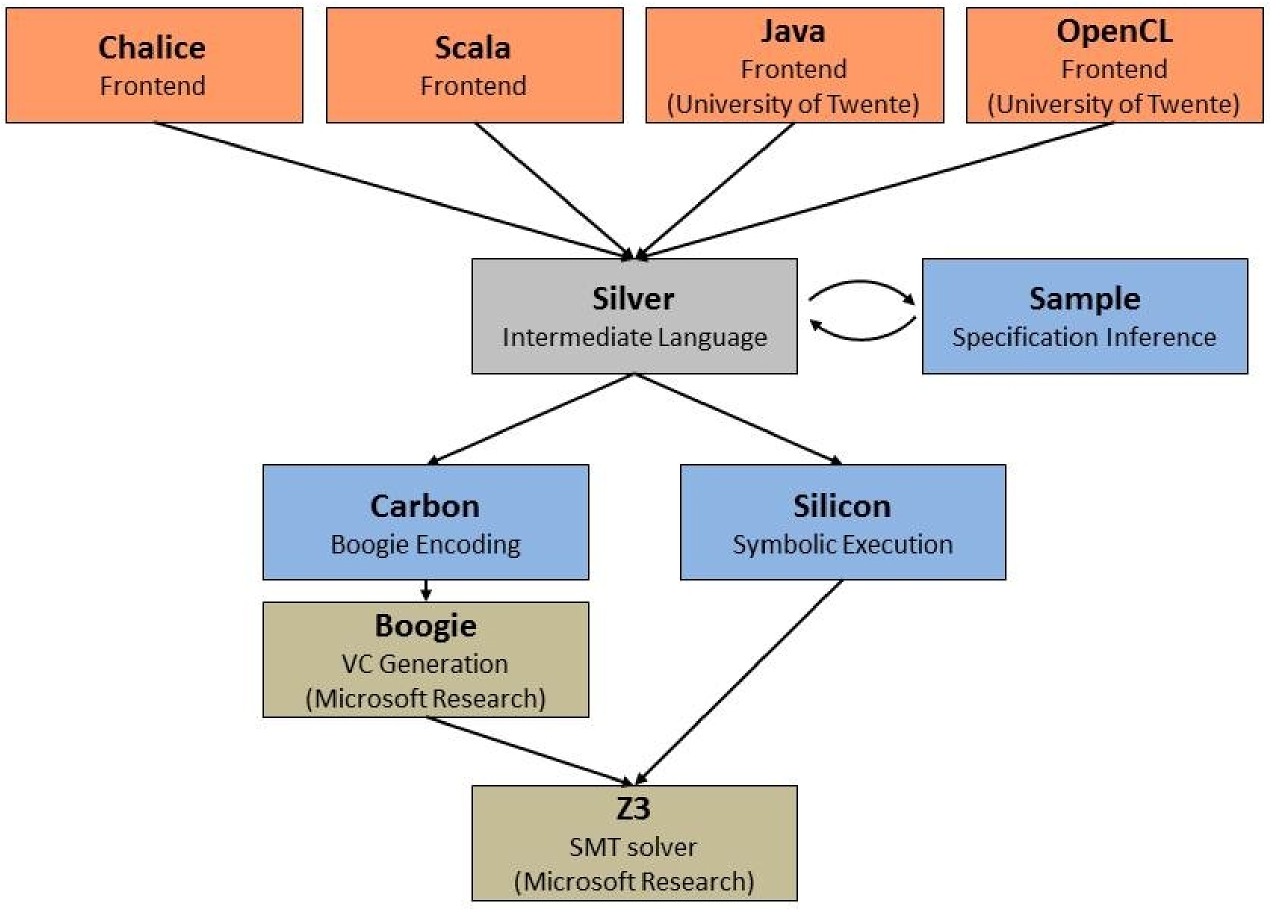


Figure 2.1: The Viper Framework. Note that Silver is the old name of the Viper language.

In the following, we will shortly describe in some more detail the most relevant Viper features for our work.

2.2.1 Permissions

A central feature of the Viper language is a built-in permission system that supports defining fine-grained access permissions for heap locations. To represent permissions, Viper has a special type `Perm`. We can use `write` to denote full permission which is needed for a field write, `none` to denote no permission and a fraction, e.g. `1 / 2`, to denote read permissions needed for a field read. To declare permissions Viper provides the special accessibility predicate `acc` which takes as arguments a location and a permission amount. It can occur in preconditions, invariants and postconditions to request or return permissions. A Viper program which performs a simple field write can be seen in Listing 2.1. It requires `write` access to `r.f` in the precondition and returns the same permission in the postcondition.

```
method assignField(r: Ref)
requires acc(r.f, write)
ensures acc(r.f, write)
{
  r.f := r
}
```

Listing 2.1: A simple field assignment.

Moreover, Viper also allows adding and removing permission from the current state through the keywords **`inhale`** and **`exhale`**. The statement **`inhale`** `A` assumes all pure assertions in `A` and adds all permissions in `A` defined through the accessibility predicate to the current state. The statement **`exhale`** `A` is the dual of the **`inhale`**. It asserts all pure assertions in `A` and removes all permissions in `A` from the current state. Listing 2.2 illustrates the use of inhales and exhales as well as fractional permissions. Before executing line 4, we start out with no permission to `r.f`. Then, we inhale the fractional permission `3 / 4` to location `r.f` which allows us to perform the field read on line 5. On line 6, the permission amount `1 / 2` is exhaled to the same location. Finally, we end up with permission **`acc`**`(r.f, 1 / 4)` after having executed line 6 which is exactly the permission we return in the postcondition defined on line 2.

```
1 method extractField(r: Ref) returns (res: Ref)
2 ensures acc(r.f, 1 / 4)
3 {
4   inhale acc(r.f, 3 / 4)
5   res := r.f
6   exhale acc(r.f, 1 / 2)
7 }
```

Listing 2.2: Fractional permissions and the use of inhale and exhale.

If a program contains field accesses or exhales for which not enough permissions are specified in preconditions or invariants, the verification of the program will fail.

2.2.2 Encoding of Arrays

Viper does not natively support the notion of an array. However, it supports custom data types through the declaration of *domains*. A domain may consist of uninterpreted functions and axioms. We can encode arrays in Viper through the definition of a custom domain. A standard array domain definition can be seen in Listing 2.3.
An array access in a higher-level programming language, e.g. `arr[i]` in Java, will be represented as `loc(arr, i).val` in Viper. The `loc` function represents the array slot corresponding to a given array and index and the `val` field represents the value stored in this array slot. The axiom `lengthNonneg` expresses that the length of every array is non-negative. Finally, the `allDiff` axiom expresses that the `loc` function is injective, i.e., that two different indexes of any array will always identify a different heap location.

```
field val: Int

domain Array {
  function loc(a: Array, i: Int): Ref
  function length(a: Array): Int
  function rToA(ref: Ref): Array
  function rToI(ref: Ref): Int

  axiom allDiff {
    forall a: Array, i: Int :: {loc(a, i)} rToA(loc(a, i)) == a && rToI(
      loc(a, i)) == i
  }
  axiom lengthNonneg {
    forall a:Array :: length(a) >= 0
  }
}
```

Listing 2.3: Encoding of Arrays in Viper.

2.2.3 Notation

In Viper it is syntactically incorrect to write `0` for no permission or `1` for write permission. Instead, we have to write `none` and `write`, respectively. Nevertheless, throughout this report, we will use `0` and `1` likewise for `none` and `write` for the sake of simplicity.
We will often omit field declarations in examples of Viper programs where the existence of that field is obvious from the code.
In addition, we will omit the encoding of arrays completely and assume that it is present in every program in the form described in Section 2.2.2. However, note that our algorithm is not restricted to work on this exact definition of arrays. We could, for example, as well handle arrays where `val` has a different type than `Int`.

2.3 Abstract Interpretation

Abstract interpretation [8] is a framework for static program analysis introduced by Patrick Cousot and Radhia Cousot. It is used to over-approximate the behavior (or *semantics*) of a program in a sound way. Thus, abstract interpretation allows to gain information about the semantics of a program without having to perform concrete executions of the program. We will now give a short overview over the most important concepts of abstract interpretation.
From a high level perspective, abstract interpretation provides the ability to reason about properties of a program at different abstraction levels. The most precise semantics are the *trace* semantics which describe all possible executions of a program exactly, usually representing program states by the program counter and a representation of the whole memory, i.e., variables and the heap. However, in general, these semantics are not decidable. Fortunately, interesting semantics do not have to be as precise and irrelevant information for the program property of interest can be abstracted away. For instance, one might only be interested in the values

integer-typed variables can have throughout the program. In that case, an abstract semantics could be derived from the concrete semantics by discarding all but integer-typed variables.
In abstract interpretation, the representation of a program state in the abstract semantics is an abstract state. The domain of concrete as well as abstract states has to form a lattice. A lattice is a partially ordered set (a so called *poset*) in which every two elements have a unique least upper bound and a unique greatest lower bound. Concrete states and abstract states are strongly connected by mathematical functions. A function that maps a concrete state to an abstract state is called an abstraction function, a function that maps an abstract state to a concrete state is called a concretization function. In particular, abstract states always represent an over-approximation of their corresponding concrete states. In other words, abstract states represent a superset of the program behaviors represented by the corresponding concrete states. As every element in the complete lattice has a unique least upper bound and a unique greatest lower bound, there are two unique elements that are larger or smaller, respectively, than every other element in the lattice. These two elements are called *top* and *bottom*, respectively.
A computation by abstract interpretation is performed in the abstract domain and starts with the bottom element. The bottom element represents the element denoting no program executions, the top element represents all possible program executions. The goal now is to reach a fixed point by repeatedly applying a monotone function f to the abstract state. f is defined in terms of *abstract transformers* which describes the effect of each statement type encountered in the program on abstract states. With every iteration before a fixed point is reached, the computation captures increasingly many program executions - possibly also some which cannot actually happen in practice. This leads to imprecision. Of bigger importance, however, is that the computation captures at least all possible executions which makes it a sound over-approximation. Reaching a fixed point is the main difficulty in abstract interpretation and depending on the lattice, such a fixed point computation can take a very long time or not terminate at all. In that case, a post-fixed point can be reached efficiently by applying a *widening* operator. However, widening potentially loses precision as often there would exist a more precise fixed point.

2.3.1 Sample

Sample [4] stands for "Static Analyzer of Multiple Programming LanguagEs" and is a static verification tool that is based on abstract interpretation. It is part of the Viper framework (see Figure 2.1). There exist parsers that convert between programs written in Viper language and their respective Sample representation (and vice versa). It can be used to strengthen existing specifications for Viper programs or to infer new ones. There already exist implementations of different abstract domains such as the octagon [15] or the polyhedra [9] abstract domains for integers. The implementation of our algorithm mainly takes place in Sample.

3 Inferring quantified permissions

In this chapter we present our algorithm to infer quantified permissions for Viper programs that allow the program to verify successfully. We first explain in detail how we collect and track required permissions by weakest precondition rules through straight line code and how we deal with changing expressions inside loops. Then, we present how we handle integer-dependent field accesses in which case a quantifier elimination algorithm is used to generate a precise expression that guarantees the required permissions. Finally, we will discuss further aspects of the approach such as what simplifications we apply on the inferred specifications to make them more readable or how we could extend the existing approach to also infer postconditions.

3.1 Overview

Our analysis works backwards, starting at the bottom of the method with zero permission for every field to any heap location. The main goal of the inference is that at each relevant point in the program, the tracked permissions represent *at least* the required permissions needed at that point in order for Viper to successfully verify the rest of the program. 'Relevant points' in our context refer to loop heads, where we will need to generate invariants, and to method entries, where we will need to generate preconditions. For this, the backward propagation mainly has to achieve two things. First, we have to add new permissions to our representation of permission expressions as we encounter new field accesses for which we have not collected permissions yet. Second, we also need to reflect changes to variables and the heap appropriately in our abstract state. Special attention needs to be given to loops as they can be executed an unknown number of times. Therefore, every variable or field assignment inside the loop body can be executed an unknown number of times. To account for that, we will have to use a special 'forget' operator similar to the one used in [24] which, broadly speaking, lifts our collected permissions to not only specify permissions for single heap locations but for a set of locations.

3.2 Collecting Permissions by Weakest Precondition Rules

We now first describe how we collect and propagate permissions through straight line code, i.e., in code that does not contain loops. For every statement type, we will provide a Hoare-style backward inference rule. A Hoare triple in our case has the form $\{\mathbf{P}\}\ \mathsf{s}\ \{\mathbf{Q}\}$ where $\mathbf{Q}$ denotes the collected permissions before traversing statement s and $\mathbf{P}$ denotes the collected permissions we get from propagating $\mathbf{Q}$ backward through s. In the following section, we will briefly describe the syntax of our abstract state describing collected permissions which resembles a tree structure.

3.2.1 Syntax of Permission Trees

Permission trees are our internal representation for the collected permissions. A leaf of the tree represents a concrete permission amount to a certain heap location and is represented as a triple (r, f, p) where r denotes the receiver expression of the field access, f the accessed field and p the actual permission amount. An inner node in the tree is built up according to the syntax in Figure 3.1. All tree operators are left-associative.

Tree ::= Tree $\bowtie_{\text{max}}$ Tree | Cond ? Tree : Tree | Tree + Tree | Tree − Tree | bound(Tree) | Leaf

Leaf ::= (Rec, Field, Perm)

Figure 3.1: The grammar of permission trees where Cond denotes a boolean condition, Rec denotes a receiver expression, Field denotes a field and Perm denotes a permission expression.

Permission trees are to be understood as point-wise declarations of permission amounts. We can imagine permission trees to be implicitly quantified over the domain of references. Therefore, a leaf (r, f, p) can be informally understood as `forall r̲ : Ref :: acc(r̲.f, r̲ == r ? p : none)`. In other words, the leaf (r, f, p) yields permission amount p for heap location `r.f` and no permission for every other heap location. We will talk more about how we actually generate permissions from the inferred permission trees in Section 3.6. The intuitive meanings of all the nodes in Figure 3.1 is as follows:

$t_1 \bowtie_{\text{max}} t_2$: The point-wise maximum of the permissions in t_1 and t_2.

b ? t_1 : t_2: The permissions in t_1 if b holds, the permissions in t_2 if b does not hold.

$t_1 + t_2$: The point-wise addition of the permissions in t_1 and t_2.

$t_1 - t_2$: The point-wise subtraction of the permissions in t_1 and t_2.

bound(t): The point-wise maximum between the permission in t and no permission (`none`). In other words, bound(t) denotes only non-negative permissions.

In the following, we will briefly explain the intuitive meaning of each tree construct. After that, we will list the Hoare-style inference rules of our backward analysis which will make use of those tree operations.

3.2.2 Permission Extraction

The function 'getAcc' detects all field accesses in an expression and collects all the necessary permissions. We will need this function in order to extract permissions e.g. for field accesses in nested receivers or inside conditions. This function is recursively defined as follows:

$$\text{getAcc(e)} = \left\{ \begin{array}{ll} \text{(e', f, read)} \bowtie_{\text{max}} \text{getAcc(e')}, & \text{if e matches e'.f} \\ \text{getAcc(e}_1\text{)} \bowtie_{\text{max}} \text{getAcc(e}_2\text{)}, & \text{if e matches e}_1 \bullet \text{e}_2 \\ \text{getAcc(e')}, & \text{if e matches -e'} \\ \varnothing, & \text{else} \end{array} \right\}$$

where $\bullet$ denotes a binary operator out of (+,−,⋆,\,%,<,<=,>,>=,==,!=).

3.2.3 Variable Assignments

For a variable assignment `v := e` we do not have to request any **`write`** permission because variables are not part of the heap. However, we have to ensure that we have `read` permission for every field access that occurs in the right expression `e`. If we have already collected enough permission for a field access, we do not want to add another read permission. This is why we do not add but apply the maximum operator to the new permissions and the previously collected permissions. At the same time, we also have to take care of the change to variable `v`. To represent the assignment in our state, we replace every occurrence of `v` with `e` in the permissions collected so far.

$$[\text{VA}]\ \frac{}{\{\mathsf{P}[\mathsf{v} \mapsto \mathsf{e}] \bowtie_{\max} \mathsf{getAcc}(\mathsf{e})\}\ \mathtt{v\ :=\ e}\ \{\mathsf{P}\}}$$

To illustrate this rule, consider Listing 3.1. The tracked permissions for the field `next` are indicated in comments after the statements. Note that we omit the field `next` in the comments for the sake of simplicity. On line 5, there is an assignment to `x.next`. Therefore, we add a permission leaf `(x, next, `**`write`**`)` to the (at that point still empty) permission tree. The variable assignment on line 4 now has two effects. First, the left hand side of the variable assignment `x` gets replaced in the existing permission tree by the right hand side of the assignment, `x.next`. Second, the new permission tree is calculated by the rule VA which tells us to add permission by means of the maximum operator for all the field accesses of the right hand side which in this case is just the field access `x.next`. Finally, the permission tree is propagated over another variable assignment on line 3. We again replace all occurrences of the left hand side with the right hand side of the assignment, meaning that every `x` is replaced by `head`. The final permission tree can be seen in the comment on line 3.

```
1 method varAssignment(head: Ref)
2 {
3   var x: Ref := head   // (head.next, write) ⋈max (head, read)
4   x := x.next          // (x.next,    write) ⋈max (x,    read)
5   x.next := x          // (x,         write)
6 }
```

Listing 3.1: Simple variable assignment.

