Automatic Inference of Quantified Permissions by Abstract Interpretation

Automatic Inference of Quantified Permissions by Abstract Interpretation

Master Thesis

By

Seraiah Walter

ETH zürich

Departement of Computer Science
Chair of Programming Methodology

Supervisors:
Prof. Dr. Peter Müller
Dr. Alexander J. Summers, Dr. Caterina Urban

August 2016

Abstract

Viper, a software verification infrastructure, employs permission logics to efficiently reason about a program. Permission logics, such as separation logic, require the programmer to specify access permissions to individual memory locations. In this thesis we present two different algorithms to automatically infer access permission to each memory location the program accesses. The first targets arrays by tracking indices and through a set of weakest precondition rules infers a sound and precise method specification. The second targets other data structures, such as linked lists, trees and graphs. This algorithm collects accesses in different sets and uses Viper's quantified permission construct to define all permissions needed. In this thesis we present the rules which our algorithms implement to infer all the permissions needed. Our evaluation shows that inferring precise contracts can be very tricky to do by hand. For example indices that are accessed might depend on multiple input parameters. For the programmer it is usually hard to quickly see the relations between all variables. On the other hand the proposed algorithm is able to quickly infer sound and also quite precise contracts.

Acknowledgements

I would like to express my gratitude to both my supervisors Dr. Alexander J. Summers and Dr. Caterina Urban for their great support and very interesting discussions throughout the project. Furthermore, I want to thank Jérôme Dohrau for participating in our meetings and sharing the experiences he had with his own project.
Many thanks also to Prof. Dr. Peter Müller for giving me the opportunity to work on this project in his group. Finally, a special thanks to my family and friends for their continued support throughout this period.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

In today's world software is ubiquitous. This makes errors in programs an even more serious problem. A famous example is NASA's Mars Climate Orbiter: on its mission to Mars in 1998 it was sent too low into Mars' atmosphere, hence the associated stresses crippled its communications and the spacecraft has been lost somewhere in space without control. This was due to a software bug which failed to make a simple conversion from Imperial to metric units. An embarrassing lapse that cost $125 million. Sadly, humans make mistakes. In order to ensure good software quality most software companies focus on testing, which is able to dramatically reduce the number of bugs in a program, but it is not able to guarantee the absence of errors. Hence researchers in the field of software verification are working on automated mathematical proofs to ensure the correctness of a program. This gives us stronger guarantees that the program is doing what it is supposed to do. In order to understand what a program is supposed to do, software verification relies on precise mathematical specification and assertions. Therefore, proving correctness of a program demands a lot more effort from programmers or the people in charge of writing the specification. This is probably the biggest obstacle and the reason why many software developers still solely rely on testing rather than also use verification.

Nevertheless, in the last couple of decades researchers have made astonishing progress in the field of software verification. One great achievement was the introduction of separation logic (SL) [14] and other permission-based logics, e.g. implicit dynamic frames [15]. SL is an extension of Hoare logic [10]. Through the use of explicit access permissions it is able to introduce the ability of local reasoning, i.e. framing, by isolating the part of the memory on which a program operates, so that the rest of the memory becomes irrelevant for the proof of the program. In addition it is also able to prove whether an assertion is affected by a heap modification or not. This makes the formal verification of programs modular and achieves better scalability.

Today's programming languages such as C or Java introduce many sources of potential bugs which cannot be handled very well by formal verification without framing. Permission-based verification tools are very strong in handling data structures such as lists, doubly-linked lists, trees, etc., since it can focus on the part of the heap that is affected by the data structure and ignore the rest of the heap. Secondly they also handle pointer bugs such as null pointer dereferences or undesired aliasing very well. On the down side, permission logics such as SL also introduce the new overhead of specifying which memory location are used in the program. Those logics only allow to read or write to a memory location if the program has enough permissions to the accessed memory location at the corresponding program point. Hence the programmer needs to annotate the program with the required access permissions. (cf. Section 2.2).

Another great achievement in the research of software verification was the introduction of verification infrastructures such as Boogie [2] and Why [8]. The development of common architectures for program verification tools simplified the development of new verifiers by allowing researchers and developers to focus their work on a single level of abstraction, without having to consider the whole pipeline from high-level source code, over intermediate representations, to eventually solving decision problems for logical formulas using SMT-solvers.

Viper [13], developed at ETH Zurich, is a verification infrastructure, which was designed with permissions in mind. Consequently it allows for the development of permission based verifiers. All our work in this thesis is part of the Viper project. Viper is further discussed in Chapter 2.

1.1 Goal of this Thesis

As previously mentioned, the introduction of permissions significantly increases the annotation overhead. The goal of this thesis is to reduce such overhead in order to make program verification more practical. In Viper, a program is only allowed to read or write to a memory location if at that program point it has sufficient permissions for that location. Hence, in the method's precondition, the programmer has to declare enough permissions for every memory access occurring in the method. While working through the program Viper keeps track of the state of all permissions. If Viper encounters a memory read or write to which the current state does not have enough permission it will report an error. We aim to automatically infer the needed permissions in order to let the programmer focus on more interesting specifications such as functional properties.

Inferring permissions can get complicated, especially if we consider complex data structures, in which case we have to handle not only single memory locations but also sets of memory locations, which may be unbounded. For such cases Viper has the *quantified permissions* feature (cf. Section 2.2.2). For example a linked list would be defined in terms of a set. The set contains all the nodes that are part of the list. One would normally annotate the program with the permissions needed to access the list by quantifying over this set. In some methods not all locations of the data structure are accessed. To precisely extract the memory locations that do get accessed

we take advantage of Sample [1] (cf.Section 2.4), which is a static analyzer based on Abstract Interpretation [5]. Hence the main goal is to develop an analysis which automatically infers quantified permissions with the help of abstract interpretation. In Viper those permissions need to be defined in the method's precondition, invariants and postcondition. Hence we transform the inferred permissions into a shape suitable for the method's contracts. The effect of permissions in the context of contracts is further explained in Section 2.2.2

- **Arrays.** We aim to infer contracts, preconditions and invariants, for methods that manipulate arrays. A property of arrays is that the memory location corresponding to each slot is uniquely defined by the array index. This facilitates our analysis, since to identify all accesses in a program it is enough to keep track of the accessed indices.
- **Graphs.** A second goal is to infer permissions for more complex data structures such as lists, trees and graphs. The main difference to arrays is that those data structures usually are accessed recursively, which introduces more complexity.
- **Soundness and Precision.** Many static analysis tools are sound, meaning that if they prove a property it is guaranteed to hold; on the other hand if they cannot prove a property it might still hold, since analyzers are typically incomplete. In our analysis we strive to stay sound and infer contracts that provide enough permissions for each memory location that the program needs to read or write to during its execution. Soundness is explained in more detail in Section 2.4.1. On the other hand, we try to be as precise as possible. Therefore we want to minimize the number of false positives: permissions to locations that in reality are not accessed by the program. We aim to infer the weakest precondition possible.
- **Fractional Permissions.** Since Viper supports fractional permissions we also consider fractions of permissions in our analysis. Different permission amounts are used for read and write actions (cf. Section 2.2). Therefore we have to distinguish between read and write permissions. Requiring only fractional permissions whenever possible makes the inferred contracts more precise.
- **Nested Quantifiers.** Currently there is work in progress to support nested quantifiers. We introduce quantifiers in Section 2.2.2. Even though nested quantifiers are not fully supported yet in Viper we theoretically discuss how this would impact our analysis and how we could take advantage of it.

In the next Chapter we further explain Viper and Sample. Then we introduce an algorithm to infer permissions for methods that mainly deal with arrays (Chapter 3) and an algorithm that works with graph data structures (Chapter 4). In Chapter 5 we discuss some implementation details and then present and evaluate the inferred specifications. We consider the advantages

and disadvantages of the proposed algorithms and draw some conclusions in the beginning of Chapter 6 and we look into how one could further improve the analysis by discussing some ideas for future work at the end of that chapter.

CHAPTER 2

VIPER AND SAMPLE

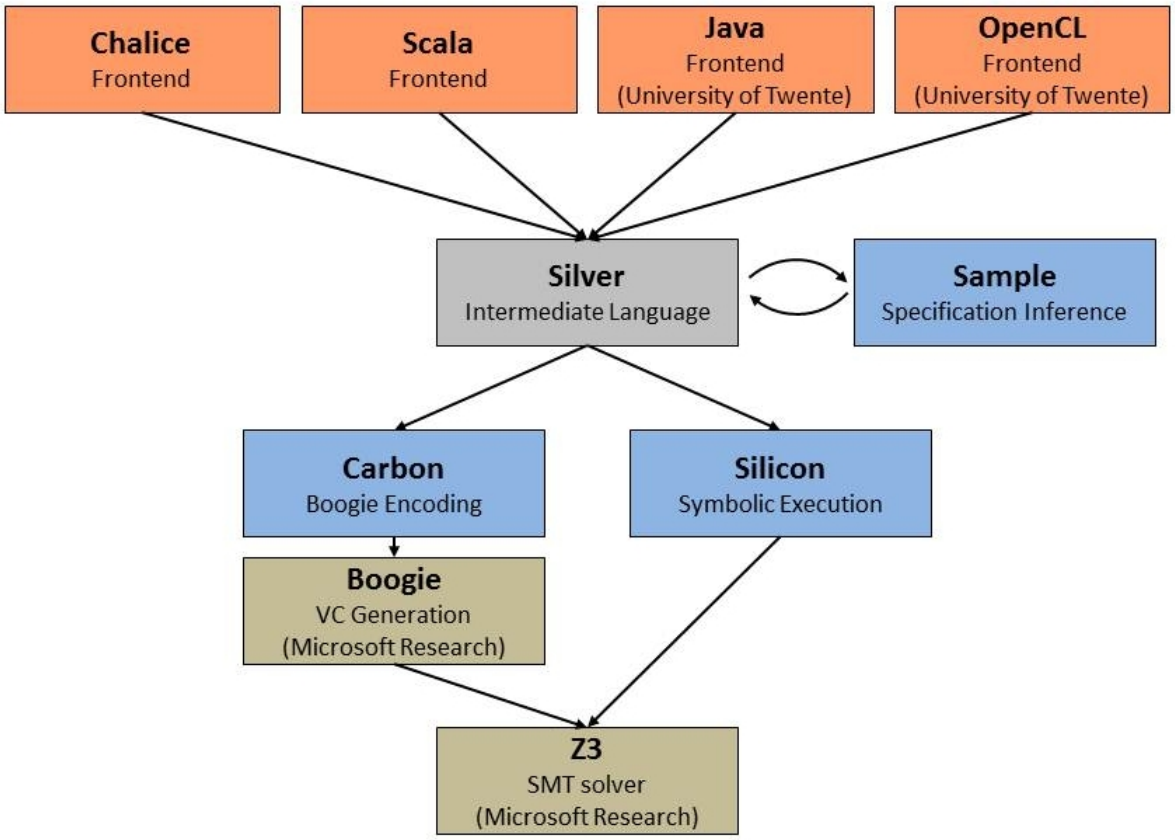
Viper is a new verification infrastructure. In this chapter we explain how Viper is designed and how its modules interact with each other. Furthermore, we discuss some of the technical details that are important to understand our work.
Most of our work is done in Sample, which is a part of Viper. We explain the basic principles and again discuss some technical details.

2.1 Viper Infrastructure

The structure of Viper is depicted in Figure 2.1. As we can see Viper consists of different modules: at the core it has its own intermediate language called Silver: a sequential, imperative and object-based language. It consists of fields, methods, functions, predicates, and custom domains. In a program they are all defined globally. There is no notion of classes; every object has every field declared in the program, and methods and functions have no implicit receiver. This intermediate language has the goal to be expressive enough to easily handle a wide variety of front-end languages. On the other hand it has to be simple enough for the verifiers to efficiently analyse the code.

The advantage of having this intermediate language is that, in order to use Viper to prove a property of a program written in an existing programming language, researchers or developers simply have to write a compiler which translates native code to Silver. Some such front-end tools already exist: at the moment, Viper has compilers for Chalice [11], Scala, Java [3] and OpenCL [3]. A translation for a subset of the Python language is being developed. This is the front-end. At the back-end the Viper infrastructure allows for the development of verifiers. Currently, there are two verifiers: Carbon, which is based on Boogie, and Silicon, which uses symbolic execution. Both rely on Microsoft's SMT solver Z3 [7].

FIGURE 2.1. Viper Infrastructure



An important feature of Viper is that, in contrast to Boogie and Why, it natively supports constructs for permissions (cf. Section 2.2), including fractional permissions and counting permissions. Hence, it can support more powerful front-end verifiers that take advantage of permission based logics, such as separation logic. This greatly facilitates reasoning for modern day concurrent programs.

Another very important part of Viper is the static analyzer Sample; its goal is to infer specifications in order the help back-ends verify the program. We further explain Sample in Section 2.4.

2.2 Permissions

In Viper there is a special type `Perm` for permission amounts. Full permission is denoted as `write` and zero permission is expressed through `none`. Whenever the context makes it clear that we are talking about permissions we write `1` or `0` for full or no permission respectively, to save space.

Since the reasoning in Viper is based on permissions, the language has a number of differ-

ent constructs to describe them. The most fundamental contract is the accessibility predicate `acc(e.f, p)`, which represents permission to a single field location: the field `f` of the reference `e`. The optional second parameter `p` defines the permission amount. The default value for the second argument is `write`, i.e. full permission. Full permission is required for writing to a location and partial permission, i.e. any amount greater than `none`, is required for reading it. With zero permission one is neither allowed to read nor write to the associated field. The permission amount should always be between zero and full permission. Having a permission amount greater than full permission or smaller than zero permission, i.e. negative permission, makes the accessibility predicate equivalent to false.

In order to check whether there is enough permission to read or write to a location, Viper keeps track of the state of the permission at each program point. It starts with adding all permission mentioned in the method's precondition to the state. Then Viper propagates the state through each path of the method. This way, at each program point Viper can check if the state holds enough permissions to either read or write to a field location.

