Automatic Inference of Permission Specifications

Jérôme Dohrau • DISS. ETH NO. 28756 • 2022

DISS. ETH NO. 28756

Automatic Inference of Permission Specifications

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich

(Dr. sc. ETH Zurich)

presented by

Jérôme Dohrau

MSc ETH CS, ETH Zurich

born on March 15, 1989

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner

Prof. Dr. James Brotherston, co-examiner

Prof. Dr. Reiner Hähnle, co-examiner.

2022

Für min Keks

Abstract

Deductive program verification tools require users to provide annotations such as method preconditions and postconditions and loop invariants. For permission-based verifiers, a substantial part of these annotations express which heap locations a method or a loop may access, for instance, via points-to assertions, quantified permissions, and recursive predicates. These *permission specifications* pose a significant annotation burden for programmers; it is, thus, highly desirable to infer permission specifications automatically. Moreover, information about the memory accessed by parts of a program also finds use in other important applications, such as compiler optimisations and program parallelisation. In this thesis, we advance the state of the art of automatic permission inference by proposing two novel inference techniques.

First, we present a permission inference for the important class of array manipulating programs. This approach represents permission specifications using numerical expressions and employs a precise backwards analysis for loop-free code. Permissions required by loops are expressed via maximum expressions over the results obtained for individual loop iterations. These maximum expressions are then solved by a novel maximum elimination algorithm, in the spirit of quantifier elimination. This first approach is proven sound.

Second, we present a learning-based permission inference for programs operating on individual heap locations and recursively defined data structures. The inference builds on the ICE framework, which alternates between a teacher to generate constraints on the desired specifications and a learner to solve these constraints to generate candidate specifications to be checked by the teacher. We express constraints over entire traces rather than individual states, which allows us to simultaneously infer all necessary preconditions, postconditions, loop invariants, and definitions of recursive predicates. Our inference targets automated permission-based verifiers, which typically treat predicates isorecursively, that is, distinguish between a predicate and its body. In particular, we automatically infer the ghost operations required by such verifiers to unfold and fold predicate instances. This approach is black-box, meaning that the teacher uses a verifier to check the specifications and does not need to be cognisant of the program's semantics, and the learner relies solely on the

information communicated by the teacher. Moreover, this second approach is sound by construction, assuming that the verifier used by the teacher is sound.

Both our inference techniques are implemented based on the Viper verification infrastructure. An experimental evaluation for each of them demonstrates that they are effective on a wide range of examples.

Zusammenfassung

Deduktive Programmverifikationstools erfordern, dass Benutzer Annotationen – wie zum Beispiel Methodenvorbedingungen und -nachbedingungen oder Schleifeninvarianten – manuell bereitstellen. Bei berechtigungsbasierter Verifikation (zum Beispiel mit Separationslogik) drückt ein wesentlicher Teil dieser Annotationen lediglich aus, auf welche Speicherorte eine Methode oder Schleife zugreifen darf; beispielsweise mittels sogenannten Points-to-Assertions, quantifizierten Berechtigungen oder rekursiven Prädikaten. Das Schreiben solcher Spezifikationen für Zugriffsberechtigungen ist mühselig und beansprucht viel Zeit. Daher ist es wünschenswert, erforderliche Zugriffsberechtigungen automatisch zu inferieren. Abgesehen von Programmverifikation finden Informationen über mögliche Speicherzugriffe auch Anwendung in anderen wichtigen Bereichen, wie zum Beispiel Kompilationsoptimierungen oder Programparallelisierung. In dieser Dissertation erweitern wir den Stand der Technik der automatischen Inferenz benötigter Zugriffsberechtigungen, indem wir zwei neue Inferenztechniken vorstellen.

Zuerst präsentieren wir eine Inferenz für die wichtige Klasse von Programmen, die Arrays manipulieren. Dieser Ansatz nutzt numerische Ausdrücke, um Zugriffsberechtigungen darzustellen, und bedient sich einer präzisen Rückwärtsanalyse für schleifenfreien Code. Zugriffsberechtigungen für Schleifen werden mittels Maxima über die Ausdrücke, die für individuelle Ausführungen des Schleifenrumpfes inferiert wurden, ausgedrückt. Diese Maxima werden dann mittels eines neuartigen Maximaeliminationsalgorithmus, ähnlich einer Quantorenelimination, umgeformt. Zudem beweisen wir die Korrektheit dieses ersten Ansatzes.

Zweitens präsentieren wir eine lernbasierte Inferenz für Programme, die auf individuelle Speicherorte zugreifen und auf rekursiven Datenstrukturen operieren. Diese Technik baut auf dem ICE Framework auf, welches zwischen zwei Komponenten, Lehrer und Schüler genannt, alterniert: Der Lehrer generiert Bedingungen für die gewünschten Zugriffsberechtigungen, worauf der Schüler aufgrund aller bereits gesammelten Bedingungen neue Kandidaten für die Zugriffsberechtigungen vorschlägt, die wiederum vom Lehrer überprüft werden. Die generierten Bedingungen beziehen sich anstelle von einzelnen Programmzuständen auf gesamte Programmausführungen.

Dies erlaubt es uns alle notwendigen Methodenvorbedingungen und -nachbedingungen, Schleifeninvarianten, sowie auch die Definitionen rekursiver Prädikate gleichzeitig zu inferieren. Da berechtigungsbasierte Verifikationstools typischerweise mit isorekursiven Prädikaten arbeiten – das heißt, zwischen einer Instanz eines Prädikats und dessen Rumpf unterscheiden – generiert unsere Inferenz auch alle Hilfsoperationen, die zum Falten und Entfalten von Prädikatsinstanzen benötigt werden. Unser zweiter Ansatz ist Black-Box; das bedeutet, der Lehrer benutzt ein Verifikationstool zum überprüfen der Spezifikationen und muss sich der Programmsemantik nicht bewusst sein und der Schüler stützt sich ausschließlich auf die vom Lehrer zur Verfügung gestellte Information. Die Korrektheit dieses Ansatzes beruht auf seiner Bauart und setzt voraus, dass das eingesetzte Verifikationstool fehlerfrei arbeitet.

Beide Inferenztechniken sind auf Basis der Viper Verifikationsinfrastruktur implementiert. Eine experimentelle Evaluation demonstriert ihre Effektivität auf einer breiten Auswahl von Beispielen.

Acknowledgements

The journey through my doctoral studies – with all its ups and downs – would not have been possible without the help and support of many friends and colleagues. I take this opportunity to thank everyone who contributed in one way or another. Without *your* support, I would not have arrived where I am now. Thank you!

First and foremost, I would like to express my gratitude towards Peter Müller. I am thankful to have had a supervisor, who always provided honest and encouraging input, gave me lots of freedom when I needed it, and made me feel supported at all times. I thank my co-examiners James Brotherston and Reiner Hähnle for taking the time to review my thesis and for their thorough and valuable feedback.

Many thanks go to Alexander Summers and Caterina Urban for their excellent collaboration during the early days of my PhD; I could learn a lot from the two of you. I also especially thank Malte Schwerhoff for the frequent and enlightening discussions in tha later stages of my PhD, and for helping me better understand the inner workings of the Silicon verifier.

Thanks to Marco Eilers for all the good times, for giving me a reason to not give up when I did not see a way, and for being my dear friend. I would not have made it without you!

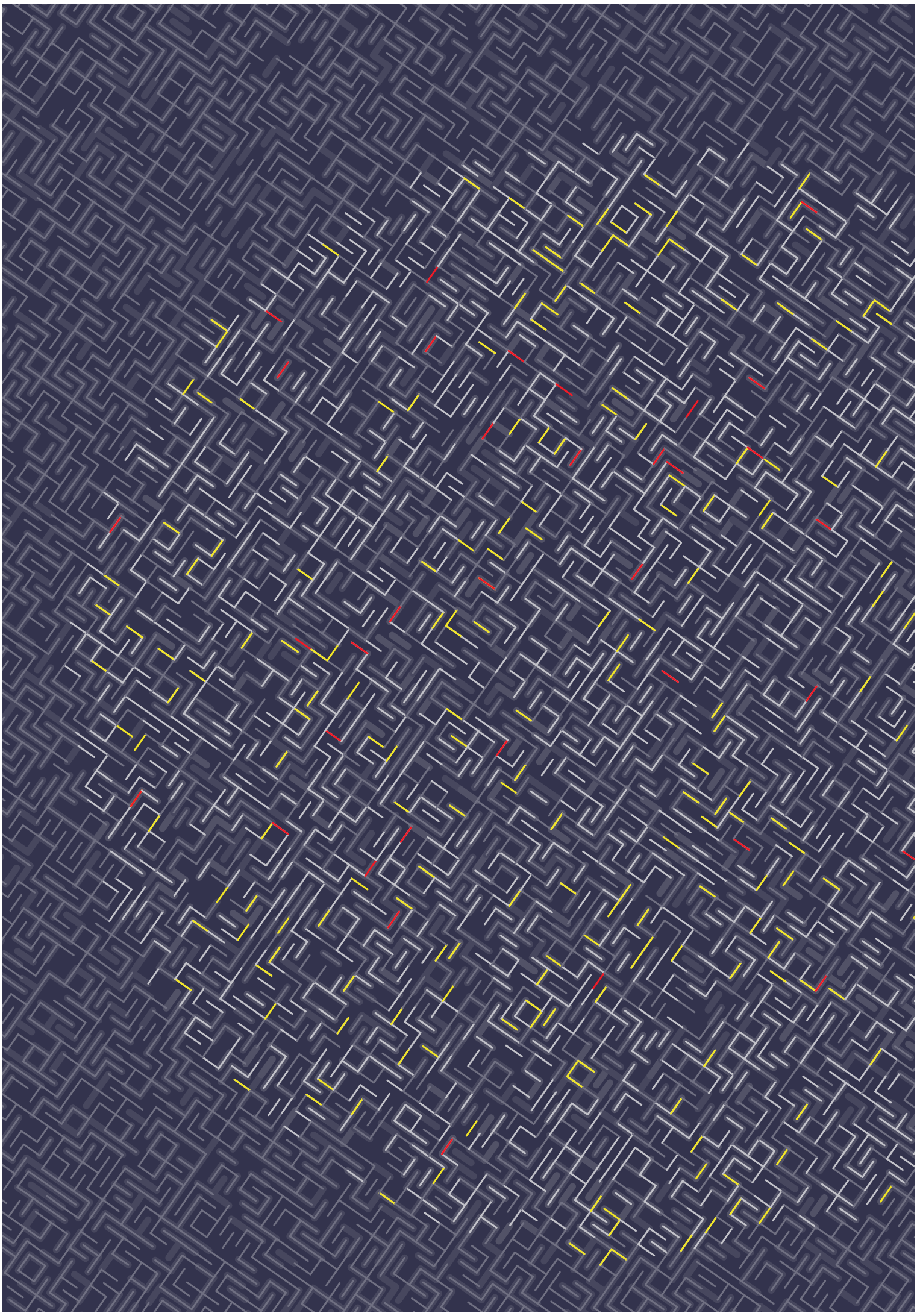
Throughout the past years, I was fortunate to be part of a great and caring research group. I will keep all our evenings at Minimum, board game sessions, ski trips, weddings, retreats, group lunches, occasional coffees and beers, research meetings, actual research meetings, paper deadlines, conference and summer school visits, and all other special and ordinary moments in good memory. With this in mind, I extend my thanks to Linard Arquint, Dimitar Asenov, Vytautas Astrauskas, Aurel Bílý, Lucas Brutschy, Alexandra Bugariu, Martin Clochard, Thibault Dardinier, Jonás Fiala, Uri Juhasz, Christoph Matheja, Wytse Oortwijn, Fábio Pakk Selmi-Dei, Gaurav Parthasarathy, João Carlos Mendes Pereira, Federico Poli, Dionisios Spiliopoulos, Arshavir Ter-Gabrielyan, Felix Wolf, and Valentin Wüstholz. Special thanks to Marlies Weissert and Sandra Schneider for making sure our group stayed organised. I wish all the best to all of you and hope to stay in touch!

I would also like to thank all Bachelor's and Master's students that I had the pleasure of supervising: Nils Becker, Mathias Blarer, Lowis Engel, Christian Knabenhans, and Flurin Rindisbacher.

I am infinitely thankful to all my friends and my family for keeping me sane throughout these years. Above all, I could not be happier to have crossed paths with Gabriela Brack, who gave me the strength to persevere and ultimately succeed. I cannot imagine a life without you. Thank you for everything!

Contents

Figures, Tables, and Listings



1 Introduction

Software is ubiquitous and governs almost every aspect of our everyday life. As a result, the consequences of a computer program crashing or not behaving as intended can be quite severe; in extreme cases, they can entail major financial losses or even be a matter of life and death. To be more confident about the correctness of their program, software developers employ various techniques, such as following established practices, adopting pair programming, conducting code reviews, and writing automated tests. However – as Dijkstra pointed out in 1972 – this is not enough to be certain about a program's correctness: *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness"* [34]. In order for such proofs to be feasible for real world programs, it is essential for the proof technique to be amenable to automation. To this end, there have been considerable efforts developing *deductive verification* tools [2, 8, 12, 37, 59, 70, 71, 83, 89, 97] that allow one to prove programs correct with respect to complex specifications.

Permission-Based Verification. To enable modular reasoning about heap-manipulating programs, many verifiers are based on separation logic or other *permission logics* [19, 91, 92, 98]. In these logics, a method may access a shared memory location only if it holds the *permission* associated with this location. The permission is first created when the location is allocated, and then may be transferred between different methods. The permissions to be transferred upon a method call and return are specified as part of the callee's precondition and postcondition, respectively. As permissions cannot be duplicated, any caller holding on to a permission during a method call can safely assume that the value of the respective heap location does *not* change, since the called method (and any recursively called method, for that matter) is guaranteed to have no permission to write this heap location. Thus, all methods can be reasoned about in isolation, taking into account only the specifications – that is, preconditions and postconditions – of other methods.

Moreover, permission logics naturally extend to a concurrent setting [85]: A fork statement may transfer permissions to the new thread, and back when the thread is

joined. Acquiring a lock temporarily obtains permission to access certain memory locations until the lock is released again. In such a setting, whenever a thread holds on to a permission for a heap location, it is guaranteed that no other thread can modify that location and, thus, enables thread-modular reasoning.

Note that there are several alternatives to permission-based verification that, for example, make use of dynamic frames [63] or employ a notion of ownership [82]; these approaches are not further discussed here as they are not the focus of this thesis.

Unbounded Data Structures. Programs commonly operate on data structure comprising a statically unbounded number of heap locations. There are two main mechanisms that are used to specify permissions for all heap locations of such data structures:

1. In permission logics, *quantified permissions* – often referred to as *iterated separating conjunctions* [92] – allow programmers to write pointwise permission specifications for unbounded data structures by quantifying over elements in a set. By virtue of their flat structure, quantified permissions do not restrict the order in which the heap locations are accessed; this makes them convenient for specifying permissions for random access data, such as arrays (on the level of individual array elements) but also for general graphs that are typically not traversed in a predefined order.

2. Moreover, *recursive predicates* [92] can be used to inductively define linked data structures. In contrast to quantified permissions that do not impose any structure, recursive predicates intrinsically define a tree-shaped backbone for the underlying data structure. A predicate definition consists of a name, a list of formal parameters and a body defining the assertion represented by a predicate instance. Like permissions for individual heap locations, predicate instances may be held by the execution of a method or loop iteration. Most automated permission-based verifiers [15, 59, 83, 89, 105, 106] distinguish between a predicate instance and its body and require the user to annotate the program with ghost statements to exchange a predicate instance with its body and vice versa.

Permission Inference. Permission logics enable sound and modular verification of sequential and concurrent heap-manipulating programs. However, they impose a substantial annotation burden on the programmers. In addition to the correctness properties they intend to verify, programmers need to provide extensive permission specifications, in particular, method preconditions and postconditions as well as loop invariants. It is, thus, highly desirable to infer permission specifications automatically,

such that programmers can complement them with specifications of the intended functional behaviour of the code at hand.

Moreover – apart from program verification – information about the memory accessed by parts of a program also finds use in other important applications that could benefit from an automatic permission inference; noteworthy examples are static race detection [103], code optimisations [41, 62, 73], and program parallelisation [14, 48].

1.1 State of the Art

There is a variety of tools and techniques that support specification inference to some extent. For instance, Clousot [40] is a tool developed by Microsoft that statically checks code contracts; it is based on an abstract interpretation [28] engine that automatically infers loop invariants (such as numerical constraints) to discharge proof obligations letting the user focus on method preconditions and postconditions. The SeaHorn verification framework [54] infers numerical invariants using the abstract interpretation based analyser IKOS [18]. Further examples of tools that incorporate some form of inference are Houdini [44], an annotation assistant for Java and Frama-C [80], a static analyser for C programs. However, none of the tools mentioned above infer permission specifications.

In the work at hand, we focus on inferring permission specifications required for deductive program verification. To the best of our knowledge there is no inference capable of inferring all permission-related annotations – which includes method preconditions and postconditions, loop invariants, predicate definitions and potentially also ghost code – required by a program verifier; both, for programs that manipulate arrays and recursive data structures. In particular, many approaches targeting linked data structures are tailored to specific data structures, such as linked lists [11, 81] or are limited to a set of predefined predicates, that is, do not infer predicate definitions [22, 84, 102].

Abstract Interpretation. An established mathematical framework for specification inference is *abstract interpretation* [28]. Developing abstract interpretation techniques for heap manipulating program is notoriously hard as it requires designing an abstract domain striking the right balance between an abstraction that generalises well and retaining precision in order to obtain sufficiently strong specifications.

One of the few abstract interpretation-based techniques that explicitly infers permission specifications is Ferrara and Müller's [42] approach, which builds upon an alias analysis to abstract heap locations and then collect constraints for the permissions required for each location. The inferred specifications are obtained by

solving the resulting system of constraints using linear programming [31]. Their inference mainly targets access permissions for individual heap locations and supports unbounded data structures only if they are manually summarised via abstract predicates; specifying unbounded data structures remains the programmer's burden.

In general, many approaches concerned with analysing the memory footprint of heap-manipulating programs are phrased as shape analyses. An example is Distefano et al.'s [36] abstract interpretation-based shape analysis for list-manipulating programs.

Graph and Grammar-Based. Many shape analyses, however, employ an internal representation of the heap that cannot easily be translated into an assertion. For example Manevich et al.'s [75] graph-based analysis and and Lee et al.'s [69] grammar-based technique can be used to check properties but do not produce annotations suitable for program verification and human inspection.

Bi-Abduction. A particularly prominent technique for shape analysis in the context of separation logic is *bi-abduction*, proposed by Calcagno et al. [22]. Within this line of work, Brotherston et al.'s [20] cyclic abduction based inference and Le et al.'s [66] second-order bi-abduction are both capable of inferring predicate definitions for recursively defined data structures; however, the former cannot handle method calls as this would require to infer preconditions and postconditions simultaneously, while the latter does not support loops.

Furthermore, the problem of automating bi-abduction entailment checking for array-based separation logic has been studied by Brotherston et al. [21]; this work has not yet been extended to handle loops or recursion.

Learning-Based. Recently, many learning-based techniques [26, 95, 96] for inferring program specifications emerged. These works typically focus on inferring a *single* specification (for example, the loop invariant of a particular loop). Notably, Garg et al. [46, 47] describe a learning-based approach that produces universally quantified value constraints for arrays and lists; this approach, however, is not concerned with inferring permission specifications. From this line of work, the technique closest to our work is undoubtedly the one proposed by Neider et al. [84] that infers separation logic specifications for recursively defined data structures; their approach requires the programmer to provide method preconditions and the definition of recursive predicates and method postconditions as well as loop invariants.

It is worth pointing out that learning has also been applied towards program synthesis; for example, there is a rather well known line of work based on counterexample-

guided inductive synthesis (CEGIS) [99] that undoubtedly shares similarities with some of the approaches mentioned above.

1.2 Research Goals and Scope

This thesis aims at developing and implementing inference techniques for permission specifications that can be used directly in a standard verification workflow. To make this possible, our techniques must (i) provide all specifications required by a verifier, (ii) infer specifications that are concise and readable, such that programmers can extend them to also express functional properties, and (iii) handle a wide range of data structures.

As stated above, there are two main mechanisms to specify permissions for data structures comprising a statically unbounded number of heap locations: on one hand, there are quantified permissions that can be used, for example, to specify permissions for integer-indexed data structures, such as arrays and matrices, and on the other hand there are recursive predicates typically used to define permissions for linked lists and tree-like data structures. Throughout this thesis, we present two separate inference techniques for both of these specification mechanisms; as discussed in more detail in Chapter 5, in most cases the permission specifications inferred by either technique are independent of each other, which allows us to run the inference techniques one after another to infer specifications for programs containing both integer-indexed and recursively defined data structures. The following two sections briefly outline the approach and contributions for both of our inference techniques.

1.3 Array Programs

In chapter Chapter 3, which is based on the CAV 2018 paper titled "Permission Inference for Array Programs" [38], we present a permission inference for array manipulating programs.

Approach. Our analysis reduces the problem of reasoning about permissions for array elements to reasoning about numerical values for permission fractions. To achieve this, we represent permission fractions for *all* array elements using a *single* numerical term $t(\mathsf{q}_a, \mathsf{q}_i)$ parameterised by two designated variables q_a and q_i ranging over all arrays and indices, respectively. For instance, the conditional term $(\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\,?\,1 : 0$ represents a full permission – denoted by 1 – for the array element $a[i]$ and no permission – denoted by 0 – for all other array elements.

Our analysis employs a precise backwards analysis for *loop-free* code: a variation on the standard notion of weakest preconditions. We apply this analysis to loop bodies to obtain a permission precondition for a single loop iteration. Per array element, the whole loop requires the *maximum* fraction over all loop iterations, adjusted by permissions lost and gained during the loop execution. Rather than computing permissions via a fixpoint iteration – for which a precise operation is difficult to design – we express them as a maximum over the variables modified by the loop execution. We then use numerical invariants obtained using an off-the-shelf numerical analysis in combination with a novel *maximum elimination algorithm* to infer a specification for the entire loop. For instance, computing the maximum of $(\mathsf{q}_a = \mathsf{a} \land \mathsf{q}_i = \mathsf{i})\ ?\ 1 : 0$ over all values for i satisfying $0 \leq \mathsf{i} \land \mathsf{i} < \mathsf{len}(\mathsf{a})$ yields $(\mathsf{q}_a = \mathsf{a} \land 0 \leq \mathsf{q}_i \land \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ 1 : 0$, which represents a full permission for all elements of the array a within bounds.

Contributions. The main technical contributions of Chapter 3 are:

1. We present a novel permission inference that expresses permission specifications by numerical expressions parameterised by program variables and summarises loops using maximum expressions ranging over an unbounded set of values.
2. Moreover, we introduce a maximum elimination algorithm that allows us to solve such maximum expressions.
3. To account for permissions lost and gained throughout the loop execution, we introduce and employ the notion of *progressive numerical invariants* that allows us to distinguish between loop iterations that have already been executed and ones that are still to be executed.
4. Our inference is proven sound, implemented, and evaluated on benchmark examples from existing papers and competitions, demonstrating that we obtain sound, precise, and concise specifications, even for challenging array patterns and parallel loops.

1.4 Black-Box Inference

In Chapter 4, we present a learning-based permission inference targeting programs manipulating recursively defined data structures.

Approach. Our inference is based on the ICE framework [47], where a teacher and a learner work in tandem to iteratively infer specifications. In every iteration, the teacher generates constraints on the desired specifications and the learner solves these

constraints to generate the next candidate specifications to be checked by the teacher. The teacher employs a program verifier as an oracle to check the correctness of the current candidate specifications. The learner translates the constraints imposed by the teacher into an SMT encoding in order to synthesise appropriate specifications. This approach is black-box since (i) the teacher only needs to understand how the program interacts with specifications but is otherwise independent of any program semantics, and (ii) the learner is entirely agnostic of the program and synthesises new hypothesis based only on the constraints produced by the teacher.

Contributions. The main technical contributions of Chapter 4 are:

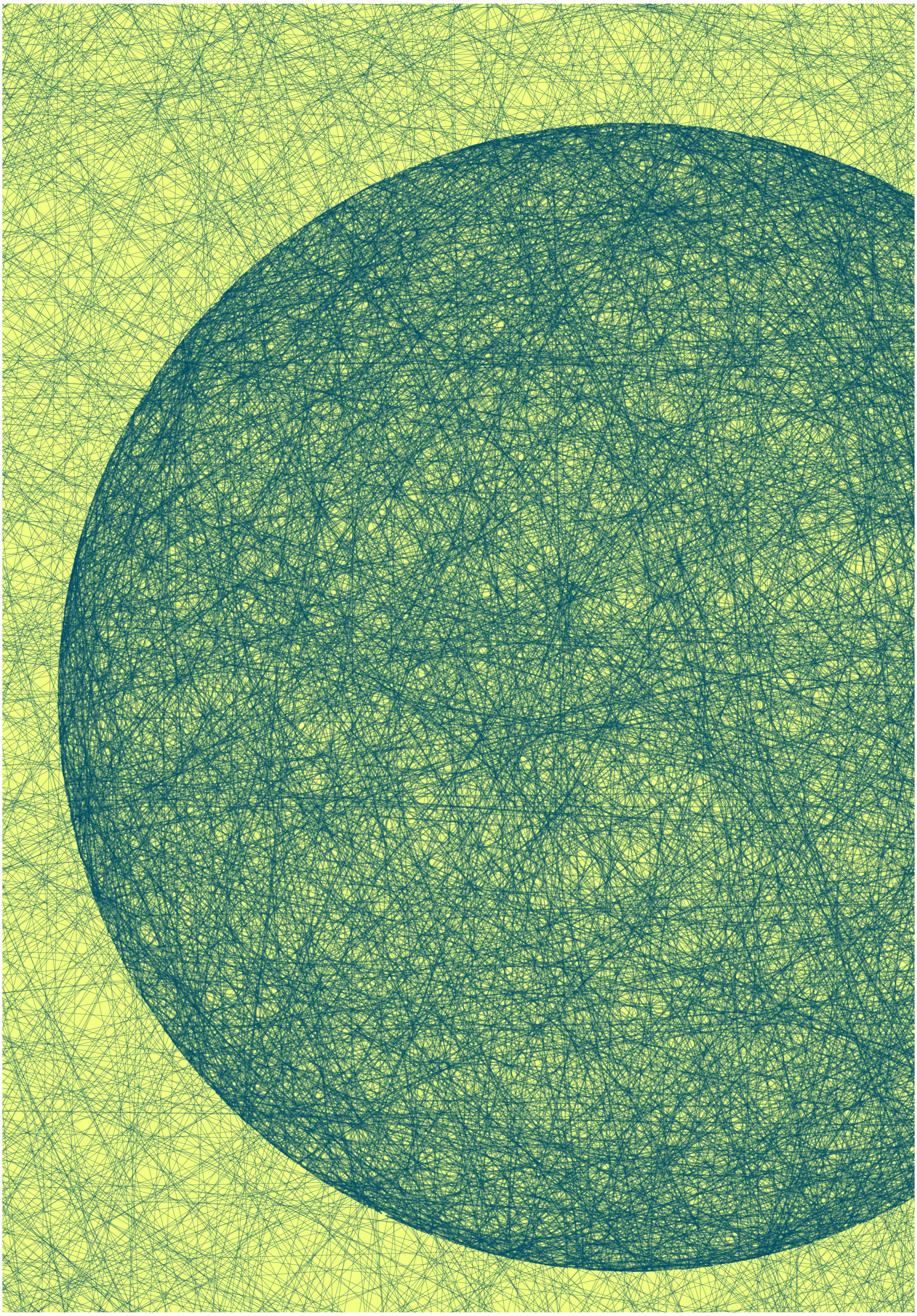
1. We present a novel black-box permission inference based on the ICE framework. Our samples summarise the permission constraints of entire traces rather than individual states, which allows us to generate specifications for the entire program, by simultaneously inferring preconditions, postconditions, loop invariants, and the definitions of recursive predicates.

2. Our inference targets automated separation logic verifiers, which typically treat predicates isorecursively, that is distinguish between a predicate instance and its body. In particular, we automatically infer ghost operations required by such verifiers to unfold and fold predicates.

3. We have implemented our inference for the Viper verification infrastructure [83] and evaluated it on a benchmark suite consisting of challenging examples. Our evaluation shows that our technique can be applied to a wide range of recursively defined data structure and code patterns to infer all permission-related annotations at once, even in the presence of partial functional or permission specifications.

1.5 Thesis Outline

The remainder of this thesis is structured as follows. In Chapter 2, we introduce some basic mathematical notation, discuss the concept of permission specifications, and define the syntax and semantics of a simple programming language used for our formalisations. In Chapter 3, we present our permission inference for array programs. In Chapter 4, we present our black-box inference technique targeting recursively defined data structures. In Chapter 5, we conclude this thesis and discuss possible future directions.



2 Preliminaries

In this chapter, we establish the most important notations and concepts used throughout this work.

Chapter Outline. We start by defining some mathematical notation. After that, we introduce permission specifications. Finally, we define the syntax and semantics of a simple programming language used for our formalisations later.

2.1 Mathematical Notations

In order to avoid ambiguity, we briefly introduce mathematical notation used throughout this thesis that may be used differently in some of the existing literature.

Equalities. Throughout this thesis, we use the symbol $=$ to denote *semantic equality* while we use the symbol $\equiv$ to denote *syntactic equality*. Moreover, we use the symbols $:=$ and $:\equiv$ for defining equalities.

Number Sets. Throughout this thesis, we use the following symbols to refer to commonly used number sets: The set $\mathbb{N} := \{0, 1, 2, 3, \ldots\}$ denotes the set of natural numbers; we use $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$ to refer to all positive natural numbers. Furthermore, the set $\mathbb{Z} := \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ denotes the set of all integers. Finally, the set $\mathbb{Q} := \{p/q \mid p, q \in \mathbb{Z} \land q \neq 0\}$ denotes to all rational numbers.

Functions. Two functions f and g are *equal* if and only if their domains are the same and their outputs agree on all input. That is, given $f\colon X \to Y$ and $g\colon X \to Y$, we have

$$f = g \quad :\Leftrightarrow \quad \forall x\colon f(x) = g(x).$$

```
1  method foo(a: Cell, b: Cell) {
2    a.val := 33333331
3    bar(a, b)
4    b.val := 333333331 / a.val
5  }
6
7  method bar(a: Cell, b: Cell) {
8    // some implementation ...
9  }
```

Listing 2.1. A heap manipulating program. The execution of the method foo may crash with a division by zero if the method bar modifies the field a.val.

For all functions $f\colon X \to Y$ and all elements $x \in X$ and $y \in Y$, we write $f[x \mapsto y]$ to denote the function that is equal to f, with the exception that it maps x to y. More formally, we define

$$f[x \mapsto y] \coloneqq \lambda z.\ \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise.} \end{cases}$$

2.2 Permission Specifications

In this section, we will first briefly discuss some aspects of permission-based verification to set the context, and then introduce the permission specifications inferred by our techniques presented later in this thesis.

2.2.1 Permission-Based Verification

Floyd and Hoare laid the foundation of deductive verification by formalising the notion of preconditions and postconditions [45, 57]: A Hoare triple $\{\,P\,\}\ c\ \{\,Q\,\}$ states that any execution of the statement c starting in a state that satisfies the precondition P will not fail and – if it terminates – result in a state that satisfies the postcondition Q.

The Problem. As already hinted in the introduction, the standard notion of Hoare triples is not suitable for modular verification of heap manipulating programs. To illustrate this, let us consider the program shown in Listing 2.1[1]. The last statement

1: The programming language will be introduced in Section 2.3 below.

of the method foo succeeds only if a.val $\neq 0$; at the beginning of the method, this is the case since a.val gets initialised to 33333331. However, the call to the method bar in between can potentially set a.val to zero. In modular verification, we want to be assured that this cannot happen by only looking at bar's specification; in particular, without having to look at its implementation. Consequently, bar's postcondition has to assert that a.val $\neq 0$, even if the value of a.val never gets updated. Clearly, as there are unboundedly many heap locations, an approach where every method has to explicitly state which heap locations it does *not* modify is impractical and does not scale. This problem is known as the *frame problem* [76].

Permission Logics. Permission logics solve the frame problem by associating every heap location with a permission; in particular, they allow a part of the program to access the location's value only if it holds said permission. In addition, permission logics provide means to decompose the heap into disjoint sub-heaps; this allows programmers to locally reason on the part of the heap that is actually relevant for their implementation.

For example, in separation logic [92] – which is arguably the most prominent permission logic – a *points-to* assertion $l \mapsto v$ denotes permission to access the heap location l and also expresses that this location's value is v. Moreover, assertions can be combined with a *separating conjunction* $A_1 * A_2$ stating that the heap can be partitioned into two disjoint sub-heaps in which assertion A_1, respectively A_2, hold. As permissions to a heap location can be understood to be exclusive, the assertion $l_1 \mapsto v_1 * l_2 \mapsto_x$, for example, implies that the heap locations l_1 and l_2 must be different; otherwise, it would denote permission for the same location twice.

Permission Transfers. A permission can be seen as a token of ownership for its associated memory location. A full-fledged programming language contains many statements that affect the ownership of locations, expressed via permissions [72, 101]. Ownership of a location is first obtained upon allocation; creating a new object in memory can therefore be represented as obtaining a fresh object and then obtaining permission to its locations, that is, the fields of the object or individual array elements if the object is an array. This ownership can then be passed to other parts of the program by transferring the corresponding permission. For example, in a concurrent setting, a fork operation may transfer permissions to the newly created thread or acquiring and releasing a lock temporarily obtains permissions to access a set of memory locations guarded by the lock. As already indicated in Chapter 1, even in a sequential setting the concept of permissions is useful for method-modular reasoning:

A call to a method transfers permissions from the caller to the callee, and back when the method call terminates.

Inhales and Exhales. For the purpose of verifying programs – and also inferring permissions, for that matter – we can reduce all operations affecting the ownership of locations to two basic operations that directly manipulate the permissions currently held [71, 83]: An inhale A statement *adds* all permissions represented by the assertion A. Dually, an exhale A statement requires that these permissions are already held, and then *removes* them.

Framing. In permission logics, a Hoare triple $\{ P \}\ c\ \{ Q \}$ additionally requires that the precondition P denotes sufficient permissions to execute the statement c and that any execution of the statement c starting in a state that satisfies P retains at least the permissions denoted by the postcondition Q. The fact that permission logics enable local reasoning is nicely captured by the *frame rule*, which can be stated as

$$\frac{\{ P \}\ c\ \{ Q \}}{\{ P * F \}\ c\ \{ Q * F \}},$$

where the statement c does not modify any variable appearing in F. Intuitively, this rule states that if the statement c can safely be executed starting in a state satisfying P, then it can also safely be executed from a state satisfying $P * F$. Moreover, the execution of c will will not affect the part of the heap nor affect any permissions captured by F; that is, the assertion F gets *framed* across the execution of c.

As an example, let us revisit the program shown in Listing 2.1 and assume that the precondition of bar does not mention any permission for a.val. In this case, the assertion $\mathsf{a.val} \mapsto V * V \neq 0$ can be framed across the method call to bar in the body of foo. Notably, this then allows us to prove that division by a.val after this method call does not fail.

2.2.2 Implicit Dynamic Frames

The specifications inferred by our inference techniques are based on *implicit dynamic frames* [98]. Parkinson and Summers have studied the relationship between implicit dynamic frames and separation logic, and have proven their weakest preconditions to be equivalent [88].

Accessibility Predicates. As stated above, a separation logic points-to assertion $l \mapsto v$ denotes permission to access the heap location l and simultaneously also makes a

statement about this location's value. Implicit dynamic frames separate these aspects: The *accessibility predicate* $\mathsf{acc}(l)$ denotes permission to access the heap location l but does not state anything about its value.

Separating Conjunctions. Implicit dynamic frames also provide a separating conjunction. For example, the assertion $\mathsf{acc}(x.f) * \mathsf{acc}(y.f)$ denotes permission to both $x.f$ and $y.f$, and excludes that x and y alias. Note that the points-to assertion $l \mapsto v$ can be expressed as $\mathsf{acc}(l) * l = v$.

Self-Framing Assertions. An implicit dynamic frames assertion is considered well-defined only if it is *self-framing*, that is, denotes at least the permissions for the heap locations that it reads. Note that, due to the decoupling of the permissions from the value constraints, this is not always guaranteed: for example, the assertion $\mathsf{acc}(\mathsf{a.val}) * \mathsf{a.val} \geq 0$ is self-framing, while $\mathsf{a.val} \geq 0$ and $\mathsf{acc}(\mathsf{a.val}) * \mathsf{b.val} \geq 0$ are not.

Fractional Permissions. To support shared read access to heap locations, our permission specifications use *fractional permissions* [16]. In this setting, a permission can be split up into fractions. Holding the full permission for a heap location allows modifying it, whereas holding a positive amount smaller than one allows reading (but not modifying) it. Crucially, since there cannot be more than one permission in total, holding on to *some* permission for a heap location guarantees that no other part of the program can alter said location.

We use $\mathsf{acc}(l, q)$, where $q \in \mathbb{Q}$ and $0 \leq q \leq 1$, to denote p permission for the heap location l; by convention, the accessibility predicate $\mathsf{acc}(l)$ is equivalent to $\mathsf{acc}(l, 1)$. Note that the assertion $\mathsf{acc}(l, q_1) * \mathsf{acc}(l, q_2)$ denotes $q_1 + q_2$ permission for l and is equivalent to false if $q_1 + q_2 > 1$.

2.2.3 Quantified Permissions

As indicated in Chapter 1, there are two mechanisms to specify permissions for unbounded data structures; one of them is *quantified permissions* [92]. Quantified assertions are of the form $\forall x \in S\colon A(x)$ and, intuitively, can be thought of as the (possibly infinite) separating conjunction $\mathbin{\text{✳}}_{x \in S} A(x)$.

Example 2.2.1. » The quantified assertion $\forall \mathsf{i} \in \mathbb{Z}\colon 0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{n} \Rightarrow \mathsf{acc}(\mathsf{a}[\mathsf{i}])$ denotes permissions for all array elements $\mathsf{a}[\mathsf{i}]$, where i is between 0 and $\mathsf{n}-1$. «

2.2.4 Recursive Predicates

Another means to specify permissions for unbounded data structures are *recursive predicates* [87]. A *predicate definition* $p(\vec{x}) \triangleq A$ consists of a name p, some parameters $\vec{x}$ and a predicate body A. A *predicate instance* $p(\vec{e})$ is defined by a predicate name p along with some arguments $\vec{e}$. A predicate $p(\vec{x})$ is called *recursive* if its body contains at least one *successor* instance $p(s(\vec{x}))$, defined by some function s of the predicate's parameters $\vec{x}$.

Example 2.2.2. » The recursive predicates

$$\mathsf{list}(\mathsf{x}) \triangleq \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.next}) * \mathsf{list}(\mathsf{x.next})$$
$$\mathsf{tree}(\mathsf{x}) \triangleq \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.left}) * \mathsf{acc}(\mathsf{x.right}) * \mathsf{tree}(\mathsf{x.left}) * \mathsf{tree}(\mathsf{x.right})$$

describe a null-terminated linked list and a binary tree, respectively. For the list predicate, the successor is given by $s(\mathsf{x}) \equiv \mathsf{x.next}$, whereas the tree predicate has two successors $s_1(\mathsf{x}) \equiv \mathsf{x.left}$ and $s_2(\mathsf{x}) \equiv \mathsf{x.right}$. «

Intuitively, a predicate instance $p(\vec{e})$ is equivalent to its body $A[\vec{e}\backslash\vec{x}]$ and can be thought of as the complete unrolling of its definition.[2]

Segment Predicates. For iterative implementations, loop invariants typically require one to express the part of the data structure that has already been traversed, in order not to leak the corresponding permissions. This is commonly done via segment predicates that describe partial data structures.

Example 2.2.3. » Let us consider the loop $\mathsf{while}\ (\mathsf{node} \neq \mathsf{null})\ \{\mathsf{node} := \mathsf{node.next}\}$. The predicate

$$\mathsf{lseg}(\mathsf{x}, \mathsf{y}) \triangleq \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.next}) * \mathsf{lseg}(\mathsf{x.next}, \mathsf{y})$$

describes a segment of a linked list starting at x and truncated at y. The predicate instance $\mathsf{lseg}(\mathsf{head}, \mathsf{node})$, where head holds the value of node at the beginning of the loop, can be used in the loop invariant of the loop above to refer to the part of the list that has already been traversed. «

2: This interpretation treats predicates *equirecursively*. Automated verifiers often treat them *isorecursively*, that is, distinguish between predicate instances and their bodies. For more details, we refer to Section 4.5.

2.3 Programming Language

The inference techniques presented in this thesis are defined for a simple imperative programming language with local variables, heap-allocated arrays or objects with fields, respectively, loops, method calls, and the usual boolean and arithmetic expressions.

2.3.1 Program States

Our programming language is mostly standard, but equipped with additional state to track how much permission is held to each heap location; a program state therefore consists of three parts capturing the local variables, the shared heap, and the permissions, respectively. As we will see below, our states comprise the entire heap; this makes our program semantics a *total-heap semantics* rather than a *partial-heap semantics* typically used by separation logic [88].

Stores and Heaps. Let X denote the set containing all program variables. Moreover, we use L to denote the set of all heap locations. To accommodate the permission inference for array programs introduced in Chapter 3, the set L contains all array elements $\langle a, i\rangle$, where a is an array and i an index. Additionally, considering our black-box inference for linked data structures presented in Chapter 4, it also includes all locations $l \in L$ consisting of pairs $\langle o, f\rangle$ of objects o and fields f.

Definition 2.3.1. » A *store* $s\colon X \to V$ is a function mapping program variables $x \in X$ to their value $s(x)$. «

Definition 2.3.2. » A *heap* $h\colon L \to V$ is a function mapping heap locations $l \in L$ to their value $h(l)$. «

Permission Maps. A permission map associates each heap location with a permission amount. For the sake of generality, we define permissions over a set of *resources* R, where $L \subseteq R$; this allows us to explicitly include predicate instances in our permission map.

Definition 2.3.3. » A *permission map* $\pi\colon R \to \mathbb{Q}$ is a function mapping resources $r \in R$ to their permission amount $\pi(r)$. «

Note that the permission fractions in our permission maps are *not* restricted to be values between zero and one; this will simplify some of our formalisations later and also permits states containing the same predicate instance more than once. For the

sake of easier notation, we define addition and subtraction on permission maps as pointwise operations. That is, for all permission maps π_1 and π_2, we define

$$\pi_1 + \pi_2 \coloneqq \lambda r.\ \pi_1(r) + \pi_2(r)$$
$$\pi_1 - \pi_2 \coloneqq \lambda r.\ \pi_1(r) - \pi_2(r).$$

Moreover, we often need to denote that a permission map represents at least as many permissions as another permission map. To this end, for all permission maps π_1 and π_2, we define

$$\pi_1 \sqsubseteq \pi_2 \quad :\Leftrightarrow \quad \forall r\colon \pi_1(r) \leq \pi_2(r).$$

We observe that this partial order on permission maps induces a least upper bound $\pi_1 \sqcup \pi_2$ and a greatest lower bound $\pi_1 \sqcap \pi_2$ that can be characterised as follows:

$$\pi_1 \sqcup \pi_2 = \lambda r.\ \max\{\pi_1(r), \pi_2(r)\}$$
$$\pi_1 \sqcap \pi_2 = \lambda r.\ \min\{\pi_1(r), \pi_2(r)\}$$

Throughout this work, we often need to refer to permission maps capturing no permission or the permission of a single heap location; to facilitate this, let us define

$$\pi_{\mathsf{zero}} \coloneqq \lambda r.\ 0$$
$$\pi_{r,q} \coloneqq \pi_{\mathsf{zero}}[r \mapsto q],$$

where $r \in R$ and $q \in \mathbb{Q}$.

States. Having introduced stores, heaps, and permission maps, we now formally define program states.

Definition 2.3.4. » A *state* $\sigma = \langle s, h, \pi \rangle$ is a tuple consisting of a store s, a heap h, and a permission map π. Moreover, let Σ to denote the set containing all states. «

When comparing two program states, we usually care only about the parts of the heaps for which we have permissions (as we are not allowed to access the remainder of the heap). This motivates the following definition.

Definition 2.3.5. » Two heaps h_1 and h_2 *agree* on a permission map π, denoted $h_1 \overset{\pi}{=} h_2$, if and only if $h_1(l) = h_2(l)$, for all heap locations $l \in L$ with $\pi(l) > 0$. «

2.3.2 Program Syntax

The syntax of our simple programming language is given by the following grammar.

$$
\begin{aligned}
c &::= \mathsf{skip} \mid c\,;c \mid \mathsf{if}\ (b)\ \{c\}\ \mathsf{else}\ \{c\} \mid \mathsf{while}\ (b)\ \{c\} \\
&\quad \mid x := e \mid x := l \mid l := x \\
&\quad \mid \mathsf{inhale}\ A \mid \mathsf{exhale}\ A \\
b &::= \mathsf{true} \mid \mathsf{false} \mid e \sim e \mid \neg b \mid b \wedge b \mid b \vee b \\
e &::= n \mid x \mid e + e \mid e - e \mid e \cdot e \mid \mathsf{mod}(e, n) \mid \mathsf{len}(a) \\
l &::= a[e] \mid x.f \\
A &::= b \mid \mathsf{acc}(l, q) \mid b \Rightarrow A \mid A * A
\end{aligned}
$$

In the grammar above, $x \in X$ ranges over variables, $\sim \in \{=, \neq, \leq, <, \geq, >\}$ over comparison operators, $n \in \mathbb{Z}$ over integers, a over array-typed variables, f over fields, and $q \in \mathbb{Q}$ over permission amounts. Throughout this thesis, we assume that variables are either integer- or reference-typed and consider only programs that are well-typed.

Note that – for the sake of simplicity – we assume that all operations affecting permissions are modelled via inhale and exhale statements. This has several advantages; in particular, this lets us omit concurrency and method calls in our program syntax and semantics. More details on methods calls are briefly discussed for both of our inference techniques in the respective chapter. Moreover, also note that, in Chapter 4, we will extend the grammar for assertions to also include predicate instances.

We observe that our program syntax allows accessing the heap only in designated ways and via location expressions l; this ensures that all other conditions b and expressions e are heap-independent. This is not a restriction, as any program can easily be rewritten into such a form. For example, the assignment $\mathsf{a.val} := \mathsf{b.val} + 1$ can be rewritten as $\mathsf{t} := \mathsf{b.val}\ ;\ \mathsf{a.val} := \mathsf{t} + 1$ and an $\mathsf{inhale\ acc}(\mathsf{a.next.val})$ statement can be rewritten as $\mathsf{t} := \mathsf{a.next}\ ;\ \mathsf{inhale\ acc}(\mathsf{t.val})$, where, in each case, t is a fresh variable.

2.3.3 Expression Semantics

The semantics of expressions is standard and defined recursively over the structure of expressions. For all expressions e and states $\sigma = \langle s, h, \pi \rangle$, we use $[\![e]\!](\sigma)$ to denote the value obtained from evaluating the expression e in the state σ. For example, we define $[\![x]\!](\sigma) := s(x)$ and $[\![e_1 + e_2]\!](\sigma) := [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$. We overload this notation to also evaluate the value of conditions $[\![b]\!](\sigma)$. For instance, we have $[\![\mathsf{true}]\!](\sigma) :\Leftrightarrow \mathsf{true}$ as well as $[\![e_1 = e_2]\!](\sigma) :\Leftrightarrow [\![e_1]\!](\sigma) = [\![e_2]\!](\sigma)$.

Heap Accesses. In order to formalise heap accesses, we need to be able to evaluate heap locations represented by location expressions l. To this end, for all location expressions l and all states σ, we define

$$[\![l]\!]_L(\sigma) \coloneqq \begin{cases} \langle s(a), [\![e]\!](\sigma)\rangle & \text{if } l \equiv a[e] \\ \langle s(x), f\rangle & \text{if } l \equiv x.f. \end{cases}$$

Note that, when we read from the heap, we want to lookup the *value* of a location expression l; this value is given by $[\![l]\!](\sigma) \coloneqq h([\![l]\!]_L(\sigma))$. When we write to the heap, we want to update the value of h associated with the corresponding *location* $[\![l]\!]_L(\sigma)$.

2.3.4 Statement Semantics

A configuration captures a single point during a program execution and, therefore, consists of the position in the program and the current program state. We represent the position in the program as a statement that, intuitively, corresponds to the part of the program that is still to be executed.

Definition 2.3.6. » A regular *configuration* $\gamma = \langle c, \sigma\rangle$ is a tuple consisting of a statement c and a state σ. In addition, let $\gamma_\lightning$ denote the unique *failing configuration*. Moreover, we use *infeasible configuration* $\gamma_\times$ to summarise configurations that do not exist in a real execution; for example, a configuration where the state provides more than a full permission for a heap location. Finally, let Γ denote the set containing all configurations, including $\gamma_\lightning$ and $\gamma_\times$. «

Below, we formalise a small steps semantics for our programming language by introducing rules that collectively define a transition relation $\rightsquigarrow \subseteq \Gamma \times \Gamma$ capturing a single step of an execution. For all $k \in \mathbb{N}$, we write $\gamma \rightsquigarrow^k \gamma'$ to denote that there is an execution from γ to γ' with k steps; note that, if $k = 0$, we must necessarily have $\gamma = \gamma'$. If the number of steps is irrelevant, we write $\gamma \rightsquigarrow^* \gamma'$ to denote that there is a $k \in \mathbb{N}$, for which $\gamma \rightsquigarrow^k \gamma'$. Furthermore, the following definition formally introduces traces that capture *all* configurations encountered along an execution.

Definition 2.3.7. » A *trace* of length $k \in \mathbb{N}^+$ is a non-empty sequence of k configurations $t = \langle \gamma_1, \ldots, \gamma_k\rangle$ such that $\gamma_i \rightsquigarrow \gamma_{i+1}$, for all $i \in \{1, \ldots, k-1\}$. Moreover, for all $i \in \{1, \ldots, k\}$, we use $t[i] \coloneqq \gamma_i$ to denote the i-th configuration of the trace t. «

Note that the length of a trace is determined by the number of encountered configurations rather than by the number of steps.

Sequential Compositions. The rules for sequential compositions $c_1 \,;\, c_2$ execute one step of the first statement c_1, or drop the first statement if it is a skip statement and, therefore, already fully executed.

$$\frac{\langle c_1, \sigma\rangle \rightsquigarrow \langle c_1', \sigma'\rangle}{\langle c_1 \,;\, c_2, \sigma\rangle \rightsquigarrow \langle c_1' \,;\, c_2, \sigma'\rangle} \qquad \frac{}{\langle \mathsf{skip} \,;\, c_2, \sigma\rangle \rightsquigarrow \langle c_2, \sigma\rangle}$$

Any failing configuration or ghost configuration encountered during the execution of the first statement of a sequential composition is simply propagated.

$$\frac{\langle c_1, \sigma\rangle \rightsquigarrow \gamma_{\lightning}}{\langle c_1 \,;\, c_2, \sigma\rangle \rightsquigarrow \gamma_{\lightning}} \qquad \frac{\langle c_1, \sigma \rightsquigarrow \gamma_{\times}\rangle}{\langle c_1 \,;\, c_2, \sigma\rangle \rightsquigarrow \gamma_{\times}}$$

Conditionals. The execution of a conditional statement $\mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}$ selects either of the branches c_1 or c_2, depending on the value of the condition b.

$$\frac{[\![b]\!](\sigma)}{\langle \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, \sigma\rangle \rightsquigarrow \langle c_1, \sigma\rangle} \qquad \frac{\neg[\![b]\!](\sigma)}{\langle \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, \sigma\rangle \rightsquigarrow \langle c_2, \sigma\rangle}$$

Note that the evaluation of the condition b cannot fail as it is heap-independent.

Loops. The execution of a loop $\mathsf{while}\ (b)\ \{c\}$ iteratively unrolls loop iterations as long as the loop condition b evaluates to true.

$$\frac{[\![b]\!](\sigma)}{\langle \mathsf{while}\ (b)\ \{c\}, \sigma\rangle \rightsquigarrow \langle c \,;\, \mathsf{while}\ (b)\ \{c\}, \sigma\rangle} \qquad \frac{\neg[\![b]\!](\sigma)}{\langle \mathsf{while}\ (b)\ \{c\}, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle}$$

Again, the evaluation of the condition b cannot fail as it is heap-independent.

Assignments. The execution of a heap-independent variable assignment $x := e$ simply updates the store accordingly.

$$\frac{\sigma = \langle s, h, \pi\rangle \quad v = [\![e]\!](\sigma)}{\langle x := e, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \langle s[x \mapsto v], h, \pi\rangle\rangle}$$

An assignment $x := l$ reading from a heap location succeeds only if some permissions for the heap location in question are currently held and fails, otherwise.

$$\frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![l]\!]_L(\sigma) \quad \pi(u) > 0}{\langle x := l, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \langle s[x \mapsto h(u)], h, \pi\rangle\rangle} \qquad \frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![l]\!]_L(\sigma) \quad \pi(u) \leq 0}{\langle x := l, \sigma\rangle \rightsquigarrow \gamma_{\lightning}}$$

Similarly, an assignment $l := x$ updating a heap location succeeds only if a full permission is held for the heap location in question and fails, otherwise.

$$\frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![l]\!]_L(\sigma) \quad \pi(u) \geq 1}{\langle l := x, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \langle s, h[u \mapsto s(x)], \pi\rangle\rangle} \qquad \frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![l]\!]_L(\sigma) \quad \pi(u) < 1}{\langle l := x, \sigma\rangle \rightsquigarrow \gamma_{\lightning}}$$

Note that the rules leading to the failing configuration – capturing cases where there are insufficient permissions – subsumes runtime failures such as null pointer dereferences and out-of-bounds errors, as it is impossible to possess permission for an invalid heap location (assuming that allocation, de-allocation, and permission transfers are modelled soundly).

Assumes and Asserts. The execution of an inhale b statement corresponds to assuming the condition b. The assumption of a condition that holds does not change the state, whereas the assumption of a condition that does not hold is modelled as a transition to the infeasible configuration $\gamma_\times$, indicating that the execution leading up to this point is not possible.

$$\frac{[\![b]\!](\sigma)}{\langle \mathsf{inhale}\ b, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle} \qquad \frac{\neg[\![b]\!](\sigma)}{\langle \mathsf{inhale}\ b, \sigma\rangle \rightsquigarrow \gamma_\times}$$

Conversely, the execution of an exhale b statement corresponds to asserting the condition b; it does not change the state and succeeds if the condition is true. One might expect a failing assertion to lead to the failing configuration. For the sake of inferring permissions, however, we do not consider non-permission failures and model a failing assertion as a transition to the infeasible configuration $\gamma_\times$. Consequently, the rules for assertions are the same as the ones for assumptions.

$$\frac{[\![b]\!](\sigma)}{\langle \mathsf{exhale}\ b, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle} \qquad \frac{\neg[\![b]\!](\sigma)}{\langle \mathsf{exhale}\ b, \sigma\rangle \rightsquigarrow \gamma_\times}$$

Inhales and Exhales. An inhale $\mathsf{acc}(r, q)$ statement adds q permissions for the resource r; unless this would lead to a state with more than a full permission for r.

$$\frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![r]\!]_R(\sigma) \quad \pi(u) + q \leq 1}{\langle \mathsf{inhale}\ \mathsf{acc}(r, q), \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \langle s, h, \pi + \pi_{u,q}\rangle\rangle} \qquad \frac{\sigma = \langle s, h, \pi\rangle \quad \pi([\![r]\!]_R(\sigma)) + q > 1}{\langle \mathsf{inhale}\ \mathsf{acc}(r, q), \sigma\rangle \rightsquigarrow \gamma_\times}$$

In the rules above, we use $[\![\cdot]\!]_R(\sigma)$ to denote the extension of $[\![\cdot]\!]_L(\sigma)$ to resources. Dually, an exhale $\mathsf{acc}(r, q)$ statement removes q permissions for the resource r; however, the statement fails if less than q permissions for r were held in the first place.

$$\frac{\sigma = \langle s, h, \pi\rangle \quad u = [\![r]\!]_R(\sigma) \quad \pi(u) - q \geq 0}{\langle \mathsf{exhale}\ \mathsf{acc}(r, q), \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \langle s, h, \pi - \pi_{u,q}\rangle\rangle} \qquad \frac{\sigma = \langle s, h, \pi\rangle \quad \pi([\![r]\!]_R(\sigma)) - q < 0}{\langle \mathsf{exhale}\ \mathsf{acc}(r, q), \sigma\rangle \rightsquigarrow \gamma_{\lightning}}$$

Further, we observe that the inhale $A_1 * A_2$ is equivalent to inhale A_1 ; inhale A_2 as well as that the inhale $b \Rightarrow A$ is equivalent to if (b) {inhale A}. Thus, all inhale statements not captured by the rules above can be handled by assuming them to be rewritten according to the aforementioned observation; the analogous is also true for exhale statements.

Interference Rule. Our rules for inhale and exhale introduced above are not yet sufficient to truthfully model the effect of other parts of the program (for example, concurrently running threads or method calls), as they might update parts of the heap for which we currently do not have any permissions. Such an interference could technically be modelled using the following rule.

$$\frac{\sigma = \langle s, h, \pi \rangle \quad \pi(u) \leq 0}{\langle c, \sigma \rangle \rightsquigarrow \langle c, \langle s, h[u \mapsto v], \pi \rangle \rangle}$$

Adding such a rule, however, unnecessarily complicates proofs as it introduces a step that is always possible. The effect of an update (by another thread, for example) on a heap location for which we currently do not have permissions cannot be observed anyway; not until we regain permissions to access said heap location, that is. Thus, we can capture such effects by complementing our rules for inhale statements above with the following rule.

$$\frac{\sigma = \langle s, h, \pi \rangle \quad u = [\![r]\!]_R(\sigma) \quad \pi(u) \leq 0 \quad \pi(u) + q \leq 1}{\langle \mathsf{inhale\ acc}(r, q), \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \langle s, h[u \mapsto v], \pi + \pi_{u,q} \rangle \rangle}$$

Intuitively, this rule lets us assume an arbitrary value for heap locations for which we gain permissions and no permissions were previously held.