3.2.4 Field Assignments

For a field assignment `l.f := r` we need to have **`write`** permission to `l.f`. Additionally we have to make sure that we have `read` permission for every field access in `r`, similarly to the variable assignment rule, as well as `read` permission to every field access appearing in `l`. Again we do not need to add any permissions if there is already enough permissions available in the tracked expressions because a field assignment does not add or remove any permission amount from the program state. Thus, we again use the maximum operator to add the new permissions. Furthermore, we also have to account for possible changes in our tracked permissions so far due to the assignment of `r` to `l.f`. In addition to simply replacing every occurrence of `l.f` with `r`, we also have to take care of the fact that the assignment can potentially affect every expression of the form `e.f` where `e` can be an arbitrary expression. The reason for this is that `e` and `l` might alias. In the case they actually alias, the assignment `l.f := r` effectively also changes `e.f` to `r`. We account for this by introducing a helper function fa which returns the appropriate replacement for every `e.f` depending on whether `e` syntactically matches `l` or not (note that here, `e` has to be understood as a placeholder for an arbitrary expression).

$$[\text{FA}]\ \frac{}{\{\mathsf{P}[\mathsf{e.f} \mapsto \mathbf{fa}(\mathsf{f, e, l, r})] \bowtie_{\max} (\mathsf{l, f,}\ \mathtt{write}) \bowtie_{\max} \mathsf{getAcc(l)} \bowtie_{\max} \mathsf{getAcc(r)}\}\ \mathtt{l.f\ :=\ r}\ \{\mathsf{P}\}}$$

where **fa** is defined as follows:

$$\mathbf{fa}(\mathrm{f, e, l, r}) = \begin{cases} \mathrm{r}, & \text{if e \textit{syntactically} matches l} \\ (\mathrm{e} == \mathrm{l}\ ?\ \mathrm{r} : \mathrm{e.f}) & \text{else} \end{cases}$$

Again, we present a simple example in Listing 3.2 to visualize this rule. On line 5, there is a field assignment to `y.f.f`. Note that in order to execute the field write to `y.f.f`, in accordance to the field assignment rule, we also need read access corresponding to the inner field access `y.f`. The two leaves, `(y.f, write)` and `(y, read)` are connected by the maximum operator. When propagating over the field assignment on line 3, the access path `y.f` gets replaced by the conditional `(y == x ? y : y.f)` which accounts for the possible aliasing of `x` and `y`. Furthermore, `write` access is added for the receiver `x` of the field assignment.

```
1 method fieldAssignment(head: Ref)
2 {
3   x.f := y    // ((y == x ? y : y.f), write) ⋈max (y, read) ⋈max
4               // (x, write)
5   y.f.f := x  // (y.f, write) ⋈max (y, read)
6 }
```

Listing 3.2: Simple field assignment.

3.2.5 Exhales

With an `exhale`, permissions can be removed from the program state. Consequently, we have to *add* the exhaled permission amount to the tracked permissions. We represent this by applying the + operator to the permission leaf corresponding to the exhale statement. Note that we also have to traverse the receiver r for an exhale to collect possible additional field accesses.

$$[\text{Exhale}]\ \frac{}{\{\mathsf{P} + (\mathsf{r, f, p}) \bowtie_{\max} \mathsf{getAcc(r)}\}\ \mathtt{exhale\ acc(r.f,\ p)}\ \{\mathsf{P}\}}$$

In the example in Listing 3.3, an exhale of `1 / 2` is followed by a read to the same location. That means, that after the exhale, there still needs to be a permission amount left that is strictly greater than `none`. In our backwards analysis, we first record the read permission for the field read on line 4. On line 3, we add the exhaled permission. Note that since we exhale to the exact same location, we can simplify the expression by merging `(x, read) + (x, 1 / 2)` to `(x, read + 1 / 2)`

```
method writeExhale(x: Ref) returns (res: Ref)
{
  exhale acc(x.next, 1 / 2)  // (x, read + 1 / 2)
  res := x.next              // (x, read)
}
```

Listing 3.3: An exhale followed by a field read to the same location.

3.2.6 Inhales

An **inhale** adds permissions to the current program state. For our analysis, this means that we can *subtract* the respective permission amount from our tracked expressions. However, we have to take additional care that we do not end up with a negative permission amount which is why a simple subtraction would be unsound. Remember that we want to track *at least* the permissions that are required for the program to successfully verify. The consequence of allowing permission amounts that are less than **none** would be that an **inhale** that follows an **exhale** could cancel out the permission potentially needed for the **exhale**. We therefore apply the 'bound' operator to the subtraction of the inhaled permission amount from the collected permission. This operator bounds negative permission values to **none**.
To illustrate the problem, consider the simple example Listing 3.4 in which we first exhale **write** permission to index 0 of `arr` and then **inhale** the same amount of permission for the same location. Remember that our analysis works backwards. We would therefore add (`loc(arr, 0).val`, -**write**) to the collected permissions on line 4 and end up with a temporary negative amount. On line 3, **write** permission would get added again and the final sum of permissions for the particular location `loc(arr, 0).val` would be **none**. Therefore, the program would fail to verify as we would not require **write** permission which would be needed for the first **exhale**.

```
1  method negativePerm(arr: Array)
2  {
3    exhale acc(loc(arr, 0).val, write) // (loc(arr, 0), 0)
4    inhale acc(loc(arr, 0).val, write) // (loc(arr, 0), -write)
5                                       // (loc(arr, 0), 0)
6  }
```

Listing 3.4: An example which uses the unsound inhale rule which does not apply the boundary function. Although the overall sum of permissions in this program is **none**, we need to require **write** permission in the precondition. This example illustrates that we always have to bound collected negative permission amounts by **none**.

Our rule therefore becomes:

$$[\text{Inhale}]\ \frac{}{\{\mathsf{bound}(\mathsf{getAcc}(\mathsf{r}) \bowtie_{\max} \mathbf{P} - (\mathsf{r}, \mathsf{f}, \mathsf{p}))\}\ \texttt{inhale acc(r.f, p)}\ \{\mathbf{P}\}}$$

3.2.7 Conditionals

For a conditional, we first traverse each branch separately, collecting permissions using the rules described above. We try to be as precise as possible in our analysis and hence merge the two branches together using a ternary operator on the head of the condition. Additionally, we also have to traverse the condition itself and add read permissions to every field access we encounter therein.

$$[\mathrm{If}]\ \frac{\{\mathbf{P}_1\}\ \mathtt{s1}\ \{\mathbf{Q}\} \qquad \{\mathbf{P}_2\}\ \mathtt{s2}\ \{\mathbf{Q}\}}{\{\mathsf{getAcc(b)} \bowtie_{\max} (\mathsf{b}\ ?\ \mathbf{P}_1 : \mathbf{P}_2)\}\ \mathtt{if\ (b)\ \{\ s1\ \}\ else\ \{\ s2\ \}}\ \{\mathbf{Q}\}}$$

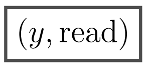
Let us consider the permission tree which is collected in Listing 3.5 for the field `next`. The permission trees for each point in the program are shown in Figure 3.2. At the bottom, on line 7, there is a field read to `y.next`. Thus we start out with the permission leaf (y, read). Next, the permission tree is propagated through both of the branches of the conditional on lines 4 to 6. The false-branch does not contain any statement, hence the permission tree does not get modified in that branch. In the true-branch, we first process the variable assignment, replacing (y, read) by $(x.next, \mathrm{read})$. Moreover, we add the leaf (x, read) to the permission tree by applying the maximum operator to the leaf and the already present leaf to account for the field read on the right side of the assignment. We end up with the permission tree $(x, \mathrm{read}) \bowtie_{\max} (x.next, \mathrm{read})$ for the true branch. Joining with the permission tree from the false branch using the conditional operator and after propagating through the assignment `y := x` on line 3 we get the permission tree b ? $(x, \mathrm{read}) \bowtie_{\max} (x.next, \mathrm{read})$: (x, read).

```
1  method simpleConditional(b: Bool, x: Ref) returns (res: Ref)
2  {
3    var y: Ref := x // see Figure 3.2d
4    if (b) {        // see Figure 3.2c
5      y := x.next   // (x.next, read) ⋈max (x, read)
6    }
7    res := y.next   // (y, read)
8  }
```

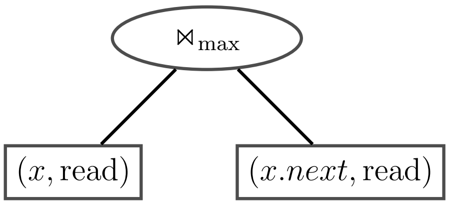
Listing 3.5: The permission tree on method entry for this method will contain the branch condition `b`.

3.2.8 Loops

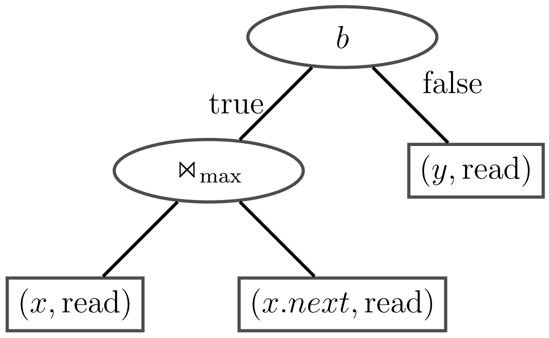
Not surprisingly, loops are the most difficult statement to handle. Consider the simple list traversal in Listing 3.6. Clearly, depending on how many times the loop iterates, we need permission to more or less heap locations. If `head` is **`null`**, the loop is never executed and thus no field access at all happens. If the loop is executed once, read permission to `head.next` is required. If the loop is executed twice, read permissions to `head.next` as well as `head.next.next` is required, and so on. The important observation is that permission to the field access on line 6 can no longer be thought of as a single comparison operation ($\underline{\mathtt{r}}$ `== head ? x :` **`none`**). Because the number of loop iterations in a concrete execution of the program depends on the shape of the heap, we have to find a representation that can express permission to the



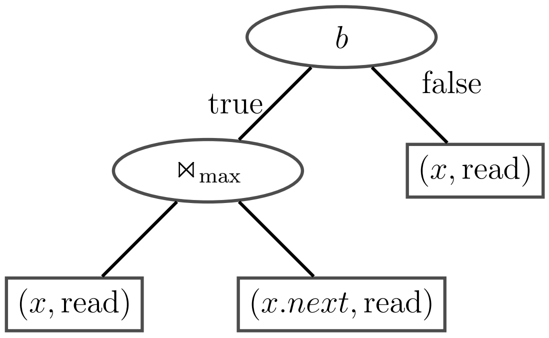
(a) Permission tree corresponding to line 7 and the false branch of the conditional.



(b) Permission tree at the top of the true branch (between lines 4 and 5).



(c) Permission tree after joining of true and false branch between lines 3 and 4.



(d) Permission tree at method entry (between lines 2 and 3).

Figure 3.2: The permission tree for the field `next` at the indicated locations in program Listing 3.5. Ellipses correspond to inner nodes, rectangles to permission leafs.

different values `x` can take on line 6 in an arbitrary number of iterations. That is where sets come into play. Instead of a comparison with the expression `x` we augment the receiver `x` to a set description of receivers representing all possible concrete values `x` can take at any loop iteration. When generating the contract, we will instead of generating the loop invariant (`r == x ? read : none`) generate an expression (`r in set_x ? read : none`).

```
1  method listTraversal(head: Ref)
2  {
3    var x: Ref := head
4    while (x != null)
5    {
6      x := x.next       // (x, read)
7    }
8  }
```

Listing 3.6: Simple list traversal.

We will get the specification of the set by abstract interpretation (see Section 3.3). An additional helper variable `set_x` will be introduced. For the above example, the specification of `set_x` would then look as follows:

```
head != null ==> head in set_x
forall r: Ref :: r in set_x && r.next != null ==> r.next in set_x
```

Also, if the condition of a conditional inside a loop contains at least one changing variable, we have to assume that both branches are executed because we will, in general, have no way to express the condition outside of the loop. In that case, a conditional (b ? t_1 : t_2) is approximated to $t_1 \bowtie_{\max} t_2$.
We represent this additional step that has to be taken in loops by applying '**forget**' in the backward rule for loops.

$$[\text{While}]\ \frac{\{\mathbf{P}\}\ \mathsf{s}\ \{\mathbf{Q}\}}{\{\mathbf{Q} \bowtie_{\max} \mathbf{forget}(\mathsf{getAcc(b)} \bowtie_{\max} \mathbf{P})\}\ \texttt{while (b) \{ s \}}\ \{\mathbf{Q}\}}$$

3.3 Over-approximating Receivers by Abstract Interpretation

In the last section we collected permissions for every field access at any point in the program. However, receivers of field accesses that happen inside or after a loop can take different values depending on how many times loops are executed. We cannot determine purely syntactically all the values those receivers will take during an execution of the program. Therefore we employ abstract interpretation [8] to over-approximate all the possible reference values each receiver can take. For this we introduce, for every field access `r.f` occurring in the method, a set which represents all the possible expressions `r` can represent at a specific point. Our abstract state then consists of a mapping from every field access to a collection of concrete expressions that each denote a possible receiver for that specific field access. These concrete expressions get modified as we propagate them backwards through the program. In general, the sets can grow indefinitely when we propagate them through loops. Therefore we will have to employ widening at some point to avoid infinite iteration of the algorithm. This abstract interpretation framework will be explained in more detail in the following.

3.3.1 Abstract State

An abstract state consists of one receiver set per field access in the program. We can uniquely identify a receiver set by a pair `(pp, rec)` where `pp` denotes the unique program point (in Sample, a program point of a statement is uniquely determined by the line number and column number of the statement) of the field access and `rec` denotes the receiver expression. A receiver set is a set of expressions that denote concrete receivers. The bottom element is defined as the state where all the receiver sets are empty, the top state is defined as the state where all receiver sets contain all possible heap locations (this is just a theoretical element, in practice we will never use this element). As usual in abstract interpretation over set domains, the join operator for the abstract state is set union. Our analysis will work backwards and we start with the bottom state, i.e., every receiver set being the empty set. Our abstract transformers are defined as follows:

Field Assignment
When we propagate the state over a field assignment `l.f := r` at program point `pp`, we add the element `l` to the receiver set corresponding to `(pp, l)`. Furthermore, for every receiver set, we replace every `l.f` occurring in any receiver expression with `r`. We also have to replace, in every receiver set, field accesses `e.f` where `e` is an arbitrary expression which is not syntactically equal to `l`. Every expression of this form is replaced with `(e == l ? r : e.f)`, similarly to the assignment rule in Section 3.2.4.

Variable Assignment

When we propagate the state over a variable assignment `v := e` at program point `pp`, we replace `v` in all expressions in any of the receiver sets with `e`.

Field Read

When we propagate the state over a field read `rec.f` at program point `pp`, we add the element `rec` to the receiver set corresponding to `(pp, rec)`.

Joining Branches

Our join or least upper bound operator is defined as the element-wise set union, as is usual in set domains. I.e. if we join two states s_1 and $s_1 \sqcup s_2$ is defined to be the state that contains for every receiver set defined by `(pp, rec)` the union of the corresponding receiver sets in s_1 and s_2.

The abstract transformers are iterated by Sample on the program until a fixed point is reached or the widening limit is reached. The widening limit is an adjustable parameter in Sample that determines, how many times a statement is allowed to be executed before the widening operator is applied.

3.3.2 Widening

Widening for our states means that we have to lift some or even all of the finite receiver sets to a potentially unbounded set description. For the example in Listing 3.7 the set of possible receivers for the field access `x.next` on line 6 is `head` and for all elements `r` that are not **`null`**, `r.next` is also in the set. Before widening the set of concrete expressions for `x` on line 6 will therefore contain longer and longer chains of `next` accesses to `x`. This is visualized in Table 3.1. After the first iteration the set would consist of `x`. After the second iteration it would contain `x` and `x.next` and so on. On the widening step we switch from a set of concrete expressions to a regular expressions like representation, in this case `x(.next)`*. After propagating the receiver set through the assignment on line 3, every `x` in the receiver set will be replaced by `head`.

```
1  method listTraversal(head: Ref)
2  {
3    var x: Ref := head
4    while (x != null)
5    {
6      x := x.next
7    }
8  }
```

Listing 3.7: A simple list traversal where the set of possible receivers on line 6 will all be repeated `next` accesses to `head`.