In practice an accessibility predicate always appears in a certain context, e.g. precondition, postcondition, invariants, exhale or inhale (cf. Section 2.2.1). Depending on the context it is used in the effect on the permission state is different. For example in a precondition the accessibility predicate means that the method requires the caller of the method to actually have the denoted permissions to the field location and passing it to the callee. Hence from the callers perspective the mentioned permissions are removed from the permissions state and from the callee perspective they are added to the permission state. The reverse holds for the postcondition: the method passes permissions back to the caller.

2.2.1 Exhale and Inhale

In the context of permissions `exhale` and `inhale` statements also play a central role. They essentially are the permission-aware analogous of assert and assume. Indeed, if we exhale a *pure* assertion P — a predicate which does not contain any permission related information — the `exhale` statement acts the same as an assert P and inhaling a pure predicate P would be equivalent to assuming P. On the other hand if the predicate P contains an accessibility predicate then exhale P means assert P and in addition also give away the mentioned permissions. Similar for inhale: here we assume P and get all permissions specified in P.
With these constructs we can simulate concurrent programs. For example if a method wants to fork off a thread executing a method, then it would simply need to exhale the method's precondition. Consequently it will pass all the permissions mentioned in the precondition to the new thread. A join can be simulated through an inhale of the postcondition. For more information see [11].

Exhaling and inhaling permissions actually has some strong implications: for example if we are left with no permission to a location then we also have to forget the value of that location.

The reason for this behavior becomes clear in a multithreading example: if one thread exhales all permissions to a location another thread could inhale them and might potentially end up having write permission to the location. Hence it would be allowed to change its value.

2.2.2 Quantified Permissions

With quantifiers one can express that a certain predicate holds not only for a concrete instance but also for a group of elements. This is especially important to have when dealing with unbounded sets of elements.

One of Viper's advantages compared to other verification infrastructures comes with the support for quantified permissions. Hence in Viper it is possible to define permission requirements for multiple elements, potentially even an unbounded number of elements at once. At the time of writing Viper restricts quantified permissions to the following structure, but there is work in progress to relaxing it:

$$\textbf{forall } q : T :: c(q) ==> \textbf{acc(}\ e(q).f\ \textbf{,}\ p(q)\ \textbf{)}$$

In this formula *c(q)* is a boolean expression, *e(q)* a reference-typed expression and *p(q)* an expression denoting a permission amount where each expression might depend on the quantified variable q. In a precondition this means we can quantify over any type `T`, where under some condition *c(q)* we require *p(q)* permission amount to the field f of any reference *e(q)*, where the expression e has to be injective. This injectivity rule has the consequence that e has to depend on the quantified variable.

2.3 Arrays

In order to make the implementation of back-ends more practical Viper tries to keep a small syntax. Therefore Viper natively only supports the types `Int, Bool, Ref, Perm, Set[T]` and `Seq[T]`. Arrays are not natively supported. There is, however, an easy way to simulate arrays with a *custom domain*. Custom domains is a feature of Viper's language which enables the programmer to specify its own type. A custom domain consists of functions and axioms, i.e. properties of the domain. The default implementation of integer arrays is depicted in Figure 2.2. With the statement `loc(a,i).val` we access the i-th slot of the array `a`. The field `val` contains the actual data of the array. The type of the field corresponds to the type of the array. As one might expect, the `len` function refers to the length of the array. The two axioms help define the behavior of this new type in order to help Viper verify certain properties of the array. If we want to express a property of the array by quantify over each slot of the array `loc` would have to be injective, as explained in Section 2.2.2. The `all_diff` axiom guarantees the injectivity property for this function. It guarantees that there are no collisions, meaning for every index `i` and array `a` we get a different reference back. This is expressed in terms of these helper functions

FIGURE 2.2. Simulation of Arrays

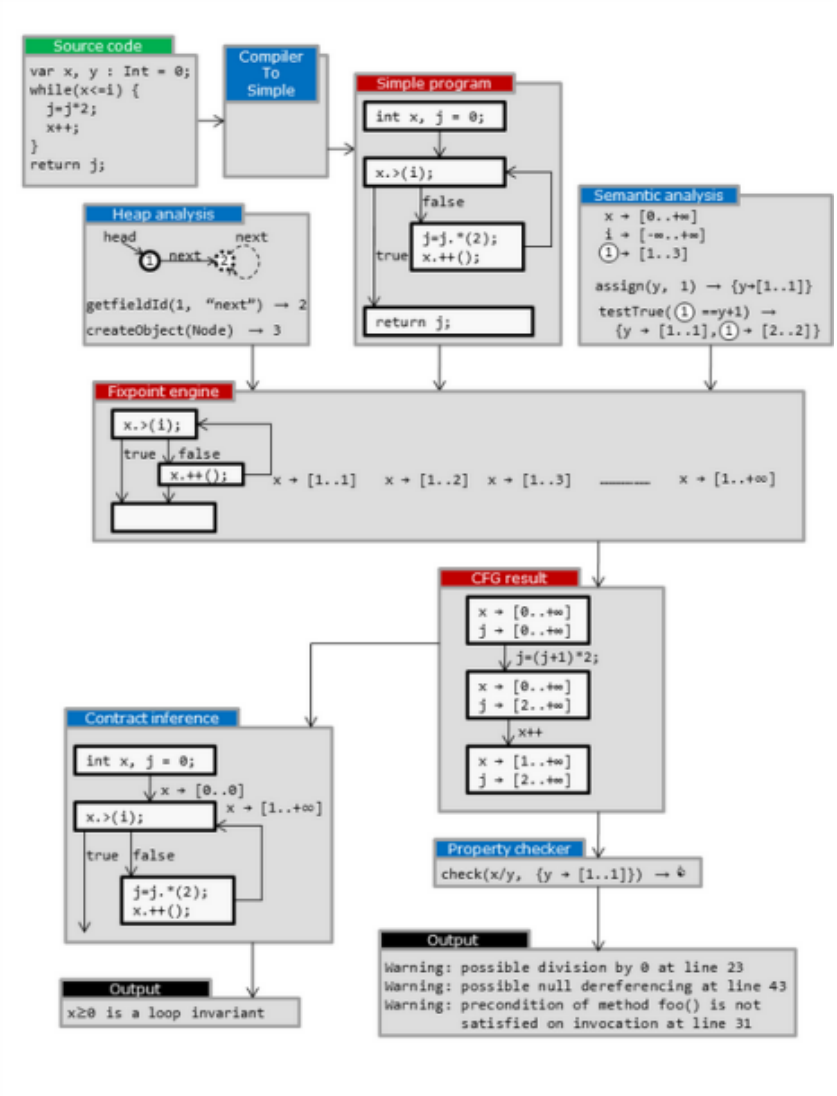
```
field val: Int
domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function first(r: Ref): Array
  function second(r: Ref): Int

  axiom all_diff {
    forall a: Array, i: Int :: {loc(a, i)}
      first(loc(a, i)) == a && second(loc(a, i)) == i
  }

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

`first` and `second`. They map the resulting reference from the function `loc` back to its input parameter `a` or `i` respectively. And since functions by their nature can only map to one single element, injectivity is guaranteed.

2.4 Sample

In Viper we are able to take advantage of Sample which stands for "Static Analyzer of Multiple Programming LanguagEs" [1]. It is is a generic static analyzer based on abstract interpretation. We explain abstract interpretation in more detail in Section 2.4.1. Sample can be used for strengthening existing specifications or inferring new ones. Figure 2.3 depicts the structure of Sample. The code (green) gets compiled into the internal language Simple, which is basically a control flow graph representation of the method. Then, through a fix-point computation and the help of other analysis modules (blue) certain properties can be inferred. There are different semantic domains such as numerical domains, strings, types, etc. For some, Sample already provides implementations: examples for numerical domains are the interval domain [4], octagon domain [12] and polyhedra domain [6]. We can see that there is also a contract inference module. This is where our work will take place. We extend the contract inference to be able to infer quantified permissions. In order to understand how Sample works we first need to briefly introduce Abstract Interpretation.

FIGURE 2.3. Structure of Sample



2.4.1 Abstract Interpretation

Abstract Interpretation (AI), formalized by the computer scientists Patrick Cousot and Radhia Cousot [5], is a theory for the sound approximation of the behavior of programs. We will briefly discuss the most important parts of AI.

Semantics. The semantics of a program is a mathematical characterization of all possible behaviors of the program when executed for all possible input data. The most precise semantics is called concrete semantics. It precisely describes the executions of a program in a concrete domain. The abstract interpretation of a program describes its executions in an *abstract domain*: a representation of a concrete domain that abstracts irrelevant properties for a given problem

away. Abstract semantics are needed because the concrete semantics is usually not computable. The abstract semantics approximates the concrete semantics.
There are many different abstract semantics depending on the property of interest. For example to guarantee that a certain program statement is never executed one has to track all possible execution paths. Or if possible return values of the method are of interest then it might be enough to solely track possible values of variables.

Fixed Point Computations. AI computes the abstract semantics through fixed point computations. The abstract domain is a lattice, where each node in the lattice describes one or more program executions. At the top of the lattice, the top state, is the set of all program executions, including executions that could never happen in practice, e.g. dead paths etc. At the bottom there is the empty set, which corresponds to no executions. Sample starts its computation with the bottom state and propagates it through the control flow graph of the program. With each iteration it will add more possible executions, behaviors that through a real execution can actually happen. Sample supports both forward and backwards iteration strategies. In addition it also allows for a refining iteration which is a special mix of both forward and backward iteration.
To reach a fixed point one might need to iterate for a very long time or it might not be possible at all. Hence AI computes an approximated post-fixpoint through the use of widening operators.

Widening. Widening is a mechanism that facilitates computation while potentially losing information. For example iterating through a loop might take very long or might not terminate at all. This is the case if the number of iterations depends on an input parameter. The values of input parameters are usually not statically known and hence the number of iterations are potentially unbounded. The widening operator overapproximates the behavior and might consider states of the program that would never be reached. Let us consider a concrete example:

```
1 method widening(){
2   var i:Int :=0
3   while(i<1000){
4     i := i+1
5   }}
```

Assume that we are interested in the values of `i`. As an abstract domain we might want to approximate the sets of possible values for a variable using intervals. The interval domain expresses the values of a variable in terms of an interval [x,y] where x is the lower bound and y the upper bound of all possible values the corresponding variable can assume. In the first iteration Sample reaches the state [0,1] after another iteration [0,2]. The widening operator will then try to extrapolate the change in future iterations. Hence it overapproximates the state to be $[0,\infty]$. This has the consequence that we now also included numbers that `i` will never contain in a real execution. Very often the loss of information is necessary to make the semantics computable. In general there is a trade off between precision and computability.

Soundness. In the context of abstract interpretation soundness means that we only over-approximate the semantics of the program and never underapproximate it. In particular, we always include all possible behaviors and maybe more, but never fewer. Concretely, if we work with a numerical domain the resulting semantics will never miss a possible value of a variable but might contain more than in a real execution.

In the context of our permission inference soundness means that we will always require at least permissions to all fields that actually get read or written to. Consequently if our analysis is sound we infer contracts that provide at least as many permissions that Viper requires to reason about the program.

CHAPTER 3

INFERRING QUANTIFIED PERMISSIONS FOR ARRAYS

In this chapter we explain the methods and rules we use to infer quantified permissions for accesses to an array. Each array slot that is accessed is uniquely defined by its index. We collect the indices and propagate the information upwards through the help of weakest precondition rules in order to lose as little information as possible. This is then used to generate method preconditions and loop invariants.

3.1 Tracking Accesses

To be able to explain how we infer the permissions for array accesses in a program we should look at the example in Figure 3.1. This method initializes an array by setting all its slots to 0. Here a human will quickly see that this method needs write access to the array `arr` from location 0 to 10. To begin with, we simplify the inference problem by assuming that there are only accesses to the

FIGURE 3.1. Motivating Array Example

```
1 method initArray(arr: Array)
2 requires forall p:Int :: 0<=p && p<=10 ==> acc(loc(arr,p).val)
3 {
4   var i:Int := 0
5   while(i<=10)
6   {
7     loc(arr,i).val := 0
8   }
9 }
```

FIGURE 3.2. Tracking

```
1 method tracking1(arr: Array, a: Int, b:Int, c:Int)
2 requires forall p:Int:: p==c+1||p==c+1||p==a||p==b ==> acc(loc(arr,p
     ).val)
3 {
4   loc(arr,a).val := loc(arr,b).val  //[p==c+1||p==c+1||p==a||p==b]
5   b:= c                             //[p==c+1||p==c+1]
6   a:= c+1                           //[p==c+1||p==b+1]
7   loc(arr,b+1).val := 0             //[p==a||p==b+1]
8   loc(arr,a).val := 0               //[p==a]
9 }
```

array and we do not consider other accesses that either access a field different from `val` or have a receiver node that is not part of the array, e.g. a reference passed as a method argument. We assume that there is only one array and that the implementation of the array corresponds to the standard implementation explained in Section 2.3. From this point on one can then generalize the approach for multiple fields and arrays. The general structure of a precondition that grants access permissions to an array looks like this:

$$\textbf{forall } p : Int :: c(p) ==> \textbf{acc(} \ loc(arr\ \textbf{,}\ p).val\ \textbf{)}$$

The expression *c(p)* might depend on *p* and describes all the indices of the array to which we grant access. It acts as a filter since we only require access permissions if the left hand side is true. For all *p*s that do not satisfy *c(p)* the right hand side is ignored.
Our main goal now is to analyse the method and "collect" all the array locations which get accessed — either through a read or a write — and then transform it into a logical formula for our *c*-expression. We do this by stepping backwards through the control flow graph of the program, which is done in Sample, and whenever we hit an array access we store the index in our collection. This approach is illustrated in Figure 3.2.

The comment on each line indicate how the collection of indices would be transformed into the logical formula *c(p)* after we evaluated the statement on that line. On line 8 the collection consists simply of the element `a`. Hence we only want to grant access to the location `a`. This is done by considering only `p`s that satisfy `p==a`. Quite interesting and at first a little unintuitive is that even though at line 7 of the program we require access to both locations `a` and `b+1` they are connected through an or (`||`). The reason is that this expression does not describe our collection directly but rather filters out all relevant `p`s. At the end the collection of `p`s that satisfy the expression should be equivalent to the collection of indices from our analysis. Hence it is correct to require access permission to the index `p` if `p` only satisfies either the first part `p==a` or the second part `p==b+1`.

FIGURE 3.3. Aliasing

