Department of Computer Science,
ETH Zürich, Switzerland

Inference of Progressive Loop Invariants for Array Programs

Master's Thesis

Nils Becker

supervised by
Jérôme Dohrau, Alexander J. Summers,
Prof. Dr. Peter Müller

April 2020

Abstract

Formal verification of programs that utilize loops typically requires the user to provide inductive loop invariants. Providing such invariants can be cumbersome which is why a wide range of inference techniques have been proposed. These techniques are, however, limited with respect to array programs. In this thesis we introduce progressive loop invariants – an abstraction that captures information about which loop iterations have already been executed and which ones will be executed in the future. We, furthermore, discuss a series of analyses based on progressive loop invariants that let us infer useful information about the behaviour of array programs and help us verify such programs. Based on a set of examples we, furthermore, show that these analyses work well in practice.

Acknowledgements

I would like to thank Jérôme Dohrau and Alexander Summers for the time they invested supervising this thesis and the feedback they gave me. I would, also, like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this project in his group.

Contents

Chapter 1

Introduction

Loops are a vital component of many programs that utilize arrays. Formal verification of such programs typically requires inductive proofs. To generate such proofs many verification tools require their users to provide inductive loop invariants. In the base case of the induction the tool then tries to show that the invariant holds before the loop is entered and in the step case it shows that the invariant is preserved when any iteration is executed. It can then, by induction, assume that this invariant holds in any state that can be reached by executing arbitrarily many loop iterations. Providing these invariants puts a burden on the user that is further increased if a tool based on separation logic is used which typically requires the user to additionally specify the heap locations that are accessed during the execution of a loop.

To alleviate some of this burden a wide range of automatic analyses have been proposed (e.g. [5, 16, 19, 24]). Many of these approaches are, however, limited with respect to array programs, e.g. because they do not distinguish between different elements in an array. Some recent work tries to overcome these limitations [15, 17] however these approaches are limited in the kind of array programs they can handle and the technique described by Dohrau et al. only produces loop pre- and postconditions and no invariants. Such invariants are, however, desirable since they e.g. allow us to verify user-provided invariants even if those user-provided invariants on their own are not sufficient to verify the program. One such application would be to automatically infer the necessary information about what heap locations are accessed while letting the user provide invariants containing information about what a program calculates.

At the beginning of any iteration an invariant typically gives information about what happened in previous iterations. After providing some background knowledge in Chap. 2 we will, therefore, introduce the novel concept of progressive loop invariants in Chap. 3 to capture information about what what loop iterations have already been executed and which ones will be executed in the future. Additionally we will discuss a technique

for automatically inferring such progressive loop invariants in that chapter. Subsequently we will see in Chap. 4 that progressive loop invariants are a useful abstraction for many different kinds of analyses. In particular we will use them to (1) extend the analysis described by Dohrau et al. [17] to produce invariants and apply it to a wider range of array programs, (2) design an analysis that automatically generates inductive proofs about the values stored in an array, and (3) design an analysis that identifies loops that terminate. We evaluate these analyses based on a number of examples and tests in Chap. 5 and in Chap. 6, after concluding, we discuss a wide range of directions for future work (Sec. 6.1). The main contributions of this thesis can be summarized as follows:

- A novel concept of progressive loop invariants that at a given point during the execution of a loop captures information about what iterations have already been executed and which ones will be executed in the future
- A technique for automatically inferring progressive loop invariants
- An inference mechanism for invariants containing information about array locations that are accessed during the execution of a loop
- An inference mechanism for invariants containing information about values stored in an array
- A termination analysis
- An implementation of these techniques
- An evaluation of these techniques based on a wide range of examples and tests

Chapter 2

Background

2.1 Language

The semantics of the language we will use in this thesis is largely standard but is extended with syntax for verification specific tasks that will be explained in the remainder of this chapter. Moreover, arrays in this language are heap-allocated and do not overlap. While we only look at one-dimensional arrays in this thesis the described techniques can be extended to multi-dimensional arrays. These techniques, moreover, only require the array implementation to provide a lookup function and are, therefore, relatively independent of the underlying implementation (e.g. C-style arrays or linked lists can be used so long as the implementation guarantees that they do not overlap).

2.2 Separation Logic

Separation logic is an extension of Hoare logic which allows reasoning about disjoint parts of the heap which can be reasoned about separately [26]. Consequently it is useful for the verification of concurrent programs [23] as well as for allowing modular verification of large programs by restricting what heap locations certain parts of a program can modify [25, 28]. Separation logic is, moreover, commonly extended by introducing *fractional permissions* and correspondingly generalized *access predicates* [7]. An access predicate $\mathbf{acc}(l, \alpha)$ denotes α *permissions* to a heap location l. $\mathbf{acc}(l, 0)$ denotes no permissions (we sometimes also write $\mathbf{acc}(l, \mathbf{none})$) whereas $\mathbf{acc}(l, 1)$ denotes read-write permissions (we sometimes also write $\mathbf{acc}(l, \mathbf{write})$). For $0 < \alpha < 1$, $\mathbf{acc}(l, \alpha)$ denotes read permissions and for $\alpha \notin [0, 1]$, $\mathbf{acc}(l, \alpha)$ is equivalent to false. In order to execute an assignment to a heap location or read from a heap location the program has to hold the corresponding permissions for that location. In particular this also means that if we want to evaluate an expression, e.g. to check that a boolean condition holds, we have to have enough permissions to read the value of each heap location

the expression refers to. If we have all necessary permissions to evaluate an expression we say that that expression is *framed*.

When handling arrays it is typically useful to use *quantified access predicates* which for the purposes of this thesis will have the form $\forall q.\mathbf{acc}(l, \alpha_q)$. For example the predicate $\forall q.\mathbf{acc}(a[q], 0 \leq q < a.\texttt{length}\ ?\ \mathbf{write} : \mathbf{none})$ denotes read-write access to every element of an array a.

To facilitate verification many tools based on separation logic (e.g. [22]) require guidance, in the form of specifications, from the user. Common forms of specifications are *pre-* and *postconditions* which are logical expressions that must, respectively, hold before and after a method or function is executed. In order to call a method we have to make sure its precondition holds and transfer all permissions required by the precondition to the callee. When a method returns we get to assume that its postcondition holds and the permissions mentioned in the postcondition are transferred back to the caller. *Invariants* are another form of specification that is often useful when verifying programs that utilize loops. We will take a closer look at these in Sec. 2.3. Additionally a user may add `assume` and `assert` statements to a program. When the verifier encounters an `assume` statement it assumes that a particular logical expression holds at that point during the program execution. When it encounters an `assert` statement it checks that a particular logical expression is guaranteed to hold at that point. Analogously we define `inhale` and `exhale` statements which can be used to, respectively, add and remove permissions from the program state. These statements are useful for encoding a number of different language features: e.g. when acquiring a lock we may `inhale` permissions to the heap locations guarded by that lock which are `exhale`d again when the lock is released.

2.3 Running Example

Example 1 shows a program that takes a black and white image as input and increases its brightness by adding 10 to the value of each pixel. We then model enqueueing that pixel in a rendering pipeline by giving up some permission to it using an `exhale` statement.

In order to verify programs, such as this one, that utilize loops one type of annotation that is often used are *inductive loop invariants*. These should be true in any state that can be reached by executing arbitrarily many loop iterations and require the verifier to do an inductive proof: it checks that the invariant holds before the loop is entered (base case) and that it is preserved when executing a loop iteration (step case). If both of these checks succeed we can then, by induction, assume that it holds in any state that can be reached by executing arbitrarily many loop iterations. For the invariant on lines 5 - 6 we first show that `image` = `original_image` since $q <$ `i` is always false before entering the loop. To prove that the invariant is preserved

Example 1. A program that brightens an image and renders it

```
1  var image := get_image()
2  var original_image := image.copy()
3  var i := 0
4  while (i < image.length)
5    invariant ∀q ∈ [0,image.length).image[q] = q < i ?
6      original_image[q] + 10 : original_image[q]
7  {
8    // increase brightness
9    image[i] := image[i] + 10
10
11   // render
12   exhale acc(image[i], 1/2)
13
14   i += 1
15 }
```

we, then, start out with only the permissions and constraints we obtain from the invariants, execute a loop iteration, and check that the invariant still holds in the new state. Finally, since we know that, after exiting the loop, the loop condition is false we can deduce that $q <$ `i` is true for all values of q and, thus, that we have incremented every pixel by 10.

Trying to verify this program will, however, fail since the invariant is not framed: when checking loop invariants we can only use permissions given to us by other invariants. Chap. 4.1 will introduce a technique to automatically infer the necessary framing invariant.

2.4 Abstract Interpretation

Abstract interpretation [9] is a form of static analysis that can be used for verification without the need for guidance from the user. In abstract interpretation we use an *abstract state* belonging to an *abstract domain* to represent many program states at once. We then simulate the program's execution on this abstract state until we reach a fixed-point, i.e. a point at which the abstract state stays the same at all program points. The choice of abstract domain depends on the kind of program-properties we are interested in. E.g. a popular choice (and the one we will be using in this thesis) for capturing numerical constraints on program variables is the polyhedra abstract domain [10]. It captures a set of constraints of the form

$$\sum_{v \in V} a_v v \leq b,$$

where V is the set of program variables and the corresponding coefficients a_v as well as b are constants determined for a particular abstract state during the

abstract interpretation, i.e. it captures linear relationships between program variables. Note that equalities can be represented as a pair of constraints: e.g. the pair of constraints $x - y \leq 1$ and $y - x \leq -1$ is equivalent to $x = y + 1$.

An abstract domain, moreover, supports a variety of operations for updating the symbolic state as a program is executed. These include operations for updating the value of variables as well as for joining states from multiple control flow branches together. Lastly an abstract domain implements a *widening* operation which allows us to obtain a fixed-point even in cases where the abstract state would otherwise change indefinitely. We can see all of these operations in action in the following where we use abstract interpretation with the polyhedra abstract domain to obtain constraints on the local variables in Example 1.

We start out with an abstraction where $\texttt{i} = \top$ indicating that `i` can have any value. After executing the statement `i := 0` we update our abstraction to reflect $\texttt{i} = 0$. After executing the first iteration of the loop our abstraction reflects $\texttt{i} = 1$. We have discovered a second state we can be in at the start of an iteration. In order to start the next iteration we need to join these two states together. The abstract domain provides a *least upper bound* (lub) operation for this purpose. A possible lub for our two states would be $0 \leq \texttt{i} \leq 1$. We can continue executing the iteration to obtain the abstract state $1 \leq \texttt{i} \leq 2$ and compute another lub. In this fashion we successively generate the abstract states $0 \leq \texttt{i} \leq 2$, $0 \leq \texttt{i} \leq 3$, ... before executing each iteration.

To ensure that we always reach a fixed-point eventually we use the widening operation after a predefined threshold of updates on the loop entry state has been reached. This operation will identify the upper bound on `i` as unstable and remove it resulting in the abstraction $0 \leq \texttt{i}$ which does not change in successive iterations. Note that each time we start a new iterations we have to check that the loop condition holds resulting in the abstract state

$$0 \leq \texttt{i} < \texttt{image.length} \tag{2.1}$$

inside the loop.

Abstract interpretation techniques have to be *sound*, i.e. they have to account for all possible behavior of a program. In turn they sacrifice *precision*, i.e. they may find behavior that does not exist in concrete executions of a program. For example polyhedra is not able to capture parity information so if we were to increment `i` by 2 in each iteration in Example 1 we would still obtain the same final constraints as above. Analogous techniques for obtaining under-approximate constraints, i.e. constraints that are only satisfied by states reached during the concrete program execution, exist (e.g. [11, 21, 27]).

2.5 Inference of Permission Pre- and Postconditions for Loops

A technique proposed by Dohrau et al. [17] automatically determines the overall amount of permissions needed to execute a loop in an array program as well as how the amount of held permission changes after the entire loop has been executed. Effectively this analysis generates permission pre- and postconditions for loops.

This technique uses abstract interpretation to obtain constraints on the values of numerical variables that are modified during the execution of the loop but crucially gets by without the need for an abstract domain that can handle permissions and, thus, without introducing additional imprecision beyond that of the initial abstract interpretation. It achieves this by first calculating a set of permissions that is sufficient for executing a single loop iteration, parameterized over the local variables that are modified during the execution of the loop. Analogously it analyzes by what amount the held permissions change over that iteration, i.e. what permissions are `inhale`d or `exhale`d. For example during a single iteration of the loop in Example 1 we require **acc**`(image[i],` **write**`)` and we lose **acc**`(image[i],` $\frac{1}{2}$`)`.

It is clear that if a heap location is only written and read it suffices to, for each location, have the maximum amount of permissions that is required in any iteration of the loop. The same is true in other cases where the amount of permissions held for a heap location does not change when a single loop iteration is executed (e.g. if a method call gives back the same permissions it received when it was called). In order to handle cases where that is not the case the analysis introduces a soundness condition stating that executing any two iterations in succession must require no more permissions than the maximum amount required in either iteration. Intuitively this can be interpreted as requiring that a location for which we lose permissions in iteration i is only accessed in that same iteration. By over-approximating the loop iterations we execute using the constraints we obtained via abstract interpretation we can calculate the amount of permissions needed to execute the entire loop. For Example 1, using the results from (2.1), we obtain the following expression for each location `image[q]` in our loop precondition:

$$\max_{0 \leq \texttt{i} < \texttt{image.length}} q = \texttt{i} \ ? \ \textbf{write} : \textbf{none}$$

similarly we obtain

$$\max_{0 \leq \texttt{i} < \texttt{image.length}} q = \texttt{i} \ ? \ \frac{1}{2} : \textbf{none}$$

as the amount of permissions to `image[q]` we lose over the execution of the entire loop.

$$
\begin{aligned}
&wlp(\textbf{assert } A, Q) = A \wedge Q \qquad wlp(\textbf{assume } A, Q) = A \longrightarrow Q \\
&wlp(x = e, Q) = Q[e/x] \qquad wlp(s_1; s_2, Q) = wlp(s_1, wlp(s_2, Q)) \\
&wlp(\textbf{if}(b)\{s_1\}\textbf{else}\{s_2\}, Q) = (b \longrightarrow wlp(s_1, Q)) \wedge (\neg b \longrightarrow wlp(s_2, Q)) \\
&wlp(\textbf{while}(b) \textbf{ invariant } A\ \{s\}, Q) = \\
&\qquad A \wedge \forall \vec{y}.((A \wedge b \longrightarrow wlp(s, A)) \wedge (A \wedge \neg b \longrightarrow Q))[\vec{y}/\vec{x}]
\end{aligned}
$$

Figure 2.1: Rules for wlp calculation

Dohrau et al., then, propose a mechanism for eliminating these unbounded maxima resulting in closed form expression we can use in our pre- and postconditions:

$$\forall q.\textbf{acc}(\texttt{image[}q\texttt{]}, 0 \leq q < \texttt{image.length}\ ?\ \textbf{write} : \textbf{none}) \tag{2.2}$$

from this precondition we can subtract the amount of permissions we give away to obtain the postcondition:

$$
\begin{aligned}
&\forall q. \quad \textbf{acc}(\texttt{image[}q\texttt{]}, 0 \leq q < \texttt{image.length}\ ?\ \textbf{write} : \textbf{none}) \\
&\quad - \quad \textbf{acc}(\texttt{image[}q\texttt{]}, 0 \leq q < \texttt{image.length}\ ?\ \frac{1}{2} : \textbf{none}) \\
\iff &\forall q. \quad \textbf{acc}(\texttt{image[}q\texttt{]}, 0 \leq q < \texttt{image.length}\ ?\ \frac{1}{2} : \textbf{none})
\end{aligned}
\tag{2.3}
$$

2.6 Weakest Liberal Preconditions

One approach used for program verification is that of generating a *weakest liberal precondition* (wlp) [4, 14]. This approach allows us to prove partial correctness of a program, i.e. that every *terminating* execution ends in a state that satisfies the postcondition.

The basic idea of this approach is to start from the postcondition Q and go through the program backwards. In each step we try to find a condition that has to hold before that particular statement in order to ensure that the postcondition holds in the end. We denote the weakest liberal precondition by $wlp(\text{program}, Q)$. Once we reach the beginning of a program we can check that the precondition P of the program is at least as strong as the wlp we calculated, i.e. that

$$P \longrightarrow wlp(\text{program}, Q).$$

Calculating the weakest liberal precondition can be done using a set of rules. E.g. for an assignment `i = 0` we simply replace occurrences of `i` with 0: $wlp(\texttt{i = 0}, Q) = Q[0/\texttt{i}]$. So for the postcondition $Q \equiv (\texttt{i} = 0)$ we get the wlp $0 = 0$ which is, of course, equivalent to true. The full set of rules for the language used in this thesis is given in Fig. 2.1. The rule for loops is an

inductive proof based on an invariant. The wlp consists of three parts: the first conjunct requires us to show that the invariant holds before we enter the loop, i.e. the base case of the induction. We then show the step case of the induction. This is done by quantifying over all possible values of the variables $\vec{x}$ that change during the execution of the loop and showing

$$A \wedge b \longrightarrow wlp(s, A),$$

i.e. that every iteration of the loop preserves the invariant A. We have then inductively shown that A is true in any state that can be reached by executing arbitrarily many iterations of the loop. As a final step we have to show that when we exit the loop (i.e. b is false) A is strong enough to obtain the postcondition Q.

Chapter 3

Progressive Loop Invariants

In this chapter we will introduce the novel concept of progressive loop invariants and design a mechanism for automatically inferring them. Applications of progressive loop invariants will be discussed in Chap. 4.

Whereas "regular" loop invariants provide constraints that hold in any state that can be reached by executing arbitrarily many loop iterations the idea behind *progressive loop invariants* is that we stop the execution of the loop in some state s and try to find constraints that were satisfied in all s' we reached in previous iterations (*past progressive invariants*) or that will be satisfied in all s' we will reach in future iterations (*future progressive invariants*). These can, furthermore, be either over-approximative or under-approximative. Over-approximative progressive loop invariants can be interpreted as a "*potentially* precedes"-/"*potentially* succeeds"-relation between a "current" state s and another state s' whereas an under-approximative progressive loop invariant would be interpreted as a "*must* precede"-/"*must* succeed"-relation.

In this chapter we will discuss how we can use an over-approximative numerical analysis to infer over-approximative progressive loop invariants. It is, however, in principal possible to infer under-approximative progressive loop invariants using a similar technique and an under-approximative numerical analysis.

As an example for what progressive loop invariants look like consider pausing the execution of the loop in Example 1 after n iterations (i.e. in a state s where $\texttt{i} = n$). A possible past progressive invariant would be

$$0 \leq \texttt{i} \leq n \tag{3.1}$$

and a possible future progressive invariant would be

$$n \leq \texttt{i} < \texttt{image.length}. \tag{3.2}$$

Note that these progressive loop invariants are chosen in such a way that both the past progressive invariant and the future progressive invariant are satisfied in the current state—s. We call progressive loop invariants that are

```
1  var i′
2  var i := i′
3  // increase brightness
4  image[i] := image[i] + 10
5
6  // render
7  exhale acc(image[i], 1/2)
8
9  i += 1
```

Figure 3.1: Single iteration of the loop in Example 1. The copy $\texttt{i}'$ of $\texttt{i}$ is intruduced in the highlighted lines.

true only for "true predecessors/successors" *strict* progressive loop invariants. We will discuss how strict progressive loop invariants can be constructed from regular progressive loop invariants in Sec. 3.3.

3.1 Constraint Generation

Inferring progressive loop invariants is done in three steps:

(1) We use abstract interpretation to obtain constraints on local variables, i.e. information about a single state.

(2) We then use this information to add some instrumentation to the program and run a second abstract interpretation giving us information about a second state occurring some time after the state from step (1) is reached.

(3) We can then use the constraints we learned about these two states to construct progressive loop invariants.