2.3.5 Useful Lemmas

Next, we prove some basic but useful lemmas about our program semantics that will be of use for our soundness proof in Chapter 3. The first two lemmas let us split up the execution of a sequential composition $c_1 \,;\, c_2$ into the parts corresponding to c_1 and c_2.

Lemma 2.3.8. » *For all non-negative integers $k \in \mathbb{N}$, all statements c_1 and c_2, and all states σ and σ', we have $\langle c_1 \,;\, c_2, \sigma \rangle \rightsquigarrow^k \langle \mathsf{skip}, \sigma' \rangle$ if and only if there is a state σ'' such that $\langle c_1, \sigma \rangle \rightsquigarrow^{k_1} \langle \mathsf{skip}, \sigma'' \rangle$ and $\langle c_2, \sigma'' \rangle \rightsquigarrow^{k_2} \langle \mathsf{skip}, \sigma' \rangle$, for some $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$.* «

Note that – as the proof below suggests – the execution of the sequential composition takes $k_1 + k_2 + 1$ rather than $k_1 + k_2$ steps because dropping the skip statement that results from fully executing the first statement also counts as a step.

Proof (Sketch). Both directions can be proven separately.

The proof for the *if*-direction goes by induction on the length k_1 of the derivation sequence for the statement c_1. For the base case $k_1 = 0$, we must have $c_1 \equiv \mathsf{skip}$ and $\sigma = \sigma''$. Thus, we have $\langle c_1 \,;\, c_2, \sigma \rangle \rightsquigarrow \langle c_2, \sigma'' \rangle \rightsquigarrow^{k_2} \langle \mathsf{skip}, \sigma' \rangle$ and, consequently,

$\langle c_1 ; c_2, \sigma\rangle \rightsquigarrow^{k_1+k_2+1} \langle \mathsf{skip}, \sigma\rangle$, as desired. In the step case $k_1 > 0$, we can split off the first step of the derivation sequence for c_1: that is, $\langle c_1, \sigma\rangle \rightsquigarrow \langle c_1', \sigma'''\rangle \rightsquigarrow^{k_1-1} \langle \mathsf{skip}, \sigma''\rangle$. From here, the claim follows by applying the induction hypothesis to the shortened derivation sequence of length $k_1 - 1$ and then reattaching the first step.

Similarly, the *only if*-direction follows by induction on the length k of the derivation sequence for the sequential composition. □

Lemma 2.3.9. » *For all non-negative integers $k \in \mathbb{N}$, all statements c_1 and c_2, and all states σ with $\langle c_1 ; c_2, \sigma\rangle \rightsquigarrow^k \gamma_\lightning$, we have*

1. *$\langle c_1, \sigma\rangle \rightsquigarrow^{k-1} \gamma_\lightning$ or*
2. *there is a state σ' such that $\langle c_1, \sigma\rangle \rightsquigarrow^{k_1} \langle \mathsf{skip}, \sigma'\rangle$ and $\langle c_2, \sigma'\rangle \rightsquigarrow^{k_2} \gamma_\lightning$, for some $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$.* «

Proof. By straightforward induction on the length k of the derivation sequence. □

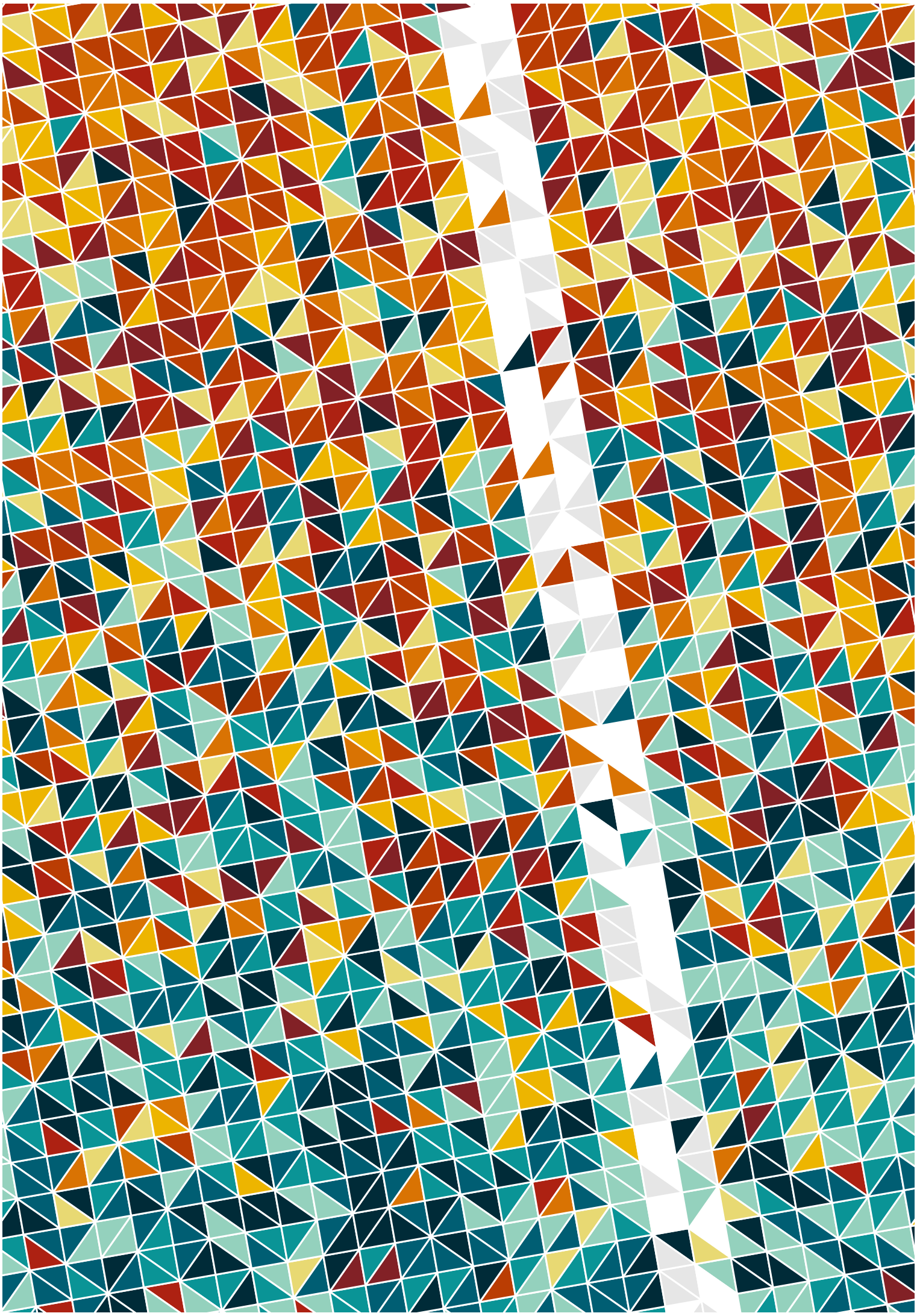
The following lemma lets us extract the last loop iteration from a derivation sequence representing multiple consecutive loop iterations.

Lemma 2.3.10. » *For all positive integers $k \in \mathbb{N}^+$, all loops $w \equiv \mathsf{while}\ (b)\ \{c\}$, and all states σ and σ' with $\langle w, \sigma\rangle \rightsquigarrow^k \langle w, \sigma'\rangle$, there is a state σ'', such that $\langle w, \sigma\rangle \rightsquigarrow^{k_1} \langle w, \sigma''\rangle$ and $\langle c, \sigma''\rangle \rightsquigarrow^{k_2} \langle \mathsf{skip}, \sigma'\rangle$, for some integers $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$.* «

Proof (Sketch). The proof goes by strong induction on the length of the derivation sequence k. Since $k > 0$, we can unroll the first loop iteration and apply Lemma 2.3.8 to obtain the two derivation sequences $\langle c, \sigma\rangle \rightsquigarrow^{n_1} \langle \mathsf{skip}, \sigma'''\rangle$ and $\langle w, \sigma'''\rangle \rightsquigarrow^{n_2} \langle w, \sigma'\rangle$, for some state σ''' and integers $n_1, n_2 \in \mathbb{N}$ with $n_1 + n_2 + 1 = k$.

If $n_2 = 0$, we are done, since the unrolled iteration is the last iteration of our entire derivation sequence, and $\langle w, \sigma\rangle \rightsquigarrow^0 \langle w, \sigma\rangle$ holds trivially.

Otherwise, we can apply our induction hypothesis to extract the last iteration from $\langle w, \sigma'''\rangle \rightsquigarrow^{n_2} \langle w, \sigma'\rangle$, which gives us another two sequences $\langle w, \sigma'''\rangle \rightsquigarrow^{n_3} \langle w, \sigma''\rangle$ and $\langle c, \sigma''\rangle \rightsquigarrow^{k_2} \langle \mathsf{skip}, \sigma'\rangle$, for some state σ'' and integers n_3, k_2 with $n_3 + k_2 + 1 = n_2$. It then remains to combine the derivation sequences $\langle c, \sigma\rangle \rightsquigarrow^{n_1} \langle \mathsf{skip}, \sigma'''\rangle$ and $\langle w, \sigma'''\rangle \rightsquigarrow^{n_3} \langle w, \sigma''\rangle$ to get $\langle w, \sigma\rangle \rightsquigarrow^{k_1} \langle w, \sigma''\rangle$, where $k := n_1 + n_3 + 1$. Some fiddling with the integers easily verifies that $k_1 + k_2 + 1 = k$. □



3 Array Programs

As mentioned in the introduction, information about the memory locations accessed by a program is, for instance, required for program parallelisation and program verification. Existing inference techniques for this information provide only partial solutions for the important class of array-manipulating programs. In this chapter – which is based on the CAV 2018 paper titled "Permission Inference for Array Programs" [38] – we present a static analysis that infers the memory footprint of an array program in terms of permission preconditions, postconditions and loop invariants, as used, for example, in separation logic. This information allows our analysis to handle concurrent programs and produces specifications that can be used by verification tools.

Approach. Our analysis employs a precise backwards analysis that constructs permission expressions capturing the memory footprint of loop-free code. We make use of numerical loop invariants to express the permissions required by a loop via pointwise maximum expressions over the individual loop iterations. We introduce the concept of progressive loop invariants that allow us to distinguish between past and future loop iterations and construct permission invariants that reflect permissions lost and gained by the execution of loops. The maximum expressions used to handle loops range over an unbounded set of values and are typically not supported by automated tools; to address this, we present a novel maximum elimination algorithm that – in the spirit of quantifier elimination – solves such pointwise maximum expression.

Our approach is proven sound and implemented; an evaluation on existing benchmarks for memory safety of array programs demonstrates accurate results, even for programs with complex access patterns and nested loops.

Contributions. The main technical contributions of this chapter are:

1. We propose a novel permission inference that generates permission preconditions, postconditions, and loop invariants expressing the memory footprint of array-manipulating programs. The specifications are expressed using permis-

sion expressions parameterised by program variables, allowing to summarise loops via pointwise maximum expressions.

2. We introduce the concept of progressive loop invariants that allows us to leverage off-the-shelf numerical analyses to obtain loop invariants that distinguish between past and future loop iterations.
3. We present a novel algorithm for eliminating pointwise maximum expression ranging over an unbounded set of values.
4. We provide formal proofs for the soundness of our permission analysis and the correctness of our maximum elimination algorithm.
5. We implemented our analysis based on the Viper verification infrastructure [83].
6. We evaluated our permission analysis based on a benchmark containing examples from existing papers and competitions, demonstrating that we obtain sound, precise, and concise specifications, even for challenging array access patterns and parallel loops.

Chapter Outline. The rest of this chapter is structured as follows. In Section 3.1, we provide an overview of our technique and introduce the permission expressions used to capture the inferred memory footprint. In Section 3.2, we start by devising rules to infer permission preconditions and postconditions for loop-free code. In Section 3.3, we extend our permission preconditions and postconditions to arbitrary loop programs; we do so by elaborating how to generalise results for a single loop iteration to the entire loop with the help of pointwise maximum expressions and numerical loop invariants. In Section 3.4, we introduce the concept of progressive loop invariants and show how it can be used to obtain permission invariants to summarise the permission changes caused by already executed loop invariants. In Section 3.5, we describe our novel maximum elimination algorithm that can be used to solve our pointwise maximum expressions. The results of our experimental evaluation is shown in Section 3.6. Finally, in Section 3.7, we conclude this chapter with a discussion that relates our approach to existing work and highlights its main strengths and limitations.

3.1 Overview

This section provides an overview of our inference techniques; in particular, it introduces permission expressions used to express the memory footprint of the array program at hand.

Section Outline. In Section 3.1.1, we outline our approach on a high level. In Section 3.1.2 we define the syntax and semantics of the permission expressions employed by our inference. In Section 3.1.3, we formally introduce pointwise maximum expressions used by our approach to summarise multiple loop iterations into a single permission expression.

3.1.1 Approach

Our analysis reduces the problem of reasoning about permissions for array elements to reasoning about numerical values for permission fractions. We do this by representing the permission fractions for *all* array elements using a *single* permission expression p parameterised by two designated variables q_a and q_i ranging over arrays and indices, respectively. This way, the permission fraction for any array element $a[i]$ is given by the term $p[a\backslash\mathsf{q}_a][i\backslash\mathsf{q}_i]$. For instance, the ternary expression $(\mathsf{q}_a = a \wedge \mathsf{q}_i = i)\ ?\ 1 : 0$ represents full permission for the array element $a[i]$ and no permission for all other array elements.

Our analysis employs a *precise* backwards analysis for *loop-free* code: a variation of the standard notion of weakest preconditions. We apply this analysis to loop bodies to obtain a permission precondition for a single loop iteration. Per array element, the *whole loop* requires the *maximum* permission fraction over all loop iterations, adjusted by permissions gained and lost during the execution of the loop. Rather than computing these permission fractions via a fixedpoint iteration – for which a precise widening operator is difficult to design – we express them as a pointwise maximum over the variables changed by the loop execution. We then use inferred numerical invariants on these variables and a novel *maximum elimination* algorithm to infer a specification for the entire loop.

Example 3.1.1. » Let us consider the copy_even_a method shown in Listing 3.1. Our analysis determines that the permission amount required by a single loop iteration is

$$p \equiv \mathsf{mod}(\mathsf{i}, 2) = 0\ ?\ ((\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ \mathsf{rd} : 0) : ((\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ 1 : 0).$$

Note that – as we will introduce formally shortly – a *ternary expression* of the form $b\ ?\ p_1 : p_2$ is equal to p_1 or p_2, depending on whether the condition b evaluates to true or false. Moreover, the symbol rd represents an arbitrary positive permission amount; a post-processing step can later replace rd by a concrete permission fraction. Using the integer invariant $0 \leq \mathsf{i}$ obtained from a suitable numerical analysis combined with the loop condition $\mathsf{i} < \mathsf{len}(\mathsf{a})$, we obtain the pointwise maximum expression

$$\max_{\mathsf{i}\,|\,0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a})} \{p\},$$

```
1  method copy_even_a(a: Int[]) {
2    var i: Int
3    var v: Int
4    i := 0
5    while (i < len(a)) {
6      if (mod(i, 2) = 0) {
7        v := a[i]
8      } else {
9        a[i] := v
10     }
11     i := i + 1
12   }
13 }
```

Listing 3.1. A method copying all array elements at even indices to its neighbouring element to the right.

representing sufficient permissions to execute the entire loop. Applying our maximum elimination to this pointwise maximum expression and carefully selected syntactic simplification rules obtains

$$(\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a})) \,?\, (\mathsf{mod}(\mathsf{q}_i, 2) = 0 \,?\, \mathsf{rd} : 1) : 0\,.$$

By ranging over all q_a and q_i, this permission expression can be read as read permission for all even indices and write permission for all odd indices within the array a's bounds. «

Permission postconditions are obtained in a similar fashion by combining the precondition with an additional analysis that infers the permissions lost and gained by the code at hand. Moreover – as detailed in Section 3.4 – loop invariants are obtained from loop preconditions by leveraging so-called progressive invariants to summarise the permission changes caused by loop iterations that have already been executed.

3.1.2 Permission Expressions

Next, we formally introduce the syntax and semantics of the permission expressions used by our analysis.

Definition 3.1.2. » The syntax of permission expressions is given by the grammar

$$p ::= q \mid p + p \mid p - p \mid \mathsf{min}(p, p) \mid \mathsf{max}(p, p) \mid b \,?\, p : p,$$

where $q \in \mathbb{Q}$ ranges over permission fractions and b over boolean expressions. «

Definition 3.1.3. » The semantics of a permission expression is a function mapping program states to permission fractions and is defined by

$$[\![p]\!](\sigma) \coloneqq \begin{cases} q & \text{if } p \equiv q \\ [\![p_1]\!](\sigma) + [\![p_2]\!](\sigma) & \text{if } p \equiv p_1 + p_2 \\ [\![p_1]\!](\sigma) - [\![p_2]\!](\sigma) & \text{if } p \equiv p_1 - p_2 \\ \min\{[\![p_1]\!](\sigma), [\![p_2]\!](\sigma)\} & \text{if } p \equiv \mathsf{min}(p_1, p_2) \\ \max\{[\![p_1]\!](\sigma), [\![p_2]\!](\sigma)\} & \text{if } p \equiv \mathsf{max}(p_1, p_2) \\ [\![p_1]\!](\sigma) & \text{if } p \equiv b \,?\, p_1 : p_2 \text{ and } [\![b]\!](\sigma) \\ [\![p_2]\!](\sigma) & \text{if } p \equiv b \,?\, p_1 : p_2 \text{ and } \neg[\![b]\!](\sigma). \end{cases}$$ «

Note that boolean expressions appearing in a permission expression may depend on our designated variables q_a and q_i ranging over arrays and indices, respectively. Throughout this chapter, it is often helpful to view a permission expression p as a mapping from array elements $a[e]$ to their corresponding permission fraction represented by $p[a\backslash\mathsf{q}_a][e\backslash\mathsf{q}_i]$. In other words, each permission expression represents a permission map that can be defined as follows.

Definition 3.1.4. » For all permission expressions p and all states σ, we define

$$[\![p]\!]_\Pi(\sigma) \coloneqq \lambda(a, i).\ [\![p[a\backslash\mathsf{q}_a][i\backslash\mathsf{q}_i]]\!](\sigma).$$ «

Note that, in the definition above, the subscript Π indicates that the permission expression p is evaluated as a permission map, rather than the single permission fraction $[\![p]\!](\sigma)$. For the following definition, we recall from Chapter 2 that the inequality $\sqsubseteq$ is defined as a pointwise comparison of permission maps and that π_{zero} denotes the permission map associating all heap locations – that is, array elements in the context of this chapter – with no permission.

Definition 3.1.5. » A permission expression p is *non-negative* if and only if, for all states σ, we have $\pi_{\mathsf{zero}} \sqsubseteq [\![p]\!]_\Pi(\sigma)$. «

Sufficient Preconditions and Guaranteed Postconditions. Next, we formally define under which conditions a permission expression represents a sufficient precondition or a guaranteed postcondition. To do so, we first remind the reader of the transition relation $\rightsquigarrow^*$ defined in Chapter 2 and that $\sigma \vDash \pi'$ is used to denote that the state $\sigma = \langle s, h, \pi \rangle$ contains at least the permissions represented by π', that is $\pi' \sqsubseteq \pi$.

Definition 3.1.6. » A permission expression p is a *sufficient permission precondition* for a statement c if and only if for all states σ with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$, we have $\langle c, \sigma\rangle \not\rightsquigarrow^* \gamma_\lightning$. «

Example 3.1.7. » Let us consider the statement $c \equiv \mathsf{a}[\mathsf{i}] := \mathsf{a}[\mathsf{j}]$ and the permission expression $p_1 \equiv (\mathsf{q}_a = \mathsf{a} \wedge (\mathsf{q}_i = \mathsf{i} \vee \mathsf{q}_i = \mathsf{j}))\ ?\ 1 : 0$. We observe that any state σ satisfying $\sigma \vDash [\![p_1]\!]_\Pi(\sigma)$ must provide a full permission for $\mathsf{a}[\mathsf{i}]$ as well as for $\mathsf{a}[\mathsf{j}]$. Thus, we have $\langle c, \sigma\rangle \not\rightsquigarrow^* \gamma_\lightning$, meaning that p_1 is a sufficient permission precondition for c.

In contrast, $p_2 \equiv (\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ 1 : 0$ is *no* sufficient precondition for c: A state σ in which $\mathsf{i} \neq \mathsf{j}$ holds and satisfying $\sigma \vDash [\![p_2]\!]_\Pi(\sigma)$ is not guaranteed to provide permissions for $\mathsf{a}[\mathsf{j}]$, making $\langle c, \sigma\rangle \rightsquigarrow^* \gamma_\lightning$ possible. «

Definition 3.1.8. » A permission expression p' is a *guaranteed permission postcondition* for a statement c with respect to the permission precondition p, if and only if, for all pairs of states σ and σ' with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, we have $\sigma' \vDash [\![p']\!]_\Pi(\sigma)$. «

Example 3.1.9. » For all statements whose execution does not give away any permissions, any permission precondition is simultaneously also a sufficient permission postcondition.

Moreover, since any execution of the statement $\mathsf{inhale}\ \mathsf{acc}(\mathsf{a}[\mathsf{i}], 1/2)$ ends up in a state with at least half a permission for $\mathsf{a}[\mathsf{i}]$, permission expression $(\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ 1/2 : 0$ is a guaranteed permission postcondition said statement with respect to any permission precondition. «

Note that guaranteed permission postconditions are expressed in terms of pre-states and are, therefore, evaluated in the state σ.

Permission Expressions as Assertions. We briefly compare our permission expressions to common hand-written specifications. Typically, assertions capturing permissions for array elements are written using universally quantified permissions and are of the form $\forall \mathsf{q}_i \in \mathbb{Z}\colon b \Rightarrow \mathsf{acc}(\mathsf{a}[\mathsf{q}_i])$, where b is a boolean expression determining for which values of q_i a permission for the array element $\mathsf{a}[\mathsf{q}_i]$ has to be provided. For example,

$$\forall \mathsf{q}_i \in \mathbb{Z}\colon 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}) \Rightarrow \mathsf{acc}(\mathsf{a}[\mathsf{q}_i])$$

represents full permission for all elements in the array a's range. In comparison, the permission expression $(0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ 1 : 0$ captures exactly the same permissions; the corresponding assertion would then look something like

$$\forall \mathsf{q}_i \in \mathbb{Z}\colon \mathsf{acc}(\mathsf{a}[\mathsf{q}_i]), (0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ 1 : 0).$$

Note that our permission expressions go one step further than the assertion shown above by also implicitly quantifying over arrays.

In contrast to using a boolean expression b to determine for which array elements permissions are required, using permission expressions has two key advantages: First, permission expressions can easily be used to require different permission amounts for individual array elements (see Example 3.1.1 above) and do not require to partition the specifications into sets of equal permission amounts and finding a condition for each of them. Second, dealing with numerical expressions allows us to easily add and subtract permissions needed when, for example, permissions are inhaled or exhaled.

3.1.3 Pointwise Maxima

As already indicated, we will use pointwise maximum expressions to generalise results obtained for individual loop iterations to the entire loop. The syntax coincides with the one used for standard mathematical notation. The semantics of such pointwise maximum expression is defined as follows:

$$\left[\!\left[\max_{x \in S}\{p\}\right]\!\right](\sigma) :\equiv \max_{e \in S}\{[\![p[e\backslash x]]\!](\sigma)\}$$

As we often deal with pointwise maxima only ranging over variables that satisfy a certain condition, for all permission expressions p and boolean expressions b, we introduce the shorthand

$$\max_{x \mid b}\{p\} :\equiv \max_{x \in \mathbb{Z}}\{b \,?\, p : 0\}.$$

Note that, throughout this chapter, we will consistently typeset pointwise maximum expressions (ranging over an unbounded set of values) using a red maximum symbol $\max$. We do this to make it easier for the reader to distinguish them from regular binary maximum expressions $\max(p_1, p_2)$; this makes it straightforward to spot all the maximum expressions that we would like to ultimately solve using our maximum elimination algorithm.

We define pointwise minimum expression via pointwise maximum expressions rather than providing analogous definitions. That is, we define

$$\min_{x \in S}\{p\} :\equiv -\max_{x \in S}\{-p\}$$
$$\min_{x \mid b}\{p\} :\equiv -\max_{x \mid b}\{-p\}.$$

This allows us to reuse our maximum elimination algorithm to also solve pointwise minimum expressions. Note that this is done solely for the sake of easier presentation; adapting the algorithm to a minimum elimination algorithm is possible and also does not pose any additional challenges.

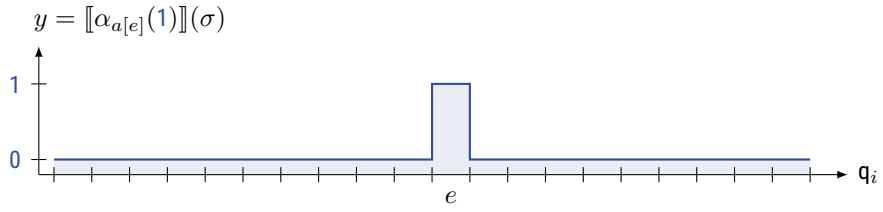


Figure 3.2. A leaf expression $\alpha_{a[e]}(1)$ capturing a full permission for the array element $a[i]$ and no permissions for all other array elements plotted as a function of the designated variable q_i. The state σ is assumed to satisfy $\mathsf{q}_a = a$.

3.2 Loop-Free Code

In this section, we define inference rules to compute sufficient permission preconditions for loop-free code. For programs which do not add or remove permissions via inhale and exhale statements, the same permissions will still be held after executing the code; however, to infer guaranteed permission postconditions in the general case, we also infer the difference in permissions between the state before and after the execution and then combine them with the permission preconditions in order to express permission postconditions. We will discuss loops in the upcoming sections. Non-recursive method calls can be handled by applying our analysis bottom-up in the call graph using inhale and exhale statements to model the permission effect of calls. An extension to recursive method calls is non-trivial and discussed in Section 3.7.3.

Section Outline. In Section 3.2.1, we first introduce leaf-expressions that capture permissions for an individual array element and serve as a basic building block for our inference rules. In Section 3.2.2, we provide the rules to infer sufficient permission preconditions for loop-free code. In Section 3.2.3, we then introduce similar rules that infer the permissions lost and gained by loop-free code that can be combined with the permission preconditions in order to obtain permission postconditions.

3.2.1 Leaf Expressions

The basic building blocks for our permission analysis are so-called *leaf expressions* $\alpha_{a[e]}(q)$ that capture q permissions for the array element $a[e]$ (cf. Figure 3.2). These leaf expressions are defined as follows.

Definition 3.2.1. » For all array variables, all integer expressions e, and all permission fractions q, we define

$$\alpha_{a[e]}(q) :\equiv (\mathsf{q}_a = a \wedge \mathsf{q}_i = e)\ ?\ q : 0. \qquad \text{«}$$

```
method copy_even_b(a: Int[]) {
  var i: Int
  var v: Int
  i := 0
  while (i < (len(a) − 1) / 2) {
    v := a[2 · i]
    a[2 · i + 1] := v
    i := i + 1
  }
}
```

Listing 3.3. A variation of the method copying array elements at even indices.

```
method swap(a: Int[], b: Int[]) {
  var i: Int
  var v: Int
  i := 0
  while (i < len(a)) {
    v := a[i]
    a[i] := b[i]
    b[i] := v
    i := i + 1
  }
}
```

Listing 3.4. A method swapping the contents of two arrays.

Observation 3.2.2. » For all leaf expressions $\alpha_{a[e]}(q)$ and all states $\sigma = \langle s, h, \pi \rangle$, we have $[\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma) = \pi_{u,q}$, where $u := [\![a[e]]\!]_L(\sigma)$ is the heap location to which $a[e]$ evaluates in the state σ (as defined in Chapter 2). In particular, if $\sigma \vDash [\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma)$, then $\pi(u) \geq q$. «

Multi-Dimensional Arrays. Our analysis can be easily extended to also support multi-dimensional arrays or other integer-indexed data structures. This can be done by introducing a slightly adapted version of the leaf expression defined above; all remaining parts of the permission analysis remain unaffected. For instance, in order to support matrices, we could define

$$\alpha_{m[e_1,e_2]}(q) :\equiv (\mathsf{q}_a = m \wedge \mathsf{q}_i = e_1 \wedge \mathsf{q}_j = e_2)\ ?\ q : 0,$$

where q_j is another designated variable ranging over the second index. For the sake of simplicity, however, we restrict our elaborations below to one-dimensional arrays.

3.2.2 Permission Preconditions

The *permission precondition* of a loop-free statement c and a permission expression p is denoted by $\tau\{\!|c|\!\}(p)$. The second parameter p acts as an accumulator and may contain our designated variables q_a and q_i. The *weakest* permission precondition for a statement c is given by $\tau\{\!|c|\!\}(0)$. Most rules for τ are straightforward adaptions of a classical weakest precondition computation. The following definition provides all rules at once; afterwards, we then elaborate on some of the details and ultimately provide a soundness proof.

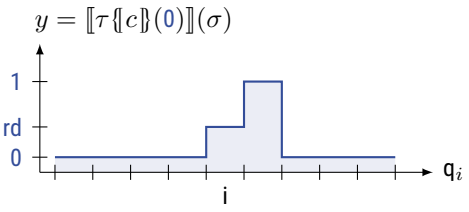


Figure 3.5. The permission precondition for the body c of the copy_even_b method's loop representing sufficient permissions to execute an individual loop iteration.

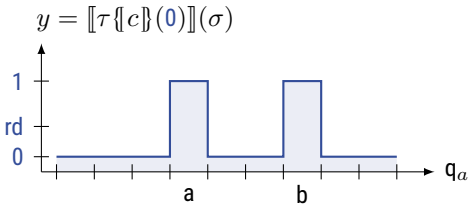


Figure 3.6. The permission precondition for the body c of the copy_swap method's loop. Note that the permission expression is plotted as a function of q_a (as opposed to q_i).

Definition 3.2.3. » For all loop-free statements c and all permission expressions p, we define

$$\tau\{\!|c|\!\}(p) :\equiv \begin{cases} p & \text{if } c \equiv \mathsf{skip} \\ \tau\{\!|c_1|\!\}(\tau\{\!|c_2|\!\}(p)) & \text{if } c \equiv c_1\,;\,c_2 \\ b\,?\,\tau\{\!|c_1|\!\}(p) : \tau\{\!|c_2|\!\}(p) & \text{if } c \equiv \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\} \\ p[e\backslash x] & \text{if } c \equiv x := e \\ \mathsf{max}(p[a[e]\backslash x], \alpha_{a[e]}(\mathsf{rd})) & \text{if } c \equiv x := a[e] \\ \mathsf{max}(p[a[e] \mapsto x], \alpha_{a[e]}(1)) & \text{if } c \equiv a[e] := x \\ b\,?\,p : 0 & \text{if } c \equiv \mathsf{inhale}\ b \text{ or } c \equiv \mathsf{exhale}\ b \\ \mathsf{max}([\![p]\!]_{a[e]} - \alpha_{a[e]}(q), 0) & \text{if } c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q) \\ p + \alpha_{a[e]}(q) & \text{if } c \equiv \mathsf{exhale}\ \mathsf{acc}(a[e], q), \end{cases}$$

where, intuitively, the term $p[a[e] \mapsto x]$ updates the value of the array element $a[e]$ to x in p and the term $[\![p]\!]_{a[e]}$ models havocs (cf. Chapter 2) by abstracting away the value of the array element $a[e]$ in p; these terms are formally defined below in Definition 3.2.10 and Definition 3.2.17, respectively, below. «

As the following examples illustrate, these rules construct permission expressions that capture the permissions for all accessed array elements, even across multiple arrays.

Example 3.2.4. » Let us consider the loop of the copy_even_b method shown in Listing 3.3. We note that the body of this loop is given by the statement $c_1\,;\,c_2\,;\,c_3$, where

$c_1 \equiv \mathsf{v} := \mathsf{a}[2{\cdot}\mathsf{i}]$, $c_2 \equiv \mathsf{a}[2{\cdot}\mathsf{i}{+}1]$, and $c_3 \equiv \mathsf{i} := \mathsf{i}{+}1$. For this statement, our analysis infers the permission precondition

$$\begin{aligned}\tau\{\!|c_1 ; c_2 ; c_3|\!\}(0) &\equiv \tau\{\!|c_1 ; c_2|\!\}(\tau\{\!|c_3|\!\}(0)) \\ &\equiv \tau\{\!|c_1 ; c_2|\!\}(0) \\ &\equiv \tau\{\!|c_1|\!\}(\tau\{\!|c_2|\!\}(0)) \\ &\equiv \tau\{\!|c_1|\!\}(\max\big(0, \alpha_{\mathsf{a}[2\cdot\mathsf{i}+1]}(1)\big)) \\ &\equiv \max\big(\max\big(0, \alpha_{\mathsf{a}[2\cdot\mathsf{i}+1]}(1)\big), \alpha_{\mathsf{a}[2\cdot\mathsf{i}]}(\mathsf{rd})\big).\end{aligned}$$

We observe that this permission expression captures both, permission to read from the array element $\mathsf{a}[2{\cdot}\mathsf{i}]$ as well as permission to write to the array element $\mathsf{a}[2{\cdot}\mathsf{i}{+}1]$; a visual representation is shown in Figure 3.5. «

Example 3.2.5. » Let us consider the loop of the `swap` method shown in Listing 3.4. For the body of this loop, our analysis infers the permission precondition

$$\max\big(\max\big(\max\big(\max\big(0, \alpha_{\mathsf{a}[\mathsf{i}]}(1)\big), \alpha_{\mathsf{b}[\mathsf{i}]}(1)\big), \alpha_{\mathsf{b}[\mathsf{i}]}(\mathsf{rd})\big), \alpha_{\mathsf{a}[\mathsf{i}]}(\mathsf{rd})\big),$$

which can be simplified to

$$\max\big(\alpha_{\mathsf{a}[\mathsf{i}]}(1), \alpha_{\mathsf{b}[\mathsf{i}]}(1)\big).$$

This permission expression grants write permissions to the array elements $\mathsf{a}[\mathsf{i}]$ and $\mathsf{b}[\mathsf{i}]$. Note that using a maximum instead of an addition ensures that we do not require two permissions for the same array element in case the array variables `a` and `b` alias, that is, refer to the same array. «

Soundness. Our permission preconditions for loop-free code satisfies the following soundness lemma.

Lemma 3.2.6. » *For all loop-free statements c, the permission expression $\tau\{\!|c|\!\}(0)$ is a sufficient permission precondition for c.* «

Intuitively, whenever our analysis encounters a statement that requires q permissions for an array element $a[e]$, our rules ensure that the constructed precondition is at least $\alpha_{a[e]}(q)$. This way, according to Observation 3.2.2, the permission expression constructed by our analysis represents sufficient permissions to successfully execute this *individual* statement. It remains to convince ourselves, that the parameter p appropriately accumulates the permissions required for the execution of *compound* statements, which we will do in what follows, and then ultimately prove Lemma 3.2.6.

Variable Assignments. To prove the correctness of the rule for variable assignments, we employ a fairly standard substitution lemma, which we formulate as follows.

Lemma 3.2.7. » *For all permission expressions p, all integer-typed variables x, all arithmetic expressions e, and all states $\sigma = \langle s, h, \pi \rangle$, we have*

$$[\![p[e\backslash x]]\!](\sigma) = [\![p]\!](\langle s[x \mapsto [\![e]\!](\sigma)], h, \pi \rangle).$$ «

Proof. By straightforward induction on the structure of the permission expression p.□

With this substitution lemma at hand, we easily see that our rules for assignments to variables correctly update the accumulated permissions p to reflect the effect of the assignment.

Lemma 3.2.8. » *For all permission expressions p, integer-typed variables x, arithmetic expressions e, and pairs of states σ and σ' with $\langle x := e, \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma' \rangle$, we have*

$$\sigma \vDash [\![p[e\backslash x]]\!]_\Pi(\sigma) \quad \Leftrightarrow \quad \sigma' \vDash [\![p]\!]_\Pi(\sigma').$$ «

Proof. The claim immediately follows by combining Lemma 3.2.7 with the definition of the variable assignment rule in Section 2.3.4. □

Example 3.2.9. » We have $\tau\{\!|\mathsf{i} := 0|\!\}(\alpha_{\mathsf{a}[\mathsf{i}]}(1)) \equiv \alpha_{\mathsf{a}[0]}(1)$. «

Array Assignments. Assignments updating array values $a[e]$ are handled similarly to variable assignments. However, here, we also have to take into account potential aliases: We cannot just syntactically replace all occurrences of $a[e]$ in the accumulator expression p as there might be syntactically different array accesses $a'[e']$ that represent the same array value; that is, when $a' = a$ and $e' = e$. This motivates the following definition and subsequent substitution lemma for array updates.

Definition 3.2.10. » For all permission expressions p, all array variables a, and all arithmetic expressions e and e'', let

$$p[a[e] \Rrightarrow e'']$$

the permission expression obtained from p by replacing each occurrence of an array access $a'[e']$ with $(a' = a \wedge e' = e)\ ?\ e'' : a'[e']$. «

Lemma 3.2.11. » *For all permission expressions p, all array variables a, all arithmetic expressions e and e', and all states $\sigma = \langle s, h, \pi \rangle$, we have*

$$[\![p[a[e] \Rrightarrow e']]\!](\sigma) = [\![p]\!](\langle s, h[[\![a[e]]\!]_L(\sigma) \mapsto [\![e']\!](\sigma)], \pi \rangle).$$ «

Proof. By straightforward induction on the structure of the permission expression p.□

Lemma 3.2.12. » *For all permission expressions p, array variables a, arithmetic expressions e, variables x, and pairs of states σ and σ' with $\langle a[e] := x, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma'\rangle$, we have*

$$\sigma \vDash [\![p[a[e] \mapsto x]]\!]_\Pi(\sigma) \quad \Leftrightarrow \quad \sigma' \vDash [\![p]\!]_\Pi(\sigma').$$ «

Proof. The claim follows by combining Lemma 3.2.11 with the definition of the heap assignment rules in Section 2.3.4. □

Example 3.2.13. » Suppose that our analysis has already accumulated a permission expression of the form $p \equiv (\mathsf{a}[\mathsf{i}] = 0)\ ?\ p_1 : p_2$; such an expression occurs, for example, when the code at hand contains a conditional statement that branches on the condition $\mathsf{a}[\mathsf{i}] = 0$. Assuming the next encountered statement is $\mathsf{a}[\mathsf{j}] := 0$, we get

$$\tau\{\!|\, \mathsf{a}[\mathsf{j}] := 0 \,|\!\}(p) \equiv ((\mathsf{a} = \mathsf{a} \wedge \mathsf{i} = \mathsf{j})\ ?\ 0 : \mathsf{a}[\mathsf{i}]) = 0\ ?\ p_1 : p_2.$$

We observe that in a state where $\mathsf{i} = \mathsf{j}$ this entire expression evaluates to p_1; otherwise, it is equivalent to the original expression p (assuming no further array values appear within p_1 or p_2), which is expected as the condition $\mathsf{a}[\mathsf{i}] = 0$ remains unaffected by the array update. «

Thread Interference. Recall from Section 2.3.4 that our program semantics models interference by potential concurrently running threads via the rules for inhale $\mathsf{acc}(a[e], q)$ statements: Another thread can only update a heap location if we currently do not hold any permission for that heap location. Therefore, whenever we inhale permission for an array element $a[e]$ for which we previously did not hold any permission, we conservatively assume that its value may have been altered. We reflect this in our analysis by abstracting away the value of the array element $a[e]$ whenever such an inhale statement is encountered. To facilitate this, we introduce an operator that allows us to abstract away the value of a given array element $a[e]$ in a permission expression p. We need both an over-approximate version $\lceil p \rceil_{a[e]}$ and an under-approximate version $\lfloor p \rfloor_{a[e]}$ of this operator; the latter will be used for the postconditions described later. Below, we first define two operators $\lceil b \rceil_{a[e]}$ and $\lfloor b \rfloor_{a[e]}$ that abstract away the value of array element $a[b]$ for boolean expressions b; afterwards, we then show how to extend these operators to permission expressions.

Definition 3.2.14. » Consider some array variable a and integer expression e. For any comparison $e_1 \circ e_2$, where $\circ \in \{=, \neq, <, \leq, >, \geq\}$, we define

$$\lceil e_1 \circ e_2 \rceil_{a[e]} := \left(\bigvee_{\langle a', e' \rangle \in A} (a' = a \wedge e' = e) \right) ?\ \mathsf{true} : (e_1 \circ e_2),$$

where A is the set of all tuples $\langle a', e' \rangle$ such that the array access $a'[e']$ appears in $e_1 \circ e_2$. The under-approximate version $\llbracket e_1 \circ e_2 \rrbracket_{a[e]}$ is defined analogously but with true replaced with false. For all remaining boolean expressions, the operators are defined recursively; for example, we define $\lceil b_1 \wedge b_2 \rceil_{a[e]} :\equiv \lceil b_1 \rceil_{a[e]} \wedge \lceil b_2 \rceil_{a[e]}$ and $\lceil \neg b \rceil_{a[e]} :\equiv \neg \llbracket b \rrbracket_{a[e]}$. «

Example 3.2.15. » Suppose our analysis has already accumulated a permission expression, in which the condition $b \equiv \mathsf{a}[\mathsf{i}] = 0$ appears. Moreover, suppose that we encounter an inhale acc(a[j], 1) statement that requires us to abstract away the value of the array element a[j] in this condition. We can do this by computing the term

$$\lceil b \rceil_{\mathsf{a}[\mathsf{j}]} \equiv (\mathsf{a} = \mathsf{a} \wedge \mathsf{i} = \mathsf{j}) \,?\, \mathsf{true} : (\mathsf{a}[\mathsf{i}] = 0).$$

Note that, in a state where $\mathsf{i} = \mathsf{j}$, the term $\lceil b \rceil_{\mathsf{a}[\mathsf{j}]}$ evaluates to true. Conversely, in a state where $\mathsf{i} \neq \mathsf{j}$, the term $\lceil b \rceil_{\mathsf{a}[\mathsf{j}]}$ evaluates to the same value as the original condition b. This correctly preserves the value of the condition b if and only if a[i] and a[j] refer to different array elements. «

Lemma 3.2.16. » *For all permission expressions p, all array variables a, all integer expressions e, all states $\sigma = \langle s, h, \pi \rangle$, and all values v, we have*

1. $[\![b]\!](\sigma') \Rightarrow [\![\lceil b \rceil_{a[e]}]\!](\sigma)$ *and*
2. $[\![\llbracket b \rrbracket_{a[e]}]\!](\sigma) \Rightarrow [\![b]\!](\sigma')$,

where $\sigma' := \langle s, h[[\![a[e]]\!]_L(\sigma) \mapsto v], \pi \rangle$. «

Proof. The claim follows by induction on the structure of the boolean expression b. The only slightly non-trivial case is for comparisons $b \equiv e_1 \circ e_2$: It is easy to see that the truth value of such a comparison depends on the value of the array element $a[e]$ if and only if the condition $\bigvee_{\langle a', e' \rangle \in A} (a' = a \wedge e' = e)$ in the definition above evaluates to true. If this is the case, we over-approximate the comparison by true (respectively, under-approximate it by false), otherwise, we use its original value $e_1 \circ e_2$. □

Definition 3.2.17. » For all permission expressions p, array variables a, and integer expressions e, we define

$$\lceil p \rceil_{a[e]} :\equiv \begin{cases} q & \text{if } p \equiv q \\ \lceil p_1 \rceil_{a[e]} + \lceil p_2 \rceil_{a[e]} & \text{if } p \equiv p_1 + p_2 \\ \lceil p_1 \rceil_{a[e]} - \llbracket p_2 \rrbracket_{a[e]} & \text{if } p \equiv p_1 - p_2 \\ \mathsf{min}(\lceil p_1 \rceil_{a[e]}, \lceil p_2 \rceil_{a[e]}) & \text{if } p \equiv \mathsf{min}(p_1, p_2) \\ \mathsf{max}(\lceil p_1 \rceil_{a[e]}, \lceil p_2 \rceil_{a[e]}) & \text{if } p \equiv \mathsf{max}(p_1, p_2) \\ \mathsf{max}((\lceil b \rceil_{a[e]} \,?\, \lceil p_1 \rceil_{a[e]} : 0), (\lceil \neg b \rceil_{a[e]} \,?\, \lceil p_2 \rceil_{a[e]} : 0)) & \text{if } p \equiv b \,?\, p_1 : p_2, \end{cases}$$

where operator $\lfloor\!\lfloor p \rfloor\!\rfloor_{a[e]}$ is defined analogously but with all approximation orders flipped (that is, all over-approximations replaced by under-approximations and vice versa). «

Lemma 3.2.18. » *For all permission expressions p, all array variables a, all integer expressions e, all states $\sigma = \langle s, h, \pi \rangle$, and all values v, we have*

1. $[\![p]\!]_\Pi(\sigma') \sqsubseteq [\![\lceil p \rceil_{a[e]}]\!]_\Pi(\sigma)$ *and*
2. $[\![\lfloor\!\lfloor p \rfloor\!\rfloor_{a[e]}]\!]_\Pi(\sigma) \sqsubseteq [\![p]\!]_\Pi(\sigma')$,

where $\sigma' \coloneqq \langle s, h[[\![a[e]]\!]_L(\sigma) \mapsto v], \pi \rangle$. «

Proof. By Lemma 3.2.16 and straightforward induction on the structure of the permission expression p. □

Now – using the over-approximate abstraction operator $\lceil p \rceil_{a[e]}$ – we can construct a permission expression that represents permissions required *after* an inhale statement in terms of the state *before* its execution. This is formalised by the following lemma.

Lemma 3.2.19. » *For all permission expressions p, all array variables a, and all integer expressions e, permission fractions q, and all pairs of states σ and σ' with*

$$\sigma \vDash [\![\lceil p \rceil_{a[e]} - \alpha_{a[e]}(q)]\!]_\Pi(\sigma)$$

and $\langle \mathsf{inhale}\ \mathsf{acc}(a[e], q), \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma' \rangle$, we have $\sigma' \vDash [\![p]\!]_\Pi(\sigma')$. «

Proof. The claim immediately follows by combining Lemma 3.2.18 with the definition of the inhale rules: The term $\lceil p \rceil_{a[e]}$ makes sure that we can guarantee that the permissions p are preserved without having to express them in terms of the array value $a[e]$ in the pre-state, while $\alpha_{a[e]}(q)$ accounts for the gained permissions. □

Preserving Permissions. Next, we prove that the permissions represented by second parameter p of our precondition operator $\tau\{\!|c|\!\}(p)$ are guaranteed to be preserved throughout the execution of c (of course, only if the permissions $\tau\{\!|c|\!\}(p)$ were present in the first place). This is important, as it allows us to use the parameter p to accumulate required permissions across compound statements.

Lemma 3.2.20. » *For all loop-free statements c, all pairs of permission expressions p and p' with $p \equiv \tau\{\!|c|\!\}(p')$, and all pairs of states σ and σ' with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ with $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$, we have $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$.* «

Proof. We prove the claim by induction on the structure of the loop-free statement c. To this end, let us consider an arbitrary loop-free statement c and – as our induction hypothesis – suppose that the claim holds for all of its sub-statements. Moreover, we consider arbitrary permission expressions p and p' with $p \equiv \tau\{\!|c|\!\}(p')$ and arbitrary states σ and σ' with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. We proceed by distinguishing the following cases:

- Case $c \equiv \mathsf{skip}$: In this case, we have $\sigma = \sigma'$ and $p \equiv p'$. Thus, we trivially get $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$.

- Case $c \equiv c_1 \,;\, c_2$: According to Lemma 2.3.8, there is a state σ'' such that $\langle c_1, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma''\rangle$ and $\langle c_2, \sigma''\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. Note that $p \equiv \tau\{\!|c_1|\!\}(\tau\{\!|c_2|\!\}(p'))$. We define $p'' :\equiv \tau\{\!|c_2|\!\}(p')$ and observe that $p \equiv \tau\{\!|c_1|\!\}(p'')$. Thus, applying the induction hypothesis to c_1 yields $\sigma'' \vDash [\![p'']\!]_\Pi(\sigma'')$. Applying the induction hypothesis another time to c_2 then yields $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$ as required.

- Case $c \equiv \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}$: We assume that $[\![b]\!](\sigma)$; the case where $\neg[\![b]\!](\sigma)$ follows entirely analogously. Here, we have $p \equiv b\,?\,\tau\{\!|c_1|\!\}(p') : \tau\{\!|c_2|\!\}(p')$. Using $[\![b]\!](\sigma)$, we observe that

  $$[\![p]\!]_\Pi(\sigma) = [\![b\,?\,\tau\{\!|c_1|\!\}(p') : \tau\{\!|c_2|\!\}(p')]\!]_\Pi(\sigma) = [\![\tau\{\!|c_1|\!\}(p')]\!]_\Pi(\sigma)$$

  and, therefore, $\sigma \vDash [\![\tau\{\!|c_1|\!\}(p')]\!]_\Pi(\sigma)$. Moreover, we observe that unrolling the first step of our derivation sequence yields $\langle c, \sigma\rangle \rightsquigarrow \langle c_1, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. With this, we can apply the induction hypothesis to c_1 and obtain $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$

- Case $c \equiv x := e$: We observe that $p \equiv p'[e\backslash x]$ and, therefore, $\sigma \vDash [\![p'[e\backslash x]]\!]_\Pi(\sigma)$. Consequently, this case immediately follows from Lemma 3.2.8.

- Case $c \equiv x := a[e]$: We observe that $p \equiv \mathsf{max}(p'[a[e]\backslash x], \alpha_{a[e]}(\mathsf{rd}))$ and, therefore, $\sigma \vDash [\![p'[a[e]\backslash x]]\!]_\Pi(\sigma)$. Thus, this case also immediately follows from Lemma 3.2.8.

- Case $c \equiv a[e] := x$: Here, we observe that $p \equiv \mathsf{max}(p'[a[e] \Mapsto x], \alpha_{a[e]}(1))$ and, therefore, $\sigma \vDash [\![p'[a[e] \Mapsto x]]\!]_\Pi(\sigma)$. This case then follows from Lemma 3.2.12.

- Cases $c \equiv \mathsf{inhale}\ b$ and $c \equiv \mathsf{exhale}\ b$: We note that $p \equiv b\,?\,p' : 0$. For any state σ satisfying $\neg[\![b]\!](\sigma)$, the only possible derivation sequence is $\langle c, \sigma\rangle \rightsquigarrow^* \gamma_\times$, which contradicts our assumption that $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$; therefore, we must have $[\![b]\!](\sigma)$. Additionally, the rules for inhaling and exhaling boolean expressions dictate that $\sigma = \sigma'$. Thus, we can also conclude $\sigma' \vDash [\![p']\!]_\Pi(\sigma)$ for this case.

- Case $c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q)$: We observe that $p \equiv \mathsf{max}(\llbracket p' \rrbracket_{a[e]} - \alpha_{a[e]}(q), 0)$ and, therefore, $\sigma \vDash \llbracket \llbracket p' \rrbracket_{a[e]} - \alpha_{a[e]}(q) \rrbracket_\Pi(\sigma)$. Thus, this case immediately follows from Lemma 3.2.19.

- Case $c \equiv \mathsf{exhale}\ \mathsf{acc}(a[e], q)$: Here, we have $p \equiv p' + \alpha_{a[e]}(q)$ and, therefore, $\sigma \vDash \llbracket p' + \alpha_{a[e]}(q) \rrbracket_\Pi(\sigma)$. Moreover, we observe that the only rule that justifies $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$ is the successful exhale rule. Thus, we have $\sigma' \vDash \llbracket p' \rrbracket_\Pi(\sigma')$ for this case too. □

Non-Negativity. Another ingredient that we need for our soundness proof is that our permission analysis only produces non-negative permission preconditions, which is formalised by the following lemma

Lemma 3.2.21. » *For all loop-free statements c and non-negative permission expressions p, the permission expression $\tau\{\!|c|\!\}(p)$ is also non-negative.* «

Proof. By straightforward induction on the structure of the loop-free statement c.□

Soundness Proof. We are now ready to combine the individual findings from above into a proof for Lemma 3.2.6 stating that our permission preconditions for loop-free code are sound. In order to be able to prove the claim by induction on the structure of the statement at hand, we prove a slightly stronger claim formalised by the following lemma:

Lemma 3.2.22. » *For all loop-free statements c, all non-negative permission expressions p, and all states σ with $\sigma \vDash \llbracket \tau\{\!|c|\!\}(p) \rrbracket_\Pi(\sigma)$, we have $\langle c, \sigma \rangle \not\rightsquigarrow^* \gamma_{\lightning}$.* «

Proof. We prove the claim by induction on the structure of the loop-free statement c. To do so, we consider an arbitrary loop-free statement c and assume that the claim holds for all of its sub-statements. Furthermore, we consider an arbitrary non-negative permission expression p and state σ with $\sigma \vDash \tau\{\!|c|\!\}(p)$. We proceed by distinguishing the following cases:

- Case $c \equiv \mathsf{skip}$: In this case, the claim follows trivially.

- Case $c \equiv c_1\ ;\ c_2$: According to Lemma 2.3.9, it suffices to show that
  1. $\langle c_1, \sigma \rangle \not\rightsquigarrow^* \gamma_{\lightning}$ and
  2. $\langle c_2, \sigma' \rangle \not\rightsquigarrow^* \gamma_{\lightning}$, for all states σ' with $\langle c_1, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$.

We prove both of these cases separately:

First, we note that $\tau\{\!|c|\!\}(p) \equiv \tau\{\!|c_1|\!\}(\tau\{\!|c_2|\!\}(p))$. We define $p' :\equiv \tau\{\!|c_2|\!\}(p)$ and observe that $\sigma \vDash [\![\tau\{\!|c_1|\!\}(p')]\!]_\Pi(\sigma)$. By Lemma 3.2.21, p' is non-negative. Thus, applying our induction hypothesis to c_1 yields $\langle c_1, \sigma\rangle \not\rightsquigarrow^* \gamma_{\lightning}$.

Second, we consider an arbitrary state σ' with $\langle c_1, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. Using Lemma 3.2.20, we get that $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$. Therefore, we can apply our induction hypothesis to c_2 and obtain $\langle c_2, \sigma'\rangle \not\rightsquigarrow^* \gamma_{\lightning}$, as required.

- Case $c \equiv \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}$: We assume $[\![b]\!](\sigma)$; the case where $\neg[\![b]\!](\sigma)$ is analogous. With the same line of arguments as in the corresponding case of the proof of Lemma 3.2.20, we also get $\sigma \vDash [\![\tau\{\!|c_1|\!\}(p)]\!]_\Pi(\sigma)$ here. As the first step of any derivation sequence starting from $\langle c, \sigma\rangle$ has to be $\langle c, \sigma\rangle \rightsquigarrow \langle c_1, \sigma\rangle$, it remains to show that $\langle c_1, \sigma\rangle \not\rightsquigarrow^* \gamma_{\lightning}$, which follows from applying the induction hypothesis to c_1.

- Case $c \equiv x := e$: In this case, the claim trivially follows as the rules for a regular variable assignment never lead to a permission failure.

- Case $c \equiv x := a[e]$: We have $\tau\{\!|c|\!\}(p) \equiv \mathsf{max}(p[a[e]\backslash x], \alpha_{a[e]}(\mathsf{rd}))$ and therefore $\sigma \vDash [\![\alpha_{a[e]}(\mathsf{rd})]\!]_\Pi(\sigma)$. Consequently, by Observation 3.2.2, the only applicable rule matching the initial configuration $\langle c, \sigma\rangle$ is the one that successfully executes the assignment. Thus, we also have $\langle c, \sigma\rangle \not\rightsquigarrow^* \gamma_{\lightning}$ in this case.

- Case $c \equiv a[e] := x$: Analogous to the previous case.

- Cases $c \equiv \mathsf{inhale}\ b$ and $c \equiv \mathsf{exhale}\ b$. This case follows from the observation that inhaling or exhaling a boolean expression (not accessing accessing any array elements) cannot result in a permission failure.

- Case $c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q)$: In this case, the claim trivially follows as the rules for the execution of an inhale statement never lead to a permission failure. Note that we can safely ignore the rule leading to the ghost configuration $\gamma_\times$.

- Case $c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q)$: We observe that $\tau\{\!|c|\!\}(p) \equiv p + \alpha_{a[e]}(q)$. Thus, by the non-negativity of p, we have $\sigma \vDash [\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma)$. Consequently, using Observation 3.2.2, we can also conclude $\langle c, \sigma\rangle \not\rightsquigarrow^* \gamma_{\lightning}$ in this case. □

Proof (of Lemma 3.2.6). The claim immediately follows by picking $p \equiv 0$ in Lemma 3.2.22 above. □

With this, we have proven that our permission analysis indeed produces sufficient permission preconditions. Next, we will turn our attention to permission postconditions for loop-free code.

3.2.3 Permission Postconditions

The final state of a method execution includes the permissions held in the method pre-state, adjusted by the permissions that are inhaled or exhaled during the method execution. To perform this adjustment, we compute the difference in permissions before and after executing a statement.

Permission Difference. The *relative permission difference* for a loop-free statement c and a permission expression p – in which q_a and q_i potentially occur – is denoted by $\delta\{\!|c|\!\}(p)$ and defined backward, analogously to $\tau\{\!|c|\!\}(p)$ in Definition 3.2.3. Analogous to the τ operator, the second parameter p of δ acts as an accumulator; the difference in permission before and after executing c is represented by evaluating $\delta\{\!|c|\!\}(0)$.