On the widening step we also over-approximate all potentially contained ternary conditionals in a changing receiver set. A ternary conditional `(x == r ? e : x.f)` will be over-approximated to the set {`e, x.f`}. This means that the condition `cond` is approximated away as if the

Iteration	1	2	3	widened
Receivers	{x}	{x, x.next}	{x, x.next, x.next.next}	x.(next)*

Table 3.1: The development of the receiver set corresponding to the field access on line 6 in Listing 3.7 at the loop head.

conditional would be non-deterministic. Note that it is possible that expressions contain nested conditionals, e.g. `(c1 ? e1 : (c2 ? e2 : x.f).f)`. In this case, the transformation step is applied recursively and we end up with the set {`e1, e2.f, x.f.f`}.
The widening operator we implemented in our analysis in Sample is imprecise. Due to lack of time, we only implemented a very basic widening operator which in general over-approximates the resulting sets heavily. However, in many interesting examples, the effect of this additional over-approximation keeps within reasonable limits.

3.4 Integrating Heap Analysis

Using information from a heap analysis we can eliminate some of the case distinctions introduced for field assignments on the fly. Before our actual algorithm we conduct an alias analysis developed by Jérôme Dohrau at the Chair of Programming Methodology [3]. The results of this alias analysis are two alias graphs at every point in the program: the 'may-alias' graph and the 'must-alias' graph. Both graphs are an approximation of the heap layout at the respective position. They consist of a store and a field map. The store is a mapping from variable names to the set of abstract heap nodes to which the variables point to. The field map maps pairs of a heap node and a field to a set of heap nodes the field of the heap node points to. A heap node is identified by an access path. An access path is a list of strings where the head of the list represents the base variable and each element of the tail represents a field. Alias graphs provide a function `pathsAlias` which takes two access paths and returns a boolean value indicating whether or not the two paths alias in the graph. In simplified terms this is the case if and only if the two heap nodes identified by the paths are the same. The meaning of the graphs is as follows. If the may-alias tells us that two given paths `p1` and `p2` do not alias, we can be sure that the two paths at that particular program point certainly do not alias. If they do alias in the may-alias graph, they might or might not alias. We can then go on and ask the same question to the must-alias graph. If `p1` and `p2` alias in the must-alias graph then we know for sure that `p1` and `p2` always point to the same heap location at that particular program point. This is illustrated in Table 3.2. Then, instead of directly replacing every `x.f` in our state by `(x == r ? e : x.f)` when propagating over an assignment `r.f := e`, we first check whether either `x` and `r` alias (in that case we can directly replace `x.f` by `e`) or if they must not alias (in that case, `x.f` is not affected by the assignment and stays the same). In the case where the alias analysis tells us that `x` and `r` maybe alias we still have to replace `x.f` by `(x == r ? e : x.f)`. We also use the case distinction in the case where the alias analysis fails. This is the case when a field access has a form which cannot be handled by the current implementation of the heap analysis, e.g. when a receiver contains a function.

	mustAlias(p1, p2) = **true**	mustAlias(p1, p2) = **false**
mayAlias(p1, p2) = **true**	p1 and p2 **alias**	p1 and p2 **maybe** alias
mayAlias(p1, p2) = **false**	not possible	p1 and p2 do **not alias**

Table 3.2: The various meanings of the alias test depending on what information we get from the may-alias and must-alias graphs. Note that it is not possible that two paths p1 and p2 alias in the must-alias graph but not in the may-alias graph.

3.5 Handling of Integer-dependent Accesses

As already mentioned in Section 3.3 we handle integer-dependent heap accesses separately. To get information about the set of possible values an integer-dependent receiver can take, we solely rely on a numerical analysis by abstract interpretation. Of particular interest in this context are array accesses, i.e., heap accesses that occur to the `val` field of arrays (cf. Section 2.2.2). We will therefore use array examples to illustrate the concepts in our numerical analysis. However, note that our algorithm is flexible and can handle other functions that depend on integers. The only restrictions we make are that every access to the same field happens through the same function and at most one integer-typed parameter is a changing variable (although we will talk about under which circumstances we can lift this restrictions in Section 3.9.3). That also means that we are still able to handle programs that contain both accesses to fields through reference variables as well as accesses to fields through functions as long as all the accesses to a particular field only happen through one of the two access types. Our description of possible receivers for the field in question will then purely be in terms of integer constraints over the arguments of the receiver. In the array case, instead of describing the set of references the expression `loc(arr, i)` can take as a receiver for the field `val`, we will talk about the possible range of integers `i` can take to produce different receivers for `val`. In other words, we will assume the fixed form `loc(arr, .)` of the receiver for `val`. Consider the example in Listing 3.8. It performs a simple traversal of an array and initializes each array location with its index.

```
1 method initializeArray(arr: Array)
2 {
3   var i: Int := 0
4   while (i < 10)
5   {
6     loc(arr, i).val := i // At this point, i ∈ [0, 9]
7     i := i + 1
8   }
9 }
```

Listing 3.8: A simple traversal of an array.

Note that the array access on line 6 happens to multiple locations. To be more specific, all the possible array slots we access correspond to indexes in the interval $[0, 9]$. We cannot define permissions in terms of `i` as it is a changing variable inside the loop. Therefore, we have to find

a way to describe a set of possible integer values `i` can take at that point without mentioning `i`. This step of over-approximating the set of integer values inside a receiver can be seen as the counterpart to the over-approximation of receiver sets.
In the rest of this section we will describe the numerical analysis we use and how we include its result into our analysis.

3.5.1 Numerical Abstract Interpretation

Before our actual algorithm, we run a numerical abstract interpretation using the polyhedra abstract domain [9]. Our algorithm is flexible to use any other integer domain instead of polyhedra, e.g. integer octagons [15] or intervals [7]. We chose the polyhedra domain because it is very precise and, in contrast to e.g. the interval domain, a relational domain that is able to infer linear relationships between variables. A polyhedron can be described by a conjunction of linear inequalities written as $\bigwedge_{i=1}^{n} \langle \vec{a}_i, \vec{x} \rangle \leq b_i$ where n is the number of inequalities, $\vec{a}_i$ is the ith row of $A \in \mathbb{Z}^{n \times m}$, the coefficient matrix, $\vec{x} \in \mathbb{Z}^m$ denotes the vector holding all integer-typed program variables and $\vec{b} \in \mathbb{Z}^n$ is a vector containing the limits of the inequalities. The inequalities can also be more compactly written as $Ax \leq b$.

```
1   method numericalExample(bound: Int)
2   {
3     var i: Int := 0
4     var j: Int := bound
5     while (i < bound)
6     {
7       i := i + 1
8       j := j - 2
9     }
10  }
```

Listing 3.9: A simple iteration over integer variables.

Consider the simple example in Listing 3.9. This program contains three integer variables: local variables `i`, `j` and the method parameter `bound`. After the two initializations on lines 3 and 4 of the program, the polyhedra would consist of the following inequalities (note that equalities $a = b$ are represented as $a \leq b \wedge -a \leq -b$ in the polyhedra domain): $i \leq 0 \wedge -i \leq 0 \wedge j \leq \mathit{bound} \wedge -j \leq 0$ or in matrix notation:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ \mathit{bound} \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

By manual inspection, one can find that $i \geq 0$, $i \leq$ and $2 * i + j == \mathit{bound}$ are invariants for the loop on line 5. This would be represented in the polyhedra domain as follows:

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & -1 \\ -2 & -1 & 1 \\ 2 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ bound \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

For our analysis it is important that we can retrieve those constraints from the analysis result in a form that we can easily handle. In Sample, there already exists an interface to the APRON library [13] which provides an implementation of the polyhedra domain among other numerical abstract domains. We can then get constraints corresponding to a certain variable as inequalities, exactly as represented above.

3.5.2 Injectivity Requirement for Receiver Functions

In order for our mapping from integers back to locations through the `loc` function to be sound, we need the receiver expression to be injective in the respective input arguments. To test whether this is the case, we perform an injectivity check on the fly during our analysis by using one or multiple additional calls to the verifier. In our concrete example for arrays, we want to know if the `loc(arr, i)` function is injective on the relevant set of integers for `i`. Consider again the example in Listing 3.8. On line 6, we learn from the polyhedra element at that point that $i \in [0, 9]$. Thus, we have to ensure that injectivity holds in this range, i.e.

$$\forall i_1, i_2 \in \mathbb{Z}.(i_1 \neq i_2 \land 0 \leq i_1 \land i_1 \leq 9 \land 0 \leq i_2 \land i_2 \leq 9) \implies \mathrm{loc}(arr, i_1) \neq \mathrm{loc}(arr, i_2) \quad (3.1)$$

We create a program consisting of all the domain information (abstract functions and axioms), functions and fields of the original program (cf. Section 2.2), but no methods. We add only one method that has an empty method body and no return value. The arguments of the method are the original arguments of the currently considered method plus additional arguments we need for the injectivity check. The precondition of the method is the left side of the injectivity implication which expresses that the corresponding variables are different from each other and they are in the relevant range. The postcondition is the right side of the injectivity implication. Then, if the verifier can successfully verify this program, we know that the injectivity constraint holds. The so generated injectivity test corresponding to Listing 3.8 can be seen in Listing 3.10. Note that for the injectivity test it is crucial that also the domain definition of *Array* and the `val` field are included. The axiom `axiom_all_diff` (cf. Section 2.2.2) is also the key to prove the injectivity test. Without it, the verification would fail.

```
method injectivityTest(a: Array, i1: Int, i2: Int)
requires i1 != i2 && i1 <= 9 && i1 >= 0 && i2 <= 9 && i2 >= 0
ensures loc(a, i1) != loc(a, i2)
{
}
```

Listing 3.10: The automatically generated injectivity test for Listing 3.8.

3.5.3 Forgetting Variables in Loops

Unfortunately, the numerical analysis only gives us an over-approximation of all the possible values that can occur inside a loop but it does not tell us in which order they occur during the loop execution. Consider the example in Listing 3.11. We need **write** permission to `loc(arr, 9).val` for the field write on line 7. Actually, it happens that we inhale **write** permission to this index on line 6, but only in a later iteration of the loop. An unsoundness in [24] is attributable to ignoring the fact that the order in which the changing values assume the values inferred by the numerical analysis is arbitrary. Thus, what we seek for is an expression that does not mention `i` or in general, that does not mention any changing variable, and that asserts at least the required permissions for any location in an arbitrary loop iteration.

```
1  method arrayExample(arr: Array)
2  {
3    var i: Int := 0
4    while (i < 10)
5    {
6      inhale acc(loc(arr, i).val)
7      loc(arr,9).val := i
8      i := i + 1
9    }
10 }
```

Listing 3.11: A simple array example with one inhale and a field write to index 9.

Intuitively speaking, the expression we seek is the maximum of the required permission amounts in any loop iteration. To formalize this, let $v_1, ..., v_n$ denote all the changing variables inside a loop, let $\mathrm{INV}(v_1, ..., v_n)$ be the loop invariant returned by the numerical analysis and let $\mathrm{collected}(q, t, v_1, ..., v_n)$ be the expression which returns permissions according to the collected permission tree t inside the loop body. The expression we aim for is

$$\forall q \in \mathbb{Z}.\underline{\mathrm{p}}(q) = \max_{v_1,...,v_n} \{\mathrm{collected}(q, t, v_1, ..., v_n) \mid \mathrm{INV}(v_1, ..., v_n)\} \tag{3.2}$$

where $\underline{\mathrm{p}}$ is a function $\underline{\mathrm{p}}$: `Int`$\rightarrow$`Perm` that returns for every array index q the required permissions for the `val` field at that location. We now consider the permission expression for a fixed index q and reformulate the maximum over the changing variables $\max_{v_1,...,v_n}$. This way, the expression can be written as follows:

$$\begin{aligned}&(\exists v_1, ..., v_n.\mathrm{INV}(v_1, ..., v_n) \wedge \underline{\mathrm{p}}(q) = \mathrm{collected}(q, t, v_1, ..., v_n)) \wedge \\ &(\forall v_1, ..., v_n.\mathrm{INV}(v_1, ..., v_n) \implies \underline{\mathrm{p}}(q) \geq \mathrm{collected}(q, t, v_1, ..., v_n))\end{aligned} \tag{3.3}$$

As already mentioned, we ultimately aim for an expression that does not mention changing variables anymore and that describes permissions purely in terms of q. Thus, we need a way to eliminate the quantification over the changing variables. This is exactly what quantifier elimination does. Quantifier elimination is a computational simplification that produces, for a given formula containing one or more quantifiers, an equivalent formula which is quantifier-free. For example, an equivalent formula to $\exists y.(x = y \wedge y \geq 0)$ without quantifiers would be $x \geq 0$.

In the next section, we will describe how we rewrite permission trees into a form that can be handled by quantifier elimination. The section thereafter will describe the algorithm we use to perform quantifier elimination in detail.

3.5.4 Rewriting Permission Trees

In the general case where the collected permission tree has a complex shape containing maximums ($\bowtie_{max}$) or conditionals, we first have to rewrite the tree to contain only formulas that the quantifier elimination can handle. The basic quantifier elimination we use can handle formulas written in Presburger arithmetic [21]. A formula in Presburger arithmetic can contain algebraic expressions with integers, variables and addition, binary comparison operators such as $<$, $=$ or $\geq$ and the logical connectives $\wedge$, $\vee$, $\neg$ with their usual meaning. Multiplication of integer variables by a constant is also possible as the multiplication can be rewritten to an addition: $2a = a + a$. The formula can be existentially quantified over one or multiple variables. Universal quantification is then trivially supported as well since $\forall x.p(x)$ can be rewritten to $\neg\exists x.\neg p(x)$.
We now first give a recursive description how we rewrite each permission tree construct to an intermediate representation in terms of the rewrite_q function. The q in rewrite_q highlights the use of q as index of interest in the rewrite process. Our intermediate representation is a set of pairs (c, p) where c is a boolean expression and p is a permission amount.
After that, we will explain how to turn the set of pairs into an assertion about $\underline{\text{p}}(q)$ which is equivalent to Equation (3.3).

Leaves
A leaf is turned into a set of two pairs. The first pair accounts for the case where the integer parameter e is equal to the quantified variable q in which case the corresponding permission is the collected permission p. The second pair denotes the case where $q \neq e$ in which case this leaf does not contribute any permission.

$$\text{rewrite}_q((\text{loc}(arr, e), p)) = \{(q = e, p), (q \neq e, 0)\} \tag{3.4}$$

Maximum
Taking the maximum of two permission trees adds for every possible combination of pairs $(c_1, p_1) \in \text{rewrite}_q(t_1)$ and $(c_2, p_2) \in \text{rewrite}_q(t_2)$ two pairs to the resulting set. One for the case where p_1 is greater than p_2 and one for the other case.

$$\begin{aligned}\text{rewrite}_q(t_1 \bowtie_{\text{max}} t_2) =& \{(c_1 \wedge c_2 \wedge p_1 \geq p_2, p_1) \mid (c_1, p_1) \in \text{rewrite}_q(t_1) \wedge (c_2, p_2) \in \text{rewrite}_q(t_2)\} \cup \\ & \{(c_1 \wedge c_2 \wedge p_2 \geq p_1, p_2) \mid (c_1, p_1) \in \text{rewrite}_q(t_1) \wedge (c_2, p_2) \in \text{rewrite}_q(t_2)\}\end{aligned} \tag{3.5}$$

Addition
Rewriting a permission addition consists of linking each condition from a pair of the first tree with each condition from a pair of the second tree with a conjunction and summing up the corresponding two permission amounts.

$$\text{rewrite}_q(t_1 + t_2) = \{(c_1 \wedge c_2, p_1 + p_2) \mid (c_1, p_1) \in \text{rewrite}_q(t_1) \wedge (c_2, p_2) \in \text{rewrite}_q(t_2)\} \tag{3.6}$$

Conditionals
Rewriting conditionals is rather straightforward. For every pair $(c_1, p_1) \in \text{rewrite}_q(t_1)$ we add the same pair with the additional condition b to the resulting set, for every pair $(c_2, p_2) \in \text{rewrite}_q(t_2)$ we add the same pair with ¬b to the resulting set.

$$\begin{aligned} \text{rewrite}_q(\text{b} \ ? \ t_1 : t_2) = &\{(c_1 \land \text{b}, p_1) \quad | \ (c_1, p_1) \in \text{rewrite}_q(t_1)\} \cup \\ &\{(c_2 \land \neg\text{b}, p_2) \ | \ (c_2, p_2) \in \text{rewrite}_q(t_2)\} \end{aligned} \tag{3.7}$$

Zero-bounded permission tree
Rewriting zero-bounded permission trees is similar to rewriting maximum permission trees. The left subset of the union accounts for all cases where a permission in $\text{rewrite}_q(t)$ is non-negative and the right subset clamps all negative permission amounts to 0.

$$\begin{aligned} \text{rewrite}_q(\mathsf{bound}(t)) = &\{(c \land p \geq 0, p) \ | \ (c, p) \in \text{rewrite}_q(t)\} \cup \\ &\{(c \land p < 0, 0) \ | \ (c, p) \in \text{rewrite}_q(t)\} \end{aligned} \tag{3.8}$$

We can now write the assertion that constrains $\underline{\text{p}}(q)$ to require enough permission according to Equation (3.2) in the same form as Equation (3.3) as follows:

$$\begin{aligned} &\exists v_1, ..., v_n. \left(\text{INV}(v_1, ..., v_n) \land \bigwedge_{(c,p) \in \text{rewrite}_q(t)} \big(c \implies \underline{\text{p}}(q) = p\big) \right) \land \\ &\forall v_1, ..., v_n. \left(\text{INV}(v_1, ..., v_n) \implies \bigwedge_{(c,p) \in \text{rewrite}_q(t)} \big(c \implies \underline{\text{p}}(q) \geq p\big) \right) \end{aligned} \tag{3.9}$$

Finally, we rewrite the universal quantification in the second part of the formula so that we can apply existential quantifier elimination. Applying double negation to the second big conjunct and propagating the inner negation inwards, we get:

$$\begin{aligned} &\exists v_1, ..., v_n. \left(\text{INV}(v_1, ..., v_n) \land \bigwedge_{(c,p) \in \text{rewrite}_q(t)} \big(c \implies \underline{\text{p}}(q) = p\big) \right) \land \\ \neg &\exists v_1, ..., v_n. \left(\text{INV}(v_1, ..., v_n) \land \bigvee_{(c,p) \in \text{rewrite}_q(t)} \big(c \land \underline{\text{p}}(q) < p\big) \right) \end{aligned} \tag{3.10}$$

In the following section we will now describe in detail how we perform existential quantifier elimination on this formula to eliminate quantification over all the changing variables $v_1, ..., v_n$ in both of the major conjuncts.