The intuitive idea behind this technique is that we simulate stopping the execution of the loop after $n \geq 0$ iterations in state s. This is done by introducing copies of all local variables in s before the loop is entered. We will denote the copy of variable $\texttt{i}$ as $\texttt{i}'$. To illustrate how this form of instrumentation can help us relate two states to each other consider the instrumentation for a single iteration of Example 1 as shown in Fig. 3.1 (note that this figure is shown for illustrative purposes and does not represent the instrumentation used by the inference algorithm). Running polyhedra (we will be using polyhedra in our examples but the techniques described work independently of the abstract domain used) on this program results in the constraint $\texttt{i} = \texttt{i}' + 1$. Hence, it allows us to relate the final value of $\texttt{i}$ back to its initial value. Note that without the copy of $\texttt{i}$ this initial value is "forgotten" by the abstract interpretation resulting in the final state $\texttt{i} = \top$.

```
1  var i′
2  assume 0 ≤ i′ < image.length
3  var i := i′
4  while (i < image.length)
5    invariant ∀q ∈ [0,image.length).image[q] = q < i ?
6      original_image[q] + 10 : original_image[q]
7  {
8    // increase brightness
9    image[i] := image[i] + 10
10
11   // render
12   exhale acc(image[i], 1/2)
13
14   i += 1
15 }
```

Figure 3.2: Example for an instrumented version of Example 1. The highlighted lines show the instrumentation that is added base on the constraints $0 \leq \texttt{i} < \texttt{image.length}$ that we know hold inside the loop based on step (1) of the inference.

Similarly, the inference algorithm for progressive loop invariants introduces $\texttt{i}'$ before the loop as shown in Fig. 3.2 allowing us to relate the states encountered during the execution of the loop back to the initial state before the loop. Since we are only interested in s that can be reached by executing arbitrarily many iterations of the loop we can assume that $\texttt{i}'$ satisfies the constraints inside the loop determined in step (1). We then run an abstract interpretation on the instrumented version of the program in step (2). This abstract interpretation simulates executing the loop starting from s. Accordingly, the abstract states we obtain abstract over all states we can reach from s, i.e. they give us constraints that relate s to its successors. For Example 1 these will look like $0 \leq \texttt{i}' \leq \texttt{i} < \texttt{image.length}$.

Based on this construction we can see that, for a fixed s (and corresponding fixed $\texttt{i}'$), these constraints give a future progressive invariant. For example fixing $\texttt{i}'$ to n results in the constraint $n \leq \texttt{i} < \texttt{image.length}$ for Example 1 which we already identified as a possible future progressive invariant in (3.2). As we will see next the constraints from (2) actually over-approximate the successor/predecessor-relation between two states s and s' and, therefore, also allow us to derive a past progressive invariant.

3.2 Construction of Progressive Loop Invariants

In this section we will formalize the concept of progressive loop invariants and formally show that they can be constructed from the constraints generated as described in the previous section.

As described above progressive loop invariants relate a current state s to another state s'. To make the distinction between these states clearer we will now use the notation $\lceil e \rfloor_s$ to denote the value of expression e in state s. Using this notation the constraint $0 \leq \texttt{i}' \leq \texttt{i} < \texttt{image.length}$ we derived in the previous section will be expressed as

$$0 \leq \lceil \texttt{i} \rfloor_s \leq \lceil \texttt{i} \rfloor_{s'} < \texttt{image.length}. \tag{3.3}$$

We will, moreover, split these constraints into a *progressive* ($P(s, s')$) and an *invariant* ($I(s)$) part such that the progressive part contains only constraints that relate the two states, s and s', to each other and the invariant part contains only constraints that relate a state to constants. Any set of constraints can then be rewritten to the form $P(s, s') \wedge I(s) \wedge I(s')$. E.g. for the constraints from (3.3) we get

$$P(s, s') := \lceil \texttt{i} \rfloor_s \leq \lceil \texttt{i} \rfloor_{s'}$$

and

$$I(s) := 0 \leq \lceil \texttt{i} \rfloor_s < \texttt{image.length}.$$

Theorem 1. *The constraints generated in step* (2) *satisfy the following properties:*

1. $\forall s_0 \in S_0 . I(s_0)$*, where* S_0 *is the set of states the program may be in before entering the loop.*
2. $\forall s, s' . \mathrm{suc}^*(s, s') \xrightarrow{\boxed{1}} \Big(I(s) \xrightarrow{\boxed{2}} P(s, s') \wedge I(s') \Big)$*, where* $\mathrm{suc}(s, s')$ *is the successor relation between* s *and* s'*, indicating that* s' *can be reached from* s *by executing a single loop iteration, and* $\mathrm{suc}^*(s, s')$ *is its reflexive transitive closure. This property is the analog of the inductivity property* $(\forall s, s' . \mathrm{suc}(s, s') \wedge I(s) \longrightarrow I(s'))$ *for regular inductive invariants.*

Proof (Sketch). Property 1 follows from the fact that $I(\cdot)$ is generated using abstract interpretation and, thus, has to at least over-approximate S_0. For property 2 we observe that executing k iterations of the instrumented loop gives us the property for $\mathrm{suc}^k(s, s')$ (k steps in suc) and, therefore, for $\mathrm{suc}^*(s, s')$ after the numerical analysis generalizes over the execution of the entire loop. □

Lemma 1. $I(s)$ *holds for every* s *that is encountered after arbitrarily many iterations during the concrete execution of the loop.*

Proof. For each $s_0 \in S_0$ we instantiate property 2 for $s = s_0$. Property 1 gives us $I(s)$ so we get $I(s')$ for every s' that is reachable by executing the loop starting form s_0. □

3.2.1 Future Progressive Invariants

The future progressive invariant $\overrightarrow{P}_s$ of a state s over-approximates the states we can reach by executing arbitrarily many loop iterations starting from s. This is exactly what we simulate during step (2), i.e. the constraints we obtain from step (2) give a future progressive invariant. Based on the components we identified above we formally define $\overrightarrow{P}_s$ as:

$$\overrightarrow{P}_s(s') := P(s, s') \wedge I(s) \wedge I(s') \tag{3.4}$$

For example the future progressive invariant based on (3.3) in the state where i is n would be:

$$\overrightarrow{P}_s(s') = n \leq \lceil i \rfloor_{s'} \wedge 0 \leq n < \texttt{image.length} \wedge 0 \leq \lceil i \rfloor_{s'} < \texttt{image.length}$$

Assuming that $\texttt{image.length} \geq n$ this can be simplified to the future progressive invariant we already identified in (3.2):

$$n \leq \lceil i \rfloor_{s'} < \texttt{image.length}$$

Let us now rewrite implication $\boxed{2}$ in property 2 by conjoining its left-hand-side to its right-hand-side:

$$I(s) \longrightarrow P(s, s') \wedge I(s) \wedge I(s') \iff I(s) \longrightarrow \overrightarrow{P}_s(s')$$

Looking at the entirety of property 2 we can, therefore, derive

$$\forall s, s'. \, \mathrm{suc}^*(s, s') \wedge I(s) \longrightarrow \overrightarrow{P}_s(s'). \tag{3.5}$$

Since we typically only instantiate $\overrightarrow{P}_s$ for s that we actually encounter during the execution of the loop, i.e. $I(s)$ is true, we can drop that term and obtain the property

$$\forall s, s'. \, \mathrm{suc}^*(s, s') \longrightarrow \overrightarrow{P}_s(s')$$

indicating that the future progressive invariant is true in at least every state that can potentially be encountered in the future.

3.2.2 Past Progressive Invariants

Analogously to future progressive invariants past progressive invariants over-approximate the the states we were in before we reached s.We construct a past progressive invariant by switching the position of s and s'. A hint for why this works is that in our construction s occurs before s' so switching their positions has to at least give us some s' we reached before s. We will

prove that we over-approximate the predecessors states of s below. Formally, we define past progressive invariants as:

$$\overleftarrow{P}_s(s') := P(s', s) \wedge I(s') \wedge I(s) \tag{3.6}$$

For example the past progressive invariant based on (3.3) in state s where i is n would be:

$$\overleftarrow{P}_s(s') = \lceil i \rfloor_{s'} \leq n \wedge 0 \leq \lceil i \rfloor_{s'} < \texttt{image.length} \wedge 0 \leq n < \texttt{image.length}.$$

Again assuming that $\texttt{image.length} > n$ this can be simplified to

$$0 \leq \lceil i \rfloor_{s'} \leq n$$

which is exactly the past progressive invariant we already identified in (3.1).

Analogously to what we did for future progressive invariants we can conjoin the left-hand-side of implication $\boxed{2}$ to obtain

$$\forall s, s'.\, \mathrm{suc}^*(s', s) \wedge I(s') \longrightarrow \overleftarrow{P}_s(s') \tag{3.7}$$

from property 2. The past progressive invariant is, therefore, true in at least every potential predecessor state ($\mathrm{suc}^*(s', s)$) that is actually encountered during the execution of the loop ($I(s')$), i.e. it over-approximates the predecessor states of s.

3.3 Strict Progressive Loop Invariants

In Sec. 3.2 we saw how we can automatically infer progressive loop invariants. These over-approximate the reflexive transitive closure $\mathrm{suc}^*(s, s')$ of the predecessor/successor relation we observe during concrete executions. Because of this underlying reflexivity progressive loop invariants are true in the "current state," i.e. $\overrightarrow{P}_s(s)$ and $\overleftarrow{P}_s(s)$ are always true. As we will see in Chap. 4 this is, in many cases, exactly what we want. There are, however, also cases in which we are only interested in "true predecessors/successors." We, therefore, introduce the strict progressive loop invariants $\overrightarrow{\overrightarrow{P}}_s(s')$ and $\overleftarrow{\overleftarrow{P}}_s(s')$ which, respectively, over-approximate the transitive closures $\mathrm{suc}^+(s, s')$ and $\mathrm{suc}^+(s', s)$ of the concrete predecessor/successor relation. We will look at two approaches that allow us to obtain strict progressive loop invariants from regular progressive loop invariants next.

3.3.1 Using Iteration Counters

A simple method for constructing strict progressive loop invariants is to introduce an iteration counter $\mathtt{c}$ to the original program. We then define

$$\overrightarrow{\overrightarrow{P}}^{IC}_s(s') := \overrightarrow{P}_s(s') \wedge \lceil \mathtt{c} \rfloor_s < \lceil \mathtt{c} \rfloor_{s'} \tag{3.8}$$

and

$$\overset{IC}{\underset{s}{\Longleftarrow P}}(s') := \overset{\leftarrow}{P}_s(s') \wedge \lceil \mathtt{c} \rfloor_s > \lceil \mathtt{c} \rfloor_{s'}, \tag{3.9}$$

which ensure that we only consider predecessors/successors that occur at least one iteration before/after the current state.

The precision of this approach is limited by the precision of the constraints we obtain using abstract interpretation. E.g. for an `i` that grows exponentially we can only get linear constraints using polyhedra, i.e. we might get the constraint $\mathtt{i} \geq \mathtt{c}$ which in the case of the strict future progressive invariant allows us to show $\lceil \mathtt{i} \rfloor_s < \lceil \mathtt{i} \rfloor_s$ but not $\lceil 2\mathtt{i} \rfloor_s \leq \lceil \mathtt{i} \rfloor_s$. The advantage of this approach is, however, that if the original constraints were within Presburger arithmetic (as is the case if we use polyhedra) then the strict progressive loop invariants will also be within Presburger arithmetic. This will later allow us to eliminate quantifiers and maximum expressions using the techniques described by Cooper [8] and Dohrau et al. [17]. We, therefore, often prefer the approach described here to the one described in Sec. 3.3.2 if the additional precision can be sacrificed.

3.3.2 Using a Data Flow Analysis

The idea behind the approach described in this section is that we can determine the values of the local variables in the post-state of a loop iteration based on their values in the pre-state relatively precisely. We do this using a data-flow analysis (e.g. [3]). E.g. if `i` is multiplied by 2 in each loop iteration this analysis would tell us that the value of `i` in the post-state is equal to that of $2\mathtt{i}$ in the pre-state. We will denote the vector of expressions we obtain by applying this analysis for all local variables $\vec{x}$ as $\vec{e}$. This allows us to essentially calculate the regular future progressive invariant for direct successors of s. Formally we define

$$\overset{DF}{\underset{s}{\Longrightarrow P}}(s') := P(s, s')[\lceil \vec{e} \rfloor_s / \lceil \vec{x} \rfloor_s] \wedge I(s) \wedge I(s') \tag{3.10}$$

and analogously

$$\overset{DF}{\underset{s}{\Longleftarrow P}}(s') := P(s', s)[\lceil \vec{e} \rfloor_{s'} / \lceil \vec{x} \rfloor_{s'}] \wedge I(s') \wedge I(s). \tag{3.11}$$

Proving that these over-approximate suc^+ can be done analogously to the proofs presented in Sec. 3.2.1 and Sec. 3.2.2.

We can see that for a loop where `i` grows exponentially $P(\cdot, \cdot)$ will include the constraint $\lceil \mathtt{i} \rfloor_s \leq \lceil \mathtt{i} \rfloor_{s'}$. Consequently the strict future progressive invariant will include the constraint $\lceil 2\mathtt{i} \rfloor_s \leq \lceil \mathtt{i} \rfloor_{s'}$. This is more precise than the strict future progressive invariant we obtained in Sec. 3.3.1. However, we can also see that e.g. if `i` gets assigned $\mathtt{j} \star \mathtt{i}$, where `j` is also a local

variable, the strict progressive loop invariants $\overset{\Longrightarrow DF}{P}_s(s')$ and $\overset{\Longleftarrow DF}{P}_s(s')$ exceed Presburger arithmetic. As a result we are not easily able to eliminate quantifiers containing this kind of strict progressive loop invariants.

Now that we have seen how we can automatically construct progressive loop invariants we will look at some of their applications next.

Chapter 4

Applications

4.1 Evolution of Held Permissions

In this section, we will use progressive loop invariants to infer framing invariants which will allow us to verify Example 1. These framing invariants have to capture how the set of permissions we hold changes as we progress through the loop. In particular they have to be 1) *sufficient*, i.e. after executing the first n iterations of the loop they must still contain enough permissions to finish executing the loop, and 2) be *inductive*, i.e. if we execute a single loop iteration starting from any state s that satisfies the invariant we have to end up in a state s' that also satisfies the invariant. In practice inductivity for framing invariants requires that when we compare the invariants before and after a single loop iteration the amount of permissions the invariant requires for each heap location must decrease by at least the the amount that is lost during that iteration.

We construct such an invariant by approximating how the permissions for each heap location change when the loop is executed up to an arbitrary iteration. This change is, then, added/subtracted from the precondition generated by the analysis proposed by Dohrau et. al. (cf. Sec. 2.5).

4.1.1 Exhales

In order to construct invariants we consider stopping the execution of the loop after n iterations in state s and the set of permissions we need to finish executing the loop as well as the set of permissions we have given away up to that point. The technique described by Dohrau et. al. (cf. Sec. 2.5) allows us to generalize the permissions we need in a single iteration over a set of loop iterations (represented by their pre-states) satisfying an over-approximative invariant. Similarly we can use the same technique to generalize over the set of iterations satisfying the future progressive invariant which over-approximates the iterations we still need to execute. In this fashion we obtain a set of permissions that is required to finish executing

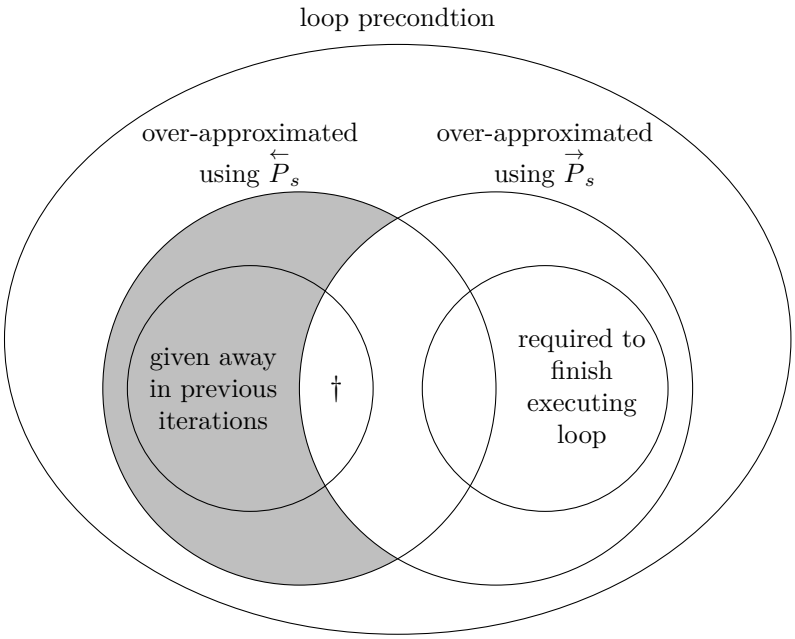

Figure 4.1: Relationships between sets of permissions after executing the first n iterations of a loop when no permissions are gained. The shaded area shows the permissions that are subtracted from the loop's precondition to obtain an invariant. The area marked by † has to be empty in order for the invariant to be inductive. This is guaranteed when the soundness condition in (4.7) is satisfied.

the loop. Dohrau et al., moreover, describe a technique for generalizing the permission-loss in a single iteration over all iterations of the loop. We can, analogously, apply this technique over the past progressive invariant to over-approximate the permissions we have already given away when we reach state s. Fig. 4.1 shows these sets in relation to each other for a program that never gains any permissions.

A sufficient invariant has to, by definition, at least contain the permissions needed to finish executing the loop. We can also see that it is a necessary condition for it not to include any permissions that have already been given away in order to be inductive: if the invariant still requires some of these permissions there has to either have been an iteration in the past where they were given away in the concrete execution but the invariant still required them afterwards or the invariant started requiring them again after they were already given away in an earlier iteration, i.e. they were "regained."

As mentioned before we use the loop precondition (which is also shown in the diagram) as the basis of our invariant. In the case where permissions are lost we calculate the set of permissions we may already have given away (based

on the past progressive invariant) and that we are guaranteed *not* to need in the future (based on the future progressive invariant). The corresponding set is shaded in the diagram. These permissions are, then, subtracted from the loop precondition. We can see that the resulting invariant contains all permissions needed to finish executing the loop. It is however not guaranteed that we do not require any permissions that have already been given away, i.e. it is not guaranteed that the constructed invariant is inductive. We will later introduce a soundness condition under which the invariant is inductive and, hence, the area marked by † in the diagram is empty.

We analyze the permissions for each heap location `a[q]` separately. Any iteration in which permissions to this location are lost can then, w.l.o.g., be summarized into a single statement **exhale acc**`(a[e], ` α_e`)`, where e is some expression and $\alpha_e \geq 0$. As described above we approximate the set of permissions $l(s)$ that have already been given away by the time we reach state s as:

$$l(s) := \max_{s'' | \overleftarrow{P}_s(s'') \wedge \neg \overrightarrow{P}_s(s'')} (\lceil e \rfloor_{s''} = q \ ? \ \alpha_e : \textbf{none}) \tag{4.1}$$

Note that in order to be able to summarize the `exhales` across multiple iterations into a single max term as we do here, we rely on the same soundness condition as the analysis described by Dohrau et al. This soundness condition states that executing any two loop iterations in succession requires no more permissions than the maximum amount needed to execute them separately. This means that if we `exhale` permissions to some array location l in iteration i we cannot access l in any other iteration (cf. Sec. 2.5), e.g. it does not allow us run this analysis on a program that reads the value of `a[0]` in one iteration and `exhale`s permissions to `a[0]` in a different iteration. In addition to requiring this soundness condition here we also inherit it directly since we rely on the soundness of the loop precondition from which we subtract $l(s)$ to form an invariant. In Sec. 4.2 we will see how progressive loop invariants can help us to, in some cases, overcome this limitation.
For Example 1 and the progressive loop invariants

$$\begin{aligned}\overleftarrow{P}_s(s'') &:= 0 \leq \lceil \texttt{i} \rfloor_{s''} \leq \lceil \texttt{i} \rfloor_s \wedge \lceil \texttt{i} \rfloor_{s''} < \texttt{image.length} \\ &= 0 \leq \lceil \texttt{i} \rfloor_{s''} \leq \texttt{i} \wedge \lceil \texttt{i} \rfloor_{s''} < \texttt{image.length}\end{aligned}$$

and

$$\begin{aligned}\overrightarrow{P}_s(s'') &:= \lceil \texttt{i} \rfloor_s \leq \lceil \texttt{i} \rfloor_{s''} < \texttt{image.length} \\ &= \texttt{i} \leq \lceil \texttt{i} \rfloor_{s''} < \texttt{image.length}\end{aligned}$$

we calculated in Sec. 3 we get

$$l(s) := 0 \leq q < \lceil \texttt{i} \rfloor_s \ ? \ \frac{1}{2} : \textbf{none}. \tag{4.2}$$

We subtract $l(s)$ from the loop precondition (2.2) to obtain the invariant

$$\forall q.\mathbf{acc}(\texttt{a[q]}, (0 \leq q < \texttt{image.length}\ ?\ \mathbf{write} : \mathbf{none}) - (0 \leq q < \texttt{i}\ ?\ \frac{1}{2} : \mathbf{none}))$$

which allows us to verify the program from Example 1.