```
1 method alias(a:Ref, b:Ref, arr:Array)
2 {
3   b.val := 5            //[p==a.val] or [p==5]
4   var i:Int := a.val    //[p==a.val]
5   loc(arr,i).val := 0   //[p==i]
6 }
```

As we can see in Figure 3.2 we start by putting the variable `a` into our collection. We then propagate our collection upwards and modify it depending on the statement that we hit. For example, on line 6 there is an assignment to variable `a`. Hence, if we want to propagate the information "we need access to location `a`" above this variable assignment while preserving its meaning, we need to replace `a` with the expression on the right. Here we introduce the extra restriction that the right expression cannot be heap dependent. Else we might end up with a heap dependent receiver. The problem with heap dependent receivers is explained in the next section.

3.1.1 Heap Dependent Receivers

In our rule for handling variable assignments we have the extra restriction that, if the variable is in our collection set, the right hand side cannot depend on the heap. At the moment we rely on the syntax to determine whether an expression in our collection changes or not: if an assignment to an expression in our collection happens we know that the value changes and we need to replace it with the new expression. Hence we can rely on the syntax. The problem with heap dependent expressions is that the value of the expression could also change if an assignment to a different expression happens. The reason for this problem is aliasing. As we can easily see in the example in Figure 3.3, if we collect the element `i` and later replace it with `a.val`, we cannot guarantee that the value of `a.val` stays the same as long as there is no assignment to `a.val`. If `a` and `b` actually do alias each other then the first assignment on line 3 would also change the value of `a.val`. To take this into consideration we would either need alias information from a heap analysis, which we do not have at the moment, or write a logical expression which takes both variants into account, which leads to very complex expressions. Hence we do not allow the expression on the right to be heap dependent.

3.2 Tracking Distinct Accesses

With the use of inhale and exhale a new level of complexity is introduced. Now, to infer a precise precondition it is not enough to guarantee permission to each occurring access in the program but we also have to consider permissions that are added to the permission state through inhale

FIGURE 3.4. Distinct Tracking

```
1 method tracking2(arr: Array, a: Int)
2 requires forall p:Int:: true ==>
3     acc(loc(arr,p).val, p==a?-1:0 + p==a?1:0 + p==1?1:0)
4 {
5   inhale acc(loc(arr,a).val,1)  //[(p==a,-1),(p==a, 1),(p==1, 1)]
6   exhale acc(loc(arr,a).val,1)  //[(p==a, 1),(p==1, 1)]
7   exhale acc(loc(arr,1).val,1)  //[(p==1, 1)]
8 }
```

and others that are removed through exhale.

If we inhale permissions to slot 0 and later read this slot, we will not need to require access permission to it in our precondition since we got it for free. On the other hand, if we exhale permissions to the same location twice we need to require twice the amount. In general we have to track the distinct accesses and "count" how much permission we need to each. This is illustrated in the example in Figure 3.4.

As before we extract the array location, but now in addition also the permission amount. Permission amounts in exhales we leave positive since we need to require them and permission amounts in inhales we negate since inhale adds permission to the permission state for free.

To take advantage of this "counting" mechanism we will have to push our inferred expressions into the permission amount, i.e. the second input parameter of the `acc`–predicate.

Viper supports ternary conditional operator: `c?a:b`. We use this form to write our precondition which says that for certain `p`s we will need +1 or -1 permission and then sum everything up. This way we let the verifier determine how much permission is required in total for each array location `p`. Our algorithm stays agnostic to the actual amount. In the example Viper now automatically determines that it does not require permissions to the position `a` since the permission amounts corresponding to the inhale and exhale cancel out. Staying agnostic to the permission amount also has the advantage that we can let Viper take care of aliases: if we modify the example by removing the first inhale then Viper would automatically determine if `a` is equal to 1 that we require the permission amount of 2, which of course is not possible, hence the precondition would result to false and if they do not alias it requires full permission to each location.

3.2.1 Keeping the Order

The analysis so far does not consider the order of inhales and exhales. If we use the same approach for the method `tracking3` in Figure 3.5 we can easily see that all the permission amounts would cancel out. This would leave us with a precondition equivalent to true, since we do not require permission to any location. This would be wrong, since we should still require access to location `a`. The problem is that we allow the inhale on line 8 to cancel out the earlier exhale.

FIGURE 3.5. Ordering

```
1  method tracking3(arr: Array, a: Int, b:Int)
2  requires forall p:Int:: true ==> acc(loc(arr,p).val,
3    f(p==a?-1:0 + p==a?1:0 + p==a?1:0 + f(p==a?-1:0)))
4  {
5    inhale acc(loc(arr,a).val) //(p==a,-1),(p==a,1),(p==a,1),(p==a,-1)
6    exhale acc(loc(arr,a).val) //(p==a, 1),(p==a, 1),(p==a,-1)
7    exhale acc(loc(arr,a).val) //(p==a, 1),(p==a,-1)
8    inhale acc(loc(arr,a).val) //(p==a,-1)
9  }
10 define f(x) x>0? x:0
```

This of course is not allowed: Viper cannot remove access permission from the permission state that it will get at a later point. In order to prevent this we introduce a "boundary"-function: `f(x) = x>0?x:0`. We can see that this function will round any negative amount to zero. In our bottom up analysis `f` will be introduced every time we deal with negative permission amounts, i.e. inhales, and it encloses all expressions that we already inferred, i.e. accesses occurring after the inhale. This is illustrated in the precondition of the method in Figure 3.5. In this way only positive amounts will be propagated further up the program state and negative amounts will contribute to anything further down the program. As an optimization we do not introduce the function `f` in between two inhales.

Expressions inside an inhale or exhale can be arbitrary complex. Our analysis handles simple expressions related to arrays either in the form without quantifiers (1) or with quantifiers (2). `exp` describes an arbitrary expression that returns an integer and `permAmount` can be an arbitrary expression which returns a permission.

```
(1) acc(loc(arr,exp).val, permAmount)
(2) forall p:Int:: c(p) ==> acc(loc(arr,p).val, permAmount)
```

3.2.2 Reads and Writes

The attentive reader might now wonder how we treat simple reads and writes. Indeed, since read and write statements do not actually modify the state of the permissions in the program neither the permission amount +1 nor -1, would be appropriate. When we evaluate a read or a write statement we actually want to check whether the necessary permission amount is already provided by the expression we collected so far, i.e. whether we accessed the location or exhaled permission to that location further down the program flow. If we already required enough permissions to the corresponding location we do not want to require more. On the other hand if it is not provided yet we want to at least require the permission amount used to perform the action, i.e. read or write. In order to express that in our logical formula we introduce a helper function

```
(1)rwMax(cond, permAmnt, oldExp) = max((cond ? permAmnt : 0),oldExp)
```

The condition `cond` refers to the array location we read or write to, `oldExp` refers to the permission amount we already collected so far and `permAmnt` refers to the permission amount that we require to perform the action. Through the use of this function we can now express that if `cond` is true, e.g. p==1, where 1 is the location we read or write to, we require at least `permAmnt` permission. But if `oldExp` already required more permission then we consider `oldExp`

In the case of a write the permission amount `permAmnt` has to be 1 since Viper requires full permission to write to a location. In the case of reads Viper only requires the permission amount to be greater than 0. There are different possibilities for actually picking a read permission amount. Here we will briefly discuss two: a fixed amount and a ghost parameter.

Fixed Amount. If we take a fixed amount it is clear that in order to read a location the amount has to be bigger than 0 and since we only want read permission and not write permission the amount also has to be smaller than 1. If we take any fixed amount, e.g. 1/10, then our specification does not correspond to the weakest precondition anymore. A caller now has to pass at least 1/10 permissions even though less would have been enough for reading this location.

Ghost Parameter. This is an artificially introduced parameter exactly for this purpose. The precondition of the method requires that this parameter, e.g. `rdAmount`, is greater than 0. The advantage is that the caller is free to pass any positive permission amount. In the case where the method itself has to pass permission to this location to another thread he can easily split the amount up by taking any fraction, e.g. `rdAmount/2`.

In our analysis we decided to use a ghost parameter to guarantee read permissions.

We briefly present two alternative ways to the handle read and write statements and explain the reasons why we abandoned them: the first alternative would be to interpret them as an exhale followed by an inhale. This approach requires the permission to a location only once even with multiple reads and writes. The reason is that the inhale always eliminates the exhale of the following read or write statement. The draw back is, as we will explain later in Chapter 3.4.1, introducing an extra inhale can lead to more imprecision.

The second alternative is to actually use a different function instead of the proposed `rwMax` function for write statements:

```
write(cond, oldExp) = cond ? write : oldExp
```

Since in Viper permission amount should always be between zero permission and full permission it seems unnecessary to take the `max(write,oldExp)` as we did in the previous version. However, in the example in Figure 3.6 we see that there actually is a problem with this alternative implementation. The problem arises if we write to a location, then exhale permission to the location and later read this location inside a loop, as depicted in Figure 3.6. Line 4 corresponds to the write permission needed for the write on line 8 and the specification on line 5 corresponds to

FIGURE 3.6. Alternative Write Function

```
1  method testWriteFunction(arr: Array, rdAmount: Perm)
2  requires rdAmount > none
3  requires forall p:Int :: true ==> acc(loc(arr,p).val,
4      p==0 ? write :
5          ((p==0? 1/2 : 0) + (p==0?rdAmount:0)) )
6  {
7    var a: Int
8    loc(arr,0).val := 0
9    exhale acc(loc(arr,0).val,1/2)
10   while(a<5)
11   invariant forall p:Int::true==>acc(loc(arr,p).val,p==0?rdAmount:0)
12   {
13     var a:= loc(arr,0).val
14   }
15 }
```

the `oldExp` that we collected previously. Viper is not able to verify this program because after the exhale on line 9 we are left with only 1/2 permission to location 0. Then it tries to verify the invariant which is not possible because `rdAmount` could be greater than 1/2. One way to deal with this is to add the extra restriction `rdAmount < 1/2` to the specification. Consequently we would need a way to identify this upper bound on `rdAmount`. On the other hand if we stick to our first implementation of the function and simply take the `max(write,oldExp)` Viper is able to verify the program. The reason is that in case that `rdAmount` happens to be greater than 1/2 then our precondition would require more than full permission which is equivalent to false (cf. Section 2.2). Hence instead of having the extra restriction `rdAmount < 1/2` explicitly, it is implicitly in our permission amount formula.

3.3 Conditionals

Having introduced the basics of how we track accesses in a program we next focus on how we propagate this information further up the control flow graph. In particular we explain what happens if we hit an `if`-statement with the example in Figure 3.7.

Before hitting the if-else statement our state consists of the access to location `v`. We propagate this information into both the true and the false branch. This happens naturally in the way Sample propagates states inside the control flow graph. Propagating the collected accesses into both branches is necessary since some of the access expression that happen after the if-else statement might depend on a variable that gets changed only in one branch. This is the case in our example. Only in the false branch does the variable `v` get changed. Hence the meaning of

FIGURE 3.7. Conditional Example

```
1  method conditional(arr:Array, b:Bool)
2  requires forall p:Int:: true ==> acc(loc(arr,p).val,
3    b? (p==0?1:0) : (p==1?1:0) )
4  {
5    var v:Int := 0       //[b?(0,1):(1,1)]
6    if(b){               //[b?(v,1):(1,1)]
7      //do nothing       //[(v,1)]
8    }else{
9      v := 1             //[(1,1)]
10   }
11   loc(arr,v).val := 0 //[(v,1)]
12 }
```

the access after the conditional statement is totally different depending on the path the program takes. In the precondition we have to consider both cases. Therefore we join the states again at the top of the `if`-statement through the ternary conditional operator where we reuse the `if`-condition. The result can be seen in the precondition of the method `conditional` in Figure 3.7. We only require permission to location 0 if b is true and otherwise we require permission to location 1.
In general if the condition contains reads then we also have to add those to our collection.

3.4 Loops

A while loop can be seen as a special conditional where the loop body is the true branch and everything after the loop is the false branch. The difference to a normal if-statement is that the true branch, i.e. the loop body, will be executed multiple times. In addition, a while loop in Viper needs an invariant. In our analysis it is enough if we collect accesses occurring inside the loop body only once. Hence no iteration is needed. In Figure 3.8 we show the result of what happens if we naively propagate the values upwards: we end up only requiring access to location 0 even though we iterate through the array and actually access all locations from 0 to 9. The problem is that x is modified within the loop. So we will not allow simply propagating variables that change inside the loop outwards. What we want is an expression which does describe the values of `x` but does not mention `x` itself.

3.4.1 Forget Operator

We need to transforms the expression `p==x` into an equivalent expression that does not mention `x` anymore but still talks about the same values `x` assumes during the loop execution. In the previous example this would be `p>=0&&p<10`. In order to do so we need information about the

FIGURE 3.8. Wrong Loop Specification