Definition 3.2.23. » For all loop-free statements c and all permission expressions p, we define

$$\delta\{\!|c|\!\}(p) :\equiv \begin{cases} p & \text{if } c \equiv \mathsf{skip} \\ \delta\{\!|c_1|\!\}(\delta\{\!|c_2|\!\}(p)) & \text{if } c \equiv c_1\,;\,c_2 \\ b\,?\,\delta\{\!|c_1|\!\}(p) : \delta\{\!|c_2|\!\}(p) & \text{if } c \equiv \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\} \\ p[e\backslash x] & \text{if } c \equiv x := e \\ p[a[e]\backslash x] & \text{if } c \equiv x := a[e] \\ p[a[e] \mapsto x] & \text{if } c \equiv a[e] := x \\ b\,?\,p : 0 & \text{if } c \equiv \mathsf{inhale}\ b \text{ or } c \equiv \mathsf{exhale}\ b \\ \lVert p \rVert_{a[e]} + \alpha_{a[e]}(q) & \text{if } c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q) \\ p - \alpha_{a[e]}(q) & \text{if } c \equiv \mathsf{exhale}\ \mathsf{acc}(a[e], q) \end{cases}$$

«

Lemma 3.2.24. » *For all loop-free statements c, all permission expressions p and p' with $p \equiv \delta\{\!|c|\!\}(p')$, and all pairs of states $\sigma = \langle s, h, \pi\rangle$ and $\sigma' = \langle s', h', \pi'\rangle$ with $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, we have*

$$[\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \sqsubseteq \pi' - \pi.$$

In particular, if $p' \equiv 0$, this implies $[\![p]\!]_\Pi(\sigma) \sqsubseteq \pi' - \pi$. «

Proof. We prove the claim by induction on the structure of the loop-free statement c. Thus, we consider an arbitrary loop-free statement c and assume that the claim hold for all of its sub-statements. Moreover, we consider two arbitrary permission expressions p and p' with $p \equiv \delta\{\!|c|\!\}(p')$, as well as two arbitrary states $\sigma = \langle s, h, \pi \rangle$ and $\sigma' = \langle s', h', \pi' \rangle$ with $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma \rangle$. We proceed by distinguishing the following cases:

- Case $c \equiv \mathsf{skip}$: In this case, the claim follows trivially.

- Case $c \equiv c_1 \,;\, c_2$: We define $p'' :\equiv \delta\{\!|c_2|\!\}(p')$ and observe that $p \equiv \delta\{\!|c_1|\!\}(p'')$. Moreover, according to Lemma 2.3.8, there is a state $\sigma'' = \langle s'', h'', \pi'' \rangle$ such that $\langle c_1, \sigma' \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'' \rangle$ and $\langle c_2, \sigma'' \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$. Thus, applying the induction hypothesis to the statements c_1 and c_2 yields

$$\begin{aligned} [\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') &= ([\![p]\!]_\Pi(\sigma) - [\![p'']\!]_\Pi(\sigma'')) + ([\![p'']\!]_\Pi(\sigma'') - [\![p']\!]_\Pi(\sigma')) \\ &\sqsubseteq (\pi'' - \pi) + (\pi' - \pi'') = \pi' - \pi, \end{aligned}$$

  as required.

- Case $c \equiv \mathsf{if}\ (b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}$: Here, the claim follows by applying the induction hypothesis to either c_1 or c_2, depending on whether $[\![b]\!](\sigma)$ evaluates to true or false.

- In the cases $c \equiv x := e$, $c \equiv x := a[e]$, and $c \equiv a[e] := x$, the execution of the statement c does not add or remove any permissions and, thus, $\pi = \pi'$. Depending on whether the assignment is to a variable or an array element, we employ either Lemma 3.2.8 or Lemma 3.2.12 to establish that $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ if and only if $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$. Consequently – as the permission maps of the states σ and σ' are equal – we get $[\![p]\!]_\Pi(\sigma) = [\![p']\!]_\Pi(\sigma')$, from which the claim immediately follows.

- Cases $c \equiv \mathsf{inhale}\ b$ and $c \equiv \mathsf{exhale}\ b$: We observe that the derivation sequence $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$ is only possible if $[\![b]\!](\sigma)$ and $\sigma = \sigma'$. Moreover, we note that $p \equiv b\,?\,p : 0$. Thus, we have $[\![p]\!]_\Pi(\sigma) = [\![p']\!]_\Pi(\sigma')$ and arrive at the claim analogous to the previous case.

- Case $c \equiv \mathsf{inhale}\ \mathsf{acc}(a[e], q)$: Let $u := [\![a[e]]\!]_L(\sigma)$ denote the heap location corresponding to the array element $a[e]$. By matching the rules for inhale

statements, we obtain that $\sigma' = \langle s, h[u \mapsto v], \pi + \pi_{u,q} \rangle$, for some value v. Moreover, we observe that $p \equiv \lfloor\!\lfloor p' \rfloor\!\rfloor_{a[e]} + \alpha_{a[e]}(q)$. Thus, we have

$$\begin{aligned}
[\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') &= [\![\lfloor\!\lfloor p' \rfloor\!\rfloor_{a[e]}]\!]_\Pi(\sigma) + [\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&\sqsubseteq [\![p']\!]_\Pi(\sigma') + [\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&= [\![\alpha_{a[e]}(q)]\!]_\Pi(\sigma) = \pi_{u,q} = \pi' - \pi,
\end{aligned}$$

where the inequality follows from Lemma 3.2.18 and the penultimate step according to Observation 3.2.2.

- Case $c \equiv \mathsf{exhale\ acc}(a[e], q)$: Analogous to the previous case, except that there is no abstracting away of the array value $a[e]$. □

Permission Postconditions. We are now ready to prove that we can use the permission difference as described above to express guaranteed permission postconditions for loop-free code.

Lemma 3.2.25. » *For all loop-free statements c with permission precondition $\tau\{\!|c|\!\}(0)$, the permission expression $\tau\{\!|c|\!\}(0) + \delta\{\!|c|\!\}(0)$ is a guaranteed permission postcondition.* «

Proof. We consider an arbitrary loop-free statement c and an arbitrary pair of states $\sigma = \langle s, h, \pi \rangle$ and $\sigma' = \langle s', h', \pi' \rangle$ with $\sigma \vDash \tau\{\!|c|\!\}(0)$ and $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$. We observe that

$$\begin{aligned}
[\![\tau\{\!|c|\!\}(0) + \delta\{\!|c|\!\}(0)]\!]_\Pi(\sigma) &= [\![\tau\{\!|c|\!\}(0)]\!]_\Pi(\sigma) + [\![\delta\{\!|c|\!\}(0)]\!]_\Pi(\sigma) \\
&\sqsubseteq \pi + \pi' - \pi = \pi',
\end{aligned}$$

where the inequality uses $[\![\tau\{\!|c|\!\}(0)]\!]_\Pi(\sigma) \sqsubseteq \pi$ together with Lemma 3.2.24. Thus, we have $\sigma' \vDash [\![\tau\{\!|c|\!\}(0) + \delta\{\!|c|\!\}(0)]\!]_\Pi(\sigma)$, as claimed. □

For code that does not inhale or exhale any permissions, the permission postcondition is – unsurprisingly – equal to the permission precondition. The following two simple examples illustrate how our difference operator accounts for permissions that are lost or gained, respectively, by the execution of the statement at hand.

Example 3.2.26. » Let us consider the statement $c \equiv \mathsf{a[i]} := \mathsf{v}$; $\mathsf{exhale\ acc(a[i])}$. Our analysis infers the permission precondition $\tau\{\!|c|\!\}(0) \equiv \mathsf{max}(\alpha_{\mathsf{a[i]}}(1), 0 + \alpha_{\mathsf{a[i]}}(1))$, which simplifies to $\alpha_{\mathsf{a[i]}}(1)$ and represents a full permission for the array element $\mathsf{a[i]}$. The permission difference is $\delta\{\!|c|\!\}(0) \equiv 0 - \alpha_{\mathsf{a[i]}}(1)$. Thus, the permission expression $\tau\{\!|c|\!\}(0) + \delta\{\!|c|\!\}(0)$ simplifies to 0, leaving us with no permissions in the postcondition; which is expected due to the $\mathsf{exhale\ acc(a[i])}$ statement. «

Example 3.2.27. » Let us consider the statement $c \equiv \texttt{inhale acc(a[i])} \,;\, \texttt{a[i]} := \texttt{v}$. Here, we have $\tau(\!|c|\!)(0) \equiv \texttt{max}(\texttt{max}(\alpha_{\texttt{a[i]}}(1), 0) - \alpha_{\texttt{a[i]}}(1), 0)$, which simplifies to 0. The permission difference is $\delta(\!|c|\!)(0) \equiv 0 + \alpha_{\texttt{a[i]}}(1)$. Thus, we can easily see that the permission expression $\tau(\!|c|\!)(0) + \delta(\!|c|\!)(0)$ simplifies to $\alpha_{\texttt{a[i]}}(1)$. That is, the precondition requires no permissions, while the postcondition guarantees a full permission for a[i], which is obtained through the inhale acc(a[i]) statement. «

Note that, since our δ operator performs a backward analysis, our permission postconditions are expressed in terms of the pre-state of the execution of c. To obtain classical postconditions, any heap access need to refer to the pre-state heap, which can be achieved in program logics by using old expressions or logical variables. Formalising the postcondition inference as a backward analysis simplifies our treatment of loops and has technical advantages over classical strongest postconditions. A limitation of our approach is that our postconditions cannot capture situations in which a statement obtains permissions to locations for which no pre-state expression exists – for example, when new arrays are allocated. Our postconditions are sound; to make them precise for such cases, our inference needs to be combined with an additional forward analysis, which we leave as future work.

3.3 Handling Loops

In this section, we extend our precondition operator τ and difference operator δ to also handle loops. A sufficient permission precondition for a loop guarantees the absence of permission failures for a potentially unbounded number of executions of the loop body. This concept is different from a loop invariant: we require a precondition for all executions of a particular loop, but it need not be inductive. We will explain how inductive permission loop invariants can be obtained in Section 3.4 below.

In the following, we describe how we can obtain a loop precondition by projecting a permission precondition for a single loop iteration over all possible initial states for the loop executions. Our analysis expresses the permissions required by a loop using a pointwise maximum operator that expresses a maximum over an unbounded set of values. As this operator is typically not supported by tools such as SMT solvers, we will provide an algorithm for eliminating these pointwise maxima in Section 3.5.

Section Outline. In Section 3.3.1, we focus on obtaining a sufficient permission precondition for the execution of a single loop in isolation – independently of the code before or after it. Similarly, in Section 3.3.2, we describe how to obtain a permission

expression that summarises the permissions lost and gained by *all* iterations of a loop combined. In Section 3.3.3, we then extend our permission analysis to generate permission preconditions and postconditions for arbitrarily nested loops.

3.3.1 Permission Preconditions for Isolated Loops

For our elaborations below, let us consider an arbitrary but fixed loop $l \equiv \texttt{while}\ (b)\ \{c\}$, where c is assumed to be a loop-free statement; nested loops are treated later in Section 3.3.3.

Individual Loop Iterations. Using our permission analysis described above, we can compute a sufficient permission precondition $p :\equiv \tau\{\!|c|\!\}(0)$ for the body of the loop c. We observe that the precise permissions represented by p depend on the initial state of the loop iteration. That is, p is parameterised by the variables $\vec{x} = \langle x_1, \ldots, x_n \rangle$ appearing within the loop body c; for the sake of simplicity, we suppose that all these variables are integer-typed.

Assuming that the permission expression p does not depend on any array values, we can use value vectors $\vec{v} = \langle v_1, \ldots, v_n \rangle \in \mathbb{Z}^n$ to capture permissions required by individual loop iterations: More specifically, the permission expression $p[\vec{v} \setminus \vec{x}]$ captures sufficient permissions to execute the loop body starting in a state σ, where $[\![x_i]\!](\sigma) = v_i$, for all $i \in \{1, \ldots, n\}$.

Conditional Approximation. In general, a permission precondition for a loop body may also depend on array *values*; for example, if those values are used in any branch conditions. To avoid the need for an expensive array value analysis, we abstract away array dependent conditions. To this end, we first define both an over-approximation $\lceil\!\lceil b \rceil\!\rceil$ and under-approximation $\lfloor\!\lfloor b \rfloor\!\rfloor$ for boolean expressions, with the guarantee that $\lceil\!\lceil b \rceil\!\rceil$ is always true when b is, and $\lfloor\!\lfloor b \rfloor\!\rfloor$ is only true when b is.

Definition 3.3.1. » For any comparison $e_1 \circ e_2$, where $\circ \in \{=, \neq, <, \leq, >, \geq\}$, we define its over-approximation as

$$\lceil\!\lceil e_1 \circ e_2 \rceil\!\rceil :\equiv \begin{cases} \texttt{true} & \text{if } e_1 \text{ or } e_2 \text{ contains any array lookup} \\ e_1 \circ e_2 & \text{otherwise.} \end{cases}$$

The under-approximation $\lfloor\!\lfloor e_1 \circ e_2 \rfloor\!\rfloor$ is defined completely analogously with the only difference that the true is replaced with false. For all remaining boolean expressions, the approximations are defined recursively; for example $\lceil\!\lceil b_1 \wedge b_2 \rceil\!\rceil :\equiv \lceil\!\lceil b_1 \rceil\!\rceil \wedge \lceil\!\lceil b_2 \rceil\!\rceil$ and $\lceil\!\lceil \neg b \rceil\!\rceil :\equiv \neg \lfloor\!\lfloor b \rfloor\!\rfloor$. «

Lemma 3.3.2. » *For all boolean expressions b, and all states σ, we have*

1. $[\![b]\!](\sigma) \Rightarrow [\![\lceil b \rceil]\!](\sigma)$ *and*
2. $[\![\lfloor b \rfloor]\!](\sigma) \Rightarrow [\![b]\!](\sigma)$. «

Proof. By straightforward induction on the structure of the boolean expression b. □

We then extend the notion of over-approximation and under-approximation to permission expressions. Here, we want that p is at most $\lceil p \rceil$ and $\lfloor p \rfloor$ is at most p independently of the program state. Again, these operators are defined recursively on the structure of the permission expression in a straightforward manner.

Definition 3.3.3. » For all permission expressions p, we define

$$\lceil p \rceil :\equiv \begin{cases} q & \text{if } p \equiv q \\ \lceil p_1 \rceil + \lceil p_2 \rceil & \text{if } p \equiv p_1 + p_2 \\ \lceil p_1 \rceil - \lfloor p_2 \rfloor & \text{if } p \equiv p_1 - p_2 \\ \mathsf{min}(\lceil p_1 \rceil, \lceil p_2 \rceil) & \text{if } p \equiv \mathsf{min}(p_1, p_2) \\ \mathsf{max}(\lceil p_1 \rceil, \lceil p_2 \rceil) & \text{if } p \equiv \mathsf{max}(p_1, p_2) \\ \mathsf{max}((\lceil b \rceil \,?\, \lceil p_1 \rceil : 0), (\lceil \neg b \rceil \,?\, \lceil p_2 \rceil : 0)) & \text{if } p \equiv b \,?\, p_1 : p_2 \end{cases}$$

The operator $\lfloor b \rfloor$ is defined analogously but with all over-approximations and under-approximations flipped. «

Example 3.3.4. » We have

$$\begin{aligned} \lceil \mathsf{a[i]} = 0 \,?\, \alpha_{\mathsf{a[i]}}(1) : 0 \rceil &\equiv \mathsf{max}\big(\big(\lceil \mathsf{a[i]} = 0 \rceil \,?\, \alpha_{\mathsf{a[i]}}(1) : 0\big), (\lceil \neg(\mathsf{a[i]} = 0) \rceil \,?\, 0 : 0)\big) \\ &\equiv \mathsf{max}\big(\big(\mathsf{true} \,?\, \alpha_{\mathsf{a[i]}}(1) : 0\big), (\neg \mathsf{false} \,?\, 0 : 0)\big), \end{aligned}$$

which simplifies to $\alpha_{\mathsf{a[i]}}(1)$. «

Lemma 3.3.5. » *For all permission expressions p and all states σ, we have*

1. $[\![p]\!](\sigma) \leq [\![\lceil p \rceil]\!](\sigma)$ *and*
2. $[\![\lfloor p \rfloor]\!](\sigma) \leq [\![p]\!](\sigma)$. «

Proof. By straightforward induction on the structure of the permission expression p.□

These approximations abstract away array-dependent conditions, and have an impact on the precision of our permission analysis only when array values are used to determine a location to be accessed. For example, a linear array search for a particular value accesses the array only up to the (a-priori unknown) point at which the value is found, but our inferred permission precondition – as formally defined below – conservatively requires permission to the full array.

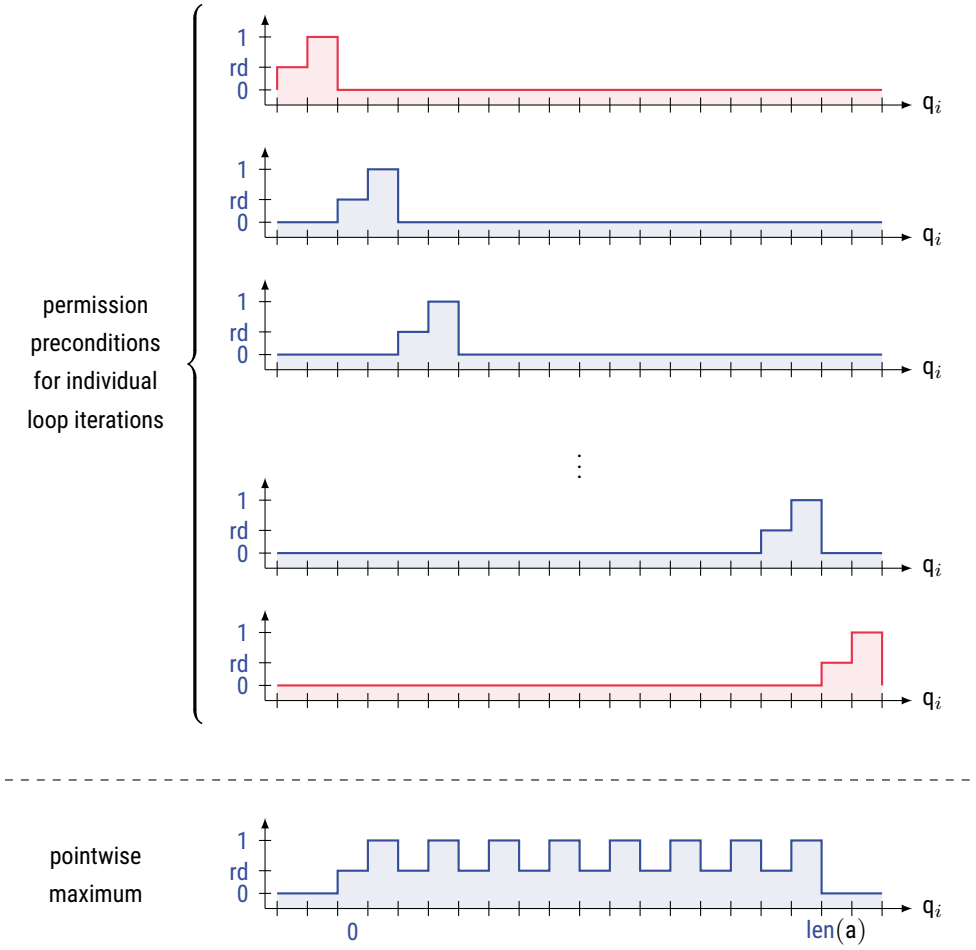


Figure 3.7. The top part of this figure shows the permission precondition of the loop body evaluated in the initial states of different loop iterations; the diagrams shown in red correspond to loop iterations not allowed by the loop invariant or loop condition. The pointwise maximum over all permitted loop iterations shown in the bottom part captures sufficient permissions to execute all iterations of the loop.

Numerical Loop Invariants. To obtain a sufficient permission precondition for the entire loop, we want to project the permission precondition for a single loop iteration over all possible initial states for the loop body executions. To this end, we leverage an *over-approximate* loop invariant I^+ from an off-the-shelf numerical analysis (for example [30]) to over-approximate all possible values of the numerical variables that get assigned in the loop. Such a loop invariant must satisfy the following properties:

1. The loop invariant must hold upon entry of the loop. That is, for all initial configurations γ of the program at hand and all states σ with $\gamma_0 \rightsquigarrow^* \langle l, \sigma \rangle$, we have $[\![I^+]\!](\sigma)$.
2. The loop invariant must be inductive. That is, for all states σ and σ' with $[\![I^+]\!](\sigma)$ and $\langle c, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$, we have $[\![I^+]\!](\sigma')$.

Exhale-Free Loop Bodies. Next, we describe how we generalise the permission precondition p for a loop body to a sufficient permission precondition for the entire loop. In a first step, we consider the simpler but common case of a loop that does not contain any $\mathsf{exhale}\ \mathsf{acc}(a[e], q)$ statements; that is, does not transfer permissions to a forked thread, for example. The solution for this case is also sound for loop bodies, where each $\mathsf{exhale}\ \mathsf{acc}(a[e], q)$ statement is followed by a matching $\mathsf{inhale}\ \mathsf{acc}(a[e], q')$, where $q \leq q'$, as used, for example, in the encoding of most method calls that do not leak permissions.

Lemma 3.3.6. » *For all loops $l \equiv \mathsf{while}\ (b)\ \{c\}$ with over-approximate numerical invariant I^+, where the body c is a loop-free and exhale-free statement that modifies the integer variables $\vec{x}$, the permission expression*

$$\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau[\![c]\!](0) \rceil\}$$

is a sufficient permission precondition for the loop l. «

We do not give a formal proof for this lemma, as it is subsumed by Lemma 3.3.8 below. Intuitively, the claim follows by proving that $\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau[\![c]\!](0) \rceil\}$ represents sufficient permissions to successfully execute any number n of unrollings of the loop while retaining at least $\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau[\![c]\!](0) \rceil\}$ permissions. The proof goes by induction on n. Since I^+ is a sound over-approximating loop invariant, we must have $[\![I^+ \wedge b]\!](\sigma)$ for any state σ at the beginning of a loop iteration. Moreover, since

$$[\![\tau[\![c]\!](0)]\!]_\Pi(\sigma) \sqsubseteq \left[\!\!\left[\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau[\![c]\!](0) \rceil\} \right]\!\!\right]_\Pi(\sigma),$$

we get from Lemma 3.2.22 that the execution of the first loop iteration cannot result in a permission failure. Furthermore, since c is exhale-free, we are guaranteed to

hold at least as many permissions after the execution of c as before, and so we can apply our induction hypothesis to conclude that there are also sufficient permissions for the remaining $n - 1$ loop iterations.

Example 3.3.7. » Let us revisit the loop of the copy_even_b method shown in Listing 3.3. Recall from Example 3.2.4 that the permission precondition inferred for the loop body is

$$p \equiv \mathsf{max}\big(\mathsf{max}\big(0, \alpha_{\mathsf{a}[2\cdot\mathsf{i}+1]}(1)\big), \alpha_{\mathsf{a}[2\cdot\mathsf{i}]}(\mathsf{rd})\big)$$

Further, we assume that we obtained the loop invariant $I^+ \equiv 0 \leq \mathsf{i}$ by running a precursory numerical analysis on the program. As illustrated by Figure 3.7, the permission precondition for the entire loop is then given by the pointwise maximum expression

$$\max_{\mathsf{i} \mid 0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a}) \,/\, 2} \{p\}.$$

Note that $\mathsf{i} < \mathsf{len}(\mathsf{a}) \,/\, 2$ comes from the loop condition and that $[\![p]\!] \equiv p$. Using our maximum elimination algorithm presented in Section 3.5, this pointwise maximum can be rewritten into $\mathsf{max}(p_1, p_2)$, where

$$p_1 \equiv (\mathsf{mod}(\mathsf{q}_i, 2) = 1 \wedge 1 < \mathsf{q}_i \wedge \mathsf{q}_i < 2 \cdot (\mathsf{len}(\mathsf{a}) - 1) \,/\, 2 + 1) \;?\; 1 : 0$$
$$p_2 \equiv (\mathsf{mod}(\mathsf{q}_i, 2) = 0 \wedge 0 < \mathsf{q}_i \wedge \mathsf{q}_i < 2 \cdot (\mathsf{len}(\mathsf{a}) - 1) \,/\, 2) \;?\; \mathsf{rd} : 0\,.$$

The permission expression p_1 captures write permission for all odd indices while p_2 captures read permission for all even indices; note that the divisibility constraints originate from the factor in front of the index in the array access $\mathsf{a}[2\cdot\mathsf{i}]$ and $\mathsf{a}[2\cdot\mathsf{i}+1]$. «

Soundness Condition. For loops that contain exhale statements, the approach described above does not always guarantee a sufficient permission precondition. For example, if a loop gives away full permission to the *same* array element in every iteration, our pointwise maximum construction yields a precondition requiring the full permission *once*, as opposed to the *unsatisfiable* precondition (since the loop is guaranteed to cause a permission failure).

As explained above, our inference is sound if each exhale statement is followed by a corresponding inhale, which can often be checked syntactically. In the following, we present another decidable condition that guarantees soundness and that can be checked efficiently by an SMT solver. If neither condition holds, we preserve soundness by inferring an unsatisfiable precondition; we did not encounter any such examples in our evaluation.

Put in words, our soundness condition requires that the maximum of permissions required by two (not necessarily subsequent) loop iterations is at least the permissions

required by executing these iterations in sequence. Intuitively, this condition holds whenever no iteration removes permissions that are required by another iteration. In order to formalise this soundness condition, let us define

$$f_{\vec{v}}(p) :\equiv \lceil \tau(\llbracket c \rrbracket(p)) \rceil [\vec{v} \setminus \vec{x}],$$

for all value vectors $\vec{v} \in \mathbb{Z}^n$ and permission expressions p. Roughly speaking $f_{\vec{v}}(p)$ captures the permissions required to successfully execute a loop iteration starting in an initial state where the variables $\vec{x}$ have values $\vec{v}$, while retaining p permissions in the end (cf. Lemma 3.2.22). Moreover, for the sake of shorter notation, we use $\vec{v}_\sigma$ to denote the values corresponding to the variables $\vec{x}$ in the state σ. That is, for all states σ, we define the vector $\vec{v}_\sigma := \langle v_1, \ldots, v_n \rangle$, where $v_i := \llbracket x_i \rrbracket(\sigma)$, for all $i \in \{1, \ldots, n\}$. We then use

$$V^+ := \{\vec{v} \in \mathbb{Z}^n \mid \exists \sigma \in \Sigma \colon \vec{v} = \vec{v}_\sigma \wedge \llbracket I^+ \wedge b \rrbracket(\sigma)\}$$

to denote the set containing all value vectors corresponding to all possible loop iterations (and possibly some spurious ones in case the loop invariant I^+ is imprecise).

Formally, our soundness condition requires that, independent of the state σ, we have

$$\forall \vec{v}_1, \vec{v}_2 \in V^+ \colon \vec{v}_1 \neq \vec{v}_2 \Rightarrow \llbracket f_{\vec{v}_1}(f_{\vec{v}_2}(0)) \rrbracket_\Pi(\sigma) \sqsubseteq \llbracket \mathsf{max}(f_{\vec{v}_1}(0), f_{\vec{v}_2}(0)) \rrbracket_\Pi(\sigma), \tag{3.1}$$

where the vectors $\vec{v}_1$ and $\vec{v}_2$ capture the initial state of the two *different* iterations.

Note that this soundness condition requires that no two different iterations of a loop observe exactly the same values for all integer variables. If that could be the case, the condition $\vec{v}_1 \neq \vec{v}_2$ would cause us to ignore a potential pair of initial states for two different loop iterations. To avoid this problem, we assume that all loops satisfy this requirement; it can easily be enforced by adding an additional variable as loop iteration counter [53].

Soundness Proof. We now formally prove that our inference produces a sufficient permission precondition whenever our soundness condition is met. We observe that we can use $f_{\vec{v}}(p)$ and V^+ defined above to express a pointwise maximum that is semantically equivalent to our pointwise maximum expression for the loop precondition: Concretely, we have

$$\left\llbracket \max_{\vec{v} \in V^+} \{f_{\vec{v}}(0)\} \right\rrbracket_\Pi(\sigma) = \left\llbracket \max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau(\llbracket c \rrbracket(0)) \rceil\} \right\rrbracket_\Pi(\sigma),$$

for all states σ. As we will see in the following, this alternative representation is well suited for our soundness proof, since there is a clear correspondence between the set of value vectors V^+ and the states encountered throughout an execution.

Lemma 3.3.8. » *Let us consider an arbitrary loop* $l \equiv \mathsf{while}\ (b)\ \{c\}$ *along with a numerical loop invariant* I^+*, where* c *is a loop-free statement that modifies the integer variables* $\vec{x}$*. If the soundness condition shown in Equation (3.1) holds, then the permission expression*

$$\max_{\vec{x}\,|\,I^+ \wedge b} \{\lceil \tau\{\!|c|\!\}(0)\rceil\}$$

is a sufficient permission precondition for the loop l*.* «

Proof. Let us consider an arbitrary loop $l \equiv \mathsf{while}\ (b)\ \{c\}$ along with a numerical loop invariant I^+ and suppose that the soundness condition Equation (3.1) holds. Moreover, for all states σ, we define

$$V_\sigma^+ := \{\vec{v} \in V^+ \mid \exists \sigma' \in \Sigma\colon \vec{v} = \vec{v}_{\sigma'} \wedge \langle l, \sigma\rangle \rightsquigarrow^* \langle l, \sigma'\rangle\}$$

to capture value vectors corresponding to initial states of loop iterations that are *reachable* from σ. Given some current state σ, this allows us to express the pointwise maximum $\max_{\vec{v}\in V_\sigma^+}\{f_{\vec{v}}(0)\}$ over all *future* loop iterations; as we will see below, this partitioning of V^+ in future and (implicit) past iterations is needed, since our soundness conditions only talks about *different* loop iterations.

To prove the statement of the lemma, it suffices to prove that, for all integers $k \in \mathbb{N}$ and all states σ with $[\![I^+]\!](\sigma)$ and $\sigma \vDash [\![\max_{\vec{v}\in V_\sigma^+}\{f_{\vec{v}}(0)\}]\!]_\Pi(\sigma)$, we have $\langle l, \sigma\rangle \not\rightsquigarrow^k \gamma_{\lightning}$. We do this by strong induction on the length k of the derivation sequence. To this end, let us fix an arbitrary $k \in \mathbb{N}$ and assume that the claim holds for all $k' \in \mathbb{N}$ with $k' < k$.

We consider an arbitrary state σ satisfying $[\![I^+]\!](\sigma)$ and $\sigma \vDash [\![\max_{\vec{v}\in V_\sigma^+}\{f_{\vec{v}}(0)\}]\!]_\Pi(\sigma)$ and aim to show that $\langle l, \sigma\rangle \not\rightsquigarrow^k \gamma_{\lightning}$. If $k = 0$, this trivially follows as $\langle l, \sigma\rangle \neq \gamma_{\lightning}$. Moreover, if $\neg[\![b]\!](\sigma)$, the claim also immediately follows since $\langle l, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle$ is the only possible derivation sequence. Therefore, for the remainder of the proof, we assume $k > 0$ and $[\![b]\!](\sigma)$. In this case, the first step of the derivation sequence has to be $\langle l, \sigma\rangle \rightsquigarrow \langle c\,; l, \sigma\rangle$. Thus, according to Lemma 2.3.9, we are left to show that

1. $\langle c, \sigma\rangle \not\rightsquigarrow^{k-1} \gamma_{\lightning}$ and
2. there is no state σ' such that $\langle c, \sigma\rangle \rightsquigarrow^{k_1} \langle \mathsf{skip}, \sigma'\rangle$ and $\langle l, \sigma'\rangle \rightsquigarrow^{k_2} \gamma_{\lightning}$, for some integers $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$.

Below, we prove both of these objectives separately. For the first objective, we observe that $\vec{v}_\sigma \in V_\sigma^+$ and, therefore, $f_{\vec{v}_\sigma}(0) \sqsubseteq \max_{\vec{v}\in V_\sigma^+}\{f_{\vec{v}}(0)\}$. Consequently, we have $\sigma \vDash [\![f_{\vec{v}_\sigma}(0)]\!]_\Pi(\sigma)$ and, therefore, also $\sigma \vDash [\![\tau\{\!|c|\!\}(0)]\!]_\Pi(\sigma)$. Thus, using Lemma 3.2.22, we can conclude that $\langle c, \sigma\rangle \not\rightsquigarrow \gamma_{\lightning}$, as required.

To prove the second objective, we consider two arbitrary integers $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$ and an arbitrary state σ' with $\langle c, \sigma \rangle \rightsquigarrow^{k_1} \langle \mathsf{skip}, \sigma' \rangle$, and aim to show that $\langle l, \sigma' \rangle \not\rightsquigarrow^{k_2} \gamma_{\lightning}$. Let us consider an arbitrary value vector $\vec{w} \in V^+_{\sigma'}$. We observe that $\vec{w} \neq \vec{v}$, since we enforce the initial state of all loop iterations to be different and, therefore, $\vec{v} \notin V^+_{\sigma'}$. We have

$$[\![f_{\vec{v}_\sigma}(f_{\vec{w}}(0))]\!]_\Pi(\sigma) \sqsubseteq [\![\mathsf{max}(f_{\vec{v}_\sigma}(0), f_{\vec{w}}(0))]\!]_\Pi(\sigma) \sqsubseteq \left[\!\!\left[\max_{\vec{v} \in V^+_\sigma} \{ f_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma),$$

where the first inequality holds due to the assumption of our soundness condition Equation (3.1) and the second inequality is a consequence of $\{\vec{v}_\sigma, \vec{w}\} \subseteq V^+_\sigma$. Similar to the previous case, this lets us infer that $\sigma \vDash [\![\tau\{\!| c |\!\}(f_{\vec{w}}(0))]\!]_\Pi(\sigma)$. We then apply Lemma 3.2.22 to get $\sigma' \vDash [\![f_{\vec{w}}(0)]\!]_\Pi(\sigma')$. Moreover, since the value vector $\vec{w} \in V^+_{\sigma'}$ was chosen arbitrarily, $\sigma' \vDash [\![f_{\vec{v}}(0)]\!]_\Pi(\sigma')$ must be true for *all* such vectors $\vec{v} \in V^+_{\sigma'}$. Thus, we have

$$\sigma' \vDash \left[\!\!\left[\max_{\vec{w} \in V^+_{\sigma'}} \{ f_{\vec{w}}(0) \} \right]\!\!\right]_\Pi(\sigma')$$

Finally, by our induction hypothesis, we get $\langle l, \sigma' \rangle \not\rightsquigarrow^{k_2} \gamma_{\lightning}$, which concludes the proof. □

3.3.2 Permission Differences for Isolated Loops

In order to extend our permission analysis to arbitrary programs, we need to account for the permissions lost and gained throughout the execution of a loop.

Lost Permissions. We over-approximate the permissions lost by the execution of the loop $\mathsf{while}\ (b)\ \{c\}$ with over-approximate loop invariant I^+ by

$$d^- :\equiv \min_{\vec{x} \mid I^+ \wedge b} \{ d_0^- \}, \quad \text{where } d_0^- :\equiv \mathsf{min}(\lfloor\!\lfloor \delta\{\!| c |\!\}(0) \rfloor\!\rfloor, 0). \tag{3.2}$$

Note that the over-approximation is in terms of the absolute value of d^-; the use of the minimum and under-approximation is due to losses being expressed by negative permission expressions.

Gained Permissions. Dually, we under-approximate the permissions gained by the execution of the loop as

$$d^+ :\equiv \max_{\vec{x} \mid I^- \wedge b} \{ d_0^+ \}, \quad \text{where } d_0^+ :\equiv \mathsf{max}(\lfloor\!\lfloor \delta\{\!| c |\!\}(0) \rfloor\!\rfloor, 0), \tag{3.3}$$

where I^- is an under-approximate loop invariant.

Soundness Condition. Analogous to our loop preconditions, the pointwise minimum in our permission expression d^- capturing the lost permissions is only sound if no two different loop iterations exhale permissions for the same array element. In contrast, the pointwise maximum in our permission expression d^+ capturing the gained permissions is – as we will formally prove below – always sound since the maximum conservatively under-approximates the gained permissions in cases where two different loop inhale permissions for the same array element.

Below, we will formulate the precise condition under which our permission differences for loops are sound. To this end, for all permission expressions p and all value vectors $\vec{v} \in \mathbb{Z}^n$, we define

$$g_{\vec{v}}(p) :\equiv \lfloor\!\lfloor \delta\{\!|c|\!\}(p) \rfloor\!\rfloor[\vec{v} \setminus \vec{x}],$$

as well as

$$g^-_{\vec{v}}(p) :\equiv \mathsf{min}(g_{\vec{v}}(p), 0)$$
$$g^+_{\vec{v}}(p) :\equiv \mathsf{max}(g_{\vec{v}}(p), 0).$$

Roughly speaking, $g_{\vec{v}}(p)$ captures the permission differences corresponding to a loop iteration starting in an initial state where the variables $\vec{x}$ have values $\vec{v}$, while $g^-_{\vec{v}}(p)$ and $g^+_{\vec{v}}(p)$ capture the lost permissions and gained permissions, respectively.

Put in words, our soundness condition for permission differences requires that the minimum of the permission differences corresponding to two (not necessarily subsequent) loop iterations is at most permission difference corresponding to the execution of these iterations in sequence. Formally, we express this soundness condition as

$$\forall \vec{v}_1, \vec{v}_2 \in V^+ \colon \vec{v}_1 \neq \vec{v}_2 \Rightarrow [\![\mathsf{min}(g^-_{\vec{v}_1}(0), g^-_{\vec{v}_2}(0))]\!]_\Pi(\sigma) \sqsubseteq [\![g^-_{\vec{v}_1}(g^-_{\vec{v}_2}(0))]\!]_\Pi(\sigma), \tag{3.4}$$

independent of the state σ. Note that, just like our soundness condition for loop preconditions, this condition also requires that no two different iterations of a loop observe exactly the same values for the integers variables.

Moreover – at least intuitively – it is not hard to see that our soundness condition for permission differences is *weaker* than the one for loop preconditions: The condition in Equation (3.4) holds whenever no iteration removes permissions that are removed (as opposed to required) by another iteration. Even syntactically, the two conditions are very similar; note that the inequalities are flipped and one has a maximum while the other has a minimum because, roughly speaking, the all leaf expressions $\alpha_{a[e]}(q)$ added by the precondition operator τ are removed by the difference operator δ, and vice versa (cf. Definition 3.2.3 and Definition 3.2.23). A formal proof of this claim, however, is not

straightforward as the two operators δ and τ are defined recursively over the structure of statements while the soundness condition might hold for a statement, while being violated by some of its sub-statements; for instance exhale acc(a[i]) ; inhale acc(a[i]). An example of a loop body that satisfies the soundness condition for permission differences but not the one for loop preconditions is a[0] := 0 ; exhale acc(a[i]), assuming i = 0 is allowed by the loop invariant.

Conjecture 3.3.9. » The soundness condition for our permission differences shown in Equation (3.4) above is implied by the soundness condition for our permission preconditions as defined by Equation (3.1). «

Of course, instead of relying on the conjecture above, we can alternatively also directly verify the soundness of our permission differences by explicitly checking the condition shown in Equation (3.4) using an SMT solver.

Soundness Proof. Before we dive into the soundness proof, let us first state the following auxiliary lemma

Lemma 3.3.10. » *For all loop-free statements c and all permission expressions p and p', where p does not contain any variables assigned by c and neither depends on array variables, we have*

$$[\![\delta\{\!|c|\!\}(p + p')]\!]_\Pi(\sigma) = [\![p + \delta\{\!|c|\!\}(p')]\!]_\Pi(\sigma),$$

for all states σ. «

Proof. The proof goes by straightforward induction on the structure of the loop-free statement c. □

For our upcoming elaborations – just like for the soundness proof for loop preconditions – we will employ pointwise maximum expressions that range over sets of value vectors. To this end, we define

$$g^*_{\langle S^+,S^-\rangle}(p) :\equiv \min_{\vec{v}\in S^+} \{g^-_{\vec{v}}(p)\} + \max_{\vec{v}\in S^-} \{g^+_{\vec{v}}(p)\},$$

for all pairs of sets of $S^+, S^- \subseteq \mathbb{Z}^n$ of value vectors and all permission expressions p. We then observe that, for all states σ, we have

$$[\![g^*_{\langle V^+,V^-\rangle}(0)]\!]_\Pi(\sigma) = [\![d^- + d^+]\!]_\Pi(\sigma),$$

where V^- is defined just as V^+ but using the under-approximate loop invariant I^- instead of I^+.

The soundness proof for our permission differences for loops is similar to the one for the loop preconditions; however, it is slightly more involved as it needs to differentiate between the permissions that are lost and the ones that are gained. First, we prove the following lemma that, intuitively, lets us bound the permissions lost and gained by a single loop iteration.

Lemma 3.3.11. » *Consider a value vector $\vec{w} \in V^+$ and two pairs of sets $A^+, B^+ \subseteq V^+$ and $A^-, B^- \subseteq V^-$ with $\vec{w} \in B^+$ and $A^+ \subseteq B^+ \setminus \{\vec{w}\}$ as well as $A^- = B^-$ or $A^- \cup \{\vec{w}\} = B^-$; that is, the pairs of sets that differ by at least $\vec{w}$ and by at most $\vec{w}$, respectively. Assuming the soundness condition given by Equation (3.4), for all pairs of states σ and σ' that differ only in the variables $\vec{x}$, we have*

$$[\![g^*_{\langle B^+, B^- \rangle}(0)]\!]_\Pi(\sigma') - [\![g^*_{\langle A^+, A^- \rangle}(0)]\!]_\Pi(\sigma) \sqsubseteq [\![g_{\vec{w}}(0)]\!]_\Pi(\sigma). \quad «$$

Proof. Let us consider some value vector $\vec{w} \in V^+$ and two pairs of sets $A^+, B^+ \subseteq V^+$ and $A^-, B^- \subseteq V^+$ the properties stated in the lemma. Moreover, let σ and σ' be arbitrary states that differ only in the variables $\vec{x}$. First, we prove an inequality that bounds the lost permissions. We observe that

$$\begin{aligned}
\left[\!\!\left[\min_{\vec{v} \in B^+} \{ g^-_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma') &= \left[\!\!\left[\min_{\vec{v} \in B^+} \{ g^-_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma) && (3.5)\\
&\sqsubseteq \left[\!\!\left[\min_{\vec{v} \in A^+} \{ \min(g^-_{\vec{w}}(0), g^-_{\vec{v}}(0)) \} \right]\!\!\right]_\Pi(\sigma) && (3.6)\\
&\sqsubseteq \left[\!\!\left[\min_{\vec{v} \in A^+} \{ g^-_{\vec{v}}(g^-_{\vec{w}}(0)) \} \right]\!\!\right]_\Pi(\sigma) && (3.7)\\
&= \left[\!\!\left[\min_{\vec{v} \in A^+} \{ g^-_{\vec{w}}(0) + g^-_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma) && (3.8)\\
&= \left[\!\!\left[g^-_{\vec{w}}(0) + \min_{\vec{v} \in A^+} \{ g^-_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma). && (3.9)
\end{aligned}$$

For the equality in the step (3.5) above, we used that $g^-_{\vec{v}}(0)$, by construction, does not depend on the variables $\vec{x}$. The step (3.6) follows from $\vec{w} \in B^+$ and $A^+ \subseteq B^+ \setminus \{\vec{w}\}$. The inequality (3.7) is due to our soundness condition; note that $\vec{w} \notin A^+ \subseteq B^+ \setminus \{\vec{w}\}$ and, therefore, $\vec{v} \neq \vec{w}$, for all $\vec{v} \in A^+$. The equality (3.8) then makes use of Lemma 3.3.10, which is applicable since $g^-_{\vec{w}}(0)$ neither depends on the variables $\vec{x}$ nor on any array values. And the final step (3.9) holds because $g^-_{\vec{w}}(0)$ does not depend on $\vec{v}$.

Next, we prove an inequality that bounds the gained permissions. We have

$$\begin{aligned}
\left[\!\!\left[\max_{\vec{v} \in B^-} \{ g^+_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma') &= \left[\!\!\left[\max_{\vec{v} \in B^-} \{ g^+_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma)\\
&\sqsubseteq \left[\!\!\left[g^+_{\vec{w}}(0) + \max_{\vec{v} \in A^-} \{ g^+_{\vec{v}}(0) \} \right]\!\!\right]_\Pi(\sigma),
\end{aligned}$$

where the first step follows analogous to (3.5) and the inequality in the second step is due to the sets A^- and B^- differing by at most $\vec{w}$ and the non-negativity of $g^+_{\vec{w}}(0)$.

Combining our findings from above then yields

$$\begin{aligned}
[\![g^*_{\langle B^+,B^-\rangle}(0)]\!]_\Pi(\sigma') &= \left[\!\!\left[\min_{\vec{v}\in B^+} \left\{g^-_{\vec{v}}(0)\right\} + \max_{\vec{v}\in B^-} \left\{g^+_{\vec{v}}(0)\right\} \right]\!\!\right]_\Pi(\sigma') \\
&\sqsubseteq \left[\!\!\left[g^-_{\vec{w}}(0) + g^+_{\vec{w}}(0) + \min_{\vec{v}\in A^+} \left\{g^-_{\vec{v}}(0)\right\} + \max_{\vec{v}\in A^-} \left\{g^+_{\vec{v}}(0)\right\} \right]\!\!\right]_\Pi(\sigma) \\
&\sqsubseteq [\![g_{\vec{w}}(0)]\!]_\Pi(\sigma) + [\![g^*_{\langle A^+,A^-\rangle}(0)]\!]_\Pi(\sigma'),
\end{aligned}$$

from which the claim immediately follows. □

Next, we aim to bound the permissions lost and gained by the first couple of – but arbitrarily many – loop iterations. Intuitively, our strategy is to keep track of the value vectors corresponding to *past loop iterations*; this will enable us to establish the conditions on the sets of value vectors required to apply Lemma 3.3.11. To facilitate this, for all states σ and $\pm \in \{+,-\}$, we define the set

$$W^\pm_\sigma \coloneqq \{\vec{v} \in V^\pm \mid \exists \sigma' \in \Sigma\colon \sigma' \neq \sigma \wedge \vec{v} = \vec{v}_{\sigma'} \wedge \langle l,\sigma'\rangle \rightsquigarrow^* \langle l,\sigma\rangle\}$$

that captures the subset of V^+ and V^-, respectively, corresponding to states at the beginning of loop iterations from which the state σ is reachable.

Lemma 3.3.12. » *For all integers $k \in \mathbb{N}$, and all states $\sigma = \langle s,h,\pi\rangle$ and $\sigma' = \langle s',h',\pi'\rangle$ with $\langle l,\sigma\rangle \rightsquigarrow^k \langle l,\sigma'\rangle$, we have*

$$\left[\!\!\left[g^*_{\langle W^+_{\sigma'},W^-_{\sigma'}\rangle}(0) \right]\!\!\right]_\Pi(\sigma) \sqsubseteq \pi' - \pi,$$

assuming that σ is an entry state of the loop. «

Proof. First, we assume that our soundness condition from Equation (3.4) holds. For the sake of shorter notation, for all states σ, we define

$$G(\sigma) \coloneqq \left[\!\!\left[g^*_{\langle W^+_\sigma,W^-_\sigma\rangle}(0) \right]\!\!\right]_\Pi(\sigma).$$

Our proof goes by strong induction on the length k of the derivation sequence. That is, we aim to prove the claim for an arbitrary $k \in \mathbb{N}$, while assuming its truth for all k' with $k' < k$. To do so, let us consider two arbitrary states $\sigma = \langle s,h,\pi\rangle$ and $\sigma' = \langle s',h',\pi'\rangle$ with $[\![I^+]\!](\sigma)$ and $\langle l,\sigma\rangle \rightsquigarrow^k \langle l,\sigma'\rangle$.

For the case where $k = 0$, we observe that $\langle l,\sigma\rangle \rightsquigarrow^0 \langle l,\sigma'\rangle$ implies $\sigma = \sigma'$ and, by extension, also $\pi = \pi'$. As every sound under-approximation of the set of loop iterations laying in the past must be empty at the entry of the loop, we have $W^-_\sigma = \varnothing$; this means that $G(\sigma)$ potentially captures some spurious lost permissions,

but certainly no gained permissions. Thus, we have $G(\sigma) \sqsubseteq \pi_{\mathsf{zero}} = \pi' - \pi$, which concludes this case.

We now consider the case where $k > 0$. In this case, we can use Lemma 2.3.10 to split off the last loop iteration to obtain the two derivation sequences $\langle l, \sigma\rangle \rightsquigarrow^{k_1} \langle l, \sigma''\rangle$ and $\langle c, \sigma''\rangle \rightsquigarrow^{k_2} \langle \mathsf{skip}, \sigma'\rangle$, for some state $\sigma'' = \langle s'', h'', \pi''\rangle$ and integers $k_1, k_2 \in \mathbb{N}$ with $k_1 + k_2 + 1 = k$. Below, we bound the permission differences for both of these sequences, separately.

First, we observe that $W^+_{\sigma''} \subseteq W^+_{\sigma'}$ and $W^-_{\sigma''} \subseteq W^-_{\sigma'}$, as the set of states that lie in the past can only grow across the execution of a loop iteration. Note that, by definition, we have $\vec{v}_{\sigma''} \notin W^+_{\sigma''}$, and $\vec{v}_{\sigma''} \in W^+_{\sigma'}$, since σ' is reachable from σ''; that is, the sets $W^+_{\sigma''}$ and $W^+_{\sigma'}$ differ by at least $\vec{v}_{\sigma''}$. Moreover, due to their under-approximate nature, the sets $W^-_{\sigma''}$ and $W^+_{\sigma'}$ can differ by at most $\vec{v}_{\sigma''}$. Thus, we can apply Lemma 3.3.11 to obtain

$$G(\sigma') - G(\sigma'') \sqsubseteq [\![g_{\vec{v}_{\sigma''}}(0)]\!]_\Pi(\sigma) \sqsubseteq [\![\delta\{\!|c|\!\}(0)]\!]_\Pi(\sigma'') \sqsubseteq \pi' - \pi'', \tag{3.10}$$

where the second inequality is due to the definition of $g_{\vec{v}_{\sigma''}}(0)$, and the last one follows from Lemma 3.2.24.

Next, we apply the induction hypothesis to k_1, instantiated the derivation sequence $\langle l, \sigma\rangle \rightsquigarrow^{k_1} \langle l, \sigma''\rangle$, which gives us

$$G(\sigma'') \sqsubseteq \pi'' - \pi. \tag{3.11}$$

With this, we are ready to conclude that

$$\begin{aligned}
[\![g^*_{\langle W^+_{\sigma'}, W^-_{\sigma'}\rangle}(0)]\!]_\Pi(\sigma) &= [\![g^*_{\langle W^+_{\sigma'}, W^-_{\sigma'}\rangle}(0)]\!]_\Pi(\sigma') && (3.12)\\
&= G(\sigma') \\
&= (G(\sigma') - G(\sigma'')) + G(\sigma'') \\
&\sqsubseteq (\pi' - \pi'') + (\pi'' - \pi) && (3.13)\\
&= \pi' - \pi,
\end{aligned}$$

as required. Above, the first step (3.12) holds since $g^*_{\langle W^+_{\sigma'}, W^-_{\sigma'}\rangle}(0)$ is independent of the heap and any variable modified by c and the inequality in step (3.13) follows from Equation (3.10) and Equation (3.11). □

Next, we prove a final lemma that bridges the gap between the permissions lost and gained by arbitrarily many loop iterations and the permission differences caused by the execution of an entire loop.

Lemma 3.3.13. » *Let us consider an arbitrary loop* $l \equiv \mathsf{while}\ (b)\ \{c\}$ *along with over- and under-approximate loop invariants* I^+ *and* I^- *and suppose that the soundness*

condition in Equation (3.4) holds. For all states $\sigma = \langle s, h, \pi \rangle$ *and* $\sigma' = \langle s', h, \pi' \rangle$ *with* $\langle l, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$*, we have*

$$\left[\!\!\left[\min_{\vec{x} \mid I^+ \wedge b} \{\min(\lfloor\!\lfloor \delta\{\!|c|\!\}(0) \rfloor\!\rfloor, 0)\} - \max_{\vec{x} \mid I^- \wedge b} \{\max(\lfloor\!\lfloor \delta\{\!|c|\!\}(0) \rfloor\!\rfloor, 0)\} \right]\!\!\right]_\Pi(\sigma) \sqsubseteq \pi' - \pi,$$

assuming that σ *is an entry state of the loop.* «

Proof. Let us consider an arbitrary loop $l \equiv \mathsf{while}\ (b)\ \{c\}$ along with over- and under-approximate loop invariants I^+ and I^- and suppose that the soundness condition in Equation (3.4) holds. Moreover, we consider two arbitrary states $\sigma = \langle s, h, \pi \rangle$ and $\sigma' = \langle s', h, \pi' \rangle$, where σ is an entry state of the loop and $\langle l, \sigma \rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma' \rangle$. First, we observe that omitting the last step of this derivation sequence yields $\langle l, \sigma \rangle \rightsquigarrow^* \langle l, \sigma' \rangle$ since the last step of the full execution of the loop must be $\langle l, \sigma' \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma' \rangle$; note that $[\![b]\!](\sigma')$, as the loop terminated in the state σ'. We observe that

$$[\![d^- + d^+]\!]_\Pi(\sigma) = \left[\!\!\left[g^*_{\langle V^+, V^- \rangle}(0) \right]\!\!\right]_\Pi(\sigma) \sqsubseteq \left[\!\!\left[g^*_{\langle W^+_{\sigma'}, W^-_{\sigma'} \rangle}(0) \right]\!\!\right]_\Pi(\sigma) \sqsubseteq \pi' - \pi,$$

where the second step follows from $W^+_{\sigma'} \subseteq V^+$ and $W^-_{\sigma'} = V^-$ and the last step is justified by Lemma 3.3.12. □

Above, we have seen how to construct a permission expression capturing the permissions lost and gained by the execution of an individual loop. In the following, we will see how we can combine this with the permission preconditions for individual loops from Section 3.3.1 above to obtain permission precondition and postconditions for statements containing arbitrarily nested loops.

3.3.3 Permission Analysis for Loop Programs

We now use our findings from Section 3.3.1 and Section 3.3.2 above to extend our permission analysis to statements containing arbitrarily nested loops.

Rules for Loops. We complement our precondition and difference operators for loop free code defined in Section 3.2.2 and Section 3.2.3 with the following rules for loops.

Definition 3.3.14. » For all loops $\mathsf{while}\ (b)\ \{c\}$ with over- and under-approximate numerical invariants I^+ and I^-, respectively, and all permission expressions p, we define

$$\tau\{\!|\mathsf{while}\ (b)\ \{c\}|\!\}(p) :\equiv b\ ?\ \max\left(\overbrace{\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau\{\!|c|\!\}(0) \rceil\}}^{p_1}, \overbrace{\max_{\vec{x} \mid I^+ \wedge \neg b} \{\lceil p \rceil\}}^{p_2} - d \right) : p$$

$$\delta\{\!|\mathsf{while}\ (b)\ \{c\}|\!\}(p) :\equiv b\ ?\ \Bigg(\underbrace{\min_{\vec{x} \mid I^+ \wedge \neg b} \{\min(\lfloor\!\lfloor p \rfloor\!\rfloor, 0)\}}_{p^-} + \underbrace{\max_{\vec{x} \mid I^- \wedge \neg b} \{\max(\lfloor\!\lfloor p \rfloor\!\rfloor, 0)\}}_{p^+} + d \Bigg) : p,$$

where

$$d :\equiv \underbrace{\min_{\vec{x} \mid I^+ \wedge b} \{\min(\lfloor\!\lfloor \delta\{\!\{c\}\!\}(0) \rfloor\!\rfloor, 0)\}}_{d^-} + \underbrace{\max_{\vec{x} \mid I^- \wedge b} \{\max(\lfloor\!\lfloor \delta\{\!\{c\}\!\}(0) \rfloor\!\rfloor, 0)\}}_{d^+} . \quad «$$

In the definition above, the permission expression p_1, as elaborated in Section 3.3.1, provides sufficient permissions to execute the loop. The expression p_2 conservatively over-approximates the permissions required to execute the code after the loop. The role of the expressions p^- and p^+ in the difference operator is to carry over the permissions p that are lost or gained by the code following the loop. Moreover, we note that the difference d is equal to the sum $d^- + d^+$ of the permissions lost and gained by the loop as defined in Equation (3.2) and Equation (3.3).

By construction, our permission preconditions and differences depend on numerical invariants that come from a precursory analysis of the given input program. As a result, our preconditions and differences for a statement c that is part of this input program are only sound for execution that start in an initial state allowed by the program. If we, say, cut out the statement c and embed it in a different program, the preconditions and differences may become invalid as – in this new context – any loops appearing within c may encounter different loop iterations. Therefore, in the following, we always implicitly assume that the statement at hand is embedded in its original program. Moreover, the permission differences for loops are only sound for *complete* executions of the loop. This is why we restrict all our claims below to derivation sequences starting from so-called *entry configurations*; intuitively, encountering an entry configuration means that it is the first time – ignoring previous iterations of loops at lower nesting levels – the execution reaches the corresponding point in the program.

Soundness of Differences. As a first step towards proving the correctness of our rules for loops, we prove that our difference operator $\delta\{\!\{\mathsf{while}\ (b)\ \{c\}\}\!\}(p)$ for loops is sound.

Lemma 3.3.15. » *For all statements c, all pairs of permission expressions p and p' with $p \equiv \delta\{\!\{l\}\!\}(p')$ and all pairs of states $\sigma = \langle s, h, \pi\rangle$ and $\sigma' = \langle s', h', \pi'\rangle$ with $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, we have*

$$[\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \sqsubseteq \pi' - \pi,$$

assuming that $\langle c, \sigma\rangle$ is an entry configuration and that the soundness condition in Equation (3.4) holds. «

Proof. The proof of Lemma 3.2.24 already proves the claim for all loop-free statements c. Here, we prove the claim for an arbitrary loop $l \equiv \mathsf{while}\ (b)\ \{c\}$, where c

itself is a loop-free statement. The claim can then be easily extended to arbitrary statements with arbitrarily nested loops by induction on the maximal depth of nested loops.

For the remainder of this proof, let us consider two permission expressions p and p' with $p \equiv \delta(\!|\, l \,|\!)(p')$ as well as two states σ and σ' with $\langle l, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, where $\langle l, \sigma\rangle$ is an entry configuration. Furthermore, suppose that the soundness condition in Equation (3.4) holds.