Inductivity

By construction the inferred invariant is sufficient but not necessarily inductive. In this section we will come up with a soundness condition we can automatically check during the runtime of the inference. If this condition holds our invariant is guaranteed to be inductive and we can use it to verify the method. If the soundness condition does not hold we generate false as the invariant. Another option would have been to produce the invariant as described above but warn the user that it might not be inductive. Besides resulting in specifications that allow us to verify the loop the advantage of the former approach is that a user who is just looking at the extended program can very easily spot an invariant that consists of only the false literal and this is a strong indication for a user that something needs their attention. Moreover, the current implementation additionally provides the invariant we would otherwise construct in a comment.

The inductivity property states that for arbitrary states s and s', where s' is reachable from s by executing a single loop iteration, i.e. $\mathrm{suc}(s, s')$ holds, the invariant requires at least α_e fewer permissions to $\texttt{a}[e]$ from s' than it requires from s. We are, therefore, interested in lower-bounding the difference between two instantiation of our loop invariant. After the loop preconditions cancel out we are left with $l(s') - l(s)$ which expands to

$$\begin{aligned} &\max_{s'' \mid \overleftarrow{P}_{s'}(s'') \wedge \neg \overrightarrow{P}_{s'}(s'')} (\lceil e \rfloor_{s''} = q\ ?\ \alpha_e : \mathbf{none}) - \\ &\max_{s'' \mid \overleftarrow{P}_{s}(s'') \wedge \neg \overrightarrow{P}_{s}(s'')} (\lceil e \rfloor_{s''} = q\ ?\ \alpha_e : \mathbf{none})\,. \end{aligned} \tag{4.3}$$

To show that the invariant we generate is inductive an additional soundness condition must, then, be strong enough to allow us to show that the lower bound

$$l(s') - l(s) \geq (\lceil e \rfloor_s = q\ ?\ \alpha_e : \mathbf{none}) \tag{4.4}$$

holds.

We observe that this bound is always violated if (4.3) is negative because α_e has to be non-negative. We formulate the following the soundness

condition under which (4.3) is non-negative:

$$\begin{aligned}&\forall s, s'.\alpha_e > \textbf{none} \wedge \mathrm{suc}(s, s') \longrightarrow \\ &\forall s''.\Big(\underbrace{\overleftarrow{P}_s(s'') \wedge \neg\overrightarrow{P}_s(s'')}_{\text{right max positive}} \wedge \underbrace{\neg\Big(\overleftarrow{P}_{s'}(s'') \wedge \neg\overrightarrow{P}_{s'}(s'')\Big)}_{\text{left max} = \textbf{none}} \longrightarrow \lceil e \rfloor_{s''} \neq q\Big)\end{aligned} \tag{4.5}$$

The intuitive interpretation of this condition is that we check that no permissions are ever "regained." Note that the set of concrete states that lie in the past only grows and the set of concrete states we will encounter in the future only shrinks as we progress through the loop. If the same is true about our progressive loop invariants (which over-approximate these sets) we can see that the soundness condition (4.5) is always satisfied. In Def. 1 we formally define this notion of *monotonicity* for progressive invariants. Some abstract domains, including polyhedra, guarantee that the progressive loop invariants we generate with their aid satisfy this property. In cases where this property is not guaranteed we can still check a version of (4.5) during the inference's runtime as we will see later.

Definition 1. *Monotonicity*
Let $\overleftarrow{S}_s := \left\{ s' \middle| \overleftarrow{P}_s(s') \right\}$ *and* $\overrightarrow{S}_s := \left\{ s' \middle| \overrightarrow{P}_s(s') \right\}$ *be the set of states satisfying the past progressive invariant and future progressive invariant of state* s *respectively. A family of past progressive invariants is called monotone iff*

$$\forall s, s'.\, \mathrm{suc}(s, s') \longrightarrow \overleftarrow{S}_s \subseteq \overleftarrow{S}_{s'}.$$

A family of future progressive invariants is called monotone iff

$$\forall s, s'.\, \mathrm{suc}(s, s') \wedge I(s) \longrightarrow \overrightarrow{S}_s \supseteq \overrightarrow{S}_{s'}$$

Some elaborations explaining under which circumstances the progressive loop invariants we generate using polyhedra are monotone are given in Appendix A. Even when the soundness condition (4.5) is satisfied we need an additional soundness condition to be satisfied in order to show that the lower bound (4.4) holds. We derive this soundness condition by lower-bounding (4.3) as

$$
\begin{aligned}
& l(s') - l(s) \\
= & \max_{s''} \overleftarrow{P}_{s'}(s'') \wedge \neg \overrightarrow{P}_{s'}(s'') \wedge \neg \left(\overleftarrow{P}_{s}(s'') \wedge \neg \overrightarrow{P}_{s}(s'') \right) \wedge \lceil e \rfloor_{s''} = q && ? \ \alpha_e : \textbf{none} \\
= & \max_{s''} \overleftarrow{P}_{s'}(s'') \wedge \neg \overrightarrow{P}_{s'}(s'') \wedge \left(\neg \overleftarrow{P}_{s}(s'') \vee \overrightarrow{P}_{s}(s'') \right) \wedge \lceil e \rfloor_{s''} = q && ? \ \alpha_e : \textbf{none} \\
\overset{*}{\geq} & \max_{s''} \mathrm{suc}^*(s'', s') \wedge I(s'') \wedge \neg \overrightarrow{P}_{s'}(s'') \wedge \left(\neg \overleftarrow{P}_{s}(s'') \vee \mathrm{suc}^*(s, s'') \right) \wedge \lceil e \rfloor_{s''} = q && ? \ \alpha_e : \textbf{none} \\
\overset{**}{\geq} & \max_{s''} \mathrm{suc}^*(s'', s') \wedge I(s'') \wedge \neg \overrightarrow{P}_{s'}(s'') \wedge \left(\neg I(s'') \vee \mathrm{suc}^*(s, s'') \right) \wedge \lceil e \rfloor_{s''} = q && ? \ \alpha_e : \textbf{none} \\
= & \max_{s''} \mathrm{suc}^*(s'', s') \wedge I(s'') \wedge \neg \overrightarrow{P}_{s'}(s'') \wedge \mathrm{suc}^*(s, s'') \wedge \lceil e \rfloor_{s''} = q && ? \ \alpha_e : \textbf{none} \\
\geq & \ \mathrm{suc}^*(s, s') \wedge I(s) \wedge \neg \overrightarrow{P}_{s'}(s) \wedge \mathrm{suc}^*(s, s) \wedge \lceil e \rfloor_{s} = q && ? \ \alpha_e : \textbf{none} \\
= & \ \neg \overrightarrow{P}_{s'}(s) \wedge \lceil e \rfloor_{s} = q && ? \ \alpha_e : \textbf{none}
\end{aligned}
$$

* under-approximate $\overleftarrow{P}_{s'}(s'')$ as $\mathrm{suc}^*(s'', s') \wedge I(s'')$ and $\overrightarrow{P}_{s}(s'')$ as $\mathrm{suc}^*(s, s'')$ ($I(s)$ is true since we encounter s during the execution of the loop)

** over-approximate $\overleftarrow{P}_{s}(s'')$ as $I(s'')$

We can, therefore, ensure that the lower-bound (4.4) holds if

$$
\left(\neg \overrightarrow{P}_{s'}(s) \wedge \lceil e \rfloor_{s} = q \ ? \ \alpha_e : \textbf{none} \right) \geq \left(\lceil e \rfloor_{s} = q \ ? \ \alpha_e : \textbf{none} \right)
$$

for any iteration. For any pair of states s and s' that directly succeed each other we get the following additional soundness condition:

$$
\forall q. \forall s, s'. \alpha_e = \textbf{none} \vee \left(\mathrm{suc}(s, s') \wedge \lceil e \rfloor_{s} = q \longrightarrow \neg \overrightarrow{P}_{s'}(s) \right) \tag{4.6}
$$

Since this condition relies on the progressive loop invariants we generate (which the user does not have ahead of time) we want to be able to automatically check this condition at the runtime of the inference. We can do this by restricting the progressive loop invariants to immediate predecessors/successors which we can achieve using a similar approach to that used when constructing $\overleftarrow{P}^{IC}_{s}(s')$ but we constrain the iteration counters to $\lceil \mathrm{c} + 1 \rfloor_{s} = \lceil \mathrm{c} \rfloor_{s'}$ instead of $\lceil \mathrm{c} \rfloor_{s} < \lceil \mathrm{c} \rfloor_{s'}$. We can, then, strengthen the soundness condition (4.6) to:

$$
\forall q. \forall s, s'. \alpha_e = \textbf{none} \vee \left(\overleftarrow{P}_{s'}(s) \wedge \lceil \mathrm{c} + 1 \rfloor_{s} = \lceil \mathrm{c} \rfloor_{s'} \wedge \lceil e \rfloor_{s} = q \longrightarrow \neg \overrightarrow{P}_{s'}(s) \right) \tag{4.7}
$$

Intuitively this condition states that we should be able to tell with certainty whether some state lies in the future or in the past. Note that even after introducing iteration counters we may still be unable to distinguish two states that are identical except in their iteration counters, e.g. if the constraints do not contain the branch counters at all. The soundness condition would be false in such cases.

Analogously we can derive the following soundness condition from (4.5):

$$\begin{aligned}&\forall s, s'.\alpha_e > \mathbf{none} \wedge \overleftarrow{P}_{s'}(s) \wedge \mathtt{c} = 1 \longrightarrow \\ &\forall s''.\left(\overleftarrow{P}_s(s'') \wedge \neg\overrightarrow{P}_s(s'') \wedge \neg\left(\overleftarrow{P}_{s'}(s'') \wedge \neg\overrightarrow{P}_{s'}(s'')\right) \longrightarrow \lceil e \rfloor_{s''} \neq q\right)\end{aligned} \tag{4.8}$$

Theorem 2. *Soundness Condition*
The invariants we generate using the technique described in this section are sufficient and inductive if

$$\forall q.\forall s, s'.\alpha_e = \mathbf{none} \vee \left(\overleftarrow{P}_{s'}(s) \wedge \lceil c+1 \rfloor_s = \lceil c \rfloor_{s'} \wedge \lceil e \rfloor_s = q \longrightarrow \neg\overrightarrow{P}_{s'}(s)\right)$$

and

$$\begin{aligned}&\forall s, s'.\alpha_e > \mathbf{none} \wedge \overleftarrow{P}_{s'}(s) \wedge c = 1 \longrightarrow \\ &\forall s''.\left(\overleftarrow{P}_s(s'') \wedge \neg\overrightarrow{P}_s(s'') \wedge \neg\left(\overleftarrow{P}_{s'}(s'') \wedge \neg\overrightarrow{P}_{s'}(s'')\right) \longrightarrow \lceil e \rfloor_{s''} \neq q\right)\end{aligned}$$

hold.

After introducing $\mathtt{c}$ the constraints from (3.3) change as follows:

$$0 \leq \mathtt{i}' < \mathtt{image.length} \wedge \mathtt{i}' + \mathtt{c} \leq \mathtt{i} < \mathtt{image.length} \tag{4.9}$$

consequently we get the following soundness condition from (4.7):

$$\begin{aligned}\forall q.\forall s, s'.\left(0 \leq q < \mathtt{image.length}\ ?\ \frac{1}{2} : \mathbf{none}\right) = \mathbf{none} \vee \\ (\lceil \mathtt{i} \rfloor_s = q \wedge (0 \leq \lceil \mathtt{i} \rfloor_s \wedge \lceil \mathtt{i} \rfloor_s + 1 \leq \lceil \mathtt{i} \rfloor_{s'} < \mathtt{image.length}) \longrightarrow \\ \lceil \mathtt{i} \rfloor_s < \lceil \mathtt{i} \rfloor_{s'} + 1 \vee \mathtt{image.length} \leq \lceil \mathtt{i} \rfloor_s)\end{aligned}$$

which we can see is true.

Note that if the abstract domain we use does not give us constraints that are within Presburger arithmetic we can use Cooper's quantifier elimination algorithm [8] to eliminate the quantifiers in (4.7), i.e. the soundness condition is decidable.

As a final step we use the maximum elimination technique described by Dohrau et al. to obtain a closed-form expression from the maximum term that we can use in our loop invariant.

Remarks

There are a number of different approaches that can be used to generate framing invariants. Firstly we consider simply generating the invariant

$$\max_{s'' | \overrightarrow{P}_s(s'')} \left(\lceil e \rfloor_{s''} = q \ ? \ p_{s''} : \textbf{none} \right), \tag{4.10}$$

where $p_{s''}$ gives the permissions needed to execute the iteration starting from state s''. This approach effectively calculates a sufficient precondition over all potential future loop iterations. It is, therefore, *sufficient*. We, moreover, require similar soundness conditions to (4.5) and (4.7) to prove *inductivity*. What makes this approach undesirable is that it *leaks permissions*: for Example 1 we would, with the future progressive invariant from (3.5), generate the invariant

$$\forall q. \texttt{i} \leq q < \texttt{image.length} \longrightarrow \textbf{acc}(\texttt{a[q]}, \ \textbf{write}),$$

which is not sufficient to evaluate the invariant on line 5 of Example 1 since it does not give us enough permission to read the values we have already incremented.

Another approach one might consider would be to simply give away the permissions `exhaled` in all iterations that satisfy the past progressive invariant, i.e. maximize over $\left\{ s'' \middle| \overleftarrow{P}_s(s'') \right\}$ instead of $\left\{ s'' \middle| \overleftarrow{P}_s(s'') \wedge \neg \overrightarrow{P}_s(s'') \right\}$ in (4.1). This approach is, however, neither sound nor inductive: from Fig. 4.1 we can see that since $\overleftarrow{P}_s(\cdot)$ is an over-approximation it may be true for iterations that have not been executed, yet, and, thus, give away permissions that are still needed.We can also see that the generated invariant is not necessarily inductive: since it is over-approximative we can choose $\overleftarrow{P}_s(\cdot) :=$ true. In this case we give away all permissions before we enter the loop and then, since the invariant does not change, we do not give away any permissions during its execution.

4.1.2 Inhales

Analogously to our approach for `exhales` we can handle iterations that gain more permissions than they lose by first summarizing them into a single statement **inhale acc**`(a[e], `α_e`)`. We then add an additional term $g(s)$ to our invariant for location `a[`q`]` that captures the gain up to the point where we reach s:

$$g(s) := \max_{s'' | \underset{\leftarrow_s}{P}(s'') \wedge \neg \underset{\rightarrow_s}{P}(s'')} \left(\lceil e \rfloor_{s''} = q \ ? \ \alpha_e : \textbf{none} \right), \tag{4.11}$$

where $\underset{\leftarrow_s}{P}$ and $\underset{\rightarrow_s}{P}$ are *under-approximative* progressive loop invariants. As mentioned before these can, in principle, be generated using the same technique described in Sec. 3.1 using an under-approximative abstract domain.

The current implementation uses user-provided under-approximative past progressive invariants and uses false if none are provided.

The resulting invariant is *inductive*, i.e. does not inhale permissions that are not `inhale`d in the concrete execution at exactly the current iteration. This is guaranteed because $\underset{\leftarrow_s}{P}$ and $\underset{\rightarrow_s}{P}$ are under-approximative, i.e. any state that satisfies them is guaranteed to be reached, correspondingly any `inhale` that happens based on these progressive loop invariants is guaranteed to happen in the concrete execution. It is, moreover, *sufficient*, i.e. contains enough permissions to execute the remainder of the loop, if the original precondition of the loop was sufficient w.r.t. the under-approximative abstract state since the progressive loop invariants we use being under-approximative guarantees that we `inhale` all necessary permissions before we need them.

As a final step we, again, use the maximum elimination technique described by Dohrau et al. to obtain a closed-form expression.

4.2 Access Order

The analysis described by Dohrau et al. calculates the loop precondition as the maximum amount of permissions required for an array location across all loop iterations. For this reason it introduces a soundness condition stating that executing any two loop iterations in succession requires no more permissions than the maximum amount of permissions required to execute each iteration individually. This condition is violated if we e.g. `exhale` permissions to a particular location and access that location in a later iteration. In this section we will use progressive loop invariants to extend the analysis described by Dohrau et al. to overcome some of the limitations imposed by the soundness condition and formulate a new more precise soundness condition. We, moreover, show that the same techniques can be used to extend the analysis described in Sec. 4.1 which relies on the same soundness condition as the analysis described by Dohrau et al.

For a program that violates the soundness condition, i.e. where executing two iterations in succession requires more permissions than is required to execute either iteration on its own, we can see that the order in which iterations are executed is important.

Let us illustrate this observation based on an example. Example 2 shows a modified version of Example 1 with an additional `exhale` statement on the highlighted line. Let us look at `image[0]` which is accessed when `i` is 0 and when `i` is 1. In the first iteration (when `i` is 0) we write to `image[0]` and `exhale` half a permission to it. We then, in the second iteration (when `i` is 1), `exhale` another half of a permission. Therefore we have to start out with full **write** permission and we are left with no permissions to `image[0]`. For Example 3 which shows the same program as Example 2 but it accesses the array in the reverse order we get different results. We first `exhale` half

Example 2. Accessing each array location twice in ascending order

```
1  var image := get_image()
2  var i := 0
3  while (i < image.length)
4  {
5    // increase brightness
6    image[i] := image[i] + 10
7
8    // render
9    exhale acc(image[i], 1/2)
10   if (i ≥ 1) exhale acc(image[i - 1], 1/2)
11
12   i += 1
13 }
```

Example 3. Accessing each array location twice in descending order

```
1  var image := get_image()
2  var i := image.length - 1
3  while (i >= 0)
4  {
5    // increase brightness
6    image[i] := image[i] + 10
7
8    // render
9    exhale acc(image[i], 1/2)
10   if (i ≥ 1) exhale acc(image[i - 1], 1/2)
11
12   i -= 1
13 }
```

a permission to `image[0]` in the second to last iteration (when `i` is 1) but we then still need to have **write** permission in order to execute the write at the beginning of the last iteration (when `i` is 0). We, therefore, need to start out with **acc**(`image[0]`, $\frac{3}{2}$) which is equivalent to false. We require different preconditions for these examples. However, for the analysis described by Dohrau et al. which only analyzes which values `i` takes on but not in which order it takes them on both of these program look the same.

In the remainder of this section we will first, in Sec. 4.2.1, introduce a technique that allows us to determine the order in which different statements of a program access the same array location. We will then, in Sec. 4.2.2, see how we can use the order we identify to extend the inference for loop pre- and postconditions described by Dohrau et al. and invariants described in Sec. 4.1 to programs that do not satisfy the original soundness condition.

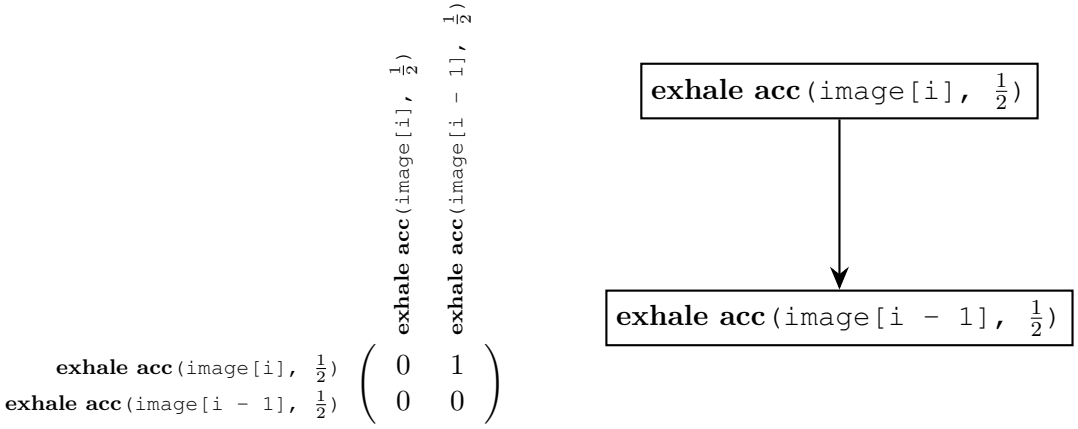

Figure 4.2: Matrix O and order graph for the `exhales` in Example 2.

4.2.1 Generating the Order Graph

We represent the order in which different statement of a program access the *same* array location using the *order graph*. Each statement we analyze is represented by a node of the graph and a (directed) edge of the graph indicates that the statement at beginning the edge potentially access a location before the statement at the end of the edge. The order graph for the exhales in Example 2 is shown in Fig. 4.2. Additionally we define the *order matrix* O as the adjacency matrix of the order graph.