3.5.5 Quantifier Elimination

The quantifier elimination we use for our analysis is based on Cooper's method and follows closely the implementation described in [17] with some tweaks from [6]. It performs quantifier elimination on formulas written in Presburger arithmetic augmented with a divisibility predicate

$k \mid \cdot$ for $k \in \mathbb{Z}^+$. The semantics of $k \mid x$ is that x is divisible by k, or more formally: $k \mid x \iff \exists y.(x = ky)$. We will call *literals* the terms in formulas that do not contain a logical connective symbol ($\wedge$ or $\vee$) (literals can, however, contain logical negations).
The basic idea of Cooper's algorithm is to distinguish two cases in which $\exists x.F(x)$ could be true. Either there is a least x making $F(x)$ true or there is no least x making $F(x)$ true, meaning that for every x for which $F(x)$ is true, we will always find a smaller number x' that still makes $F(x')$ true. Cooper's algorithm will construct two formulas corresponding to both those cases. We will now briefly describe the four steps of the algorithm to eliminate existential quantification of x in the formula $\exists x.F(x)$.

Step 1 - Convert $F(x)$ to negation normal formal (NNF)
A formula is in NNF if it is a positive Boolean combination of conjunctions and disjunctions of literals. To convert a formula to negation normal form, we have to propagate all negations to the innermost algebraic literals. We can do this by repeatedly applying the De Morgan's laws and eliminating double negations. In our case, we are actually able to eliminate negations altogether (except around divisibility literals); our non-divisibility literals will after step 1 all be of the form $a \bullet b$ or $\neg(a \bullet b)$ where a and b are integer terms and $\bullet$ is one of $=$, $\neq$, $<$, $\leq$, $>$ or $\geq$. In the second case, we can further push the negation inwards by 'negating' the operator $\bullet$. This can be done by replacing $=$ by $\neq$, $<$ by $\geq$ and $>$ by $\leq$ and vice versa. Thus, after having executed step 1, we end up with the formula $F_1(x)$ where $\exists x.F_1(x)$ is equivalent to $\exists x.F(x)$ and $F_1(x)$ only contains negations around divisibility literals.

Step 2 - Collect x in terms
Collect terms containing the variable x such that x only occurs on one side of literals and only with positive coefficient. That means, all literals mentioning x have the form $hx \bullet t$, $t \bullet hx$, $k \mid hx + t$ or $\neg(k \mid hx + t)$ where t is a term that does not contain x and $h \in \mathbb{Z}_0^+$, $k \in \mathbb{Z}^+$. The result of this step is $F_2(x)$ where again $\exists x.F_2(x)$ is equivalent to $\exists x.F(x)$.

Step 3 - Normalize coefficients of x
In this step, let $\delta' = \mathbf{lcm}\{h : h \text{ is a coefficient of } x \text{ in } F_2(x)\}$ where **lcm** denotes the least common multiple. Now, literals in $F_2(x)$ are multiplied by constants so that δ' is the coefficient of x everywhere. That means, $hx \bullet t$ gets multiplied by h' such that $hh' = \delta'$, $k \mid hx + t$ by h' such that $hh' = \delta'$, and so on. This results in a formula $F_2'(x)$ in which each coefficient of x is δ'. Next, $\delta' x$ is replaced by a fresh variable x' to form $F_2'' := F_2'[\delta' x \mapsto x']$. By replacing $\delta' x$ by x' we implicitly assume that x' is divisible by δ'. To make this explicit, we add a divisibility constraint to the final result of step 3 which then becomes $F_3(x') := F_2''(x') \wedge \delta' \mid x'$. Each literal of $F_3(x')$ that contains x' has one of the following forms:

(A) $x' \bullet a$

(B) $b \bullet x'$

(C) $h \mid x' + c$

(D) $\neg(k \mid x' + c)$

where a, b, c and d are terms that do not contain x and $h, k \in \mathbb{Z}^+$. Moreover, $\exists x'.F_3(x')$ is equivalent to $\exists x.F(x)$.

Step 4.1 Construct the left infinite projection $F_{-\infty}(x')$
The left infinite projection is the formula corresponding to the case where there does not exist a

least x satisfying $F(x)$. The left infinite projection is formed by replacing terms $x' \bullet a$ and $b \bullet x'$ in $F_3(x')$ with either $\top$ or $\bot$ according to Table 3.3 where $\top$ means 'true' and $\bot$ means 'false'.

	$<$	$\leq$	$\geq$	$>$	$=$	$\neq$
$x' \bullet a$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$b \bullet x'$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$

Table 3.3: The replacements for $x' \bullet a$ and $b \bullet x'$, respectively, for the corresponding values of $\bullet$.

E.g. $x' < 4$ is replaced by $\top$ and $x' = 10$ is substituted by $\bot$. Intuitively speaking, every term that expresses that x' is smaller than a certain term is replaced by $\top$ and each term that expresses that x' is equal to or bigger than a certain value is replaced by $\bot$.

Step 4.2 Construct the final equivalent formula
Let $\delta = \mathbf{lcm}\{h \ : \ h \mid x' + c \text{ or } \neg(h \mid x' + c) \text{ is a literal in } F_3(x')\}$ and let B be the set of terms for each literal according to Table 3.4.

	$<$	$\leq$	$\geq$	$>$	$=$	$\neq$
$x' \bullet a$	–	–	$a-1$	a	$a-1$	a
$b \bullet x'$	b	$b-1$	–	–	$b-1$	b

Table 3.4: The terms that are added to the B set for every literal ('–' means that there is no term added for this literal).

The final quantifier-free formula which is equivalent to $\exists x.F(x)$ is defined by $F_4 := \bigvee_{j=1,\dots,\delta} F_\infty(j) \vee \bigvee_{j=1,\dots,\delta} \bigvee_{b \in B} F_3(b+j)$. $F_\infty(x')$ only mentions x' in divisibility literals. It asserts that an infinite number of small integers n satisfy F_3. If there exists an n that satisfies F_∞, then for every $k \in \mathbb{Z}^+$, $n - k\delta$ also satisfies F_∞. If there exists a least $n \in \mathbb{Z}$ satisfying F_3, this will be captured by the second major disjunct in F_4. For a more thorough explanation and a proof of correctness of the algorithm we refer to [17] and [6].

Note that existential quantification distributes over disjunction, i.e., $\exists x.(P(x) \vee Q(x)) \equiv \exists x.P(x) \vee \exists x.Q(x)$. We take advantage of this by splitting a formula into its disjuncts, applying quantifier elimination on each disjunct separately and then joining the resulting formulas again by disjunction. This can help in some cases to reduce the potential formula blowup caused by quantifier elimination.

3.5.6 Example

Let us consider a simple example. We want to eliminate existential quantification over i in the formula $\exists i.(q == i \wedge 0 \leq i \wedge i \leq 9)$ corresponding to the field access on line 6 of the example in

Listing 3.8. The formula is already in NNF. Therefore, after step 1, the formula is unchanged. The formula does not change in step 2 either as i is already collected in all literals. The same holds for step 3 as the only coefficient of i is 1, so $F_1(i) = F_2(i) = F_3(i) = (q == i \wedge 0 \leq i \wedge i \leq 9)$ (note that for $F_3(i)$ we can eliminate the additional divisibility literal as the least common multiple of the coefficients of i in F is 1 and $1 \mid x$ is $\top$ for all $x \in \mathbb{Z}$). The left infinite projection is equivalent to $\bot$ since $F_3(i)$ contains literals as conjuncts that get replaced by $\bot$ according to step 4.1. Our B set is, according to Table 3.4 $B = \{q-1, -1\}$. Thus, the final formula is $F_4 = \bigvee_{j=1} \bot \vee \bigvee_{j=1} \bigvee_{b \in \{q-1,-1\}} F_3(b+j) = \bigvee_{b \in \{q-1,-1\}} F_3(b+1) = F_3(q) \vee F_3(0) = (q == q \wedge 0 \leq q \wedge q \leq 9) \vee (q == 0 \wedge 0 \leq 0 \wedge 0 \leq 9)$ which can be simplified to $0 \leq q \wedge q \leq 9$.

3.5.7 Simplifying Formulas

Our approach of applying quantifier elimination in general leads to a blowup of formulas, in the worst case exponentially in terms of the number of literals contained in the formula. However, in many cases, we are able to simplify the resulting formula drastically. For example, the result from quantifier elimination will often contain constructs like $1 \geq 0$ which can be simplified to $\top$ or $a \wedge \bot$ which can be simplified to $\bot$. We implemented a whole list of such simplifications which as a whole help to reduce the size of formulas a lot. Some advanced simplifications could even go so far and analyze integer-related expressions semantically and simplify e.g. $A = 1 \vee 1 \leq A$ to $1 \leq A$. However, we leave more complex simplifications as future work since simplifying formulas was not our main focus.

3.6 Generating Specifications

In the sections so far, we described in detail how we collect permissions in our approach as well as the special handling of integer-dependent heap accesses. In this section we will explain how our algorithm actually translates our analysis results into Viper specifications and uses them to extend the program. Permissions generally have to be specified in preconditions of methods and in loop invariants. We will talk about a possible extension under certain assumptions to also infer postconditions in order to return remaining permissions from the callee to the caller. The generated specifications consist of two main parts corresponding to the two phases of our algorithm and some additional helper specifications. Again, we will cover the case of integer-dependent receivers separately (cf. Section 3.6.4).
For non-integer-dependent heap accesses, we will first describe how we turn permission trees into quantified permission expressions in Section 3.6.1. As described in Section 3.3, we introduce, for every leaf (r, p) in our collected permission trees, a set representing all the possible values r can take. In general, we will introduce one `Set` variable for every such receiver set. In Section 3.6.2 we will show how to specify those sets. In cases where the sets are not widened we will be able to omit the introduction of a set.
We will also briefly talk about how we specify symbolic variables that denote read permissions and how custom trigger generation will help us to decrease verification time in some cases.

3.6.1 Quantified Permission Expressions

The first part of the generated specifications corresponds to the collected permission trees. For every accessed field in the method, we generate exactly one quantified permission expression that declares the permissions expressed by the collected permission tree. The basic form of quantified permission expressions looks as follows:

```
forall r: Ref :: acc(r.f, permAmount(r))
```

where `permAmount(r)` is a permission expression that represents the required amount of permission for reference `r` depending on the permission tree. For the simple list traversal in Listing 3.12, the permission tree at top of the method consists of a single leaf corresponding to the field read on line 13. The quantified permission expression for the field `next` is

```
forall r: Ref :: acc(r.next, r in sx ? rdAmount : none)
```

```
 1  field next: Ref
 2
 3  method listTraversal(head: Ref, rdAmount: Perm, sx: Set[Ref])
 4  requires none < rdAmount && rdAmount < write
 5  requires forall r: Ref :: acc(r.next, r in sx ? rdAmount : none)
 6  requires head in sx && forall r: Ref :: r in sx ==> r.next in sx
 7  {
 8    var x: Ref := head
 9    while (x != null)
10    invariant forall r: Ref :: acc(r.next, r in sx ? rdAmount : none)
11    invariant x in sx && forall r: Ref :: r in sx ==> r.next in sx
12    {
13      x := x.next
14    }
15  }
```

Listing 3.12: List traversal

Note that we introduced a variable `rdAmount` which also appears as a formal argument to denote read permission. We also added a special precondition `none < rdAmount && rdAmount < write` to specify this variable, similarly to [24]. We will cover this symbolic read variable in more detail in Section 3.6.5.
In the following we will explain how each permission tree node or leaf gets converted to a permission expression.

Leaves
A leaf (x, p) in general gets converted to `(r in set_x ? p : none)` where `set_x` is the set corresponding to the receiver set of x. In case the receiver set corresponding to x is finite, in other words, is not widened, we can omit the introduction of a set and instead use one or multiple comparisons. To visualize this, consider the example Listing 3.13. In this example, the receiver set corresponding to `x` in the field access on line 8 is finite and contains the elements `y`

and `z`. In that case, the corresponding permission expression could be written as `forall r: Ref :: acc(r.f, (r == z || y != null && r == y ? write : none))`. As a further optimization, even the quantification could in this example be omitted as well and we could just require access to `y.f` and `z.f` separately.

```
1 method finiteReceiver(y: Ref, z: Ref)
2 {
3   var x: Ref         // (y != null ? (y, write) : (z, write))
4   x := z             // (y != null ? (y, write) : (x, write))
5   if (y != null) {
6     x := y           // (y, write)
7   }
8   x.f := x           // (x, write)
9 }
```

Listing 3.13: A simple example where the possible receivers for the field access on line 7 are `y` and `z`.

Maximum

To handle maximum constructs $t_1 \bowtie_{\text{max}} t_2$, we introduce a helper function `max` which returns the largest of two permission amounts, see Listing 3.14. Now, let `p1` and `p2` denote the permission expressions corresponding to t_1 and t_2, respectively. Then, the generated expression becomes `max(p1, p2)`.

```
function max(x: Perm, y: Perm): Perm {
  (x > y ? x : y)
}
```

Listing 3.14: The maximum function.

Addition

$t_1 + t_2$ is converted similarly to a maximum. Let again `p1` and `p2` denote the permission expressions corresponding to t_1 and t_2, respectively. The permission expression corresponding to $t_1 + t_2$ is `p1 + p2`.

Conditionals

The conversion of b ? t_1 : t_2 is straightforward. Let again `p1` and `p2` denote the permission expressions corresponding to t_1 and t_2, respectively. The generated expression then is `(b ? p1 : p2)`.

Zero-bounded permission tree

For the conversion of zero-bounded permission trees $\mathsf{bound}(t)$, we introduce another helper function `bound` (see Listing 3.15). Then, the conversion of $\mathsf{bound}(t)$, is `bound(p)` where `p` is the permission expression corresponding to permission tree t.

```
function bound(x: Perm): Perm {
  (x < none ? none : x)
}
```

Listing 3.15: The boundary function.