First, let us consider the case where $[\![b]\!](\sigma)$. We define $p_0^- :\equiv \mathsf{min}(\lfloor\!\lfloor p' \rfloor\!\rfloor, 0)$ as well as $p^- :\equiv \mathsf{min}_{\vec{x}\,|\,I^+\wedge\neg b}\{p_0^-\}$ and observe that

$$[\![p^-]\!]_\Pi(\sigma) \sqsubseteq [\![p_0^-[\vec{v}_{\sigma'}\backslash\vec{x}]]\!]_\Pi(\sigma') \sqsubseteq [\![p_0^-]\!]_\Pi(\sigma'), \tag{3.14}$$

where the first inequality holds since $[\![I^+ \wedge b]\!](\sigma')$; note that σ' is the state in which the loop terminates. Similarly, we also define $p_0^+ :\equiv \mathsf{max}(\lfloor\!\lfloor p' \rfloor\!\rfloor, 0)$ as well as $p^+ :\equiv \mathsf{max}_{\vec{x}\,|\,I^-\wedge\neg b}\{p_0^+\}$ and observe that

$$[\![p^+]\!]_\Pi(\sigma) \sqsubseteq [\![p_0^+[\vec{v}_{\sigma'}\backslash\vec{x}]]\!]_\Pi(\sigma') \sqsubseteq [\![p_0^+]\!]_\Pi(\sigma'), \tag{3.15}$$

where the first inequality holds since the condition $I^- \wedge \neg b$ is always false or captures exactly the final state σ'. Combining Equation (3.14) and Equation (3.15) then yields

$$[\![p^- + p^+]\!]_\Pi(\sigma) \sqsubseteq [\![\mathsf{min}(\lfloor\!\lfloor p' \rfloor\!\rfloor, 0) + \mathsf{max}(\lfloor\!\lfloor p' \rfloor\!\rfloor, 0)]\!]_\Pi(\sigma') \sqsubseteq [\![p']\!]_\Pi(\sigma'). \tag{3.16}$$

With this, we are ready to conclude that

$$\begin{aligned}
[\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') &= [\![\delta(\!|\, l \,|\!)(p')]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&= [\![b\,?\,(p^- + p^+ + d) : p']\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&= [\![p^- + p^+ + d]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&= [\![p^- + p^+]\!]_\Pi(\sigma) + [\![d]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') \\
&\sqsubseteq [\![d]\!]_\Pi(\sigma) \\
&\sqsubseteq \pi' - \pi,
\end{aligned}$$

where the penultimate step follows from Equation (3.16) and the last step is justified by Lemma 3.3.13; note that $d \equiv d^- + d^+$, as used there.

Next, we consider the case where $\neg[\![b]\!](\sigma)$. In this case, we have $\langle l, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma\rangle$ and, therefore, also $\sigma = \sigma'$ and $\pi = \pi'$. Moreover, we observe that

$$\begin{aligned}
[\![p]\!]_\Pi(\sigma) &= [\![\delta(\!|\, l \,|\!)(p')]\!]_\Pi(\sigma) \\
&= [\![b\,?\,(p^- + p^+ + d) : p']\!]_\Pi(\sigma) \\
&= [\![p']\!]_\Pi(\sigma) = [\![p']\!]_\Pi(\sigma').
\end{aligned}$$

Thus, we can conclude $[\![p]\!]_\Pi(\sigma) - [\![p']\!]_\Pi(\sigma') = \pi_{\mathsf{zero}} = \pi' - \pi$, as required. □

Soundness of Preconditions. Next, we prove the soundness of our precondition operator $\tau\{\!|\mathsf{while}\ (b)\ \{c\}|\!\}(p)$ for loops. As for the loop-free counterpart of this proof, we first show that rule appropriately handles the permission expression p acting as an accumulator. After that, we then show that our operator indeed produces a sufficient permission precondition.

Lemma 3.3.16. » *For all statements c, all pairs of permission expressions p and p' with $p \equiv \tau\{\!|c|\!\}(p')$, and all pairs of states σ and σ' with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, we have $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$, assuming the soundness conditions from Equation (3.1) and Equation (3.4) hold for all loops in c.* «

Proof. The proof of Lemma 3.2.20 already proves the claim for all loop-free statements. Here, we prove the claim for an arbitrary loop $l \equiv \mathsf{while}\ (b)\ \{c\}$, where c itself is a loop-free statement. The original claim then follows by induction on the maximal depth of nested loops.

Let us consider two permission expressions p and p' with $p \equiv \tau\{\!|c|\!\}(p')$, and some states σ and σ' with $\sigma \vDash [\![p]\!]_\Pi(\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. Throughout this proof, we will use the shorthands $p_1 :\equiv \mathsf{max}_{\vec{x}\,|\,I^+\wedge b}\{\lceil\tau\{\!|c|\!\}(0)\rceil\}$ and $p_2 :\equiv \mathsf{max}_{\vec{x}\,|\,I^+\wedge\neg b}\{\lceil p'\rceil\}$.

First, we consider the case where $\neg[\![b]\!](\sigma)$. In this case, we have $\langle l, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma\rangle$ and, therefore, $\sigma = \sigma'$ and $\pi = \pi'$. Moreover, we observe that

$$[\![p]\!]_\Pi(\sigma) = [\![\tau\{\!|l|\!\}(p')]\!]_\Pi(\sigma) = [\![b\,?\,\mathsf{max}(p_1, p_2 - d) : p']\!]_\Pi(\sigma) = [\![p']\!]_\Pi(\sigma)$$

from which the claim immediately follows for this case.

Next, we turn our attention to the case where $[\![b]\!](\sigma)$. We observe that

$$[\![p_2 - d]\!]_\Pi(\sigma) \sqsubseteq [\![b\,?\,\mathsf{max}(p_1, p_2 - d) : p']\!]_\Pi(\sigma) = [\![\tau\{\!|l|\!\}(p')]\!]_\Pi(\sigma) \sqsubseteq \pi, \tag{3.17}$$

where the last inequality holds due to $\sigma \vDash [\![p]\!]_\Pi(\sigma)$. Moreover, we have

$$\begin{aligned}
[\![p']\!]_\Pi(\sigma') &= [\![\lceil p'\rceil[\vec{v}_{\sigma'}\backslash\vec{x}]]\!]_\Pi(\sigma) && (3.18)\\
&\sqsubseteq \left[\!\!\left[\mathsf{max}_{\vec{x}\,|\,I^+\wedge\neg b}\{\lceil p'\rceil\}\right]\!\!\right]_\Pi(\sigma) = [\![p_2]\!]_\Pi(\sigma) && (3.19)\\
&= [\![p_2 - d]\!]_\Pi(\sigma) + [\![d]\!]_\Pi(\sigma)\\
&\sqsubseteq \pi + (\pi' - \pi) = \pi && (3.20)
\end{aligned}$$

and, therefore, $\sigma' \vDash [\![p']\!]_\Pi(\sigma')$, as required. Above, the first step (3.18) exploits that the states σ and σ' differ only in the variables $\vec{x}$ and the heap, but $\lceil p'\rceil[\vec{v}_{\sigma'}\backslash\vec{x}]$ is independent of $\vec{x}$ and array values. The inequality in step (3.19) holds since $[\![I^+ \wedge \neg b]\!](\sigma')$. Finally, the last inequality in (3.20) is justified by Equation (3.17) and Lemma 3.3.13. □

Lemma 3.3.17. » *For all statements c, all non-negative permission expressions p, and all states σ with $\sigma \vDash \tau\{\!|c|\!\}(p)$, we have $\langle c, \sigma\rangle \not\rightsquigarrow^* \gamma_{\lightning}$, assuming the soundness conditions from Equation (3.1) and Equation (3.4) hold for all loops in c.* «

Proof. The proof of Lemma 3.2.22 already proves the claim for all loop-free statements. Here, we prove the claim for an arbitrary loop $l \equiv \mathsf{while}\ (b)\ \{c\}$, where c itself is a loop-free statement. The claim then easily extends to arbitrary statements by induction on the maximal depth of nested loops.

Let us consider an arbitrary permission expressions p, and two arbitrary states σ with $\sigma \vDash \tau\{\!|l|\!\}(p)$. In this case where $\neg[\![b]\!](\sigma)$, the claim trivially follows since $\langle l, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma\rangle$ is the only possible derivation sequence. For the case where $[\![b]\!](\sigma)$, we observe that

$$[\![\tau\{\!|l|\!\}(p)]\!]_\Pi(\sigma) \sqsubseteq \left[\!\!\left[\max_{\vec{x} \mid I^+ \wedge b} \{\lceil \tau\{\!|c|\!\}(0) \rceil\} \right]\!\!\right]_\Pi(\sigma),$$

from which the claim immediately follows by applying Lemma 3.3.8. □

Picking $p \equiv 0$ in the lemma above yields the desired result that our rules indeed produce sufficient permission preconditions.

Corollary 3.3.18. » *For all statements c, the permission expression $\tau\{\!|c|\!\}(0)$ is a sufficient permission precondition for c, assuming the soundness conditions from Equation (3.1) and Equation (3.4) hold for all loops in c.* «

Soundness of Postconditions. With our permission precondition and difference rules extended to loops, we can express permission postconditions for arbitrary statements – just like for loop-free statements – as the sum of the respective permission precondition and the permissions difference (to account for the permissions lost or gained by the execution of the statement).

Theorem 3.3.19. » *For all statements c with permission precondition $\tau\{\!|c|\!\}(0)$, the permission expression $\tau\{\!|c|\!\}(0) + \delta\{\!|c|\!\}(0)$ is a guaranteed permission postcondition, assuming the soundness conditions from Equation (3.1) and Equation (3.4) hold for all loops in c.* «

Proof. Analogous to the proof of Lemma 3.2.25. □

So far, we have described how to construct permission preconditions and postconditions for code containing arbitrarily nested loops. Most program verifiers, however, handle loops modularly and require the user to annotate all loops with an inductive loop invariant, which are different from loop preconditions in cases where individual

loop iterations give away or gain permissions. In the upcoming section, we will describe how to capture the permissions lost and gained by already executed loop iterations, and how to combine them with loop preconditions in order to obtain a loop invariant suitable for program verifiers.

3.4 Loop Invariants

In this section, we describe how we can construct a permission invariant that is both sufficient and inductive. Here, *sufficient* means that the permission invariant must provide enough permissions to successfully execute all remaining loop iterations. Analogous to classical invariants, a permission invariant is *inductive* if any execution of a single loop iteration starting in a state holding the permissions captured by the invariant is guaranteed to end up in a state that still holds the permissions captured by the invariant.

The permission preconditions for loops, as described in Section 3.3, provide sufficient permissions to successfully execute all iterations of the loop for which they were constructed. It is not hard to see that they are even inductive, as long as the loop at hand does not give away permissions. If, however, the loop at hand exhales permissions in some or all of its loop iterations, the loop precondition ends up referring to permissions that are no longer present and, consequently, is *not* inductive.

Roughly speaking, our permission invariants described in this section are based on our loop preconditions from Section 3.3.1 but adjusted for permissions lost and gained by already executed loop iterations. We facilitate this permission adjustment by introducing the novel concept of *progressive loop invariants* that allows us to distinguish between past and future loop iterations. These progressive loop invariants can then be used to refine the permission expressions from Section 3.3.2 to capture the permissions lost and gained by already executed loop iterations (rather than the permission changes caused by the entire loop).

This section is based on joint work with Becker conducted in the scope of his Master's thesis [10].

Section Outline. In Section 3.4.1, we first introduce progressive loop invariants that let us distinguish between past and future loop iterations. In particular, we show how to leverage existing numerical analyses to construct such progressive loop invariants. In Section 3.4.2, we then make use of these progressive loop invariants to construct pointwise maximum expressions capturing the permission differences caused by the execution of the loop so far. Moreover, we show how to combine them with our loop preconditions to obtain sufficient and inductive permission invariants.

```
var i: Int
i := 0
while (i < len(a)) {
  exhale acc(a[i])
  i := i + 1
}
```

Listing 3.8. A code snippet that encodes a loop forking a series of threads, each of which requiring permissions to read the i-th array element.

```
var i: Int
var i': Int
i := 0
assume 0 ≤ i' ∧ i' ≤ len(a)
i := i'
while (i < len(a)) {
  exhale acc(a[i])
  i := i + 1
}
```

Listing 3.9. The instrumented loop used to infer our progressive loop invariants.

3.4.1 Progressive Invariants

Regular loop invariants capture the set of reachable states and are expressed as a constraint on a *single* program state. We introduce the novel concept of *progressive loop invariants* that capture how states at different points in time throughout the execution relate to each other and, therefore, are expressed as constraints over *pairs* of program states.

Below, we focus on obtaining *over-approximate* progressive invariants by leveraging an off-the-shelf over-approximating numerical analysis. Note, however, that *under-approximate* progressive invariants can be obtained in a similar fashion using an under-approximating numerical analysis.

Approach. Roughly speaking, we obtain our progressive loop invariants by simulating stopping the numerical analysis at the beginning of an arbitrary loop iteration, copying all program variables modified by the loop, and then resuming the execution. Intuitively, the numerical analysis then infers constraints that relate the program variables with a copy of themselves from an arbitrary but earlier loop iteration. Following this idea, the generation of progressive loop invariants consists of the following three steps:

1. First, we run a standard numerical analysis on the original program to obtain a regular loop invariant I^+ for the loop $l \equiv$ while (b) $\{c\}$ at hand.

2. We then simulate the stopping of the loop at the beginning of an arbitrary loop iteration by replacing the loop with

$$l' :\equiv \texttt{assume}\ I^+[\vec{x}'\backslash\vec{x}]\,;\vec{x} \coloneqq \vec{x}'\,;\texttt{while}\ (b)\ \{c\},$$

   where $\vec{x} = \langle x_1, \ldots, x_n\rangle$ denotes the variables modified by the loop body c and $\vec{x}' = \langle x'_1, \ldots, x'_n\rangle$ are fresh variables. The $\texttt{assume}\ I^+[\vec{x}'\backslash\vec{x}]$ statement constrains the variables $\vec{x}'$ to capture an arbitrary state allowed by the loop invariant I^+; in particular, this includes all states that can potentially be encountered at the beginning of a loop iteration. The assignment $\vec{x} \coloneqq \vec{x}'$ then ensures that the execution of the loop starts in the state captured by the variables $\vec{x}'$.

3. Finally, we run the numerical analysis again – now, on the instrumented program – in order to obtain numerical constraints that relate the program variables $\vec{x}$ with the copy $\vec{x}'$ of themselves from an arbitrary but earlier loop iteration.

Example 3.4.1. » Let us consider the loop shown in Listing 3.8 and assume that a numerical analysis infers the loop invariant $0 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{len}(\mathsf{a})$. Running the numerical analysis again on the instrumented loop shown in Listing 3.9 yields

$$\mathsf{i}' \leq \mathsf{i} \wedge 0 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{len}(\mathsf{a}) \wedge 0 \leq \mathsf{i}' \wedge \mathsf{i}' \leq \mathsf{len}(\mathsf{a}).$$

While this constraint repeats the original loop invariant for i and i', it also contains the conjunct $\mathsf{i}' \leq \mathsf{i}$ telling us that i is never going to decrease. «

Relational Constraints. Let R denote the loop invariant inferred by the numerical analysis for our instrumented loop l' that was constructed as described above. We observe that the invariant R relates two states: The constraint does not only refer to the original program variables $\vec{x}$ representing an arbitrary state σ but also comprises variables $\vec{x}'$ collectively capturing another state σ' at an earlier point in the execution; below, we will call any such constraint *relational*.

For our elaborations, we want to be able to determine whether a relational constraint R is satisfied by any given pair of states σ and σ' for the original program. To facilitate this, we use $\sigma \times \sigma'$ to denote a state for the instrumented program in which the variables $\vec{x}$ and $\vec{x}'$ capture the states σ and σ', respectively; more formally, for all $i \in \{1, \ldots, n\}$, we have $[\![x_i]\!](\sigma \times \sigma') = [\![x_i]\!](\sigma)$ and $[\![x'_i]\!](\sigma \times \sigma') = [\![x_i]\!](\sigma')$. We can then evaluate $[\![R]\!](\sigma \times \sigma')$ to determine whether the relationship described by the relational constraint R holds for the states σ and σ'.

Progressive Constraint. Next, we prove that the loop invariant R resulting from the numerical analysis on our instrumented program captures the successor relation indicating whether a state can be reached from another state by executing zero, one, or more iterations of the loop.

Lemma 3.4.2. » *For all states σ and σ' with $[\![I^+]\!](\sigma)$ and $\langle l, \sigma\rangle \rightsquigarrow^* \langle l, \sigma'\rangle$, we have $[\![R]\!](\sigma' \times \sigma)$, where R denotes an over-approximate loop invariant for l'.* «

Proof. First, we observe that $\sigma \times \sigma$ is a possible initial state for the loop l'. Moreover, since $\langle l, \sigma\rangle \rightsquigarrow^* \langle l, \sigma'\rangle$ is a valid derivation sequence for the original program, we also have $\langle l', \sigma \times \sigma\rangle \rightsquigarrow^* \langle l', \sigma' \times \sigma\rangle$ for the instrumented program; note that the variables $\vec{x}\,'$ never get reassigned and, therefore, remain unchanged. Thus, we can conclude that any sound over-approximate loop invariant R must satisfy $[\![R]\!](\sigma' \times \sigma)$. □

Past Progressive Invariants. We expect the relational constraint R inferred as a loop invariant for our instrumented loop to also imply the original loop invariant I^+, as well as the adapted loop invariant $I^+[\vec{x}\,' \backslash \vec{x}]$, for that matter. Although, there are no formal guarantees that the numerical analysis at hand produces a constraint R that satisfies this property, we can easily construct an expression that does: We use R' to denote the constraint R restricted to the parts that actually relate the variables $\vec{x}$ and $\vec{x}\,'$ and then define our *past progressive invariant* as

$$I^+_{\leftarrow} :\equiv R' \wedge I^+[\vec{x}\,' \backslash \vec{x}] \wedge I^+.$$

Note that, for Example 3.4.1, this transformation does not change anything as the original constraint R already includes the invariants I^+ and $I^+[\vec{x}\,' \backslash \vec{x}]$. Moreover, we observe that the claim from Lemma 3.4.2 is still true for this transformed relational constraint: We clearly do not strengthen the relational part by omitting the non-relational bits of R and also do not exclude any loop iterations by conjoining the over-approximate loop iterations I^+ and $I^+[\vec{x}\,' \backslash \vec{x},]$.

Future Progressive Invariants. By symmetry, we obtain a *future progressive invariant*, denoted $I^+_{\rightarrow}$, from a past progressive invariant $I^+_{\leftarrow}$, simply by switching the roles of $\vec{x}$ and $\vec{x}\,'$.

Strict Progressive Invariants. Note that we can easily generate strict versions of our past and future progressive invariants by adding an iteration counter $\mathtt{c}$ to each loop: For example, $I^+_{\leftarrow} \wedge \mathtt{c}' < \mathtt{c}$ over-approximates all states that were encountered in strictly earlier iterations. Similarly, we can use $I^+_{\leftarrow} \wedge \mathtt{c}' + 1 = \mathtt{c}$ to capture states that are *exactly* one iteration apart. Below, we will make use of this; we therefore assume that our instrumentation equips all loops with iteration counters.

Example 3.4.3. » Adding an iteration counter c to our loop from Example 3.4.1 results in a past progressive invariant $I^+_{\leftarrow}$ capturing $\mathsf{i}' + \mathsf{c} = \mathsf{i} + \mathsf{c}'$, assuming a sufficiently precise numerical analysis. We observe that the constraint $I^+_{\leftarrow} \wedge \mathsf{c}' < \mathsf{c}$ describes strictly past loop iterations as it implies $\mathsf{i}' < \mathsf{i}$. Moreover, we observe that $I^+_{\leftarrow} \wedge \mathsf{c}' + 1 = \mathsf{c}$ implies $\mathsf{i}' + 1 = \mathsf{i}$, which is exactly the constraint we would expect to describe subsequent loop iterations. «

Monotonicity. Assuming some current state σ, the set of states captured by any relational constraint R is given by

$$S_R(\sigma) \coloneqq \{\sigma' \in \Sigma \mid [\![R]\!](\sigma \times \sigma')\}.$$

For a past progressive invariant $I^+_{\leftarrow}$, we expect this set $S_{I^+_{\leftarrow}}(\sigma)$ to not shrink from one iteration to the next iteration, as the set of states that lie in the past can only grow. Conversely, for a future progressive invariant $I^+_{\rightarrow}$, we expect this set $S_{I^+_{\rightarrow}}(\sigma)$ to not grow from one iteration to another, as the set of states that lie in the future can only shrink. This observation motivates us to introduce the notion of monotonicity:

Definition 3.4.4. » A relational constraint R is *monotonically non-decreasing*, if and only if for all pairs of states σ and σ' with $\langle l, \sigma\rangle \rightsquigarrow^* \langle l, \sigma'\rangle$, we have $S_R(\sigma) \subseteq S_R(\sigma')$. Likewise, the constraint R is *monotonically non-increasing*, if and only if for all pairs of states σ and σ' with $\langle l, \sigma\rangle \rightsquigarrow^* \langle l, \sigma'\rangle$, we have $S_R(\sigma') \subseteq S_R(\sigma)$. «

Typically, we expect the past and future progressive invariants obtained as described above to be monotonically non-decreasing and monotonically non-increasing, respectively. In general, however, there are no formal guarantees that ensure that this is always the case. Therefore, the following lemma introduces a condition that allows us to check their monotonicity; this condition is decidable and can automatically and efficiently be checked using an SMT solver.

Lemma 3.4.5. » *A relational constraint R is monotonically non-decreasing if*

$$\forall v_1, v_2 \in \mathbb{Z}^n \colon [\![I^+_{\leftarrow}[\vec{v}_1 \backslash \vec{x}\,'][\vec{v}_2 \backslash \vec{x}\,] \Rightarrow (R[\vec{v}_1 \backslash \vec{x}\,'] \Rightarrow R[\vec{v}_2 \backslash \vec{x}\,'])]\!](\sigma),$$

independent of the state σ. Moreover, by replacing the past progressive invariant $I^+_{\leftarrow}$ with the future progressive invariant $I^+_{\rightarrow}$ in the equation above, we can check whether the relational constraint R is monotonically non-increasing. «

Proof. By Lemma 3.4.2, the condition $I^+_{\leftarrow}[\vec{v}_1 \backslash \vec{x}\,'][\vec{v}_2 \backslash \vec{x}\,]$ is true for all pairs of vectors $\vec{v}_1$ and $\vec{v}_2$ where $\vec{v}_1$ corresponds to loop iteration preceding the loop iteration represented by $\vec{v}_2$. The implication $R[\vec{v}_1 \backslash \vec{x}\,'] \Rightarrow R[\vec{v}_2 \backslash \vec{x}\,']$ then captures the subset relation required for the relational constraint to be monotonically non-decreasing. □

Note that, due to $I^+_{\leftarrow}$ being over-approximate, the converse of Lemma 3.4.5 is not necessarily true.

3.4.2 Permission Invariants

As indicated above, the construction of our loop invariants makes use of permission expressions that represent the permissions lost and gained by previous loop iterations.

Lost Permissions. We start by describing how to express the permissions lost by previous loop iterations. Roughly speaking, our goal is to generalise the permission losses $d^-_0 :\equiv \mathsf{min}(\lfloor\!\lfloor \delta[\![c]\!](0) \rfloor\!\rfloor, 0)$ caused by a single loop iteration using pointwise minimum expressions ranging over states corresponding to already executed loop iterations. We observe that the pointwise maximum $\mathsf{max}_{\vec{x} \mid I^+ \wedge b}\{\lceil\!\lceil \tau[\![c]\!](0) \rceil\!\rceil\}$ used for our loop preconditions captures the variables $\vec{x}$ and, therefore, results in an expression that is independent of the values of $\vec{x}$. In contrast, we want our permission losses to depend on the current values of $\vec{x}$; otherwise, they would represent the same permissions in every loop iteration and, consequently, be unsuitable for our purposes. We get around this, by letting our minimum expression capture the variables $\vec{x}'$ corresponding to the *other* state introduced by our progressive loop invariants (cf. Section 3.4.1). To do so, we first need to adapt the permission losses to be expressed in terms of $\vec{x}'$, which can be done by a simple variable substitution $d^-_0[\vec{x}'\backslash\vec{x}]$. Our pointwise minimum expression then reintroduces the variables $\vec{x}$ via the relational constraint $P^+ :\equiv I^+_{\leftarrow} \wedge \neg I^+_{\rightarrow} \wedge b[\vec{x}'\backslash\vec{x}]$. Intuitively, this generalises the state represented by the variables $\vec{x}'$ to all states described by P^+, which can be thought of as all states occurring before the state captured by the variables $\vec{x}$.

Following our elaborations above, we define the permission expression that summarises all permissions lost so far as

$$d^-_{\leftarrow} :\equiv \min_{\vec{x}' \mid P^+}\{d^-_0[\vec{x}'\backslash\vec{x}]\}, \quad \text{where} \begin{cases} P^+ & :\equiv I^+_{\leftarrow} \wedge \neg I^+_{\rightarrow} \wedge b[\vec{x}'\backslash\vec{x}] \\ d^-_0 & :\equiv \mathsf{min}(\lfloor\!\lfloor \delta[\![c]\!](0) \rfloor\!\rfloor, 0). \end{cases}$$

We observe that the permission losses $d^-_{\leftarrow}$ only range over loop iterations for which $\neg I^+_{\rightarrow}$ are true; that is, they are guaranteed to *not* include loop iterations that have not been encountered yet. The last conjunct $b[\vec{x}'\backslash\vec{x}]$ ensures that we do not consider spurious loop iterations that do not satisfy the loop condition.

Example 3.4.6. » Let us revisit the loop shown in Listing 3.8, for which we infer the past invariant $I^+_{\leftarrow} \equiv \mathsf{i}' \leq \mathsf{i} \wedge 0 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{len}(\mathsf{a}) \wedge 0 \leq \mathsf{i}' \wedge \mathsf{i}' \leq \mathsf{len}(\mathsf{a})$. Recall that the corresponding future progressive invariant is obtained by switching the roles of i and i'; that is, we have $I^+_{\rightarrow} \equiv \mathsf{i} \leq \mathsf{i}' \wedge 0 \leq \mathsf{i}' \wedge \mathsf{i}' \leq \mathsf{len}(\mathsf{a}) \wedge 0 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{len}(\mathsf{a})$. Furthermore, we

observe that $d_0^-[\mathsf{i}'\backslash\mathsf{i}] \equiv \min\big(0 - \alpha_{\mathsf{a}[\mathsf{i}']}(1), 0\big)$ can be simplified to $-\alpha_{\mathsf{a}[\mathsf{i}']}(1)$ and that the loop condition is given by $b \equiv \mathsf{i} < \mathsf{len}(\mathsf{a})$. Thus, we can express the permissions lost by already executed loop iterations as

$$\min_{\mathsf{i}' \mid P^+} \{-\alpha_{\mathsf{a}[\mathsf{i}']}(1)\},$$

where P^+, by construction, entails $0 \leq \mathsf{i}' \wedge \mathsf{i}' < \mathsf{len}(\mathsf{a})$ as well as $\mathsf{i}' \leq \mathsf{i} \wedge \neg(\mathsf{i} \leq \mathsf{i}')$, that is, $\mathsf{i}' < \mathsf{i}$. Rewriting the minimum $\min_{\mathsf{i}' \mid P^+}\{-\alpha_{\mathsf{a}[\mathsf{i}']}(1)\}$ as $-\max_{\mathsf{i}' \mid P^+}\{\alpha_{\mathsf{a}[\mathsf{i}']}(1)\}$, using our maximum elimination algorithm introduced in Section 3.5, and applying a suitable set of simplification rules yields

$$-\left((\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{i} \wedge \mathsf{i} \leq \mathsf{len}(\mathsf{a}))\ ?\ 1 : 0\right).$$

We observe that this permission expression captures precisely all permissions given away by previous loop iterations. «

Gained Permissions. Analogous to the lost permissions, we also define an expression that summarises all permissions gained so far as

$$d_{\leftarrow}^+ :\equiv \max_{\vec{x}' \mid P^-} \{d_0^+[\vec{x}'\backslash\vec{x}]\}, \quad \text{where} \begin{cases} P^- & :\equiv I_{\leftarrow}^- \wedge \neg I_{\rightarrow}^- \wedge b[\vec{x}'\backslash\vec{x}] \\ d_0^+ & :\equiv \max(\lfloor\!\lfloor \delta\{\!\{c\}\!\}(0) \rfloor\!\rfloor, 0). \end{cases}$$

Note that, due to P^- being a refinement of the under-approximate past progressive invariant $I_{\leftarrow}^-$, the permission gains $d_{\leftarrow}^+$ are guaranteed to only reflect permissions gained in iterations that actually occurred.

Permission Invariant. Using, the permission expressions $d_{\leftarrow}^-$ and $d_{\leftarrow}^+$ summarising the permission losses and gains, respectively, so far, we can express our permission invariant as

$$\tau\{\!\{l\}\!\}(p) + d_{\leftarrow}, \qquad \text{where } d_{\leftarrow} :\equiv d_{\leftarrow}^- + d_{\leftarrow}^+$$

and p is the accumulator expression from our loop preconditions.

Example 3.4.7. » Let us revisit the loop shown in Listing 3.8, for which we already have computed a permission expression capturing lost permissions in Example 3.4.6. For this loop, our permission analysis produces the permission invariant

$$\max_{\mathsf{i} \mid 0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a})} \{\alpha_{\mathsf{a}[\mathsf{i}]}(1)\} + \min_{\mathsf{i}' \mid P^+} \{(-\alpha_{\mathsf{a}[\mathsf{i}']}(1))\}.$$

Note that we omit gained permissions, as they are 0 for all loop iterations. Using our maximum elimination algorithm and simplifications, this loop invariant gets rewritten into

$$\big((\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ 1 : 0\big) + \big((\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a}))\ ?\ -1 : 0\big),$$

which can be further simplified into

$$(\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ (\mathsf{q}_i < \mathsf{i}\ ?\ 0 : 1) : 0\,.$$

We observe that this permission expression captures precisely all permissions required for the current and future loop iterations. «

First, we observe that this permission invariant is sufficient by construction: The loop precondition $\tau\{\!|l|\!\}(0)$ provides sufficient permissions to execute *all* loop iterations (regardless of potentially gained permissions) and the lost permissions $d^-_{\leftarrow}$ are guaranteed to only *not* reflect loop iterations that have not been encountered yet.

For the permission invariant to be inductive, the relational constraints P^+ and P^- must be monotonically non-decreasing. Additionally, the set of states described by P^+ must grow at just about the right pace. Below, we will present a sufficient condition under which this is the case. Note that, due to its under-approximate nature, the set P^- cannot grow too quickly: In any loop iteration, it captures at most one additional loop iteration compared to the previous loop iteration.

Inductiveness Condition. Recall that our soundness condition for permission differences requires that we never encounter two iterations where the variables $\vec{x}$ have exactly the same values at the beginning of the loop iterations. This was essential to be able to apply Lemma 3.3.11 in our soundness proof (proving Lemma 3.3.12) for the permission differences, which essentially went by induction on the loop iterations of an arbitrary execution (which we did in disguise over the length of the derivation sequence).

Roughly speaking, our goal is to transfer the same idea to our loop invariants. To this end, let us consider the initial states σ_1 and σ_2 of two subsequent loop iterations; that is, we have $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma\rangle$. Let us have a look at the sets of states $S_{P^+}(\sigma_1)$ and $S_{P^+}(\sigma_2)$ described by the relational constraint P^+ evaluated in the states σ_1 and σ_2, respectively. As illustrated in Figure 3.10 we want that the state σ_1 corresponding to the earlier loop iteration is contained within the difference of these sets; combined with the monotonicity of the progressive invariants, we can then establish the requirements to apply Lemma 3.3.11 after every loop iteration. As we always have $\sigma_1 \notin S_{P^+}(\sigma_2)$, by definition of the relational constraint P^+, it suffices if our soundness condition enforces $\sigma_1 \in S_{P^+}(\sigma_2)$; this is the case, whenever $\sigma_1 \notin S_{I^+_{\rightarrow}}(\sigma_2)$. Therefore, we formulate our soundness condition as

$$\forall \vec{v}_1, \vec{v}_2 \in \mathbb{Z}\colon [\![\mathsf{c}' + 1 = \mathsf{c} \Rightarrow \neg I^+_{\leftarrow}[\vec{v}_1\backslash\vec{x}\,'][\vec{v}_2\backslash\vec{x}\,] \vee \neg I^+_{\rightarrow}[\vec{v}_1\backslash\vec{x}\,'][\vec{v}_2\backslash\vec{x}\,]]\!](\sigma), \tag{3.21}$$

independent of the state σ, where c and c' are an iteration counter and the copy thereof, respectively. Intuitively, this condition holds whenever the progressive invariants

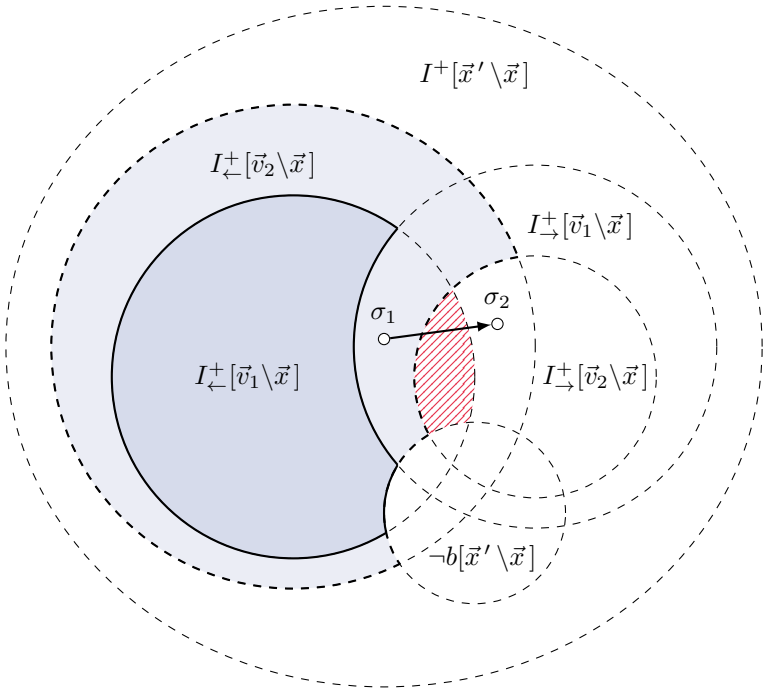


Figure 3.10. An illustration showing the relationship between sets of states corresponding to the variables $\vec{x}\,'$ captured by different relational constraints. The circles represent the (non-relational) loop invariant $I^+[\vec{x}\,'\setminus\vec{x}]$ and negated loop condition $\neg b$, as well as the (relational) past and future progressive invariants $I^+_{\leftarrow}$ and $I^+_{\rightarrow}$ evaluated in the initial states σ_1 and σ_2 (captured by the value vectors $\vec{v}_1$ and $\vec{v}_2$, respectively) of two subsequent loop iterations. The shaded areas highlight the set of states $S_{P^+}(\sigma_1)$ (◯) and $S_{P^+}(\sigma_2)$ (◌). Our loop invariant is inductive only if the hatched area (▨) is empty.

are strong enough to distinguish strictly past loop iterations from strictly future loop iterations. For all examples in our benchmark suite, we were able to obtain sufficiently strong progressive invariants by using the *polyhedra abstract domain* [30] provided by Apron [60].

Correctness Proof. With our inductiveness condition at hand, we are ready to prove the correctness of our loop invariant. First, we show that it can always be established upon entering a loop. To do so, we prove that – upon entering the loop – the loop invariant is at most the loop precondition, for which we are guaranteed to have sufficient permissions.

Lemma 3.4.8. » *For all entry states σ of the loop l at hand and all non-negative permission expressions p, we have*

$$[\![\tau\{\!|l|\!\}(p) + d_{\leftarrow}]\!]_\Pi(\sigma) \sqsubseteq [\![\tau\{\!|l|\!\}(p)]\!]_\Pi(\sigma)$$ «

Proof. Let us consider an arbitrary entry state σ of the loop at hand. Note that it suffices to show that $[\![d^-_{\leftarrow} + d^+_{\leftarrow}]\!]_\Pi(\sigma) \sqsubseteq \pi_{\mathsf{zero}}$. First, we observe that, by construction, we have $[\![d^-_{\leftarrow}]\!]_\Pi(\sigma) \sqsubseteq \pi_{\mathsf{zero}}$. Moreover, since P^- describes an empty set of past states when evaluated in an entry state of the loop, we have $[\![d^+_{\leftarrow}]\!]_\Pi(\sigma) = \pi_{\mathsf{zero}}$. From this, the claim immediately follows. □

Next, we show that the permissions described by our loop invariant are preserved throughout the execution of the loop. We do this by proving that the permissions are preserved by an arbitrary loop iteration.

Lemma 3.4.9. » *For all states σ and σ' and all non-negative permission expressions p with $\sigma \vDash [\![\tau\{\!|l|\!\}(p) + d_{\leftarrow}]\!]_\Pi(\sigma)$ as well as $[\![I^+ \wedge b]\!](\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$, we also have $\sigma' \vDash [\![\tau\{\!|l|\!\}(p) + d_{\leftarrow}]\!]_\Pi(\sigma')$, assuming our soundness and inductiveness conditions hold* «

Proof. Let us consider two arbitrary states $\sigma = \langle s, h, \pi\rangle$ and $\sigma' = \langle s', h', \pi'\rangle$ corresponding to subsequent loop iterations; that is, states with $[\![I^+ \wedge b]\!](\sigma)$ and $\langle c, \sigma\rangle \rightsquigarrow^* \langle \mathsf{skip}, \sigma'\rangle$. Also, suppose that our soundness and inductiveness conditions hold. To prove the claim, it suffices to show

$$[\![d_{\leftarrow}]\!]_\Pi(\sigma') - [\![d_{\leftarrow}]\!]_\Pi(\sigma) \sqsubseteq \pi' - \pi,$$

where, as before, $d_{\leftarrow} \equiv d^-_{\leftarrow} + d^+_{\leftarrow}$. To this end, for $\pm \in \{+, -\}$, we define

$$A^\pm \coloneqq \{\vec{v} \in \mathbb{Z}^n \mid \exists \sigma'' \in \Sigma\colon \vec{v}_{\sigma''} = \vec{v} \wedge \sigma'' \in S_{P^\pm}(\sigma)\}$$
$$B^\pm \coloneqq \{\vec{v} \in \mathbb{Z}^n \mid \exists \sigma'' \in \Sigma\colon \vec{v}_{\sigma''} = \vec{v} \wedge \sigma'' \in S_{P^\pm}(\sigma')\}$$

We observe that, since our relational constraint $P^\pm$ include the loop invariant $I^\pm$, we have $A^\pm \subseteq V^\pm$ and $B^\pm \subseteq V^\pm$. Furthermore, by monotonicity of the relational constraints P^+ and P^-, we have $A^+ \subseteq B^+$ and $A^- \subseteq B^-$. Following our elaborations around our inductiveness condition above, we have $\vec{v}_\sigma \notin A^+$ but $\vec{v}_\sigma \in B^+$. Moreover, A^- and B^- differ by at most $\vec{v}_\sigma$. Thus, we can conclude

$$\begin{aligned}
[\![d_{\leftarrow}]\!]_\Pi(\sigma') - [\![d_{\leftarrow}]\!]_\Pi(\sigma) &= [\![g^*_{\langle B^+, B^-\rangle}(0)]\!]_\Pi(\sigma') - [\![g^*_{\langle A^+, A^-\rangle}(0)]\!]_\Pi(\sigma) && (3.22)\\
&\sqsubseteq [\![g_{\vec{v}_\sigma}(0)]\!]_\Pi(\sigma) && (3.23)\\
&\sqsubseteq [\![\delta\{\!|c|\!\}(0)]\!]_\Pi(\sigma) && (3.24)\\
&\sqsubseteq \pi' - \pi, && (3.25)
\end{aligned}$$

as claimed. Above, the first step (3.22) holds by the definition of g^*. The inequality (3.23) is justified by Lemma 3.3.11. The next inequality (3.24) is due to the definition of $g_{\vec{v}_\sigma}(0)$. Finally, the last inequality then follows by Lemma 3.2.24. □

Finally, we briefly convince ourselves that the loop invariant leaves sufficient permissions to execute everything that comes after the loop upon termination of the loop: We know that $\tau\{\!|l|\!\}(p) + \delta\{\!|l|\!\}(0)$ is a guaranteed postcondition of the loop. Since our relational constraints P^+ and P^- do not describe more states than the loop invariants I^+ and I^-, we easily see that, at the end of the loop, $d_{\leftarrow}$ describes at most the permissions $\delta\{\!|l|\!\}(0)$; thus, our loop invariants retains all permissions promised by the loop postcondition. Note that this also shows, that our loop invariants seamlessly integrate in our permission analysis described in the previous section.

Theorem 3.4.10. » *Assuming that our soundness and inductiveness conditions hold, the permission expression*

$$\tau\{\!|l|\!\}(p) + d_{\leftarrow}$$

is a sound and inductive permission invariant for the loop l. «

Proof. The claim follows from our elaborations above. □

Our permission invariants are compatible with the loop preconditions and postconditions described earlier. With this, our permission analysis is capable of inferring all permission specifications required by a typical permission-based verifier: We can use our permission analysis to infer sufficient permission preconditions and guaranteed permission postconditions for methods, as well as inductive permission invariants for loops.

3.5 Maximum Elimination

In this section, we present our maximum elimination algorithm. Roughly speaking, this algorithm performs an involved syntactic transformation that replaces a pointwise maximum expression over an *unbounded* set of values with an equivalent maximum expression ranging over a *finite* set of values, which can then be expressed using a finite number of nested binary expressions.

Cooper's Quantifier Elimination. Our maximum elimination algorithm builds upon ideas from Cooper's classical *quantifier elimination* algorithm [27] which, given a formula $\exists x \in \mathbb{Z} \colon b$, where b is a quantifier-free Presburger formula, computes an equivalent and quantifier-free formula b'.

Essentially, eliminating the quantifier corresponds to finding out whether there is a value $i \in \mathbb{Z}$ for which $b[i\backslash x]$ is true; throughout this section, we will call any such value a *solution*. In order to find out whether there are any solutions, Cooper's algorithm – using boolean and arithmetic manipulations – first rewrites the boolean

expression b into a *simpler* form in which the variable x occurs at most once per literal and with no coefficient. The key idea is then to reduce the problem of checking for the existence of the solution to a disjunction of two high-level cases: If there is a solution, there is either a *smallest* solution or there have to be *arbitrarily small* solutions, otherwise. In our elaborations below, these cases will be treated separately.

Our Maximum Elimination. In principle, any maximum $q \coloneqq \max_{x \in \mathbb{Z}}\{p\}$ can be defined using two first-order quantifiers $\forall x \in \mathbb{Z}\colon p \leq q$ and $\exists x \in \mathbb{Z}\colon p = q$. One might therefore be tempted to tackle our maximum elimination problem using quantifier elimination directly. We explored this possibility and found two serious drawbacks. First, the resulting formula does not yield a permission-typed expression that we can plug back into our analysis. Second, the resulting formulas are extremely large and hard to simplify since the relevant information is often spread across many terms due to the two separate quantifiers.

Our maximum elimination algorithm addresses these drawbacks by natively working with arithmetic expressions, while mimicking the basic ideas of Cooper's quantifier elimination algorithm and incorporating domain-specific optimisations. Analogous to Cooper's algorithm, our algorithm also exploits the case split distinguishing between arbitrarily small solutions and smallest ones. Note that – in the context of maximum elimination – a solution is any value $i \in \mathbb{Z}$ that maximises $p[i \backslash x]$.

Section Outline. Throughout this section, we describe Cooper's classical quantifier elimination algorithm and our extension to a maximum elimination algorithm in lockstep. In Section 3.5.1, we introduce a simple form of boolean expressions and permissions expressions, upon which the elimination algorithms operate. In particular, we elaborate how more general expressions can be rewritten into this simpler form. We then proceed by detailing the construction of expressions capturing solutions corresponding to each case of our high-level case split: In Section 3.5.2 and Section 3.5.3, we show how to construct expressions covering arbitrarily small and smallest solutions, respectively. Ultimately, in Section 3.5.4, we then combine these expressions to obtain the resulting expression free of quantifiers and pointwise maxima.

3.5.1 Simple Expressions

Next, we introduce the simpler form of expressions upon which the elimination algorithms operate. Moreover, we also elaborate how general boolean and permission expressions can be rewritten into this simpler form.

Simple Boolean Expressions. We start by defining the simple boolean expressions used by Cooper's quantifier elimination algorithm.

Definition 3.5.1. » A boolean expression b is *simple* over variable x, if and only if it is either

- a boolean expression b with $x \notin \text{free}(b)$
- a comparison of the form $b \equiv x \sim e$, where $\sim \in \{=, \neq, <, \leq, >, \geq\}$ and $x \notin \text{free}(e)$,
- a divisibility constraint of the form $b \equiv \mathsf{mod}(x + e, n) \sim 0$, where $x \notin \text{free}(e)$ and $\sim \in \{=, \neq\}$,
- a logical combination of the form $b \equiv b_1 \diamond b_2$, where b_1 and b_2 are simple boolean expressions and $\diamond \in \{\wedge, \vee\}$. «

Note that in the definition above $\text{free}(b)$ and $\text{free}(e)$ denote the sets of free variables appearing in the boolean expression b and arithmetic expression e, respectively.

Rewriting Boolean Expressions. In general, boolean expressions produced by our permission analysis are not in this simple form introduced above. Therefore, we describe how we can rewrite an arbitrary boolean expression b in Presburger arithmetic into an equivalent boolean expression that is in this simpler form

- First, the boolean expression is converted to *negation normal form* by pushing all negations down to its individual literals. This is done by repeatedly applying De Morgan's law and double negation eliminations. In addition, negations around comparisons and divisibility constraints are eliminated by flipping the operator. For example, the comparison $\neg(x < 0)$ gets replaced with $x \geq 0$ and the divisibility constraint $\neg(\mathsf{mod}(x, 2) = 0)$ is replaced with $\mathsf{mod}(x, 2) \neq 0$.
- Next, we group all occurrences of the variable x in a literal such that each literal is either a comparison of the form $c_i \cdot x \sim_i e_i$, where $\sim_i \in \{=, \neq, <, \leq, >, \geq\}$ or a divisibility constraint of the form $\mathsf{mod}(c_i \cdot x + e_i, n_i) \sim_i 0$, where $\sim_i \in \{=, \neq\}$. For example, the literal $2x + 3 \geq 1 - x$ is rewritten to $3x \geq -2$.
- Finally, we eliminate the coefficients appearing before the variable x. To this end we collect all coefficients $c_1, c_2, \ldots, c_n$ and compute their least common multiple $c \coloneqq \text{lcm}(c_1, \text{lcm}(c_2, \text{lcm}(\ldots, c_n)))$.

  We then normalise the coefficients by multiplying each literal containing the variable x with an appropriate constant such the coefficient before the variable x

becomes c: For each literal where the current coefficient is c_i, this constant is given by $d_i := c/c_i$; note that this is always an integer. Now, if the literal is a comparison $c_i \cdot x \sim_i e_i$, we replace it with $c \cdot x \sim_i d_i \cdot e_i$. Otherwise, if the literal is a divisibility constraint $\mathsf{mod}(c_i \cdot x + e_i, n_i) \sim_i 0$, we replace it with $\mathsf{mod}(c \cdot x + d_i \cdot e_i, n_i) \sim_i 0$.

Once all coefficients have been normalised to be c, we can ultimately replace each occurrence of $c{\cdot}x$ with a fresh variable x'. This yields a boolean expression b' that – when conjoined with $\mathsf{mod}(x', c) = 0$ – is equivalent with the original boolean expression b. Note that the additional divisibility constraint is necessary since b' might be true for values of x' that do not correspond to integer values x'/c for b.

Example 3.5.2. » The quantifier $\exists \mathsf{i} : \mathsf{a} \leq \mathsf{i} \land 2 \cdot \mathsf{i} < \mathsf{b}$ gets rewritten into the equivalent quantifier $\exists \mathsf{i}' : \mathsf{i}' \geq 2 \cdot \mathsf{a} \land \mathsf{i}' < \mathsf{b} \land \mathsf{mod}(\mathsf{i}', 2) = 0$. «

Note that, any boolean expression that does not admit to be rewritten into a simple boolean expression can be handled – although, at the cost of losing precision – by approximating all problematic conditions appearing within them.

Example 3.5.3. » Due to its non-linearity in x, the boolean expression $x \geq 0 \land x^2 \neq 3$ cannot be rewritten into a simple boolean expression over x. However, it can be over-approximated using only its first conjunct $x \geq 0$. «

Simple Permission Expressions. Next, we introduce the simple form of permission expressions used by our maximum elimination algorithm.

Definition 3.5.4. » A permission expression p is *simple* over the variable x, if and ony if it is either

- a leaf expression of the form $p \equiv b \,?\, q : 0$
- an addition of the form $p \equiv p_1 + p_2$,
- a subtraction of the form $p \equiv p_1 - (b \,?\, q : 0)$,
- an extremum expression of the form $p \equiv \mathsf{min}(p_1, p_2)$ or $p \equiv \mathsf{max}(p_1, p_2)$, or
- a ternary expression of the form $p \equiv b \,?\, p_1 : 0$,

where b is a simple boolean expression over x, p_1 and p_2 are simple permission expressions x, and q a permission fraction. «

Rewriting Permission Expressions. Below, we describe howe we can reduce the problem of finding a method for eliminating a general pointwise maximum expression $\mathsf{max}_{x \mid b}\{p\}$ to that of eliminating *simple* maximum expressions.

To this end, we introduce a set of rewriting rules. First, we distribute any additions and subtractions over minimum, maximum, and ternary expressions. For example, we can use the rules (read from left to right)

$$\mathsf{max}(p_1, p_2) + p_2 = \mathsf{max}(p_1 + p_3, p_2 + p_3)$$
$$p_1 + \mathsf{max}(p_2, p_3) = \mathsf{max}(p_1 + p_2, p_1 + p_3)$$

to distribute additions over maximum expressions. Note that distributing subtractions over maximum expressions appearing as its second operand require the introduction of minimum expressions:

$$p_1 - \mathsf{max}(p_2, p_3) = \mathsf{min}(p_1 - p_2, p_1 - p_3)$$

This is the motivation for including minimum expressions in our grammar for permission expressions. All other rewriting rules for the distribution of additions and subtractions are defined similarly. In particular, we use the rules

$$p_1 - (p_2 + p_3) = (p_1 - p_2) - p_3$$
$$p_1 - (p_2 - p_3) = (p_1 - p_2) + p_3,$$

since the grammar for simple permission expression does not allow subtracting expressions that themselves have addition or subtraction operators inside them. Using the rewriting rules defined so far, we obtain an equivalent expression $\mathsf{max}_{x \mid b}\{p'\}$ in which any occurrences of minimum, maximum, and ternary expressions are nested at the topmost levels of p'.

Next, we describe how we can further rewrite this already slightly more structured expression: We observe that any binary maximum expression at the topmost level of p can be simply pulled outside the unbounded maximum. Therefore, we define

$$\underset{x \mid b}{\mathsf{max}}\{\mathsf{max}(p_1, p_2)\} = \mathsf{max}(\underset{x \mid b}{\mathsf{max}}\{p_1\}, \underset{x \mid b}{\mathsf{max}}\{p_2\}).$$

Similarly, for ternary expressions where the condition does *not* depend on the variable x, we can also pull the conditional outside the the unbounded maximum by pushing the unbounded maxima into either cases. That is, we define

$$\underset{x \mid b}{\mathsf{max}}\{b' \,?\, p_1 : p_2\} = b' \,?\, \underset{x \mid b}{\mathsf{max}}\{p_1\} : \underset{x \mid b}{\mathsf{max}}\{p_2\},$$

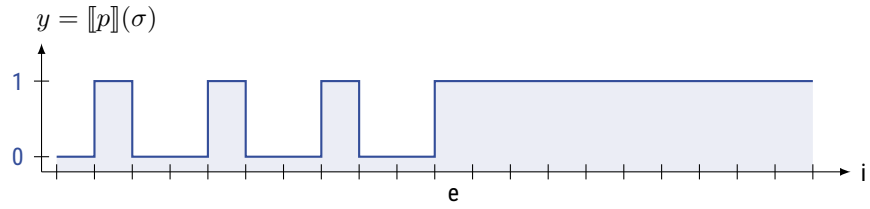


Figure 3.11. A graph showing the permission expression $p :\equiv (\mathsf{mod}(\mathsf{i}, 3) = 0 \vee \mathsf{e} \leq \mathsf{i})\ ?\ 1 : 0$ plotted as a function of i. There are arbitrarily small values for i that maximise p.

if $x \notin \text{free}(b')$. For ternary expressions where the condition *does* depend on the variable, we decompose the unbounded maximum into two simpler ones. More specifically, we define

$$\max_{x \mid b}\{b'\ ?\ p_1 : p_2\} = \mathsf{max}(\max_{x \mid b' \wedge b}\{p_1\},\ \max_{x \mid \neg b' \wedge b}\{p_2\}),$$

if $x \in \text{free}(b')$.

Iteratively and exhaustively applying these aforementioned rewriting rules yield an equivalent expression consisting of potentially many but simpler expressions $\max_{x \mid b'}\{p''\}$, where the permission expressions p'' are simple over x.

3.5.2 Arbitrarily Small Solutions

Next, we deal with arbitrarily small solutions. The key insight for the elaborations below is that – for the supported grammar of boolean and permission expressions – if one considers sufficiently small values of x, all evaluations of the expression at hand will yield the same value, modulo the divisibility constraints appearing in it. Thus, intuitively, to capture arbitrarily small solutions, it suffices to study the behaviour of the boolean or permission expression at hand as the value of x approaches negative infinity.

Example 3.5.5. » As an example, let us consider the maximum expression $\max_{\mathsf{i} \in \mathbb{Z}}\{p\}$, where $p :\equiv (\mathsf{mod}(\mathsf{i}, 3) = 0 \vee \mathsf{e} \leq \mathsf{i})\ ?\ 1 : 0$; a visual representation is shown in Figure 3.11. We observe that, for sufficiently small values for i, the condition $\mathsf{e} \leq \mathsf{i}$ is always false and the permission expression p behaves just like $(\mathsf{mod}(\mathsf{i}, 3) \vee \mathsf{false})\ ?\ 1 : 0$. «

Infinite Projection. In order to be able to reflect the behaviour of boolean and permission expression for values of x approaching negative infinity, we introduce an operator called *infinite projection*. We first define this infinite projection for boolean expressions.

Definition 3.5.6. » The *infinite projection* $\lim_x(b)$ of a simple boolean expression b over x is defined as

$$\lim_x(b) :\equiv \begin{cases} \mathsf{false} & \text{if } b \equiv x \circ e \text{ and } \circ \in \{=, \geq, >\} \\ \mathsf{true} & \text{if } b \equiv x \circ e \text{ and } \circ \in \{\neq, \leq, <\} \\ \lim_x(b_1) \diamond \lim_x(b_2) & \text{if } b \equiv b_1 \diamond b_2 \text{ and } \diamond \in \{\wedge, \vee\} \\ b & \text{otherwise.} \end{cases}$$ «

Note that, above, divisibility constraints are handled by the last case, which preserves them.

Lemma 3.5.7. » *For all simple boolean expressions b over x and all states σ, there is an integer $i_0 \in \mathbb{Z}$ such that, for all integers $i \in \mathbb{Z}$ with $i \leq i_0$, we have*

$$[\![\lim_x(b)[i\backslash x]]\!](\sigma) = [\![b[i\backslash x]]\!](\sigma).$$ «

Proof. The claim follows by straightforward induction on the structure of the boolean expression b. □

Example 3.5.8. » The infinite projection of $b \equiv 0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a}) \wedge \mathsf{mod}(\mathsf{i}, 2) = 0$ with respect to the variable i is given by

$$\lim_{\mathsf{i}}(b) \equiv \mathsf{false} \wedge \mathsf{true} \wedge \mathsf{mod}(\mathsf{i}, 2) = 0,$$

which can be simplified to false. «

The following definition extends the infinite projection to permission expressions.