The main idea of the technique we use to calculate the order graph is that for any given states s and s' the progressive loop invariants can tell us whether s' can be reached before s (the past progressive invariantof s is true for s') and/or after s (the future progressive invariantis true). If we can, therefore, find the iterations in which two statements access the same array location we can use progressive loop invariants to decide whether and in what direction there should be an edge between them.

For any pair of statements $stmt_1$ accessing $a[i]$ that is executed if b_1 is true and $stmt_2$ accessing $a'[i']$ that is executed if b_2 is true we can see that two iterations starting from states s and s' respectively satisfy the constraint

$$L :\equiv \lceil b_1 \rfloor_s \wedge \lceil b_2 \rfloor_{s'} \wedge (\lceil a \rfloor_s = \lceil a' \rfloor_{s'}) \wedge (\lceil i \rfloor_s = \lceil i' \rfloor_{s'}) \tag{4.12}$$

iff $stmt_1$ accesses the same location in the iteration starting from s as $stmt_2$ accesses in the iteration starting from s'. We can, therefore, check in what order these statements access the same location by checking whether (4.12) is compatible with $\overset{IC}{\Longrightarrow}_s(s')$ and $\overset{IC}{\Longleftarrow}_{s'}(s)$ as defined in Def. 2. The reason we use strict progressive loop invariants is that we are only interested in accesses to the same location in different iterations. We handle accesses to the same

location in a the same iteration during the analysis of a single iteration that is already part of the analyses described by Dohrau et al. and in Sec. 4.1.

Definition 2. *Compatibility*
A formula L is compatible with $\overset{\Longrightarrow IC}{P}_s(s')$ iff

$$L \wedge \overset{\Longrightarrow IC}{P}_s(s')$$

is satisfiable.
Likewise, a formula L is compatible with $\overset{\Longleftarrow IC}{P}_{s'}(s)$ iff

$$L \wedge \overset{\Longleftarrow IC}{P}_{s'}(s)$$

is satisfiable.

Checking compatibility can be done using an SMT-solver: the satisfiability modulo theories (SMT) problem is the problem of deciding whether a logical formula containing expressions belonging to some set of theories (e.g. integer arithmetic) is satisfiable. Since many theories are undecidable an SMT-solver can typically output one of three results: "satisfiable," "unsatisfiable," and "unknown." An edge in the order graph indicates that a statement *potentially* accesses a location before another. Conversely if there is no edge from $stmt_1$ to $stmt_2$ we can be sure that $stmt_1$ does not access a location before $stmt_2$ accesses it. In cases where we are unsure whether there should be an edge between $stmt_1$ and $stmt_2$ because the SMT-solver returned the result "unknown" we have to conservatively add the edge so we do not violate the definition of the order graph.

If (4.12) is compatible with both $\overset{\Longrightarrow IC}{P}_s(s')$ and $\overset{\Longleftarrow IC}{P}_{s'}(s)$ the statements potentially access the same location in any order. If (4.12) is only compatible with $\overset{\Longrightarrow IC}{P}_s(s')$ then $stmt_1$ potentially accesses each location before $stmt_2$ accesses it but never after $stmt_2$ accesses it. Analogously if (4.12) is only compatible with $\overset{\Longleftarrow IC}{P}_{s'}(s)$ $stmt_2$ potentially accesses each location before $stmt_1$ does. If neither check succeeds both statements are guaranteed to never access the same location.

We can, analogously, use the under-approximative strict progressive loop invariants $\underset{\Longrightarrow}{P}_s(s')$ and $\underset{\Longleftarrow}{P}_{s'}(s)$ to check whether $stmt_1$ is *guaranteed* to access a location before $stmt_2$ does and vice versa. An edge in the resulting order graph is interpreted as a "must access before" relation. We can, therefore, only add an edge if we are sure that it should be part of the order graph, i.e. we do not add an edge if the SMT-solver returns "unknown."

Once we have constructed the order graph we can use the information it stores to extend the analyses described by Dohrau et al. and in Sec. 4.1 as described in the next section.

4.2.2 Extending Pre-/Postcondition and Invariant Inference

The technique for inferring loop postconditions described by Dohrau et al. relies on us being able to calculate a sound loop precondition. It then calculates the total change in permissions over the entire loop and adds or subtracts those permissions to the precondition. Analogously the invariant inference described in Sec. 4.1 calculates the change in permissions up to the current iteration and adds or subtracts those permissions from the loop precondition. We will, therefore, first look at how we can generate sound loop preconditions even when the soundness condition described by Dohrau et al. does not hold.

The precondition inference described by Dohrau et al. calculates the amount of permissions we need to execute the single iteration starting from state s. We will denote this terms with $r(s)$. It then calculates the loop precondition as

$$\max_{s|I(s)} r(s),$$

where $I(s)$ over-approximates the loop iterations we execute. The idea of how we extend this approach is that if by the time we reach s we have lost some permissions $l(s)$ and gained some permissions $g(s)$ we still need to have the permissions $r(s)$ to execute the iteration starting from s, i.e. we are looking for some amount of permissions $r'(s)$ we need to start the loop with such that

$$r'(s) - l(s) + g(s) \geq r(s)$$

After isolating $r'(s)$ we obtain

$$r'(s) \geq r(s) + l(s) - g(s).$$

Since we cannot require negative amounts of permissions we lower-bound $r'(s)$ by **none**. We get

$$r'(s) := \max(r(s) + l(s) - g(s), \mathbf{none}). \tag{4.13}$$

If we are able to calculate this $r'(s)$ for every iteration we can then calculate a sound precondition

$$\max_{s|I(s)} r'(s).$$

For Example 2 we will see that we can calculate:

$$
\begin{aligned}
r(s) &\equiv (q = \texttt{i}\ ?\ \textbf{write} : \textbf{none}) + (0 \leq q = \texttt{i} - 1\ ?\ \frac{1}{2} : \textbf{none}) \\
l(s) &\equiv (0 \leq q < \texttt{i}\ ?\ \frac{1}{2} : \textbf{none}) + (0 \leq q < \texttt{i} - 1\ ?\ \frac{1}{2} : \textbf{none}) \\
g(s) &\equiv \textbf{none} \\
r'(s) &\equiv \max(r(s) + l(s) - g(s), \textbf{none}) \\
&\overset{*}{\equiv} (0 \leq q \leq \texttt{i})\ ?\ \textbf{write} : \textbf{none}
\end{aligned}
$$

* for simplicity we assume $0 \leq \texttt{i}$ here which we would usually only get from $I(s)$ during the generalization step

from which we can calculate the precondition

$$\forall q.\textbf{acc}(\texttt{image[}q\texttt{]}, 0 \leq q < \texttt{image.length}\ ?\ \textbf{write} : \textbf{none}).$$

We will likewise see how we can calculate these terms for Example 3:

$$
\begin{aligned}
r(s) &\equiv (q = \texttt{i}\ ?\ \textbf{write} : \textbf{none}) + (0 \leq q = \texttt{i} - 1\ ?\ \frac{1}{2} : \textbf{none}) \\
l(s) &\equiv (\texttt{i} < q < len\ ?\ \frac{1}{2} : \textbf{none}) + (0 \leq q \wedge \texttt{i} - 1 < q < len - 1\ ?\ \frac{1}{2} : \textbf{none}) \\
g(s) &\equiv \textbf{none} \\
r'(s) &\equiv \max(r(s) + l(s) - g(s), \textbf{none}) \\
&\geq (0 \leq q = \texttt{i} < len - 1)\ ?\ \frac{3}{2} : \textbf{none},
\end{aligned}
$$

where $len = \texttt{image.length}$. Since permissions may not be larger than 1 this will, as we expected based on the discussion at the beginning of the section, result in the precondition false (unless `image` contains at most one element).

We will discuss how we can automatically calculate the permission loss $l(s)$ up to s and the permission gain $g(s)$ up to s next.

Losing Permissions

In this section we will see how we can calculate the permission loss $l(s)$ up to the point where we reach s. We lose permissions when an `exhale` statement is executed. We will, therefore, only look at `exhale` statements in this section and only consider the oder graph for the `exhale` statements of a program. If some statement `exhales` p permissions to location l and then at a later point in time another statement `exhales` p' permissions to

Example 4. No location is accessed multiple times

```
1  var image := get_image()
2  var i := ramdom_number(from: 0, to: a.length - 1)
3  while (i < image.length)
4  {
5    if (i % 2 == 0) {
6      exhale acc(image[i], 1/2)
7    } else {
8      exhale acc(image[i - 1], 1/2)
9    }
10   i += 2
11 }
```

location l the total loss of permissions is $p + p'$. The idea of the approach we present is that we can identify this kind of scenario using the order graph.

For Example 2 we have already seen the order graph for its `exhale` statements in Fig. 4.2. For now, we will assume that the order graph is acyclic. We will later discuss how we can detect and, in some cases, eliminate cycles. In an acyclic graph we can then observe that every `exhale` only accesses each location once (otherwise there would have to be an edge from some statement to itself which would be a cycle). If we look at each `exhale` statement in isolation the soundness condition of the analysis described in Sec. 4.1.1 stating that we `exhale` permissions to each array location in at most one iteration, therefore, holds. Consequently we can use the analysis described in Sec. 4.1.1 to calculate the loss $l_i(s)$ for each `exhale` statement i separately and obtain the overall permission loss for n `exhale` statements

$$l(s) = \sum_{i=1}^{n} l_i(s).$$

Moreover, we noted in Sec. 4.2.1 that any two statements that are not connected by an edge in the order graph are guaranteed to never access the same array location. We can, therefore, also analyze any set of `exhale` statements where there is no edge between any two statements in that set together using the analysis described in Sec. 4.1.1. For c such sets we can calculate the total loss by summing up the loss $l_i(s)$ we calculate for each set:

$$l(s) = \sum_{i=1}^{c} l_i(s). \tag{4.14}$$

Forming such sets instead of calculating the loss for each `exhale` statement separately can sometimes improve the precision of the loss we calculate. This occurs when the over-approximation of the states we encounter during the loop execution leads us to believe that two `exhale` statements can

# exhales	# checks	inference time [s]	check time [s]	total time [s]	overhead
1	0	7.1	0.0	7.1	0.2%
2	1	33.2	0.9	34.1	2.6%
3	3	71.9	2.3	74.2	3.2%
4	6	230.3	4.8	235.1	2.1%
5	10	298.4	10.2	308.6	3.4%

Figure 4.3: Results for simulating checks required to calculate optimal graph coloring based on a program similar to the one shown in Example 4 that is extended to different numbers of `exhale` statements.

access the same location but we can show that that is not the case using progressive loop invariants. We illustrate this problem based on the program shown in Example 4. Using polyhedra we would get the constraints $0 \leq \texttt{i} < \texttt{image.length}$. Analyzing both statements separately and adding the results together tells us that we lose **write** permission to almost every element with an even index. This is sound since we always have more permissions in the concrete execution than are required by the specifications. However, we can see that the parity of the initial value of `i` is preserved. We, therefore, know that in any concrete execution either only the `exhale` on l. 6 or only the `exhale` on l. 8 is executed. We can also deduce this using progressive loop invariants: the future progressive invariant of a state with an even `i` is only true for states where `i` is even and the future progressive invariant for a state with an odd `i` is only true for states where `i` is odd. The order graph for Example 4, therefore, does not contain an edge between the two `exhale` statements and, hence, allows us to assume that the two statements never access the same array location. We can then use the analysis described in Sec. 4.1.1 to calculate the more precise loss of half a permission to every array element with an even index.

Finding sets of nodes in the order graph such that there is no edge between any nodes within that set is equivalent to the vertex coloring problem of the order graph. A vertex coloring needs to ensure that there is no edge between any two vertices that are assigned the same color. However, two nodes that are assigned different colors need not necessarily be connected by an edge. It should be noted that an optimal vertex coloring in terms of the loss we calculate, i.e. a coloring that minimizes the loss we calculate, is one that maximizes the kind of "overlap" we just discussed between vertices of the same color and is not necessarily a minimal vertex coloring, i.e. a vertex coloring that uses the fewest colors possible. A technique for calculating a suitable vertex coloring is presented next.

Calculation of Vertex Coloring

As we mentioned a vertex coloring resulting in the smallest possible loss $l(s)$ should maximize the amount of "overlap" between vertices of the same color. We can calculate this "overlap" for any pair of statements $stmt_1$ exhaling p permissions from $a[i]$ that is executed if b_1 is true and $stmt_2$ exhaling p' permissions from $a'[i']$ that is executed if b_2 is true by looking only at the locations that both statements access and taking the minimum amount of permissions that is required is exhaled by both statements. We can express this as

$$\max_{s,s' | I(s) \wedge I(s')} L(s, s') \ ? \ \min(p, p') : \textbf{none},$$

where

$$L(s, s') :\equiv \lceil b_1 \rfloor_s \wedge \lceil b_2 \rfloor_{s'} \wedge (\lceil a \rfloor_s = \lceil a' \rfloor_{s'}) \wedge (\lceil i \rfloor_s = \lceil i' \rfloor_{s'}) \, .$$

We can, moreover, eliminate this unbounded maximum using the maximum elimination mechanism described by Dohrau et al. Calculating the overlap for any combination of two exhale statements (not including the overlap with itself which obviously always exists) in a loop containing n `exhale` statements requires us to calculate

$$\frac{n(n-1)}{2}$$

of these maximum terms, i.e. the number of checks we have to perform grows quadratically in n. Furthermore, even a single such check can be quite expensive: Fig. 4.3 shows the runtimes for simulating the calculation of the necessary checks for a series of programs with different numbers of `exhale` statements. This simulation performs the checks described above but it does not analyze the results as we would have to if we wanted to use them to find an optimal vertex coloring. We can see that the time required for these checks increases rapidly. At the same time the runtime of the inference itself increases because the size of the terms we give to the maximum elimination to calculate $l_i(c)$ increases. Overall the relative overhead introduced by the overlap checks stays about the same. It should be noted that this is a relatively simple example and accordingly the overhead is relatively small for some examples the overhead is about 30%.

Moreover, one can imagine scenarios where we have a choice between reducing the amount of permissions lost for a few locations by a large amount or for many locations by a small amount. It is unclear what the better choice in such a scenario would be. It is, however, clear that unifying two colors if possible can only improve our solution. Instead of trying to find a solution that globally minimizes $l(s)$ we, therefore, use an algorithm that efficiently calculates such a local optimum which works well in practice.

```
1  var roots = nodes.filter(no incoming edges)
2  var generations
3  generations.push(roots)
4  while (generations.top is non-empty) {
5    currentGeneration = generations.top
6    nextGeneration = currentGeneration.flatMap(successors)
7    generations.push(nextGeneration)
8  }
9  var currentColor = 0
10 var alreadyColored
11 while (generations is non-empty) {
12   generation = generation.pop()
13   toColor = generation without alreadyColored
14   toColor.assignColor(currentColor)
15   currentColor += 1
16   alreadyColored.add(toColor)
17 }
```

Figure 4.4: Pseudo code for calculating a graph coloring of the order graph.

This algorithm finds some roots (vertices with no incoming edges) and calculates the length of the longest path from these roots to each vertex. Vertices with the same longest path length are then assigned the same color. Pseudo code for the algorithm is shown in Fig. 4.4. It starts out by finding the roots from which we explore the graph in generations by following the the outgoing edges of every node in the previous iteration. Since the graph is acyclic this search terminates. Nodes that are part of the n^{th} generation we discover can be reached from a root via a path of length n. We, then, remove each node from every generation except the last one of which it is a member. Afterwards, the n^{th} generation contains only nodes for which the longest path that reaches it has length n. Finally, we assign each generation a different color.

Lemma 2. *Soundness*
The coloring calculated by the algorithm shown in Fig. 4.4 assigns every pair of vertices that are connected by an edge a different color.

Proof. We show that assuming that there is an edge connecting two vertices that are assigned the same color leads to contradiction. If there were an edge from a vertex v_1 to a vertex v_2 and they are both assigned the same color the longest path that reaches them has to have the same length of n for each of them. However we can construct a path to v_2 by using the longest path to v_1 and appending the edge to v_2 to it. This path has a length of $n + 1$ which contradicts the assumption that the longest path reaching v_2 has length n. □

Lemma 3. *For every vertex v_1 that is assigned color $c > 0$ there exists a v_2 that is assigned color $c - 1$ such that there is an edge $v_2 \to v_1$ in the order graph.*

Proof. Recall that a vertex v is assigned to color n iff the longest path from a root that reaches v has length n. Let us consider a longest path that reaches v_1: $v_1' \to v_2' \to \cdots \to v_2 \to v_1$. This v_2 exists because the lemma states that v_1 is assigned some color $c > 0$, i.e. the longest path to v_1 has at least length 1. We will, moreover, show that this v_2 satisfies the lemma, i.e. that the color n it is assigned is $n = c - 1$. We observe that v_2 cannot be assigned to a color $n < c - 1$ since the path $v_1' \to v_2' \to \cdots \to v_2$ has length $c - 1$. Furthermore assuming that v_2 is assigned a color $n > c - 1$ leads to a contradiction: we could extend the longest path to v_2 of length n by adding the edge $v_2 \to v_1$ to it and construct a path to v_1 of length $n + 1 > c$. This contradicts the assignment of v_1 to color c. We have shown that $n \not< c - 1$ and $n \not> c - 1$. Consequently $n = c - 1$. □

Lemma 4. *Local Optimality*
No two colors calculated by the algorithm shown in Fig. 4.4 can be merged without violating the soundness property from Lemma 2.

Proof. We show that for any pair of distinct colors c_1 and c_2 there exists a vertex v_1 that is assigned color c_1 and a vertex v_2 that is assigned color c_2 such that there is an edge between v_1 and v_2. Recall that we assign colors based on the length of the longest path from a root that reaches a vertex, i.e. the longest path that reaches each vertex in c_1 has length c_1 and the longest path that reaches each vertex in c_2 has length c_2. We will w.l.o.g. assume that $c_1 < c_2$ (this implies that $0 < c_2$). Let v_2 be an arbitrary vertex that is assigned color c_2. We can apply Lemma 3 to get a v' that is assigned color $c_2 - 1$ such that an edge $v' \to v_2$ exists. Analogously we can find a v'' in color $c_2 - 2$ such that the path $v'' \to v' \to v_2$ exists and so on, i.e. by induction we can show that for some v_1 that is assigned color c_1 there exists a path from v_1 to v_2. The intuitive interpretation of an edge in the order graph is a "potentially accesses before" relation. It is clear that if $stmt_1$ potentially accesses a location before $stmt_2$ and $stmt_2$ potentially accesses that location before $stmt_3$ then $stmt_1$ also potentially accesses the location before $stmt_3$, i.e. the "potentially accesses before" relation is transitive. In terms of the order graph this means that whenever there is a path from v to v' there also is an edge $v \to v'$ that connects v and v' directly. We have shown that there is a path from v_1 to v_2. Consequently there also is an edge $v_1 \to v_2$, i.e. we have found a v_1 in color c_1 and a v_2 in color c_2 that are connected by an edge in the order graph. We can, therefore, not merge c_1 and c_2 without violating Lemma 2. □

For Example 2 and Example 3 only the vertex coloring that assigns a different color to each of the two `exhale` statements is possible. Using the technique described in Sec. 4.1.1 we can separately derive

$$l_1(s) = (0 \leq q < \texttt{i} \ ? \ \frac{1}{2} : \textbf{none})$$

as the loss for the `exhale` on l. 9 of Example 2,

$$l_2(s) = (0 \leq q < \texttt{i} - 1 \ ? \ \frac{1}{2} : \textbf{none})$$

for the `exhale` on l. 10 of Example 2,

$$l_3(s) = (\texttt{i} < q < \texttt{image.length} \ ? \ \frac{1}{2} : \textbf{none})$$

for the `exhale` on l. 9 of Example 3, and

$$(0 \leq q \wedge \texttt{i} - 1 < q < \texttt{image.length} - 1 \ ? \ \frac{1}{2} : \textbf{none})$$

for the `exhale` on l. 10 of Example 3. For each example we then calculate the total loss up to the iteration starting from state s by summing the results for the two `exhales` together. For Example 2 we get

$$l(s) \equiv l_1(s) + l_2(s) \equiv (0 \leq q < \texttt{i} \ ? \ \frac{1}{2} : \textbf{none}) + (0 \leq q < \texttt{i} - 1 \ ? \ \frac{1}{2} : \textbf{none})$$

and for Example 2 we get

$$l(s) \equiv l_3(s) + l_4(s) \equiv (\texttt{i} < q < \mathit{len} \ ? \ \frac{1}{2} : \textbf{none}) + (0 \leq q \wedge \texttt{i} - 1 < q < \mathit{len} - 1 \ ? \ \frac{1}{2} : \textbf{none}).$$

Handling Cycles. During the previous discussion we assumed that the order graph does not contain any cycles. We will now look at how we handle programs for which the order graph contains cycles. We distinguish two kinds of cycles the order graph may contain: *true cycles* indicate that a statement potentially accesses the same location again later on, i.e. that it potentially accesses the same location multiple times. In the context of `exhale` statements this means that we keep losing permissions. While it might in principle be possible to upper-bound the amount of permissions that is lost in some such cases our analysis simply requires that the order graph does not contain any true cycles as a soundness condition. Since the order graph is constructed based on the progressive loop invariants we calculate we will look at how true cycles can be detected next so we can automatically check that the soundness condition holds during the inference.