3.6.2 Receiver Set Descriptions

We have already seen in the last section that a finite receiver set can be directly used inline in the permission expression and therefore does not require an additional set variable. Furthermore, if the receiver is a function that has at least one integer-typed argument, it is handled specially and no receiver sets are needed (cf. Section 3.5). In this section, we therefore focus on the general case where a set contains an unknown number of elements that do not depend on integers. The three main expression types receivers can be made of are variables (local variables or arguments), field accesses and functions. Here, we present a basic way of expressing the receiver sets requiring a minimum amount of additional sets. To be more precise, we need exactly one variable of type `Set[Ref]` for each receiver set we define. Later we will see that in some cases we are even able to reduce the number of needed sets further by eliminating duplicate sets (cf. Section 3.8.3).
Given a widened receiver set, we build up the expression as follows. Let the set be called `set_x`. For every variable `x` occurring in the receiver set, we add the constraint `x in set_x`. For every field occurring in the receiver set, we add the constraint `forall r: Ref :: r in set_x ==> r.next in set_x`. For every function `f` that occurs in the receiver set, we add `forall v1, ..., vn: Ref :: v1 in set_1 && ... && vn in set_n ==> f(v1, ..., vn) in set_x` where `v1, ..., vn` denote the arguments of the function `f` and `set_1, ..., set_n` are the receiver sets corresponding to the arguments of `f`.
In some cases, we need an additional permission check (`perm(r.f) > none`) before we add a field to a set because the underlying verifier is not able to assert that we have the necessary permissions at that point. This might seem like an unsound restriction of possible values for the set. However, if the field `f` occurs in a receiver set description at some point then by definition our algorithm will already have requested permission to `r.f` for each possible `r` at that point. Thus, we always add this additional permission check to receiver set definition expressions.

3.6.3 Influence of Branch Conditions

In general we have to ignore conditions in our analysis (cf. Section 3.3.2). This means that we have to take the maximum of permissions of the two branches of the conditional. However, branch conditions often consist of one or multiple null checks that effectively prevent null dereferences in subsequent field accesses if those accesses are dominated by the respective condition. If we ignore such conditions completely we end up with permission expressions that request permissions to fields for too many receivers even though this could never happen during a concrete execution of the program. In Viper, requesting permission to a field with a possibly null-valued receiver automatically asserts that the receiver is non-null. Since we would like to infer the weakest possible precondition, we do not want to unnecessarily strengthen

preconditions that way. We therefore improved our inference of receiver sets to at least take into account conditions that impose such null-constraints.
To illustrate this, consider again Listing 3.12. Clearly, we never dereference null as the field access `x.next` on line 13 is guarded by the loop condition `x != null` (line 9). However, if we just ignore this condition we will keep adding `next` elements, hence implicitly restricting the list to be cyclic (e.g. if `head` is not null and `head.next` is pointing to `head` itself). Therefore, we do not only track receiver expressions but we track them together with a boolean value that indicates whether or not the expression is 'guarded' by a null check. The criterion whether or not a set can contain null is whether all concrete expressions contained in the set are guarded by a null check. If so, we include a null check for every expression that adds an element to that set. The list traversal example with included null checks can be seen in Listing 3.16.

```
field next: Ref

method listTraversal(head: Ref, rdAmount: Perm, sx: Set[Ref])
requires none < rdAmount && rdAmount < write
requires forall r: Ref :: acc(r.next, r in sx ? rdAmount : none)
requires (head != null ==> head in sx) &&
forall r: Ref :: r in sx ==> r.next != null ==> r.next in sx
{
  var x: Ref := head
  while (x != null)
  invariant forall r: Ref :: acc(r.next, r in sx ? rdAmount : none)
  invariant (x != null ==> x in sx) &&
  forall r: Ref :: r in sx ==> r.next != null ==> r.next in sx
  {
    x := x.next
  }
}
```

Listing 3.16: List traversal with null checks

A sketch of how our analysis performs the first two iterations on the program in Listing 3.16 is depicted in Table 3.5. The 'root' expression `x in sx` becomes `x != null ==> x in sx` and the quantified part becomes `r in sx ==> r.next != null ==> r.next in sx`. An expression is guarded by a condition if the program location where the expression occurs is dominated by that condition, for instance, if all paths that pass through the expression also pass through the condition and the expression and condition do not contain variables that change in between.
In some cases part of a condition can be influenced by a previous null check in the condition itself. Consider e.g. the condition `node != null && !node.marked`. Clearly, the set of possible receivers for the field access `node.marked` can never be `null`. We try to account for this fact at least partially as far as the Viper framework allows by splitting up branch conditions to their conjuncts and processing each conjunct one after the other backwards to detect cases where a conjunct contains a field access guarded by a null check in a previous conjunct. Therefore, for the example `node != null && !node.marked`, we split up the condition around the `&&` and

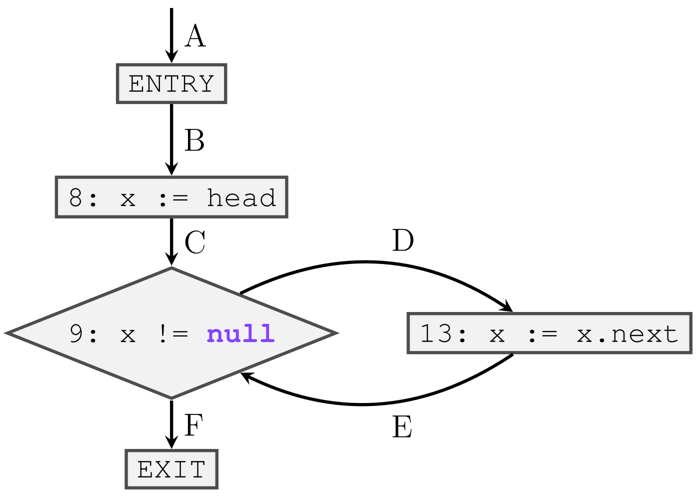


Figure 3.3: The control flow graph to the program in Listing 3.12.

Code		**Analysis steps**		
			Iteration 1	Iteration 2
#	**`ENTRY`**	A	{`(head, true)`}	{`(head, true), (head.next, true)`}
8	`x := head`	B	{`(head, true)`}	{`(head, true), (head.next, true)`}
9	`while(x != null)`	C	{`(x, true)`}	{`(x, true), (x.next, true)`}
13	`x := x.next`	D	{`(x, false)`}	{`(x, false), (x.next, true)`}
	`EXIT`	E	{}	{`(x, true)`}
		F	{}	{}

Table 3.5: 2 iterations of our analysis run on the program in Listing 3.12. The letters A - F denote the edges in the control flow graph in Figure 3.3.

first process the second conjunct where we record the field access to `marked` in `!node.marked`. Only afterwards we process the first conjunct `node != null` and hence can add the non-null condition to the receiver of `!node.marked`.

3.6.4 Integer-Dependent Quantified Permissions

As we have seen in Section 3.5, we convert permission trees corresponding to integer-dependent receivers to an assertion about a placeholder function $\underline{\mathrm{p}}$. Therefore, the basic layout of permission expressions corresponding to such an assertion looks as follows:

```
forall q: Int :: acc(rec(q, e1, e2, ...).val, p(q))
```

where `rec` is the receiver function which depends on `q` and in general one or multiple further constant expressions. In the array case, the receiver function would be `loc(arr, q)`. Finally, we also have to specify the helper function `p` appropriately. We do this by introducing an abstract function with an empty body, no preconditions and one postcondition which is exactly the assertion obtained in the procedure described in Section 3.5. Note that we replaced $\underline{\mathrm{p}}(q)$ with **`result`** in order to specify the postcondition correctly. In Listing 3.17 we can see how this will look like for a simple array traversal of indexes 0 to 9. Moreover, we can also see that even for rather simple examples, the quantifier elimination produces big formulas. This blowup of formulas is one of the current problems of our approach which we will discuss in Section 5.7.

```
function p(q: Int): Perm
ensures ((q != 0 || result == write) && (q == 0 || result == none) || q
  <= 8 && q >= -1 && result == none || q <= 9 && q >= 0 && result ==
  write) && ((q != 0 || result >= write) && (q == 0 || result >= none) &&
   (q > 9 || q < 0 || result >= write) && (q > 8 || q < -1 || result >=
  none))

method arrayExample(arr: Array)
requires (forall q: Int :: acc(loc(arr, q).val, p(q)))
{
  var i: Int := 0
  while (i < 10)
  invariant (forall q: Int :: acc(loc(arr, q).val, p(q)))
  invariant i >= 0
  {
    loc(arr, i).val := i
    i := i + 1
  }
}
```

Listing 3.17: Simple array traversal.

Note that in general the assertion in the $\underline{\mathrm{p}}$ function can depend on other variables present in the method scope. In that case, we have to pass these variables on to $\underline{\mathrm{p}}$ and extend the declaration of $\underline{\mathrm{p}}$ with additional corresponding formal arguments.

3.6.5 Symbolic Read Variable

While field writes require **write** permission, field reads only require a fractional permission amount that is bigger than **none**. For this, we introduce a symbolic read variable called `rdAmount`. Whenever a symbolic read access occurs in a permission leaf of any of the collected permission trees inside a method, we will have to add this variable as a formal argument to the argument list of the method. Moreover, we have to restrict the variable to be positive and we also know for sure that the variable will always be less than **write**. Thus, `none < rdAMount && rdAmount < write` might seem to be a reasonable specification. However, in some cases, the upper bound **write** is not strong enough. In Listing 3.18, a permission amount of `1 / 2` is exhaled to `r.f` before a field read happens to the same location. Constraining `rdAmount` to be smaller than **write** here is not enough. In that case, a caller that passes `3 / 4` for `rdAmount` would also fulfill the precondition. However, the permission expression would then in total require a permission amount of `5 / 4` to `r.f` which is equivalent to false. Hence, we have to restrict `rdAmount` to be less than `1 / 2`. If `rdAmount` is specified that way, we are sure that also `1 / 2 + rdAmount` will be strictly less than **write**. Note that this is not an issue related to precision and not to soundness as our program would still verify with **write** as the upper bound.

```
method exhaleRead(r: Ref, rdAmount: Perm) returns (res: Ref)
requires none < rdAmount && rdAmount < 1 / 2
requires (forall r0: Ref :: acc(r0.f, (r0 == r ? 1 / 2 : none) + (r0 == r
   ? rdAmount : none)))
{
  exhale acc(r.f, 1 / 2)
  res := r.f
}
```

Listing 3.18: An exhale followed by a read. The permission required to `r.f` is strictly larger than `1 / 2` as after the exhale, there still has to be enough permission for the following read.

We will now briefly describe how we calculate an upper bound for the `rdAmount` variable. Note that in general this bound can be too restrictive. The bound could be **none** or even negative in which case we would fall back to **write** and accept the imprecision caused thereby. If we would restrict `rdAmount` by `none < rdAmount && rdAmount < none` this would result in a contract that is unsatisfiable, i.e., that is even more imprecise.
We compute a bound for `rdAmount` by examining all possible permission sums through every permission tree by summing up the corresponding permissions and occurrences of symbolic read amounts separately. Sums are written as pairs $(n/d, k)$ where n/d denotes the fractional permission amount and k denotes the number of reads. Permission sums are recursively computed as follows:

Leaves

$$\text{paths}((r,p)) = \begin{Bmatrix} \{(0/1,1)\}, & \text{if } p = \text{read} \\ \{(n/d,0)\} & \text{if } p = \text{n / d} \end{Bmatrix}$$

Maximum
$\text{paths}(t_1 \bowtie_{\max} t_2) = \text{paths}(t_1) \cup \text{paths}(t_2)$

Addition
$\text{paths}(t_1+t_2) = \{((n_1 d_2+n_2 d_1)/(d_1 d_2), k_1+k_2) \mid (n_1/d_1, k_1) \in \text{paths}(t_1), (n_2/d_2, k_2) \in \text{paths}(t_2)\}$

Conditionals
$\text{paths}(\text{b} \ ? \ t_1 \ : \ t_2) = \text{paths}(t_1) \cup \text{paths}(t_2)$

Zero-bounded permission tree

$$\text{paths}(\mathsf{bound}(t)) = \begin{array}{ll} \{(n/d, k) & \mid (n/d, k) \in \text{paths}(t) \wedge n \geq 0\} \ \cup \\ \{(0, k) & \mid (n/d, k) \in \text{paths}(t) \wedge n < 0\} \end{array}$$

For every pair $(n/d, k)$, the restriction we put on `rdAmount` is as follows:

$$\frac{n}{d} + k \cdot \texttt{rdAmount} < 1 \tag{3.11}$$

Thus, the final upper bound for `rdAmount` in a permission tree t will be calculated as follows:

$$\texttt{rdAmount} < \max_{(n/d,k) \in \text{paths}(t)} \left\{ \frac{d-n}{kd} \right\} \tag{3.12}$$

3.6.6 Triggers

Universal quantifiers in Viper may be annotated with triggers. Triggers are Viper expressions that restrict the possible instantiations of the quantified expression. It is possible to declare triggers explicitly in curly braces directly after the double colon in **forall** expressions. We have already seen the use of triggers in the injectivity axiom for arrays (see Listing 3.19).

```
axiom allDiff {
  forall a: Array, i: Int :: {loc(a, i)} rToA(loc(a, i)) == a && rToI(loc
    (a, i)) == i
}
```

Listing 3.19: The injectivity axiom from the array definition (cf. Listing 2.3).

The trigger in this case restricts the possible instantiations of the axiom `allDiff` to expressions of the form `loc(a, i)`. If no trigger is specified, Viper will automatically infer one or multiple trigger sets. However, in our experimentations, we found that adding custom triggers to our specifications of receiver sets reduces verification time substantially for some examples. Consider the following part of a set specification.

```
forall r: Ref :: (r in s_n && r != null && r in s_n) ==> r.next in s_n
```

By default, Viper will infer the following trigger sets:
`{ (r in s_n) }, { (r.left in s_n) }`
We prevent this by providing a single custom trigger set
`{ (r in s_n), r.left }`.

3.7 Inferring Postconditions using a Forward Analysis

In this section, we summarize the main findings of our research in inferring postconditions. Originally, our goal was to incorporate a forward analysis to infer method postconditions into our algorithm. However, there are some obstacles that prevent a completely automatic inference of postconditions at the moment. We motivate the automatic inference of postconditions, give basic ideas how a forward inference could look like and show up the main difficulties in developing a fully automated approach.

3.7.1 Motivation

The main motivation to also infer postconditions of a method is precision. When no explicit postconditions are specified for a method in Viper, the default postcondition is **`true`**. For single methods this is no problem as a method will also verify without specified postconditions. However, in general programs, methods can call other methods and for a caller method, **`true`** is a very weak assertion. The goal is to be as precise as possible, in other words, to come up with strongest postconditions with respect to permissions. Consider the method `caller` in Listing 3.20 which calls another method, `callee`. The method `callee` requires **`write`** permission to `x.next` and performs a simple write to it. In the program state after the assignment, `callee` still holds **`write`** permission to `x.next`. When there are no postconditions specified, the problem is that this **`write`** permission is not returned to the caller because Viper will not infer automatically that this permission could be returned to the caller without an explicit declaration in a postcondition. Thus, for `caller`, the method call to `callee` looks like an **`exhale`** to `x.next`. As a consequence, the caller himself will in turn not be able to, e.g., write to `x.next` although a stronger postcondition of `callee` would ensure that it returns the permissions to `x.next`. The verification of this program would fail which is imprecise.

```
method callee(x: Ref)
requires acc(x.next)
{
  x.next := x
}

method caller(x: Ref)
requires acc(x.next)
{
  callee(x)    // Permission to x.left is leaking
  x.next := x  // Will fail to verify
}
```

Listing 3.20: Leaking permissions.

In this example, the postcondition of `callee` is exactly the same assertion as the postcondition, namely **`ensures`** **`acc`**`(x.next)`, which is at the same time the strongest postcondition. However, coming up with sound and precise postconditions in general is hard if the method body contains loops.

3.7.2 Tracking changing References

An additional difficulty with forward propagation is that we have to keep track of changing references. In Listing 3.21, if we recorded the receiver of the field assignment on line 11 simply as `x.left`, we would not have captured the fact that `x.left` points to `y` at that point of the program (due ot the assignment on line 10). We have to record the previous assignment to `x.left` and appropriately plug in `y` for `x.next` when generating the postcondition.
Moreover, the field access `x.left.right` on line 9 cannot be represented in the post-state of the method. Because `x.left` is overwritten on lines 9 and 10, there exists no access path that could denote the original location `x.left`. Therefore, the only way to express the original heap location `x.left` pointed to is with the use of the **`old`** keyword.

```
 1 method oldExample(x: Ref, y: Ref)
 2 requires acc(x.left)
 3 requires acc(y.right)
 4 requires acc(x.left.right)
 5 ensures acc(x.left)
 6 ensures acc(y.right)
 7 ensures acc(old(x.left).right)
 8 {
 9   x.left := x.left.right
10   x.left := y
11   x.left.right := y
12 }
```

Listing 3.21: An example that demonstrates the use of the **`old`** keyword. Note that the use of **`old`** is inevitable in this case. Because of the assignments to `x.left` on lines 10 and 11, there is no other expression that could describe the memory location `x.left` was pointing to at the beginning of the method.

This example illustrates well the main challenges compared to the backward analysis: it is much more difficult to keep track of how variables and the heap changes. Moreover, the generated specifications have to be expressed in the post state of the method, possibly using the **`old`** keyword. We have seen that is not sufficient to only track permissions. We also have to keep record of which expressions denote which heap locations. We believe that integrating a heap analysis here could help us again.