Definition 3.5.9. » The *infinite projection* $\lim_x(p)$ of a simple permission expression p over x is defined as

$$\lim_x(p) :\equiv \begin{cases} \lim_x(b) \mathbin{?} q : 0 & \text{if } p \equiv b \mathbin{?} q : 0 \\ \lim_x(p_1) - (\lim_x(b) \mathbin{?} q : 0) & \text{if } p \equiv p_1 - (b \mathbin{?} q : 0) \\ \lim_x(b) \mathbin{?} \lim_x(p_1) : 0 & \text{if } p \equiv b \mathbin{?} p_1 : 0 \\ \mathsf{min}(\lim_x(p_1), \lim_x(p_2)) & \text{if } p \equiv \mathsf{min}(p_1, p_2) \\ \mathsf{max}(\lim_x(p_1), \lim_x(p_2)) & \text{if } p \equiv \mathsf{max}(p_1, p_2). \end{cases}$$ «

Lemma 3.5.10. » *For all simple permission expressions b and all states σ, there is an integer $i_0 \in \mathbb{Z}$ such that, for all integers $i \in \mathbb{Z}$ with $i \leq i_0$, we have*

$$[\![\lim_x(b)[i\backslash x]]\!](\sigma) = [\![b[i\backslash x]]\!](\sigma).$$ «

Proof. The claim follows by straightforward induction on the structure of the permission expression p and using Lemma 3.5.7. □

Example 3.5.11. » Let us consider the permission expression $p \equiv \mathsf{max}(\mathsf{i} \geq 0\ ?\ 1 : 0, \mathsf{rd})$. We have

$$\lim_{\mathsf{i}}(p) \equiv \mathsf{max}(\mathsf{false}\ ?\ 1 : 0, \mathsf{rd}),$$

which simplifies to rd. «

Period. Roughly speaking, while the infinite projection $\lim_x(p)$ preserves all divisibility constraints involving x, it eliminates all other constraints involving x. Consequently, the pattern of values of the infinite projection – if seen as a function of the variable x – repeats periodically. That is, to probe all possible values that can be assumed by the infinite projection, it suffices to look at values of x within the range of one *period*. Motivated by this insight, we define a function capturing the period of simple boolean and permission expressions:

Definition 3.5.12. » The *period* $\varphi_x(b)$ of a simple boolean expression b over x is defined as

$$\varphi_x(b) = \begin{cases} n & \text{if } b \equiv \mathsf{mod}(x + e, n) \circ 0 \text{ and } \circ \in \{=, \neq\} \\ \mathrm{lcm}(\varphi_x(b_1), \varphi_x(b_2)) & \text{if } b \equiv b_1 \diamond b_2, \text{ where } \diamond \in \{\wedge, \vee\} \\ 1 & \text{otherwise.} \end{cases}$$

«

Lemma 3.5.13. » *For all integer-typed variables x, all boolean expressions b that are simple with respect to x, all states σ, and all integers $i, i', k \in \mathbb{Z}$ with $i' = i + k \cdot \varphi_x(b)$, we have*

$$[\![\lim_x(b)[i \backslash x]]\!](\sigma) = [\![\lim_x(b)[i' \backslash x]]\!](\sigma).$$

«

Proof. The claim follows by straightforward induction on the structure of the boolean expression b. □

Definition 3.5.14. » The *period* $\varphi_x(p)$ of a simple permission expression p over x is defined as

$$\varphi_x(p) \coloneqq \begin{cases} \varphi_x(b) & \text{if } p \equiv b\ ?\ q : 0 \\ \mathrm{lcm}(\varphi_x(p_1), \varphi_x(p_2)) & \text{if } p \equiv p_1 + p_2 \\ \mathrm{lcm}(\varphi_x(b), \varphi_x(p_1)) & \text{if } p \equiv b\ ?\ p_1 : 0 \\ \mathrm{lcm}(\varphi_x(p_1), \varphi_x(p_2)) & \text{if } p \equiv \mathsf{min}(p_1, p_2) \text{ or } p \equiv \mathsf{max}(p_1, p_2). \end{cases}$$

«

Lemma 3.5.15. » *For all simple permission expressions p, all states σ, and all integers $i, i', k \in \mathbb{Z}$ with $i' = i + k \cdot \varphi_x(p)$, we have*

$$[\![\lim_x(p)[i\backslash x]]\!](\sigma) = [\![\lim_x(p)[i'\backslash x]]\!](\sigma)$$ «

Proof. The claim follows by straightforward induction on the structure of the permission expression p and using Lemma 3.5.7. □

Arbitrarily Small Solutions for Quantifier Elimination. We are now ready to construct the expressions capturing the arbitrarily small solutions. We start with the one for the quantifier elimination.

Definition 3.5.16. » For all simple boolean expressions b with respect to x, we define

$$A_x(b) :\equiv \bigvee_{i=0}^{\varphi_x(b)-1} \lim_x(b)[i\backslash x].$$ «

Example 3.5.17. » We have $A_{\mathsf{i}}(\mathsf{i} \geq 0) \equiv \mathsf{false}$, which correctly captures that $\mathsf{i} \geq 0$ is false for sufficiently small values of i. Conversely, we have

$$A_{\mathsf{i}}(\mathsf{mod}(\mathsf{i}, 2) = 0) \equiv \bigvee_{j=0}^{1} \mathsf{mod}(\mathsf{i}, 2) = 0[j\backslash \mathsf{i}] \equiv \mathsf{mod}(0, 2) = 0 \vee \mathsf{mod}(1, 2) = 0,$$

which is equivalent to true and indicates that $\mathsf{mod}(\mathsf{i}, 2) = 0$ is satisfied by arbitrarily small values of i. «

The following lemma states that our expression $A_x(b)$ captures all arbitrarily small solutions and also does not introduce any spurious solutions.

Lemma 3.5.18. » *For all simple boolean expressions b over x and all states σ, the following statements are true:*

1. *If there are arbitrarily small integers $i \in \mathbb{Z}$ with $[\![b[i\backslash x]]\!](\sigma)$, then $[\![A_x(b)]\!](\sigma)$.*
2. *If $[\![A_x(b)]\!](\sigma)$, then there is an integer $j \in \mathbb{Z}$ such that $[\![b[j\backslash x]]\!](\sigma)$.* «

Proof. We consider an arbitrary simple boolean expression b over x and an arbitrary σ. Moreover, let i_0 denote the constant from Lemma 3.5.7.

To prove the first claim, we assume that there are arbitrarily small integers $i \in \mathbb{Z}$ with $[\![b[i\backslash x]]\!](\sigma)$ and aim to show that $[\![A_x(b)]\!](\sigma)$. To this end, we fix the integer i

to be sufficiently small such that $i \leq i_0$. Moreover, we let $i' \coloneqq i + k \cdot \varphi_x((b))$ for some $k \in \mathbb{Z}$ such that $i' \in \{0, \ldots, \varphi_x(b) - 1\}$. Thus, we have

$$[\![b[i\backslash x]]\!](\sigma) = [\![\lim_x(b)[i\backslash x]]\!](\sigma) = [\![\lim_x(b)[i'\backslash x]]\!](\sigma),$$

where the first equality is justified by Lemma 3.5.7 and the second one follows from Lemma 3.5.13. We observe $[\![\lim_x(b)[i'\backslash x]]\!](\sigma)$ appears as a disjunct of $A_x(b)$ and can therefore conclude that $[\![A_x(b)]\!](\sigma)$ as claimed.

To prove the second claim, we assume that $[\![A_x(b)]\!](\sigma)$ and derive that $[\![b[j\backslash x]]\!](\sigma)$, for some $j \in \mathbb{Z}$. Since we know that at least one disjunct of $A_x(b)$ has to be true, there is an $j' \in \{0, \ldots, \varphi_x(b) - 1\}$ such that $[\![\lim_x(b)[j'\backslash x]]\!](\sigma)$. Now, let $j \coloneqq j' - k \cdot \varphi_x(b)$, for some $k \in \mathbb{Z}$ chosen to be sufficiently large such that $j \leq i_0$. Analogous to above, we get

$$[\![\lim_x(b)[j'\backslash x]]\!](\sigma) = [\![\lim_x(b)[j\backslash x]]\!](\sigma) = [\![b[j\backslash x]]\!](\sigma),$$

which concludes the proof. □

Arbitrarily Small Solutions for Maximum Elimination. Next, we show how to construct the permission expression capturing the arbitrarily small expressions for the maximum elimination. For the sake of easier notation, for any non-empty and *finite* set of expressions $S = \{e_1, \ldots, e_n\}$, we introduce the shorthand

$$\max_{e \in S}\{p[e\backslash x]\} :\equiv \mathsf{max}(p[e_1\backslash x], \mathsf{max}(p[e_2\backslash x], \mathsf{max}(\ldots, p[e_n\backslash x]))).$$

Moreover – since we are dealing with non-negative permission – we can easily extend this notation to empty sets by setting $\mathsf{max}_{e \in S}\{p[e\backslash x]\} :\equiv 0$ in cases where $S = \varnothing$.

Definition 3.5.19. » For all simple permission expressions p over x, we define

$$A_x(p) :\equiv \max_{i \in S}\{\lim_x(p)[i\backslash x]\},$$

where $S \coloneqq \{0, \ldots, \varphi_x(p) - 1\}$. «

Example 3.5.20. » We have $A_{\mathsf{i}}(\mathsf{i} \geq 0\ ?\ 1 : 0) \equiv \mathsf{false}\ ?\ 1 : 0$, which simplifies to 0. «

Lemma 3.5.21. » *Consider an arbitrary simple permission expression p and an arbitrary state σ and let $q \coloneqq \max_{i \in \mathbb{Z}}\{[\![p[i\backslash x]]\!](\sigma)\}$. The following statements are true:*

1. *If there are arbitrarily small integers $i \in \mathbb{Z}$ with $[\![p[i\backslash x]]\!](\sigma) = q$, then we also have $[\![A_x(p)]\!](\sigma) = q$.*
2. *We have $[\![A_x(p)]\!](\sigma) \leq q$.* «

Proof. The proof of this lemma is analogous to the one of Lemma 3.5.18. □

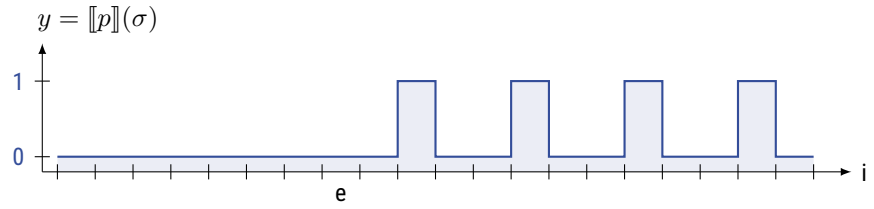


Figure 3.12. A graph showing the permission expression $p :\equiv (\mathsf{mod}(\mathsf{i}, 4) = 0 \wedge \mathsf{e} \leq \mathsf{i})\ ?\ 1 : 0$ plotted as a function of i. The smallest value for i that maximises p is the smallest value above e satisfying $\mathsf{mod}(\mathsf{i}, 3) = 0$.

3.5.3 Smallest Solutions

In the following, we describe how to construct expressions that capture all smallest solutions. Roughly speaking, the idea is to first consider potential smallest solutions ignoring any divisibility constraint and then take into account the divisibility constraints by probing values of the expression at hand for all values of x within one period from any such potential solution.

Example 3.5.22. » As an example, let us consider the maximum expression $\mathsf{max}_{\mathsf{i} \in \mathbb{Z}}\{p\}$, where $p \equiv (\mathsf{mod}(\mathsf{i}, 3) = 0 \wedge \mathsf{e} \leq \mathsf{i})\ ?\ 1 : 0$; a visual representation is shown in Figure 3.12. Ignoring the divisibility constraint $\mathsf{mod}(\mathsf{i}, 3) = 0$, the smallest value for i maximising the permission expression p is e. However, as $\mathsf{mod}(\mathsf{i}, 3) = 0$ might not be true for $\mathsf{i} = \mathsf{e}$, we also have to probe $\mathsf{i} = \mathsf{e} + 1$ and $\mathsf{i} = \mathsf{e} + 2$; this is enough because the pattern of the divisibility constraint's truth value repeats for larger values of i. «

Boundary Expression. Let us consider an arbitrary simple boolean expression b over x and think of it as a the function $f(i) \coloneqq [\![b[i \backslash x]]\!](\sigma)$, for some fixed state σ. We observe that – for increasing values of i and ignoring any potential divisibility constraints appearing in b – there are only *finitely* many values for i where the value of function $f(i)$ switches from false to true (and vice versa, for that matter). Intuitively, the set of these values must also contain the smallest value for x making b true. The following definition shows how we can extract a set of *boundary expressions* from any simple boolean expression b that precisely capture these values; roughly speaking, these boundary expressions can be computed based on a syntactic analysis of b's literals (cf. Figure 3.13).

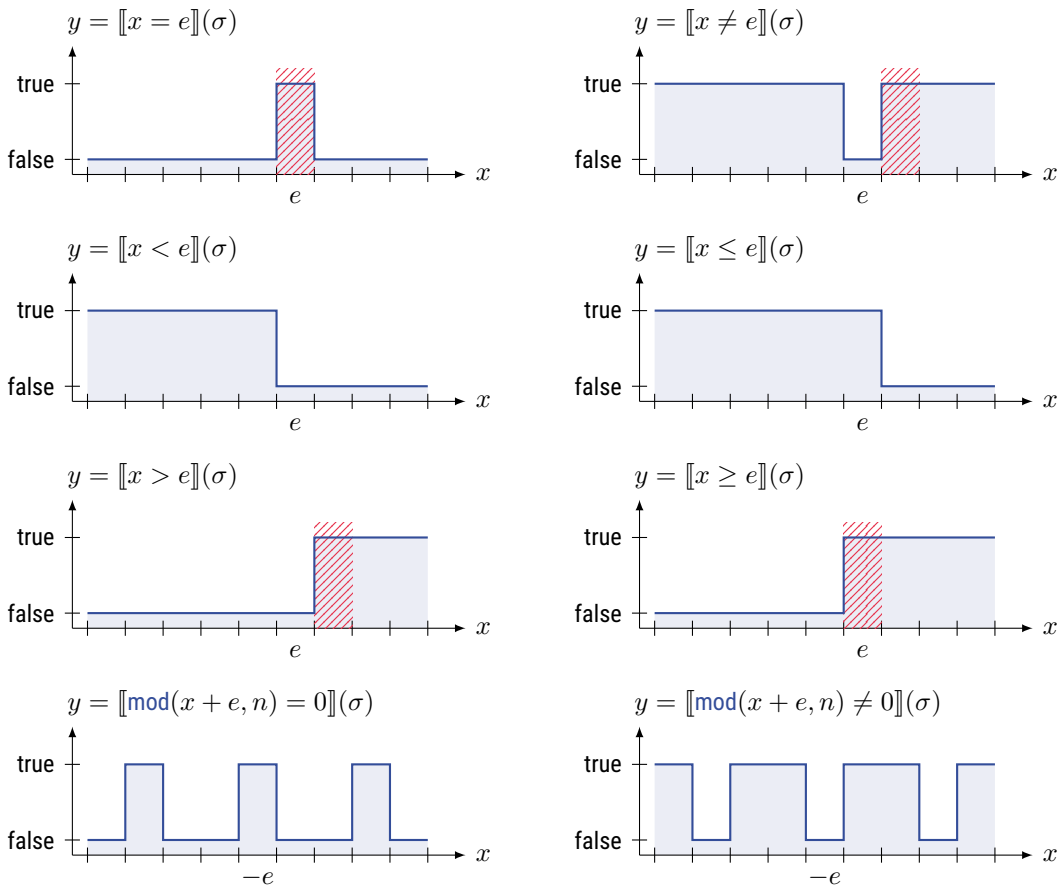


Figure 3.13. All types of literals depending on the variable x that may appear in a simple boolean expression over x. The literals are depicted as a function of x in some fixed state σ. For each type of literal, the corresponding boundary expression – that is, the smallest value for which the expression switches from false to true – is highlighted (▨), if it exists.

Definition 3.5.23. » The set of *boundary expressions* $E_x(b)$ of a simple boolean expression b over x is defined as

$$E_x(b) :\equiv \begin{cases} \{e\} & \text{if } b \equiv x \circ e \text{ and } \circ \in \{=, \geq\} \\ \{e+1\} & \text{if } b \equiv e \text{ and } \circ \in \{\neq, >\} \\ E_x(b_1) \cup E_x(b_2) & \text{if } b \equiv b_1 \circ b_2 \text{ and } \diamond \in \{\wedge, \vee\} \\ \varnothing & \text{otherwise.} \end{cases}$$

«

The following lemma states that the boundary expressions, as defined above, indeed capture the smallest solution; the formulation of that statement is a bit cumbersome

as it also accounts for any potential divisibility constraints by incorporating the period $\varphi_x(b)$.

Lemma 3.5.24. » *For all simple boolean expressions b over x, all states σ, and all integers $s, t \in \mathbb{Z}$ and $k \in \mathbb{N}$ with $t = s + k \cdot \varphi_x(b)$ as well as $\neg[\![b[s\backslash x]]\!](\sigma)$ and $[\![b[t\backslash x]]\!](\sigma)$, there is a pair of a boundary expression $e \in E_x(b)$ and an integer $i \in \{0, \ldots, \varphi_x(b) - 1\}$, such that $[\![e + i]\!](\sigma) = t$.* «

Proof. We prove the statement by structural induction on the simple boolean expression b. To this end, we consider an arbitrary simple expression b and – as our induction hypothesis – assume that the statement holds for all of its sub-expressions. Moreover, we let the state σ be arbitrary and assume that there are integers $s, t \in \mathbb{Z}$, and $k \in \mathbb{N}$ such that

1. $t = s + k \cdot \varphi_x(b)$ as well as
2. $\neg[\![b[s\backslash x]]\!](\sigma)$ and $[\![b[t\backslash x]]\!](\sigma)$.

Throughout this proof, we will refer to these properties as Property 1 and Property 2, respectively. We proceed by distinguishing the following cases:

- Case $x \notin \text{free}(b)$: In this case, the value of $[\![b]\!](\sigma)$ does not depend on x. Thus, we have $[\![b[s\backslash x]]\!](\sigma) = [\![b[t\backslash x]]\!](\sigma)$, which contradictions Property 2. Consequently, there is nothing left to prove in this case.

- Case $b \equiv x \sim e$ for some e and $\sim \in \{=, \neq, \geq, >\}$: For this case, we define

$$e_0 :\equiv \begin{cases} e & \text{if } \sim \in \{=, \geq\} \\ e + 1 & \text{otherwise} \end{cases}$$

  and observe that $E_x(b) = \{e_0\}$. We consider the value $v_0 \coloneqq [\![e_0]\!](\sigma)$ and note that it coincides with the unique point at which the function $f(i) \coloneqq [\![b[i\backslash x]]\!](\sigma)$ switches from false to true. By the assumed Property 2, we have $f(s) = \mathsf{false}$ and $f(t) = \mathsf{true}$. Thus, we have $s < v_0 \leq t$. Since, by Property 1, the integers s and t differ by exactly $k \cdot \varphi_x(b)$, there is an integer $i_0 \in \{0, \ldots, k \cdot \varphi_x(b) - 1\}$ such that $t = v_0 + i_0$. Thus, there is a boundary expression and an integer, namely e_0 and i_0, satisfying $t = [\![e_0 + i]\!](\sigma)$, as claimed.

- Case $b \equiv x \sim e$ for some e and $\sim \in \{\leq, <\}$: Similarly to the first case, we arrive at a contradiction to Property 2.

- Case $b \equiv \mathsf{mod}(x + e, n) \sim 0$ for some e, n and $\sim \in \{=, \neq\}$. We observe that $\varphi_x(b) = n$ and therefore $t = s + k \cdot n$. Thus, we have

$$[\![b[s\backslash x]]\!](\sigma) = [\![b[s + k \cdot n\backslash x]]\!](\sigma) = [\![b[t\backslash x]]\!](\sigma),$$

  which, again, contradicts Property 2.

- Case where $b \equiv b_1 \diamond b_2$, for some $\diamond \in \{\wedge, \vee\}$: This case follows by applying the induction hypothesis to both simple boolean expressions b_1 and b_2 and the observation that, for $i \in \{1, 2\}$, we have $\varphi_x(b) = k_i \cdot \varphi_x(b_i)$, for some $k_i \in \mathbb{N}$. □

Smallest Solutions for Quantifier Elimination. The following definitions shows how our boundary expressions can be used to construct a boolean expression capturing all smallest solutions.

Definition 3.5.25. » For all simple boolean expressions b, we define

$$B_x(b) :\equiv \bigvee_{e \in E_x(b)} \bigvee_{i=0}^{\varphi_x(b)-1} b[e + i\backslash x]$$ «

Example 3.5.26. » Let $b :\equiv \mathsf{i'} \geq 2 \cdot \mathsf{a} \wedge \mathsf{i'} < \mathsf{b} \wedge \mathsf{mod}(\mathsf{i'}, 2) = 0$ be the boolean expression from Example 3.5.2. We observe that $E_{\mathsf{i'}}(b) = \{2{\cdot}\mathsf{a}\}$ and $\varphi_{\mathsf{i'}}(b) = 2$. We have

$$B_{\mathsf{i'}}(b) \equiv b_1 \vee b_2, \quad \text{where } \begin{cases} b_1 \equiv 2 \cdot \mathsf{a} + 0 \geq 2 \cdot \mathsf{a} \wedge 2 \cdot \mathsf{a} + 0 < \mathsf{b} \wedge \mathsf{mod}(2 \cdot \mathsf{a} + 0, 2) = 0 \\ b_2 \equiv 2 \cdot \mathsf{a} + 1 \geq 2 \cdot \mathsf{a} \wedge 2 \cdot \mathsf{a} + 1 < \mathsf{b} \wedge \mathsf{mod}(2 \cdot \mathsf{a} + 1, 2) = 0. \end{cases}$$

We easily see that b_1 simplifies to $2 \cdot \mathsf{a} < \mathsf{b}$, while b_2 simplifies to false. Thus, $\exists \mathsf{i'}\colon b$ has a smallest solution if and only if $2 \cdot \mathsf{a} < \mathsf{b}$. «

Lemma 3.5.27. » *For all simple boolean expressions b over x and all states σ, the following statements are true:*

1. *If there is a smallest integer $i \in \mathbb{Z}$ with $[\![b[i\backslash x]]\!](\sigma)$, then $[\![B_x(b)]\!](\sigma)$.*
2. *If $[\![B_x(b)]\!](\sigma)$, then there is an integer $j \in \mathbb{Z}$ such that $[\![b[j\backslash x]]\!](\sigma)$.* «

Proof. Throughout this proof, we consider an arbitrary simple boolean expression b and an arbitrary state σ.

To prove the first claim, we assume that there is a smallest integer $t \in \mathbb{Z}$ with $[\![b[t\backslash x]]\!](\sigma)$. We define $s \coloneqq t - \varphi_x(b)$ and observe that the integers s and t satisfy the requirements of Lemma 3.5.24. Applying this lemma yields that there is a boundary

expression $e \in E_x(b)$ and an integer $i \in \{0, \ldots, \varphi_x(b) - 1\}$ such that $[\![e + i]\!](\sigma) = t$. Thus, we have $[\![b[e + i \backslash x]]\!](\sigma)$. We observe that $b[e + i \backslash x]$ appears as a disjunct of $B_x(b)$ and can therefore conclude that $[\![B_x(b)]\!](\sigma)$, as claimed.

The second claim trivially follows from the fact that $B_x(b)$ is a disjunction of terms that correspond to restricting the variable x in b to specific values. □

Filtered Boundary Expressions. Next, we discuss an appropriate selection boundary expressions for permission expressions that can be used for our maximum elimination algorithm. Just as for Cooper's algorithm, these boundary expressions must include the *smallest* value of x defining the maximum value in question. The set must be finite, and be as small as possible for efficiency of our overall algorithm. We refine the notion of a boundary expression, and compute a set of pairs $\langle e, b \rangle$ of integer expressions e and its *filter condition* b. Intuitively, the filter condition b represents an additional condition under which e must be included as a boundary expression. In particular, in contexts where b is false, the boundary expression e can be ignored. This gives us a way to symbolically define an ultimately-smaller set of boundary expressions, particularly in the absence of contextual information which might later show b to be false. We call these pairs *filtered boundary expressions*.

Definition 3.5.28. » The set of *filtered boundary expressions* $E_x(p)$ of a simple permission expression p is defined as

$$E_x(p) \coloneqq \begin{cases} \{\langle e, \mathsf{true} \rangle \mid e \in E_x(b)\} & \text{if } p \equiv b \,?\, q : 0 \\ E_x(p_1) \cup E_x(p_2) & \text{if } p \equiv p_1 + p_2 \\ E_x(p_1) \cup \{\langle e, p_1 > 0 \rangle \mid e \in E_x(\neg b)\} & \text{if } p \equiv p_1 - (b \,?\, q : 0) \\ E_x(p_1) \cup \{\langle e, C_x(b, p_1) \rangle \mid e \in E_x(b)\} & \text{if } p \equiv b \,?\, p_1 : 0 \\ E_x(p_1) \cup E_x(p_2) & \text{if } p \equiv \mathsf{min}(p_1, p_2) \text{ or } p \equiv \mathsf{max}(p_1, p_2), \end{cases}$$

where

$$C_x(b, p) :\equiv \bigvee_{\langle e', b' \rangle \in E_x(p)} \bigvee_{i'=0}^{\varphi_x(p)-1} (\neg b \wedge b')[e' + i' \backslash x]. \qquad \text{«}$$

In the definition above, there are three key points which ultimately make our algorithm efficient:

- First, the case for $p \equiv b \,?\, q : 0$ only includes boundary expressions for making b true. The case of b being false is not relevant for trying to maximise the permission expression's value. Note that this follows from the structure of the permission expression: This case will never apply under a subtraction operator,

due to our simplified grammar, and the case for subtraction not recursing into the right-hand operand.

- Second, the case for $p \equiv p_1 - (b ? q : 0)$ dually only considers boundary expressions for making b false – along with the boundary expressions for maximising p_1, of course. The filter condition $p_1 > 0$ is used to drop the boundary expressions for making b false: In cases where p_1 is not strictly positive, we know that the evaluation of the whole permission expression will not yield a strictly-positive value, and hence is not an interesting value for a non-negative maximum.
- Third, in the case for $p \equiv b \,?\, p_1 : 0$, we combine the boundary expressions for p_1 with those for b. We exploit that, if all boundary expressions for p whose filter conditions are true make the condition b true, then we can safely discard the boundary expressions for b.

Lemma 3.5.29. » *For all simple permission expressions p over x, all states σ, and all integers $s, t \in \mathbb{Z}$ and $k \in \mathbb{N}$ with*

- $t = s + k \cdot \varphi_x(p)$ *as well as*
- $[\![p[s\backslash x]]\!](\sigma) < [\![p[t\backslash x]]\!](\sigma)$,

there is a pair of a filtered boundary expression $\langle e, b\rangle \in E_x(p)$ and an integer $i \in \{0, \ldots, \varphi_x(p) - 1\}$, such that $[\![p[t\backslash x]]\!](\sigma) \leq [\![p[e + i\backslash x]]\!](\sigma)$ and $[\![b]\!](\sigma)$. «

Proof. We prove the statement by induction on the structure of the simple permission expressions p. To this end, let us consider an arbitrary simple permissions expression p over x and suppose that the claim holds for all of its sub-expressions. Moreover, let the state σ be arbitrary and assume that there are integers $s, t \in \mathbb{Z}$ and $k \in \mathbb{N}$ such that

- $t = s + k \cdot \varphi_x(p)$ as well as
- $[\![p[s\backslash x]]\!](\sigma) < [\![p[t\backslash x]]\!](\sigma)$.

Throughout this proof, we will refer to these properties as Property 1 and Property 2, respectively. We proceed by distinguishing the following cases

- Case $p \equiv b \,?\, q : 0$. We observe that we must have $\neg[\![b[s\backslash x]]\!](\sigma)$ and $[\![b[t\backslash x]]\!](\sigma)$. Thus, we can apply Lemma 3.5.24 to obtain that there is a boundary expression $e \in E_x(b)$ and an integer $i \in \{0, \ldots, \varphi_x(b) - 1\}$ such that $[\![e + i]\!](\sigma) = t$. We observe that $\langle e, \mathsf{true}\rangle \in E_x(p)$ and $\varphi_x(p) = \varphi_x(b)$. Thus, for this case, the claim follows from
$$[\![p[t\backslash x]]\!](\sigma) = q = [\![p[e + i\backslash x]]\!](\sigma),$$

where the first equality holds by Property 2.

- Case $p \equiv p_1 + p_2$. We observe that Property 2 must be satisfied by at least one of the sub-expressions p_1 or p_2. Thus, this case follows by applying the induction hypothesis to p_1 and p_2 and the observation that, for $i \in \{1, 2\}$, we have $\varphi_x(p) = k_i \cdot \varphi_x(p_i)$, for some $k_i \in \mathbb{N}$.

- Case $p \equiv p_1 - (b \,?\, q : 0)$. Here, we observe that we must have $\neg[\![b[s\backslash x]]\!](\sigma)$ and $[\![b[t\backslash x]]\!](\sigma)$ or, equivalently $\neg[\![(\neg b)[s\backslash x]]\!](\sigma)$ and $[\![(\neg b)[t\backslash x]]\!](\sigma)$. Thus, similarly to the first claim, we can apply Lemma 3.5.24 to arrive at the claim, similarly to the first case. Note that, as explained above, the filter condition $p > 1$ is guaranteed to be true for all boundary expressions of interest.

- Case $p \equiv b \,?\, p_1 : 0$. In this case, we must have $\neg[\![b[s\backslash x]]\!](\sigma)$ and $[\![b[t\backslash x]]\!](\sigma)$ or $[\![p_1[s\backslash x]]\!](\sigma) < [\![p_2[t\backslash x]]\!](\sigma)$, or both. Thus, the claim follows by using Lemma 3.5.24 or applying the induction hypothesis to p_1. Note that if our filter condition $C_x(b, p_1)$ is true, then we know for sure that $[\![p_1[s\backslash x]]\!](\sigma) < [\![p_2[t\backslash x]]\!](\sigma)$ is true; that is the claim follows from the induction hypothesis applied to p_1 alone, which makes it safe to discard the boundary expressions corresponding to the condition b.

- Case $p \equiv \mathsf{min}(p_1, p_2)$ and $p \equiv \mathsf{max}(p_1, p_2)$. Analogous to the case $p \equiv p_1 + p_2$. □

Smallest Solutions for Maximum Elimination. Using our filtered boundary expressions introduced above, we can define a permission expression capturing smallest solutions of the maximum elimination problem at hand.

Definition 3.5.30. » For all simple permission expressions p over x, we define

$$B_x(p) :\equiv \max_{\langle e,b\rangle \in E_x(p)} \left\{ b \,?\, \left(\max_{i \in \{0,\ldots,\varphi_x(p)-1\}} \{p[e + i\backslash x]\} \right) : 0 \right\},$$

where $S \coloneqq \{0, \ldots, \varphi_x(p) - 1\}$. «

Lemma 3.5.31. » *Consider an arbitrary simple permission expression p over x and an arbitrary state σ and let $q \coloneqq \max_{i \in \mathbb{Z}}\{[\![p[i\backslash x]]\!](\sigma)\}$. The following statements are true:*

1. *If there is a smallest integer $i \in \mathbb{Z}$ with $[\![p[i\backslash x]]\!](\sigma) = q$, then $[\![B_x(p)]\!](\sigma) = q$.*
2. *We have $[\![B_x(p)]\!](\sigma) \leq q$.* «

Proof. Throughout this proof, we consider an arbitrary simple permission expression p over x and an arbitrary state σ and let $q \coloneqq \max_{i\in\mathbb{Z}}\{[\![p[i\backslash x]]\!](\sigma)\}$. For the sake of easier argumentation, we prove the statement for

$$B'_x(p) :\equiv \max_{\langle e,b\rangle\in E_x(p)} \left\{ \max_{i\in S}\{b\,?\,(p[e+i\backslash x]) : 0\} \right\},$$

which can be easily obtained from $B_x(x)$ by pushing the condition b into the inner maximum expression and is, therefore, equivalent.

We start by proving the second claim. To this end, let us consider an arbitrary filtered boundary expression $\langle e'', b''\rangle \in E_x(b)$ and integer $i'' \in \{0,\dots,\varphi_x(p)-1\}$. We observe that

$$[\![b''\,?\,p[e''+i''\backslash x] : 0]\!](\sigma) \leq [\![p[e'', i''\backslash x]]\!](\sigma) \leq \max_{i\in\mathbb{Z}} [\![p[i\backslash x]]\!](\sigma) = q.$$

From this, the claim immediately follows, since the inequality holds for *all* filtered boundary expressions $\langle e'', b''\rangle$ and integers i'', it must also hold for the one maximising $B'_x(p)$ and, thus, we have $B'_x(p) \leq q$

To prove the first claim, we assume that there is a smallest integer $t \in \mathbb{Z}$ with $[\![p[t\backslash x]]\!](\sigma) = q$. We define $s \coloneqq t - \varphi_x(p)$ and observe that $[\![p[s\backslash x]]\!](\sigma) < q$. Thus, the integers s and t satisfy all requirements of Lemma 3.5.29. Applying this lemma yields that there is a filtered boundary expression $\langle e', b'\rangle \in E_x(p)$ and an integer $i' \in \{0,\dots,\varphi_x(p)-1\}$ such that $[\![e'+i']\!](\sigma) = t$ and $[\![b']\!](\sigma)$. We observe that the expression $b'\,?\,p[e'+i'\backslash x] : 0$ appears as a sub-term of $B'_x(p)$. Thus, we have

$$[\![B_x(p)]\!](\sigma) \geq [\![b'\,?\,p[e'+i'\backslash x] : 0]\!](\sigma) = [\![p[e'+i'\backslash x]]\!](\sigma) = [\![p[t\backslash x]]\!](\sigma) = q.$$

Together with $B'_x(p) \leq q$ from the second claim, we get $B'_x(p) = q$, as required. □

To see how our filter conditions help to keep the result of the maximum elimination small, we consider a simple yet illustrative example.

Example 3.5.32. » Let us consider the pointwise maximum expression

$$\max_{\mathsf{i}\,|\,0\leq\mathsf{i}} \{\mathsf{q}_i = \mathsf{i}\,?\,1 : 0\}.$$

that is, we have $p \equiv 0 \leq \mathsf{i}\,?\,(\mathsf{q}_i = \mathsf{i}\,?\,1 : 0)$. Evaluating the filtered boundary expressions for p yields $E_{\mathsf{i}}(p) = \{\langle 0, \mathsf{q}_i < 0\rangle, \langle \mathsf{q}_i, \mathsf{true}\rangle\}$ with the meaning that the boundary expression q_i originating from the inner condition $\mathsf{q}_i = \mathsf{i}$ has to be considered in all cases, while the boundary expression 0 originating from the outer condition $0 \leq \mathsf{i}$ is only if interest if $\mathsf{i} < 0$. With this, we get

$$B_{\mathsf{i}}(p) \equiv \max(p_1, p_2), \qquad \text{where} \begin{cases} p_1 :\equiv \mathsf{q}_i < 0\,?\,(0 \leq 0\,?\,(\mathsf{q}_i = 0\,?\,1 : 0)) \\ p_2 :\equiv \mathsf{true}\,?\,(0 \leq \mathsf{q}_i\,?\,(\mathsf{q}_i = \mathsf{q}_i\,?\,1 : 0)). \end{cases}$$

We observe that the term p_1 corresponding to the boundary expression 0 can be simplified to 0 since it contains the two contradictory conditions $\mathsf{q}_i < 0$ and $\mathsf{q}_i = 0$. Thus, the entire expression $B_i(p)$ can be simplified to $0 \leq \mathsf{q}_i \,?\, 1 : 0$. Note that, without the filter conditions, the simplified result would be $\max((\mathsf{q}_i = 0 \,?\, 1 : 0), (0 \leq \mathsf{q}_i \,?\, 1 : 0))$ instead. «

As the example above illustrates, in the context of our permission analysis, the filter conditions allow us to disregard boundary expressions corresponding, for example, to the integer loop invariants, provided that the expressions generated by analysing permission expression stemming from analysing the loop body already suffice. For our evaluation, we employed aggressive syntactic simplifications of the resulting expressions, in order to exploit these filter conditions to produce a succinct final permission expression.

3.5.4 Combining all Solutions

We now describe how to combine the expressions capturing all arbitrarily small solutions with the expression capturing all smallest solutions to obtain a final expression capturing all solutions.

Quantifier-Free Boolean Expressions. For the quantifier elimination algorithm, this final quantifier-free expression equivalent to $\exists x\colon b$ is obtained by simply disjoining $A_x(b)$ and $B_x(b)$.

Definition 3.5.33. » For all simple boolean expressions b, we define

$$F_x(b) :\equiv A_x(b) \vee B_x(b).$$ «

Lemma 3.5.34. » *For all simple boolean expressions b and all states σ, we have*

$$\exists i \in \mathbb{Z}\colon [\![b[i\backslash x]]\!](\sigma) \quad \Leftrightarrow \quad [\![F_x(b)]\!](\sigma).$$ «

Proof. The claim follows by combining Lemma 3.5.18 and Lemma 3.5.27. □

Permission Expressions without Pointwise Maxima. Similarly, our maximum elimination algorithm computes the final expression capturing the maximum $\max_{x\in\mathbb{Z}}\{p\}$ by constructing the maximum of $A_x(p)$ and $B_x(p)$.

Definition 3.5.35. » For all simple permission expressions p over x, we define

$$F_x(p) :\equiv \max(A_x(p), B_x(p))$$ «

Theorem 3.5.36. » *For all simple permission expressions p over x and all states σ, we have*

$$[\![F_x(p)]\!](\sigma) = \max_{i \in \mathbb{Z}} \{[\![p[i \backslash x]]\!](\sigma)\}. \quad «$$

Proof. The claim follows by combining Lemma 3.5.21 and Lemma 3.5.31. □

Revisiting our Motivating Example. The definition and theorem above complete the elaborations of our maximum elimination algorithm. With this, we have described all components of our permission inference for array programs. To wrap the description of our inference technique, let us briefly revisit the copy_even_a method shown in Listing 3.1 and already discussed in Example 3.1.1. We recall that this method's body contains the loop $l \equiv \mathsf{while}\ (\mathsf{i} < \mathsf{len}(\mathsf{a}))\ \{c_0\}$, where

$$c_0 :\equiv \mathsf{if}\ (\mathsf{mod}(\mathsf{i}, 2) = 0)\ \{\mathsf{v} := \mathsf{a}[\mathsf{i}]\}\ \mathsf{else}\ \{\mathsf{a}[\mathsf{i}] := \mathsf{v}\}\ ;\ \mathsf{i} := \mathsf{i} + 1$$

Our permission analysis first analyses the loop's body; the rules for loop-free code yield the following permission precondition and difference.

$$\tau\{\!|c_0|\!\}(0) \equiv \mathsf{mod}(\mathsf{i}, 2) = 0\ ?\ ((\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ \mathsf{rd} : 0) : ((\mathsf{q}_a = \mathsf{a} \wedge \mathsf{q}_i = \mathsf{i})\ ?\ 1 : 0)$$
$$\delta\{\!|c_0|\!\}(0) \equiv \mathsf{mod}(\mathsf{i}, 2) = 0\ ?\ 0 : 0$$

The latter can be simplified to 0 and indicates that the loop does not gain or lose any permissions. Using the numerical loop invariant $I^+ \equiv 0 \leq \mathsf{i}$ obtained from an off-the-shelf analysis, our analysis expresses the permissions required by the loop by the maximum expression $\mathsf{max}_{\mathsf{i} \mid 0 \leq \mathsf{i} \wedge \mathsf{i} < \mathsf{len}(\mathsf{a})}\{\lceil\!\lceil \tau\{\!|c_0|\!\}(0) \rceil\!\rceil\}$; the terms capturing the permissions required for the code that comes after the loop are omitted here, as they can also be simplified to 0. In a next step, the analysis then applies our maximum elimination algorithm to aforementioned maximum expression, which – after also applying syntactic simplification rules – yields the permission expression

$$p \equiv (\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}))\ ?\ (\mathsf{mod}(\mathsf{q}_i, 2) = 0\ ?\ \mathsf{rd} : 1) : 0\,.$$

As the loop does not gain or lose any permissions and the only statement before the loop l is $\mathsf{i} := 0$, the method copy_even_a's permission precondition and postcondition as well as the permission invariant for the loop l are all captured by p.

3.6 Evaluation

We developed a prototype implementation of our approach, which we evaluated on a benchmark containing examples from existing papers and competitions.

Section Outline. This section presents our experimental evaluation and is structured as follows. In Section 3.6.1, we first briefly discuss our implementation. In Section 3.6.2, we then describe our benchmark and discuss the experimental results.

3.6.1 Implementation

We have developed a prototype implementation of our inference.[3] This tool was developed as an artifact for the paper [38] this chapter is based on and is limited to permission preconditions and postconditions. An extension of the tool that also infers loop invariants was developed by Becker; for more details on this version of the tool, we refer to his Master's thesis [10].

Our tool is written in Scala and accepts programs written in the Viper language [83]. Given a Viper program, the tool first performs a forward numerical analysis to infer over-approximating numerical loop invariants needed for our handling of loops. The implementation is parametric in the numerical abstract domain used for the analysis; we currently support the abstract domains provided by the Apron library [60]. As we have yet to integrate the implementation of under-approximate invariants (for example [77]), we rely on user provided invariants, or assume them to be `false` if none are provided. In a second step, our tool performs the inference and maximum elimination. Finally, it annotates the input program with the inferred specifications.

3.6.2 Experimental Results

We evaluated our implementation on 43 programs taken from various sources; included are all programs that do not contain strings from the memory safety category of SV-COMP 2017, all programs from Dillig et al. [35] (except three examples involving arrays of arrays), loop parallelisation examples from VerCors [14], and a few programs that we crafted ourselves. We manually checked that our soundness condition holds for all considered programs. The parallel loop examples were encoded as two consecutive loops where the first one models the forking of one thread per loop iteration – by iteratively exhaling the permissions required for all loop iterations – and the second one models the joining of all these threads – by inhaling the permissions that are left after each loop iteration again. For the numerical analysis we used the *polyhedra abstract domain* proved by Apron.

An overview of our experimental results is given in Table 3.14. The experiments were performed on a dual core machine with a 2.60 GHz Intel Core i7-6600U CPU and 16 GB of RAM, running Ubuntu 16.04. The running times were measured by

3: https://github.com/viperproject/sample/tree/master/sample_qp

	Lines	Loops	Size	Precision	Time [ms]
add_last.vpr	12	1 (1)	1.9	✓	21
append.vpr	13	1 (1)	1.9	✓	32
array_1.vpr	17	2 (2)	0.9	✗	28
array_2.vpr	23	3 (2)	0.9	✗	35
array_3.vpr	23	2 (2)	1.1	✓	24
array_reverse.vpr	18	1 (1)	3.2	✓✓	28
bubble_sort.vpr	23	2 (2)	1.8	✓✓	34
copy.vpr	16	2 (1)	1.6	✓	27
copy_even_a.vpr	17	1 (1)	1.6	✓	27
copy_even_b.vpr	14	1 (1)	2.2	✓✓	23
copy_even_c.vpr	14	1 (1)	1.4	✗	20
copy_odd.vpr	21	2 (1)	2.4	✓	55
copy_odd_bug.vpr	19	2 (1)	7.1	✓	57
copy_part.vpr	17	2 (1)	1.7	✓	30
count_down.vpr	21	3 (2)	1.1	✓	32
diff.vpr	31	2 (2)	2.0	✗	70
find.vpr	19	1 (1)	3.0	✓	43
find_non_null.vpr	19	1 (1)	3.0	✓	40
init.vpr	18	2 (1)	1.1	✓	28
init_2d.vpr	23	2 (2)	2.1	✓	52
init_even.vpr	18	2 (1)	0.9	✗	26
init_even_bug.vpr	18	2 (1)	1.5	✗	28
init_non_const.vpr	18	2 (1)	1.1	✓	27
init_part.vpr	19	2 (1)	1.1	✓	30

	Lines	Loops	Size	Precision	Time [ms]
init_part_bug.vpr	19	2 (1)	1.5	✓	31
insertion_sort.vpr	21	2 (2)	2.5	✓✓	35
java_bubble.vpr	24	2 (2)	2.3	✓✓	32
knapsack.vpr	21	2 (2)	1.3	✗	45
lis.vpr	37	4 (2)	4.2	✓	73
matrix_mult.vpr	33	3 (3)	1.5	✓	78
merge_inter.vpr	23	2 (1)	3.4	✗	56
merge_inter_bug.vpr	23	2 (1)	2.6	✗	59
memcopy.vpr	16	2 (1)	1.6	✓	28
multarray.vpr	26	2 (2)	2.1	✓	40
par_array.vpr	20	2 (1)	1.2	✓	30
par_copy.vpr	20	2 (1)	1.2	✓	31
par_copy_even.vpr	22	2 (1)	5.0	✓✓	79
par_matrix.vpr	35	4 (2)	1.1	✓	80
par_nested.vpr	31	4 (2)	0.5	✗	57
relax.vpr	33	1 (1)	1.4	✓✓	55
reverse.vpr	21	2 (1)	3.9	✓	42
reverse_bug.vpr	21	2 (1)	1.7	✓	42
sanfoundry.vpr	27	2 (1)	2.1	✓	37
selection_sort.vpr	26	2 (2)	1.0	✗	38
string_copy.vpr	16	2 (1)	0.9	✗	21
string_length.vpr	10	1 (1)	0.8	✗	15
swap.vpr	15	1 (1)	1.5	✓	19
swap_bug.vpr	15	1 (1)	1.5	✓	19

Table 3.14. The experimental results. For each program, we list the lines of code and the number of loops (in brackets their nesting depth). We report the relative size of the inferred specifications compared to hand-written specifications, and whether the inferred specifications are equally precise ✓ or not ✗ (a double tick indicates slightly more precise than hand-written specifications). All inference times are given in milliseconds.

```
method copy_even_c(a: Int[]) {
  var i: Int
  var v: Int
  i := 0
  while (i + 1 < len(a)) {
    v := a[i]
    a[i+1] := v
    i := i + 2
  }
}
```

Listing 3.15. Yet another variation of the method copying array elements at even indices.

first running the analysis 50 times to warm up the JVM and then computing the average time needed over the next 100 runs. The results show that the inference is very efficient.

Precision. In 35 out of 48 cases, our inference inferred precise specifications. In one example, namely knapsack.vpr, the source of imprecision is our abstraction of array-dependent conditions. All other imprecisions are due to the inferred numerical loop invariants; in all these cases, manually strengthening the invariants yields a precise specification: As an example, let us consider copy_even_c.vpr, which is yet another variation of the method copying array elements at even indices shown in Listing 3.15. The numerical invariant obtained for the loop in this method is $0 \leq \mathsf{i}$ and does *not* capture that the index i is always even. As a result, the inferred specifications require write permission for all but the first array element. However, manually annotating the loop with the additional constraint $\mathsf{mod}(\mathsf{i}, 2) = 0$ allowed our analysis to generate specifications that only required write permission for elements at even indices.

Conciseness. For each program, we compared the size and precision of the inferred specifications with respect to hand-written ones. Thanks to aggressive syntactic simplification, the inferred specifications are concise for the vast majority of the examples.

An example for which the inferred specifications are roughly twice the size compared to hand-written ones is copy_even_b.vpr, for which we have already seen the inferred specifications in Example 3.3.7. For convenience, we repeat the inferred loop

precondition (which is simultaneously also the method precondition), which is given $\mathsf{max}(p_1, p_2)$, where

$$p_1 \equiv (\mathsf{mod}(\mathsf{q}_i, 2) = 1 \wedge 1 < \mathsf{q}_i \wedge \mathsf{q}_i < 2 \cdot (\mathsf{len}(\mathsf{a}) - 1) \,/\, 2 + 1) \,?\, 1 : 0$$
$$p_2 \equiv (\mathsf{mod}(\mathsf{q}_i, 2) = 0 \wedge 0 < \mathsf{q}_i \wedge \mathsf{q}_i < 2 \cdot (\mathsf{len}(\mathsf{a}) - 1) \,/\, 2) \,?\, \mathsf{rd} : 0\,.$$

We see that the two terms p_1 and p_2 share a lot of similar conditions that would not be repeated by hand-written specifications. A more concise specification, for example, would be

$$(0 < \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a})) \,?\, (\mathsf{mod}(\mathsf{q}_i, 2) = 0 \,?\, \mathsf{rd} : 1) : 0$$

Note, however, that this specification is slightly less precise, as it always requires permission for the last array element, which the inferred specifications shown above does not in cases where the array is of odd length.

Loop Invariants. As mentioned, the version of the tool used for our experiments above does not infer permission invariants; instead, it infers loop preconditions and postconditions. Recall that, as long as the loop at hand does not gain or lose permissions, these are identical and – at the same time – also loop invariants. The only programs for which this is not the case are the five parallel loop examples (their names are all prefixed with par_). The evaluation conducted by Becker as part of his thesis [10] shows that our technique described in Section 3.4 produces suitable permission invariants for all them. As described above, all these examples follow the (also otherwise common) pattern, where there are two loops, one exhaling permissions and one inhaling the permissions again. Below, we show the permission invariants inferred for such a pair of loops in their plainest incarnation, given by the statements c_1 and c_2, where

$$c_1 :\equiv \mathsf{i} := 0\,;\, \mathsf{while}\ (\mathsf{i} < \mathsf{len}(\mathsf{a}))\ \{\mathsf{exhale}\ \mathsf{acc}(\mathsf{a}[\mathsf{i}])\,;\, \mathsf{i} := \mathsf{i} + 1\}$$
$$c_2 :\equiv \mathsf{i} := 0\,;\, \mathsf{while}\ (\mathsf{i} < \mathsf{len}(\mathsf{a}))\ \{\mathsf{inhale}\ \mathsf{acc}(\mathsf{a}[\mathsf{i}])\,;\, \mathsf{i} := \mathsf{i} + 1\}.$$

The permission invariant for the loop in c_1 iteratively giving away permission to all elements of the array at hand is given by

$$(\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a})) \,?\, (\mathsf{q}_i < \mathsf{i} \,?\, 0 : 1) : 0\,.$$

Note that we have already seen this permission invariant as part of Example 3.4.7; The permission invariant inferred for the loop in c_2 that then iteratively regains all lost permissions is given by

$$(\mathsf{q}_a = \mathsf{a} \wedge 0 \leq \mathsf{q}_i \wedge \mathsf{q}_i < \mathsf{len}(\mathsf{a}) \wedge \mathsf{q}_i < \mathsf{i}\,) \,?\, 1 \,?\, 0\,.$$

3.7 Discussion

We conclude this section by discussing related work, strengths and limitations of our approach, and possible future directions.

Section Outline. In Section 3.7.1, we first compare our technique with related work. In Section 3.7.2, we then summarise the main strengths and limitations of our approach. Finally, in Section 3.7.3, we lay out some possible future directions.

3.7.1 Related Work

Much work has been dedicated to the analysis of array programs, but most of it focuses on array content, whereas we infer permission specifications. The simplest approach to analyse such array manipulating programs is to summarise all array elements into one single abstract memory location [13]. This is generally quite imprecise, as only weak updates can be performed. A simple alternative is to consider array elements as distinct variables [13], which is feasible only when the length of the array is statically known. More advanced approaches perform syntax-based [49, 56, 61], semantics-based [29, 74] or symbolic [55] partitioning of an array into segments – with the exception of [74] – and do not easily generalise to multidimensional arrays, unlike our approach. Gulwani et al. [51] propose an approach for inferring quantified invariants for arrays by lifting quantifier-free abstract domains. Their technique requires user-defined templates for the invariants.

Dillig et al. [35] avoid an explicit array partitioning by maintaining constraints that over- and under-approximate the array elements being updated by a program statement. Their work employs a technique for directly generalising the analysis of a single loop iteration – based on quantifier elimination – which works well when different loop iterations write to disjoint array locations. Blom et al. [14] provide a specification technique for a variety of parallel loop constructs; our work can infer specifications which their technique requires to be provided.

Another alternative for generalising the effect of a loop iteration is to use a first order theorem prover as proposed by Kovács and Voronkov [65]. In their work, however, they did not consider nested loops or multidimensional arrays. Other works rely on loop acceleration techniques [3, 17]. In particular, the work of Bozga et al. [17] does not synthesise loop invariants; they directly infer postconditions of loops with respect to given precondition, while we additionally infer the preconditions. The acceleration technique proposed in [3] is used for the verification of array programs in the tool Booster [4].

Monniaux and Gonnord [78] describe an approach for the verification of array programs via a transformation to array-free Horn clauses. Chakraborty et al. [25] use heuristics to determine the array accesses performed by a loop iteration and split the verification of an array invariant accordingly. Their non-interference condition between loop iterations is similar, but stronger than our soundness condition (cf. Section 3.3). Neither work is concerned with specification inference.

A wide range of static analyses and shape analyses employ tailored separation logics as abstract domain (for example[12, 22, 50, 66, 93]); these works handle recursively-defined data structures, but no random access data structures, such as arrays and matrices. Of these works, Gulavani et al. [50] is perhaps closest to ours: they employ an integer-indexed domain for describing recursive data structures. It would be interesting to combine our work with such separation logic shape analyses. The problem of automating bi-abduction entailment checking for array-based separation logics have been recently studied by Brotherston et al. [21] and Kimura et al. [64], but has not yet been extended to handle loop-based or recursive programs.

3.7.2 Strengths and Limitations

We now discuss some strengths and limitations of our technique.

Strengths. Our technique is very efficient and produces precise specifications that are reasonably concise and human readable. In particular, the inferred specifications are suitable for automated tools, such as program verifiers, but they can also be used for other applications: For instance, the specifications can be leveraged to automatically construct a condition that expresses whether an iteration of a loop writes to a location read by other loop iterations, which can be automatically be checked using an SMT solver in order to determine whether the loop can safely be parallelised.

By being based on a purely syntactic analysis for loop-free code, our approach inherently supports complex conditions appearing in the input program, without the need for a sophisticated abstract domain as used, for example, in abstract interpretation. Imprecisions are only introduced – if at all – when results for individual loop iterations are generalised to multiple iterations. As we have demonstrated with our evaluation, these imprecisions typically originate from the precursory numerical analysis and can be solved by either switching to a more precise numerical analysis or by manually strengthening the loop invariants.

Moreover, our inference depends only on widely available numerical analyses. Even the progressive invariants – used to distinguish between past and future loop

iterations and leveraged for our permission invariants – can be obtained by running an off-the-shelf analysis on a transformed version of the input program.

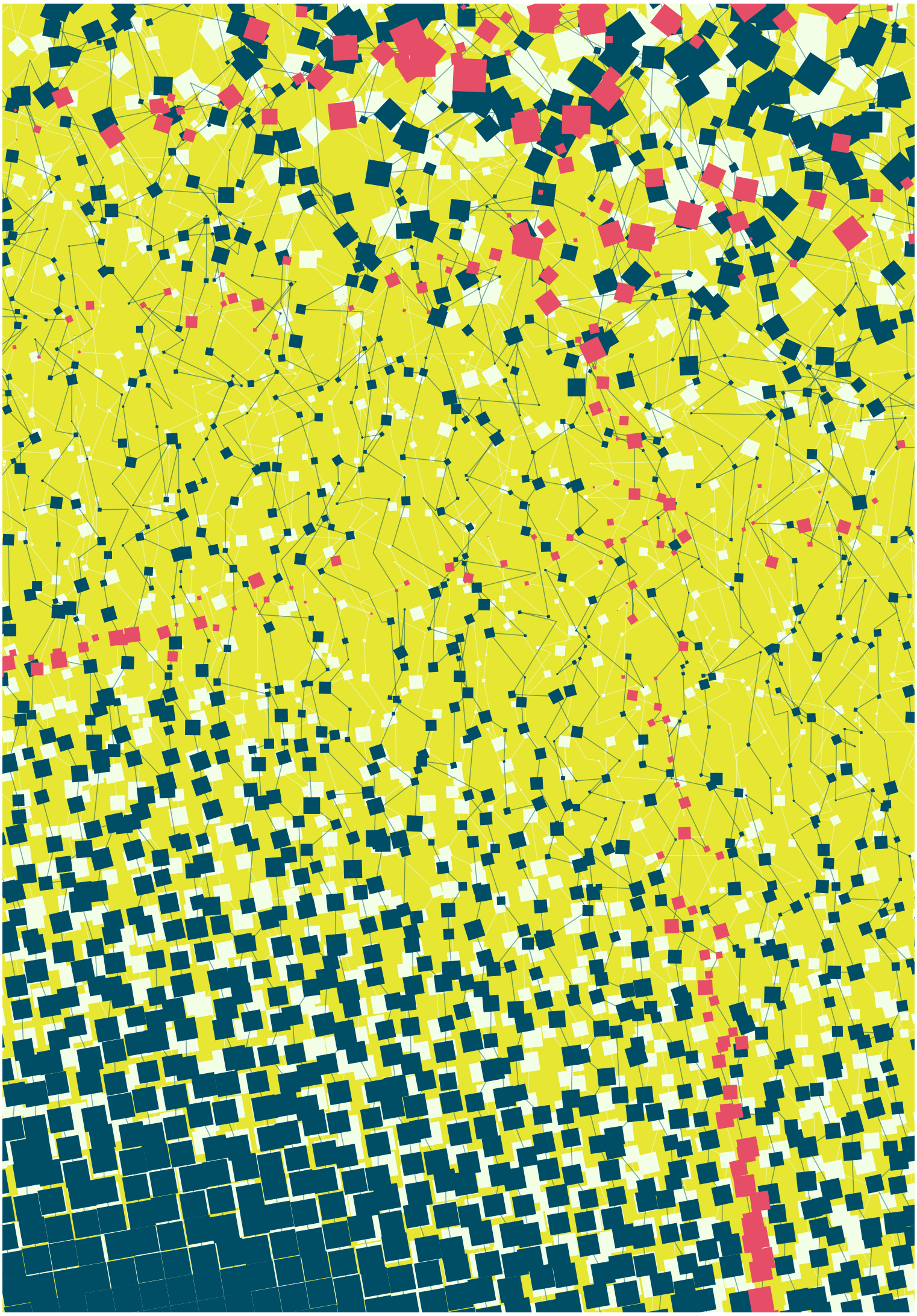
Limitations. A limitation of our approach is that our postconditions cannot capture situations in which a statement obtains permissions to locations for which no pre-state expression exists; for example, postconditions cannot capture permissions obtained through the allocation of new arrays. Moreover, while it would not be hard to support non-recursive method calls, it would be non-trivial to extend our approach to also support recursive method calls. In Section 3.7.3 below, we briefly outline how these limitations could be addressed.

Although our inferred specifications are proven sound, they are not guaranteed to be automatically verifiable. To see why, recall that our permission expressions range over all arrays and indices; therefore, their respective assertion can, in general, only be expressed using quantifiers (which makes verification undecidable). Typically, SMT solvers – and, by extension, also program verifiers – require triggers to handle quantifiers [33].

3.7.3 Future Directions

One possible future direction would be to support richer control flow, for example, by extending the technique to an interprocedural analysis. As already indicated, an extension to programs with no recursion is rather straightforward: Method calls can be handled by applying our analysis bottom-up in the acyclic call graphs and modelling the permission transfer of method calls using inhale and exhale statements. Supporting recursion, however, seems significantly more challenging as, for example, permission preconditions for recursive methods depend on the permissions required by themselves. A conceivable approach could be to introduce symbolic placeholders for such specifications, use an adaption of our technique to infer constraints on them, and then use some kind of solver to synthesise a suitable permission expression.

In order to make our inference support memory allocation, the backwards analysis used to infer the permission differences could be rephrased as a forward analysis; such an analysis would likely have similarities with a strongest postcondition calculus and introduce existential quantifiers for assignments. These existentials would either have to be supported by our maximum elimination algorithm or be solved in another way.