Theorem 3. *Soundness Condition*
If the order graph on the `exhale` statements of a program is free from true cycles the loss $l(s)$ we calculate upper-bounds the loss we observe in concrete executions.

Example 5. Program that generates a pseudo cycle

```
1 var image := get_image()
2 var i := 0
3 while (i < image.length)
4 {
5   exhale acc(image[i], 1/2)
6   if (0 ≤ i + k < image.length) exhale acc(image[i + k], 1/2)
7   i += 1
8 }
```

As we discussed during the proof of Lemma 4 the "potentially accesses before" relation that is represented by edges in the order graph is transitive. In terms of the graph this means that if there is a path from a vertex v_1 to a vertex v_2 then there is also an edge that connects v_1 and v_2 directly. For a node v that is part of a true cycle this means that since there is a path from v to v there also has to be an edge from v to itself. We can, therefore, check whether an order graph contains any true cycles by checking whether any entries on the main diagonal of its adjacency matrix, i.e. the order matrix O are 1. If we find such entries the soundness condition is violated and we simply generate false as the invariant and pre- and postconditions as we already do when the soundness condition in Theorem 2 is violated.

The other kind of cycle the order graph can contain are *pseudo cycles*. A pseudo cycle exists when two statements may access the same array location in any order in different concrete executions, i.e. for different parameter values, user-input, randomness, etc. but in a single concrete execution they access each array location in a particular order and they each access each array location at most once.

To illustrate what pseudo cycles are and when they occur we consider Example 5 which contains two `exhale` statements that exhale some amount of permissions from `image[i]` and `image[i + k]` respectively, where k is some *unknown* constant (e.g. a method parameter). We can see that for negative k we first `exhale` from a particular location using the `exhale` on l. 5 and, then, from the same location again using the `exhale` on l. 6. Analogously if k is positive we first use the `exhale` on l. 6 and then the one on l. 5 for each location. Both of these scenarios are potential program behaviours. Consequently the order graph contains edges in both directions between the corresponding two nodes. In any single concrete execution (i.e. for a fixed k) we can, however, only observe one of the two edges, i.e. we can never execute the same `exhale` statement on the same location twice. We can observe this based on the order graph as well: there is no edge from either of the statements to itself (`i + k` cannot take on the same value for different values of `i` while k remains constant). We will use this observation to automatically detect pseudo cycles and eliminate them.

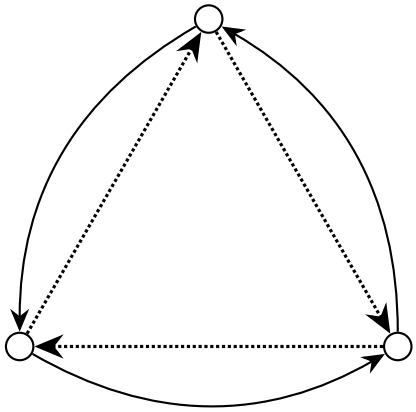

Figure 4.5: A order graph containing three pseudo cycles.

Pseudo cycles only exist between pairs of nodes so, after checking that the graph does not contain true cycles, they can be found by checking the order graph for nodes that are connected by edges in each direction. This can be done by checking the order matrix O for entries where $O_{ij} = O_{ji} = 1$. Furthermore, we can observe that we can remove either edge without affecting the possible vertex colorings of the graph since the other edge remains and prevents us from assigning the same color to each of its endpoints. However, since nodes can be part of multiple pseudo cycles we have to be careful to make sure the resulting graph does not contain any additional cycles that are a result of the original pseudo cycles. Fig. 4.5 illustrates this issue: if we remove the dotted edges we have resolved each of the three pseudo cycles but the remaining graph still contains a cycle. We can avoid this issue by always keeping the edge that goes from a node with a lower index in the order matrix O to a node with a higher index in O. Note that this also preserves the transitivity property required to prove Lemma 4: for any two edges from v_1 to v_2 and from v_2 to v_3 where the index of v_2 is larger than that of v_1 and the index of v_3 is larger than that of v_2 the transitive edge from v_1 to v_3 is also kept.

Remarks. In practice we summarize the permissions we lose in a single iteration into a single expression. For Example 2 this expression would look similar to

$$\left(q_1 = \texttt{image} \land q_2 = \texttt{i} \;?\; \frac{1}{2} : \textbf{none}\right) + \left(q_1 = \texttt{image} \land q_2 = \texttt{i} - 1 \land \texttt{i} \geq 1 \;?\; \frac{1}{2} : \textbf{none}\right).$$

This expression gives us the amount of permissions we lose for any location $q_1[q_2]$. We can obtain a condition under which this expression is (potentially)

greater than 0 by analyzing it syntactically. To separate out different locations that are accessed during a single iteration we convert this condition into disjunctive normal form (DNF) and assume the disjuncts each refer to a different statement. We can see that if this separation does not work, i.e. a single disjunct refers to multiple statements, we may observe additional cycles in our graph. While this reduces the precision of our approach (we generate false) it should be noted that this can never cause unsoundnesses.

Gaining Permissions

In this section we will see how we can calculate the permission gain $g(s)$ up to the point where we reach s. The soundness condition described by Dohrau et al. states that executing any two iterations in succession must not require more permissions than the maximum that is needed to execute each of them individually. If we gain permissions during one of the iterations we provide additional permissions that then no longer have to be given before the iterations are executed, i.e. we reduce the amount of permissions that is needed and, therefore, do not violate the soundness condition. We can, however, still profit from the order information we calculate to calculate a more precise gain.

As we mentioned in Sec. 4.2.1 we can use order checks based on the under-approximative strict future progressive invariant $\underset{s}{\overset{P}{\Longrightarrow}}(s')$ to construct an order graph where each edge is interpreted as a "must access before" relationship. The calculation of $g(s)$ works analogously to the calculation of $l(s)$: We find a vertex coloring and use the technique described in Sec. 4.1.2 to calculate the combined gain $g_i(s)$ for each set of statements assigned the same color i. For c colors we then calculate the overall gain as the sum

$$g(s) = \sum_{i=1}^{c} g_i(s). \tag{4.15}$$

We handle `inhales` under-approximatively, i.e. we only consider program behaviour that is guaranteed to happen in concrete executions, in the analysis described in Sec. 4.1.2. Therefore, "overlap"-scenarios where the analysis described in Sec. 4.1.2 thinks two `inhales` can access the same location but this is never the case in concrete executions cannot occur. Because there is no overlap between the locations accessed by any two statements that are not connected by an edge calculating the combined $g_i(s)$ using the technique described in Sec. 4.1.2 for sets of statements with no edges between them gives the same result as calculating the gain for each statement separately and adding both results together. Any vertex coloring of the order graph, hence, results in the same permission gain being calculated.

Furthermore, we observe that the order graph calculated from under-approximative progressive loop invariants cannot contain true cycles: if a

statement is part of a cycle it can only be executed after it has already been executed before. Consequently the statement can never be executed which is a contradiction since the under-approximative progressive loop invariants *guarantee* that every state that satisfy them is reached, i.e. that the statement is executed. Pseudo cycles can occur since they are resolved for any concrete execution and can be eliminated using the same technique we described for `exhales`.

Remarks. Similarly to what we described for `exhales` we separate different locations that are accessed in an iteration by deriving some condition under which we gain permissions and converting it to DNF. If this separation is not exact we may end up with true cycles in the order graph. In such cases we can simply ignore these cycles since, as we mentioned at the beginning of this section, the soundness condition described by Dohrau et al. holds.

Invariants and Postconditions

Once we have calculated a sound precondition using $l(s)$ and $g(s)$ we can calculate invariants and loop postconditions. For invariants this is done in a similar fashion to what we described in Sec. 4.1: the invariant for state s is given by

$$\text{precondition} - l(s) + g(s).$$

Analogously to what we described above we can extend the analysis for calculating the overall change in permissions over the execution of the entire loop described by Dohrau et al. by constructing sets of `exhales` and sets of `inhales` that can be given to their analysis without violating their soundness condition. These sets are constructed in the same manner we just described. Consequently we calculate the loss l_i over the execution of the entire loop and the gain g_i over the execution of the entire loop. We then form the sum for c colors analogously to what we described for $l(s)$ and $g(s)$:

$$l = \sum_{i=1}^{c} l_i \tag{4.16}$$

$$g = \sum_{i=1}^{c} g_i \tag{4.17}$$

We then calculate the postcondition in a similar fashion to what Dohrau et al. described: the loop postcondition is given by

$$\text{precondtion} - l + g.$$

We saw during the construction of $l(s)$ that we require the order graph to not contain any true cycles. The extended analysis can, therefore, handle

Example 6. Partial sums

```
1 var a = get_array()
2 var a_orig = a.copy()
3 var i = 1
4 while (i < a.length) {
5   a[i] = a[i - 1] + a[i]
6   i += 1
7 }
8 ensures ∀q. 0 ≤ q < a.length ⟶ a[q] = sum(of: a_orig, from: 0, to: q)
```

programs where each statement accesses each array location at most once provided the generated progressive loop invariants are strong enough to show that that is the case. This soundness condition is, moreover, automatically checked by the implementation of this analysis.

4.3 Functional Specifications

In this section we will look at a technique that utilizes progressive loop invariants to infer *functional specifications*, i.e. specifications containing information about what a method calculates (e.g. that the final array is sorted), for array programs. This approach requires the user to provide a method postcondition and infers inductive loop invariants and method preconditions that allow us to verify the program.

The approach is based on the technique for calculating weakest liberal preconditions described in Sec. 2.6. The main idea is to use wlp generation for loop-free code and handle loops by automatically inferring invariants instead of requiring the user to provide them. In order generate these invariants the inference looks at the read/write-dependencies between elements of an array and across iterations of a loop. We distinguish two kinds of dependencies: those where we read a value that was previously written during the execution of the loop (we call these *recursive dependencies*) and those where we read the original value that was present in an array before we entered the loop (we say that these dependencies *reach outside the loop*).

For Example 6 which stores the partial sums of an array up to and including element `i` at location `a[i]` part of the dependency graph is shown in Fig. 4.6. While this graph can be unboundedly large the analysis exploits that loops typically access an array in a regular fashion resulting in a dependency graph that repeats some pattern. For Example 6 we can clearly see such a pattern. The analysis then constructs an inductive proof over this repeating pattern. The main idea behind generating wlps is that requirements a postcondition imposes on the value of some variable or heap location l can be translated to a requirement on the values of variables and heap locations l depends on. Similarly we try to construct an inductive proof over the

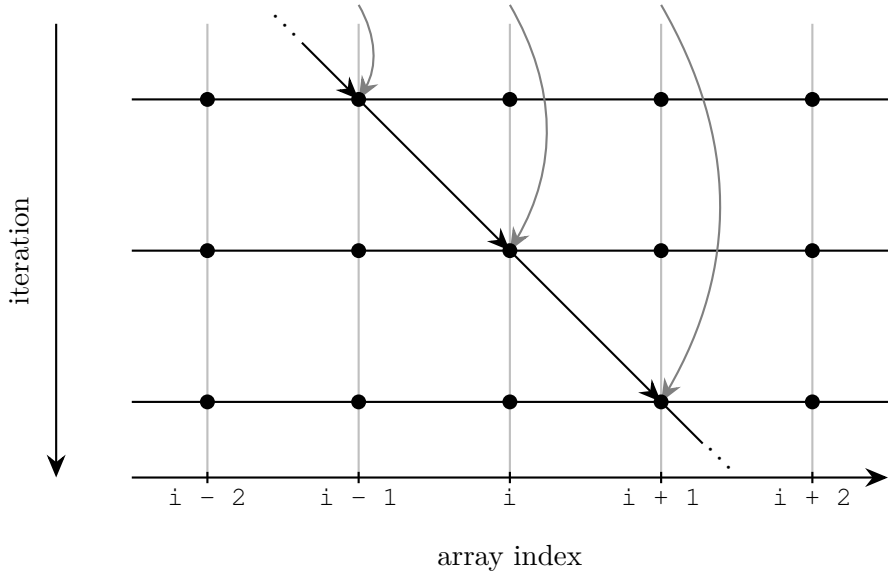

Figure 4.6: Read/write-dependencies between array elements across multiple iterations for Example 6. Arrows indicate dependencies from the point where a value is written to the point where it is read. Black arrows indicate dependencies on values generated in earlier loop iterations (recursive dependencies) while curved, gray arrows indicate dependencies on values generated outside the loop (reaching outside the loop).

repeating structure we identify to show that the requirements imposed by the user-provided postcondition are always met during the execution of the loop. For requirements imposed on recursive dependencies we show this via induction whereas any requirements on dependencies reaching outside the loop are passed on to the loop's precondition. We will first take a look at how we can handle single loops in Sec. 4.3.1 and then extend the approach to successive and nested loops in Sec. 4.3.2.

4.3.1 Single Loops

We will first look at a technique that allows us to identify a repeating pattern in the read/write-dependency graph and how we can distinguish recursive dependencies from dependencies that reach outside the loop. Based on this pattern we then define the structure of the inductive proof, i.e. what we want to show in the base and step cases of the induction and which loop iteration corresponds to which induction step. We define this structure for a *template invariant* which we construct heuristically based on the user-provided postcondition.

In order to discover repeating patterns we first need to introduce some notation. We define the universe of array locations as the cartesian product of arrays with their indices:

$$\mathcal{L} := \mathcal{A} \times \mathbb{Z}$$

We then define the set

$$W \subseteq \mathcal{L}$$

of array locations we write to during a single loop iteration and for each $w \in W$ we define the set

$$R_w \subseteq \mathcal{L}$$

of locations the value we write to w depends on. We can calculate these sets using a data-flow analysis (e.g. [3]).

For Example 6 a data-flow analysis will indicate that we write to $W = \{\texttt{a[i]}\}$ and the new value of `a[i]` depends on the previous values of $R_{\texttt{a[i]}} = \{\texttt{a[i - 1]}, \texttt{a[i]}\}$.

As described above we want to separate recursive dependencies from dependencies that reach outside the loop next. We will then try to prove the postcondition inductively over the recursive dependencies (e.g. for Example 6 we have to inductively show that we have calculated the partial sums up to the current `i`) and we pass on the requirements for the remaining dependencies to the precondition (for Example 6 this will be $\forall \texttt{q}.\texttt{a[q]} = \texttt{a\_orig[q]}$). We can identify recursive dependencies using an analogous approach to the one described in Sec. 4.2.1: we use progressive loop invariants to check which dependencies (potentially) read values that were previously written to by the loop. We define the set of *recursive dependencies*

$$\hat{R}_w := \{r \mid r \in R_w \wedge \text{written}(r)\}\,. \tag{4.18}$$

Analogously to the checks described in Sec. 4.2.1 we calculate written(r) using the check:

$$\text{written}(r) :\equiv \exists w \in W, s, s'.\, (\lceil r.\text{array} \rfloor_s = \lceil w.\text{array} \rfloor_{s'}) \wedge (\lceil r.\text{index} \rfloor_s = \lceil w.\text{index} \rfloor_{s'}) \wedge \overset{\Longleftarrow IC}{P}_s(s').$$

For Example 6 we generate the checks

$$(\lceil \texttt{a} \rfloor_s = \lceil \texttt{a} \rfloor_{s'}) \wedge (\lceil \texttt{i - 1} \rfloor_s = \lceil \texttt{i} \rfloor_{s'}) \wedge \overset{\Longleftarrow IC}{P}_s(s') \tag{4.19}$$

and

$$(\lceil \texttt{a} \rfloor_s = \lceil \texttt{a} \rfloor_{s'}) \wedge (\lceil \texttt{i} \rfloor_s = \lceil \texttt{i} \rfloor_{s'}) \wedge \overset{\Longleftarrow IC}{P}_s(s') \tag{4.20}$$

of which only the check (4.19) succeeds. Accordingly we get the set of recursive dependencies for Example (6):

$$\hat{R}_{\texttt{a[i]}} = \{\texttt{a[i - 1]}\}$$

We have, thus, identified the recursive structure indicated by black arrows in Fig. 4.6.

We will construct an inductive proof over this recursive structure. We do this based on a *template invariant* $T(x; s)$ which for a given array location x checks whether s satisfies some version of the postcondition that essentially captures what work we have already done. We will introduce a heuristic for generating these templates later on. The idea is to generate an inductive proof over these templates. In particular in the step case of the induction we will assume that we have T(x; s) for all recursive dependencies of every $w \in W$ and we show that after executing a single loop iteration we obtain $T(w; s)$. In addition to the templates for recursive dependencies we also use the *precondition requirements* $C_w(s)$ which for every $w \in W$ give the requirements on the dependencies that reach outside the loop and that we pass on to the loop precondition. We will also see how these precondition requirements are constructed later on. These requirements are an additional assumption during the step case. In the full step case we consider an arbitrary iteration starting from state s and every $w \in W$ that we write to in that iteration. We assume that s satisfies

$$C_w(s) \wedge \bigwedge_{r \in \hat{R}_w} T(r; s)$$

and we show that after executing the iteration we end up in a state s' that satisfies

$$T(w; s').$$

Note that if $\hat{R}_w$ is empty we simply start from $C_w(s)$. Note also that we omit showing that $C_w(s')$ is established for the next induction step. Since C_w, by definition, only refers to locations that have not, yet, been written to we know that $C_w(s')$ does not refer to any locations we write to during the iteration starting from s. The iteration starting from s, therefore, trivially preserves C_w.

This construction works well for postconditions that only refer to a single array location and, therefore, result in a proof over a single induction variable. In cases where the postcondition refers to multiple array locations and we perform a proof over multiple induction variables the template will also be parameterized in multiple locations and we have to account for all possible combinations of reads and writes. E.g. for two locations we would have to consider every pair of writes $w_1, w_2 \in W$ and show that if we start out in a

state s satisfying

$$C_{w_1}(s) \wedge C_{w_2}(s) \wedge \bigwedge_{r_1 \in \hat{R}_{w_1}, r_2 \in \hat{R}_{w_2}} T(r_1, r_2; s)$$

we reach a state s' satisfying

$$T(w_1, w_2; s').$$

If the postcondition refers to n array locations the number of tuples in W^n for which we have to perform the step case of the induction grows exponentially in n. We will, therefore, generate a template $T(s)$ and precondition requirements $C(s)$ that consider all reads/writes that occur during a single iteration at once. While the size of such templates can, in the worst case, still grow exponentially it is oftentimes possible to simplify the resulting formula to avoid this fate. In the step case for such templates we start out from a state s that satisfies

$$C(s) \wedge T(s)$$

and we show that we reach a state s' satisfying

$$T(s')$$

We will look at how these templates are generated next.

Template Generation

To generate the template we first obtain a loop postcondition by executing the wlp calculation until we reach the end of the loop. We will convert the resulting loop postcondition into the form

$$\forall \vec{v}. F(\vec{v}), \tag{4.21}$$

where $F(\cdot)$ gives us the loop postcondition at a single point and does not contain quantifiers. This is not a restriction since we can convert any postcondition into this form by eliminating existential quantifiers via Skolemization [13] and pulling universal quantifiers out of inner expressions. For Example 6 the postcondition is already in this form so we obtain.