3.7.3 Strongest Postcondition Rules

In the following we informally describe strongest postcondition rules for straight-line code, similarly to the weakest precondition rules described in Section 3.2 for the backward analysis. Soundness for strongest postcondition rules in terms of permission means that we return *at most* all the permissions that are available at the end of the method. The main difference will be the join operator for branches of conditionals or loops. For this, we will introduce the $\bowtie_{\min}$ operator where $t_1 \bowtie_{\min} t_2$ returns the minimum of the permission amounts represented by two trees t_1 and t_2. In other words, $\bowtie_{\min}$ is the counterpart of $\bowtie_{\max}$. We will again use the same concept of permission trees to denote the collected permissions. Note that the described approach collects information about postconditions independently of the inferred precondition for the method. Another possibility would be to assume that the backward inference to infer postcondition has already been conducted and to start with the collected permissions at the top of the method. For that analysis, different rules would be needed and we do not consider this case here. Strongest postconditions in terms of permissions means that after executing every statement, we want to maintain as many permissions as possible.

Variable Assignments
For a reference variable assignment `v := e`, every occurrence of `v` in the tracked permission expressions has to be replaced by `e`. Furthermore, we have to reflect the change of variable in tracked heap information accordingly. This is important for cases where, e.g., the variable assignment is followed by a field assignment `v.f := a`. This field assignment would then need to be correctly recognized as an effective write to `e.f`.

Field Assignments
For a field assignment `l.f := r`, we add write permissions to field `f` using the $\bowtie_{\text{max}}$ operator only for every expression `e` that *must* alias with `l` at that point. If we would add permissions to references that only *may* alias with `l`, this would be unsound. Furthermore, we have to reflect the heap update in the tracked heap information.

Inhales
For an `inhale acc(r.f, p)`, we add the inhaled permissions `p` to the tracked permissions using the $+$ operator. While doing this, we have to under-approximate the locations to which permission is inhaled. Consequently, we only add the permission to locations that *must* alias with `r` at that point.

Exhales
For an `exhale acc(r.f, p)`, we subtract the exhaled permissions `p` from the tracked permissions using the $-$ operator. While doing this, we have to over-approximate the locations to which permission is exhaled. Therefore, we have to subtract the permission from all locations that *may* alias with `r` at that point.

Conditionals
When joining conditionals, we can again use a ternary expression as in Section 3.2.7. However, in cases where the condition has to be over-approximated, we do not use the $\bowtie_{\text{max}}$ operator as this was the case for weakest preconditions, but we use the $\bowtie_{\text{min}}$ operator to join the permission trees from each branch.

Loops
Similarly to conditionals, we join the collected permissions from both branches using the $\bowtie_{\text{min}}$ operator. The main problem here is that applying the $\bowtie_{\text{min}}$ operator is generally a heavy over-approximation and will presumably often imprecisely result in zero permissions. Again, we would need to approximate receiver sets in the true branch of the loop, but this time we would have to *under-approximate* them. In other words, we would need guarantees how many times a loop is *at least* executed. This is where our ideas for a forward analysis get vague, but we believe that an under-approximating abstract interpretation could help us here. There already exists work about under-approximating backward integer analysis [16].

3.8 Simplification of Inferred Contracts

In this section, we briefly present the various simplifications we apply before the actual generation of the specifications.

3.8.1 Simplifications of Permission Trees

In a first step we simplify the collected permission trees on a purely syntactical level. In particular, we perform the following simplifications:

Flattening nested maximum operators
As an example, consider the permission tree $(t_1 \bowtie_{\max} t_2) \bowtie_{\max} t_3$. We simplify this tree by flattening the tree structure into a set $\{t_1, t_2, t_3\}$. The maximum operator will then operate as a maximum over all the trees in this set.

Eliminating identical permission trees inside the maximum
Clearly, taking the maximum of two identical permission trees is just one of the trees:
$t \bowtie_{\max} t = t$.

Simplifying conditionals
We can simplify a conditional syntactically if either the condition is a literal or both trees are the same. E.g., (true ? t_1 : t_2) is simplified to t_1 and (b ? t : t) to t.

Eliminating unnecessary bound operators
bound(bound($-t_1$) $-$ t_2)) can be simplified to bound($-t_1 - t_2$).

3.8.2 Semantic Simplifications

In the second step we simplify the collected permissions further by also considering the corresponding receiver sets. We simplify permission expressions like

```
max(r in set1 ? p1 : none, r in set2 ? p2 : none)
```

to (`r in set2 ? p2 : none`)

if `p2` $\geq$ `p1` and if we can statically determine that the receiver set corresponding to `set1` is a subset of the receiver set corresponding to `set2`. This simplification can be done for the example in Listing 3.22. The resulting permission trees at each point in the program are indicated in comments. On line 4, we will be able to statically determine that the receiver sets corresponding to the field accesses `x.f` on lines 4 and 5 both consist of the single element `x`. Furthermore, `write` ¿ `rdAmount`. Therefore, we can simplify `max(r == x ? write : none, r == x ? rdAmount : none)` to `(r == x ? write : none)`. Listing 3.23 shows the simplified example.

```
method simplification(x: Ref, rdAmount: Perm) returns (res: Ref)
requires none < rdAmount && rdAmount < write
requires (forall r: Ref :: acc(r.f, max(r == x ? write : none, r == x ?
  rdAmount : none)))
{
  x.f := x    // (x, write) ⋈max (x, read)
  res := x.f  // (x, read)
}
```

Listing 3.22: A simple field write followed by a field read.

```
method simplification(x: Ref) returns (res: Ref)
requires (forall r: Ref :: acc(r.f, (r == x ? write : none)))
{
  x.f := x    // (x, write) ⋈max (x, read)
  res := x.f  // (x, read)
}
```

Listing 3.23: The example from Listing 3.22 with the simplification applied.

3.8.3 Eliminating Duplicate Sets

Remember that we track one receiver set per field access (cf. Section 3.6.2). We then introduce an additional set helper variable for every receiver set that is not finite and can therefore only be expressed in terms of an additional helper set. Thus, in the worst case, we introduce one helper set for every field access that occurs in the program.
However, in a lot of cases we infer multiple receiver sets that capture exactly the same heap locations. For readability and especially to make the inferred contracts shorter and more concise, it is be desirable to eliminate those duplicate sets and instead just use one set everywhere. The main advantage will be a reduced amount of sets that we have to generate set descriptions for and thus a simplifications of the generated specifications. An example of the effect of merging sets can be seen in the method `sumList` for which we show both versions, the unoptimized one in Listing 3.24 and the optimized one Listing 3.25. The two helper sets `s_n` and `s_n0` in the unoptimized version have the exact same specification and only differ by the name. Therefore, the two sets can be merged.

3.9 Restrictions and current Limitations

We will shortly list in this section the most important limitations our current algorithm has. In Chapter 5, we will discuss how some of these restrictions could be lifted in future work.

```
method sumList(nd: Ref, rdAmount: Perm, s_n: Set[Ref], s_n0: Set[Ref])
  returns (sum: Int)
requires none < rdAmount && rdAmount < write
requires forall _r: Ref :: acc(_r.next, _r in s_n ? rdAmount : none)
requires forall _r: Ref :: acc(_r.data, _r in s_n0 ? rdAmount : none)
requires nd != null ==> (nd in s_n)
requires (forall _r: Ref :: { (_r in s_n),_r.next } (_r in s_n) && (_r !=
    null) && perm(_r.next) > none && _r.next != null ==> (_r.next in s_n))
requires nd != null ==> (nd in s_n0)
requires (forall _r: Ref :: { (_r in s_n0),_r.next } (_r in s_n0) && (_r
  != null) && perm(_r.next) > none && _r.next != null ==> (_r.next in
  s_n0))
{
  var node: Ref := nd
  sum := 0
  while (node != null)
  invariant forall _r: Ref :: acc(_r.next, _r in s_n ? rdAmount : none)
  invariant forall _r: Ref :: acc(_r.data, _r in s_n0 ? rdAmount : none)
  invariant node != null ==> (node in s_n)
  invariant (forall _r: Ref :: { (_r in s_n),_r.next } (_r in s_n) && (_r
      != null) && perm(_r.next) > none && _r.next != null ==> (_r.next in
    s_n))
  invariant node != null ==> (node in s_n0)
  invariant (forall _r: Ref :: { (_r in s_n0),_r.next } (_r in s_n0) && (
    _r != null) && perm(_r.next) > none && _r.next != null ==> (_r.next
    in s_n0))
  {
    sum := sum + node.data
    node := node.next
  }
}
```

Listing 3.24: Summation over a list (unoptimized).

3.9.1 Heap Analysis: Assuming that Argument Heaps are Disjoint

The specific implementation of heap analysis we use assumes that for method arguments, all the memory chunks reachable through any access path from each argument are disjoint with respect to the memory chunks of the other arguments. We use this heap analysis as it is, hence making the same assumption for our analysis. Of course we can always fall back to our general rule for field assignments (cf. Section 3.2.4).

3.9.2 Changing permissions in Loops

In our analysis we over-approximate point-wise required permissions and receiver sets. For inhales and exhales, we do not over-approximate the permission amount itself. Inside loops, we assume that exhales and inhales always happen to distinct locations. This has the consequence

```
method sumList(nd: Ref, rdAmount: Perm, s_n: Set[Ref])
  returns (sum: Int)
requires none < rdAmount && rdAmount < write
requires forall r: Ref :: acc(r.next, (r in s_n) ? rdAmount : none)
requires forall r: Ref :: acc(r.data, (r in s_n) ? rdAmount : none)
requires nd != null ==> (nd in s_n)
requires (forall r: Ref :: { (r in s_n),r.next } (r in s_n) && (r != null
  ) && perm(r.next) > none && r.next != null ==> (r.next in s_n))
{
  // ...
}
```

Listing 3.25: The optimized example from Listing 3.24. We only show the method declaration together with the precondition, the invariant will be simplified accordingly as well.

that we are not able to handle multiple inhales or exhales to the same location if they occur inside a loop. Consider Listing 3.26 where a permission amount of `1 / 10` is exhaled to index 0 in the loop body. In our permission tree, the exhale would be represented by a single leaf `(loc(arr, 0).val, 1 / 10)`. The generated permission expression for both the invariant and the precondition would then provide a permission amount of `1 / 10` to index 0. This is clearly unsound as the exhaled permission to index 0 will in total be `10 / 10` which is equal to `write`.
There is one more problem with this example. If we would be able to over-approximate permission amounts as well, a precondition equivalent to `acc(loc(arr, 0).val, write)` would be sound. However, we would still have to come up with an invariant that expresses the changing permission amount. However, we only infer invariants that express a fixed permission amount for every single heap location. In order to express the invariant, we would have to include `i` in our specification. A possible invariant for this example would be `acc(loc(arr, 0).val, write - (i / 10))`.

```
method multiExhale(arr: Array)
requires acc(loc(arr, 0).val, write)
{
  var i: Int := 0
  while (i < 10)
  invariant acc(loc(arr, 0).val, write - (i / 10))
  {
    exhale acc(loc(arr, 0).val, 1 / 10) // (loc(arr, 0).val, 1 / 10)
    i := i + 1
  }
}
```

Listing 3.26: Multiple exhales to the same location.

3.9.3 No Support of Nested Quantifiers in Quantified Permissions

There is currently no support in Viper for nested quantifiers in quantified permission expressions. This means, it is not possible to quantify over multiple variables when specifying permissions. The consequence for us is that we cannot handle cases where the receiver function depends on more than one changing integer argument. This would be interesting in many cases. For example, we could be interested in modeling a matrix domain as in Listing 3.27. A memory location in the matrix is uniquely determined by its column and row indexes and can be accessed by the `loc2` function. In order to express a quantified permissions for matrices, our permission expression would in general look as follows:

```
forall i: Int, j: Int :: acc(loc2(mat, i, j).val, p(i, j))
```

The Viper verifiers currently cannot handle such permission expressions, but this restriction is expected to be lifted in future.

```
domain Matrix {
  function loc2(a: Matrix, i: Int, j: Int): Ref
  function rows(a: Matrix): Int
  function cols(a: Matrix): Int
  function rToM(ref: Ref): Matrix
  function rToR(ref: Ref): Int
  function rToC(ref: Ref): Int

  axiom allDiff {
    forall m: Matrix, i: Int, j: Int :: {loc2(m, i, j)} rToM(loc2(m, i, j
      )) == m && rToR(loc2(m, i, j)) == i && rToC(loc2(m, i, j)) == j
  }

  // Further axioms ...
}
```

Listing 3.27: A possible matrix domain.

4 Evaluation

In the following, we present some concrete Viper programs on which we ran our algorithm. We discuss the inferred specifications in terms of precision and readability and also do some comparisons to specifications inferred using the algorithm of previous work [24]. For programs with heap-dependent field accesses, the generated formula is very large even for small examples. We therefore only consider basic examples and moved in some cases the specification for the permission function $\underline{\mathrm{p}}(q)$ to the appendix.

4.1 Simple List Traversal

The program in Listing 4.1 is a slightly adapted example from [24] and traverses a list. It sets the `is_marked` flag of every node to **`true`**. At first sight, the specification looks very complex compared to the actual program. However, there is only one helper set. On lines 5 and 6 we can see the permission expressions generated for the fields `is_marked` and `next` as described in Section 3.6.1, respectively. Lines 7 and 8 describe the specification of the set `s_n` according to Section 3.6.2. Compared to [24] we have an imprecision due to our handling of branch and loop conditions. In fact, we would only need to put consecutive nodes into the set until we encounter the first node which has its `is_marked` field set to true. However, we actually put the whole list into the set. This is attributable to our over-approximation of conditions to the maximum of both branches in the widening step (cf. Section 3.3.2).
On the other side, what especially stands out in this example is that, despite there are three field accesses in the whole program, we only need one receiver set. In the simplification phase, our algorithm is able to merge all three receiver sets corresponding to the field accesses in the program into one set. [24] on the other hand needed three helper sets in order to specify this program.

```
method traverse(nd: Ref, rdAmount: Perm, s_n: Set[Ref])
requires none < rdAmount && rdAmount < write
requires forall r: Ref :: acc(r.is_marked, (r in s_n) ? write : none)
requires forall r: Ref :: acc(r.next, (r in s_n) ? rdAmount : none)
requires nd != null ==> (nd in s_n)
requires (forall r: Ref :: { (r in s_n),r.next } (r in s_n) && (r != null
  ) && perm(r.next) > none && r.next != null ==> (r.next in s_n))
{
  var node: Ref := nd
  while (node != null && !node.is_marked)
  invariant forall r: Ref :: acc(r.is_marked, (r in s_n) ? write : none)
  invariant forall r: Ref :: acc(r.next, (r in s_n) ? rdAmount : none)
  invariant node != null ==> (node in s_n)
  invariant (forall r: Ref :: { (r in s_n),r.next } (r in s_n) && (r !=
    null) && perm(r.next) > none && r.next != null ==> (r.next in s_n))
  {
    node.is_marked := true
    node := node.next
  }
}
```

Listing 4.1: Simple list traversal.

4.2 Simple Heap Manipulation

The example in Listing 4.2 shows how our field assignment rule works in practice. Let us consider the field access to `right` on the right side of the assignment on line 10. The receiver of this access is `b.left.left`. When we propagate over the field assignment on line 9, we have to consider three cases:

Case 1: `a` aliases with `b`. Therefore, `b.left.left` has to be replaced with `c.left`.

Case 2: `a` aliases with `b.left` in which case `b.left.left` has to be replaced with `c`.

Case 3: `a` does not alias with `b`, nor with `b.left` in which case `b.left.left` remains unchanged.

Note the interesting sub-case of case 1 in which `a` and `b` both alias with `c` as well. In that case, the assignment of `c` to `a.left` effectively makes `a.left`, `b.left` and `c.left` all point to `c`. In that case, `b.left.left` would have to be replaced by `c`. Our rule takes care of this as well. Furthermore, note that we introduced an additional abstract helper function `getLeft`. We cannot directly use field accesses in the permission expression on lines 5 and 6 because they these expressions not self-framing: in order to mention, e.g., `b.next`, we first need permission to `b.next` which we only get in the expression itself. Therefore, we use an inhale-exhale pair which inhales that `getLeft(r)` returns `r.next` for all references `r` whenever it has required permission.

```
function getLeft(x: Ref): Ref

method fieldAssignment(a: Ref, b: Ref, c: Ref, rdAmount: Perm)
requires none < rdAmount && rdAmount < write
requires forall r: Ref :: acc(r.left, max(max(r == a ? write : none, r == b ? write : none), r == (b == a ? c : getLeft(b)) ? rdAmount : none))
requires forall r: Ref :: acc(r.right, r == ((b == a ? c : getLeft(b)) == a ? c : getLeft(b == a ? c : getLeft(b))) ? rdAmount : none)
requires [forall r: Ref :: perm(r.left) > none ==> getLeft(r) == r.left, true]
{
  a.left := c
  b.left := b.left.left.right
}
```

Listing 4.2: Field assignments in straight line code which illustrate our field assignment rule.