4 Black-Box Learning

In the previous chapter, we have described a permission inference for the important class of array-manipulating programs. Another important and complementary class of heap-manipulating programs operates on linked data structures. Many of these data structures – for example, list-like and tree-like structures – exhibit sufficient regularity to admit a recursive definition. In this chapter, we present a novel learning-based permission inference that is targeted at programs manipulating such recursively defined data structures.

The Problem. Our goal is to infer permission specifications that can be used directly in a standard verification workflow. To make this possible, the inference needs to satisfy the following main requirements:

- *Comprehensive specifications*: The inference provides all permission specifications required by a verifier. This includes all method preconditions and postconditions, loop invariants, but also the predicate definitions of any recursively defined data structures at hand.

  Moreover, to prevent indefinite unrolling of recursive predicate definitions, most automated verifiers use an *isorecursive* semantics [100] that distinguishes between a predicate instance and its body. These verifiers require additional guidance to manipulate predicate instances, which is typically provided via ghost code consisting of designated *unfold* and *fold* statements that exchange a predicate instance with its body and vice versa, and *lemma applications* in cases where an inductive argument is required. To ensure that the inferred specifications can readily be consumed by an isorecursive verifier, they have to include all the required ghost operations.

- *Wide applicability*: The inference handles a large spectrum of data structures, including recursive structures, acyclic structures, and structure with back-pointers. Moreover, it also handles a wide range of control structures, such as iteration and mutual recursion. Moreover, in order to support modular verification, it should itself be able to operate on individual modules; for example, infer specifications for a collection of method without having its clients.

- *Specification process*: The inferred permission specifications must be concise and readable, such that they are amenable for programmers to extend them to also express functional properties. Moreover, the inference should handle programs with partial specifications, for instance because the code at hand calls into already-specified libraries, or because the programmer provides initial specifications to guide the inference.

Our Approach. As already indicated in Chapter 1, our approach is based on the ICE framework [47], where a *teacher* and a *learner* work in tandem to iteratively infer specifications. Roughly speaking, in every iteration, the learner proposes candidate specifications, collectively called *hypothesis*, for the input program.

The teacher employs a verifier to check whether this hypothesis is valid, that is, checks whether it provides permissions with which the input program can be verified. If the verification succeeds, the inference terminates successfully and returns the current hypothesis. Otherwise, the teacher produces a *sample* that characterises the missing permission responsible for the verification failure and communicates it to the learner. Our teacher uses the verifier as an oracle for the program semantics. That is, it extracts all information needed by the learner from the verification failure and the verifier's counterexample. This includes the memory location to which the permission is missing and the failing program trace; our samples comprise both.

The learner combines the current and all previous samples and proposes a hypothesis for the next iteration. This hypothesis is produced by first generating a parameterised template and then finding suitable parameters using an SMT solver.

Our approach satisfies the three requirements stated above:

- *Comprehensive specifications*: Our samples are defined over traces rather than individual states, which enables us to infer *all* specifications for a program simultaneously, including predicate definitions and the ghost code and lemmas required by isorecursive verifiers.

- *Wide applicability*: The specification templates our learner synthesises from the samples are general enough to capture linked list and tree-like structures. Our inference supports both iteration and (mutual) recursion. It is modular, that is, can handle libraries without requiring client code.

- *Specification process*: Driven by verification errors, our inference gradually extends specifications, until the verification succeeds. This approach does not formally guarantee minimal specifications, but in practice produces concise and human-readable specifications, as demonstrated by our evaluation in Section 4.6. Moreover, it also supports partial specifications.

Contributions. We make the following technical contributions:

1. We present a novel black-box inference for permission specifications. Our samples summarise the permission constraints of entire traces rather than individual states, which allows us to generate the specifications for an entire program, by simultaneously inferring method preconditions and postconditions, loop invariants, and recursive predicate definitions.

2. We extend the inference to automatically generate all ghost code required to manipulate isorecursive predicates.

3. We implement our approach based on the Viper verification infrastructure [83]. Our evaluation demonstrates that our technique is reasonably efficient and effective for a wide range of examples.

Chapter Outline. The rest of this chapter is structured as follows. First, in Section 4.1 we provide some background and outline our learning framework. In Section 4.2, we detail the teacher. In Section 4.3, we then describe the learner in more detail. Subsequently, in Section 4.4, we present a concrete SMT encoding that can be used by our learner to synthesise appropriate specifications using an off-the-shelf SMT solver. Afterwards, in Section 4.5, we describe how to overcome the challenges posed by using a verifier supporting isorecursive predicates. Finally, in Section 3.7, we conclude the chapter with a short discussion.

4.1 Learning Framework

Next, we give an overview of our learning framework. In particular, we introduce its individual components and describe how they interact in order to infer permission specifications.

Section Outline. Below, we start this section by providing some background in Section 4.1.1 and then outlining our learning framework in Section 4.1.2. In Section 4.1.3, we proceed by introducing and discussing our running example. After that, we formally define the interface between the teacher and the learner, which allows us to treat them independently: This interfaces comprises of the hypotheses, described in Section 4.1.4, and the samples, introduced in Section 4.1.6. Before the introduction of the samples, however, Section 4.1.5 we first formally define the state and trace abstraction used by our samples. In Section 4.1.7, we conclude this section by discussing the formal guarantees provided by our learning framework.

4.1.1 Background

In recent years, many learning-based inference techniques emerged [5, 6, 26, 46, 95, 96]. These works popularised the *guess and check* paradigm implemented via a teacher and a learner that also our inference is based on. In contrast to *white-box* techniques which synthesise specifications cognisant of the program semantics, the learner of such a guess and check approach is *black-box* and generates candidate specifications based only on the information – that is, the samples – communicated by the teacher. Roughly speaking, early learning-based approaches employed two kinds of samples: examples that indicate which program states have to be allowed by the specifications, and counterexamples that indicate which ones have to be excluded. However – as illustrated below – neither kind of sample is suitable to indicate that the current specifications are not inductive, that is, allow a program state from which there is an execution leading to another program state in which a specification cannot be established.

ICE Learning. To address this issue, Garg et al. proposed ICE, a robust learning framework for synthesizing invariants using examples, counterexamples as well as implications [47]. To illustrate the use for each of these three kinds of samples, let as assume that we want to prove the Hoare triple

$$\{\, P \,\}\ c_1 \,;\, \textsf{while}\ (b)\ \{c_2\} \,;\, c_3\ \{\, Q \,\}$$

and, in order to do so, need to infer a suitable loop invariant I first. The inferred loop invariant should satisfy the following three properties:

- First, the loop invariant I it must be weak enough to be implied by the precondition P across the statement c_1 before the loop; that is, formally, it must satisfy $\{\, P \,\}\ c_1\ \{\, I \,\}$. If this is not the case, the loop invariant forbids a program state σ that must be allowed by a valid loop invariant; the teacher communicates this to the learner by passing a *positive* sample, which requires the loop invariant to allow the state σ.

- Second, the loop invariant I must be strong enough to – in conjunction with the negation of the loop condition – imply the postcondition Q across the statement c_3 after the loop; that is $\{\, I \wedge \neg b \,\}\ c_3\ \{\, Q \,\}$. If this is not the case, the current loop invariant allows a program state σ that must be excluded by any valid loop invariant; the teacher communicates this to the learner by passing a *negative* sample, which requires the invariant to disallow the state σ.

- Third, the loop invariant I must be *inductive* and – in conjunction with the loop condition – imply itself across the loop body c_2; that is $\{ I \wedge b \}\ c_2\ \{ I \}$. If this is not the case, the current loop invariant allows a program state that leads to another program state that it excludes. This can be fixed by either disallowing the former program state or by including the latter. The key insight of the ICE framework is that – since the teacher does not know the precise invariant – it cannot know whether it should pass a positive or negative example and that it would be incorrect to make an arbitrary choice. The solution proposed by Garg et al. is to use an *implication* sample instead; this implication sample carries the intuitive meaning that if the loop invariant allows the former state then it must also allow the latter state.

Advantages of Learning. A learning-based approach has many advantages over a white-box technique. Most prominently – due to the guess-and-check nature of the approach – the learner is unaware of the program semantics and therefore also synthesises specifications for an arbitrarily complex program, as long as it can be verified using specifications within the hypothesis space.

White-box techniques typically require additional heuristics or tactics to generalise the inferred specifications. For example, abstract interpretation uses the widening operator for this purpose [28]; designing them is not easy, as it is notoriously hard to strike the right balance between maintaining precision and achieving a strong enough generalisation that ultimately guarantees convergence. In contrast, learning-based approaches typically try to find the simplest possible specifications permitted by the current set of samples and thereby inherently achieve generalisation.

Finally, a learning-based approach allows the integration of a concrete verifier. As a consequence, the inferred specifications are, by construction, readily verifiable. Moreover, the teacher does not need to duplicate the semantics of the programming language at hand.

4.1.2 Overview

Next, we give an overview of our learning framework. We do this by informally describing all components and how they work in tandem to iteratively infer permission specifications; in particular, we point out how our approach compares to the standard ICE framework and in which points it differs therefrom. An outline of our learning process is shown in Figure 4.1.

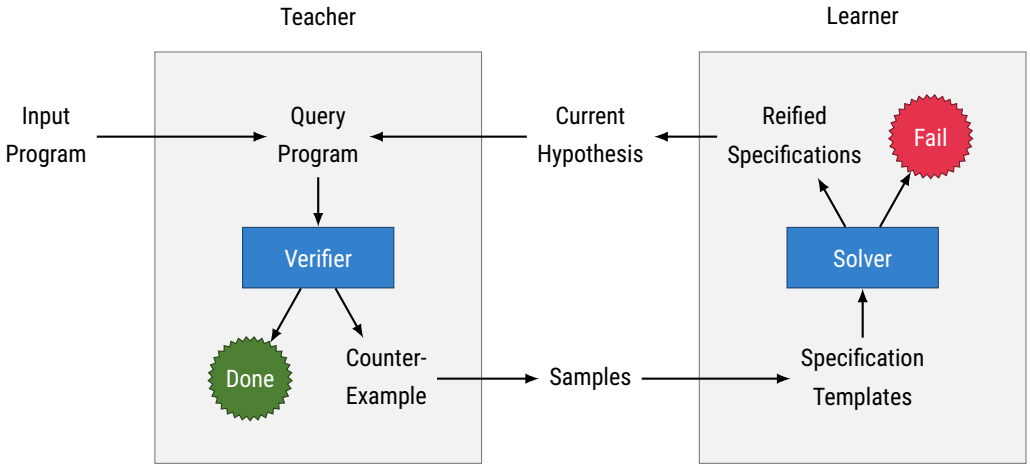


Figure 4.1. An overview of the inference process: The *teacher* attempts to verify the given *input program* with respect ot the current candidate specifications, collectively called *hypothesis*. If the verification fails due to a missing permission, the teacher extracts a *sample* from the reported verification *counterexample*; this sample represents a constraint characterising the missing permission. Based on all samples produced by the teacher so far, the *learner* synthesises a new hypothesis that provides the missing permission. This process is iterated until the verification succeeds or the learner has accumulated an unsatisfiable set of constraints. Note that the case where the verification fails due to a non-permission failure is omitted in this overview.

Input. The input to the inference is an *input program* for which we want specifications to be inferred. These specifications typically include all method preconditions and postconditions as well as a loop invariant for every loop in the program.

Hypotheses. A *hypothesis* provides a candidate for every specification that needs to be inferred. We call a hypothesis *valid* if the input program annotated with the specifications provided by this hypothesis verifies successfully.

Main Loop. In every iteration, the *teacher* constructs a *query program* by annotating the input program with the specifications provided by the current hypothesis. If the verification succeeds, the inference terminates and returns the current hypothesis. Otherwise, the teacher produces a *sample* that characterises the missing permissions responsible for the permission failure and sends it to the learner. The teacher uses the verifier as a black box; that is, it extracts all information to construct a sample from the reported failure and the *verification counterexample*.

The *learner* combines the newly produced sample with all previous samples and synthesises another hypothesis that is consistent with all these samples. The learner

does this by generating *specification templates* and then employs a *solver* to *reify* these templates into concrete specifications.

The inference fails if either the teacher encounters a verification failure that is *not* caused by a missing permission or if the learner's accumulated constraints become unsatisfiable. Note that the former case cannot be fixed by inferring a suitable permission specification; we therefore exclude this case in the following by assuming that all verification failures are due to missing permission.

Abstract Samples over Traces. The standard ICE framework uses concrete program states as samples. In contrast, we produce *symbolic* samples over *traces*. Intuitively, our samples consist of a sequence of specifications and a permission amount, and bound the sum of the corresponding permission changes – where inhaled specifications contribute positively and exhaled specifications negatively – to be at least said permission amount. This novel approach has several important advantages:

- Using symbolic samples does not require footprint information obtained from the verifier to be reified into concrete program states.
- A single symbolic sample can summarise many concrete program states at once, which reduces the number of iterations between the teacher and the learner.
- Using samples over traces enables us to represent disjunctive constraints in a natural way: If there is more than one inhaled specification along a trace our samples only require that they collectively – in their sum – provide the missing permission. The choice as to which specification ultimately includes the permission is left to the learner.
- Moreover, in a context where fractional permissions are used, the permission sums also allow the learner to propose specifications that split required permission gains over multiple specifications: For instance, a full permission may be provided via two separate specifications that each contribute one half of it.

It is also worth pointing out that we only use two types of samples, which we will refer to as *regular samples*[4], and *implication samples*. The former are the

4: We call them regular samples since – depending on how we think about them – we can either interpret them as positive or as negative samples: On one hand, regular samples impose bounds on the gained permissions and therefore *exclude* concrete program states and traces. From this point of view, regular samples are negative samples. On the other hand, if we think of them in terms of the permissions that are required to be *included* in the specifications, they can justifiable also be seen as positive samples.

aforementioned samples over traces, while the latter are – just like in the standard ICE framework – used to address non-inductiveness. Note that, in the context of our work where we aim to infer *all* program specifications at once, *inductiveness* does not only apply to loop invariants, but any constellation where we have cyclic dependencies between specifications; examples are preconditions and postconditions of recursive and mutually recursive method calls, or resource invariants.

Requirements. Our inference framework can be instantiated for different programming languages, permission models, verifiers, and solvers. For concreteness, all elaborations below are with respect to the language introduced in Section 2.3 with a specification language based on implicit dynamic frames [98] and fractional permissions [16].

- To annotate the input program with the current hypothesis, our teacher needs to understand where the verifier syntactically expects specifications, but not the semantics of the input program.
- Our inference requires that the verifier returns a verification counterexample in cases where the verification results in a permission failure. This counterexample needs to contain information about the missing permission and the verification trace that caused the permission failure. In particular, the teacher needs to be able to determine for which heap location permissions were missing.
- The underlying program semantics must be reasonably well-behaved with respect to permissions. Roughly speaking, we assume that the standard separation logic frame rule holds for our program semantics; the precise assumption is formalised in Section 4.2.2.
- We initially assume a verifier with equirecursive predicates; in Section 4.5, we then address the additional challenges posed by integrating an isorecursive verifier into our inference framework.

4.1.3 Running Example

We consider the contains method shown in Listing 4.2; In the elaborations below, we restrict ourselves to the loop in the method's body and illustrate how our technique infers a suitable loop invariant.

Hypothesis Space. The inference starts with an initial hypothesis H_1 containing no permissions; that is, the invariant provided by the initial hypothesis is $H_1(\mathsf{inv}) \equiv \mathsf{true}$.

```
1  method contains(head: Node, key: Int) returns (found: Bool)
2    requires H(pre)
3    ensures H(post)
4  {
5    var node: Node
6    node := head
7    found := false
8    while (node ≠ null ∧ ¬found)
9      invariant H(inv)
10   {
11     if (node.val = key) {
12       found := true
13     } else {
14       node := node.next
15     }
16   }
17 }
```

Listing 4.2. Our running example consisting of an iterative implementation of a method contains that checks whether the linked-list starting at head contains a value that is equal to key.

In general, specifications inferred by our inference, can be thought of to have the form

$$\bigstar_{i=1}^{n} (b_i \Rightarrow r_i), \tag{4.1}$$

for some $n \in \mathbb{N}$, where each boolean expression b_i, called *guard*, determines when the resource r_i is required. The latter is either an accessibility predicate $r_i \equiv \mathsf{acc}(e.f, q)$ or an instance of a recursive predicate $r_i \equiv p(\vec{e})$ introduced by our inference. For the sake of this example, we only consider accessibility predicates with full permissions, that is, $q = 1$.

First Iteration. Using the initial hypothesis H_1, the teacher attempts to verify the input program. We assume that the verifier reports missing permissions to access node.val in the condition on line 11, together with a verification counterexample that includes the trace leading up to the permission failure. The teacher produces a sample that, in its simplified form, can be represented as

$$\langle \underbrace{\langle \mathsf{inv}, \{\mathsf{node} \neq \mathsf{null}\}\rangle}_{\text{snapshot}}, \underbrace{\mathsf{acc}(\mathsf{node.val})}_{\text{permissions}} \rangle.$$

The first part of the sample is a simplified representation of a so-called *snapshot* that we will introduce later in Section 4.1.5 and indicates that the loop invariant was

encountered along the failing trace. This tells the learner that the missing permission can be provided by adding permissions to the loop invariant. The inequality $\mathsf{node} \neq \mathsf{null}$ in the snapshot above captures which states lead to the permission failure and originates from the loop condition. The second part of the sample represents a lower bound on the permissions that must be provided by future hypotheses. Note that, in general, a sample can consist of multiple snapshots as more than one specification may be encountered along a failing trace.

Based on this sample, the learner first generates the specification template

$$\tau_{\mathsf{inv}} \equiv b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val}),$$

where $b_1^?$ represents an undetermined guard. The learner then uses a solver to determine the guard such that the resulting hypothesis satisfies the constraint imposed by the sample. A suitable choice is $b_1^? \equiv \mathsf{true}$; this yields a hypothesis with $H_2(\mathsf{inv}) \equiv \mathsf{acc}(\mathsf{node.val})$ which is then used in the second iteration.

Second Iteration. Analogous to the first iteration, in order to allow accessing the field node.next on line 14, the second iteration adds the conjunct $\mathsf{acc}(\mathsf{node.next})$ to the loop invariant. This yields a hypothesis $H_3(\mathsf{inv}) \equiv \mathsf{acc}(\mathsf{node.val}) * \mathsf{acc}(\mathsf{node.next})$ for the third iteration.

Third Iteration. In the third iteration, the loop invariant already provides sufficient permissions to successfully execute the loop body. However, the verifier reports that re-establishing the conjunct $\mathsf{acc}(\mathsf{node.val})$ of the loop invariant at the end of the loop iteration may require the permissions $\mathsf{acc}(\mathsf{node.next.val})$ at the beginning of the loop invariant. In general, a permission failure caused by the current hypothesis can be resolved in two ways: Either by adding more permissions to the specifications such that the missing permission is present, or by removing permissions from the specifications such that the missing permission is no longer needed. The teacher therefore produces an implication sample that captures this choice. In our case, this implication sample can be represented as

$$\underbrace{\langle\langle \mathsf{inv}, \{\mathsf{node} \neq \mathsf{null}\}\rangle, \mathsf{acc}(\mathsf{node.val})\rangle}_{\text{left-hand side sample } S_1} \rightarrow \underbrace{\langle\langle \mathsf{inv}, \{\mathsf{node.next} \neq \mathsf{null}\}\rangle, \mathsf{acc}(\mathsf{node.next.val})\rangle}_{\text{right-hand side sample } S_2}$$

The implication sample consists of two regular samples (labelled with S_1 and S_2, respectively) and roughly speaking carries the meaning that whenever the specifications satisfy the constraint imposed by the left-hand side sample S_1 they also need to satisfy the constraint imposed by the right-hand side sample S_2. As the sample produced in the first iteration requires that the left-hand side of this implication

is satisfied, the next candidate for the loop invariant must provide the permission acc(node.next.val) mentioned by the right-hand side of the implication.

Forwarding to Last Iteration. Extrapolating this process suggests that the access paths appearing in the loop invariant will grow indefinitely. We address this issue by allowing the learner to introduce a recursive predicate once a some threshold on the length of the access paths appearing in the samples is reached. That is, the learner syntactically analyses all field accesses appearing in the samples and tries to detect potential recursions; it then automatically generates a template for a recursive predicate and introduces suitable predicate instances (cf. Section 4.3.1). For our example, we jump to the last iteration and assume that the learner generates the following templates, where rec(x) is a recursive predicate introduced by the learner:

$$\tau_{\mathsf{inv}} \equiv (b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val})) * (b_2^? \Rightarrow \mathsf{acc}(\mathsf{node.next})) * (b_3^? \Rightarrow \mathsf{rec}(\mathsf{node}))$$
$$\tau_{\mathsf{rec}} \equiv (b_4^? \Rightarrow \mathsf{acc}(\mathsf{x.val})) * (b_5^? \Rightarrow \mathsf{acc}(\mathsf{x.next})) * (b_6^? \Rightarrow \mathsf{rec}(\mathsf{x.next}))$$

For appropriate choices of the guards $b_1^?, \ldots, b_6^?$, this yields a hypothesis, where the loop invariant is $H(\mathsf{inv}) \equiv \mathsf{rec}(\mathsf{node})$ and the body of the recursive predicate is $H(\mathsf{rec}) \equiv \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.next}) * \mathsf{rec}(\mathsf{x.next})$. The loop invariant provided by this hypothesis is sufficient to verify the loop in the next iteration, and the inference terminates successfully. In practice, the learner may not immediately come up with suitable guards and require additional iterations before arriving at a valid hypothesis.

4.1.4 Hypotheses

Next, we describe how hypotheses are represented in our formal exposition, define the shape of the specifications they comprise, and briefly elaborate how they affect the program semantics.

Identifying Specifications. We recall that a hypothesis provides a candidate for every specification that needs to be inferred. We introduce a unique identifier id for each of these specifications and use $\mathcal{I}$ to denote the set containing all these identifiers.

Definition 4.1.1. » A *hypothesis* H: is a function mapping specification identifiers id their corresponding candidate specification $H(\mathsf{id})$. Each of these candidate specifications $H(\mathsf{id})$ is an assertion that may depend on some parameters $\vec{x}_{\mathsf{id}}$. We use $H(\mathsf{id}, \vec{e}) :\equiv H(\mathsf{id})[\vec{e} \backslash \vec{x}_{\mathsf{id}}]$ to denote the specification $H(\mathsf{id})$ instantiated with the arguments $\vec{e}$. «

The parameters $\vec{x}_{\mathsf{id}}$ of a candidate specification may be any subset of the program variables that are defined in the respective scope. For the purpose of this work, we restrict ourselves to reference-typed variables. As we will see below, this restriction is justified by the kind of terms that we use as the basic building blocks for our specifications.

Example 4.1.2. » For the contains method from our running example shown in Listing 4.2, our inference introduces specification identifiers for the method precondition pre and postcondition post as well as for the loop invariant inv. The method precondition and postcondition may both depend on the method parameter head, whereas the loop may additionally depend on the local variable node; that is, we have $\vec{x}_{\mathsf{pre}} = \vec{x}_{\mathsf{post}} = \langle \mathsf{head} \rangle$ and $\vec{x}_{\mathsf{inv}} = \langle \mathsf{head}, \mathsf{node} \rangle$. «

The instantiation of the specification parameters with some given arguments is required whenever we need to adapt a specification to its context.

Example 4.1.3. » For any method call, to represent the method's specification in the context of its caller, the formal parameters have to be substituted with the actual arguments. For instance, if we consider the method call contains(list, 42), we can use $H(\mathsf{pre}, \langle \mathsf{list} \rangle) \equiv H(\mathsf{pre})[\mathsf{list} \backslash \mathsf{head}]$ to express the method's precondition at the call site. «

Definition 4.1.4. » A hypothesis H is *valid* if and only if the input program annotated with this hypothesis verifies successfully. «

Shape of Specifications. The shape of our specifications can be described by the following grammar:

$$A ::= \mathsf{true} \mid A * A \mid b \Rightarrow A \mid \mathsf{acc}(e.f, q) \mid p(\vec{e})$$

Note that any assertion that can be derived in this grammar can easily be rewritten into the general form mentioned in our running example and shown in Equation (4.1). The reason why we keep the grammar above slightly more general than this form is to allow the recursive predicates introduced by our learner to include an additional *truncation guard* required for predicate segments (cf. Section 4.3.3).

Resource Guards. Our inference is parametric with respect to the candidate guards b used in our specifications. However, for the sake of concreteness, we use guards in disjunctive normal form over a fixed set of so-called *atoms*. For our evaluation, we instantiate this set of atoms to be the set containing all equality terms that can be formed from null and all reference-typed variables currently in scope.

```
1  method client() {
2    var cell: Cell
3    cell := new Cell(42)
4    swap(cell, cell)
5  }
6
7  method swap(one: Cell, two: Cell)
8    requires H(pre)
9  {
10   var tmp: Int
11   tmp := one.val
12   one.val := two.val
13   two.val := tmp
14 }
```

Listing 4.3. A simple swap method with a client method that indicates that the two cells passed as arguments may alias.

Example 4.1.5. » For a specification with parameters $\vec{x}_{\mathsf{id}} = \langle \mathsf{one}, \mathsf{two} \rangle$, we use the set of atoms $\{\mathsf{one} = \mathsf{null}, \mathsf{two} = \mathsf{null}, \mathsf{one} = \mathsf{two}\}$. «

This choice of atoms is motivated by the following observations:

- First, a specification that unconditionally requires a permission $\mathsf{acc}(e.f)$ implicitly also requires the receiver expression to be non-null, that is, $e \neq \mathsf{null}$. In contrast, a specification requiring $e \neq \mathsf{null} \Rightarrow \mathsf{acc}(a.f)$ allows states in which $e = \mathsf{null}$.
- Second, it is rather common for permission specifications to include certain permissions only if two references do not alias. As illustrated by the Example 4.1.6 below, this can be used to avoid requiring a permission twice in case two receiver expressions alias.

Example 4.1.6. » Let us consider the swap method shown in Listing 4.3. Looking at this method in isolation, we – and also our inference, for that matter – may come to the conclusion that $H(\mathsf{pre}) \equiv \mathsf{acc}(\mathsf{one.val}) * \mathsf{acc}(\mathsf{two.val})$ is a suitable method precondition.

If we consider the method call swap(cell, cell) on line 3, however, this precondition translates to requiring the permission acc(cell.val) *twice*, which cannot be satisfied and therefore is equivalent to false. In order to make the precondition feasible for the client method, the precondition may only require the two permissions acc(one.val) and acc(two.val) simultaneously if $\mathsf{one} \neq \mathsf{two}$; a precondition that achieves this is, for instance, $H(\mathsf{pre}) \equiv \mathsf{acc}(\mathsf{one.val}) * (\mathsf{one} \neq \mathsf{two} \Rightarrow \mathsf{acc}(\mathsf{two.val}))$. «

Parameterised Transition System. We now discuss how the current hypothesis affects the program semantics. Recall that, in each iteration, the teacher constructs a query program that checks whether the input program annotated with the current hypothesis verifies. This means that the semantics of the query program depends on the current hypothesis. For the elaborations in this chapter, we therefore formalise the program semantics as a transition relation $\rightsquigarrow_H \subseteq \Gamma \times \Gamma$ that is parameterised by a hypothesis H. We use $\rightsquigarrow := \bigcup_H \rightsquigarrow_H$ to denote the transition relation that summarises the transitions for all possible hypotheses. For any hypothesis H, an H-trace is a trace with respect to the transition relation $\rightsquigarrow_H$.

Permission Changes. Next, we have closer look at how a hypothesis affects the permissions held along any trace. To this end, we distinguish between *program transitions* and *hypothesis transitions*. The former describe transitions that correspond to program statements and are therefore independent of the current hypothesis, whereas the latter describe a transition in which a specification is inhaled or exhaled. For any trace t of length k, we use $I_t \subseteq \{1, \ldots, k\}$ to denote the indices corresponding to hypothesis transitions. The distinction between program transitions and hypothesis transitions may seem rather apparent but is rather central for our inference: We can control the permission changes caused by hypothesis transitions by choosing a suitable hypothesis; In contrast, program transition remain unaffected by the current hypothesis.

Definition 4.1.7. » For all traces t of length k and all $i \in \{1, \ldots, k-1\}$, the *permission changes* in the i-th step of the trace t are defined as

$$\Delta_t(i) := \pi_{i+1} - \pi_i,$$

where π_i and π_{i+1} denote the permission maps corresponding to the configurations $t[i]$ and $t[i+1]$, respectively. «

We observe that, for any trace t and any index $i \in I_t$, we have

$$\Delta_t(i) = \pm [\![A]\!]_\Pi(\sigma),$$

where $A := H(\mathsf{id}, \vec{e})$ denotes the specification encountered in the i-th step of the trace t, σ denotes the state corresponding to the i-th configuration $t[i]$, and the sign depends on whether the specification was inhaled or exhaled.

4.1.5 State and Trace Abstractions

Next we will give the formal definitions for the state and trace abstractions that will be used for our samples.

State Abstractions. As mentioned earlier, our inference employs symbolic samples. Apart from the benefits that come from using symbolic samples there is also another, more technical motivation: A verifier may internally abstract over many concrete program states; for example, a verifier based on symbolic execution may collect the path constraint $\mathsf{a} \neq \mathsf{null}$ but leave the precise object to which a points undetermined. Therefore, by extension, the verification counterexample may only be detailed enough to describe an abstract verification trace that summarises multiple concrete program traces. Thus, conceptually, there is a pair of a *state abstraction function* $\alpha_\Sigma \colon \Sigma \to \hat{\Sigma}$ and a *state concretisation function* $\gamma_\Sigma \colon \hat{\Sigma} \to 2^\Sigma$ mapping between concrete states Σ and abstracts states $\hat{\Sigma}$ and vice versa. Note that an abstract state may represent multiple concrete states at once, which is why the concretisation function maps to sets of states. Neither function will ever be explicitly computed; they exist merely for the purpose of our formalisations.

Boolean Evaluation. As we will see later, the teacher as well as the learner need to be able to evaluate certain boolean expressions in a given abstract state. However, as an abstract state $\hat{\sigma}$ corresponds to a set $\gamma_\Sigma(\hat{\sigma})$ of potentially many concrete states, it might not be possible to uniquely define the value of the expression. Therefore, we introduce two functions $\lfloor\!\lfloor b \rfloor\!\rfloor(\hat{\sigma})$ and $\lceil\!\lceil b \rceil\!\rceil(\hat{\sigma})$ that under-approximate and over-approximate, respectively, the value of the boolean expression b in the abstract state $\hat{\sigma}$. More formally, we require that

$$\begin{aligned} \lfloor\!\lfloor b \rfloor\!\rfloor(\hat{\sigma}) \quad &\Rightarrow \quad \forall \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon [\![b]\!](\sigma) \\ \forall \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon [\![b]\!](\sigma) \quad &\Rightarrow \quad \lceil\!\lceil b \rceil\!\rceil(\hat{\sigma}), \end{aligned}$$

for all boolean expressions b and abstract states $\hat{\sigma}$.

Example 4.1.8. » Let us consider an abstract state $\hat{\sigma}$ in which x is known to be null but the value of y is unknown. For such an abstract state, we have $\lfloor\!\lfloor \mathsf{x} = \mathsf{null} \rfloor\!\rfloor(\hat{\sigma})$ and $\neg\lfloor\!\lfloor \mathsf{x} \neq \mathsf{null} \rfloor\!\rfloor(\hat{\sigma})$ as well as $\neg\lfloor\!\lfloor \mathsf{y} = \mathsf{null} \rfloor\!\rfloor(\hat{\sigma})$ and $\neg\lfloor\!\lfloor \mathsf{y} \neq \mathsf{null} \rfloor\!\rfloor(\hat{\sigma})$. «

Permission Evaluation. Analogous to the boolean expressions, we also introduce two functions $\lfloor\!\lfloor A \rfloor\!\rfloor_\Pi(\hat{\sigma})$ and $\lceil\!\lceil A \rceil\!\rceil_\Pi(\hat{\sigma})$ that under-approximate and over-approximate, respectively, the permissions captured by the assertion A in the abstract state $\hat{\sigma}$. More formally, for all assertions A and all abstract states $\hat{\sigma}$, we require

$$\begin{aligned} \lfloor\!\lfloor A \rfloor\!\rfloor_\Pi(\hat{\sigma}) &\sqsubseteq \min_{\sigma \in \gamma_\Sigma(\hat{\sigma})} \{[\![A]\!]_\Pi(\sigma)\} \\ \max_{\sigma \in \gamma_\Sigma(\hat{\sigma})} \{[\![A]\!]_\Pi(\sigma)\} &\sqsubseteq \lceil\!\lceil A \rceil\!\rceil_\Pi(\hat{\sigma}). \end{aligned}$$

Example 4.1.9. » For an accessibility predicate $\mathsf{acc}(e.f, q)$ and some abstract state $\hat{\sigma}$, we can compute $[\![\mathsf{acc}(e.f, q)]\!]_\Pi(\hat{\sigma}) = \pi_{\mathsf{zero}}[\langle o, f\rangle \mapsto q]$ in cases where we are sure that $[\![e]\!](\sigma) = o$, for all $\sigma \in \gamma_\Sigma(\hat{\sigma})$, and $[\![\mathsf{acc}(e.f, q)]\!]_\Pi(\hat{\sigma}) = \pi_{\mathsf{zero}}$, otherwise. Moreover, the we can approximate guarded assertions as

$$[\![b \Rightarrow A]\!]_\Pi(\hat{\sigma}) = \begin{cases} [\![A]\!]_\Pi(\hat{\sigma}) & \text{if } [\![b]\!](\hat{\sigma}) \\ \pi_{\mathsf{zero}} & \text{otherwise.} \end{cases}$$

Finally, permissions corresponding to separating conjunctions can be approximated by $[\![A_1 * A_2]\!]_\Pi(\hat{\sigma}) = [\![A_1]\!]_\Pi(\hat{\sigma}) + [\![A_2]\!]_\Pi(\hat{\sigma})$. «

Snapshots. Intuitively, a snapshot captures a state in which a specification is inhaled or exhaled along the failing trace and carries enough information to compute the permission changes corresponding to compute the related permission changes corresponding to any trace permitted by some future hypothesis.

Definition 4.1.10. » A *snapshot* $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma}\rangle$ is a tuple consisting of a sign $\pm$, a specification identifier id, a vector of arguments $\vec{e}$, and an abstract state $\hat{\sigma}$. Moreover, in order to easily refer to the states abstracted by $\hat{\sigma}$, we define $\gamma_\Sigma(\varepsilon) \coloneqq \gamma_\Sigma(\hat{\sigma})$. «

The snapshot $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma}\rangle$, captures that the specification $H(\mathsf{id}, \vec{e})$ was encountered in some state $\sigma \in \gamma_\Sigma(\hat{\sigma})$; and the sign $\pm$ indicates whether the specification was inhaled or exhaled.

For the sake of shorter notation, for all snapshots $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma}\rangle$, and all boolean expressions, we define

$$[\![b]\!](\varepsilon) \coloneqq [\![b[\vec{e}\backslash\vec{x}_{\mathsf{id}}]]\!](\hat{\sigma})$$
$$\lceil\!\lceil b \rceil\!\rceil(\varepsilon) \coloneqq \lceil\!\lceil b[\vec{e}\backslash\vec{x}_{\mathsf{id}}] \rceil\!\rceil(\hat{\sigma})$$

and, analogously, for all assertions A, we also define

$$[\![A]\!]_\Pi(\varepsilon) \coloneqq [\![A[\vec{e}\backslash\vec{x}_{\mathsf{id}}]]\!]_\Pi(\hat{\sigma})$$
$$\lceil\!\lceil A \rceil\!\rceil_\Pi(\varepsilon) \coloneqq \lceil\!\lceil A[\vec{e}\backslash\vec{x}_{\mathsf{id}}] \rceil\!\rceil_\Pi(\hat{\sigma}).$$

Trace Abstractions. As already mentioned, our samples abstract over entire traces. Roughly speaking, we abstract a trace by focusing on only the states in which specifications were encountered and computing the corresponding sequence of snapshots; this gives us enough information to determine how a given hypothesis affects the permission changes along the original concrete trace.

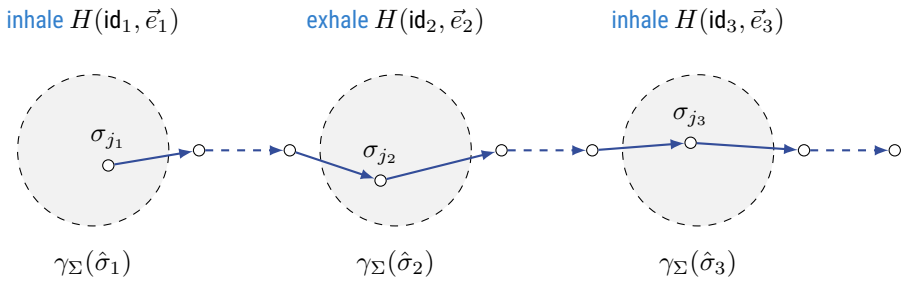


Figure 4.4. A trace abstracted by a sequence of snapshots.

In the following, we define this trace abstraction more formally; this is easiest done algorithmically. That is, we consider an arbitrary trace t and describe how to compute its abstraction $\alpha_T(t)$. To this end, let us consider the trace's specification indices in ascending order: $I_t = \{j_1, \ldots, j_n\}$, where $j_i < j_{i+1}$, for all $i \in \{1, \ldots, n-1\}$. For each encountered specification, we compute the corresponding snapshot. That is, for all $i \in \{1, \ldots, n\}$, we compute the snapshot $\varepsilon_i = \langle \pm_i, \mathsf{id}_i, \vec{e}_i, \hat{\sigma} \rangle$, where the sign $\pm_i$ indicates whether the encountered specification $H(\mathsf{id}_i, \vec{e}_i)$ was inhaled and exhaled and the abstract state $\hat{\sigma}_i$ captures the state σ_{j_i} in which the specification was evaluated; that is $\sigma_{j_i} \in \gamma_\Sigma(\hat{\sigma}_i)$. As illustrated in Figure 4.4, the trace abstraction is then given by the sequence of snapshots $\alpha_T(t) = \langle \varepsilon_1, \ldots, \varepsilon_n \rangle$.

Note that this trace abstraction function uniquely defines a corresponding concretisation function: Specifically, for all sequences of snapshots $\langle \varepsilon_1, \ldots, \varepsilon_n \rangle$, we have

$$\gamma_T(\varepsilon_1, \ldots, \varepsilon_n) \coloneqq \{t \in T \mid \alpha_T(t) = \langle \varepsilon_1, \ldots, \varepsilon_n \rangle\}.$$

4.1.6 Regular and Implication Samples

We are now ready to formally introduce the samples used by our learning framework.

Regular Samples. A regular sample is used whenever the teacher needs to communicate to the learner that a permission is missing and is defined as follows:

Definition 4.1.11. » A *regular sample* $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$ is a tuple consisting of a (potentially empty) sequence of snapshots $\varepsilon_1, \ldots, \varepsilon_k$ and a permission map δ. «

Such a regular sample $S = \langle \varepsilon, \ldots, \varepsilon_n, \delta \rangle$ carries the intuitive meaning that the cumulative permission changes caused by the inhaled and exhaled specifications along any trace captured by the snapshots $\varepsilon_1, \ldots, \varepsilon_n$ have to be at least δ.

Implication Samples. An implication sample is used whenever there are insufficient permissions to establish a specification and the teacher needs to communicate to the learner that – as explained in our running example (cf. Section 4.1.3) – it can either fix the problem by adding the missing permission or by removing the permission from the specification that caused the permission failure.

Definition 4.1.12. » An *implication sample* $S = S_1 \to S_2$ consists of two regular samples S_1 and S_2. The samples S_1 and S_2 are referred to as the *left-hand side* and the *right-hand side* of S, respectively. «

Such an implication sample $S = S_1 \to S_2$ carries the intuitive meaning that whenever a hypothesis satisfies the left-hand side S_1, then it must also satisfy the right-hand side S_2.

Consistent Hypotheses. Next, we formally define under which conditions a hypothesis is consistent with a sample. To facilitate this, we first make the following auxiliary definition:

Definition 4.1.13. » A hypothesis H is $\forall$-*consistent* with a regular sample S if and only if, for all H-traces t with $t \in \gamma_T(S)$, we have $\delta \sqsubseteq \sum_{i \in I_t} \Delta_t(i)$. Similarly, a hypothesis H is $\exists$-*consistent* with a regular sample S if and only if there is an H-trace t with $t \in \gamma_T(S)$ such that $\delta \sqsubseteq \sum_{i \in I_t} \Delta_t(i)$. «

The consistency with respect to regular and implication samples can then be defined as follows:

Definition 4.1.14. » A hypothesis H is *consistent* with a regular sample S if and only if it is $\forall$-consistent with the sample S. «

Definition 4.1.15. » A hypothesis H is *consistent* with an implication sample $S_1 \to S_2$ if and only if it is $\exists$-consistent with the left-hand side sample S_2 and $\forall$-consistent with the right-hand side sample S_2. «

Example 4.1.16. » As an example for a regular sample, let us recall the sample from the first iteration of our running example (cf. Section 4.1.3). For convenience, we repeat this sample in its full form here:

$$S = \langle \langle +, \mathsf{inv}, \vec{x}_{\mathsf{inv}}, \{\mathsf{node} \neq \mathsf{null}\} \rangle, \delta \rangle,$$

where $\vec{x}_{\mathsf{inv}} = \langle \mathsf{head}, \mathsf{node} \rangle$ and δ denotes a permission map capturing a single permission for the field access node.val. A hypothesis H is consistent with this sample S if and only if the invariant $H(\mathsf{inv}, \vec{x}_{\mathsf{inv}})$ provides the permission acc(node.val).

As an example for an implication, let us recall the third iteration of our running example. There, the teacher produced the implication sample $S_1 \to S_2$ with

$$S_1 = \langle\langle +, \mathsf{inv}, \vec{x}_{\mathsf{inv}}, \{\mathsf{node} \neq \mathsf{null}\}\rangle, \delta\rangle$$
$$S_2 = \langle\langle +, \mathsf{inv}, \vec{x}_{\mathsf{inv}}\{\mathsf{node.next} \neq \mathsf{null}\}\rangle, \delta'\rangle,$$

where δ is as above and δ' denotes a permission map capturing a single permission for the field access node.next.val. A hypothesis H is consistent with this sample S if and only if the invariant $H(\mathsf{inv}, \vec{x}_{\mathsf{inv}})$ does *not* provide the permission acc(node.val) or does provide *both* permissions acc(node.val) and acc(node.next.val). «

4.1.7 Formal Guarantees

To wrap up the framework section, we now discuss the formal guarantees provided by our inference. In particular, we formalise the precise criteria that need to be met by the teacher and the learner.

Soundness. Our technique is sound by construction. This stems from the fact that – as we will elaborate in more detail in Section 4.2 – our teacher satisfies the following lemma.

Lemma 4.1.17. » *In every iteration, the query program generated by the teacher verifies if and only if the input program annotated with the current hypothesis verifies.* «

As a consequence and as stated by the following theorem, the specifications inferred by our technique are sound.

Theorem 4.1.18. » *Assuming that the underlying verifier used by the teacher is sound, the specifications inferred by our inference are sound.* «

Proof. The claim directly follows from Lemma 4.1.17 and the fact that the query program checking the final hypothesis in the last iteration verifies successfully. □

Progress and Termination. Roughly speaking, our inference makes progress as any sample produced by the teacher indicates that the verifier has found a deficiency with the current hypothesis, which the teacher then immediately fixes by the learner proposing a next hypothesis that is consistent with this newly produced sample. The following two lemmas formalise this notion:

Lemma 4.1.19. » *In every iteration, the current hypothesis is inconsistent with any newly produced sample.* «

Lemma 4.1.20. » *Any newly synthesised hypothesis is consistent with all samples produced by the teacher so far.* «

The proof of Lemma 4.1.19 is deferred until the discussion of the teacher in Section 4.2. Likewise, the proof of Lemma 4.1.20 is given as part of the discussion of the learner in Section 4.3.

Theorem 4.1.21. » *Our inference is guaranteed to make progress; that is, in every iteration, the hypothesis synthesised by the learner is different from all previous hypotheses.* «

Proof. The claim immediately follows from combining Lemmas 4.1.19 and 4.1.20. □

As a consequence of Theorem 4.1.21, our inference terminates if there are only finitely many specifications for the learner to propose.

Corollary 4.1.22. » *Our inference terminates if the hypothesis space is finite.* «

Proof. We observe that the number of times the inference can make progress is bounded by the number of hypotheses in the hypothesis space. Thus, if the hypothesis space is finite, the claim is immediately implied by Theorem 4.1.21. □

4.2 Teacher

In the following, we describe our teacher in more detail. As mentioned in Section 4.1 – in each iteration – the teacher's task is to construct a query program that checks the validity of the current hypothesis and to extract a suitable sample from the verification counterexample if the verification of this query program fails.

Section Outline. Throughout this section, we consider a *single* iteration of our inference process. In Section 4.2.1, we first discuss how the teacher performs the construction of a query program that verifies successfully (ignoring non-permission failures) if and only if the current hypothesis is valid. Afterwards, we elaborate how the teacher produces a sample in case of a permission failure. To do this, in Section 4.2.2, we first investigate how the teacher can extract a constraint from the verification counterexample that – when satisfied by future hypotheses – prevents the same permission failure from happening again. Based on this constraint, in Section 4.2.3, we then describe how the teacher produces new samples.

4.2.1 Query Programs

We now elaborate how our teacher constructs the query program. We want to point out that the details of this construction – as one would expect – heavily depend on which verifier is used by the teacher. Our elaborations are slightly biased towards the VIPER verifier, which we used for our evaluation. We nonetheless tried to keep the discussion as general as possible and, consequently, at a fairly high level. Roughly speaking, when constructing the query program, the teacher achieves the following three points:

- First, the teacher annotates the input program with the specifications from the current hypothesis. This ensures that the verification of the query program fails due to a permission failure if and only if the current hypothesis is not yet valid. To do so, it first locates all points in the program where a specification is inhaled or exhaled and then inserts the respective specifications from the current hypothesis at these. For the sake of clarity – in our elaborations below – we do this by explicitly adding appropriate inhale and exhale statements.

- Second, the teacher uses the specifications as abstraction boundaries to decompose the input program into methods containing only loop-free code. As a result, any potential failing trace (captured by the verification counterexample in case of a permission failure) will be guaranteed to be finite. As we will see later, this simplifies some of our elaborations – especially when we will be dealing with isorecursive predicates in Section 4.5.

- Third, the teacher – if necessary – adds additional instrumentation such that it will be able to extract all the required information for the samples from the verification counterexamples.

Below, we briefly outline how our teacher implements these three points.

Havoc Statements. For the sake of our elaborations, we assume that the verifier at hand supports a $\mathsf{havoc}(x)$ statement that assigns an arbitrary value to the variable x. Note that, if the verifier does not support such a havoc statement, it can easily be modelled via a call to a method that nondeterministically returns any value. For any sequence of variables $\vec{x} = \langle x_1, \ldots, x_n \rangle$, we write $\mathsf{havoc}(\vec{x})$ as a shorthand for $\mathsf{havoc}(x_1) \,;\, \ldots \,;\, \mathsf{havoc}(x_n)$.

In the following, we will use havoc statements to encode variable updates; For instance, the assignment $\mathsf{x} := \mathsf{e}$ is equivalent to the statement $\mathsf{havoc}\ \mathsf{x} \,;\, \mathsf{inhale}\ \mathsf{x} = \mathsf{e}$. An advantage of constraining the resulting value of an operation via an assertion is that

it allows us to capture all possible behaviours of an operation in cases where the resulting value of a variable is not uniquely determined. For example, the effect of a call $\mathsf{x} := \mathsf{random}(0, 10)$ to a method random returning a random value in the interval defined by its two parameters can be encoded as $\mathsf{havoc}\ \mathsf{x}\ ;\ \mathsf{inhale}\ 0 \leq \mathsf{x} \wedge \mathsf{x} \leq 10$.

Methods and Method Calls. We first explain how method-related specifications are handled in the query program. To this end, we consider an arbitrary method m and assume that its precondition and postcondition are identified by pre and post, respectively.

In order to check whether the method's precondition provides sufficient permissions to successfully execute the method, it is inhaled at the beginning of the method's body. Conversely, to check whether the method's postcondition can be established upon returning from the method, it is exhaled at the end of the method's body. Thus, the teacher replaces the method's body c with

$$\mathsf{inhale}\ H(\mathsf{pre})\ ;\ c\ ;\ \mathsf{exhale}\ H(\mathsf{post}).$$

Method calls are modelled by exhaling the called method's precondition followed by inhaling its postcondition; this checks whether the caller owns sufficient permissions to call the method and accounts for the permission from the caller to the callee and back. That is, the teacher replaces any method call $\vec{x} := \mathsf{m}(\vec{e})$ with

$$\mathsf{exhale}\ H(\mathsf{pre}, \vec{e})\ ;\ \mathsf{havoc}(\vec{x})\ ;\ \mathsf{inhale}\ H(\mathsf{post}, \vec{e}\,')$$

where $\vec{e}\,'$ denotes the sequence obtained from concatenating $\vec{e}$ with $\vec{x}$ (treating return values as out-parameters), and the expression e does not refer to any variable in $\vec{x}$. Note that, in cases where e refers to $\vec{x}$, we can simply rewrite the method call as $\vec{y} := \vec{x}\ ;\ \vec{x} := \mathsf{m}(e[\vec{y}\backslash\vec{x}])$, where $\vec{y}$ is a sequence of fresh variables

Loops. Next, we consider an arbitrary loop $\mathsf{while}\ (b)\ \{c\}$ and assume that its invariant is identified by inv. The teacher replaces each such loop with

$$\mathsf{exhale}\ H(\mathsf{inv})\ ;\ \mathsf{havoc}(\vec{x})\ ;\ \mathsf{inhale}\ H(\mathsf{inv}) \wedge \neg b,$$

where $\vec{x}$ denotes the variables written by the loop body c. This checks whether the loop invariant is weak enough such that it can be established from upstream specifications and whether it is strong enough to establish downstream specifications. The inductivity of the loop invariant can then be checked independently by verifying

$$\mathsf{inhale}\ H(\mathsf{inv}) \wedge b\ ;\ c\ ;\ \mathsf{exhale}\ H(\mathsf{inv}).$$

This can be achieved, for instance, by constructing a separate method with the code fragment above as its body and declares all necessary variables.

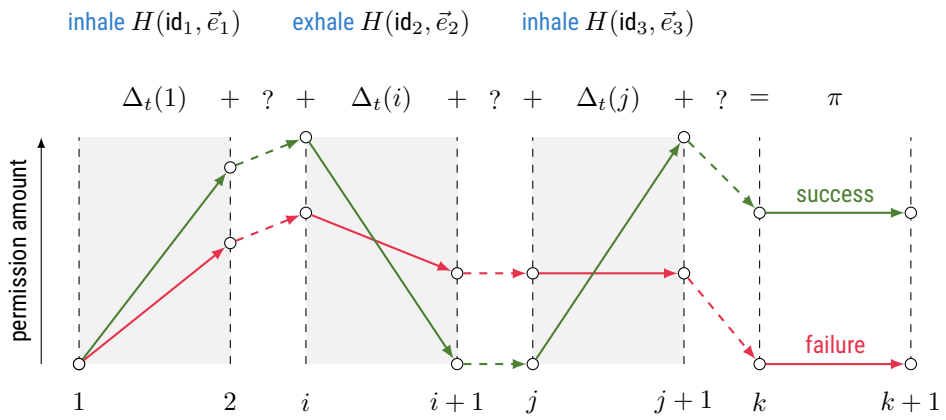


Figure 4.5. A schematic depiction of the permission changes along a failing trace and a similar successful trace

High-Level Concepts. There are many high-level concepts that can be modelled using permissions. We illustrate how our approach could be used to infer the related permission specifications for two such concepts:

- Forking and joining a thread can be modelled as exhaling the thread's precondition and inhaling its postcondition, respectively. That is, the teacher replaces any $t := \text{fork } m(\vec{e})$ statement with $H(\mathsf{pre}, \vec{e})$, where pre identifies the precondition of the forked method. Conversely, each corresponding join t statement is replaced with inhale $H(\mathsf{post}, \vec{e})$, where post identifies the postcondition of the forked method.

- In thread-modular verification, locks are typically associated with a *resource invariant* [85], which is inhaled upon acquiring the lock and then exhaled again when the lock is released. Thus, wherever a lock is acquired, the teacher inserts an inhale $H(\mathsf{res}, l)$ statement, where res identifies the resource invariant and l refers to the lock object. And, dually, at the point where this lock is released again, the teacher inserts an exhale $H(\mathsf{res}, l)$ statement.

Additional Instrumentation. Depending on the verifier at hand, the query program might contain additional instrumentation to allow the teacher to extract all the necessary information from the verification counterexample. For instance, for our prototype implementation, we had to introduce auxiliary variables that saved the value of local variables at points where specifications were inhaled or exhaled in order to retrieve the corresponding program states.

4.2.2 Learning from Failures

An unsuccessful verification of the query program resulting in a permission failure indicates that the verifier found a trace that leads to a final state with insufficient permissions to continue the execution. Below, we investigate how the teacher can extract a constraint from a corresponding verification counterexample and then communicate this constraint via a sample to the learner in order to ensure that any future hypotheses proposed by the learner will not exhibit the same permission failure again.

Roughly speaking, the idea is the following: We consider all traces that differ from the failing trace only in the permission held. For future hypotheses, we want the specifications along these traces – as illustrated in Figure 4.5 – to cumulatively gain sufficient permissions allowing the execution to successfully continue without the permission failure. As we will see below, the teacher can achieve this by constructing a suitable sample that imposes a lower bound on these cumulative permission changes.

Similar Traces. In our elaborations below, we often need to talk about traces that are equal to the failing trace, except that their permission changes correspond to a hypothesis other than the current one that caused the failure. To facilitate this, we introduce the notion of *similarity* that captures the equality of states and traces without taking into account the permission values of their respective permission maps.

Definition 4.2.1. » Two states $\sigma_1 = \langle s_1, h_1, \pi_1 \rangle$ and $\sigma_2 = \langle s_2, h_2, \pi_2 \rangle$ are *similar*, denoted $\sigma_1 \simeq \sigma_2$, if and only if $s_1 = s_2$ and $h_1 \overset{\pi}{=} h_2$, where $\pi \coloneqq \pi_1 \sqcap \pi_2$. Moreover, two configurations $\gamma_1 = \langle c_1, \sigma_1 \rangle$ and $\gamma_2 = \langle c_2, \sigma_2 \rangle$ are *similar*, also denoted $\gamma_1 \simeq \gamma_2$, if and only if $c_1 = c_2$ and $\sigma_1 \simeq \sigma_2$. Finally, two traces t_1 and t_2 are *similar*, also denoted $t_1 \simeq t$, if and only if they are both of length k and $t_1[i] \simeq t_2[i]$, for all $i \in \{1, \ldots, k\}$. «

Note that, in the definition above, the comparison of states is restricted to parts of the heap for which both states hold permissions; this is done to reflect the circumstance that values of all heap locations for which no permissions are held may not be accessed and therefore cannot be distinguished. Moreover, for this exact reason, state similarity – and by extension also trace similarity – are *no* equivalence relations.

Well-Behavedness Assumption. Before we dive into our elaborations, we want to point out that the underlying program semantics – which is implicitly given via the

programming language supported by the verifier – must be reasonably well-behaved with respect to permissions; otherwise, our teacher could not use the verifier as a black-box oracle. Roughly speaking, we require that a missing permission at some point in the program can be fixed by adding more permissions to a specification further up in the control flow. Therefore, we make the following natural assumption.

Assumption 4.2.2. » For all traces t_1 and t_2 with $t_1 \simeq t_2$ and all program transition indices $i \notin I_{t_1}$, we have $\Delta_{t_1}(i) = \Delta_{t_2}(i)$. «

Intuitively, this assumption requires that the permission changes caused by program transitions may only depend on the memory state and must not depend on the permissions currently held. This is closely related to the monotonicity properties required for the soundness of the frame rule [24, 107]. Moreover, note that our assumption requires that permissions gained or lost by a program transition must be deterministic; however, this restriction still allows to non-deterministically branch and then inhale or exhale different permission amounts in the branches as the respective traces would not be similar.

Necessary Permissions. For the sake of our elaborations, we assume that the current hypothesis $H_\bullet$ leads to a permission failure. That is, the verifier finds a *failing trace* $t_\bullet$ of some length k that leads to a final state with insufficient permissions to execute the next statement (originating from the input program) or to establish a specification (encoded by an exhale statement).

In a first step, we identify the permissions that would have been *necessary* to successfully continue the execution of the failing trace. We observe that we can characterise such necessary permissions using a permission map π_{min} such that, for all configurations γ with $\gamma \simeq t_\bullet[k]$ and $\gamma \not\rightsquigarrow \gamma_\lightning$, we have $\gamma \vDash \pi_{\mathsf{min}}$.

Example 4.2.3. » Let us consider the assignment one.val := two.val from our swap method with no annotations shown in Listing 4.3. A permission map π_{min} capturing either permission acc(one.val) or acc(two.val) represents necessary permissions to execute the assignment. «

Note that, alternatively, we could also have tried to identify the permissions that are *sufficient* to successfully execute the failing statement. However this is not always possible as verifiers typically stop as soon as they encounter the first missing permission and therefore may only report part of the missing permissions (for our example above, a verifier probably only reports missing permissions to evaluate the right-hand-side of the assignment and then only encounters the missing permission for the left-hand side once the first permission failure has been fixed).

In the following, we consider an arbitrary trace $t_\circ$ with $t_\circ \simeq t_\bullet$. Our goal is to find a constraint that ensures $t_\circ[k] \vDash \pi_{\mathsf{min}}$. Intuitively, we can think of $t_\circ$ as a hypothetical trace that is similar to the failing one, but allows the trace to successfully continue after the k-th step. Below, we use $t \in \{t_\bullet, t_\circ\}$ to refer to either trace.