```
1  method loop(a:Array)
2  requires forall p:Int:: true ==> acc(loc(a,p).val, p==0?1:0)
3  {
4    var x:Int := 0       //[p==0]
5    while(x<10)
6    invariant forall p:Int:: true ==> acc(loc(a,p).val, p==x?1:0)
7    {
8      loc(a,x).val:=0  //[p==x]
9      x:=x+1
10   }
11 }
```

values of x, which we get from a separate numerical analysis. We call this operator forget:

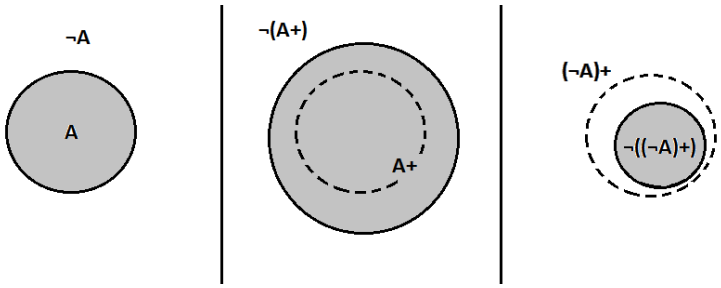
$$\textbf{forget}(A\ ,\ Vars\ ,\ numInf\ ) = A'$$

The forget operator takes a formula A and a set of changing variables *Vars* and with the help of information from the numerical analysis *numInf* tries to find the strongest formula A' such that A implies A' and such that A' does not contain any variables from *Vars*. In the context of array indices this means that the indices expressed by A are also expressed by A'. But due to imprecision in the numerical analysis A' might also express indices that are not expressed by A
For the numerical analysis we need a numerical domain. Sample already supports a number of numerical domains; in our case we wanted the most precise domain available, hence we decided to use the polyhedra domain. The numerical analysis provides us with an invariant that expresses all values changing variables can assume. The numerical domain itself already incorporates a forget operator. What we do is taking the invariant from the numerical domain and enriching it with the loop condition. To do so we first need to transform the loop condition into a suitable shape for the polyhedra domain and then add it to the state, through the use of its assume operation. Similarly we add the information from the assertion A. Then we use the forget operator from the numerical domain to eliminate all variables occurring in *Vars*. We then transform the resulting information back into a shape suitable for our indices collection.
In our example the numerical analysis will give us as very precise information about the variable x, i.e. x>=0&&x<10. We can put the pieces together:

numInf:	x>=0 && x<10
A:	p==x
Vars:	{x}
A':	p>=0 && p<10

Here the resulting formula is precise but we have to keep in mind that the numerical analysis will in general overapproximate the values of x (cf. Section 2.4.1). This can lead to weaker

FIGURE 3.9. Forget operator demonstrated with sets: to the left we have set of locations A and the negation of it ¬A. In the middle is the state of the sets after the forget operator. To the right we have the overapproximation of ¬A.



formulas: in our example we know the value of x at the beginning of the loop is between 0 and 9. Due to imprecision in the numerical analysis this might be overapproximated to the range 0 to 10. For our analysis this actually means that sometimes we might requires access permissions to too many locations. This is ok since our goal is to collect at least those locations that do get accessed.

On the other hand if we overapproximate an inhale we would then assume to already have accesses that in reality we do not have. In order to prevent this from happening we propose to use the forget operator in the following way:

$$\neg\textbf{forget}(\neg A\ ,\ Vars\ ,\ numInf\ ) = A'$$

Let us look at it in terms of sets (see Figure 3.9): the assertion A talks about certain array locations that we access. The corresponding set (A) now contains exactly those array locations. Through the use of the forget operator the set described by the assertion A was overapproximated, meaning the set (A+) now contains more array locations than those that were actually accessed. This is ok for exhales, reads and writes, but not for inhales. To handle inhales we use negations.

Taking the negation of that set (A) will lead to a set (¬A) that contains all possible locations except those that we just had previously. Through the use of **forget** we enlarge this set ((¬A)+), hence the set now includes all possible locations except a few from our original set. At the end we take the negation again. This will lead to an underapproximation, meaning the set (¬((¬A)+)) now contains less or equal locations than A. In practice this underapproximation is often quite significant meaning we end up with the empty set. The reason for this is that we rely on the forget operator from the numerical domain and our numerical domain does not handle negations very well. The formula `p!=x` will be transformed into `p>x || p<x` and because the polyhedra domain cannot handle disjunctions we forget each part separately. This is illustrated here:

numInf:	x>=0 && x<10
A:	p==x
¬*A*:	p>x || p<x
Vars:	{x}
forget:	p>0 || p<9
A':	p<=0 && p>=9 (false)

On the other hand if we inhale a range of indices, e.g. `p>=x && p<=20`, assuming the same numerical information, this would result in the formula `p>=9 && p<=20`:

numInf:	x>=0 && x<10
A:	p>=x && p<=20
¬*A*:	p<x || p>20
Vars:	{x}
forget:	p<9 || p>20
A':	p>=9 && p<=20

The problem of imprecision when forgetting a formula from an inhale is the reason why we do not treat simple reads and write as an exhale followed by an inhale.

3.4.2 Forget if Conditions

We also include branch conditions in our collection as explained in Section 3.3 and since we are not allowed to propagate changing variable outside the loop we also have to take care of changing variables occurring in if conditions. If the condition of an if statement occurring inside a loop contains a changing variable then we have to assume that both branches are executed. For example if we have the if condition `x<5` and x iterates from 0 to 10 we have to consider both branches of the if statement. Hence we introduce the artificial condition "forgotten". It indicates that the condition contained a changing variable and that we have to consider both branches since both will be executed. Later when transforming the collection into an expression we transform conditions that are not forgotten to `cond?accTrue:accFalse` where `cond` corresponds to the if condition and `accTrue` corresponds to the expression that we get from the true branch and `accFalse` corresponds to the expression from the false branch. In case of a forgotten branch condition we take the maximum of both branches: `max(accTrue, accFalse)`.

3.4.3 Invariants

Now that we know how to track accesses and how to transform them in order to propagate them outside the loop body we will continue by discussing how we generate the invariants.

Invariants are a little bit more delicate than preconditions. At first the invariant has to hold on entry, meaning we have to also infer a precondition which enables the verifier to prove that the invariant holds on entry. Plus the invariant then also needs to be preserved by the loop body. Viper will try to prove this as usual: assuming the invariant was true in the last iteration, is it true that it still holds after an extra iteration? This is one reason why we have to forget about changing variables. If we were to only assume that we had access to exactly one location `x` then it would not be possible to prove, if `x` is incremented by 1, that we now have access to this new `x`. Another issue that we have to take care of is the need for non-permission invariants. Let us go back to our loop example in Figure 3.8 and assume we have the correct invariant:

```
forall p:Int:: true ==> acc(loc(a,p).val, p>=0&&p<10?1:0)
```

This would not be enough for Viper to guarantee that we are allowed to access the array at position `x` on line 8. The problem is that Viper does not know anything about the values of x. Thanks to our numerical domain we already know a good invariant about the values of `x` and hence we just reuse it in our specification. The previously mentioned invariant plus the invariant from the numerical domain, `x>=0&&x<=10`, are enough for Viper to prove that we have permissions to all array locations that the program accesses.

Invariants also have a second special property: they serve not only as a precondition for the loop body, but also as a postcondition for everything after the loop. This is how Viper treats a while-loop with loop condition `b` and invariant `inv` after it verified the loop body and the invariant:

```
1 //before loop
2 exhale inv
3 //havoc written vars
4 inhale inv && !b
5 //after loop
```

As we see Viper inhales the invariant in the state after the loop. This behavior is especially important in cases where the loop changes the state of the permissions, either by adding new permission when, for example, we create new objects, or by removing permissions. Since adding new permissions happens by inhaling them and as we explained in Section 3.4.1 we cannot always handle inhale inside a loop precisely, it is hard to get new permissions from inside the loop outwards. This is a limitation of our current analysis and we will discuss how future work can address this issue in Section 6.1.3

3.5 Derivation Rules

In this section we will summarize the weakest precondition rules used to implement the analysis explained above. Applying the rules to a method body will result in a logical formula `A`. In our specification we will then have to plug this formula into the permission amount parameter of the accessibility predicate like this:

```
forall p:Int:: true==>acc(loc(arr,p).val, A)
```

3.5.1 Extract Accesses

First we need a way to extract the accessed array locations. At the moment we can extract array locations from normal reads and writes, from simple inhales and exhales and finally from inhales and exhales that use quantifiers, if it is in the supported form (cf. Section 3.2.1).

getRWAccess(*stmt* **,** *oldForm***)** =

$$\begin{cases} rwMax(p == i, write, oldForm), & \text{if } stmt \text{ matches } loc(arr,i).val := n \\ rwMax(p == i, rdAmount, oldForm), & \text{if } stmt \text{ contains read to } loc(arr,i).val \\ 0, & \text{else} \end{cases}$$

getExhaleAccess(*exp*) =

$$\begin{cases} p == i ? permAmnt : 0, & \text{if } exp \text{ matches } acc(loc(arr,i).val,\ permAmnt) \\ c(p) ? permAmnt : 0, & \text{if } exp \text{ matches } forall\ p{:}Int{::}\ c(p) \texttt{ ==> } acc(loc(arr,p).val,\ permAmnt) \\ 0, & \text{else} \end{cases}$$

getInhaleAccess(*exp*) =

$$\begin{cases} p == i ? -1 * permAmnt : 0, & \text{if } exp \text{ matches } acc(loc(arr,i).val,\ permAmnt) \\ c(p) ? -1 * permAmnt : 0, & \text{if } exp \text{ matches } forall\ p{:}Int{::}\ c(p) \texttt{ ==> } acc(loc(arr,p).val,\ permAmnt) \\ 0, & \text{else} \end{cases}$$

In our bottom up analysis we walk backwards through the control flow graph. In the next sections we will explain how we apply the introduced functions depending on what statement we hit. We start out with a formula that corresponds to zero permission, i.e. *none* and with each statement enrich it with new permission amounts. The formula that we inferred so far is denoted as *A* and the resulting formula as *A'*

3.5.2 Variable Assignment (`var := right`)

In the case of a variable assignment we first have to extract all accesses in the expression `right` and add them to the formula A. Then in the resulting formula we have to replace all occurrences of `var` with `right`:

$$A' = \textit{getRWAccess(right,A)}, \text{replace all occurrences of } \texttt{var} \text{ with } \texttt{right}$$

3.5.3 Field Assignment (`fieldAssign := right`)

In our analysis we assume that the field assignment is a write to the `val` field of the array. If this is the case we simply have to add the write as defined in the *getRWAccess* function. *stmt* refers to the field assignment.

$$A' = \textit{getRWAccess(stmt, A)}$$

3.5.4 Exhale (`exhale arrayExp`)

If we exhale permission to array locations and the expression is in a supported form then we simply have to extract the array locations.

$$A' = \textit{getExhaleAccess(arrayExp)} + A$$

3.5.5 Inhale (`inhale arrayExp`)

Again if `arrayExp` is in one of the supported forms we can simply extract the location with the defined function *getInhaleAccess()*. Additionally we add the boundary function f. Note that this rule does not show the optimization of omitting the boundary function when dealing with consecutive inhales.

$$A' = f(\textit{getInhaleAccess(arrayExp)} + A)$$

3.5.6 Conditional (`if(b) {S1} else {S2}`)

In the case of a conditional we apply the weakest precondition rules (wp) on each branch separately and we connect the results through Viper's ternary conditional operator. Additionally we also add all accesses from the branch condition `b`

$$A' = \textit{getRWAccess}(b, b ? wp(\texttt{S1}, A) : wp(\texttt{S2}, A))$$

3.5.7 Loop (`while(b) {S}`)

First we infer the loop invariant. We have to apply the wp rules for all statements inside the loop body. To infer the invariant we do not consider statements after the loop. Hence we start again

with zero permission. The function *forget* corresponds to the forget operator explained in Section 3.4.1.

$$inv = forget(getRWAccess(b, wp(S, 0)))$$

For the precondition we rely on the invariant inv from the previous rule. First we simulate the exhale by taking the positive permission amounts in inv and then do an inhale by taking the negation of inv.

$$A' = inv + f(-1 * (inv) + A)$$

CHAPTER 4

INFERRING QUANTIFIED PERMISSIONS FOR GRAPH DATA STRUCTURES

So far we explained how to infer permissions for arrays. The goal is now to also create an analysis which handles other data structures such as linked lists, trees and graphs in general. Since our algorithm handles general graphs we will call the analysis graph analysis. In this chapter we are going to motivate our approach by looking at different examples and discussing which parts of the array analysis we can reuse and which ones we cannot. We will again explain how we handle each programming construct, i.e. assignments, if-statement, loops etc.

4.1 Inference

We start the discussion by looking at an example program and its specification. Figure 4.1 shows a method which traverses a linked list by following the next field. The provided specification is written by a human. The contract can be split into three different parts: the first part requires field accesses for every reference in the set `nodes`. The second part defines the set `nodes` by recursively adding the next node to the set. The third part, which can be seen on line 9, is the "anchor" of the set, which defines a concrete element, here `nd`. Usually the anchor is the starting node of the data structure, i.e. the head of the list or the root of the tree.

As before, the analysis will collect all the receivers of a field access. In the array analysis the receiver was uniquely defined by the index. Now we will track the whole receiver expression: in our example on line 17 it is simply `node`. Again, if we hit a variable assignment, we need to replace all occurrences of this variable with the right-hand side. When dealing with graph-like

FIGURE 4.1. Motivating List Example

```
1  field next: Ref
2  define INV(nodes)
3         !(null in nodes)
4      && (forall n: Ref :: n in nodes ==> acc(n.next))
5      && (forall n: Ref :: n in nodes && n.next != null
6                                              ==> n.next in nodes)
7
8  method traverse(nodes: Set[Ref], nd: Ref)
9      requires nd in nodes
10     requires INV(nodes)
11 {
12     var node :Ref := nd
13     while(node != null)
14     invariant INV(nodes)
15     invariant node != null ==> node in nodes
16     {
17         node := node.next
18     }
19 }
```

data structures the restriction that the right-hand side cannot be heap dependent is too strong, since every graph-algorithm has to de-reference a field, e.g. `next`, to get to another node. Hence we end up having a field read where the receiver node itself depends on a field read. To ensure the aliasing problem will not arise we assume that the algorithm does not modify the data structure, except for some modifications, i.e. writes to locations that do not affect our state that we collected. Concretely, if the algorithm traverses the data structure through a field, e.g. `next`, we do not allow a write to this field. On the other hand writes, e.g. to an integer field `data`, which are independent and do not affect other accesses are allowed.

4.2 Heap Dependent Receivers

Another problem that arises when we allow heap dependent receivers is how to handle chained field reads, e.g. `nd.next.next`. If we apply the same algorithm from the array approach to the method `getThird` in Figure 4.2 we end up with the specification on line 2. As we can see in the comment of line 5 one of the receiver nodes is `nd.next`, this means we effectively access `nd.next.next`. The derived specification would actually give us the permissions to this field. But the problem is that the formula is not self-framing: to be able to talk about the second part of the specification, which contains `nd.next`, we first need permissions to field `next` of node `nd`. But we only get permission to it in the same precondition.

FIGURE 4.2. Field Accesses in a Row