$F(\texttt{q}) \equiv 0 \leq \texttt{q} < \texttt{a.length} \longrightarrow \texttt{a}[q] = \texttt{sum(of: a\_orig, from: 0, to: q)}.$

We then syntactically replace all array locations referred to in $F(\cdot)$ with n new quantified variables to obtain a new, equivalent, formula

$$\forall (\vec{v}, \vec{l}) \in B.\ F'(\vec{v}, \vec{l}), \tag{4.22}$$

where B contains the original locations referred to in $F(\vec{v})$. E.g. for Example 6 we obtain

$$B = \{(q, l) | l = \texttt{a[q]}\} \tag{4.23}$$

and

$$F'(\texttt{q}, l) \equiv 0 \leq \texttt{q} < \texttt{a.length} \longrightarrow \lceil l \rfloor_s = \texttt{sum(of: a\_orig, from: 0, to: q)}.$$

We saw earlier that we construct an inductive proof over the locations $\vec{l}$. The main idea behind the templates we generate is that we define a set $B'(s)$ which conceptually gradually collects locations for which we have already established the point-wise postcondition $F'(\vec{v}, \vec{l})$ as we execute additional loop iterations. We will later see that we may need $F'(\cdot, \cdot)$ for additional locations during the inductive proof. These locations will also be included in $B'(s)$. Based on $B'(s)$ we construct the template

$$T(s) := \forall (\vec{v}, \vec{l}) \in B'(s).\ F'(\vec{v}, \vec{l}). \tag{4.24}$$

In order to make sure that every element in B eventually ends up in $B'(s)$ we define the set

$$E(s) := \left\{ (\vec{v}, \vec{l}) \;\middle|\; (\vec{v}, \vec{l}) \in B \wedge \bigwedge_{i=1}^{n} \neg \text{fut-wrt}(\vec{l_i}) \right\} \tag{4.25}$$

which will make up part of $B'(s)$. Similarly to the checks we used in Sec. 4.2 we can also generate conditions under which we potentially write to a location l later on:

$$\begin{aligned} \text{fut-wrt}(l) :\equiv \exists w \in W, s'.\ & (\lceil w.\text{array} \rfloor_{s'} = \lceil l.\text{array} \rfloor_s) \wedge (\lceil w.\text{index} \rfloor_{s'} = \lceil l.\text{index} \rfloor_s) \wedge \\ & \underset{P}{\Longrightarrow}{}^{IC}_{s}\ (s') \wedge \text{loop-condition}(s) \end{aligned}$$

We want $B = E(s)$ when we exit the loop so we can show the loop postcondition. Since the future progressive invariant can be over-approximative we add the loop condition as an additional conjunct here. This makes fut-wrt(l) always false after we exit the loop and, thus, ensures that $B = E(s)$ when we exit the loop. Note that every $B'(s)$ contains the locations that are required in B but that are never written to during the execution of the loop, i.e. $F'(\cdot, \cdot)$ for these locations becomes a loop precondition.[1] Furthermore, there will typically be equalities between variables in $\vec{v}$, $\vec{l}$, and variables of state s' in the definition of $E(s)$, allowing us to eliminate many of these variables. For Example 6 we can derive

$$E(s) = \{(\texttt{q}, l) \mid l = \texttt{a[q]} \wedge 0 \leq \texttt{q} < \lceil \texttt{i} \rfloor_s\}.$$

[1] It can be useful to separate these locations into a second invariant to improve readability of the generate specifications.

The induction we described is over all array locations we write to. Not all of these locations are necessarily in B. We, therefore, have to additionally include at least the locations required by future induction hypotheses in $B'(s)$. We formulated the induction hypothesis

$$C_w(s) \wedge \bigwedge_{r \in \hat{R}_w} T(r; s)$$

earlier and, therefore, need to ensure that $R_w \subseteq B'(s)$ for each step. Since we are constructing a combined template for all $w \in W$ we will use

$$\hat{R} := \bigcup_{w \in W} \hat{R}_w. \tag{4.26}$$

We want to add locations that are part of $\hat{R}$ in a future iteration. We can formulate that a location l is needed as part of the induction hypothesis during the iteration starting from state s using the predicate

$$\text{req}(l, s) :\equiv \exists r \in \hat{R}. \left(\lceil r.\text{array} \rfloor_s = l.\text{array}\right) \wedge \left(\lceil r.\text{index} \rfloor_s = l.\text{index}\right). \tag{4.27}$$

Moreover, we only want to include l in $B'(s)$ once the point-wise postcondition $F'(\cdot, \cdot)$ has been established for that location. Similarly to how we constructed $E(s)$ from locations that have taken on their final values we include these additional locations once their value does not change (i.e. l is not written to) before we require it as part of the induction hypothesis in s. Analogously to $\text{req}(l, s)$ we can define

$$\text{wrt}(l, s) :\equiv \exists w \in W. \left(\lceil w.\text{array} \rfloor_s = l.\text{array}\right) \wedge \left(\lceil w.\text{index} \rfloor_s = l.\text{index}\right)$$

to check whether we write to a location in a particular iteration. Moreover, as we mentioned we want to be able to check whether such a write happens between two iterations s and s'. We can do this by checking whether there exists an iteration starting from s'', during which we write to l, and that *must* happen after s and before s'. In terms of progressive loop invariants we can formulate this property as

$$\text{betw}(l, s, s') := \exists s''. \text{wrt}(l, s'') \wedge \underset{\rightarrow s}{P}(s'') \wedge \underset{\Leftarrow s'}{P}(s''),$$

where $\underset{\Leftarrow s'}{P}(s'')$ is a strict under-approximative past progressive invariant. Analogously to how we defined strict over-approximative progressive loop invariants $\underset{\Leftarrow s'}{P}(s'')$ is only true for s'' that do not belong to the same iteration as s'. We use the strict progressive loop invariant here because the induction hypothesis has to hold before the iteration starts. Any writes that occur during the iteration, therefore, have no effect on it. We can now put all of these predicates together to formulate what requirements have already been established when we reach state s:

$$\text{ind}(l, s) = \exists s'. \text{req}(l, s') \wedge \neg \text{betw}(l, s, s') \tag{4.28}$$

Moreover, we define the set of these locations

$$h(s) = \{l \mid \text{ind}(l, s)\}$$

We discussed earlier that if the loop postcondition refers to multiple locations we have to consider every possible combination of locations. We can define the cartesian product

$$H(s) := h(s)^n = \left\{ \vec{l} \;\middle|\; \bigwedge_{i=1}^{n} \text{ind}(\vec{l_i}, s) \right\}$$

to represent these combinations. For Example 6 we get

$$H(s) = \{\text{a}[\lceil \texttt{i - 1} \rfloor_s]\}\,.$$

We, however, also need to allow for any combinations with locations in $E(s)$. Moreover, we need the locations $h(s)$ for our inductive proof independent of what the values of the quantified variables $\vec{v}$ are. We, therefore define $B'(s)$ as

$$B'(s) := \left\{ (\vec{v}, \vec{l}) \middle| \bigwedge_{i=1}^{n} \text{ind}(\vec{l_i}, s) \vee \exists (\vec{v'}, \vec{l'}) \in E(s).\vec{v} = \vec{v'} \wedge \vec{l_i} = \vec{l'}_i \right\}. \quad (4.29)$$

For Example 6, after some simplification, we get the template

$$T(s) = \forall \texttt{q}.\, 0 \leq q < i \longrightarrow \texttt{a[q] = sum(of: a\_orig, from: 0, to: q)}\,.$$

We then check that this invariant is inductive as we normally would during the calculation of a wlp: we check

$$T(s) \longrightarrow wlp(\text{loop-body}, T(s')) \quad (4.30)$$

using an SMT-solver. This check fails for Example 6 with the invariant we just identified. This is due to the dependencies that reach outside the loop. We can obtain the conditions under which (4.30) is violated by looking at

$$c(s) := T(s) \wedge \neg wlp(\text{loop-body}, T(s')). \quad (4.31)$$

For our example this simplifies to the condition

$$\texttt{a[i]} \neq \texttt{a\_orig[i]}. \quad (4.32)$$

This is a condition on a dependency outside the loop so we want to choose a C_w that ensures that this condition cannot hold. We can obtain such a C_w by generalizing its negation over the loop. Since we may write to these locations we, moreover, try to require the resulting constraints only for as long as we have to and drop them when they are no longer required in any future

iteration. Hence, we only generalize over future iterations. Accordingly we define

$$C(s) := \forall s'.\ \overrightarrow{P}_s(s') \longrightarrow \neg \mathrm{c}(s'). \tag{4.33}$$

Since locations in $C(s)$ are, by definition, never written to in earlier iterations it is clear that it suffices for $\overrightarrow{P}_s(s')$ to be monotone, i.e. for the set of states s' that satisfy it not to grow (cf. Def. 1), for $C(s)$ to be inductive.

If (4.31) contains constraints on recursive dependencies we can try to *refine* $T(s)$ by strengthening it. By only strengthening $T(s)$ we ensure that we are still able to prove that the loop postcondition holds after we exit the loop. The current implementation refines invariants by choosing the strongest possible invariant — false. Though this invariant is not necessarily the weakest possible invariant, this invariant fits well into the context of wlps since it allows us to verify the loop and the resulting loop precondition is still a sufficient precondition. Note that even if we do not require refinement the invariants we generate may not be as weak as they can be. This can be due to imprecisions in our progressive loop invariants or because a weaker invariant does not follow the schema used by our heuristic generation of templates.

Since (4.31) contains no such constraints in the case of our example we calculate

$$C(s) = \forall q.\ \mathtt{i} \leq q < \mathtt{a.length} \longrightarrow \mathtt{a}[q] = \mathtt{a\_orig}[q]. \tag{4.34}$$

We are then able to verify the loop with the inductive invariant $C(s) \wedge T(s)$. In the case of our example we get the loop precondition

$$\forall q.\ 0 \leq q < \mathtt{a.length} \longrightarrow \mathtt{a}[q] = \mathtt{a\_orig}[q]$$

which is guaranteed to hold because of the assignment on l. 2 of Example 6.

4.3.2 Extension to Multiple Loops

In this section we will extend the technique described in Sec. 4.3.1 to multiple loops. For successive loops this is relatively straight forward: we first calculate the wlp up to the postcondition of the last loop. We then use the technique described in Sec. 4.3.1 to infer an invariant for that loop. This invariant also has to hold before the loop is entered and, therefore, is also a precondition for the loop. We can then continue the wlp generation based on that precondition until we reach the next loop and repeat the process.

We will discuss nested loops based on an outer loop containing a single inner loop. The technique we describe can be applied recursively for deeper nestings and can similarly be extended to multiple successive inner loops. The idea for handling nested loops is that we push the obligation for the

inductive proof over the inner loop to the the inner loop. The outer loop simply collects the point-wise conditions generated by the inner loop ($E(s)$) and generates a proof ($H(s)$) only over locations it directly writes to (writes outside the inner loop).

To construct $E(s)$ we first need to extend the results of the data-flow-analysis for inner loops to the outer one. To distinguish the dependencies we identify for the inner and outer loop we will use W' to refer to the writes we identify for a single iteration of the inner loop and W for the outer loop. We over-approximatively extend the result for the inner loop using the over-approximative invariant $I(\cdot)$ of the inner loop. We write to a location l during the execution of the inner loop only if (i.e. all l that satisfy this formula are in W)

$$\exists s. I(s) \wedge l \in W'[\lceil \vec{x} \rfloor_s / \vec{x}],$$

where $\vec{x}$ are the local variables of the program. Additionally there may be locations the outer loop writes to directly that also end up in W. We can use this projection to correctly calculate the set $E(s)$ of locations whose value will not change during the execution of the remainder of the outer loop.

As mentioned above, because the inductive proof for locations written to by an inner loop is pushed to the inner loop we only calculate $\text{req}(l, s)$ over $\hat{R}$ for locations that the outer loop writes to directly. This requires us to be able to soundly approximate $\hat{R}$. If the outer loop writes to a location l after the inner loop is executed and R_l contains dependencies on locations that the inner loop writes to this requires us to generalize the dependencies of the inner loop. This requires us to calculate the transitive closure over the dependency graph of the inner loop which is not easily possible. We instead introduce a soundness condition shown in Def. 4 which we can check using an SMT-solver.

Theorem 4. *Soundness Condition*
If for every write to location l that, within a single iteration of the outer loop, that occurs after the inner loop

$$\nexists s. I(s) \wedge l \in W'[\lceil \vec{x} \rfloor_s / \vec{s}],$$

holds the dependencies we identify can be used to construct an inductive proof.

Based on $E(s)$ and $\text{req}(l, s)$ we can construct $B'(s)$ as we did before and obtain the invariant template

$$T(s) := \forall (\vec{v}, \vec{l}) \in B'(s).\ F'(\vec{v}, \vec{l})$$

for the outer loop analogously to the template we construct for loops with loop-free bodies. We then calculate the weakest liberal precondition over the

body of the outer loop. During this calculation we encounter the inner loop which we treat recursively based on the loop postcondition we calculate for the invariant template of the outer loop.

Once we have calculated wlp(outer-loop-body, T(s)) we can calculate $C(s)$ for the locations inner loops do not write to as we did before. Additionally we have some $C'(s)$ containing a precondition for an iteration of the outer loop that ensures that $C(s)$ of the inner loop is satisfied. We can obtain this $C'(s)$ by tracking the inner loops' $C(s)$ separately from the remaining preconditions during the calculation of wlp(outer-loop-body, T(s)). Contrary to what is the case for $C(s)$ there is no guarantee that locations referenced in $C'(s)$ are not written to in earlier iterations of the outer loop. We, therefore, have to explicitly check that

$$C(s) \wedge C'(s) \wedge T(s) \longrightarrow wlp(\text{outer-loop-body}, C'(s'))$$

holds, i.e. that $C'(s)$ is inductive. If this check fails we can try to *refine* our invariant. As mentioned before the current implementation simply refines every invariant to false.

4.4 Termination Analysis

In this section we will explore a technique that utilizes progressive loop invariants to automatically prove loop termination. The approach presented in this section can for certain kinds of loops give us a guarantee that they terminate.

The basic idea behind this technique is to show that a the execution of a loop must eventually reach a state from which it can only exit the loop. Using progressive loop invariants we can check that the execution of the loop cannot continue after reaching state s by showing that no state s' can succeed it, i.e. there is no s' that satisfies the future progressive invariant for s:

$$Q(s) :\equiv \nexists s'.\ \overset{\Longrightarrow}{P}{}^{DF}_{s}(s'). \tag{4.35}$$

We use the strict future progressive invariant here because were interested whether we can execute additional iterations *after* the current iteration. For Example 1 we can see that the state s where $\texttt{i} \mapsto \texttt{image.length} - 1$ and the strict future progressive invariant

$$\overset{\Longrightarrow}{P}{}^{DF}_{s}(s') = \lceil \texttt{i} \rfloor_{s'} < \lceil \texttt{i} \rfloor_{s'} < \texttt{image.length}$$

satisfies $Q(s)$.

In order to prove termination it is, however, not sufficient to find a state s for which $Q(s)$ holds but we also have to show that we are guaranteed to

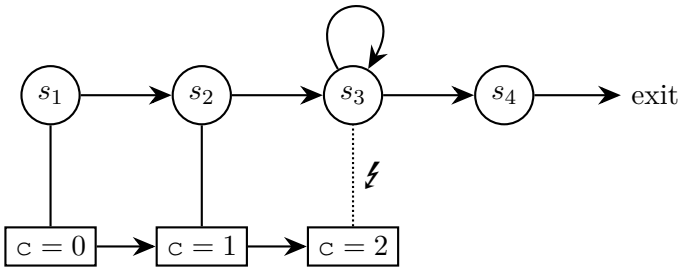

Figure 4.7: Illustration of a possible transition through states during the execution of a loop and the connection between states and the iteration counter $\subset$. The connection marked with ↯ is not possible since we may loop in s_3, i.e. there are multiple values of the iteration counter associated with s_3.

reach such an s during the execution of the loop. Fig. 4.7 illustrates this issue. We can find a state (s_4) from which we have to exit the loop. However, we may loop in s_3 forever and never reach s_4. One approach for solving this issue would be to under-approximate the states reached during the concrete execution of the loop (e.g. using [21] which under-approximates polyhedra) and show that it contains such an s. Such under-approximative numerical analyses are, however, not a standard. Moreover, we can resolve this problem using a different approach: iteration counters. Fig. 4.7 indicates how they can be useful. We know that the iteration counter never takes on the same value twice, i.e. they cannot loop in the way s_3 does in Fig. 4.7. If we are then able to connect each value of the iteration counter to a state during the execution of the loop as shown for s_1 and s_2 in Fig. 4.7 we know that we cannot loop on these states. In contrast to that we are not able to connect a value of an iteration counter to s_3 since we reach s_3 multiple times with different values of the iteration counter.

We automate this analysis by introducing an iteration counter ($\subset$) to the original program. We then try to show that at some point during the loop's execution *every possible* state satisfies $Q(s)$:

$$\exists i \geq 0. \forall s. \lceil \subset \rfloor_s = i \wedge I(s) \longrightarrow Q(s). \tag{4.36}$$

Lemma 5. *Soundness*
Any program that satisfies (4.36) terminates.

Proof. We want to assure ourselves that this check can never succeed for a loop that does not terminate and distinguish two cases of loops that do not terminate: we can (1) always keep executing additional iterations or (2) a single iteration does not terminate. We can see that in the first case the

iteration counter (c) reaches every non-negative value i and, hence, that there is always some state s that we reach during the concrete execution for which the left-hand-side of the implication is true. The check in (4.36) can, therefore, never succeed in case (1).

In case (2) the loop body must contain an inner loop or a method/function call that does not terminate. We handle such cases by recursively checking inner loops. Since method/function calls may be recursive we do not want to check their termination recursively. Instead we conservatively assume that they may not terminate which also allows the analysis to be modular. □

After introducing c we obtain

$$I(s) :\equiv (0 \leq \lceil \texttt{c} \rfloor_s = \lceil \texttt{i} \rfloor_s < \texttt{image.length})$$

for Example 1 and we can see that (4.36) is satisfied. Since the loop does not contain any nested loops or method/function calls we, therefore, know that the loop terminates.

In practice we distinguish three possible results of the analysis: (1) the loop terminates, (2) if we can show that all inner loops and method/function calls terminate we can also assume that the outer loop terminates, and (3) we do not know whether the loop terminates.

Note that, again, if $I(\cdot)$ and $\overset{\Longrightarrow DF}{P}_s(\cdot)$ are within Presburger arithmetic we can use Cooper's quantifier elimination technique [8] to eliminate the quantifiers in (4.36). If $I(\cdot)$ is generated using polyhedra it is guaranteed to be within Presburger arithmetic. As mentioned in Sec. 3.3.2 $\overset{\Longrightarrow DF}{P}_s(\cdot)$ is not guaranteed to be within Presburger arithmetic. We use it because we are interested in obtaining the best precision possible. We over-approximate any subexpressions of $\overset{\Longrightarrow DF}{P}_s(\cdot)$ which exceed Presburger arithmetic in order to eliminate the quantifiers in (4.36). Consequently we loose some but not all of the additional precision $\overset{\Longrightarrow DF}{P}_s(\cdot)$ gives us.

Chapter 5

Evaluation

An implementation in Sample [2] of the techniques described in this thesis is available at [1]. It uses Apron's [20] implementation of the polyhedra abstract domain for abstract interpretation and the SMT-solver Z3 [12] to aid in simplifying terms as well as to check various conditions (soundness-conditions, ordering, etc.). This implementation automatically infers over-approximative progressive loop invariants and strict progressive loop invariants that are generated using the technique described in Sec. 3. It, moreover, lets the user specify under-approximative progressive loop invariants which are used by the various analyses that rely on them. Furthermore, the implementation includes the extended inference for invariants and loop pre- and postconditions described in Sec. 4.1 and Sec. 4.2. It implements the automatic generation of inductive proves for functional postconditions described in Sec. 4.3. This analysis is implemented orthogonally to the inference of framing specifications, i.e. the framing of the functional invariants we generate has to be done manually. Moreover, it does not currently implement Skolemization for existential quantifiers and can, therefore, only be used for postconditions that do not contain existential quantifers. Furthermore, the implementation includes an implementation of the termination analysis described in Sec. 4.4.

We evaluate the techniques described in this thesis based on this implementation and a set of test programs written in the Viper intermediate language [22] and we use the Silicon verifier to check that the generated specifications are correct, i.e. that they let us verify the program. All tests were run on a 3.6 GHz Intel Core i9-9900K CPU running macOS 10.15.

In the remainder of this chapter we will look at the inference of framing specifications, functional specifications and at the termination analysis separately.