Permission Changes. Next, we take a closer look at how the permissions held evolve along these traces (cf. Figure 4.5). We recall that the permission changes in the i-th step – as defined in Definition 4.1.7 – is given by $\Delta_t(i) = \pi_{i+1} - \pi_i$, where π_i and π_{i+1} are the permission map corresponding to configurations $t[i]$ and $t[i+1]$, respectively. At the point of the failure, the permission map π is equal to the sum of all these changes; that is,

$$\pi = \sum_{i=1}^{k-1} \Delta_t(i).$$

In particular, we have $t[k] \vDash \pi$.

Constraint. Recall that – due to the black-box nature of our approach – our teacher does not know how arbitrary parts of the program affect permissions and therefore cannot determine *all* permission changes. However, we know where our specifications are inhaled and exhaled and can, thus, compute the corresponding permission changes (by evaluating field receiver and permission expressions in the verifier's counterexample). The following lemma formalises that hypothesis transitions suffice to determine a constraint that prevents the current permission failure:

Lemma 4.2.4. » *Consider some hypothesis $H_\bullet$ and a $H_\bullet$-trace $t_\bullet$ of length k. Let $\pi_\bullet$ be the permission map corresponding to the final configuration $t_\bullet[k]$ of the trace $t_\bullet$. Moreover, let π_{min} be a permission map such that, for all configurations γ with $\gamma \simeq t_\bullet[k]$ and $\gamma \not\rightsquigarrow \gamma_\lightning$, we have $\gamma \vDash \pi_{\mathsf{min}}$. For all valid hypotheses $H_\circ$, and all $H_\circ$-traces $t_\circ$ with $t_\circ \simeq t_\bullet$, we have*

$$\delta \sqsubseteq \sum_{i \in I_{t_\circ}} \Delta_{t_\circ}(i), \qquad \textit{where } \delta \coloneqq \delta_0 + \sum_{i \in I_{t_\bullet}} \Delta_{t_\bullet}(i)$$

and $\delta_0 \coloneqq \pi_{\mathsf{min}} - \pi_\bullet$. «

Proof. We consider some hypothesis $H_\bullet$ and a $H_\bullet$-trace $t_\bullet$ of length k. Let $\pi_\bullet$ be the permission map corresponding to the final configuration $t_\bullet[k]$ of the trace $t_\bullet$. Moreover, let π_{min} be a permission map such that, for all configurations γ with $\gamma \simeq t_\bullet[k]$ and $\gamma \not\rightsquigarrow \gamma_\lightning$, we have $\gamma \vDash \pi_{\mathsf{min}}$.

Moreover, we let $H_\circ$ be an arbitrary valid hypothesis and consider an arbitrary $H_\circ$-trace $t_\circ$ with $t_\circ \simeq t_\bullet$. By the validity of the hypothesis, we have $t_\circ[k] \not\rightsquigarrow \gamma_\lightning$

and therefore $t_\circ[k] \vDash \pi_{\mathsf{min}}$. Thus, the permission map $\pi_\circ$ corresponding to the configuration $t_\circ[k]$ of the trace $t_\circ$ satisfies $\pi_{\mathsf{min}} \sqsubseteq \pi_\circ$. Using $\pi_* = \sum_{i=1}^{k} \Delta_{t_*}(i)$, for $* \in \{\bullet, \circ\}$ yields

$$\pi_\circ - \pi_\bullet = \sum_{i=1}^{k} \Delta_{t_\circ}(i) - \sum_{i=1}^{k} \Delta_{t_\bullet}(i) = \sum_{i \in I_{t_\circ}} \Delta_{t_\circ}(i) - \sum_{i \in I_{t_\bullet}} \Delta_{t_\bullet}(i),$$

where the second equality follows from Assumption 4.2.2. The claim then follows from combining this with $\pi_{\mathsf{min}} \sqsubseteq \pi_\circ$, followed by some reordering of the terms. □

Note that the constraint from the lemma above is expressed in terms of permission maps and, therefore, represents a bound for *all* resources. A permission failure reported by a verifier, however, typically only refers to a *single* heap location or predicate instance. Thus, in practice, it suffices to evaluate the involved permission maps for this individual heap location or predicate instance; the remaining parts of the permission maps can safely be ignored.

In the next subsection, we explain how our teacher represents this constraint in the samples it sends to the learner.

4.2.3 Sample Extraction

We now describe the actual steps involved when the teacher extracts a sample from the verification counterexample. In a very first step, the teacher determines whether the failure was caused by a *program failure* or by a *hypothesis failure*. The former is caused by insufficient permissions to execute a program statement originating from the input program, whereas the latter is caused by insufficient permissions to establish a specification coming from the current hypothesis.

Program Failures and Regular Samples. We first have a look at the case, where a program failure occurred; for instance, because permissions to read from or write to a field were missing. To convey to the learner that more permissions are needed, the teacher creates a regular sample $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$ capturing the constraint from Lemma 4.2.4:

That is, the teacher first identifies all specifications encountered along the failing trace and computes the corresponding snapshots $\varepsilon_1, \ldots, \varepsilon_n$. That is, each snapshot $\varepsilon_i = \langle \pm_i, \mathsf{id}_i, \vec{e}_i, \hat{\sigma}_i \rangle$ consisting of a sign $\pm_i$, a specification identifier id_i, the specification arguments $\vec{e}_i$, and an abstract state $\hat{\sigma}_i$, and is computed such that $t_\bullet \in \gamma_T(\varepsilon_1, \ldots, \varepsilon_n)$, as defined in Section 4.1.5

The teacher then identifies permissions $\delta_0 = \pi_{\mathsf{min}} - \pi_\bullet$ that were missing but would have been necessary to successfully continue the execution. It finally obtains the lower

bound δ for the sample by adjusting these permissions adjusted for the permissions lost and gained by the encountered specifications; more specifically, it computes

$$\delta_0 + \sum_{i=1}^{n} c_i \cdot [\![A_i]\!]_\Pi(\hat{\sigma}_i) = \delta_0 + \sum_{i \in I_{t_\bullet}} \Delta_{t_\bullet}(i) = \delta,$$

where $A_i \coloneqq H_\bullet(\mathsf{id}_i, \vec{e}_i)$ denotes the i-th specification encountered along the failing trace and each $c_i \in \{-1, 1\}$ depends on whether the respective specification was inhaled or exhaled.

Recall that the permission map δ_0 captures the permissions that a successful trace has to gain *in addition* of what the failing trace has cumulatively gained. Thus, it is always possible to find necessary permission π_{min} such that

$$\delta_0(r) = \pi_{\mathsf{min}}(r) - \pi_\bullet(r) > 0 \tag{4.2}$$

where $r \in R$ is assumed to be the resource for which there were insufficient permissions in the final state of the failing trace. As we will see below, this is important in order to guarantee that our inference makes progress.

Example 4.2.5. » Let us consider a method foo(x) that calls another method bar(x) and then updates x.val := 0, and assume that the current hypothesis provides no permission for the preconditions and postconditions of these two methods. To check the body of the method foo, our teacher verifies the statement

$$\mathsf{inhale}\ H(\mathsf{id}_1, \vec{e})\ ;\ \underbrace{\mathsf{exhale}\ H(\mathsf{id}_2, \vec{e})\ ;\ \mathsf{inhale}\ H(\mathsf{id}_3, \vec{e})}_{\text{encodes method call bar(x)}}\ ;\ \mathsf{x.val} := 0\ ;\ \mathsf{exhale}\ H(\mathsf{id}_4, \vec{e}),$$

where $\vec{e} = \langle \mathsf{x} \rangle$, and id_1 and id_2 identify the preconditions of foo and bar, respectively, whereas id_4 and id_3 identify their postconditions.

Since none of the specifications add or remove any permission, the assignment a.val := 0 fails. In the following, let r denote the resource corresponding to the field access a.val. The specifications encountered along the failing trace are id_1, id_2, and id_3. From the verification counterexample, the teacher determines that there are at least the permissions $\delta_0 = \pi_{\mathsf{zero}}[r \mapsto 1]$ missing in the failing state and computes

$$\delta = \delta_0 + \sum_{i=1}^{3} c_i \cdot \underbrace{[\![H(\mathsf{id}_i, \vec{e})]\!]_\Pi(\hat{\sigma}_i)}_{=\pi_{\mathsf{zero}}} = \pi_{\mathsf{zero}}[r \mapsto 1],$$

where $\hat{\sigma}_1$, $\hat{\sigma}_2$, and $\hat{\sigma}_3$ capture the states at the corresponding specifications. The teacher then produces the sample $S = \langle \varepsilon_1, \varepsilon_2, \varepsilon_3, \delta \rangle$, where

$$\varepsilon_1 = \langle +, \mathsf{id}_1, \vec{e}, \hat{\sigma}_1 \rangle$$
$$\varepsilon_2 = \langle -, \mathsf{id}_2, \vec{e}, \hat{\sigma}_2 \rangle$$
$$\varepsilon_3 = \langle +, \mathsf{id}_3, \vec{e}, \hat{\sigma}_3 \rangle.$$

Note that this sample allows the learner to provide the missing permissions via the precondition of foo or the postcondition of bar (or to split it up between the two). «

Hypothesis Failures and Implication Samples. A *hypothesis failure* occurs when there are insufficient permissions to establish a specification provided by the current hypothesis; for instance, because permissions are missing to reestablish a loop invariant. In this case, the teacher produces an implication sample $S = S_1 \to S_2$ that constrains future hypotheses to include the missing permissions only if they still the part of the specification that caused the failure:

- The left-hand side $S_1 = \langle \varepsilon, \delta \rangle$ is a regular sample, where the snapshot $\varepsilon = \langle +, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$ is such that $H(\mathsf{id}, \vec{e})$ represents the specification that could not be established and $\hat{\sigma}$ abstracts the failing state. Moreover, the permission map δ captures the part of the specification that caused the failure.
- The right-hand side S_2 is a regular sample that is computed just as in the case of a program failure described above.

Self-Framing Checks. Recall that implicit dynamic frames formulas are not necessarily self-framed; that is, they may not contain enough permissions to evaluate some sub-expressions appearing within themselves. We observe that permission failures caused by a specification not being self-framed can be seen as a special case of a hypothesis failure. Thus – as the following example illustrates – they can also be handled using implication samples.

Example 4.2.6. » Assume that the current candidate for the body of the recursive predicate is

$$H(\mathsf{rec}) \equiv \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{rec}(\mathsf{x.next}).$$

Clearly, the field access x.next appearing in the recursive instance rec(x.next) is not framed. Therefore, the teacher produces the implication sample

$$\langle \langle +, \mathsf{rec}, \langle \mathsf{x} \rangle, \{\mathsf{x} \neq \mathsf{null}\} \rangle, \mathsf{rec}(\mathsf{x.next}) \rangle \to \langle \langle +, \mathsf{rec}, \langle \mathsf{x} \rangle, \{\mathsf{x} \neq \mathsf{null}\} \rangle, \mathsf{acc}(\mathsf{x.next}) \rangle$$

in order to ensure that whenever the learner includes the instance rec(x.next), it also includes the necessary permissions to frame it. «

Formal Guarantees. Recall from Section 4.1.7 that – in order to guarantee progress – our teacher has to satisfy Lemma 4.1.19, which states that *in every iteration, the current hypothesis is inconsistent with any newly produced sample.* As the following proof shows, this is indeed the case:

Proof (of Lemma 4.1.19). Let us consider an arbitrary regular sample $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$ that our teacher produced as described above. Towards contradiction, we assume that the current hypothesis $H_\bullet$ is consistent with this sample S. Clearly, by construction of the sample, the failing trace satisfies $t_\bullet \in \gamma_T(S)$. Moreover, we have

$$\delta_0 + \sum_{i \in I_{t_\bullet}} \Delta_{t_\bullet}(i) = \delta \sqsubseteq \sum_{i \in I_{t_\bullet}} \Delta_{t_\bullet}(i),$$

where the equality holds by the construction of the sample according to the constraint from Lemma 4.2.4 and the inequality due to the assumption of the current hypothesis being consistent with the sample S. Thus, we have $\delta_0 \sqsubseteq \pi_{\mathsf{zero}}$. However, this cannot be: by Equation (4.2), we have $\delta_0(r) > 0$, for some $r \in R$ (the resource for which the permission was missing and was causing the permission failure). Hence, our assumption has lead to a contradiction, which allows us to conclude that the current hypothesis $H_\bullet$ is inconsistent with the sample S.

Similarly, for a newly produced implication sample $S = S_1 \to S_2$, we observe that the current hypothesis is consistent with the left-hand side S_1 (as it captures the permission that caused the failure) while it is inconsistent with the right-hand side S_2 (analogous to regular samples). Thus, the implication constraint imposed by the sample S is not satisfied. □

4.3 Learner

In this section, we turn our attention to the learner. As mentioned in Section 4.1 – in every iteration – the learner tries to synthesise a new hypothesis that is consistent with all samples that have been produced by the teacher so far. Roughly speaking, the synthesis of each new hypothesis consists of the following two phases

- In a first *template generation phase*, the learner generates a specification template for every specification that needs to be inferred. Intuitively, each of these specification templates has the same shape as the specifications ultimately provided by the hypothesis (cf. Section 4.1.4) but leaves all guards and permission amounts undetermined.

- In a second *reification phase*, the learner then reifies these specifications by synthesizing concrete guards and permission amounts such that the resulting hypothesis is consistent with all samples. The synthesis of these guards and permission amounts is performed via an encoding to a suitable solver.

Section Outline. Below, we describe the aforementioned template generation and reification phase in more detail. To this end, in Section 4.3.1, we first introduce the specification templates used by our learner. Then, in Section 4.3.2, we elaborate how the specification templates get computed based on the samples provided by the teacher. Thereby, we first focus on specification templates without recursive predicates. In Section 4.3.3, we then explain how this template generation is extended to also handle recursive predicates. Thereafter, we address the reification phase in which the specification templates are turned into concrete specifications that are consistent with the samples at hand. As our inference employs samples over traces, a single sample potentially constrains multiple specifications at once. Therefore, in Section 4.3.4, we elaborate how a constraint imposed by a sample can be decomposed into several simpler constraints involving individual specifications. A concrete translation of these simpler constraints into an SMT encoding that can be solved using an off-the-shelf SMT solver is postponed until Section 4.4.

4.3.1 Specification Templates

The specification templates generated by the learner unsurprisingly closely follow the shape of our specifications as introduced in Section 4.1.4. The most notable difference is that they also comprise placeholders for guards and permission amounts; these placeholders are then concretised during the reification phase.

Guard Placeholders. A *guard placeholder* $b_k^?$ represents an undetermined guard. Any resource appearing in a specification template will be guarded by such a guard placeholder.

Permission Placeholders. All accessibility predicates appearing in a specification template are of the form $\mathsf{acc}(e.f, q_k^?)$, where $q_k^?$ is a *permission placeholder* that allows a suitable permission fraction to be picked during the reification phase.

Predicates. Our specification templates allow the learner to introduce instances of recursive predicates; as will be discussed in more detail later, the definition of such a predicate is captured by a separate specification template. In our elaborations, we use `rec` to name the predicate introduced by the teacher (for the sake of simpler presentation, we assume that there is at most one) and `seg` to refer to a segmented version thereof (cf. Section 4.3.3). Intuitively, `seg(x, y)` describes the predicate `seg(x)` truncated at `y`; conversely, we can think of `seg(x)` being equivalent to `seg(x, null)`.

Choices. In addition to guard and permission placeholders, we also employ placeholders for expressions $e_k^?$. Such an expression placeholder represents a *choice* among a given set of options $O_k = \{e_1, \ldots, e_n\}$ that is generated along with the specification template. We will make use of such expression placeholders to defer the choices of suitable truncation arguments for recursive predicates to the reification phase; that is, all segment predicates appearing in our specifications will have the form $\mathsf{seg}(e, e_k^?)$.

Template Grammar. In general, all template expressions generated by the learner can be derived in the following grammar:

$$\tau ::= \mathsf{true} \mid \tau * \tau \mid b \Rightarrow \tau \mid b_k^? \Rightarrow \tau \mid \mathsf{acc}(e.f, q_k^?) \mid \mathsf{rec}(e) \mid \mathsf{seg}(e, e_k^?)$$

Apart from the placeholders introduced above, this grammar coincides with our specification grammar. Note that the third rule $b \Rightarrow \tau$ allows one to introduce *concrete* guards b that are fixed and therefore cannot be determined during the reification phase. We will make use of this in Section 4.3.3 to introduce termination conditions of segment predicates.

4.3.2 Template Generation

The generation of the specification templates is based on analysing which resources are mentioned by the samples and, roughly speaking, can be broken down into the following two steps:

- First, for all specification identifiers id, the learner computes a *resource set* R_{id} containing all resources mentioned by any sample for the respective specification. Intuitively, these sets over-approximate the resources for which – under consideration of the current set of samples – the specification $H(\mathsf{id})$ provided by the hypothesis for the next iteration needs to contain some permissions.

- Second, for all specification identifiers id, the learner generates a specification template τ_{id} that, intuitively speaking, contains a conjunct $b_k^? \Rightarrow r$, for every resource $r \in R_{\mathsf{id}}$. Thus, the resulting specification templates allow the learner to provide all permissions mentioned by the samples. Note that these specification templates may also contain conjuncts for resources that need not – and sometimes even must not — be included in the final specification; such a resource can be omitted entirely by setting its respective guard to false during the reification phase.

Adapting Expressions. Recall that a sample imposes a lower bound on the permissions cumulatively gained by the specifications encountered along some trace. Thus, the learner needs to be able to construct assertions – consisting of accessibility predicates and predicate instances – that can be used to express the required permissions within these specifications. As the specifications are evaluated in different states – each of which is captured by a snapshot – the assertion that can be used to express a given permission may vary from specification to specification. We therefore need a way to adapt an expression (for instance the receiver of a field access) to a state captured by a snapshot. To this end, for all snapshots $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$ and all values v, we define

$$E_\varepsilon(v) := \{e \mid \forall \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon [\![e[\vec{e} \backslash \vec{x}_{\mathsf{id}}]]\!](\sigma) = v\},$$

which will be used for the computation of the resource sets as described below.

Note that there might be situations where there are no expressions evaluating to the value v, in which case the set $E_\varepsilon(v)$ is empty. Moreover, we also note that these sets $E_\varepsilon(v)$ can be arbitrarily large and – in cases where pointers form cycles or loops – can even be infinite. In practice, however, it suffices to consider the subset of $E_\varepsilon(v)$ restricted to expressions that are actually allowed to appear in the final specification. This subset is finite and can, therefore, be computed explicitly: Due to syntactic restrictions such as bounding the length of access paths that may appear in specifications all of the finitely many possible candidate expressions can be enumerated and checked.

Resource Sets. We now describe how the learner computes the resource set R_{id} for some given specification with identifier id. The learner does this by iterating over all samples and collecting the resources $R_{\mathsf{id}}(S)$ contributed by each sample S. The final resource set is then obtained by computing the union of all these sets, that is $R_{\mathsf{id}} := \bigcup_S R_{\mathsf{id}}(S)$.

For any implication sample $S = S_1 \to S_2$, we compute $R_{\mathsf{id}}(S) := R_{\mathsf{id}}(S_1) \cup R_{\mathsf{id}}(S_2)$. Since an implication consists of two regular samples, and any sample is either an implication sample or a regular one, this allows us to consider only regular samples from now on. For any regular samples $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$, we further break down the computation of the resource set $R_{\mathsf{id}}(\mathsf{id})$ to the level of individual snapshots ε: That is, we define

$$R_{\mathsf{id}}(S) := \bigcup_{i=1}^{n} R_{\mathsf{id}}(\varepsilon_i, \delta)$$

and elaborate how to compute the resource set $R_{\mathsf{id}}(\varepsilon, \delta)$ for an arbitrary snapshot $\varepsilon = \langle \pm, \mathsf{id}', \vec{e}, \hat{\sigma} \rangle$. If $\mathsf{id} \neq \mathsf{id}'$, then the snapshot ε does not contribute to the resource set

and we have $R_{\mathsf{id}}(\varepsilon_i, \delta) = \varnothing$. For the case where $\mathsf{id} = \mathsf{id}'$, we further distinguish the following cases:

- If the permission map δ represents a permission for a single heap location $l = \langle o, f \rangle$, that is, we have $\delta = \pi_{\mathsf{zero}}[l \mapsto q]$, for some positive permission fraction $q > 0$, then, we compute

  $$R_{\mathsf{id}}(\varepsilon, \delta) \coloneqq \{\mathsf{acc}(e.f) \mid e \in E_\varepsilon(o)\}.$$

  Note that, for the computation of the resource sets, the permission amount q is ignored entirely. This is because the permission amounts will be reintroduced as permission placeholders in the templates for which suitable values will be determined during the reification phase.

- If the permission map δ represents a permission for a single predicate instance $p(v_1, \ldots, v_m) \equiv \mathsf{rec}(v_1)$ or $p(v_1, \ldots, v_m) \equiv \mathsf{seg}(v_1, v_2)$, then we define

  $$R_{\mathsf{id}}(\varepsilon, \delta) \coloneqq \{p(e_1, \ldots, e_m) \mid \forall i \in \{1, \ldots, m\} \colon e_i \in E_\varepsilon(v_i)\}$$

- In the general case, we decompose the permission map $\delta = \sum_{i=1}^{m} \delta_i$ into several permission maps δ_i capturing permissions for individual heap locations or predicate instances and then compute

  $$R_{\mathsf{id}}(\varepsilon, \delta) \coloneqq \bigcup_{i=1}^{m} R_{\mathsf{id}}(\varepsilon, \delta_i)$$

  using the definitions from the previous two cases.

Example 4.3.1. » Recall that in the first two iterations of our running example (cf. Section 4.1.3), the teacher produces two samples that both consist of a snapshot corresponding to the loop invariant inv and their permission maps correspond to the permissions $\mathsf{acc}(\mathsf{node.val})$ and $\mathsf{acc}(\mathsf{node.next})$, respectively. Thus, these two samples yield the resource set

$$R_{\mathsf{inv}} = \{\mathsf{acc}(\mathsf{node.val}), \mathsf{acc}(\mathsf{node.next})\}$$

used as a basis for the template generation in the second iteration. «

Template Conjuncts. As already mentioned above, the learner computes the specification templates τ_{id} by adding a conjunct for each resource $r \in R_{\mathsf{id}}$. More formally, given some resource set $R_{\mathsf{id}} = \{r_1, \ldots, r_n\}$, the learner computes the corresponding specification template as

$$\tau_{\mathsf{id}} \equiv \mathop{\ast}_{i=1}^{n} A_i, \qquad \text{where } A_i :\equiv \begin{cases} b_i^? \Rightarrow \mathsf{acc}(e.f, q_i^?) & \text{if } r_i \equiv \mathsf{acc}(e.f) \\ b_i^? \Rightarrow r_i & \text{if } r_i \equiv \mathsf{rec}(e) \text{ or } r_i \equiv \mathsf{seg}(e, e_i^?). \end{cases}$$

Example 4.3.2. » Given the resource set from Example 4.3.1 above, the learner computes the specification template

$$\tau_{\mathsf{inv}} \equiv (b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val}, q_1^?)) * (b_2^? \Rightarrow \mathsf{acc}(\mathsf{node.next}, q_2^?))$$

for the loop invariant. «

4.3.3 Recursion Detection

Next, we describe how the template generation can be extended to also handle recursive predicates. For simplicity, we assume that there is only one recursive predicate; the extension to multiple recursive predicates is straightforward and mostly a matter of allowing the learner to generate more than one predicate instance.

Predicate Templates. As most commonly used recursive data structures can be described using a recursive predicate rec with one *recursive* parameter, we fix the parameters of the recursive predicate to be $\vec{x}_{\mathsf{rec}} = \langle \mathsf{x} \rangle$; segment predicates seg with a second parameter are treated separately later. Note that, for the purpose of our permission inference, we can safely ignore any additional parameters potentially required to express value constraints.

Moreover, for our elaborations, it is helpful to conceptually divide each recursive predicate into a *recursive* and a *non-recursive* part. The recursive part – as its name suggests – consists of all the recursive instances of the predicate itself; in accordance with our introduction of recursive predicates in Section 2.2.4, we call each recursive instance $\mathsf{rec}(s(\mathsf{x}))$ a *successor* and associate it with a corresponding successor function s. Dually, the non-recursive part consists of everything else. The specification templates for recursive predicates employed by our learner reflects this division and are of the form

$$\tau_{\mathsf{rec}} \equiv \underbrace{\mathop{\Large *}_{i=1}^{n} (b_i^? \Rightarrow \mathsf{acc}(\mathsf{x}.f_i, q_i^?))}_{\text{non-recursive part}} * \underbrace{\mathop{\Large *}_{i=1}^{m} (b_{n+i}^? \Rightarrow \mathsf{rec}(s_i(\mathsf{x})))}_{\text{recursive part}} .$$

Detection Heuristics. A simple but central insight of our recursion detection heuristics is that it only makes sense to include a permission for a field access in our recursive predicate, if the permission for that field is required for two different nodes of the data structure at hand. Moreover, it is highly likely that the permission is required for a node and its successor. Therefore, if $\mathsf{acc}(e.f_1)$ and $\mathsf{acc}(e.f_2.f_1)$ both appear in the resource set R_{id} for some specification, we do the following:

- First, we include permissions for the field f_1 in our recursive predicate. That is, we add the conjunct $b_k^? \Rightarrow \mathsf{acc}(\mathsf{x}.f_1, q_k^?)$ to the template τ_{rec} of the recursive predicate.
- Second, we consider $s(e) :\equiv e.f_2$ as a potential successor for the recursive predicate and, therefore, add the conjunct $b_k^? \Rightarrow \mathsf{rec}(s(\mathsf{x}))$ to the template τ_{rec} of the recursive predicate.

  Note that – as an optimisation that aims at reducing the total number of iteration that our inference needs – we can also add the conjunct $b_k^? \Rightarrow \mathsf{acc}(\mathsf{x}.f_2, q_k^?)$ to the predicate template; this then ensures that predicate template allows to frame the field access appearing in the successor expression $s(\mathsf{x}) \equiv \mathsf{x}.f_2$. If, however, we chose not to do this, the self-framing checks that our teacher performs will take care of this (cf. Section 4.2.3).
- Finally, we allow the learner to provide the permissions for the field accesses $e.f_1$ and $s(e).f_1 \equiv e.f_2.f_1$ via the recursive predicate by adding the conjunct $b_k^? \Rightarrow \mathsf{rec}(e)$ to the template τ_{id} of the specification in question.

Example 4.3.3. » Let us assume that the permissions $\mathsf{acc}(\mathsf{node.val})$ and $\mathsf{acc}(\mathsf{node.next.val})$ appear in the resource set for some loop invariant; that is, we have

$$\{\mathsf{acc}(\mathsf{node.val}), \mathsf{acc}(\mathsf{node.next.val})\} \subseteq R_{\mathsf{inv}}.$$

Note that this was exactly the situation occurring in the last iteration of our running example (cf. Section 4.1.3).

Based on these field accesses and according to our elaborations above, the learner adds the successor $s(e) \equiv e.\mathsf{next}$ and permissions for the field val to the recursive predicate. This yields the specification template

$$\tau_{\mathsf{inv}} \equiv \ldots * (b_3^? \Rightarrow \mathsf{rec}(\mathsf{node}))$$
$$\tau_{\mathsf{rec}} \equiv (b_4^? \Rightarrow \mathsf{acc}(\mathsf{x.val}, q_4^?)) * (b_5^? \Rightarrow \mathsf{acc}(\mathsf{x.next}, q_5^?)) * (b_6^? \Rightarrow \mathsf{rec}(\mathsf{x.next}))$$

Note that the conjunct $b_5^? \Rightarrow \mathsf{acc}(\mathsf{x.next}, q_5^?)$ is added as part of our optimisation to allow to frame all arguments to the recursive predicate instance. «

Segment Predicates. We now briefly discuss how our learner handles segment predicates used to capture permissions corresponding to partial data structures. Such segment predicates are typically needed to write specifications for code that iteratively traverses a recursive data structure; for example, to capture the permissions corresponding to the part of the data structure that has already been traversed. For our work,

we restrict ourselves to list-like predicates; a generalisation to segmented tree-like predicates is possible but nontrivial.

For the sake of illustration, let us consider an arbitrary list-like predicate $\mathsf{rec}(\mathsf{x})$ consisting of a non-recursive part $\mathsf{rec}'(\mathsf{x})$ and single successor instance $\mathsf{rec}(s(\mathsf{x}))$ guarded by some condition b; that is, we have $\mathsf{rec}(\mathsf{x}) \triangleq \mathsf{rec}'(\mathsf{x}) * (b \Rightarrow \mathsf{rec}(s(\mathsf{x})))$. From this decomposed, we can easily generate a segmented version of the predicate:

$$\mathsf{seg}(\mathsf{x}, \mathsf{y}) \triangleq \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{rec}'(\mathsf{x}) * (b \Rightarrow \mathsf{seg}(s(\mathsf{x}), \mathsf{y}))$$

Intuitively $\mathsf{seg}(\mathsf{x}, \mathsf{y})$ describes the predicate rec *truncated* at y. Thus, in order to directly produce such a segment predicate, our learner can generate a specification template of the from

$$\tau_{\mathsf{seg}} \equiv \mathsf{x} \neq \mathsf{y} \Rightarrow \left(\mathop{\ast}_{i=1}^{n} (b_i^? \Rightarrow \mathsf{acc}(\mathsf{x}.f, q_i^?)) * (b_{n+1}^? \Rightarrow \mathsf{seg}(s(\mathsf{x}), \mathsf{y})) \right).$$

Example 4.3.4. » The template for the segment predicate corresponding to the recursive predicate from Example 4.3.3 is

$$\tau_{\mathsf{seg}} \equiv \mathsf{x} \neq \mathsf{y} \Rightarrow ((b_4^? \Rightarrow \mathsf{acc}(\mathsf{x.val}, q_4^?)) * (b_5^? \Rightarrow \mathsf{acc}(\mathsf{x.next}, q_5^?)) * (b_6^? \Rightarrow \mathsf{seg}(\mathsf{x.next}, \mathsf{y}))). \quad \text{«}$$

It remains to discuss how our recursion detection heuristics handles the truncation parameter added with the introduction of segment predicates. In order not to prematurely exclude any predicate instances, the choice for the second parameter is delegated to the reification phase. That is, wherever our recursion detection heuristics described above introduces the predicate instance $\mathsf{rec}(e)$, we add the predicate instance $\mathsf{seg}(e, e_k^?)$ instead. In our evaluation, we observed that a suitable choice of options for the expression placeholder $e_k^?$ is $O_k = \{x_1, \ldots, x_n\} \setminus \{e\}$, where $\vec{x}_{\mathsf{id}} = \langle x_1, \ldots, x_n \rangle$ are the parameters of the specification in question.

4.3.4 Reification Phase

Next, we turn our attention to the reification phase in which the learner reifies the specification templates into concrete specifications that are consistent with all samples. In this section, we explain how to extract constraints on samples from the samples. In Section 4.4 we then elaborate how these constraints can be translated into a concrete SMT encoding and solved by an off-the-shelf solver, such as Z3 [79].

Each sample potentially constrains the permission changes caused by multiple specifications. To obtain constraints on individual specifications, we decompose the constraint imposed by any samples into several simpler ones corresponding to their individual snapshots; As each snapshot corresponds to *one* inhaled or exhaled specification these simpler constraints involve single specifications.

Snapshot Constraints. First, we introduce so-called snapshot constraints that capture lower bounds on the permission changes corresponding to a single snapshot.

Definition 4.3.5. » For all snapshots $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$ and all permission maps δ, we define the corresponding *snapshot constraints* as

$$\begin{aligned} \psi_H^{\forall}(\varepsilon, \delta) \quad &:\Leftrightarrow \quad \forall \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon (\delta \sqsubseteq c \cdot [\![A]\!]_\Pi(\sigma)) \\ \psi_H^{\exists}(\varepsilon, \delta) \quad &:\Leftrightarrow \quad \exists \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon (\delta \sqsubseteq c \cdot [\![A]\!]_\Pi(\sigma)), \end{aligned}$$

where $A \coloneqq H(\mathsf{id}, \vec{e})$ and $c \in \{-1, 1\}$ adjusts the sign of the permissions depending on whether the snapshot corresponds to an inhaled or exhaled specification. «

Note that, for snapshots ε corresponding to an inhaled specification, we could equivalently define $\psi_H^{\forall}(\varepsilon, \delta) :\Leftrightarrow \llbracket A \rrbracket_\Pi(\varepsilon)$ and $\psi_H^{\exists}(\varepsilon, \delta) :\Leftrightarrow \lceil A \rceil_\Pi(\varepsilon)$; for snapshots corresponding to exhaled specifications, the approximation needs to be flipped.

Intuitively, the first constraint $\psi_H^{\forall}(\varepsilon, \delta)$ captures that the permission changes corresponding to the snapshot ε are *guaranteed* to be at least δ and – as we will see below – is required to prove that a hypothesis is $\forall$-consistent with a sample (cf. Definition 4.1.13). In contrast, the second constraint $\psi_H^{\exists}(\varepsilon, \delta)$ captures that there *may* be states for which the permission changes are δ and is required to prove a hypothesis' $\exists$-consistency.

Decomposed Sample Constraint. We are now ready to formulate the decomposed constraint corresponding to a regular sample:

Definition 4.3.6. » For all samples $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$, we define

$$\psi_H^{\forall}(S) \quad :\Leftrightarrow \quad \exists \delta_1, \ldots, \delta_n \colon \left(\delta \sqsubseteq \sum_{i=1}^{n} \delta_i \wedge \bigwedge_{i=1}^{n} \psi_H^{\forall}(\varepsilon_i, \delta_i) \right).$$

The constraint $\psi_H^{\exists}(S)$ is defined analogously but uses $\psi_H^{\exists}(\varepsilon_i, \delta_i)$ instead of $\psi_H^{\forall}(\varepsilon_i, \delta_i)$. «

As stated by the following lemma, these constraints precisely capture a hypothesis' $\forall$-consistency and $\exists$-consistency, respectively.

Lemma 4.3.7. » *For all hypotheses H and all regular samples S, the following two statements are true:*

1. *If $\psi_H^{\forall}(S)$ then the hypothesis H is $\forall$-consistent with the sample S.*
2. *If $\psi_H^{\exists}(S)$ then the hypothesis H is $\exists$-consistent with the sample S.* «

Proof. We only prove the first claim; the proof for the second claim follows analogously. Let us consider an arbitrary hypothesis H and an arbitrary regular sample $S = \langle \varepsilon_1, \ldots, \varepsilon_n, \delta \rangle$, where $\varepsilon_i = \langle \pm_i, \mathsf{id}_i, \vec{e}_i, \hat{\sigma}_i \rangle$, for all $i \in \{1, \ldots, n\}$. To prove the claim, we assume $\psi_H^{\forall}(S)$ and aim to show that the hypothesis H is $\forall$-consistent with the sample S. According to Definition 4.1.13, we can do this by showing $\delta \sqsubseteq \sum_{i \in I_t} \Delta_t(i)$ for an arbitrary H-trace t with $t \in \gamma_T(S)$.

We consider the indices $I_t = \{j_1, \ldots, j_n\}$ corresponding to the hypothesis transitions of the trace t, and, without loss of generality, assume that $j_1 < j_2 < \ldots < j_n$. Moreover, for all $i \in \{1, \ldots, n\}$, we consider the i-th specification $A_i \coloneqq H(\mathsf{id}_i, \vec{e}_i)$ encountered in the j_i-th step and let σ_{j_i} denote the state in which this specification is evaluated. We observe that $\sigma_{j_i} \in \gamma_\Sigma(\hat{\sigma}_i)$ as well as

$$\Delta_t(j_i) = c_i \cdot [\![A_i]\!]_\Pi(\sigma_{j_i}), \tag{4.3}$$

where $c_i \in \{1, -1\}$ depends on whether the specification A_i is inhaled or exhaled. With this, we are ready to conclude that

$$\delta \sqsubseteq \sum_{i=1}^{n} \delta_i \sqsubseteq \sum_{i=1}^{n} c_i \cdot [\![A_i]\!]_\Pi(\sigma_{j_i}) = \sum_{i=1}^{n} \Delta_t(j_i) = \sum_{i \in I_t} \Delta_t(i),$$

where the first two steps follows by Definitions 4.3.5 and 4.3.6, respectively, the third step is justified by Equation (4.3), and, finally, the last step holds due to $I_t = \{j_1, \ldots, j_n\}$. □

Corollary 4.3.8. » *For all hypotheses H, if $\psi_H^{\forall}(S)$ then the hypothesis H is consistent with a regular sample S.* «

Proof. The statement immediately follows from Definition 4.1.14 combined with the result from Lemma 4.3.7 above. □

Corollary 4.3.9. » *For all hypotheses H, if $\psi_H^{\exists}(S_1) \Rightarrow \psi_H^{\forall}(S_2)$ then hypothesis H is consistent with an implication sample $S = S_1 \to S_2$.* «

Proof. The statement immediately follows from Definition 4.1.15 combined with the result from Lemma 4.3.7 above. □

Formal Guarantees. We conclude this section by showing how we can accomplish that our learner satisfies Lemma 4.1.20 required for our formal guarantees mentioned in Section 4.1.7. Recall that this lemma states that *any newly synthesised hypothesis is*

consistent with all samples produced by the teacher so far. Thus, according to our elaborations above, this can be achieved by synthesizing a hypothesis that satisfies

$$\bigwedge_{i=1}^{n} \psi_H^{\forall}(S_i),$$

where $S_1, \ldots, S_n$ denote the samples produced by the teacher. As indicated earlier, a concrete SMT encoding that can be leveraged to reify our specification templates to yield such a hypothesis is described in Section 4.4.

Proof (of Lemma 4.1.20). The claim follows by combining the results from Corollary 4.3.8 and Corollary 4.3.9. □

4.4 SMT Encoding

We now describe a concrete SMT encoding that can be leveraged by our learner to reify specification templates. We recall from Section 4.3.4 above that, in order for the resulting specifications to be suitable for the next hypothesis, they need to satisfy a set of snapshot constraints that can be extracted from the samples. For this section, we therefore assume that some specification templates are already given and explain how to encode the snapshot constraints as SMT formulas. Assuming the constraints are satisfiable, the learner can then reify the specification templates by invoking an off-the-shelf SMT solver, such as Z3 [79], CVC4 [9], or CVC5 [7], and then extract concrete specification from the model returned by the solver.

Section Outline. In Section 4.4.1, we start by outlining how our learner computes the reified specifications from the specification templates and the model returned by the SMT solver. This sets the context for the upcoming subsections that then further detail this template reification and also provide the concrete SMT encodings that ensure that the resulting specifications are, indeed, suitable.

The basic building blocks of our encoding are so-called guard encodings that constrain individual guards. These guard encodings are formally introduced in Section 4.4.2. In Section 4.4.3, we then show how such guard encodings can be combined to express the desired snapshot constraints. Finally, in Section 4.4.4, we briefly describe how choices represented by expression placeholders can be handled.

4.4.1 Template Reification

Below, we first briefly introduce SMT models and then outline how a specification template is reified according to such a model.

SMT Models. Let X_{SMT} denote the set containing all variables appearing somewhere in our formulas. A *model* $\mu\colon X_{\mathsf{SMT}} \to V_{\mathsf{SMT}}$ is a function mapping each of these variables $\mathfrak{x} \in X_{\mathsf{SMT}}$ to its corresponding value $\mu(\mathfrak{x})$. We write $\mu \vDash F$ to denote that the formula F is true in the model μ.

Most of the variables appearing in our encoding are boolean-typed variables; Clearly, for all of these variables $\mathfrak{x}$, we have $\mu(\mathfrak{x}) \in \{\mathsf{true}, \mathsf{false}\}$. Moreover, we model permission-typed variables $\mathfrak{q}$ as variables over rationals; that is, for all these variables, we have $\mu(\mathfrak{q}) \in \mathbb{Q}$. Therefore, we set $V_{\mathsf{SMT}} \coloneqq \{\mathsf{true}, \mathsf{false}\} \cup \mathbb{Q}$.

Reified Specifications. Given a specification template τ and some model μ, we use $\llbracket \tau \rrbracket_\mu$ to denote the specifications obtained by reifying the template τ according to the model μ. We remind ourselves that our specification templates may contain placeholders for guards, permission amounts, and expressions; the details related to these placeholders are discussed in the upcoming sections:

- In Section 4.4.2, we show our learner can compute the reified guards $\llbracket b_k^? \rrbracket_\mu$ by introducing a guard encoding that can be used to impose constraints on individual guards.
- In Section 4.4.3, we show how our guard encodings can be combined to express snapshots constraints, as well as how the concrete permission amounts $\llbracket q_k^? \rrbracket_\mu$ are computed.
- Finally, in Section 4.4.4, where we discuss how choices represented expression placeholders $e_k^?$ are handled, we also define their reification $\llbracket e_k^? \rrbracket_\mu$.

Apart from the placeholders, the reification of specification templates is defined inductively over the structure of the template in a straightforward manner:

Definition 4.4.1. » For all specification templates τ and all models $\mu\colon X_{\mathsf{SMT}} \to V_{\mathsf{SMT}}$, we define

$$\llbracket \tau \rrbracket_\mu :\equiv \begin{cases} \mathsf{true} & \text{if } \tau \equiv \mathsf{true} \\ \llbracket \tau_1 \rrbracket_\mu * \llbracket \tau_2 \rrbracket_\mu & \text{if } \tau \equiv \tau_1 * \tau_2 \\ b \Rightarrow \llbracket \tau_1 \rrbracket_\mu & \text{if } \tau \equiv b \Rightarrow \tau_1 \\ \llbracket b_k^? \rrbracket_\mu \Rightarrow \llbracket \tau_1 \rrbracket_\mu & \text{if } \tau \equiv b_k^? \Rightarrow \tau_1 \\ \mathsf{acc}(e.f, \llbracket q_k^? \rrbracket_\mu) & \text{if } \tau \equiv \mathsf{acc}(e.f, q_k^?) \\ \mathsf{rec}(e) & \text{if } \tau \equiv \mathsf{rec}(e) \\ \mathsf{seg}(e, \llbracket e_k^? \rrbracket_\mu) & \text{if } \tau \equiv \mathsf{seg}(e, e_k^?), \end{cases}$$

where – as mentioned above – the definitions of $\llbracket b_k^? \rrbracket_\mu$, $\llbracket q_k^? \rrbracket_\mu$, and $\llbracket e_k^? \rrbracket_\mu$ are given in the upcoming subsections. «

4.4.2 Guard Encoding

The guards appearing in our specification templates determine under conditions certain resources are provided by the final reified specifications. In order to ensure that these specifications provide the right resources – that is, the ones requested by the samples – we introduce a *guard encoding* that allows us to constrain individual guards. To this end, we consider an arbitrary but fixed guard placeholder $b_k^?$ and elaborate how we can encode a constraint that guarantees that the resulting guard $\{\!|b_k^?|\!\}_\mu$ is always true or false, respectively, for all states corresponding to some abstract state $\hat{\sigma}$. By the virtue of our samples, this abstract state always comes from a snapshot $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$. For our elaborations below, we assume that this snapshot is given. Note that, as the snapshot corresponds to a specific specification with identifier id – to warrant consistency – our guard placeholder $b_k^?$ has to appear in the template τ_{id} for this same specification.

Disjunctive Normal Form. We recall from Section 4.1.4 that our guards are formulas in disjunctive normal form over a given set of atoms $\{a_1^{\mathsf{id}}, \ldots, a_{m_{\mathsf{id}}}^{\mathsf{id}}\}$, for some $m_{\mathsf{id}} \in \mathbb{N}$, that depend on the parameters $\vec{x}_{\mathsf{id}}$ of the specification they appear in. Moreover, we assume that the maximal number of clauses $n \in \mathbb{N}$ that may appear in a guard is given as a parameter. Accordingly, in the following, we describe a fairly standard encoding for such a formula in disjunctive normal form; we start by introducing the underlying variables.

Clause and Literal Variables. Our guard encoding always includes exactly n clauses, but, for every $i \in \{1, \ldots, n\}$, we introduce a *clause activation variable* $\mathfrak{c}_i^k$ that indicates whether the i-th clause should be included in the guard. Moreover, for every clause and every $j \in \{1, \ldots, m_{\mathsf{id}}\}$, we introduce a *literal activation variable* $\mathfrak{l}_{i,j}^k$ and a *literal sign variable* $\mathfrak{s}_{i,j}^k$ that indicate whether the j-th literal of the i-th clause should be included in the i-th clause and the literal's sign, respectively.

Guard Reification. Having introduced all guard-related variables, we now define how to construct a concrete guard $\{\!|b_k^?|\!\}_\mu$, given some model $\mu \colon X_{\mathsf{SMT}} \to V_{\mathsf{SMT}}$ that assigns a value to all these variables:

Definition 4.4.2. » For all guard placeholders $b_k^?$ appearing in some specification template for a specification with identifier id and all models $\mu\colon X_{\mathsf{SMT}} \to V_{\mathsf{SMT}}$, we define $[\![b_k^?]\!]_\mu :\equiv \bigvee_{i=1}^{n} c_i^k(\mu)$, where

$$c_i^k(\mu) :\equiv \begin{cases} \bigwedge_{j=1}^{m_{\mathsf{id}}} l_{i,j}^k(\mu) & \text{if } \mu(\mathfrak{c}_i^k) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$l_{i,j}^k(\mu) :\equiv \begin{cases} a_j^{\mathsf{id}} & \text{if } \mu(\mathfrak{l}_{i,j}^k) = \mathsf{true} \text{ and } \mu(\mathfrak{s}_{i,j}^k) = \mathsf{true} \\ \neg a_j^{\mathsf{id}} & \text{if } \mu(\mathfrak{l}_{i,j}^k) = \mathsf{true} \text{ and } \mu(\mathfrak{s}_{i,j}^k) = \mathsf{false} \\ \mathsf{true} & \text{otherwise,} \end{cases} \tag{4.4}$$

for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_{\mathsf{id}}\}$. «

An illustration of how a guard placeholder is reified is given in Example 4.4.5 below; first, we describe the actual guard encoding.

Guard Encoding. Roughly speaking, the learner tries to identify the value of each atom a_j^{id} and then – based on the atom's values – constrains the aforementioned clause activation, literal activation, and sign variables such that the resulting guard $[\![b_k^?]\!]_\mu$ is true or false, as required.

To evaluate an atom a_j^{id} under consideration of the snapshot $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$, our learner can compute $[\![a_j^{\mathsf{id}}]\!](\varepsilon)$. Recall from Section 4.1.5 that this corresponds to checking whether the adapted atom $a_j^{\mathsf{id}}[\vec{e} \backslash \vec{x}_{\mathsf{id}}]$ is always true for all states $\sigma \in \gamma_\Sigma(\hat{\sigma})$. If neither $[\![a_j^{\mathsf{id}}]\!](\varepsilon)$ nor $[\![\neg a_j^{\mathsf{id}}]\!](\varepsilon)$ is true, the atom's value cannot be uniquely determined. In this case, its value is therefore soundly approximated by a default value $v \in \{\mathsf{true}, \mathsf{false}\}$, depending on whether the guard encoding is supposed to appear in a negative or in a positive position. With this, we are ready to give the full definition of our guard encoding:

Definition 4.4.3. » For all guard placeholders $b_k^?$, all snapshots ε, and all boolean values $v \in \{\mathsf{true}, \mathsf{false}\}$, we define

$$\varphi^k(\varepsilon, v) :\equiv \bigvee_{i=1}^{n} \left(\mathfrak{c}_i^k \wedge \bigwedge_{j=1}^{m_{\mathsf{id}}} \left(\mathfrak{l}_{i,j}^k \Rightarrow \varphi_{i,j}^k(\varepsilon, v) \right) \right)$$

$$\varphi_{i,j}^k(\varepsilon, v) :\equiv \begin{cases} \mathfrak{s}_{i,j}^k & \text{if } [\![a_j^{\mathsf{id}}]\!](\varepsilon) \\ \neg \mathfrak{s}_{i,j}^k & \text{if } [\![\neg a_j^{\mathsf{id}}]\!](\varepsilon) \\ v & \text{otherwise,} \end{cases} \tag{4.5}$$

where $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_{\mathsf{id}}\}$. «

Intuitively, in the definition above, each *literal encoding* $\varphi^k_{i,j}(\varepsilon, v)$ signifies the truth value of the i-th literal in the j-th clause: If the truth value of the atom a^{id}_j can be uniquely determined, we constrain the literal's sign variable $\mathfrak{s}^k_{i,j}$ accordingly. Otherwise, the truth value of the literal is conservatively approximated by the provided default value $v \in \{\mathsf{true}, \mathsf{false}\}$.

Our guard encoding $\varphi^k(\varepsilon, v)$, then uses clause and literal activation variables $\mathfrak{c}^k_i$ and $\mathfrak{l}^k_{i,j}$ to only select the literal encodings that actually show up in the resulting guard. The following lemma formalises the guarantees provided by this encoding.

Lemma 4.4.4. » *For all guard placeholders $b^?_k$, all snapshots ε, and all models μ, we have*

1. *if $\mu \vDash \varphi^k(\varepsilon, \mathsf{false})$ then $[\![\{\![b^?_k]\!\}_\mu]\!](\varepsilon)$, and*
2. *if $\mu \vDash \neg\varphi^k(\varepsilon, \mathsf{true})$ then $[\![\neg\{\![b^?_k]\!\}_\mu]\!](\varepsilon)$.* «

Proof. We only prove the first claim; the proof for the second claim follows analogously.

Let the guard placeholder $b^?_k$, the snapshot $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$, and the model μ be arbitrary. First, we consider an arbitrary literal – that is, we consider some indices $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_{\mathsf{id}}\}$ – and aim to prove

$$\mu \vDash \varphi^k_{i,j}(\varepsilon, \mathsf{false}) \quad \Rightarrow \quad [\![l^k_{i,j}(\mu)]\!](\varepsilon). \tag{4.6}$$

Essentially, this implication captures the statement of the lemma restricted to a single literal $l^k_{i,j}(\mu)$,. To prove this implication, we assume its left-hand side and then show its right-hand side; we thereby distinguish the following cases:

- We first consider the case where $\mu(\mathfrak{l}^k_{i,j}) = \mathsf{true}$ and $\mu(\mathfrak{s}^k_{i,j}) = \mathsf{true}$. In this case, combining the assumption $\mu \vDash \varphi^k_{i,j}(\varepsilon, \mathsf{false})$ with the definition of the literal encoding Equation (4.5) yields $\varphi^k_{i,j}(\varepsilon, \mathsf{false}) \equiv \mathfrak{s}^k_{i,j}$; this is only possible if $[\![a^{\mathsf{id}}_j]\!](\varepsilon)$ is true. Note that, by Equation (4.5), we have $l^k_{i,j}(\mu) \equiv a^{\mathsf{id}}_j$. Thus, we have $[\![l^k_{i,j}(\mu)]\!](\varepsilon)$, as required.
- The case where $\mu(\mathfrak{l}^k_{i,j}) = \mathsf{true}$ and $\mu(\mathfrak{s}^k_{i,j}) = \mathsf{false}$ is analogous to the previous case.
- In the remaining case where $\mu(\mathfrak{l}^k_{i,j}) = \mathsf{false}$, the implication holds trivially as, by Equation (4.4), we have $l^k_{i,j}(\mu) \equiv \mathsf{true}$.

The claim restricted to literals captured by Equation (4.6) above can easily be extended to clauses $c^k_i(\mu)$ and then to the entire guard $\{\![b^?_k]\!\}_\mu$ – which yields the statement to prove – by straightforward induction on the number of literals and the number of clauses, respectively. □

We wrap up the description of our guard encoding with a small example that illustrates how constraints are encoded for a guard placeholder and how to extract the resulting concrete guard.

Example 4.4.5. » Let us consider a guard placeholder $b_k^?$ appearing in some specification template τ_{id} and, for simplicity, assume that the number of clauses is set to $n = 1$ as well as there is only one single atom $a_1^{\mathsf{id}} \equiv \mathsf{node} = \mathsf{null}$, meaning $m_{\mathsf{id}} = 1$. Suppose, we want to encode that the guard should be true for all states in $\mathsf{node} \neq \mathsf{null}$. This can be achieved via the formula

$$F_1 :\equiv \varphi^k(\varepsilon_1, \mathsf{false}) \equiv \mathfrak{c}_1^k \wedge (\mathfrak{l}_{1,1}^k \Rightarrow \neg \mathfrak{s}_{1,1}^k),$$

where $\varepsilon_1 = \langle +, \mathsf{id}, \vec{x}_{\mathsf{id}}, \{\mathsf{node} \neq \mathsf{null}\}\rangle$ is a snapshot with an abstract state that captures $\mathsf{node} \neq \mathsf{null}$. Clearly, due to the first conjunct of this formula, any valid model μ_1 with $\mu_1 \vDash F_1$ must exhibit $\mu_1(\mathfrak{c}_1^k) = \mathsf{true}$. We observe that the implication of the second conjunct can be satisfied by setting $\mu_1(\mathfrak{l}_{1,1}^k) = \mathsf{false}$; the value of $\mu_1(\mathfrak{s}_{1,1}^k)$ is irrelevant in this case. Such a model μ_1 yields the guard $\{\!|b_k^?|\!\}_{\mu_1} \equiv \mathsf{true}$.

Next, suppose that we want to add a constraint that says that the guard should be false in cases where $\mathsf{node} = \mathsf{null}$. To this end, we construct the formula

$$F_2 :\equiv \neg\varphi^k(\varepsilon_2, \mathsf{true}) \equiv \neg(\mathfrak{c}_1^k \wedge (\mathfrak{l}_{1,1}^k \Rightarrow \mathfrak{s}_{1,1}^k)),$$

where $\varepsilon_2 = \langle +, \mathsf{id}, \vec{x}_{\mathsf{id}}, \{\mathsf{node} = \mathsf{null}\}\rangle$. Now, any model μ_2 with $\mu_2 \vDash F_1 \wedge F_2$ has to satisfy $\mu_2(\mathfrak{c}_1^k) = \mu_2(\mathfrak{l}_{1,1}^k) = \mathsf{true}$ as well as $\mu_2(\mathfrak{s}_{1,1}^k) = \mathsf{false}$. Such a model μ_2 corresponds to the guard $\{\!|b_k^?|\!\}_{\mu_2} \equiv \mathsf{node} \neq \mathsf{null}$. «

4.4.3 Snapshot Encoding

We now show how our guard encodings can be combined to express a snapshot constraint from Section 4.3.4. We remind ourselves that there are two flavours of snapshot constraints: $\psi_H^\forall(\varepsilon, \delta)$ and $\psi_H^\exists(\varepsilon, \delta)$. As the encodings for both versions are analogous (except that all approximation orders are flipped), we only describe the encoding for the former and omit the encoding for the latter.

In the following, we fix an arbitrary snapshot $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma}\rangle$ and recall that

$$\psi_H^\forall(\varepsilon, \delta) \quad \Leftrightarrow \quad \forall \sigma \in \gamma_\Sigma(\hat{\sigma}) \colon (\delta \sqsubseteq c \cdot [\![A]\!]_\Pi(\sigma)),$$

where $A :\equiv c \cdot H(\mathsf{id}, \vec{e})$ and $c \in \{-1, 1\}$, depending on whether the snapshot corresponds to an inhaled or exhaled specification. In order to avoid case distinctions in our definitions below, we assume that the snapshot at hand corresponds to an inhaled specification; the only difference for exhaled snapshots is that inequalities and approximation orders need to be flipped.

Moreover – also for the sake of easier illustration – we assume that the permission map δ at hand captures a single permission; that is, $\delta = \pi_{\mathsf{zero}}[l \mapsto q]$, for some heap location $l = \langle o, f \rangle$ and some permission fraction q. Permission maps capturing predicate instances can be handled analogously. And, in general, any permission map $\delta = \sum_{i=1}^{n} \delta_i$ can be decomposed into disjoint parts δ_i corresponding individual permissions and predicate instances; the corresponding snapshot constraint can than be expressed as $\bigwedge_{i=1}^{n} \psi_H^{\forall}(\varepsilon, \delta_i)$.

Permission Variables. As snapshot constraints impose bounds on permissions, we need to be able to reflect permission values in our encoding. To this end, for each permission placeholder $q_k^?$ appearing in a specification template, we introduce a corresponding *permission variable* $\mathfrak{q}_k$. As formalised by the following definition, the values of these permission variables directly correspond to the permission amounts used for the resulting specifications:

Definition 4.4.6. » For all permission placeholders $q_k^?$ and all models μ, we define $\{\!|q_k^?|\!\}_\mu \equiv \mu(\mathfrak{q}_k)$. «

In order to ensure that we only obtain meaningful permission amounts, for each permission placeholder $q_k^?$, we add the constraint $0 \leq \mathfrak{q}_k \wedge \mathfrak{q}_k \leq 1$ to our encoding.

Effective Guards. As a first step on our path towards constructing the snapshot encoding is to figure out how many permissions individual accessibility predicates appearing in the specification template at hand contribute to the heap location l. To this end, we introduce the notion of effective guards:

An *effective guard* $g = \langle \mathfrak{q}_k, F \rangle$ for a heap location l is a tuple consisting of a permission variable $\mathfrak{q}_k$ and a formula F. Intuitively, such an effective guard indicates that if the formula F is satisfied then the reification of the specification template τ at hand provides $\mathfrak{q}_k$ permissions for the heap location l.

Any given accessibility predicate $\mathsf{acc}(e.f', q_k^?)$ contributes $\mathfrak{q}_k$ permissions to the heap location l if the following conditions are met:

- First, under consideration of our snapshot ε at hand, the field access $e.f'$ needs to express the heap location $l = \langle o, f \rangle$; that is, $e \in E_\varepsilon(o)$ and $f = f'$.
- Second, all guards guarding the accessibility predicate must evaluate to true, for all states captured by the snapshot ε.

Example 4.4.7. » We consider the specification template $\tau \equiv b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val}, q_1^?)$ and, for the sake of this example, assume that the field access node.val expresses the heap

location of interest. In this scenario, an effective guard for the heap location l is given by $g = \langle \mathsf{q}_1, F \rangle$, where $F :\equiv \varphi^1(\varepsilon, \mathsf{false})$ encodes that the guard $\{\!|b_1^?|\!\}_\mu$ should be true.