```
1 method getThird(nodes: Set[Ref], nd:Ref) returns (third:Ref)
2 requires forall p:Ref:: true ==>
3   acc(p.next, (p==nd?1:0) + (p==nd.next?1:0))
4 {
5     var first: Ref := nd              //[nd, nd.next]
6     var second:Ref := first.next      //[first, first.next]
7     third := second.next              //[second]
8 }
```

There are multiple ways to handle heap dependent receivers. We will discuss two: splitting the accesses into multiple contracts and introducing sets.
With the first approach we have to split the specification into multiple separate preconditions, where every field access with paths of length 1 will get into the first and everything with path of length 2 into the second and so on. The first precondition does not rely on any other access permissions, since they are all of length 1. The second precondition only needs permissions from the first precondition, and so on. Unfortunately this has some negative consequences. First, we do not take care of aliases anymore, hence a node occurring in the first precondition might also be part of the second precondition and we would effectively require permission to the same field twice. The second problem arising with this approach is that we lose the order of accesses of different path length, since they are now in different predicates. In the array analysis we use the order to guarantee that inhales do not affect earlier exhales. This cannot be guaranteed with this approach.

The second approach and probably a more natural way to write the specification is to introduce a set that contains all the receiver nodes. The analysis has to collect all the receiver nodes of accesses and the specification then requires that they are a part of that set. By quantifying over this set we require all the needed access permissions. The advantage is that the set automatically takes care of duplicates and aliases. The disadvantage is that here we also lose the order of accesses and hence cannot handle exhales and inhales.
This approach is what we use in our analysis. As we saw also in Figure 4.1 data structures like lists, trees and graphs are usually defined in terms of sets. The specification would then look like the following:

```
1 requires forall p:Ref:: p in nodes ==> acc(p.next)
2 requires nd in nodes && nd.next in nodes
```

We put all the receiver nodes into our set `nodes` and require access to the `next` field for each element in that set. In order to specify the set `nodes` on line 2 it is very important that we

FIGURE 4.3. Multiple Fields

```
1  method
2   travRightAndLeft(left_nodes:Set[Ref], right_nodes:Set[Ref], nd:Ref)
3  requires forall r:Ref:: r in left_nodes ==> acc(r.left)
4  requires forall r:Ref:: r in right_nodes ==> acc(r.right)
5  requires nd in right_nodes && nd.right in right_nodes
6  requires nd in left_nodes && nd.left in left_nodes
7  {
8      var nL :Ref := nd.left//L[nd, nd.left]  R[.]
9      nL := nL.left          //L[nL]           R[.]
10
11     var nR:Ref := nd.right//R[nd, nd.right]
12     nR:= nR.right          //R[nR]
13 }
```

require the permissions on line 1 first. This enables us to talk about `nd.next`. Also the order in which we mention the receiver nodes is important: to be able to talk about `nd.next` Viper first needs access to the field, consequently it first needs to know that `nd` is in `nodes`.

4.3 Multiple Fields

When dealing with graph data structures it is often the case that multiple fields are needed. For example if we implement a graph where each node has three outgoing edges we would require three fields `e1`, `e2` and `e3`. Each of those fields point to another node in the graph. Since graphs could have aribtrary many outgoing edges we generalize our algorithm to handle any number of fields.

Analogously to what we did in the previous example we introduce a new set for each field: each set contains all the receiver nodes from which we access the corresponding field. As before, to require permission we quantify over each set individually. The method `traverseRightAndLeft` in Figure 4.3 shows such an example. The method traverses the tree by following the left field two levels down the tree and then the right field for two levels. We again apply a bottom up approach where we collect all accesses. The `R` in the comment on each line denotes the set containing all receiver nodes which access the right field and the `L` collects all receiver nodes that access the left field. In the example we can see that we do not access the left and right field of all nodes.

Many graphs actually have the property that each node potentially has accesses to all fields. For example in a simple tree traversal each node accesses both its left and right field. In those cases it would make sense to merge the different sets into a single one. But this is not the case in general. Imagine a list where each node in the list points to a value node, which stores some data. In this case only the value nodes need access to the data fields, whereas only the list nodes need

FIGURE 4.4. Conditionals

```
1  method getThird(nodes: Set[Ref], nd:Ref) returns (third:Ref)
2  requires forall p:Ref:: p in nodes ==> acc(p.next)
3  requires nd in nodes && ((nd.next!=null)?
4                 (nd in nodes  && nd.next in nodes) : true) {
5
6                                      //N[nd,(nd.next!=null)?
7    var first: Ref := nd              // {nd, nd.next}:{}]
8
9                                      //N[first,(first.next!=null)?
10   if(first.next!=null){             // {first, first.next}:{}]
11
12      var second:Ref := first.next //N[first, first.next]
13      third := second.next          //N[second]
14   }
15 }
```

access to the `next` field. Therefore we need two separate sets, one describing all the nodes of the list and the other describing all the value nodes.

4.4 Conditionals

Similarly to our array analysis we take branch conditions into account in our analysis in order to get a more precise formula. Figure 4.4 shows an alternative implementation of the method `getThird` which adds an extra if statement to check whether the second node actually exists or not. The comments on each line illustrate the way we collect accesses. The `N` set contains all receiver nodes of accesses to the field `next`. On line 10 we add the if condition into our collection. In the comments on line 9 we see the element `first`. This element was added due to the access in the if condition. The second element is the ternary conditional operator where the true part contains all accesses that occur inside the true branch and the opposite for the false part. When propagating this information further up we will end up with the set described on line 7 which will be transformed to the specification on line 3 and 4.
The difference to the previous specification is that in the case where `nd.next`, is null we do not require permission to its `next` field, i.e `nd.next.next`. In the specification of Figure 4.2 we always require permission to that field, which implied the existence of the second node, since requiring permission to a field location in Viper implicitly requires that the receiver is nonnull.
The implementation of `getThird` in Figure 4.4 is actually weaker than the previous implementation in Figure 4.2 due to the extra if statement, which makes an extra null check: `first.next!=null`. The first implementation simply accessed the `next` field of the second

FIGURE 4.5. Change of Variables

```
1  while(bw){
2    z.next := n     //not allowed
3    if(b)           //{Z->b?z.next:x.next,N->b?z.next.next:x.next.next}
4           {y:=z}   //{Z->z.next, N->z.next.next}
5    else   {y:=x}   //{Z->x.next, N->x.next.next}
6    n := y          //{Z->y.next, N->y.next.next}
7    n := n.next     //{Z->n.next, N->n.next.next}
8    z := n          //{Z->n, N->n.next}
9    n := z.next     //{Z->z, N->z.next}
10 }                 //{Z->z, N->n}
```

node without checking whether it is null, hence it assumed the second node to exists. The second implementation takes the special case of a list with only one node into consideration and does not access the second node if it is null. Hence in contrast to the first implementation we are able to call the method with a linked list containing only one node. To reflect this change in our specification we also need to including the if condition, i.e. the null check, into our specification. Consequently also our precondition will be weaker, meaning easier to satisfy.

4.5 Loops

The way loops are handled is very similar to the way we handle them in the array analysis. We treat a loop as a special if condition. Consequently, as before, we have to deal with the problem of changing variables. Hence we introduce a forget operator.

4.5.1 Forget Operator

Unfortunately we cannot simply reuse the forget operator from the array analysis. The reason is that before we dealt with integers and now with references. Before we could rely on the pre-analysis based on a numerical domain and now we would need a heap analysis to give us enough information about the changing variables.
To still get enough information about the changing variables in our bottom up analysis we also keep track of how a variable changes. The idea is to get an expression that precisely describes the value of the variable at the end of one loop iteration. Therefore we track how each statement changes the variable. This is depicted in Figure 4.5.
The expression `N->exp` denotes that the variable `n` at the end of the current loop iteration will be equivalent to the expression `exp` at the beginning of the loop iteration. At the bottom of the loop `n` did not change yet. Hence we initialize the information we know about the variable `n` with `N->n`. Then, while propagating this information upwards we keep updating the expression on

the right hand side of the arrow exactly the same way we update information about field accesses: for every variable assignment we replace all occurrences of the left hand side with the right hand side and we introduce conditionals for if statements. Assignments to a field, as it is the case on line 2, are not allowed. This statement actually modifies the data structure and we would again run into the problems of aliasing discussed before.
At the top, on line 3, we have a precise description of how the variable `n` changes. We now want to use this information to transform expressions in our collection into a new expression that does not mention changing variables anymore. In order to do so we introduce for every changing variable a new set, e.g. `n_nodes`, which describes all values that the variable `n` contains after a loop iteration and hence also at the beginning of the next loop iteration. We define the set like this:

```
forall _n:Ref:: (_n in n_nodes) ==> transition(_n) in n_nodes
```

The term `transition(_n)` corresponds to the change of the variable that we track. In our example in Figure 4.5 this is how the changing variable sets would be defined for the variables `n` and `z`:

```
forall _n:Ref, _z:Ref:: (_n in n_nodes) && (_z in z_nodes) ==>
              (b ? _z.next : x.next ) in z_nodes
        && (b ? _z.next.next : x.next.next) in n_nodes
```

As we see here we can now quantify over multiple variables. This is possible since we do not talk directly about permissions in this quantifier. The changing variable sets are defined recursively, hence we again have to define an "anchor". The anchor at the beginning of a loop iteration, which we need for our invariant, is simply the variable itself: *`z in z_nodes && n in n_nodes`*. After we have our invariant we still have to propagate this information upwards to the precondition. While doing so the expression of the anchor also might change because of other variable assignments.

4.5.2 Invariants

The set describing all possible values of a changing variable can now be used to transform an expression in our collection into a new one that does not contain any changing variables. We do this by quantifying over the set and replacing the changing variable with the quantified variable. Let us again have a look at our initial traversal example in Figure 4.1. Before we apply the forget operator we will have collected the following expression at the top of the while loop:

```
node!=null ? node in next_nodes : true
```

Since node is a changing variable we take advantage of its corresponding changing variable set, i.e. `node_nodes`. We transform the derived expression by quantifying over the elements of the `node_nodes` set and replacing all occurrences of `node` with the quantified variable `_node`:

FIGURE 4.6. Invariant

```
1 //require access
2 invariant forall q:Ref:: (q in next_nodes) ==> acc(q.next)
3 //define receiver nodes of next field
4 invariant forall _node:Ref:: (_node in node_nodes) ==>
5                         _node!=null ? _node in next_nodes : true
6 //define changing variable set
7 invariant node in node_nodes  //anchor
8 invariant forall _node:Ref:: (_node in node_nodes) ==>
9                         _node!=null ? _node.next in node_nodes : true
```

```
forall _node:Ref:: (_node in node_nodes) ==> _node!=null ? _node in
                        next_nodes : true
```

The whole invariant is shown in Figure 4.6. The first part, line 2, requires access permission to all field accesses that we collected. In this example we only have accesses to the field `next`, hence we only have the set `next_nodes`. As a reminder, `next_nodes` contains all receiver nodes of field accesses to field `next`. The second part, line 4, defines this set `next_nodes`. As we can see it relies on a changing variable, hence it had to be replaced with a quantified variable. The condition on line 5 corresponds to the loop condition. The last part, lines 7 and 8, defines the set describing the change of the variable `node`. Line 7 defines the anchor. Then on line 8 we recursively add the next pointers to the set.

From the definition of both sets we can conclude that both describe the same nodes, except `node_nodes` might also contain null. Therefore it would be possible to merge those sets and only introduce one new set either `node_nodes` or `next_nodes`. But as we explained in Section 4.3 this is not the case in general. We will further discuss this optimization in in Section 6.1.1

One optimization that one can do though is to merge both definitions of the sets into one invariant, while still keeping two separate sets. In the above example we can see that both definitions quantify over the same set. Hence we can include both formulas in the same quantifier. While analysing the inferred definitions of sets also from other examples we realized that they all have a similar structure. Especially the branch conditions are everywhere the same. Hence the logical consequence was to merge those definitions into one formula. This significantly shrinks the size of the contracts.

4.6 Consequences of Introducing Sets

This section summarizes advantages and disadvantages of the analysis with respect to sets. We will also examine different ways of extending the analysed program with the introduced sets.

4.6.1 Advantages and Disadvantages

Introducing sets has some advantages: they come in handy especially for separating access predicates from the predicates which define the receiver nodes. This was necessary to handle heap dependent receivers. Then we also introduced new sets for our forget operator. A third advantage is that they take care of duplicates and aliases.
On the other hand sets also have disadvantages. The main disadvantage is that we lose track of the order of the accesses, since sets are naturally unordered. In the array analysis we used the order in combination with the boundary function to handle inhales and exhales correctly by pushing the collected permission amounts into the permission amount parameter of the accessibility predicate. Due to the separation of requiring access an collecting the receiver nodes the array approach is not possible anymore. The consequence is that our analysis does not handle exhale and inhale statements. Luckily there are still many programs dealing with graph data structures and that do not use inhale and exhale. A consequence of not supporting inhales and exhales is that it is now very easy to infer postconditions. Since the permission do not change inside the method one can simply ensure the same permissions that were required.

4.6.2 Extending the Program

As one can see our algorithm introduces many different sets: one for each field and for each changing variable. Those sets have to be defined somewhere. We will discuss two ways to introduce these sets: the first through ghost functions and the second through ghost parameters.

Ghost Functions. Figure 4.7 shows again our starting example but this time instead of using ghost parameters it uses a ghost function. Instead of having the definitions of the set in the precondition and the invariant they are now part of the functions specification.
There are multiple issues with this approach. Firstly, the precondition of the function `fNodes` is not well-formed. In order to talk about `n.next` we need access to this field. We could then require the access to it in the precondition of the function. But when we use it in the precondition of the method `traverse` we cannot guarantee these permissions at line 6, since we only require the permission there. Effectively we need permission to talk about the set and we need the set to talk about the permissions. Hence we end up with a chicken and egg problem. Another problem with this approach is that this set is defined globally. If the data structure were modified we would need some way to change this set. Finally, this set defined by the function is underspecified, meaning we define what has to be part of that set but not which nodes are excluded. Hence it is hard to live up to the precondition on line 6, if one needs to guarantee access permissions to an unknown number of field accesses.

On the other hand the big advantage with ghost functions would be that the data structure is defined globally and the caller of the method does not need to explicitly define the set himself. He just has to guarantee enough permissions.

FIGURE 4.7. Ghost Functions