4.3 Tree Traversal

The method `traverseRandom` in Listing 4.3 traverses a tree by taking two steps per loop iteration. First, it follows the left subtree and then, depending on the iteration variable `i`, it follows the left or the right subtree. This example is also taken from [24]. The main shortcoming in our inferred specifications is that there is an imprecision compared to [24]. The field accesses on lines 26 and 28 are not dominated by a **`null`** check. That means, that we will have to mark those nodes to be potentially **`null`** in our analysis. However, if the program argument `nd` is **`null`**, the loop will not be executed at all and therefore no field accesses happen. On line 5 we see that we require `nd` to be in the set `s_n`, independently of whether or not `nd` is **`null`**. As we require read permission to all elements in `s_n`, we will implicitly restrict `nd` to be non-null. Thus, we unnecessarily restrict the possible input parameters which is imprecise.

```
1  method traverseRandom(nd: Ref, rdAmount: Perm, s_n: Set[Ref], s_n0: Set[Ref])
2  requires none < rdAmount && rdAmount < write
3  requires forall r: Ref :: acc(r.right, (r in s_n) ? rdAmount : none)
4  requires forall r: Ref :: acc(r.left, (r in s_n) ? rdAmount : none)
5  requires (nd in s_n)
6  requires forall r: Ref :: { (r in s_n),r.left } (r in s_n) && r != null && perm(r.left) > none ==> (r.left in s_n)
7  requires forall r: Ref :: { (r in s_n),r.right } (r in s_n) && r != null && perm(r.right) > none ==> (r.right in s_n)
8  requires nd != null ==> (nd in s_n0)
9  requires forall r: Ref :: { (r in s_n0),r.left } (r in s_n0) && r != null && perm(r.left) > none && r.left != null ==> (r.left in s_n0)
10 requires forall r: Ref :: { (r in s_n0),r.right } (r in s_n0) && r != null && perm(r.right) > none && r.right != null ==> (r.right in s_n0))
11 {
  var i: Int := 0
  var node: Ref := nd
  while (node != null)
  invariant forall r: Ref :: acc(r.right, (r in s_n) ? rdAmount : none)
  invariant forall r: Ref :: acc(r.left, (r in s_n) ? rdAmount : none)
  invariant (node in s_n)
  invariant forall r: Ref :: { (r in s_n),r.left } (r in s_n) && r !=
    null && perm(r.left) > none ==> (r.left in s_n)
  invariant forall r: Ref :: { (r in s_n),r.right } (r in s_n) && r !=
    null && perm(r.right) > none ==> (r.right in s_n)
  invariant node != null ==> (node in s_n0)
  invariant forall r: Ref :: { (r in s_n0),r.left } (r in s_n0) && r !=
    null && perm(r.left) > none && r.left != null ==> (r.left in s_n0)
  invariant forall r: Ref :: { (r in s_n0),r.right } (r in s_n0) && r !=
    null && perm(r.right) > none && r.right != null ==> (r.right in s_n0)
  {
    node := node.left
    if (i < 5 || i > 10) {
      node := node.left
    } else {
      node := node.right
    }
    i := i + 1
  }
}
```

Listing 4.3: Tree traversal.

4.4 Morris' Tree Traversal

An interesting algorithm for traversing a binary tree in linear time with respect to the number of nodes and using only a constant amount of additional space was described by Morris in 1979 [18]. An implementation of the original algorithm written in Java can be seen in Listing 4.4. Note, that the only way to translate `do { body } while(cond)` to Viper without having to use `goto` is using a loop unrolling `body; while(cond) { body }`. Since the number of fields in this algorithm is 4 and the program contains 20 field accesses in total (8 writes and 12 reads), we would expect the generated specification to be rather complex. However, we can see in Listing 4.5 that thanks to our several simplifications (cf. Section 3.8) we are able to reduce the number of sets to one. Quite surprisingly, there is also only one ternary condition operator in every permission expression in the loop invariant (lines 26 to 29) as we are able to determine statically that field reads and writes happen to the exact same set of references and therefore can eliminate the maximums in our collected permissions completely by repeated application of the rule presented in Section 3.8.2. However, in the precondition, we have an additional leaf in every permission tree, which corresponds to the accesses to `x` in the first (unrolled) iteration. We cannot merge `root` with the set `s_x` because `s_x` only contains `root` if `root` is not `null`. Therefore, we end up at method entry having only 2 leaves per permission tree or 8 in total

(two for every of the 4 fields), compared to the before mentioned total 20 field accesses in this program.
For this example, we also observed a big positive impact of our custom triggers to the verification runtime. We did not perform exact statistic measurements, but verification takes about half of the time with our custom triggers than it would take with the generic ones.

```java
void markTree(Tree root) {
  Tree x, y;
  x = root;
  do {
      x.mark = true;
      if (x.left == null && x.right == null) {
            y = x.parent;
      } else {
          y = x.left;
          x.left = x.right;
          x.right = x.parent;
          x.parent = y;
      }
      x = y;
  } while (x != null);
}
```

Listing 4.4: Morris' tree traversal algorithm in Java.

```
field mark: Bool; field left: Ref; field right: Ref; field parent: Ref

method markTree(root: Ref, set_x: Set[Ref])
requires forall r: Ref :: acc(r.parent, max((r == root ? write : none), (
  r in s_x ? write : none)))
requires (forall r: Ref :: acc(r.right, max((r == root ? write : none), (
  r in s_x ? write : none))))
requires forall r: Ref :: acc(r.left, max((r == root ? write : none), (r
  in s_x ? write : none)))
requires forall r: Ref :: acc(r.mark, max((r == root ? write : none), (r
  in s_x ? write : none)))
requires root != null ==> (root in s_x)
requires forall r: Ref :: { (r in s_x),r.left } (r in s_x) && r != null
  && perm(r.left) > none && r.left != null ==> (r.left in s_x)
requires forall r: Ref :: { (r in s_x),r.parent } (r in s_x) && r != null
   && perm(r.parent) > none && r.parent != null ==> (r.parent in s_x)
requires forall r: Ref :: { (r in s_x),r.right } (r in s_x) && r != null
  && perm(r.right) > none && r.right != null ==> (r.right in s_x)
{
  var x: Ref := root
  var y: Ref
  x.mark := true
  if (x.left == null && x.right == null) {
    y := x.parent
  } else {
    y := x.left
    x.left := x.right
    x.right := x.parent
    x.parent := y
  }
  x := y
  while (x != null)
  invariant forall r: Ref :: acc(r.parent, (r in s_x ? write : none))
  invariant forall r: Ref :: acc(r.right, (r in s_x ? write : none))
  invariant forall r: Ref :: acc(r.left, (r in s_x ? write : none))
  invariant forall r: Ref :: acc(r.mark, (r in s_x ? write : none))
  invariant x != null ==> (x in s_x)
  invariant forall r: Ref :: { (r in s_x),r.parent } (r in s_x) && r !=
    null && perm(r.parent) > none && r.parent != null ==> (r.parent in
    s_x)
  invariant forall r: Ref :: { (r in s_x),r.right } (r in s_x) && r !=
    null && perm(r.right) > none && r.right != null ==> (r.right in s_x)
  invariant forall r: Ref :: { (r in s_x),r.left } (r in s_x) && r !=
    null && perm(r.left) > none && r.left != null ==> (r.left in s_x)
  {
    x.mark := true
    if (x.left == null && x.right == null) {
      y := x.parent
    } else {
      y := x.left
      x.left := x.right
      x.right := x.parent
      x.parent := y
    }
    x := y
  }
}
```

Listing 4.5: Morris' tree traversal algorithm in Viper with inferred specifications.

4.5 Write to Single Index

The method `writeSingleIndex` in Listing 4.6 writes to a single location inside a loop. The index expression is `i + j` where `i` and `j` are both changing variables. However, because we use the polyhedra abstract domain for the numerical analysis, we are able to infer the loop invariant `i + j == bound`. The generated specification then correctly requires read permission to index `bound` of the array. This example demonstrates that the resulting formula from quantifier elimination already expands a lot even for simplest examples. Note that we added by hand two additional methods `inferredImpliesHuman` and `humanImpliesInferred`. With these methods, we test whether or not the contract inferred by our algorithm implies the human-written contract and vice versa, respectively, by inserting one of the assertions as precondition and the other as postcondition. A similar approach was also used by [24]. For this example the human contract is just `acc(loc(arr, bound).val)`. Both methods verify successfully which means that the human contract implies our inferred specification and vice versa. In other words, our inferred contract is both sound and precise with respect to the human specification.

```
function p(q: Int, bound: Int): Perm
ensures (bound >= 1 && ((q != bound || result == write) && (q == bound ||
   result == none)) || (bound >= 1 && ((q != bound || result == write) &&
   (q == bound || result == none)) || q + 1 == bound && bound >= 1 &&
  result == none || q == bound && bound >= 1 && result == write) || q + 1
   == bound && bound >= 1 && result == none || q == bound && bound >= 1
  && result == write) && ((bound < 1 || (q != bound || result >= write)
  && (q == bound || result >= none)) && ((bound < 1 || (q != bound ||
  result >= write) && (q == bound || result >= none)) && (q != bound ||
  bound < 1 || result >= write) && (q + 1 != bound || bound < 1 || result
   >= none)) && (q != bound || bound < 1 || result >= write) && (q + 1 !=
   bound || bound < 1 || result >= none))

method writeSingleIndex(arr: Array, bound: Int)
requires (forall q: Int :: acc(loc(arr, q).val, p(q, bound)))
{
  var i: Int := 0
  var j: Int := bound
  while (i < bound)
  invariant (forall q: Int :: acc(loc(arr, q).val, p(q, bound)))
  invariant i + j == bound && i >= 0
  {
    loc(arr, i + j).val := i
    i := i + 1
    j := j - 1
  }
}

method humanImpliesInferred(arr: Array, bound: Int)
requires acc(loc(arr, bound).val)
ensures forall q: Int :: acc(loc(arr, q).val, (q == bound ? write : none)
  )
{
}

method inferredImpliesHuman(arr: Array, bound: Int)
requires forall q: Int :: acc(loc(arr, q).val, p(q, bound))
ensures acc(loc(arr, bound).val)
{
}
```

Listing 4.6: A loop example where the only index which is written to is `bound`.

4.6 Array Maximum

The method `maxArray` in Listing 4.7 finds the maximum element by iterating two variables `x` and `y`. This example is taken from [24]. Again, the inferred specification is precise. However, it is not very difficult for a human to come up with a precise specification, given that he uses the same variable `rdAmount` to denote read permission. Refer to Listing A.1 in the appendix for the full specification of the function `p`.

```
method maxArray(arr: Array, lenA: Int, rdAmount: Perm) returns (x: Int)
requires none < rdAmount && rdAmount < write
requires forall q: Int :: acc(loc(arr, q).val, p(q, lenA, rdAmount))
{
  var y: Int
  if (lenA <= 0) {
    x := -1
  } else {
    x := 0
    y := lenA - 1
    while (x != y)
    invariant forall q: Int :: acc(loc(arr, q).val, p(q, lenA, rdAmount))
    invariant lenA >= 1 + y && y >= x && x >= 0
    {
      if (loc(arr, x).val <= loc(arr, y).val) {
        x := x + 1
      } else {
        y := y - 1
      }
    }
  }
}

method humanImpliesInferred(arr: Array, lenA: Int, rdAmount: Perm)
requires none < rdAmount && rdAmount < write
requires forall q: Int :: q >= 0 && q < lenA ==> acc(loc(arr, q).val,
  rdAmount)
ensures forall q: Int :: acc(loc(arr, q).val, p(q, lenA, rdAmount))
{
}

method inferredImpliesHuman(arr: Array, lenA: Int, rdAmount: Perm)
requires none < rdAmount && rdAmount < write
requires forall q: Int :: acc(loc(arr, q).val, p(q, lenA, rdAmount))
ensures forall q: Int :: q >= 0 && q < lenA ==> acc(loc(arr, q).val,
  rdAmount)
{
}
```

Listing 4.7: An algorithm to find the maximum element inside an array.

4.7 Array and List

Listing 4.8 shows an example which demonstrates that our algorithm is able to handle programs containing arrays as well as graph-like data structures. Note that our single restriction is that each field is only accessed by one type of receivers. Here, the `val` field as usual is accessed on array locations and the `next` field on the list nodes.

```
function p(q: Int): Perm
ensures ((q != 0 || result == write) && (q == 0 || result == none) || q
  <= 8 && q >= -1 && result == none || q <= 9 && q >= 0 && result ==
  write) && ((q != 0 || result >= write) && (q == 0 || result >= none) &&
   (q > 9 || q < 0 || result >= write) && (q > 8 || q < -1 || result >=
  none))

method allInOne(arr: Array, head: Ref, rdAmount: Perm, s_x: Set[Ref])
requires none < rdAmount && rdAmount < write
requires forall r: Ref :: acc(r.next, ((r in s_x) ? rdAmount : none))
requires forall q: Int :: acc(loc(arr, q).val, p(q))
requires head != null ==> (head in s_x)
requires forall r: Ref :: { (r in s_x),r.next } (r in s_x) && r != null
  && perm(r.next) > none && r.next != null ==> (r.next in s_x)
{
  var i: Int := 0
  var x: Ref := head
  while (i < 10 && x != null)
  invariant forall r: Ref :: acc(r.next, ((r in s_x) ? rdAmount : none))
  invariant forall q: Int :: acc(loc(arr, q).val, p(q))
  invariant x != null ==> (x in s_x)
  invariant forall r: Ref :: { (r in s_x),r.next } (r in s_x) && r !=
    null && perm(r.next) > none && r.next != null ==> (r.next in s_x)
  invariant i >= 0
  {
    loc(arr, i).val := i
    x := x.next
    i := i + 1
  }
}
```

Listing 4.8: A program containing arrays as well as lists.

4.8 Inhale, Write and Exhale in a Loop

The method `inhaleWriteExhale` in Listing 4.9 traverses an array and on every iteration first inhales **`write`** permission to index `i`, then writes to the same index and finally exhales **`write`** permission again. For this program, we would expect the inferred specification to require no permission at all. Indeed, the permission returned by `p` is **`none`** for all arguments. The human contract would in this case be **`true`** (or no contract at all as **`true`** is the default). Refer to

Listing A.2 in the appendix for the full specification of the function `p`. It can easily be seen that the postcondition of `p` could be simplified to just `result == none`.

```
method inhaleWriteExhale(arr: Array)
requires (forall q: Int :: acc(loc(arr, q).val, p(q)))
{
  var i: Int := 0
  while (i < 10)
  invariant (forall q: Int :: acc(loc(arr, q).val, p(q)))
  invariant i >= 0
  {
    inhale acc(loc(arr, i).val, write)
    loc(arr, i).val := i
    exhale acc(loc(arr, i).val, write)
    i := i + 1
  }
}

method humanImpliesInferred(arr: Array)
requires true
ensures forall q: Int :: acc(loc(arr, q).val, p(q))
{
}

method inferredImpliesHuman(arr: Array)
requires forall q: Int :: acc(loc(arr, q).val, p(q))
ensures true
{
}
```

Listing 4.9: An inhale followed by a write and an exhale to the same location.

4.9 Access every second Index

Another array example can be seen in Listing 4.10. What is interesting here is that the condition inside the loop is `i % 2 == 0`. Since formulas of this form can be handled by our quantifier elimination, we expect the inferred contract to be rather precise. This is indeed the case as it only requires `write` permission for even indexes and requires only read permission to odd indexes. Unfortunately, the postcondition of `q` in this case spans about 2 pages which is why we did not include its definition in the appendix. However, we added an additional method `preciseImpliesInferred` which specificies the desired amount precisely and compares this specification to the inferred one and indeed, this method verifies which means that our inferred contract is precise. Also, we checked again the human specification and in this case, the human specification implies the inferred one but not vice versa. That means, that the inferred specification is more precise. To come up with a contract that is as precise as our inferred one, a human would have to write a specification similar to `inferredImpliesPrecise` which requires some effort.

```
method secondIndex(arr: Array, rdAmount: Perm) returns (res: Int)
requires none < rdAmount && rdAmount < write
requires (forall q: Int :: acc(loc(arr, q).val, p(q, rdAmount)))
{
  var i: Int := 0
  while (i < 10)
  invariant (forall q: Int :: acc(loc(arr, q).val, p(q, rdAmount)))
  invariant i >= 0
  {
    if (i % 2 == 0) {
      loc(arr, i).val := i
    } else {
      res := loc(arr, i).val
    }
    i := i + 1
  }
}

method preciseImpliesInferred(arr: Array, rdAmount: Perm)
requires none < rdAmount && rdAmount < write
requires forall q: Int :: acc(loc(arr, q).val, (0 <= q && q <= 9 ? (q % 2
   == 0 ? write : rdAmount) : none))
ensures forall q: Int :: acc(loc(arr, q).val, p(q, rdAmount))
{
}

method humanImpliesInferred(arr: Array, rdAmount: Perm)
requires none < rdAmount && rdAmount < write
requires forall q: Int :: 0 <= q && q <= 9 ==> acc(loc(arr, q).val,write)
ensures forall q: Int :: acc(loc(arr, q).val, p(q, rdAmount))
{
}
```

Listing 4.10: An array traversal which performs a write to every even index and a read to every odd index.