Program	LOC	Loops	Prec.	Prec. New	Time*	Time New
addLast	12	1 (1)	✓	✓	21	4 303
append	13	1 (1)	✓	✓	32	13 763
array1	17	2 (2)	X	✓	28	11 826
array2	23	3 (2)	X	✓	35	28 481
array3	23	2 (2)	✓	✓	24	25 733
arrayRev	18	1 (1)	✓	✓	28	24 404
bubbleSort	23	2 (2)	✓	✓	34	29 967
copy	16	2 (1)	✓	✓	27	25 151
copyEven	17	1 (1)	✓	✓	27	26 267
copyEven2	14	1 (1)	X	✓	20	21 956
copyEven3	14	1 (1)	✓	✓	23	24 973
copyOdd	21	2 (1)	✓	✓	55	58 386
copyOddBug	19	2 (1)	✓	✓	57	45 920
copyPart	17	2 (1)	✓	✓	30	38 762
countDown	21	3 (2)	✓	✓	32	46 965
diff	31	2 (2)	X	✓	70	11 394
find	19	1 (1)	✓	✓	43	20 693
findNonNull	19	1 (1)	✓	✓	40	22 154
init	18	2 (1)	✓	✓	28	36 300
init2d	23	2 (2)	✓	✓	52	65 450
initEven	18	2 (1)	X	✓	26	57 565
initEvenbug	18	2 (1)	X	✓	28	67 074
initNonCnst	18	2 (1)	✓	✓	27	43 013
initPart	19	2 (1)	✓	✓	30	48 891

Program	LOC	Loops	Prec.	Prec. New	Time*	Time New
initPartBug	19	2 (1)	✓	✓	31	9 351
insertSort	21	2 (2)	✓	✓	35	24 329
javaBubble	24	2 (2)	✓	✓	32	29 521
knapsack	21	2 (2)	X	X	45	61 796
lis	37	4 (2)	✓	✓	73	179 213
matrixmult	33	3 (3)	✓	✓	78	88 812
mergeinter	23	2 (1)	X	✓	56	117 311
mergeintbug	23	2 (1)	X	✓	59	129 545
memcopy	16	2 (1)	✓	✓	28	39 048
multarray	26	2 (2)	✓	✓	40	70 569
parcopy	20	2 (1)	✓	✓	30	44 150
pararray	20	1 (1)	✓	✓	31	46 909
parCopyEven	22	2 (1)	✓	✓	79	233 871
parMatrix	35	4 (2)	✓	✓	80	175 806
parNested	31	4 (2)	X	X	57	66 689
relax	33	1 (1)	✓	✓	55	56 578
reverse	21	2 (1)	✓	✓	42	267 022
reverseBug	21	2 (1)	✓	✓	42	272 241
sanfoundry	27	2 (1)	✓	✓	37	82 357
selectSort	26	2 (2)	X	✓	38	139 992
strCopy	16	2 (1)	X	X	21	72 783
strLen	10	1 (1)	X	X	15	23 174
swap	15	1 (1)	✓	✓	19	98 100
swapBug	15	1 (1)	✓	✓	19	72 126

* Times as reported by Dohrau et al. in [17]. These were generated using a different machine.

Figure 5.1: Experimental results for the test suite used by Dohrau et al. and comparison to their results. "LOC" indicates the lines of code for each example. "Loops" indicates the number of loops and the maximum nesting depth in parentheses. "Prec." and "Prec. New" respectively indicate whether the generated specifications are precise for the analysis described by Dohrau et al. and the one described in this thesis. All times are given in ms.

5.1 Framing Specifications

We will first compare the framing specifications we generate using the analysis described in this thesis to those generated by the analysis described by Dohrau et al., then look at some examples that violate their soundness condition and finally remark upon some general points.

5.1.1 Comparison to Dohrau et al.

Fig. 5.1 shows the results for running the analysis described in this thesis on the suite of examples used by Dohrau et al. For all examples our analysis is able to infer invariants which the analysis described by Dohrau et al. was not able to do and we can use Silicon to verify the resulting programs which

because the approach described by Dohrau et al. did not generate invariants was previously also not possible. Moreover, the soundness condition in Theorem 2 holds in every example, i.e. the progressive loop invariants we generate are reasonably precise and we do not simply generate false as specifications. This suite does not contain examples where the techniques described in Sec. 4.2 are required as the analysis described by Dohrau et al. cannot handle such cases. All experiments were, however, performed using the entire analysis.

In cases where our approach is imprecise, i.e. based on manual inspection of the generated invariants we could find an invariant that requires less permissions this stems from imprecisions in the numerical analysis which can potentially be alleviated by using a different abstract domain. We can, furthermore, see that the new analysis is precise in a couple of examples where the analysis described by Dohrau et al. is imprecise. This precision gain stems from the fact that the new analysis automatically instruments programs with *branch counters*, i.e. counters for each if-branch and else-branch in the body of the loop that count how often this branch is taken. Additionally there is a counter that counts the number of iterations of a loop we have executed. These counters allow the numerical analysis to infer more precise constraints e.g. if a program only accesses array elements with an even index we can capture that with a constraint like $\mathtt{i} = 2b$, where b is the iteration counter. Without such a counter polyhedra is not able to express such divisibility constraints.

We can, moreover, observe that the runtime of of the new analysis is much longer than that of the one described by Dohrau et al. This is in part due to the additional work the new analysis has to do to infer additional specifications. Additionally our analysis uses the SMT-solver Z3 to simplify terms which the analysis described by Dohrau et al. does not. For simple examples this method is much slower than the internal simplification used by Dohrau et al. (typically simplifications using Z3 account for 95-98% of the runtime of the inference). The results we obtain from Z3 are however simpler, benefiting both the readability of the generated specifications and in some cases reducing the (potentially exponential in the number of variables) blowup in the size of a term during the maximum elimination described by Dohrau et al. For some large examples this reduction in term size and accordingly the runtime of the maximum elimination can outweigh the overhead introduced by using Z3.

5.1.2 Further Examples

Over the course of this thesis I developed a test suite containing 30 tests for framing specifications. These test cases do not necessarily represent real-world examples but are interesting to look at as they show the limitations of the new analysis. The results for this test suite are shown in Fig. 5.2.

Program	Loops	# Acc.	Orig. Soundn.	Prec.	Time
basic1	1 (1)	1	✓	✓	12 919
basic2	1 (1)	1	✓	✓	8 593
basic3	1 (1)	1	✓	✓	11 722
basic4	1 (1)	1	✓	✓	20 619
advanced1	1 (1)	1	✓	X	1 642
advanced2	1 (1)	1	✓	✓	7 240
advanced3	1 (1)	1	✓	X	26 434
advanced4	1 (1)	1	✓	✓	300
advanced5	1 (1)	1	✓	✓	46 837
overapproximate1	1 (1)	1	✓	✓	53 105
overapproximate2	1 (1)	1	✓	✓	39 365
overapproximate3	1 (1)	1	✓	✓	6 338
imprecision	1 (1)	1	✓	X	6 478
doubleExhale1	1 (1)	2	X	✓	18 863
doubleExhale2	1 (1)	2	X	✓	22 053
doubleExhale3	1 (1)	2	X	✓	19 511
doubleExhale4	1 (1)	2	X	✓	29 054
doubleExhale5	1 (1)	2	X	✓	48 762
exhaleRequire1	1 (1)	3	X	✓	27 905
exhaleRequire2	1 (1)	3	X	✓	42 369
exhaleRequire3	1 (1)	3	X	✓	47 272
exhaleRequire4	1 (1)	3	X	✓	62 309
exhaleRequire5	1 (1)	3	X	✓	85 697
exhaleRequire6	1 (1)	3	X	✓	86 093
inhaleRequire1	1 (1)	3	X	✓	45 018
inhaleRequire2	1 (1)	3	X	✓	41 390
inhaleRequire3	1 (1)	3	X	✓	34 462
precision1	1 (1)	1	✓	✓	312 067
precision2	1 (1)	1	✓	✓	77 780
pseudoCycles	1 (1)	2	X	✓	43 565

Figure 5.2: Results for the test suite developed during this thesis. “Loops” gives the number of loops and the maximum nesting depth for each test case. “# Acc.” gives the maximum number of accesses to the same array location. “Orig. Soundn.” indicates whether the soundness condition described by Dohrau et al. is satisfied. “Prec.” indicates whether the generated specifications are reasonably precise. All times are given in ms.

We can see a couple of examples where we generate specifications that are imprecise, i.e. they require more permissions than is necessary. These are due to a violation of the soundness condition in Theorem 2. In such cases we generate false as an invariant and precondition. In most cases the soundness condition is violated because of imprecisions introduced during the abstract interpretation. Mostly, these imprecisions occur when the the value of the iteration variable does not behave monotonically since polyhedra is not able to capture constraints that relate earlier values of that variable to later ones in such cases. Additionally there are a couple of examples where the order graph (cf. Sec. 4.2) is cyclic and, therefore, violates the soundness condition in Theorem 4. In all test cases shown in Fig. 5.2 where that is the case this is due to a single statement accessing the same location multiple times during concrete executions and not due to imprecisions in the progressive loop invariants. I was not able to construct an example where such imprecisions would result in a cyclic order graph where the soundness condition described in Theorem 2 was not also violated.

Apart from violations of soundness conditions we can observe a couple of other examples where the generated specifications are imprecise. These are all due to imprecisions during the numerical analysis.

Program	Loops	Prec.	Their Time*	Our Time
init2d	2 (2)	✓	~40	12 518
append	1 (1)	✓	~20	7 953
copy	1 (1)	✓	~10	5 090
copyOdd	1 (1)	✗	~40	11 588
copyOddBug	1 (1)	✗	~50	7 983
copyPartial	1 (1)	✓	~10	4 653
find	1 (1)	✗	~20	8 219
findNonnull	1 (1)	✓	~20	10 501
init	1 (1)	✓	~10	2 618
initEven	1 (1)	✓	~40	4 939
initEvenBug	1 (1)	✓	~40	4 692
initNonConst	1 (1)	✓	~20	2 169
initPartial	1 (1)	✓	~10	2 333
initPartialBug	1 (1)	✓	~20	2 704
memcopy	1 (1)	✓	~40	4 109
mergeInter	2 (1)	✗	~90	22 151
mergeInterBug	2 (1)	✓	~110	20 719
reverse	2 (1)	✗	~30	17 401
reverseBug	2 (1)	✓	~40	18 189
strcopy	1 (1)	✓	~70	3 843
strlen	1 (1)	✓	~20	1 458

* Times as reported by Dillig et al. in [15]. These were generated using a different machine.

Figure 5.3: Experimental results for the inference of functional specifications based on the examples used by Dillig et al. "Loops" gives the number of loops and maximum nesting level each example contains. "Prec." indicates whether the specifications we generate are precise. "Their Time" gives the runtime reported by Dillig et al. and "Our Time" gives the runtime of our analysis. All times are given in ms.

5.1.3 Discussion

The framing specifications generated by the analysis described in this thesis for simple programs are generally relatively easily readable. For example we get the following invariant for Example 1:

$$\forall q_1.\mathbf{acc}(a[q_1], (\texttt{a.length} > q_1 \wedge 0 \leq q_1 ? (q_1 \leq \texttt{i} \wedge \texttt{i} \neq q_1 ? \frac{1}{2} : \mathbf{write}) : \mathbf{none}))$$

For more complicated programs especially those where permissions to the same location are `exhaled` in multiple iterations the generated specifications quickly become less readable.

5.2 Functional Specifications

5.2.1 Comparison to Dillig et al.

A technique described by Dillig et al. [15] automatically infers information about the values stored at different array locations. In contrast to our analysis it does this without the need for a user-provided postcondition. In turn their analysis is limited in the kinds of programs it can generate precise constraints for: it introduces a soundness condition that requires that if we write to an array location l in one loop iteration we may not read or write to l in any other iteration. This soundness condition checks that there are no read/write-dependencies across loop iterations. Our analysis does not

have this restriction, i.e. it can generate precise specifications for programs with arbitrary read/write-dependencies across loop iterations of a single loop. However, because our analysis relies on potentially imprecise progressive loop invariants and a heuristic template invariant generation the specifications it generates are also imprecise in some cases.

Fig. 5.3 compares the results of our analysis to those of the analysis described by Dillig et al. The table includes results for all examples used by Dillig et al. except for three examples that use arrays of arrays which the current implementation cannot handle and two examples for which we cannot easily formulate meaningful postconditions. The remaining examples are extended by manually providing the postconditions and under-approximative progressive loop invariants required by our analysis.

Additionally, I manually looked at the invariants and preconditions our analysis generates for each example to decide whether they are as precise as handwritten specifications. For most examples this means that we can infer the wlp true indicating that the program postcondition is guaranteed to hold for all program executions. Additionally, the suite contains some examples that contain bugs. Our analysis is able to correctly identify these bugs and either infers false as the precondition or a precondition that is strong enough to ensure that the buggy behaviour is not exhibited. The imprecisions in some of the examples stem from approximations we use during the analysis in order to be able to decide some quantified formulas. Disabling these approximations allows us to generate precise specifications for some of these examples. However, without these approximation the analysis does not terminate in some cases. Lastly we can observe that our analysis is a lot slower than that described by Dillig et al. A large portion (95-98%) of the runtime of our analysis can again be attributed to the simplifications performed by Z3.

5.2.2 Discussion

For simple examples the functional specification generated by the analysis described in this thesis are relatively easily readable. We can e.g. look at the "init" example used by Dillig et al. This example iterates over an array and initializes each element with some value `c` that is passed as a method parameter. For this example we provided the postcondition

$$\forall q.\ 0 \leq q < \texttt{a.length} \longrightarrow \texttt{a}[q] = \texttt{c}.$$

Our analysis then generates the loop invariant

$$\forall q.\ \texttt{a}[q] = \texttt{c} \vee 0 > q \vee \texttt{i} \leq q.$$

We can rewrite this invariant into an implication

$$\forall q.\ 0 \leq q < \texttt{i} \longrightarrow \texttt{a}[q] = \texttt{c}$$

Program	Loops	Term.	Res.	Res. w/o PI	Time
additionalStrength	1 (1)	✓	✓	X	118
basic1	1 (1)	✓	✓	✓	17
basic3	1 (1)	✓	✓	✓	16
basic4	1 (1)	X	X	X	9
basic5	1 (1)	✓	✓	✓	7
basic6	1 (1)	X	X	X	18
linear1	1 (1)	✓	✓	✓	29
linear2	1 (1)	✓	✓	✓	102
nested1	2 (2)	X	X	X	22
nested2	2 (2)	X	X	X	21
nested3	2 (2)	X	X	X	161
nested4	2 (2)	✓	X	X	211
nested5	2 (2)	✓	X	X	131
nested6	3 (2)	X	X	X	26
nested7	3 (3)	X	X	X	171

Figure 5.4: Experimental results for the termination analysis. "Loops" gives the number of loops and maximum nesting level each test contains. "Term." indicates whether a program is guaranteed to terminate in concrete executions. "Res." gives the result we obtain with the progressive loop invariant based analysis and "Res. w/o PI" gives the result we obtain using the alternative analysis based on unreachable iterations. All times are given in ms.

which is what we would typically choose when writing the invariant by hand. Moreover, we infer true as the precondition for this example which indicates that the postcondition is always satisfied. For more complicated examples the invariants we generate quickly become much less readable, however.

5.3 Termination Analysis

5.3.1 ExperimentalResults

Fig. 5.4 shows the results for the termination analysis based on its test suite. It also compares these results to those of an alternative analysis that we will introduce and discuss in Sec. 5.3.2. The tests contained in this suite do not access the heap resulting in much faster runtimes of the overall analysis. Similarly to what we discussed in Sec. 5.1 and Sec. 5.2 the progressive loop invariants are reasonably precise in cases where the iteration variables behave monotonically. Overall we can observe that in such cases the termination analysis is often able to prove termination (of programs that terminate in concrete executions).

5.3.2 Discussion

We will compare the approach described in Sec. 4.4 to two alternative approaches that are not based on progressive loop invariants. The first one is a simple analysis based on iteration counters and we will see that our analysis is stronger than it. The second alternative is existing work by Boralleras et al. [6] and we will see that many concepts of our analysis are similar to aspects of their analysis.

Unreachable Iterations

As we discussed before if we keep executing additional iterations the iteration counter reaches every non-negative value. Conversely if the loop terminates there has to be some value i that the iteration counter never takes on. We can approximate this check using

$$\exists i \geq 0. \nexists s. \lceil \circ \rfloor_s = i \wedge I(s) \tag{5.1}$$

and handle nested loops recursively as we did during the analysis described in Sec. 4.4. Similarly we also handle method and function calls by assuming that they may not terminate as we did in Sec. 4.4. The results for this analysis are given as "Res w/o PI" in Fig. 5.4.
The termination check of the analysis described in Sec. 4.4 is given as

$$\exists i \geq 0. \forall s. \lceil \circ \rfloor_s = i \wedge I(s) \longrightarrow \nexists s'. \overset{\Longrightarrow}{P}{}^{DF}_{s}(s'). \tag{4.36}$$

We can see that for any i that makes the outer existential in (5.1) true the left-hand-side of the implication in (4.36) is always false making the termination check of the analysis described in Sec. 4.4 true. Hence, the progressive loop invariant based approach is at least as strong as (5.1), i.e. it recognizes that a loop terminates in at least all of the cases that are recognized by (5.1). It is, moreover, clear that the progressive loop invariant based approach is stronger whenever $\overset{\Longrightarrow}{P}{}^{DF}_{s}(s')$ is the deciding factor. Based on the results shown in Fig. 5.4 we can see that in practice this additional strength can almost never be observed. There is a single example ("additionalStrength") where the analysis described in Sec. 4.4 is stronger.

This example was specifically designed to show this additional strength and is given in Example 7. In this example `j` grows exponentially whereas `i` only grows linearly. It is, therefore, clear that `j` will at some point be larger than `i` and the loop will terminate. Because polyhedra cannot represent exponentials, however, it will approximate `j` using some linear function. For example in my experiments the analysis lower-bounded `j` with some function with a slope of 8. Based on these constraints it looks like the difference between `i` and `j` might always stay the same. With the strict progressive loop invariant we are however able to show that all future values of `j` are at

Example 7. Progressive loop invariants show termination

```
1 var i = 60
2 var j = 1
3 while (j < i) {
4   i += 8
5   j *= 2
6 }
```

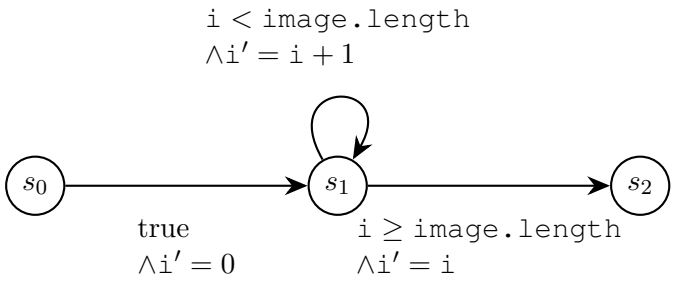

Figure 5.5: Control flow graph for Example 1. The initial state is s_0, s_1 represents states during the loop execution, and s_2 represents the final states of the program. The first line of the conditions for each edge give constraints under which the edge can be used. The second line restricts the post-states fo the edge by relating them back to the corresponding pre-states.

least twice as large as the current value. We, therefore, only have to show that the ratio $\frac{\mathtt{i}}{\mathtt{j}}$ will at some point be ≤ 2 to show that we exit the loop after the next iteration. This is possible with the linear constraints we have. Note that it is possible for `i` to grow so fast compared to the linear constraint we get for `j` that the ratio $\frac{\mathtt{i}}{\mathtt{j}}$ is always larger than 2.

We can, therefore, summarize that progressive loop invariants are, at least in some cases, useful in the context of termination analysis. We will look at some existing work in the domain of termination analysis next to see how progressive loop invariants relate to a more advanced technique.

Conditional Termination (Existing Work)

A technique proposed by Boralleras et al. [6] represents a program with its control flow graph (CFG) where each node represents a set of states and edges represent pieces of straight-line-code. The CFG for our Example 1 is shown in Fig. 5.5. Their analysis proves termination by trying to generate conditions under which each edge is only used finitely many times.

In their approach each edge is associated with a condition that indicates when the edge is used and that relates the pre- and post-states of that edge to each other. Conceptually this is somewhat similar to progressive

loop invariants. However, it should be noted that the conditions used by Boralleras et al. are exact but only relate the pre- and post-states of an edge to each other whereas progressive loop invariants are over-approximative and are able to relate any two states to each other. Based on these conditions Boralleras et al. are, then, able to construct *ranking functions* for each edge of the CFG. Ranking functions are integer-valued functions over the values of local (integer) variables whose value is (1) lower-bounded, (2) decreases each time the corresponding edge in the CFG is used, and (3) does not increase when any other edge is used. If such a function exists the corresponding edge can only be used finitely many times.