```
1  function fNodes(anchor: Ref): Set[Ref]
2  ensures anchor in result
3  ensures forall n: Ref :: n in result ==>
4                  (n.next!=null) ? n.next  in result : true
5  method traverse(nd: Ref)
6      requires forall n: Ref :: n in fNodes(nd) ==> acc(n.next)
7  {
8      var node :Ref := nd
9      while(node != null)
10     invariant forall n: Ref :: n in fNodes(node) ==> acc(n.next)
11     {
12         node := node.next
13     }
14 }
```

Ghost Parameters. Ghost parameters on the other hand do not suffer from these problems. Therefore in our analysis we define the sets in terms of ghost parameters. They still have some disadvantages: they modify the method's signature. This has the consequence that the algorithm has to modify the program everywhere where the method is used.
Another disadvantage is that the caller actually has to pass an instance of each set to the method. Depending on the form in which the caller already has the data structure it might be easy to pass the correct sets or he would have to first split his data structure into different sets to be able to instantiate the required ones. Let us consider an example where the method deals with a linked list of person data, where the list nodes in addition to the next node also point to a person node and this person node accesses fields containing information about the person. There the caller would need to provide different sets for the list nodes and for the person nodes, since they access different fields.

An intriguing question that might arise in the context of ghost parameters is whether introducing multiple sets is not just pushing the problem of defining precise contracts to the caller, since he now needs to instantiate those sets. This is only partially true. Considering our changing variable set the caller now has to pass the possible values of the variable, which is in contrast to the array analysis where we got the concrete values from our numerical domain. On the other hand the precondition actually defines the minimal requirements of the set and the caller just needs to pass any set that satisfies this condition. He does not need to pass the smallest such set. In our previous example `travRightAndLeft` in Figure 4.3 we have sets `left_nodes` and `right_nodes`. As a concrete instance the caller could now pass the set of all tree nodes twice. Effectively granting access to the left and right field to all nodes, where the specification would only require the access to a small subset.

CHAPTER 5

EVALUATION

We will evaluate the two algorithms in this chapter. We first show some results of the array analysis and then some results of the graph analysis. We discuss the inferred specifications, elaborate on some of the limitations and also explain some implementation details.

5.1 Inferred Array Specification

In this section we are going to inspect the array algorithm. To evaluate the results of our analysis we compared the specification inferred by our analysis with specifications written by humans. We ran the analysis on example methods for which we already had specifications. We then compared both specifications by verifying whether the human contract implies the computer contract or the other way around. We did this by using a helper method `humanImpliesComp`. Its precondition was the human contract and the postcondition was the automatically inferred one. The body of the method was empty. If Viper can verify the method it is true that the human contract implies the computer contract. We switched the pre- and postcondition for the method `compImpliesHuman`.

5.1.1 Max Array

Figure 5.1 shows the method `max_array` with the inferred contracts. The human contract is written for reference at line 31. The verification of our helper methods showed that the human contract implies the computer contract, but the computer contract does not imply the human contract. This means that the computer specification is more precise, requiring less permission than the human specification. Indeed, if we have a closer look at the specifications we see that the computer precondition only requires access to the array if the length of the array is at least 2,

FIGURE 5.1. Max Array

```
1  method max_array(arr: Array, lenA: Int, eps: Perm) returns (x: Int)
2    requires (forall p: Int :: true ==> acc(loc(arr, p).val,
3     (lenA <= 0 ?
4         none
5       : ((lenA>=1+p) && (p>=1) || (lenA>=2+p) && (p>=0) ? eps:none)
6  + f(-1*((lenA>=1+p) && (p>=1) || (lenA>=2+p) && (p>=0) ? eps:none)))
       ))
7    requires (eps > none) && (eps < write)
8  {
9    var y: Int
10   if (lenA <= 0) {
11     x := -1
12   } else {
13     x := 0
14     y := lenA - 1
15     while (x != y)
16       invariant (forall p: Int :: true ==> acc(loc(arr, p).val,
17        ((lenA>=1+p) && (p>=1) || (lenA>=2+p) && (p>=0) ? eps:none)))
18       invariant (lenA >= 1 + y) && ((y >= x) && (x >= 0))
19     {
20       if (loc(arr, x).val <= loc(arr, y).val) {
21         x := x + 1
22       } else {
23         y := y - 1
24       }
25     }
26     //loc(arr, 10).val := 0 //max_array_plus
27   }
28 }
29
30 //Human contract:
31 requires forall j: Int :: 0 <= j && j < lenA ==> acc(loc(arr,j).val)
```

whereas the human contract always requires the accesses. From the code we can see that if the length of the array is 0 we do not enter the else branch of the if statement on line 10 and if the length of the array is 1 we do not enter the loop. Hence it is correct to require access to the array only if the length is at least 2. In addition, the inferred contract is more precise because it only requires read permission, while the human made contract makes no distinction between read and writes.
On line 3 we can see the branch condition added due to the first if statement. Line 5 and 6 correspond to the invariant: first we simulate an exhale, positive permission amount, and then an inhale, hence the -1 in front of the expression. Line 7 then shows the requirements for our read permission `eps`.

The second example that we will analyse is almost equivalent to the previous example, except that we have an additional array access after the loop. This is depicted on line 26 in Figure 5.1. What the reader can try to do now is to come up with a precise specification. One will quickly realize that it is actually quite tricky. Here is the specification that we wrote by hand:

```
1 requires (lenA>1 ==> (forall j: Int :: 0 <= j && j <= lenA && j!=10
2                                     ==> acc(loc(arr, j).val,eps)))
3 requires (lenA > 0 ==> acc(loc(arr,10).val))
```

The first part `lenA>1` we got from the result of the algorithm run on the previous example. We only need to require permissions if the array has at least length 2. Then the tricky part is to take care of duplicates, else we would require access to the array at location 10 twice. Therefore we added the condition `j!=10`. Of course this is a made up example and it might not make much sense as is, but we can see that taking care of duplicates complicates the contract and usually is quickly forgotten. In our analysis the inhale of the invariant removes the need to explicitly check for duplicates: if the loop already requires permission to the array location 10 the simulated inhale would cancel out the permission required after the loop.

5.1.2 Longest Common Prefix

The method in Figure 5.2 starts iterating through the array from two arbitrary positions x and y. It keeps iterating until the value of the array at pointer x differs from pointer y. Hence it effectively determines the size of the longest common prefix of sub sequences of the array starting at positions x and y. Interestingly our test showed again that the computer specification is more precise than the human specification. The reason is that the inferred contract also considers the input parameter x and y, whereas the human contract simply requires access to the whole array. In most cases x and y are bigger than 0, hence we do not need to require access to the locations smaller than x or y.
The assertions on line 16 in the invariant restrict `p` to be between `x` or `y` and `lenA`. On line 4

FIGURE 5.2. Longest Common Prefix

```
1  method lcp(arr: Array, x: Int, y: Int, lenA: Int, eps: Perm)
2                                                    returns (n: Int)
3  requires (forall p: Int :: true ==> acc(loc(arr, p).val,
4         ((lenA+y>=1+p+x) && ((lenA>=1+p)      && (p>=y)) ||
5          (lenA>=1+p)     && ((lenA+x>=1+p+y) && (p>=x)) ? eps:none)
6  +f(-1* ((lenA+y>=1+p+x) && ((lenA>=1+p)      && (p>=y)) ||
7          (lenA>=1+p)     && ((lenA+x>=1+p+y) && (p>=x)) ? eps:none)
8      )))
9  requires (eps > none)  && (eps < write)
10 requires (0 <= x) && ((0 <= y) && ((x < lenA) && (y < lenA)))
11 {
12   n := 0
13   while ((x+n < lenA) && (y+n < lenA) &&
14                               loc(arr,x+n).val == loc(arr,y+n).val)
15   invariant (forall p: Int :: true ==> acc(loc(arr, p).val,
16         ((lenA+y>=1+p+x) && ((lenA>=1+p)      && (p>=y)) ||
17          (lenA>=1+p)     && ((lenA+x>=1+p+y) && (p>=x)) ? eps:none)))
18   invariant (lenA >= n+y) && ((lenA >= 1+y)
19         && ((lenA >= n+x) && ((lenA >= 1+x)
20         && ((n >= 0) && ((y >= 0) && (x >= 0))))))
21   {
22     n := n + 1
23   }
24
25   //human specification
26   requires (forall k: Int :: 0<=k && k<lenA ==> acc(loc(arr,k).val))
27   requires (0 <= x && 0 <= y && x < lenA && y < lenA)
28 }
```

and 6 we reuse this invariant to simulate an exhale and inhale operation. The reason why the inferred contract is so big is that our forget operator replaces the changing variable `n` in the formula `p==x+n` with another formula. This new formula tries to keep all relations between `p`, `lenA`, `x` and `y`. The same happens for the formula `p==y+n`. Finally, on line 18 we can see the invariant that we got from the numerical analysis, which Viper needs to have enough information about n in order to verify that the access happens in the range we provide access permission to.

5.1.3 Parallel Array Replace

The example in Figure 5.3 shows a method which forks off two threads by exhaling the precondition of another method and later inhaling the postcondition. This example shows how our algorithm handles multiple exhales and inhales. The test with our helper methods indicated

FIGURE 5.3. Parallel Array Replace

```
1  method Replace(arr: Array, left: Int, right: Int, from: Int, to: Int
       , lenA: Int, eps: Perm)
2  requires (forall p: Int :: true ==> acc(loc(arr, p).val,
3    (right - left <= 1 ?
4      rwMax(p == left, eps, (p == left ? write : none))
5      : ((left <= p) && (p < left + (right-left)\2)  ?    1/1 :none)
6    +  (((left + (right-left)\2 <= p) && (p < right) ?    1/1 :none)
7    + f(((left <= p) && (p < left + (right-left)\2)  ?-1*(1/1):none)
8    +   ((left + (right-left)\2 <= p) && (p < right) ?-1*(1/1):none)))
       )))
9  requires (eps > none) && (eps < write)
10 requires (0  <= left) && ((left < right) && (right <= lenA))
11 {
12   var mid: Int
13   if (right - left <= 1) {
14     if (loc(arr, left).val == from) {
15       loc(arr, left).val := to
16     }
17   } else {
18     mid := left + (right - left) \ 2
19     inhale mid * 2 == 2 * left + (right - left)
20
21     //fork-left
22     exhale 0 <= left && left < mid && mid <= lenA
23     exhale forall i: Int :: left<=i && i<mid ==> acc(loc(arr,i).val)
24     //fork-right
25     exhale 0 <= mid && mid < right && right <= lenA
26     exhale forall i: Int :: mid<=i && i<right==> acc(loc(arr,i).val)
27
28     //join-left
29     inhale forall i: Int :: left<=i && i<mid ==> acc(loc(arr,i).val)
30     //join-right
31     inhale forall i: Int :: mid<=i && i<right==> acc(loc(arr,i).val)
32   }
33 }
34 //human contract
35 requires (0 <= left && left < right && right <= lenA
36 requires forall i:Int:: left<=i && i<right ==> acc(loc(arr,i).val))
```

that both the human written and the inferred specifications imply each other, hence they are equivalent. In the specification each expression that we got from an exhale and inhale has its own line. We can see that all expressions we introduced due to inhales have negative and all expressions introduced due to exhales have positive permission amounts. Summing the permission amounts up results in the final permission amount we require. We can also see that at the boundary between an exhale and an inhale, i.e. between line 6 and 7, we introduce the boundary function `f`.
One small implementation detail can be seen on line 4: the `rwMax` function corresponds to the read function explained in Section 3.2.1. This function is defined globally; we introduced it in order to keep our specifications short.

5.1.4 Stress Test

The method `example_array` in Figure 5.4 is supposed to test all features of our algorithm together. Therefore it contains among other things an if statement, a while-loop, exhales and inhales, real array accesses, where we access the `val` field, and "fake" array accesses, e.g. on line 35. For the first time the results of our implication test showed that the inferred contract implies the human contract but not the other way around. This means that the human contract is more precise. We have to notice though that writing down the precise precondition was very tricky. While analysing the example we found out that the reason for this behaviour is due to our choice of handling read accesses. If we have a look at line 20 we see that we inhale permissions to location 3. Inside the while loop we will then read the array from location 3 to 10. Someone smart writing the specification can now conclude that we already have read permission to position 3, since any amount greater than 0 is enough to read the location, hence we do not need to require more.
The algorithm on the other hand just knows that it needs `eps` permission amount. When inhaling 1/10 of permissions he concludes that he still needs `eps - 1/10` permissions. If we add the extra assumption `eps<1/10` then our test succeeds in verifying that both contracts are equal.

To understand the contract one can map every occurrence of `rwMax` to a read or a write in the program. Each `rwMax` ensures enough permission to perform the corresponding action. Line 5 corresponds to the exhale and line 6 to the inhale of the invariant. Line 7 and following correspond to all accesses after the loop. At the end our precondition requires access permission to the array locations from 3 to 10, 20 and 100. This corresponds exactly to the array locations we access.

5.1.5 Extend Array

The method `extendArray` in Figure 5.5 takes as an input parameter an array; it assumes that the array has size 10. Then the program will extend it to size 19. This example demonstrates multiple features: if we have a look at the if condition inside the loop we see that it contains

FIGURE 5.4. Stress Test