Note, that with a slight modification (see Listing 4.11), our inference suffers from imprecision caused by the numerical abstract interpretation. If we omit the conditional in the loop body and instead just increase i by 2 in every iteration, we are no longer able to determine that only every second index gets accessed because the invariant we get from the polyhedra analysis will not reflect this fact.

```
function p(q: Int): Perm
ensures ((q != 0 || result == write) && (q == 0 || result == none) || q
  <= 8 && q >= -1 && result == none || q <= 9 && q >= 0 && result ==
  write) && ((q != 0 || result >= write) && (q == 0 || result >= none) &&
   (q > 9 || q < 0 || result >= write) && (q > 8 || q < -1 || result >=
  none))


method secondIndex(arr: Array) returns (res: Int)
requires (forall q: Int :: acc(loc(arr, q).val, p(q)))
{
  var i: Int
  i := 0
  while (i < 10)
  invariant (forall q: Int :: acc(loc(arr, q).val, p(q)))
  invariant i >= 0
  {
    loc(arr, i).val := i
    i := i + 2
  }
}
```

Listing 4.11: An array traversal which performs just one write to every even index.

5 Future Work

In this thesis, we addressed many aspects of the automatic inference of quantified permissions for general programs. However, the presented approach is far from being a complete and final specification inference. There is room for improvement and we will discuss the most important points in this section.

5.1 Missing Implementations

There are some Viper language features we currently do not handle. Implementing support for those constructs would further generalize our approach and extend the range of different programs we can handle.

5.1.1 Quantified Inhales or Exhales

In Viper, it is possible to inhale from or exhale to multiple heap locations in one statement using **forall** in **inhale** or **exhale** statements as in Listing 5.1. We do not currently handle statements of this form. However, [24] provided support for these constructs and implementing support for quantified inhales and exhales in our algorithm would not require much effort as well.

```
method quantifiedExhale(arr: Array)
requires forall i: Int :: 0 <= i && i <= 10 ==> acc(loc(arr, i).val)
{
  exhale forall i: Int :: 0 <= i && i <= 10 ==> acc(loc(arr, i).val)
}
```

Listing 5.1: A method containing a quantified exhale.

5.1.2 Gaining and Losing Permissions In Loops

As discussed in Section 3.9.2, our algorithm cannot track changing permissions in loops. A question that might arise is whether programs that gain or lose permissions over loop iterations could not be as well modeled by quantified inhales or exhales (cf. Section 5.1.1). But of course, general programs may change the permission amount during loops. Therefore, it would be desirable to track those changing permissions in the future.

5.1.3 Tracking changing Domain-typed Expressions

The handling of changing domain values would further generalize our algorithm. Domain values such as arrays can not occur as receivers in field accesses directly but indirectly through field accesses that have function as a receiver, such as the `loc` function. In our examples, we always assume that domain-typed values do not change their value throughout the program. However, we expect implementing support for the tracking of changing domain expressions to be straightforward as they can be handled similarly to references. We would mainly have to add support for receiver sets to track domain values in addition to references.

5.2 More precise Sets

Due to lack of time, we only implemented a very basic and rather imprecise widening operator for the abstract interpretation over receiver sets (cf. Section 3.3.2). This in turn leads to imprecision in the generated sets. We have seen in Chapter 4 that for many interesting examples, this imprecision does not have a big impact. However, the goal is to be as precise as possible which is why devising a more precise widening operator is definitively worth investigating.
Another fact that reduces precision of our sets is that we only include conditions into our receiver set definitions that talk about `null` (cf. Section 3.6.3). Listing 5.2 shows an example where the loop condition contains a second conjunct besides `x != null`, namely `x.val > 5`. For this example, we could include this condition as well into the specification of `s_x`. However, this is not the case in general when the condition depends on a changing variable. It could be worth investigating further under which circumstances we are allowed to incorporate such conditions into the specifications. Developing a sound rule which incorporates conditions into specifications could have a big impact on precision.

```
method unhandledCondition(head: Ref, rdAmount: Perm, s_x: Set[Ref])
requires none < rdAmount && rdAmount < write
requires head != null ==> head in s_x
requires forall r: Ref :: acc(r.next, (r in s_x) ? rdAmount : none)
requires (forall r: Ref :: { (r in s_x),r.next } (r in s_x) && r != null
  && perm(r.next) > none && r.next != null ==> (r.next in s_x))
{
  var x: Ref := head
  while (x != null && x.val > 5)
  // Invariant omitted here
  {
    x := x.next
  }
}
```

Listing 5.2: A loop with a condition we currently do not handle. The inferred contract will therefore be imprecise.

5.3 Improve Precision of Integer Analysis

There are many ways how to improve the precision of the numerical analysis. An interesting approach could be to come up with a different, even more precise abstract domain than the polyhedra abstract domain, although we have to find a good balance between precision and a longer runtime of the algorithm. Another possible direction worth investigating could be the combination of a numerical analysis with a heap analysis to also track numerical information on the heap. There is already research about corresponding approaches [11].

5.4 Nested Quantifiers

Currently, Viper does not support quantification over more than one variable if the quantified assertion contains the `acc` predicate. This in turn means that we cannot handle functions with multiple changing integer-typed arguments (cf. Section 3.9.3). As soon as Viper provides support for nested quantifiers in quantified permissions, the algorithm can be further generalized to support functions with arbitrarily many arguments.

5.5 Inferring Postconditions

As discussed in Section 3.7, inferring postconditions would help to increase precision. An under-approximating abstract interpretation would be beneficial to infer postconditions for loops. Concepts for an under-approximating numerical backward analysis by abstract interpretation have already been presented in recent work [16]. However, to the best of our knowledge, there does not yet exist a similar approach for a forward analysis. Future research in this direction could pave the way for a sound and precise forward analysis to infer postconditions.

5.6 Inter-procedural Analysis

General programs contain methods that call other methods. In our algorithm, we analyzed methods isolated from each other and we did not allow method calls. In other words, we conducted a purely intra-procedural analysis. It would be interesting to extend the algorithm by also analyzing call sites. We expect the task of matching preconditions of a callee from the call site to be rather tricky, considering the fact that we introduce helper sets that would have to be specified accordingly in the caller. However, this would increase usability of the overall approach a lot. Clearly, this extension is strongly connected to the inference of postconditions (cf. Section 5.5) as we would also have to be able to return permissions from the callee.

5.7 Reducing Formula Expansion

In our implementation of quantifier elimination, there are mainly two sources of formula expansion. First, the quantifier elimination procedure itself, in particular step 4.2 (cf. Section 3.5.5) where new disjuncts are produced depending on δ and the B set which in turn are both determined by the integers occurring in the formula. This can be avoided at least for a certain subset of formulas as described in [6]. The second source is our procedure of rewriting permission trees to the actual permission expression (cf. Section 3.5.4). Recall the resulting formula which expresses a permission tree t was

$$\begin{aligned}&\exists v_1, ..., v_n.\left(\mathrm{INV}(v_1, ..., v_n) \wedge \bigwedge_{(c,p)\in\mathrm{rewrite}_q(t)} \big(c \implies \underline{\mathrm{p}}(q) = p\big)\right) \wedge \\ &\forall v_1, ..., v_n.\left(\mathrm{INV}(v_1, ..., v_n) \implies \bigwedge_{(c,p)\in\mathrm{rewrite}_q(t)} \big(c \implies \underline{\mathrm{p}}(q) \geq p\big)\right)\end{aligned} \tag{5.1}$$

Note that in the first conjunct, $\underline{\mathrm{p}}(q)$ only occurs in equalities while in the second conjunct, $\underline{\mathrm{p}}(q)$ only occurs in inequalities. Thus, we can interpret the first conjunct as a specification of a concrete permission amount of $\underline{\mathrm{p}}(q)$ and the second conjunct as giving lower bounds for $\underline{\mathrm{p}}(q)$, i.e., expressing the *least* value of $\underline{\mathrm{p}}(q)$. As we are only interested in inferring *at least* the permissions that are required to successfully verify a program, we could just omit the first conjunct. This reduces the complexity of the resulting formula significantly. On the other side, we lose precision because then we do not have any information that $\underline{\mathrm{p}}(q)$ actually *is* one of the permission amounts. Here, we have to balance between precision and complexity of inferred contracts. Finally, more advanced formula simplifications could be included into the algorithm which would further reduce the effect of formula expansion after the quantifier elimination (cf. Section 3.5.7).

6 Conclusion

In this work, we presented an algorithm that can infer permissions for general Viper programs using a backward program analysis. We proposed an approach that extended and generalized two pre-existing separate algorithms to a unified framework which can handle graph-like data structures and arrays likewise. In our approach, permissions are collected using weakest precondition rules, and possible receivers of field accesses are over-approximated using abstract interpretation. A sound rule for propagating permissions over heap modifications has been presented. Furthermore, we explained in detail how to soundly incorporate the results of a numerical analysis into our algorithm. We also described basic ideas for the development of a forward analysis to infer postconditions. This turned out to be more delicate than initially expected and we therefore did not further develop this approach in detail.
Our evaluation on real programs in Chapter 4 shows that our algorithm is indeed able to infer specifications soundly and precisely for many interesting examples. Thanks to our various built-in simplifications, the generated contracts can be made conciser and more readable to some extent.
On the other side, we also showed that there are still a lot of opportunities for further enhancements. The presented approach could in future be extended to handle even more general programs such as programs containing method calls or quantified inhales and exhales. More advanced optimizations, especially with respect to quantifier elimination, could help to reduce the complexity of the inferred contracts. Developing a more precise widening operator could help to improve overall precision of the algorithm.

Bibliography

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[2] S. Blom and M. Huisman. The vercors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[3] Chair of Programming Methodology, ETH Zurich. Chair homepage. `http://www.pm.inf.ethz.ch`. [Online; accessed 18-March-2017].

[4] Chair of Programming Methodology, ETH Zurich. Sample project page. `http://www.pm.inf.ethz.ch/research/sample.html`. [Online; accessed 7-March-2017].

[5] Chair of Programming Methodology, ETH Zurich. Viper project page. `http://www.pm.inf.ethz.ch/research/viper.html`. [Online; accessed 7-March-2017].

[6] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(91-99):300, 1972.

[7] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 302–321. Springer, 2014.

[12] S. Gulwani and M. Musuvathi. Cover algorithms and their combination. In *European Symposium on Programming*, pages 193–207. Springer, 2008.

[13] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009.

[14] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[15] A. Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[16] A. Miné. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Science of Computer Programming*, 93:154–182, 2014.

[17] T. Mkrtchyan. Quantifier elimination. `https://people.mpi-sws.org/~piskac/teaching/decpro-ws12/slides/quantifierElimination.pdf`. [Online; accessed 13-February-2017].

[18] J. M. Morris. Traversing binary trees simply and cheaply. *Information Processing Letters*, 9(5):197–200, 1979.

[19] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *International Conference on Computer Aided Verification*, pages 405–425. Springer, 2016.

[20] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[21] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Compte-Rendus du I Congres des Mathematiciens des pays Slavs.*, pages 92–101, 1929.

[22] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[23] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

[24] S. Walter. Automatic Inference of Quantified Permissions by Abstract Interpretation. Master thesis, ETH Zürich, 2016.

A Appendix

A.1 Implementation Overview

In the following we will give a brief overview over the most important classes in our algorithm. For each class we shortly describe what it does and to which part of the algorithm it corresponds.

`Context`
Various helper functions and storing context of current program, methods, variables etc.

`Main`
Main class. Contains algorithm work-flow as well as the logic for program extension. This is where the conversion from our inference results to Viper specifications takes place.

`NumericalAnalysis`
Contains everything related to the numerical analysis in the polyhedra abstract domain (cf. Section 3.5.1).

`PermissionTree`
Represents permission trees and contains the implementation of the nodes and leaves described in Section 3.2.1

`QuantifiedPermissionsParameters`
An object defining various boolean flags which influence our analysis as well as the types for the numerical analysis. At the moment, only polyhedra is functional but other integer abstract domains can be easily plugged in.

`QuantifiedPermissionsState`
Represents the abstract state used for the over-approximation of receiver sets (cf. Section 3.3).

`QuantifierElimination`
Encapsulates the quantifier elimination algorithm, one method for each step

`SetDescription`
Represents the concept of receiver description sets and contains specializations for both reference description sets as well as integer sets.

`Utils`
Various utils and helper functions that do not fit anywhere else such as expression simplifications (cf. Section 3.8) and the injectivity check (cf. Section 3.5.2).

A.2 Specification of Permission Functions

A.2.1 Array Maximum

```
function p(q: Int, lenA: Int, rdAmount: Perm): Perm
ensures (lenA >= 3 + q && q >= -1 && result == none || lenA >= 2 + q && q
   >= 0 && result == rdAmount || lenA >= 2 && ((q == 0 || q == 1 ||
  result == none) && (q != 0 || q != 1 || result == rdAmount) && (q != 0
  || q == 1 || result == rdAmount) && (q == 1 || q == 0 || result == none
  ) && (q != 1 || q != 0 || result == rdAmount) && (q != 1 || q == 0 ||
  result == rdAmount)) || lenA >= 1 + q && q >= 1 && result == rdAmount
  || lenA >= 2 + q && q >= 0 && result == rdAmount || q >= 1 && lenA >= 1
   + q && ((q != 0 || result == rdAmount) && (q == 0 || result ==
  rdAmount)) || (q >= 0 && lenA >= 2 + q && ((q == 0 || result == none)
  && (q != 0 || result == rdAmount) && (q == 0 || result == none)) ||
  lenA >= 2 + q && q >= 0 && result == rdAmount)) && ((lenA < 3 + q || q
  < -1 || result >= none) && (lenA < 2 + q || q < 0 || result >= rdAmount
  ) && (lenA < 2 || (q == 0 || q == 1 || result >= none) && (q != 0 || q
  != 1 || result >= rdAmount) && (q != 0 || q == 1 || result >= rdAmount)
   && (q == 1 || q == 0 || result >= none) && (q != 1 || q != 0 || result
   >= rdAmount) && (q != 1 || q == 0 || result >= rdAmount)) && (lenA < 1
   + q || q < 1 || result >= rdAmount) && (lenA < 2 + q || q < 0 ||
  result >= rdAmount) && ((q < 0 || lenA < 2 + q || (q == 0 || result >=
  none) && (q != 0 || result >= rdAmount) && (q == 0 || result >= none))
  && (lenA < 2 + q || q < 0 || result >= rdAmount)) && (q < 1 || lenA < 1
   + q || (q != 0 || result >= rdAmount) && (q == 0 || result >= rdAmount
  )))
```

Listing A.1: The specification of the permission function for Listing 4.7.

A.2.2 Inhale, Write and Exhale in a Loop

```
function p(q: Int): Perm
ensures ((q != 0 || (q != 0 || q == 0) || result == none) && (q != 0 || result == none) && (q != 0 || q == 0 || result == none) && (q != 0 || (q != 0 || q == 0) || result == none) && (q != 0 || result == none) && (q == 0 || result == none) && (q == 0 || (q != 0 || q == 0) || result == write) && (q == 0 || q != 0 || result == write) && (q == 0 || (q != 0 || q == 0) || result == write) && (q == 0 || q != 0 || result == write) || q <= 8 && q >= -1 && result == none || q <= 9 && q >= 0 && result == none) && ((q != 0 || (q != 0 || q == 0) || result >= none) && (q != 0 || result >= none) && (q != 0 || q == 0 || result >= none) && (q != 0 || (q != 0 || q == 0) || result >= none) && (q != 0 || result >= none) && (q == 0 || result >= none) && (q == 0 || (q != 0 || q == 0) || result >= write) && (q == 0 || q != 0 || result >= write) && (q == 0 || (q != 0 || q == 0) || result >= write) && (q == 0 || q != 0 || result >= write) && (q > 9 || q < 0 || result >= none) && (q > 8 || q < -1 || result >= none))
```

Listing A.2: The specification of the permission function for Listing 4.9

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Inference of Pointwise Specifications for Heap Manipulating Programs

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

Name(s):	**First name(s):**
Münger	Severin

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

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

Place, date	**Signature(s)**
Zürich, 20.3.2017	[illegible]

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