We can construct such a function based on a template which we adjust so the function satisfies the properties (1) - (3) listed above. Boralleras et al. propose linear combinations of program variables as a template for ranking functions and, then, try to find a coefficient for each variable to obtain a ranking function. E.g. for our running example a viable ranking function would be $\texttt{image.length} - \texttt{i}$. We can observe that these ranking functions have the same shape as constraints captured by polyhedra. Using progressive loop invariants we can, therefore, potentially capture the same information that this approach can capture.

The main advantage over our approach comes from the fact that the approach proposed by Boralleras et al. also allows them to find ranking functions that only work once some condition is met. The authors propose a method for filtering the CFG using these conditions and exclude the corresponding cases for which we have already proven termination. Thus we can successively eliminate additional conditions under which the program terminates until we are either able to show that the whole program terminates or we cannot find any further ranking functions. In future work it may be interesting to try obtaining conditions under which (4.36) holds which would allow us to apply a similar technique.

In summary, we note that progressive loop invariants are somewhat similar to concepts used by state of the art termination analyses. It, therefore, seems worthwhile to investigate further applications of progressive loop invariants for termination analyses in the future.

Chapter 6

Conclusion

We have introduced the concept of progressive loop invariants and seen that they are useful as the foundation of a wide range of analyses. We have, furthermore, seen that we can infer such progressive loop invariants using an abstract interpretation based approach. We then used these progressive loop invariants to extend existing work on framing specifications for loops enabling us to infer inductive loop invariants. Additionally we introduced a method for obtaining information about the order in which statements in a loop access the same array location. We introduced the order graph to represent this information and used it extend our analysis such that we can infer invariants and loop pre- and postconditions for a wide range of array programs. We, moreover, used the order graph as part of a technique that automatically generates inductive proofs for properties on the values stored in an array. Finally we introduced a technique that uses progressive loop invariants to prove that a loop terminates.

Furthermore, we have seen that using polyhedra we can infer sufficiently precise progressive loop invariants for a range of examples that allow the techniques we built based on progressive loop invariants to produce good results.

While the provided implementation uses the polyhedra abstract domain the techniques presented in this thesis are independent of the abstract domain that is used (however, it should be noted that the analyses profited from the fact that constraints generated using polyhedra are within Presburger arithmetic in order to efficiently eliminate quantifiers and maximum-expressions). These techniques can, therefore, be adapted to other abstract domains and directly benefit from future developments in the field of abstract interpretation.

6.1 Future Work

There is a wide range of directions for the techniques described in this thesis that are worth exploring in the future. One such direction is handling cyclic order graphs. Possible approaches to this issue include techniques to upper-bound the number of times the cycle occurs in concrete executions or showing that the cycle has no effect (e.g. that the total amount of permission gain/ loss is always 0). As discussed earlier the specifications for programs that `exhale` from the same location in multiple iterations are not necessarily as precise as they can be. In future work it may be useful to look at techniques to improve the precision of these specifications, including trying to split `exhale` statements (e.g. into one that `exhale`s from even indices and one that `exhale`s from odd ones) in the order graph to allow us to further maximize the overlap between `exhale`s that are maximized together.

The inference for functional specifications could, furthermore, benefit from improved refinement techniques. The current implementation, moreover, does not preserve the triggers of the user-provided postcondition. Preserving these triggers may help both during the inductivity checks performed during the inference and when verifying the program extended with the inferred specifications. Another approach for inferring framing specifications is to use the dependencies we determine to produce recursive functions. These functions would rely only on the initial values of the array and, thus, eliminate explicit dependencies on the values of other array elements. This approach has the advantage of not requiring user-provided postcondition. The reason it was not pursued within the scope of this thesis is that it requires the verifier to generate an inductive proof for many interesting properties. It may however still be interesting to explore further in the future. Conversely, recursive programs can exhibit similar dependency structures as those captured by the dependency graph used by the inference for functional specifications. Consequently it may be worth exploring how a similar analysis could be used to generate inductive proofs over recursive programs.

In the context of the termination analysis it may be interesting to try splitting the program similarly to what Borralleras et al. describe. Analogously to how we used over-approximative progressive loop invariants to prove that at some point there is no state left that allows us to continue executing the loop it may, furthermore, be possible to use under-approximative progressive loop invariants to show that we are guaranteed to never reach such a state and, therefore, prove loop non-termination.

The techniques described in this thesis are imprecise if arrays are accessed at indices that are not expressible in Presburger arithmetic or that depend on heap values. Furthermore, they cannot handle overlapping arrays or arrays of arrays. In the future it might be worth investigating if and how these limitations can be overcome. Additionally we mentioned that the specifications we generate are sometimes hard for the user to understand.

Improved simplification techniques may help in such cases.

We used user-provided under-approximative progressive loop invariants at several point during the analyses described in this thesis. Implementing an automatic inference for such progressive loop invariants is another direction for future work. Lastly it may be worth exploring additional applications where progressive loop invariants are useful. We have seen how we can use progressive loop invariants to determine dependencies between array values. Such information may, e.g. be useful for code optimization (e.g. [18]) or parallelization (e.g. [29]).

References

1. https://bitbucket.org/nilsbecker_/sample/src/default/. Online; accessed 24-Mar-2020.

2. Sample. https://www.pm.inf.ethz.ch/research/sample.html. Online; accessed 24-Mar-2020.

3. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, Mar. 1976.

4. M. Barnett and R. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM Press, September 2005.

5. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Found. Trends Program. Lang.*, 2(2–3):71–190, Dec. 2015.

6. C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination through conditional termination. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–117, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

7. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

8. D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

9. 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, pages 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

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

11. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, Mar. 1997.

12. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

13. D. Déharbe, P. Fontaine, and B. W. Paleo. Quantifier inference rules for smt proofs. In *First International Workshop on Proof eXchange for Theorem Proving – PxTP*, Wroclaw, Poland, 2011.

14. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

15. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In A. D. Gordon, editor, *Programming Languages and Systems*, pages 246–266, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

16. I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. volume 48, pages 443–456, 10 2013.

17. J. Dohrau, A. J. Summers, C. Urban, S. Münger, and P. Müller. Permission inference for array programs. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification (CAV)*, volume 10982 of *LNCS*, pages 55–74. Springer-Verlag, 2018.

18. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

19. C. A. Furia and B. Meyer. *Inferring Loop Invariants Using Postconditions*, pages 277–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

20. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 661–667, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

21. A. Miné. Inferring sufficient conditions with backward polyhedral underapproximations. *Electronic Notes in Theoretical Computer Science*, 287:89 – 100, 2012. Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012.

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

23. P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

24. S. Padhi, R. Sharma, and T. Millstein. Loopinvgen: A loop invariant generator based on precondition inference. *CoRR*, abs/1707.02029, 2018.

25. M. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), January 2012.

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

27. D. A. Schmidt. Closed and logical relations for over- and under-approximation of powersets. In R. Giacobazzi, editor, *Static Analysis*, pages 22–37, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

28. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

29. Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations: Extended version. *International Journal of Parallel Programming*, 23(4):359–388, 1995.

List of Figures

List of Examples

Symbols Glossary

$P(s, s')$	A set of constraints that relate an earlier state s to a later state s'.
$I(s)$	An over-approximative invariant that indicates whether we can reach s at the start of an iteration during the execution of a loop.
$\overrightarrow{P}_s(s')$	The *future progressive invariant* indicates that the iteration starting from s' occurs after the one starting from s (cf. also (3.4) in Sec. 3.2.1).
$\overleftarrow{P}_s(s')$	The *past progressive invariant* indicates that the iteration starting from s' occurs before the one starting from s (cf. also (3.6) in Sec. 3.2.2).
$\overrightarrow{\overrightarrow{P}}_s(s')$	The *strict future progressive invariant* indicates that the iteration starting from s' is a "true successor" to the iteration starting from s. In contrast to $\overrightarrow{P}_s(s')$ s' may not occur in the same iteration as s (cf. also (3.8) in Sec. 3.3.1 and (3.10) in Sec. 3.3.2).
$\overleftarrow{\overleftarrow{P}}_s(s')$	The *strict past progressive invariant* indicates that the iteration starting from s' is a "true predecessor" to the iteration starting from s. In contrast to $\overleftarrow{P}_s(s')$ s' may not occur in the same iteration as s (cf. also (3.9) in Sec. 3.3.1 and (3.11) in Sec. 3.3.2).
$l(s)$	The permission loss up to (but not including) the iteration starting from state s (cf. also (4.1) in Sec. 4.1.1 and (4.14) in Sec. 4.2.2).

$g(s)$	The permission gain up to (but not including) the iteration starting from state s (cf. also (4.11) in Sec. 4.1.2 and (4.15) in Sec. 4.2.2).
l	The total permission loss over the execution of the entire loop (cf. also (4.16 in Sec. 4.2.2).
g	The total permission gain over the execution of the entire loop (cf. also (4.17 in Sec. 4.2.2).
$r(s)$	The amount of permissions required to execute the single iteration starting from state s.
$r'(s)$	The amount of permissions required at the beginning of the loop to execute the single iteration starting from s after accounting for the permission gain and loss up to s (cf. also (4.13) in Sec. 4.2.2).
$\mathcal{A}$	The universe of arrays.
$\mathcal{L}$	The universe of array locations is defined as pairs of arrays and indices ($\mathcal{L} := \mathcal{A} \times \mathbb{Z}$).
W	The set of array locations we write to during a single loop iteration.
R_w	The set of array locations that are data dependencies for the value we write to $w \in W$.
$\hat{R}_w$	The set of *recursive dependencies*, i.e. dependencies on values we generate during the execution of the loop for location $w \in W$ (cf. also (4.18) in Sec. 4.3.1).
$\hat{R}$	The set of all *recursive dependencies*, i.e. the union of all $\hat{R}_w$ for $w \in W$ for a single loop iteration (cf. also (4.26) in Sec. 4.3.1).
$C_w(s)$	The *precondition requirements* are a logical expression containing requirements for dependencies that reach outside the loop, i.e. dependencies on values that are not produced by the loop (cf. also Sec. 4.3).
$C(s)$	The combined *precondition requirements* for all writes in an iteration (cf. also (4.33) in Sec. 4.3.1).

$T(x; s)$	The *template invariant* is the basis of the inductive proof we generate for a loop postcondition. It is constructed heuristically based on the loop postcondition (cf. also Sec. 4.3).
$T(s)$	The combined invariant template for a single loop iteration (cf. also (4.24) in Sec. 4.3.1).
$F(\vec{v})$	The point-wise loop postcondition gives the loop postcondition for each assignment of the quantified variables $\vec{v}$ (cf. also (4.21) in Sec. 4.3.1).
$F'(\vec{v}, \vec{l})$	Point-wise loop postcondition that is additionally parameterized in the array locations $\vec{l}$ it references (cf. also (4.22 in Sec. 4.3.1).
B	Set containing tuples $(\vec{v}, \vec{l})$ with all possible assignments of the quantified variables and array locations used in the loop postcondition (cf. also (4.22) in Sec. 4.3.1).
$B'(s)$	Set containing tuples $(\vec{v}, \vec{l})$ with assignments of the quantified variables and array locations we construct the invariant template over (cf. also (4.29) in Sec. 4.3.1).
$E(s)$	The set containing elements of B that have been finalized, i.e. that we will not write to during the remainder of the loop (cf. also (4.25) in Sec. 4.3.1).
$\text{req}(l, s)$	Indicates whether the induction hypothesis of the proof we are constructing refers to location l in the iteration starting from state s (cf. also (4.27) in Sec. 4.3.1).
$\text{betw}(l, s, s')$	Indicates whether location l is guaranteed to be written to between the time we reach s and the time we reach s' (cf. also (4.32) in Sec. 4.3.1).
$\text{ind}(l, s)$	Indicates whether we need the template invariant for location l as part of our inductive proof in the future and whether we have already established the template invariant for l (cf. also (4.28) in Sec. 4.3.1).
$Q(s)$	indicates whether we are guaranteed to exit the loop after reaching s (cf. also (4.35) in Sec. 4.4).

Appendix A

Monotonicity Using Polyhedra

In Sec. 4.1 we introduced the concept of monotonicity for progressive loop invariants. We will now discuss under which circumstances the progressive loop invariants generated using polyhedra are monotone. For convenience the corresponding definition is repeated here.

Definition 1. *Monotonicity*
Let $\overleftarrow{S}_s := \left\{ s' \middle| \overleftarrow{P}_s(s') \right\}$ *and* $\overrightarrow{S}_s := \left\{ s' \middle| \overrightarrow{P}_s(s') \right\}$ *be the set of states satisfying the past progressive invariant and future progressive invariant of state* s *respectively. A family of past progressive invariants is called monotone iff*

$$\forall s, s'.\, \mathrm{suc}(s, s') \longrightarrow \overleftarrow{S}_s \subseteq \overleftarrow{S}_{s'}.$$

A family of future progressive invariants is called monotone iff

$$\forall s, s'.\, \mathrm{suc}(s, s') \wedge I(s) \longrightarrow \overrightarrow{S}_s \supseteq \overrightarrow{S}_{s'}$$

Fig. A.1 shows the state space we analyze for the instrumented version of a simple program. The monotonicity property requires that the horizontal slice (past progressive invariant) does not shrink when following arrows and that the vertical slice (future progressive invariant) does not grow when following arrows. This has to be the case for the concrete states (indicated by points in the diagram) since conceptually we can see that the set of past states grows while the set of future states shrinks as we progress through the loop. It is therefore oftentimes the case that the progressive loop invariants we generate are also monotone.

In the optimal case polyhedra finds the convex hull of these concrete sets. The shaded area in Fig. A.1 indicates the convex hull. We can see that, as we follow the arrows, the horizontal slice of the convex hull does not get narrower and the vertical slice of the convex hull does not get wider, i.e. progressive

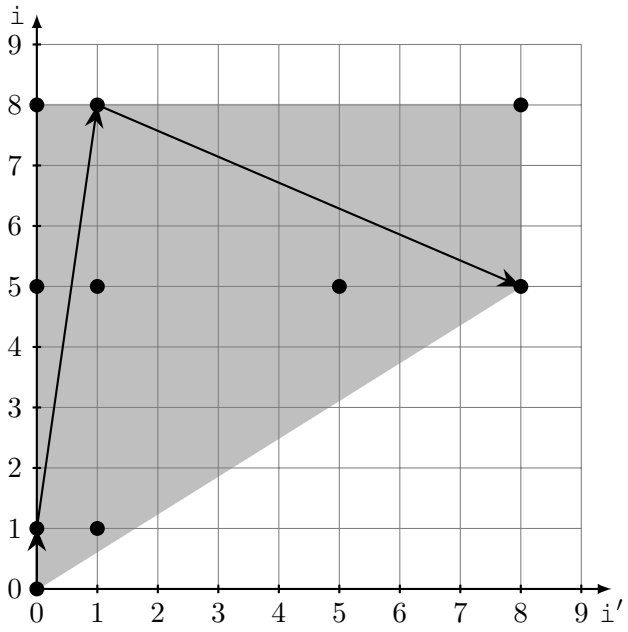

Figure A.1: Instrumented numerical states for a program where the local variable i successively takes on the values 0, 1, 8, and 5. Similarly to the instrumentation we used in Chap. 3 $\mathtt{i}'$ stores previous values of i. The progressive loop invariant inference technique can be interpreted as taking a horizontal slice of the graph for past progressive invariants and a vertical slice for future progressive invariants. For example if we are in a state where $\mathtt{i} = 8$ we find points on the line $\mathtt{i} = 8$ to find previous values of i and points on the line $\mathtt{i}' = 8$ to find future values of i. Arrows show an execution where $\mathtt{i}'$ stores the value of i from the previous iteration, i.e. the points reached correspond to $\mathrm{suc}(\mathtt{i}', \mathtt{i})$. The convex hull of the concrete states is indicated by the shaded area.

loop invariants based on the convex hull are also monotone. In general we can see that the monotonicity properties relate to the orientation of the edges of the convex hull in the plane. For the future progressive invariant we look at the edges that are above and below the arrow we follow and for the past progressive invariant we look at the edges that are left and right of the arrow. The top and bottom edges have to be parallel to the $\mathtt{i}'$-axis or angled toward the the arrow we follow for the set of states satisfying the future progressive invariant to not get bigger, i.e. for the future progressive invariant to be monotone. Analogously the left and right edges have to be parallel to the i-axis or angled away from the arrow for the set of states that satisfy the past progressive invariant to not become smaller, i.e. for the past progressive invariant to be monotone.

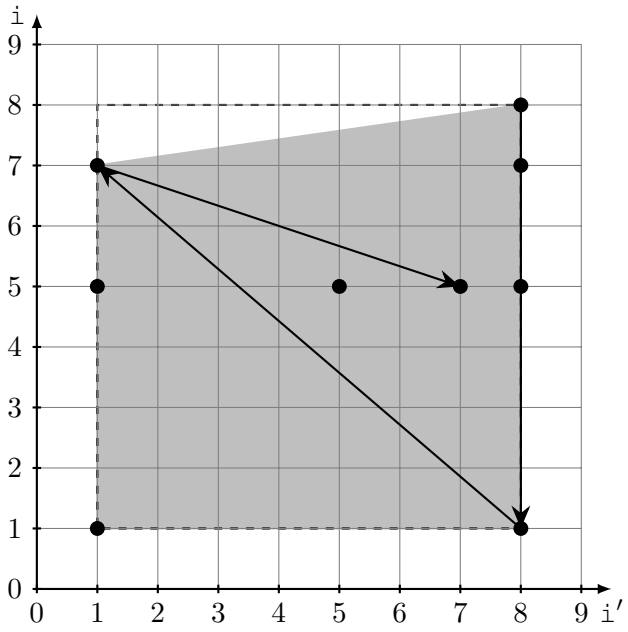

Figure A.2: Instrumented numerical states for a program where the local variable i successively takes on the values 8, 1, 7, and 5. The area indicated by the dashed line is what we typically get using polyhedra.

Since the concrete sets of past states (represented by points in the diagram) is monotone as we follow an arrow we add points left or right of where we end up. If this point falls within the interval between existing points (e.g. as is the case for the point (5, 5) in Fig. A.1) the left and right edges of the convex hull simply connect the outermost points vertically to their equivalents at the start of the arrow, i.e. the left and right edges of the convex hull are parallel to the i-axis. If a new point falls outside this interval the corresponding edge of the convex hull must angle away from the arrow to enclose that point.

For the concrete set of future states we take points away as we follow an arrow. We cannot argue analogously to how we argued for the past states however. Fig. A.2 shows a simple example where the set of concrete future states behaves monotonically but the convex hull does not. While polyhedra can, in principal, find this convex hull experience shows that polyhedra usually finds weaker constraints for examples where i does not increase or decrease monotonically especially if widening is used. For the example in Fig. A.2 we would typically expect to get

$$1 \leq \mathtt{i} \leq 8 \wedge 1 \leq \mathtt{i}' \leq 8.$$

This constraint corresponds to the area indicated by the dashed line in

Fig. A.2. Constraints like these that do not relate $\texttt{i}'$ to $\texttt{i}$ are always parallel to the coordinate-axes.

For cases where $\texttt{i}$ increases or decreases monotonically we can argue analogously to how we argued for past progressive invariants that if we take away a point such that there still remain points above and below it the top and bottom constraints are parallel to the $\texttt{i}'$ axis and if we take away a point on the edge of that interval the corresponding constraint is angled towards the arrow.

We can, moreover, extend these deliberations to programs with multiple local variables, i.e. higher dimensional spaces. Furthermore, we note that if the constraints we identify are oriented the way described above and the local variables behave monotonically we can use widening which typically means removing a constraint, i.e. an edge in the diagram and the remaining constraints will still have to be oriented correctly. We can see this based on the fact that the orientation to all arrows is the same for each constraint if the local variables are monotone.

Lastly we should note that the constraints we generate using polyhedra are not guaranteed to (and in many cases do not) correspond to the convex hull. The convex hull, however, gives the most precise constraints that polyhedra can find and any implementation will try to find constraints as close to it as possible. We have seen that the convex hull guarantees that we construct monotone past progressive invariants but *not* that we construct monotone future progressive invariants. In practice the progressive loop invariants we construct from polyhedra are almost always monotone (e.g. the progressive loop invariants for every example in the test suites we discussed in Sec. 5 are monotone).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Inference of Progressive Loop Invariants for Array Programs

Authored by (in block letters):

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

Name(s):	**First name(s):**
Becker	Nils

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)**
Winterthur, 26. Mar. 2020	NB Becker

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