```
1  method example_array(arr: Array, a: Int, eps: Perm)
2    requires (forall p: Int :: true ==> acc(loc(arr, p).val,
3    f((p==3?(-1)*(1/10):none) +
4              rwMax(p==3,   eps, rwMax(p == 2 * 3, write,
5              rwMax(p==100, eps,((p<=9) && (p>=3) || (a==p)?eps:none))
6    + f(-1 * rwMax(p==100, eps,((p<=9) && (p>=3) || (a==p)?eps:none))
7    + rwMax(p == 10, eps,
8        f(((p >  10) && (p < 15) ? (-1) * (1 / 1) : none)
9        + ((p == 20) || (p == 8) ?            eps : none)))))))))
10   requires (eps > none) && (eps < write)
11 {
12   var re : Ref; var t: Int
13   var end: Int; var r: Int
14   t := a
15   if (t < 0) {
16     t := 9
17   } else {
18     t := 0
19   }
20   inhale acc(loc(arr, 3).val, 1 / 10)
21   t := 3
22   loc(arr, 2 * t).val := 0
23   r := loc(arr, t).val
24   r := t
25   while ((t < 10) && (loc(arr, 100).val == 0))
26     invariant (forall p: Int :: true ==> acc(loc(arr, p).val,
27       rwMax(p==100, eps, ((p<=9) && (p>=3) || (a==p)? eps:none))))
28     invariant (t <= 10) && (t >= 3)
29   {
30     r := r + 1
31     r := loc(arr, t).val
32     r := loc(arr, a).val
33     t := t + 1
34   }
35   re := loc(arr, 1)
36   end := 99
37   end := loc(arr, t).val
38   inhale (forall q: Int :: (q>10) && (q<15) ==> acc(loc(arr,q).val))
39   r := loc(arr, 20).val
40   r := loc(arr, 8).val
41 }
42 // human contract
43 requires forall i:Int :: (((i>3 && i<=10) || i==a) && i!=6) || i==20
       || i==100==> acc(loc(arr,i).val,eps)
44 requires acc(loc(arr,6).val)
```

a changing variable, hence we cannot use it in our specification: instead the `max` operator is introduced (cf. line 16). Lines 16 and 17 correspond to the true branch and line 18 corresponds to the false branch. Another feature that shows up is the way we handle inhales inside a loop: as we can see, the true branch accesses the array from position 11 to 19, hence the specification on line 17, but before the method writes to those locations it inhales the corresponding permission. In the invariant we can see this on line 16. Here, the result of the forget operator, i.e. `(p < 12) && (p > 18)`, is equivalent to false. Consequently the permissions inhaled will not be considered and the precondition requires access permission to the array from 0 to 19. The human contract on the other hand takes the inhale into consideration and requires access permission only to the array locations from 0 to 10. With a more precise forget operator we could improve the inferred contract.

FIGURE 5.5. Extend Array

```
method extendArray(arr: Array)
  requires (forall p: Int :: true ==> acc(loc(arr, p).val,
    max(f(((p < 12) && (p > 18) ? -1 * (1 / 1) : none)
        + ((p <=19) && (p >=11) ? write : none)),
          ((p <=10) && (p >= 0) ? write : none))
  +f(-1 *
    max(f(((p < 12) && (p > 18) ? -1 * (1 / 1) : none)
        + ((p <=19) && (p >=11) ? write : none)),
          ((p <=10) && (p >= 0) ? write : none))
    + (p == 15 ? write : none))))
{
  var i: Int
  i := 0
  while (i < 20)
    invariant (forall p: Int :: true ==> acc(loc(arr, p).val,
        max(f(((p <  12) && (p >  18) ? -1 * (1 / 1) : none)
            + ((p <= 19) && (p >= 11) ? write : none)),
              ((p <= 10) && (p >=  0) ? write : none))))
    invariant i >= 0
  {
    if (i > 10) {
      inhale acc(loc(arr, i).val, write)
      loc(arr, i).val := 0
    } else {
      loc(arr, i).val := 0
    }
    i := i + 1
  }
  loc(arr, 15).val := 0
}

//human contract
requires forall i:Int:: 0<=i && i<=10 ==> acc(loc(arr,i).val))
```

5.2 Inferred Graph Specification

In this section we evaluate the specifications inferred for graph examples. Since the inferred specifications rely on multiple sets and the human specifications sometimes only depend on one set we cannot use the helper methods from before to check which one is more precise. This time we do the comparison by hand. Additionally we discuss the different parts of the specification. First we have a look at some linked list examples and then we move on with tree examples and finish with a general graph example.

5.2.1 Linked List Traversal

The method `traverse` in Figure 5.6 is a basic implementation of a list traversal, where each visited node will be marked. As already explained in Chapter 4 the first part of the specification requires the access permissions to the field: lines 2 and 3. The second part is the definition of the sets: from line 4 onwards. One can easily map parts of the set specification to the program. For example line 6 belongs to the field accesses in the while condition on line 13. Then line 7 is introduced due to the loop condition. The part on line 8 belongs to the field accesses inside the loop and the last part on line 9 describes the changes of the variable `node`.
Comparing the human specification with the computer specification, we see that the latter requires permission to the `next` and `is_marked` field for the whole list. The computer specification on the other also takes the loop condition into account, which makes the specification more precise: it effectively requires access to the fields only for nodes in the list until the `is_marked` field is false. We can conclude this, since we only require the `_node.next` element to be in the `node_nodes` set (line 9) if it is not marked yet (line7). And only from this set `node_nodes` we actually require elements to be in `next_nodes` set. Concretely: if we have a list whose third element is for some reason already marked our specification only requires access permission for the first three nodes, whereas the human specification requires access permissions for all nodes in the list.
Even though this is the minimal requirement to satisfy the precondition, the caller can still instantiate the set with all nodes of the list. Then of course he also has to provide permission to the fields for all nodes.

5.2.2 Node Before Five

The example shown in Figure 5.7 shows another list traversal. This time we deal with a doubly linked list. The method tries to find the value 5 and then returns the previous node.
Again the specification is more precise than just requiring access to the whole list. Firstly because the `next_nodes` set only needs to contain all nodes from the list until one node has the value 5, and secondly because we only require access to the `prev` field for the nodes that have value 5.

FIGURE 5.6. Traverse List

```
1  method traverse(nd: Ref, node_nodes: Set[Ref],
       is_marked_nodes:Set[Ref], next_nodes: Set[Ref])
2    requires (forall r:Ref::(r in next_nodes)     ==>acc(r.next))
3    requires (forall r:Ref::(r in is_marked_nodes)==>acc(r.is_marked))
4    requires (nd in node_nodes) &&
5     (forall _node: Ref :: (_node in node_nodes) ==>
6         (_node in next_nodes) && (_node.next in is_marked_nodes) &&
7        ((_node != null)  && !_node.next.is_marked ?
8          (_node in is_marked_nodes) && (_node in next_nodes) &&
9          (_node.next in node_nodes) : true))
10 {
11   var node: Ref
12   node := nd
13   while ((node != null) && !node.next.is_marked)
14    invariant forall r:Ref::(r in next_nodes)     ==>acc(r.next)
15    invariant forall r:Ref::(r in is_marked_nodes)==>acc(r.is_marked)
16    invariant (node in node_nodes) &&
17      (forall _node: Ref :: (_node in node_nodes) ==>
18         (_node.next in is_marked_nodes) && (_node in next_nodes) &&
19        ((_node != null) && !_node.next.is_marked ?
20             (_node in is_marked_nodes) && (_node in next_nodes) &&
21             (_node.next in node_nodes) : true))
22   {
23     node.is_marked := true
24     node := node.next
25   }
26 }
27 //human specification
28 define INV(nodes)
29      !(null in nodes)
30   && (forall n: Ref :: n in nodes ==> acc(n.next))
31   && (forall n: Ref :: n in nodes ==> acc(n.is_marked))
32   && (forall n: Ref :: n in nodes && n.next  != null ==> n.next  in
       nodes)
33
34 method trav_rec(nodes: Set[Ref], nd: Ref)
35   requires nd in nodes
36   requires INV(nodes)
```

FIGURE 5.7. Node Before Five

```
method findNodeBeforeFive(nodesN: Set[Ref], nd: Ref,
   node_nodes: Set[Ref], data_nodes: Set[Ref],
   prev_nodes: Set[Ref], next_nodes: Set[Ref]) returns (res: Ref)
  requires (forall r: Ref :: (r in next_nodes) ==> acc(r.next))
  requires (forall r: Ref :: (r in prev_nodes) ==> acc(r.prev))
  requires (forall r: Ref :: (r in data_nodes) ==> acc(r.data))
  requires (nd in node_nodes) &&
  (forall _node: Ref :: (_node in node_nodes) ==>
    (_node in data_nodes) &&
    ((_node != null) && (_node.data != 5) ?
        (_node in next_nodes) && (_node.next in node_nodes)
      : (_node in data_nodes) &&
        ((_node != null) && (_node.data == 5) ?
            (_node in prev_nodes) : true)))
{
  var node: Ref
  node := nd
  while ((node != null) && (node.data != 5))
    invariant (forall r: Ref :: (r in next_nodes) ==> acc(r.next))
    invariant (forall r: Ref :: (r in prev_nodes) ==> acc(r.prev))
    invariant (forall r: Ref :: (r in data_nodes) ==> acc(r.data))
    invariant (node in node_nodes) &&
    (forall _node: Ref :: (_node in node_nodes) ==>
        (_node in data_nodes) &&
        ((_node != null) && (_node.data != 5) ?
            (_node in next_nodes) && (_node.next in node_nodes)
          : (_node in data_nodes) &&
            ((_node != null) && (_node.data == 5) ?
                (_node in prev_nodes) : true)))
  {
    node := node.next
  }

  if ((node != null) && (node.data == 5)) {
    res := node.prev
  }
}
```

FIGURE 5.8. Get Third Element

```
1 method getThird(nd: Ref, next_nodes: Set[Ref]) returns (third: Ref)
2   requires (forall r: Ref :: (r in next_nodes)==>acc(r.next, write))
3   requires (nd in next_nodes) && (nd.next in next_nodes)
4 {
5   var second: Ref
6   second := nd.next
7   third := second.next
8 }
```

5.2.3 Get Third Element

The method `getThird` depicted in Figure 5.8 traverses a list until it reaches its third element and returns it. The first thing to notice is that the contract grows in proportion with the methods body. Here we do not have a while loop and only access one field, hence we also introduce only one set. The contract is very easy to understand, it just adds the node `nd` and `nd.next` to the set.

5.2.4 Tree Traversals

Let us now have a look at a tree example. Method `traverseRandom` in Figure 5.9 traverses the tree by sometimes following the left node and sometimes the right node. What makes this example interesting is that we have an if statement, on line 29, with a changing integer variable. Hence, we cannot just copy the condition into our specification but rather we have to forget about it. The way we do this is by merging both branches. This can be seen in the specification on lines 8 and 9. Line 8 corresponds to the true branch and line 9 corresponds to the false branch. They are not connected by a ternary conditional operator but simply by an `&&`.
Secondly we show this example because it nicely demonstrates the difference between the receiver node sets and the changing variable sets. Here the algorithm actually traverses two levels of the tree in one loop iteration. This means `node_nodes` contains only every other level of the tree and therefore we only require access to the `right` field for nodes on every other level instead of all nodes. This might be useful if an algorithm somehow wants to partition the tree and forks off two threads where each works on its own part of the tree.

5.2.5 Find Local Mimimum

Even though the methods from the previous examples were working on lists or trees the algorithm did not assume so. The algorithm does not assume any properties of the graph. We make this explicit in the example shown in Figure 5.10. The goal of this method is to traverse the graph and find a local minimum. We define a local minimum to be a node whose value is smaller than the values of its neighbors, i.e. the nodes he can reach in one step. Each node in the graph can

FIGURE 5.9. Tree Traversals

```
1  method traverseRandom(nd: Ref, node_nodes: Set[Ref], right_nodes:
       Set[Ref], left_nodes: Set[Ref])
2    requires (forall r: Ref :: (r in left_nodes)  ==> acc(r.left))
3    requires (forall r: Ref :: (r in right_nodes) ==> acc(r.right))
4    requires (nd in node_nodes) &&
5    (forall _node: Ref :: (_node in node_nodes) ==>
6    (_node != null ?
7      (_node in left_nodes) &&
8      ((_node.left in left_nodes) &&(_node.left.left in node_nodes)&&
9      ((_node.left in right_nodes)&&(_node.left.right in node_nodes)))
10     : true))
11 {
12   var i: Int
13   var node: Ref
14   i := 0
15   node := nd
16   while (node != null)
17    invariant (forall r: Ref :: (r in left_nodes)  ==> acc(r.left))
18    invariant (forall r: Ref :: (r in right_nodes) ==> acc(r.right))
19    invariant (node in node_nodes) &&
20    (forall _node: Ref :: (_node in node_nodes) ==>
21    (_node != null ?
22      (_node in left_nodes) &&
23      ((_node.left in left_nodes)&&(_node.left.left in node_nodes) &&
24      ((_node.left in right_nodes)&&(_node.left.right in node_nodes))
25      : true))
26     invariant i >= 0
27   {
28     node := node.left
29     if ((i < 5) || (i > 10)) {
30       node := node.left
31     } else {
32       node := node.right
33     }
34     i := i + 1
35   }
36 }
```

reach other nodes through the fields `e1` and `e2`. The value of each node is stored in the field `data`. Viper is able to verify this program, meaning we inferred enough permission for each field access. In our contract, we have a set for each field, plus also a set for the changing variable `nd`. The specification is similar to the specification from the tree traversal example. Line 11 and line 14 correspond to the if conditions inside the loop. All accesses happening in the true branch correspond to an assignment of the receiver node to the corresponding set in the true branch of the ternary conditional operator. For example the first part of line 15, i.e. `node in e2_nodes`, corresponds to the field read on line 40.

5.2.6 Instantiating the Sets

In the discussed example we have seen that the introduction of multiple sets introduces more precision. Depending on which factor is more important, precision or the number of sets, one might easily also merge those sets. For us the goal clearly was precision. Having multiple sets might at first seem to unnecessary complicate the specification. But in general these sets are quite intuitive. For example if the caller knows what kind of data structure he is working with, then it is also very easy and quite natural to him to see which nodes access which fields, hence he can quickly instantiate the correct receiver node set. Also if he knows what the algorithm does and in particular in which way the data structure is traversed, he can precisely instantiate the changing variable sets.

FIGURE 5.10. Find Local Minimum

```
field e1: Ref; field e2: Ref; field data: Int

method findMin(e2_nodes: Set[Ref], node: Ref, data_nodes: Set[Ref],
   e1_nodes: Set[Ref], nd_nodes: Set[Ref]) returns (nd: Ref)
requires (forall r: Ref :: (r in e1_nodes)   ==> acc(r.e1, write))
requires (forall r: Ref :: (r in e2_nodes)   ==> acc(r.e2, write))
requires (forall r: Ref :: (r in data_nodes) ==> acc(r.data, write))
requires (node in nd_nodes) &&
  (forall _nd: Ref :: (_nd in nd_nodes) ==>
    (_nd in data_nodes) && (node in e1_nodes) &&
    (node.e1 in data_nodes) &&
    ((node.e1 != null) && (node.e1.data < _nd.data) ?
      (node in e1_nodes) && (node.e1 in nd_nodes)
    : (node in e2_nodes) && (node.e2 in data_nodes) &&
      ((node.e2 != null) && (node.e2.data < _nd.data) ?
        (node in e2_nodes) && (node.e2 in nd_nodes) : true)))
{
  var moved: Bool;  var value: Int
  nd := node
  moved := true
  while (moved)
    invariant (forall r:Ref:: (r in e1_nodes)  ==>acc(r.e1, write))
    invariant (forall r:Ref:: (r in e2_nodes)  ==>acc(r.e2, write))
    invariant (forall r:Ref:: (r in data_nodes)==>acc(r.data,write))
    invariant (nd in nd_nodes) &&
    (forall _nd: Ref :: (_nd in nd_nodes) ==>
    (_nd in data_nodes) && (node in e1_nodes) &&
    (node.e1 in data_nodes) &&
    ((node.e1 != null) && (node.e1.data < _nd.data) ?
      (node in e1_nodes) && (node.e1 in nd_nodes)
    : (node in e2_nodes) && (node.e2 in data_nodes) &&
      ((node.e2 != null) && (node.e2.data < _nd.data) ?
        (node in e2_nodes) && (node.e2 in nd_nodes) : true)))
   {
    moved := false
    value := nd.data
    if ((node.e1 != null) && (node.e1.data < value)) {
      nd := node.e1
      moved := true
    } elseif ((node.e2 != null) && (node.e2.data < value)) {
      nd := node.e2
      moved := true
    }
  }
}
```

CHAPTER 6

CONCLUSION AND FUTURE WORK

The previous chapters introduced, explained and analysed our two permission inference algorithms: the first for arrays and the second for graphs.

The array algorithm was able to infer invariants and preconditions, which included quantified permissions, that granted enough permission to all fields that were accessed by the method. The algorithm handles a wide variety of methods with only a few restrictions: it cannot handle heap dependent receivers. This means that we cannot handle data structures such as arrays of arrays (2D arrays). Additionally, it solely handles arrays, so for other data structures one has to use the graph analysis. The proposed analysis deals with loops by overapproximating the possible values of changing variables. In order to do so we developed a forget operator which takes advantage of the numerical domain's forget operator. The developed forget operator transforms one expression mentioning changing variables by approximating it with a new expression that does not contain changing variables. The algorithm handles any number of exhales and inhales, as long as they are in the supported form, by summing up the permission amounts. It further handles fractional permissions. In the case of read permissions we decided to use ghost parameters to define a concrete permission amount that we require in order to read an array location (cf. Section 3.2.2). Some shortcomings are that the forget operator introduces imprecision which can lead to imprecise contracts. Furthermore the inferred contracts rely on ghost parameters and functions, so the original program has to be extended.

The graph analysis is quite different from the array analysis. First we got rid of some restrictions. Since recursive data structures usually rely on a field read where the receiver itself is affected by a field read we allow heap dependent receivers. Second we generalized the algorithm to handle different fields. On the other hand we added the restriction that we do not allow writes

to locations that are part of our state (cf. Section 4.1). Another limitation, due to the extensive use of sets, is that the algorithm does not handle exhales and inhales. Even with these restrictions there are several examples which the algorithm can handle.
Differences to the array algorithm are of course the introduction of sets and the separation of requiring access permission and defining the sets. A major difference is the way we forget about changing variables. In methods with recursive data structures changing variables are usually references rather than integers so we proposed a new forget operator which collects possible values of a changing variable in a set.
Some disadvantages of the inferred contracts are that they are again quite long. Moreover the introduction of sets introduces the need for the caller to instantiate those sets. But as explained in Section 5.2.6 the instantiation should be intuitive.

Based on the evaluation in Chapter 5 we can conclude that both algorithms infer specifications that are sound and actually very precise. In many cases they are even more precise than what humans generally infer, since keeping all special cases, e.g. aliases or branches, in mind can be very cumbersome.

6.1 Future Work

There are a number of ways one can address the discussed short comings. In this section we will elaborate on our ideas to further improve the proposed analysis.

6.1.1 Simplify Contracts

Since the inferred contracts are sometimes not very easy to read, simplifying them would automatically be beneficial. In this section we propose multiple ways to make them more concise and make them look more similar to the ones human would write, which supports readability.

- **Arrays.** It sometimes happens that contracts for arrays contain negative permission amounts that do not affect any positive permission amounts, for example due to an inhale at the bottom of the method. Those permissions usually will get cut off by the boundary function. Hence an optimization would be to just omit this negative part at the end. Figure 6.1 shows such an example where one could omit line 3.

  If there are no inhales or exhales and we do not need to distinguish between read and write permission amounts then there is no need to push the inferred expression into the permission amount parameter of the accessibility predicate, but it can rather be used on the left hand side of the implication, i.e. `c(p)`, of the quantified permission expression:

```
forall p:Int:: c(p) ==> acc(loc(arr,p).val)
```

FIGURE 6.1. Remove Negative Permission

```
1 method negPerm(arr: Array)
2   requires forall p:Int:: true ==> acc(loc(arr,p).val, p==1? 1:0
3                                                       + f(p==1?-1:0))
4 {
5   exhale acc(loc(arr,1).val,1)
6   inhale acc(loc(arr,1).val,1)
7 }
```

We can take this one step further: if in our `c(p)` expression we only have equalities, e.g. `p==1`, then we do not necessarily need quantifiers at all. For each equality `p==exp` we would simply add a separate `acc(loc(arr,exp).val)` to the precondition. But one has to take care to not require access permission to the same location multiple times, since with this approach we would require the access permissions separately.

- **Graphs.** Having a close look at the inferred contract one might sometimes see that in the set definition part we have the same element assignment more than once (see lines 24 and 27 of the example in Figure 5.7). This can happen when the assignments occur in different branches or subbranches. As a reminder the branches in the contracts were introduced due to an if condition. If an access to the same field happens in the true and the false branch of an if condition, then also in our contract we will end up having two assignments corresponding to the accesses. One could push those assignments in the expression one level up, and consequently merging those duplicates. Similarly if we have an assignment higher up the condition-tree and the same further down,as is the case in Figure 5.7, one could remove the latter one.

  Next one should investigate the effects of choosing fewer sets. This would probably lead to less precision, but it would be interesting to see how much. The advantage is that the contract is a lot shorter and easier to understand and in addition the caller has to initialize fewer sets. Furthermore, the contracts would be more uniform across methods that work on the same data structure.

6.1.2 Improve Forget Operator

Especially in the array algorithm we could gain more precision if we had a more precise forget operator. Since our forget operator heavily relies on a numerical domain, one way to improve the precision of the contracts would be to use a domain or develop a new domain that is more precise than our current one, which is polyhedra.

Another probably more precise way is to apply the algorithm explained in [9]: the algorithm computes the cover of an expression. The paper defines cover as the most precise quantifier-free

FIGURE 6.2. Inhale inside Loop

```
1  method toOutside(arr: Array, ab: Ref, lenA: Int) returns (x: Int)
2  {
3    var t: Int := 0
4    while (t<10)
5      invariant t>=0 && t<=10
6      invariant forall p:Int :: (p>=0 && p<t) ==> acc(loc(arr,p).val)
7    {
8      inhale acc(loc(arr,t).val)
9      loc(arr,t).val := 0
10     t := t+1
11   }
12 }
```

overapproximation to existential quantifier elimination, and describes algorithms to compute the cover of formulas. We believe that cover could improve our forget operator.

For the graph algorithm one could investigate whether it is possible to reuse our forget operator from the array analysis and instead of relying on the forget operator of the numerical domain use a forget operator of an abstract domain for references.

6.1.3 Postconditions & Invariants

In our backward propagation we infer, through a weakest precondition analysis, a precise precondition. Now to also infer postconditions we would need another forward propagation: starting from the permissions we collected we would propagate this information forward by a strongest postcondition analysis and therefore derive a postcondition which includes all permissions that are left at the end of the method.
Since invariants are both pre- and postconditions for loops at the same time we can analogously strengthen the invariants through a forward propagation. As already discussed in Section 4.5.2 inhales inside the loop have some challenges. Figure 6.2 shows a method which initializes an array and passes the new permissions outwards. What is interesting to see is that in this case the changing variable `t` is an important part of the specification. Hence we cannot simply forget about it. Finding a way to deal with this is an interesting task left for future work.

6.1.4 Heap Analysis

Having a heap analysis in order to get information about aliases and certain field values would help with removing some of the current limitations. It would be possible to handle heap dependent receivers in an array. Additionally one might also be able to handle more modifications of the

data structure in the graph analysis. For example switching the left and right branch inside a tree.

With the help of a heap analysis one can also work on combining both algorithms. They both have their own advantages and disadvantages, hence combining them and creating an algorithm with the advantages of both algorithm is the logical next step.

6.1.5 Nested Quantifiers

Right now there is work in progress to relax some restrictions on the form of quantified permissions in Viper. In the future it will also support nested quantifiers and quantifiers with multiple variables. In this section we want to discuss how this could benefit our analysis.
If we were allowed to use multiple quantified variables we could extend our array analysis to handle methods that deal with multiple arrays. In addition to quantify over the array index we would also quantify over the array itself. The shape of our contract would then look like this:

```
forall a:Array, i:Int :: true ==> acc(loc(a,i).val, p(a,i))
```

In fact we can generalize the analysis to handle not only `loc` but any function that returns a reference. As long as the function is injective we would simply need to quantify over each parameter separately. Assuming we have a function `foo(x:X ,y:Y, z:Z):Ref` we can quantify over each variable like this:

```
forall x:X, y:Y, z:Z :: true ==> acc(foo(x,y,z).val, p(x,y,z))
```

The expression `p` defines the permission needed depending on `x`, `y` and `z`

6.2 Final Remarks

With this work we wanted to give a better understanding on the challenges that arise when inferring permissions and how one could address them. We did this by developing and implementing an inference algorithm that focuses on different data structures. When further improving the analysis, for example by inferring cleaner specifications, introducing heap analyses or taking advantage of nested quantifier, we believe that in the future we can see better tool support that help the programmer specify the permissions needed.

BIBLIOGRAPHY

[1] *Sample project page*.
http://www.pm.inf.ethz.ch/research/sample.html.

[2] 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, Springer, 2005, pp. 364–387.

[3] S. BLOM AND M. HUISMAN, *The vercors tool for verification of concurrent programs*, in FM 2014: Formal Methods, C. Jones, P. Pihlajasaari, and J. Sun, eds., vol. 8442 of Lecture notes in computer science, Berlin, Germany, May 2014, Springer, pp. 127–131.

[4] P. COUSOT AND R. COUSOT, *Static determination of dynamic properties of programs*, in Dunod, 1976.

[5] 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, POPL '77, New York, NY, USA, 1977, ACM, pp. 238–252.

[6] 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, ACM, 1978, pp. 84–96.

[7] 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, Springer, 2008, pp. 337–340.

[8] J.-C. FILLIÂTRE AND A. PASKEVICH, *Why3—where programs meet provers*, in European Symposium on Programming, Springer, 2013, pp. 125–128.

[9] S. GULWANI AND M. MUSUVATHI, *Cover algorithms and their combination*, in European Symposium on Programming, Springer, 2008, pp. 193–207.

[10] C. A. R. HOARE, *An axiomatic basis for computer programming*, Communications of the ACM, 12 (1969), pp. 576–580.

[11] K. R. M. LEINO, P. MÜLLER, AND J. SMANS, *Verification of concurrent programs with chalice*, in Foundations of Security Analysis and Design V, Springer, 2009, pp. 195–222.

[12] A. MINÉ, *The octagon abstract domain*, Higher-order and symbolic computation, 19 (2006), pp. 31–100.

[13] 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, Springer, 2016, pp. 41–62.

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

[15] J. SMANS, B. JACOBS, AND F. PIESSENS, *Implicit dynamic frames: Combining dynamic frames and separation logic*, in European Conference on Object-Oriented Programming, Springer, 2009, pp. 148–172.

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):

Automatic Inference of Quantified Permissions by Abstract Interpretation

Authored by (in block letters):

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

Name(s):	**First name(s):**
Walter	Seraiah

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)**
Bern, 05.08.16	Seraiah Walter

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