Note that this means that we can require the resulting specifications to include a full permission for the field access node.val by adding the constraint $1 \leq F \,?\, \mathsf{q}_1 : 0$ to our encoding. «

Collecting Effective Guards. In general, a specification template may offer more than one way to provide permissions for a heap location. Ideally, we could compute the set containing all effective guards for our heap location l. However, as specification templates may potentially also contain predicate instances, this set can be infinite. Thus, the best we can hope for, in general, is to compute a finite but reasonable subset thereof.

To this end, we first unroll all predicate instances appearing in the specification template at hand up to a predefined depth. After that, we exhaustively collect all effective guards corresponding from this unrolled template. The following definition formalises this collection process.

Definition 4.4.8. » For all snapshots $\varepsilon = \langle \pm, \mathsf{id}, \vec{e}, \hat{\sigma} \rangle$, all heap locations $l = \langle o, f \rangle$, and all specification templates τ, we define $G_{\varepsilon,l}(\tau) \coloneqq G_{\varepsilon,l}(\tau, \mathsf{true})$, where

$$G_{\varepsilon,l}(\tau, F) :\equiv \begin{cases} G_{\varepsilon,l}(\tau_1, F) \cup G_{\varepsilon,l}(\tau_2, F) & \text{if } \tau \equiv \tau_1 * \tau_2 \\ G_{\varepsilon,l}(\tau_1, F) & \text{if } \tau \equiv b \Rightarrow \tau_1 \text{ and } [\![b]\!](\varepsilon) \\ G_{\varepsilon,l}(\tau_1, F \wedge \varphi^k(\varepsilon', \mathsf{false})) & \text{if } \tau \equiv b_k^?[\vec{e}_1 \backslash \vec{x}_{\mathsf{rec}}] \Rightarrow \tau_1 \\ \{\langle \mathsf{q}_k, F \rangle\} & \text{if } \tau \equiv \mathsf{acc}(e.f, q_k^?) \text{ and } e \in E_\varepsilon(o) \\ \varnothing & \text{otherwise,} \end{cases}$$

for all formulas F, and the snapshot in the third case is defined as $\varepsilon' \coloneqq \langle \pm, \mathsf{rec}, \vec{e}_1, \hat{\sigma} \rangle$. «

Example 4.4.9. » Let us revisit the specification templates for a loop invariant and a recursive predicate from Example 4.3.3:

$$\tau_{\mathsf{inv}} \equiv (b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val}, q_1^?)) * (b_2^? \Rightarrow \mathsf{acc}(\mathsf{node.next}, q_2^?)) * (b_3^? \Rightarrow \mathsf{rec}(\mathsf{node}))$$
$$\tau_{\mathsf{rec}} \equiv (b_4^? \Rightarrow \mathsf{acc}(\mathsf{x.val}, q_4^?)) * (b_5^? \Rightarrow \mathsf{acc}(\mathsf{x.next}, q_5^?)) * (b_6^? \Rightarrow \mathsf{rec}(\mathsf{x.next}))$$

Unrolling the definition of the recursive predicate in the loop invariant's specification template once – and omitting irrelevant conjuncts – yields the specification template

$$\tau'_{\mathsf{inv}} \equiv (b_1^? \Rightarrow \mathsf{acc}(\mathsf{node.val}, q_1^?)) * \ldots * (b_3^? \Rightarrow (b_4^?[\mathsf{node} \backslash \mathsf{x}] \Rightarrow \mathsf{acc}(\mathsf{node.val}, q_4^?)) * \ldots).$$

We easily observe that we can provide $q_1^?$ permissions for node.val by ensuring that $b_1^?$ is true. Moreover, we can provide another $q_4^?$ permissions by ensuring that $b_3^? \wedge b_4^?[\mathsf{node}\backslash\mathsf{x}]$ is true. This is exactly what is reflected by our set of effective guards

$$\{\langle \mathfrak{q}_1, \varphi^1(\varepsilon, \mathsf{false})\rangle, \langle \mathfrak{q}_4, \varphi^3(\varepsilon, \mathsf{false}) \wedge \varphi^4(\varepsilon', \mathsf{false})\rangle\},$$

where the snapshot $\varepsilon' = \langle +, \mathsf{rec}, \langle \mathsf{node}\rangle, \hat{\sigma}\rangle$ accounts for the context switch from the recursive predicate to the loop invariant. «

Next, we formulate and prove an auxiliary lemma that will later allow us to prove the correctness of our snapshot encoding.

Lemma 4.4.10. » *For all heap locations l, all permission amounts q, all specification templates τ, and all effective guards $\langle \mathfrak{q}_k, F\rangle \in G_l(\tau, \mathsf{true})$, we have*

$$\mu \vDash F \quad \wedge \quad \mu \vDash q \leq \mathfrak{q}_k \quad \Rightarrow \quad \forall \sigma \in \gamma_\Sigma(\hat{\sigma})\colon (\delta \sqsubseteq c \cdot [\![A]\!]_\Pi(\sigma)),$$

where $\delta \coloneqq \pi_{\mathsf{zero}}[l \mapsto q]$. «

Proof. Intuitively, in Definition 4.4.8, the formula F is used to accumulate the constraint under which a permission can be provided. We therefore prove a slightly more general statement that captures this accumulation: For all formulas F, all heap locations l, all permission amounts q, all specification templates τ, and all effective guards $\langle \mathfrak{q}_k, F_0\rangle \in G_l(\tau, F)$, we have

$$\mu \vDash F_0 \wedge F \quad \wedge \quad \mu \vDash q \leq \mathfrak{q}_k \quad \Rightarrow \quad \forall \sigma \in \gamma_\Sigma(\hat{\sigma})\colon (\delta \sqsubseteq c \cdot [\![A]\!]_\Pi(\sigma)),$$

where $\delta \coloneqq \pi_{\mathsf{zero}}[l \mapsto q]$.

This more general statement follows by straightforward induction on the specification template τ. The original claim then follows by setting $F :\equiv \mathsf{true}$. □

Snapshot Constraint. We now describe how to construct our snapshot constraint based on the set of effective guards $G = \{g_1, \ldots, g_n\}$ described above. We recall that each effective guards $g_i = \langle \mathfrak{q}_{k_i}, F_i\rangle$ corresponds to an accessibility predicate appearing in the specification template. Due to the way we computed them, we know that, conceptually, they all appear in different separating conjuncts. Therefore, the snapshot encoding capturing the snapshot constraint $\psi_H^\forall(\varepsilon, \delta)$ can be formulated as

$$q \leq \sum_{i=1}^{n} (F_i \; ? \; \mathfrak{q}_{k_i} : 0).$$

Example 4.4.11. » The set of effective guards from Example 4.4.9 yields the snapshot encoding

$$q \leq (\varphi^1(\varepsilon, \mathsf{false})\ ?\ \mathfrak{q}_1 : 0) + (\varphi^3(\varepsilon, \mathsf{false}) \wedge \varphi^4(\varepsilon', \mathsf{false})\ ?\ \mathfrak{q}_4 : 0),$$

where $\varepsilon' = \langle +, \mathsf{rec}, \langle \mathsf{node} \rangle, \hat{\sigma} \rangle$. «

The following lemma captures the correctness of our snapshot encoding.

Lemma 4.4.12. » *Let us consider an arbitrary snapshot ε, a permission map δ, and a model μ with $\mu \vDash q \leq \sum_{i=1}^{n}(F_i\ ?\ \mathfrak{q}_{k_i} : 0)$ as described above. Any hypothesis H with $H(\mathsf{id}) \equiv \{\!|\tau_{\mathsf{id}}|\!\}_\mu$ satisfies the snapshot constraint $\psi_H^{\forall}(\varepsilon, \delta)$.* «

Proof. The claim follows from Lemma 4.4.10 together with the observation that the accessibility predicates corresponding to the effective guards conceptually come from different separating conjuncts. □

Incompleteness. While our encoding takes the unrolling of predicate instances into account, it does not consider the possibility to obtain a predicate instance by requiring all permissions required by its body or rewriting predicate instances in other ways. Although the details might be a bit intricate, an extension of our encoding to also incorporate this is rather straightforward. In general, we can imagine situations where such a more complete encoding would be necessary to successfully synthesise a suitable next hypothesis. However, our less complete encoding was sufficient to successfully infer all specifications, for every program in our evaluation.

In Section 4.5, we will discuss how our teacher handles isorecursive predicates. We observe that, in the elaborations above, the unrolling of the predicate definitions can be seen as counterparts of the unfold statements discussed there; a more complete encoding would also include the counterparts of fold statements and lemma applications.

4.4.4 Choice Encoding

Finally, we briefly discuss how our learner handles choices represented by expression placeholders $e_k^?$. Recall that such choices are used only to allow the solver to pick a second argument for segment predicates $\mathsf{seg}(e, e_k^?)$ and are associated with a set of options $O_k = \{e_1, \ldots, e_n\}$ that was generated along with the specification templates.

Hot-Bit Encoding. For each $i \in \{1, \ldots, n\}$, we introduce an *option variable* $\mathfrak{o}_i^k$ that indicates whether the i-th option e_i is chosen or not. In order to ensure that exactly one option is chosen, we add the constraint

$$\bigvee_{i=1}^{n} \mathfrak{o}_i^k \wedge \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg(\mathfrak{o}_i^k \wedge \mathfrak{o}_j^k)$$

to our encoding. Note that the first part of this constraint ensures that at least one option is picked while the second part ensures that no two options are picked simultaneously. The reification of the expression placeholder $e_k^?$ is then defined accordingly:

Definition 4.4.13. » For any expression placeholder $e_k^?$ associated with the set of options $O_k = \{e_1, \ldots, e_n\}$, we define $[\![e_k^?]\!]_\mu := e_i$, where $i \in \{1, \ldots, n\}$ is the unique index such that $\mu(\mathfrak{o}_i^k) = \mathsf{true}$. «

Enumerating Choices. It remains to show how our learner can force the solver to choose a suitable – and not just *any* – option. To do so, the learner conceptually replaces each segment predicate $\mathsf{seg}(e, e_k^?)$ with the conjunction

$$\bigstar_{i=1}^{n} (\mathfrak{o}_i^k \Rightarrow \mathsf{seg}(e, e_i)).$$

We observe that the learner can treat the option variables $\mathfrak{o}_i^k$ just like regular boolean expressions b, which then allows us to use the snapshot encoding from the previous subsection without any further alterations.

4.5 Isorecursive Predicates

An *equirecursive* verifier treats a predicate instance as the complete unrolling of its definition [1]. This also coincides with our intuitive understanding of recursive predicates and is how our formal expositions so far treated recursive predicates. The integration of our inference with an equirecursive verifier is therefore straightforward. Most automated verifiers, however, employ an *isorecursive* semantics [100] that distinguishes between a predicate instance and its body, while providing some means to switch back and forth between the two. For the sake of our elaborations below, we assume that there are two ghost statements $\mathsf{unfold}\ \mathsf{rec}(\vec{e})$ and $\mathsf{fold}\ \mathsf{rec}(\vec{e})$ that exchange the predicate instance $\mathsf{rec}(\vec{e})$ with its body, and vice versa. Such statements are used, for example, in Viper [83] and VeriFast [59]. The user typically has to provide ghost code that explicitly tells the verifier where to unfold or fold a

predicate or how to manipulate predicate instances via lemma applications. Without such ghost code, an isorecursive verifier could report a permission failure even if the required permissions are present but folded within a predicate instance. In our setting, this would cause the teacher to produce spurious samples and thereby hinder the learning process from making progress. For example, the teacher could produce a sample requiring a permission that is already contained in the current hypothesis and the learner could end up proposing the same hypothesis again. In summary, an isorecursive verifier cannot readily be used by our teacher.

Section Outline. Throughout this section, we show how to overcome the limitations of an isorecursive verifier such that it can be used by our teacher nonetheless. In Sections 4.5.1 to 4.5.4, we first outline and then detail a strategy to automatically generate fold and unfold statements. After that, in Section 4.5.5, we briefly discuss how our teacher handles segment predicates.

4.5.1 Unfolding and Folding Strategy

In the presence of recursive predicates, the teacher has to factor in that our learning framework treats predicate instances and their bodies interchangeably. While it is already nontrivial to manually write ghost code that unfolds and folds predicate instances for correct specifications, it is even harder to automatically generate ghost code for potentially incorrect specifications. One of the challenges is to ensure that, in case of a failure, this ghost code provokes the verifier to report a suitable verification counterexample that can be leveraged to advance the inference.

Unfolding Strategy. First and foremost, the ghost code should prevent the verification from failing when sufficient permissions are present but not exposed, that is, still folded within a predicate instance. To this end, we introduce a ghost code fragment $u_d\{\!|\, A \,|\!\}$ that *statically* unfolds all predicate instances appearing in the assertion A up to a fixed depth $d \in \mathbb{N}$; the code fragments for our unfolding strategy are formally defined in Section 4.5.3 below.

Right after every point, where the query program inhales a specification $H(\mathsf{id}, \vec{e}\,)$, we then add the fragment $u_{d_1}\{\!|\, H(\mathsf{id}, \vec{e}\,) \,|\!\}$, for some unfolding depth d_1 This exposes the permissions contained within the d_1 top-most layers of the predicate instances appearing in $H(\mathsf{id}, \vec{e}\,)$. For sufficiently large choices of d_1 and an appropriate choice for the predicate instances included in the specifications, this is enough to handle any program: A loop-free sequence of statements can only access a data structure

up to a constant depth, and due to abstraction boundaries such as loop invariants, verifiers conceptually analyse straightline code.

Folding Strategy. At every point where a specification is exhaled, we potentially have to fold any predicate instance that syntactically appears in the said specification. However, as the code for which the specifications are inferred might traverse, modify, create or delete the data structures at hand, we cannot blindly mirror the previous unfolds. As the counterpart for our unfolding strategy, we therefore introduce a ghost code fragment $f_d\{\!|A|\!\}$ that *adaptively* folds required predicate instances up to a fixed depth $d \in \mathbb{N}$ if they are not already present in the verifier's permission state; the code fragments for our folding strategy are formally defined in Section 4.5.4 below.

Right before every point, where the query program exhales a specification $H(\mathsf{id}, \vec{e})$, we then add the fragment $f_{d_2}\{\!|H(\mathsf{id}, \vec{e})|\!\}$, for some folding depth $d_2 \geq d_1$. The folding depth d_2 should be at least the unfolding depth d_1, so that it is possible to entirely re-fold any previously unfolded predicate instances. If the code at hand builds up a data structure – for instance, prepends a node to a linked list – a larger fold depth is required. For our evaluation, a suitable choice was $d_2 = d_1 + 1$.

Note that this folding strategy outlined above requires querying whether a permission for a predicate instance is present or not. For our prototype implementation, we were working with a verifier that supports *permission introspection* [94], that is, provides language primitives that allow us to reflect on the permissions currently held. As this is a non-standard feature, our ghost code fragments introduced below additionally simulate the necessary permission introspection using auxiliary variables to track permissions.

4.5.2 Simulating Permission Introspection

As stated above, our fold strategy relies on knowing whether a particular predicate instance is present in the verifier's state. To this end, we now elaborate how to generate the additional ghost code that keeps track of all predicate instances that are added and removed by inhaling and exhaling specifications as well as all related ghost operations. For the sake of simplicity, we assume that there is only one recursive predicate; the extension to multiple recursive predicates is straightforward. Note that, as the recursive predicate was introduced by our learner, we can be sure that the original input program does not mention any instances of the recursive predicate; permission changes corresponding to statements from the input program are, thus, orthogonal.

Tracking Variables. We can syntactically determine all places an inhale or unfold statement of our ghost code potentially adds permissions held for a predicate instance; in the following, we will refer to them as *syntactic instances*: For an inhale A statement, these are the predicate instances appearing in the assertion A. And, for an unfold $\mathsf{rec}(\vec{e})$ statement, these are the predicate instances appearing in $A_{\mathsf{rec}}[\vec{e}, \setminus \vec{x}_{\mathsf{rec}}]$, where A_{rec} and $\vec{x}_{\mathsf{rec}}$ are the predicate body and parameters, respectively.

In the elaborations below we assume that there are n syntactic instances in total, and identify each of them with a unique index $i \in \{1, \ldots, n\}$. For each syntactic instance, we introduce a boolean variable b_i that tracks whether the instance is *alive*, that is, whether its related permission for the predicate instance has been gained but not yet lost. Moreover, we also introduce variables $\vec{\mathsf{v}}_i$ used to record the values of the predicate instance's arguments. For all arguments $\vec{e}$, we then define

$$q(\vec{e}) :\equiv \sum_{i=1}^{n} q_i(\vec{e}), \qquad \text{where } q_i(\vec{e}) :\equiv (\mathsf{b}_i \wedge \vec{\mathsf{v}}_i = \vec{e})\ ?\ 1 : 0.$$

Intuitively, each $q_i(\vec{e})$ indicates how much the i-th syntactic predicate instance contributes to the permissions held for the predicate instance $\mathsf{rec}(\vec{e})$. Thus, their sum $q(\vec{e})$ reflects the total permissions held for $\mathsf{rec}(\vec{e})$.

Permission Accounting. We now briefly outline how the aforementioned variables have to be initialised and updated in order to keep track of the permissions currently held for any predicate instance. As the execution starts with no permissions, all variables b_i are initialised as $\mathsf{b}_i := \mathsf{false}$.

At any point where a permission for a predicate instance $\mathsf{rec}(\vec{e})$ is gained through the i-th syntactic instance, we add the ghost code $\mathsf{b}_i := \mathsf{true}\ ;\ \vec{\mathsf{v}}_i := \vec{e}$ to update the tracking variables accordingly; note that – as we are dealing with loop-free code – any syntactic instance is encountered at most once. In Section 4.5.3 below, we explain how this is achieved for our ghost code realising the unfolding strategy.

Conversely, if a permission for a predicate instance $\mathsf{rec}(\vec{e})$ is lost, our ghost code goes through all syntactic instances and check whether there is a matching one that is still alive; this can be done by checking is an index $i \in \{1, \ldots, n\}$ for which $q(\vec{e}) = 1$. If such an instance is found, it is marked as dead by setting $\mathsf{b}_i := \mathsf{false}$. In Section 4.5.4 below, we show how our ghost code for the folding strategy achieves this.

4.5.3 Static Unfolding

We now formally define the ghost code fragment $u_d[\![A]\!]$ that statically unfolds predicate instances appearing in the assertion A up to a predefined depth d.

```
1  // initialise variables
2  b1 := false
3  // inhale specification
4  inhale rec(node)
5  // statically unfold "rec(node)"
6  unfold rec(node)
7  if (node ≠ null) {
8    // remember instance "rec(node.next)"
9    b1 := true
10   v1 := node.next
11 }
12 // randomly advance pointer (from original program)
13 if (coinflip()) {
14   node := node.next
15 }
16 // adaptively fold "rec(node)"
17 if (b1 ∧ v1 = node) {
18   // consume instance "rec(node)"
19   b1 := false
20 } else {
21   if (node ≠ null) {
22     // adaptively fold "rec(node.next)"
23     if (b1 ∧ v1 = node.next) {
24       // consume instance "rec(node.next)"
25       b1 := false
26     } else {
27       // no instance found
28       skip
29     }
30   }
31   fold rec(node)
32 }
33 // exhale specification
34 exhale rec(node)
```

Listing 4.6. An example showcasing our static unfolding and adaptive folding strategies to handle the recursive predicate rec(x) $\triangleq$ x $\neq$ null $\Rightarrow$ acc(x.val) $*$ acc(x.next) $*$ rec(x.next), including the ghost code simulating permission introspection. The condition coinflip() represents a nondeterministic choice and prevents us from being able to determine whether or not the predicate instance exhaled at the very end needs to be folded – just like, in general, the teacher is not due to being agnostic about the program semantics.

Definition 4.5.1. » For all assertions A and all depths $d \in \mathbb{N}$, we define

$$u_d\{\!|A|\!\} :\equiv \begin{cases} u_d\{\!|A_1|\!\}\,;\, u_d\{\!|A_2|\!\} & \text{if } A \equiv A_1 * A_2 \\ \mathsf{if}\ (b)\ \{u_d\{\!|A_1|\!\}\} & \text{if } A \equiv b \Rightarrow A_1 \\ \mathsf{unfold}\ \mathsf{rec}(\vec{e})\,;\, u_{d-1}\{\!|A_{\mathsf{rec}}[\vec{e}\backslash\vec{x}_{\mathsf{rec}}]|\!\} & \text{if } A \equiv \mathsf{rec}(\vec{e}) \text{ and } d \geq 1 \\ \mathsf{b}_i := \mathsf{true}\,;\, \vec{\mathsf{v}}_i := \vec{e} & \text{if } A \equiv \mathsf{rec}(\vec{e}) \text{ and } d = 0 \\ \mathsf{skip} & \text{otherwise,} \end{cases}$$

where A_{rec} and $\vec{x}_{\mathsf{rec}}$ denote the body and the arguments of the predicate p, respectively, and the index i in the fourth case indicates which syntactic instance we are dealing with. «

In the definition above, the first two cases are straightforward. The third case recursively unfolds all predicate instances appearing in the body of the predicate instance $\mathsf{rec}(\vec{e})$. The penultimate case updates the variables b_i and $\vec{\mathsf{v}}_i$ corresponding to the i-th syntactic instance $\mathsf{rec}(\vec{e})$ according to our elaborations in Section 4.5.2 above.

Example 4.5.2. » Lines 6 through 11 of Listing 4.6 correspond to the ghost code fragment $u_1\{\!|\mathsf{rec}(\mathsf{node})|\!\}$, where $\mathsf{rec}(\mathsf{x}) \triangleq \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.next}) * \mathsf{rec}(\mathsf{x.next})$. The first line of this fragment is $\mathsf{unfold}\ \mathsf{rec}(\mathsf{node})$, which simply unconditionally unfolds the predicate instance $\mathsf{rec}(\mathsf{node})$. The subsequent conditional statement $\mathsf{if}\ (\mathsf{node} \neq \mathsf{null})\ \{\mathsf{b}_1 := \mathsf{true}\,;\, \mathsf{v}_1 := \mathsf{node.next}\}$ implements the permission accounting; intuitively, it remembers the predicate instance $\mathsf{rec}(\mathsf{node.next})$, which is added by the $\mathsf{unfold}\ \mathsf{rec}(\mathsf{node})$ statement, given that $\mathsf{node} \neq \mathsf{null}$.

Note that the skip statement on line 28 is reached when the predicate instance $\mathsf{rec}(\mathsf{node.next})$ required to fold $\mathsf{rec}(\mathsf{node})$ is not held, but the maximal fold depth is reached. We observe that if the verifier deems this path reachable, the $\mathsf{exhale}\ \mathsf{rec}(\mathsf{node})$ at the very end of Listing 4.6 fails. If this happens during verification, there are either indeed insufficient resources to obtain the predicate instance $\mathsf{rec}(\mathsf{node})$ and the verification rightfully fails, or the picked unfold depth is not large enough and needs to be increased; in the latter case, the teacher typically produces a sample requiring a spurious resource, which can eventually be detected once the learner has accumulated an unsatisfiable set of constraints. «

4.5.4 Adaptive Folding

Next, we formally define the ghost code fragment $f_d\{\!|A|\!\}$ that constitutes the counterpart to the unfolding strategy and adaptively folds any predicate instance required to establish an assertion.

Definition 4.5.3. » For all assertions A and all depths $d \in \mathbb{N}$, we define

$$f_d\{\!|A|\!\} :\equiv \begin{cases} f_d\{\!|A_2|\!\}\,;\, f_d\{\!|A_1|\!\} & \text{if } A \equiv A_1 * A_2 \\ \mathsf{if}\ (b)\ \{f_d\{\!|A_1|\!\}\} & \text{if } A \equiv b \Rightarrow A_1 \\ g_{d,n}\{\!|\mathsf{rec}(\vec{e})|\!\} & \text{if } A \equiv \mathsf{rec}(\vec{e}) \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$g_{d,i}\{\!|\mathsf{rec}(\vec{e})|\!\} :\equiv \begin{cases} \mathsf{if}\ (\mathsf{b}_i \wedge \vec{\mathsf{v}}_i = \vec{e})\ \{\mathsf{b}_i := \mathsf{false}\}\ \mathsf{else}\ \{g_{d,i-1}\{\!|p(\vec{e})|\!\}\} & \text{if } d \geq 1 \text{ and } i \geq 1 \\ f_{d-1}\{\!|A_{\mathsf{rec}}[\vec{e} \backslash \vec{x}_{\mathsf{rec}}]|\!\}\,;\, \mathsf{fold}\ \mathsf{rec}(\vec{e}) & \text{if } d \geq 1 \text{ and } i = 0 \\ \mathsf{skip} & \text{otherwise,} \end{cases}$$

where A_{rec} and $\vec{x}_{\mathsf{rec}}$ denote the body and the arguments of the predicate p, respectively, and $i \in \{1, \ldots, n\}$. «

The ghost code fragment $f_d\{\!|A|\!\}$ essentially does the reverse of $u_d\{\!|A|\!\}$ with the exception that it makes use of the auxiliary definition $g_{d,i}\{\!|\mathsf{rec}(\vec{e})|\!\}$, which ensures that the predicate instance $\mathsf{rec}(\vec{e})$ is only folded if needed. Intuitively $g_{d,i}\{\!|\mathsf{rec}(\vec{e})|\!\}$ goes through all syntactic instances and checks whether there is a matching one that is still alive. If such an instance is found, its corresponding variable b_i is set to false to indicate that this instance is now (about to be) consumed. Otherwise, if no such instance was found after going through all syntactic instances, the second case recursively folds predicate instances appearing in the predicate's body.

Example 4.5.4. » Lines 17 through 32 of Listing 4.6 correspond to the ghost code fragment $f_1\{\!|\mathsf{rec}(\mathsf{node})|\!\}$. The condition $\mathsf{b}_1 \wedge \mathsf{v}_1 = \mathsf{node.next}$ on line 17 checks whether the predicate instance $\mathsf{rec}(\mathsf{node})$ is already present; if so, it updates $\mathsf{b}_1 := \mathsf{false}$ in order to mark the remembered predicate instance (cf. Example 4.5.2) as consumed and, otherwise, recursively adaptively folds all predicate instances appearing in the predicate's body $\mathsf{node} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{node.val}) * \mathsf{node}(\mathsf{node.next}) * \mathsf{rec}(\mathsf{node.next})$. «

4.5.5 Lemma Definitions and Applications

Occasionally, our adaptive folding strategy is not enough and the verifier needs to apply an inductive argument. This typically only happens when code iteratively traverses a recursive data structure. Such an inductive argument can be made using a *lemma method* that recursively applies fold or unfold statements in order to achieve the desired rewriting of the predicate instances; intuitively, the body of a lemma method can be seen as the proof of the lemma while the precondition and the postcondition capture the statement that is proven.

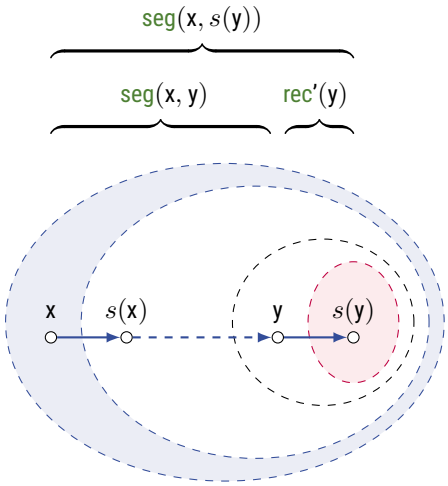


Figure 4.7. A visualisation of the layers of a segment predicate. In order to get from the predicate instance seg(x, y) to the instance seg(x, s(y)) a layer has to be added at the innermost nesting level (◌). Since fold and unfold statements only add and remove layers at the outermost nesting level (◌), an inductive argument is needed.

Below, we elaborate how our teacher can generate such lemma methods based on the inferred segment predicate and how lemma applications are handled. Recall from Section 4.3.3 that we generate the specification templates for recursive predicates in a structured form

$$\mathsf{seg}(\mathsf{x}, \mathsf{y}) \triangleq \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{rec'}(\mathsf{x}) * (b \Rightarrow \mathsf{seg}(s(\mathsf{x}), \mathsf{y}))$$

that directly corresponds to their recursive structure; ultimately, this is what allows the teacher to generate the lemma definitions fully automatically. Our evaluation demonstrates this for generic *append* and *concatenation* lemmas for segment predicates describing parts of list-like data structures.

Append Lemma. Roughly speaking, an append lemma is used whenever a segment predicate needs to be extended by one node at its end; That is, the goal is to rewrite the predicate instance seg(x, y) into seg(x, s(y)). The required permissions for this are, of course, seg(x, y), but also the permissions to form the last link: Unless y = s(y), these additional permissions correspond to the non-recursive part rec'(y) for the node y. note that, if y = s(y), the predicate instances seg(x, y) and seg(x, s(y)) are equivalent.

Conceptually – as illustrated in Figure 4.7 – the permissions rec'(y) need to be added at the innermost layer of seg(x, y). Since fold and unfold statements only allow us to add or remove the topmost layer of a recursive predicate, an inductive argument is

needed; hence the necessity of the a corresponding lemma method. A method proving the append lemma for a generic list-like segment predicate is shown in Listing A.1 in the appendix.

Concatenation Lemma. A concatenation lemma is needed whenever two adjacent predicate instances `seg(x, y)` and `seg(y, z)` need to be merged into a single instance `seg(x, z)`. A method proving the concatenation lemma for a generic list-like segment predicate is shown in Listing A.2 in the appendix.

Lemma Applications. In the same spirit as our adaptive folding strategy, our teacher also applies lemmas adaptively. Intuitively, we want to apply an individual lemma whenever the following conditions are met: First, we need to establish a predicate instance mentioned in the lemma's postcondition (which can be determined statically). Second, the permissions for the predicate instances mentioned by the lemma's precondition are already present (which can be determined dynamically using permission introspection).

This notion can easily be extended to any sequence of lemma applications, even in combination with `fold` statements. As it is not apriori clear which such sequence is suitable to establish the specifications in question, one option would be to let the teacher generate code that exhaustively enumerates all strategies consisting of sequences of lemma applications and `fold` statements, up to a predefined length, and picks the one matching our requirements. Unfortunately, such an exhaustive search quickly leads to infeasible verification times (due to a combinatorial explosion introduced by many branching statements). To alleviate this problem, as described in the following, guide our inference in cases where a lemma is needed.

Lemma Hints. We manually annotate the input program with *lemma hints* whenever lemma applications are required. A hint indicates at what point the program a lemma may be required; the hint comprises the name of the lemma and suitable arguments. Note that, even in the presence of hints, our teacher produces code that applies the lemmas adaptively, as there might be iterations for which the current hypothesis does not require the lemma application (and the verification would result in a spurious permission failure if we were to apply it anyway). Importantly, these lemma hints seamlessly integrate into our unfolding and folding strategy described above.

In general, the required hints are fairly idiomatic and therefore easy to come up with: For our entire evaluation, the following two rules were sufficient:

```
1  method contains(head: Node, key: Int) returns (found: Bool)
2    requires seg(head, null)
3    ensures seg(head, null)
4  {
5    var node: Node
6    var prev: Node
7    node := head
8    found := false
9    fold seg(head, node)
10   while (node ≠ null ∧ ¬found)
11     invariant seg(head, node) ∗ seg(node, null)
12   {
13     unfold seg(node, null)
14     if (node.val = key) {
15       found := true
16       fold seg(node, null)
17     } else {
18       prev := node
19       node := node.next
20       append_lemma(head, prev, node)
21     }
22   }
23   concat_lemma(head, node, null)
24 }
```

Listing 4.8. The fully specified running example with applications of the append lemma and concatenation lemma.

- Apply an append lemma at the end of a loop that traverses a recursive data structure to construct the segment predicate describing the part of the data structure that has already been traversed.
- Apply a concatenation lemma after a loop that traverses a recursive data structure to join the traversed part of the data structure with the left-over part, if any.

Example 4.5.5. » The contains method from our running example requires both, an application of the append lemma and an application of the concatenation lemma, if we want to establish that the predicate instance required by the precondition is not leaked and can be given back by the postcondition. The fully specified method including the lemma applications is shown in Listing 4.8. «

4.6 Evaluation

We have developed a prototype implementation of our learning-based permission inference and evaluated its effectiveness on a wide range of examples.

Section Outline. This section presents our evaluation and is structured as follows. In Section 4.6.1, we briefly discuss our implementation. Afterwards, in Section 4.6.2, we describe our benchmark and discuss the experimental results.

4.6.1 Implementation

We have developed a prototype implementation[5] of our inference for the Viper verification infrastructure [83]; our teacher uses Viper's symbolic execution verification backend [94]. The inference takes a Viper program as input and outputs the program annotated with suitable specifications, including predicate definitions. The permission model employed by our evaluation uses binary permissions, which is a subset of fractional permissions – where permission amounts are restricted to be either 0 or 1 – used for our formal presentation above. This yields a finite hypothesis space and therefore guarantees termination (cf. Section 4.1.7).

Escalating Complexity. As an optimisation aimed at obtaining simpler specifications, our implementation makes use of different levels of complexity used for the resource guards. As a measure for complexity, we used the number of clauses. At first, the

5: https://github.com/dohrau/inference

learner tries to synthesise specifications where the guards have no clauses; that is, are either true or false (cf. Section 4.4.2). If this fails, the learner then gradually (and globally) increases the number of clauses used for the guards (typically no or one clause per guard were sufficient). Once a predefined upper bound on the number of clauses is exceeded, the learner reports that it was unable to synthesise a hypothesis for the next iteration. In addition to yielding simpler specification, we also observed that this optimisation typically reduced the number of iterations required to converge to a valid hypothesis.

Syntactic Upper Bounds. Another optimisation that we implemented was to extend the SMT encoding to disallow specifications that – under consideration of unrolling of predicate definitions – contain two syntactically identical accessibility predicates. For example, our encoding does not allow the learner to infer a loop invariant acc(node.val) ∗ rec(node) while simultaneously adding the permission acc(x.val) to the recursive predicate rec(x). Without this optimisation, it occasionally happened that our learner synthesised specifications that were equivalent to false.

4.6.2 Experimental Results

Next, we describe and discuss our experimental results. An overview of the results is shown in Table 4.9. All experiments were performed on a 2020 MacBook Pro with an M1 chip and 16 GB of RAM.

Quantitative Evaluation. We evaluated our inference on a benchmark comprising a variety of programs creating, traversing, modifying, and deleting data structures. Since Viper checks for memory safety but does not perform permission leak checks, our inference typically only infers method preconditions and loop invariants for programs consisting of a single method. In order to also get method postconditions, we added a client that calls the method of interest and then deallocates the data structure at hand in order to enforce that the permissions are not leaked.

The first two groups of programs in our benchmark suite comprise a wide range of non-recursive, list-like, and tree-like data structures. The programs making up the second group demonstrate our inference's ability to infer segment predicates typically needed for specifications for iterative implementations traversing or modifying list-like data structures. As elaborated in Section 4.5.5, these were the only programs for which we used lemma hints.

	Lines	Methods	Calls	Loops	Specifications	Lemma Hints	Unfold Depth	Iterations	Samples	Total Time [s]	Verifier [s]
cell_swap.vpr	32	4	5	0	8	0	0	5	6	2.54	2.38
cell_swap_alias.vpr	29	4	3	0	8	0	0	6	7	2.73	2.59
list_contains_rec.vpr	34	4	5	0	8	0	1	8	13	3.79	3.58
list_copy_rec.vpr	37	4	6	0	8	0	1	10	19	7.75	7.52
list_create_it.vpr	45	5	5	1	11	0	1	8	14	4.22	4.02
list_create_rec.vpr	49	5	6	0	10	0	1	7	12	3.78	3.64
list_delete_it.vpr	21	2	1	1	5	0	1	9	13	4.86	4.68
list_delete_rec.vpr	17	2	2	0	4	0	1	5	6	2.03	1.91
list_equal_rec.vpr	44	4	6	0	8	0	1	17	33	22.09	21.60
list_filter_rec.vpr	49	4	6	0	8	0	1	7	15	3.74	3.50
list_mutual_rec.vpr	51	5	7	0	10	0	1	9	19	5.53	5.35
list_reverse_it.vpr	39	4	4	1	9	0	1	12	26	10.58	10.26
list_sum_rec.vpr	37	4	5	0	8	0	1	8	17	3.58	3.42
list_zip_rec.vpr	39	4	6	0	8	0	1	9	20	7.33	7.13
sorted_insert_rec.vpr	49	5	6	0	10	0	1	9	17	5.19	5.02
sorted_merge_rec.vpr	51	4	6	0	8	0	1	12	24	14.86	14.61
sorted_remove_rec.vpr	43	4	6	0	8	0	2	8	18	5.53	5.34
stack_pop.vpr	34	4	4	0	8	0	2	8	17	4.74	4.56
stack_push.vpr	39	5	5	0	10	0	1	12	20	7.33	7.10
tree_create_rec.vpr	58	5	8	0	10	0	1	17	28	105.21	104.79
tree_delete_rec.vpr	20	2	3	0	4	0	1	7	9	3.35	3.22
tree_sum_rec.vpr	43	4	7	0	8	0	1	17	36	96.57	96.05
bst_contains_rec.vpr	47	4	7	0	8	0	1	9	25	10.43	10.21
bst_insert_rec.vpr	62	5	8	0	10	0	1	12	21	19.76	19.53
list_contains_it.vpr	44	4	4	1	9	2	1	9	19	6.42	6.19
list_copy_first_it.vpr	35	4	4	1	9	1	1	8	15	4.85	4.66
list_create_append_it.vpr	60	5	6	1	11	2	1	10	22	8.35	7.87
list_filter_it.vpr	53	4	5	1	9	2	2	12	23	36.53	36.16
list_sum_it.vpr	37	4	4	1	9	1	1	9	22	5.66	5.43
sorted_insert_it.vpr	55	5	5	1	11	2	2	10	23	20.88	20.63
sorted_remove_it.vpr	57	5	5	1	11	2	2	15	33	37.48	37.08
lifo_channel.vpr	72	6	4	2	15	0	2	17	31	64.02	63.43
lock_invariant.vpr	67	4	4	3	14	0	0	8	12	4.74	4.43
nagini.vpr	541	4	3	0	8	0	0	6	8	5.89	5.60
tree_to_dll.vpr	62	4	6	0	8	0	1	11	24	31.84	31.32
faulty_functional.vpr	37	4	5	0	8	0	0	-	-	-	-
faulty_twice.vpr	7	1	0	0	2	0	0	-	-	-	-
faulty_infinite.vpr	21	2	1	1	5	0	0	-	-	-	-

Table 4.9. Our experimental results. For each program, we list numer of lines of code, methods, method calls, and loops. Moreover, we indicate how many specifications are inferred and the number of hints for lemmas used. We report the smallest unfold depth d with which the inference succeeded (the adaptive fold depth was always chosen to be $d + 1$), how many iterations it took, the number of samples produced by the teacher, and the total time as well as the time spent waiting for the verifier.

Qualitative Evaluation. Apart from the aforementioned benchmark, we ran some additional qualitative experiments that evaluated other aspects of our technique; these correspond to the third and last group of programs in our benchmark.

- The first two of these programs shows that our inference can be leveraged to infer permission specifications for high-level programming concepts: For these programs we introduced placeholder predicates and inhaled and exhaled the related predicate instances wherever a permission transfer happened (cf. Section 4.2.1). Our inference was able to infer suitable definitions for these predicates – notably, without understanding which concepts they encoded:

  The program lifo_channel.vpr encodes of a producer and a consumer thread that repeatedly and concurrently write to and read from, respectively, a last-in-first-out channel. Our inference successfully inferred the *channel invariant* describing the stack-based data structure at hand.

  The program lock_invariant.vpr forks and then joins a variable number of threads; each of the threads operates on local data as well as on shared data guarded by some lock. Our inference successfully inferred that the permissions to the local data had to be passed via the *thread preconditions and postconditions* while the permission to the global data could only be transferred via the *resource invariant* associated with the lock.

- The program nagini.vpr is the VIPER encoding of an unspecified Python program generated by the verification frontend NAGINI [39]. Frontend encodings are notoriously verbose and obfuscated; therefore, one might think it is easier to infer the specifications on the source code level. However, being able to infer them on the level of the encoding allows us to readily reuse our implementation.

- The program tree_to_dll.vpr demonstrates that our inference works in the presence of partial permission specifications: It implements a recursive method that takes a binary tree as input and – reusing the left and right fields of each node – transforms it into a doubly-linked list. While the permission specifications for the binary tree was given, the predicate for the doubly-linked list was successfully inferred.

  Unfortunately, as our inference only infers permission specifications, it does not infer the constraint that the left field of any node actually points back to the previous node in the list. Specifically, the predicate inferred for the doubly linked list by our inference is $\mathsf{rec}(\mathsf{x}) \triangleq \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.left}) * \mathsf{acc}(\mathsf{x.right}) * \mathsf{rec}(\mathsf{x.right})$. However, the definition of this predicate, can easily be manually extended

to incorporate the constraint about back pointers by adding the conjunct $\mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{unfolding}\ \mathsf{rec}(\mathsf{x.right})\ \mathsf{in}\ \mathsf{x.right.left} = \mathsf{x}$; note that the unfolding expression instructs our isorecursive verifier to temporarily unfold the recursive instance rec.right in the evaluation of the condition $\mathsf{x.right.left} = \mathsf{x}$.

- Finally, the last three programs in our benchmark demonstrate that our inference also correctly handles faulty programs. The program faulty_functional.vpr contains a statement that potentially divides by zero; our inference correctly reports that the verification failed due to an error that cannot be fixed by inferring permission specifications.

  If the program is not memory safe, our inference will either yield an unsatisfiable specification or abort because the learner accumulated an unsatisfiable set of constraints; due to our optimisations mentioned in Section 4.6.1, we only observed the latter. For example, the program faulty_twice.vpr consists of a main method forking two threads that concurrently write to the same memory location.

  The final program faulty_infinite.vpr contains a loop that indefinitely iterates over a loop end never terminates. the execution of such a loop requires an infinite predicate which cannot be constructed and therefore is equivalent to false. Also in this case, our inference correctly reports that it is not possible to infer specifications.

Conciseness. We manually inspected all inferred specifications and observed that they closely resemble manually written ones; in particular, they are typically equivalent and consist of an equal number of conjuncts.

For instance, for the contains method from our running example – which was introduced in Listing 4.2 and corresponds to list_contains_it.vpr in Table 4.9 – our inference infers the precondition pre, postcondition post, invariant inv, and recursive predicate seg given by the following final hypothesis:

$$
\begin{aligned}
H(\mathsf{pre}) &\equiv \mathsf{seg}(\mathsf{head}, \mathsf{null}) \\
H(\mathsf{post}) &\equiv \mathsf{seg}(\mathsf{head}, \mathsf{null}) \\
H(\mathsf{inv}) &\equiv \mathsf{seg}(\mathsf{head}, \mathsf{node}) * \mathsf{seg}(\mathsf{node}, \mathsf{null}) \\
H(\mathsf{seg}) &\equiv \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.next}) * \mathsf{seg}(\mathsf{x.next}, \mathsf{y})
\end{aligned}
$$

These are precisely the specifications from the fully specified example shown in Listing 4.8 and likely what a programmer would write themself.

We observed that, occasionally, our inference produces specifications that contain unnecessary guards or additional conjuncts; importantly, however, never at the

expense of the specifications' correctness or satisfiability. To illustrate this, let us look at two concrete examples. First, let us consider consider the program list_copy_rec.vpr that contains a method that copies a list; for this method, our inference produces the precondition

$$\mathsf{list} \neq \mathsf{null} \Rightarrow \mathsf{rec}(\mathsf{list}),$$

where the condition $\mathsf{list} \neq \mathsf{null}$ is unneccessary as it is already included in inferred the predicate $\mathsf{rec}(\mathsf{x}) \triangleq \mathsf{x} \neq \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{x.val}) * \mathsf{acc}(\mathsf{x.val}) * \mathsf{rec}(\mathsf{x.next})$. Next, we turn our attention to the program lifo_channel.vpr that contains a method that pops a value from the stack that is used to implement the channel; for this method, our inference produces the precondition

$$(\mathsf{stack} = \mathsf{null} \Rightarrow \mathsf{acc}(\mathsf{stack.next})) * \mathsf{rec}(\mathsf{stack})\,. \tag{4.7}$$

From a permission perspective, the first conjunct is not needed. It implicitly requires that $\mathsf{stack} \neq \mathsf{null}$, as it is impossible to hold a permission for null.next; in other words, it requires the stack to be non-empty and, thus, prevents the method to be used to pop a value from an empty stack, which would lead to a failure. Interestingly – as our inference only infers permission specifications – adding this seemingly spurious first conjunct in Equation (4.7) is the only way the inference can enforce the condition $\mathsf{stack} \neq \mathsf{null}$ and guarantee memory safety. Notably, manually adding the condition $\mathsf{stack} \neq \mathsf{null}$ to the precondition of the method causes our inference to not infer this first conjunct of Equation (4.7) above.

Conclusion. Our inference runs in reasonable time, which for the most part is spent by the verifier – on average, more than 95%. All inferred specifications are sound, concise, and satisfiable:

- As already discussed in Section 4.1.7, our inference is sound by construction: The inferred specifications contain sufficient permissions to successfully verify the program.
- As mentioned above, we observed that all inferred specifications closely resemble manually written ones; they are typically equivalent and consist of an equal number of conjuncts.
- Finally, we also manually checked that the specifications are satisfiable; that is, they do not require more than one permissions for a single memory location and predicate instances appearing within them can actually be constructed.

Thanks to the automatically generated ghost code, our inference can effectively leverage an isorecursive verifier to infer specifications including recursive predicates.

Moreover, this ghost code allows one to successfully verify the annotated output program. Although the ghost code is not minimal, we manually confirmed that it can easily be reduced using a few elementary simplification rules (for instance, dead code elimination). Overall, our evaluation shows that our inference can be applied to a wide range of data structures and code patterns to infer all specifications at once, even in the presence of partial functional or permission specifications.

4.7 Discussion

Section Outline. In Section 4.7.1, we first compare our technique with related work. Then, in Section 4.7.2, we then summarise its strengths and limitations. Finally, Section 4.7.3, we conclude this chapter and briefly outline possible future directions.

4.7.1 Related Work

Several powerful inference techniques for permission specifications have been developed, but they do not satisfy all of our practicality requirements – comprehensive specifications, wide applicability, and specification process – identified at the beginning of this chapter. In particular, many approaches are limited to a set of predefined predicates, or they cannot infer all specifications and permission-related annotations that typical verifiers require. Below, we first discuss the most relevant permission inference techniques.

Neider et al. [84] present a black-box inference also based on the ICE framework [47] and also integrates an existing verifier. Their inference needs predicate definitions and method preconditions (which we both infer), and produces loop invariants and method postconditions with permissions and value constraints (we do not infer the latter). Due to the integration of a concrete verifier, their specifications are, by construction, readily verifiable, and their inference does not need to duplicate the semantics of the programming language; both are also the case for our approach.

Guo et al.'s [52] white-box inference used inductive recursion synthesis to infer predicate definitions for tree-like data structures as well as method summaries and loop invariants. They handle complex recursive data structures (back-pointers, composed, overlaid, and nested with arrays), including partial versions thereof, which are expressed via the magic wand connective [92]. Their inference requires the presence of code that constructs an instance of the data structure to reveal its *recursive backbone* and is, thus, not fully modular. Moreover, it is unclear how much manual effort is necessary to use the inferred specifications in an existing verifier, for example, because additional annotations are needed for working with magic wands.

Le et al.'s [66] inference needs to be integrated into a verifier with a second-order bi-abduction separation logic solver; this is a strong requirement, given that most automated verifiers are based on SMT solvers. Their inference produces predicate definitions, corresponding folding and unfolding lemmas for the entailment solver, and all program specifications. Their approach can handle complex data structures, but does not support loops (which we do), and it remains unclear how to extend the approach to infer predicates for partial data structures and the corresponding ghost code needed for loops.

Next, we give a broader overview over other closely related techniques for specification inference and for heap analysis. Many existing inference techniques for permission specifications are tailored to specific data structures or only support predefined predicates. For instance, the dynamic shape analysis by Mühlberg et al. [81] and the static shape analysis by Berdine et al. [11] are limited to linked lists. This restriction allows them to infer permission specifications for more complex list structures, but comes at the expense of not supporting other recursive data structures.

More common are approaches that infer specifications involving recursive predicate instances, but not the corresponding predicate definitions: these must be provided by users, potentially with further input, such as proof rules for their folding and unfolding. Examples are Calcagno et al.'s seminal bi-abduction inference [22], Vogels et al.'s VeriFast-integrated inference [102], Lee et al.'s shape analysis for overlaid data structures [68], Qin et al.'s inference for shape and value constraints [90], and Le et al.'s debugger-based dynamic analysis [67], and Neider et al.'s learning-based approach mentioned above [84]. In contrast to these techniques, our approach also infers the definitions of the recursive predicates, and the ghost code necessary to manipulate them, contributing to the *comprehensive specifications* requirement from the introduction of this chapter. Similarly, Calcagno et al.'s bi-abductive synthesis for resource invariants [23] assumes that the user annotates the input program with thread preconditions, while we infer all specifications.

Many other inference techniques produce specifications that are not intended for permission-based verification. Examples include the grammar-based shape analysis by Lee et al. [69], Holík et al.'s shape analysis [58], and Ferrara et al.'s shape analysis [43] from the long line of abstract interpretation based invariant inferences. It is unclear if the specifications resulting from these techniques can be transformed and extended automatically to obtain permission specifications directly usable by (isorecursive) verifiers, as required by our *specification process* requirement.

Brotherston et al. propose a cyclic abduction technique [20] to infer predicate definitions, and method preconditions with safety and termination specifications.

Their approach, however, does not handle procedure calls, since this would require the ability to infer method preconditions and postconditions simultaneously. Therefore, this work does not satisfy our *wide applicability* requirement, and neither do Guo et al.'s [52] and Le et al.'s [66] inferences, both already mentioned above.

4.7.2 Strengths and Limitations

We now briefly discuss some of the strengths and limitations of our approach.

Strengths. Our inference is capable of inferring multiple specifications at once; that is, it is not limited to inferring a single specification but rather can simultaneously infer all method preconditions and postconditions, loop invariants and potentially also other permission specifications for a program. Moreover, our inference is also able to infer the definition of the recursive predicate used in the inferred specifications and does not rely on predefined or user-defined predicates. In addition to the specifications, our inference also generates ghost code to successfully verify the annotated output program using an isorecursive verifier.

Since our technique builds upon a learning-based approach, we also inherit some of it strengths. Most importantly, the black-box nature of our approach allows the learner to be unaware of the semantics of the program at hand. Therefore, it can synthesise permission specifications for arbitrarily complex programs, as long as their permission specifications reside within our hypothesis space.

While some of the existing work views the teacher as a white-box component that needs to understand the semantics of the program at hand, our teacher can be seen as using the verifier as an oracle: Although, the teacher needs to understand where the verifier syntactically expects specifications, it does not need to understand the semantics of the input program.

Limitations. Our inference is not guaranteed to infer weakest possible method preconditions and loop invariants. Weaker specifications are generally desired as they provide stronger framing. Possible approaches to weaken our inferred specifications could be to perform satisfiability checks or – even better – to incorporate an additional analysis that checks which permissions are added to the specifications but *not* required to execute the code at hand.

When inferring specifications for individual methods, we require client code in order to also obtain a postcondition. Note, however, that this limitation mostly stems from the fact that the verifier used for our evaluation does not check whether permissions are leaked. We believe that our framework could also incorporate leak

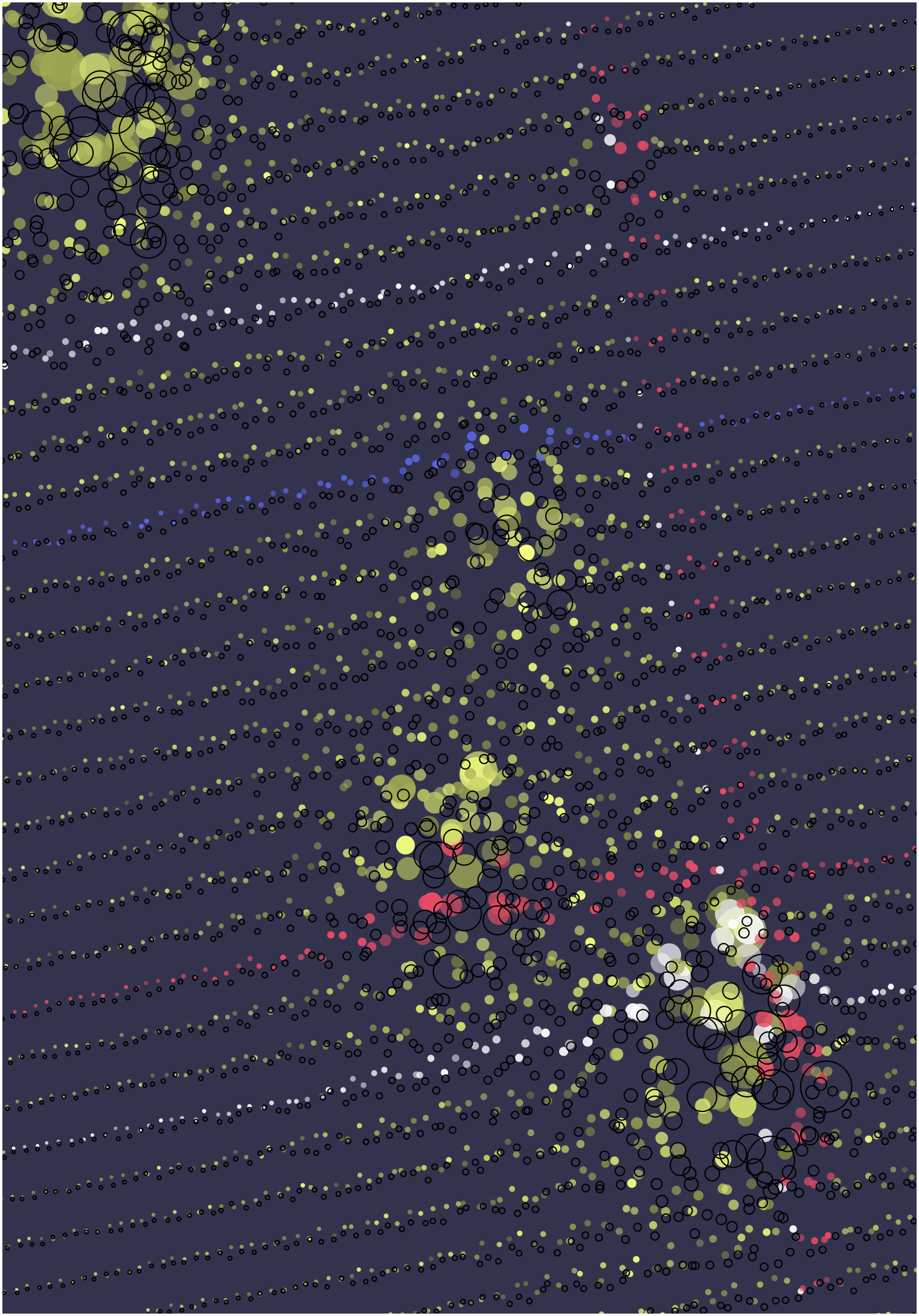
checks with some modifications to the teacher: For example, the teacher could produce an implication sample requiring that if a permission is added to a method's precondition then it must also be added to the method's postcondition.

4.7.3 Conclusion and Future Directions

We presented a black-box inference that uses abstract samples over traces to effectively infer all specifications along with related ghost code required to automatically and successfully verify heap manipulating programs. We demonstrated that our inference can handle a wide range of data structures and code patterns, even in the presence of partial functional and permission specifications.

Future Work. Based on the strengths and limitations of our inference in its current form, there are many possible direction one could pursue. One rather obvious future direction is to extend our inference to also infer functional specifications; that is, to infer value constraints that complement the permission specifications. A possible approach is to first run our permission inference and then a value analysis on top of it. However, we believe that it is possible to infer permission and value specifications simultaneously by extending our samples and encoding with value constraints, for instance, similar to the octagonal constraints described by Garg et al. [47]. Such a unified approach has the potential to infer stronger specifications compared to consecutively running a permission and a value analysis; as an example, a predicate capturing permissions of all elements of a sorted list up to a given value can likely only be inferred if the permission inference is aware of value constraints.

Moreover, it would be interesting to explore extensions to our ghost code generation to support richer specifications, for instance, for composed and overlaid data structures. More details on this and other extensions are discussed in Section 5.2.



5 Conclusion and Future Work

In this chapter, we conclude the work at hand and briefly discuss possible future directions.

5.1 Conclusion

In this thesis, we have explored automatic inference techniques for permission specifications. Concretely, we have developed and implemented two complementary permission inference technique that both cover an important class of heap manipulating programs.

Array Programs. The first inference technique that we have developed is a permission inference for array manipulating programs. We have seen that expressing permission specifications as numerical expressions has two key advantages. First, this allows us to capture the permission amounts for all array elements using a single permission expression parameterised by program variables. Second, we can express the permissions required by loops via maximum expressions over the results obtained for individual loop iterations. While most existing array analyses focus on array content, our inference is the first technique that automatically infers permission specifications for array programs that also handles loops. In particular, our approach is capable of inferring all permission specifications required by a typical permission-based verifier.

Furthermore, for our construction of the permission invariants, we introduced the novel concept of *progressive loop invariants*. We showed how an off-the-shelf numerical analysis can be leveraged to infer such progressive loop invariants. We believe that this variation of loop invariants is useful beyond the scope of our permission inference, for instance, to express progress and termination properties for loops.

In addition, we also introduced a novel *maximum elimination* algorithm that – in the spirit of quantifier elimination – can be used to solve maximum expressions ranging over an unbounded set of values by rewriting them into an equivalent expression consisting of finitely many binary maximum expressions.

Black-Box Inference. The second inference technique that we have developed is a learning-based approach targeted at inferring permissions for individual heap locations and recursively defined data structures. Our approach is based on the ICE framework and employs samples defined over symbolic traces rather than concrete individual states. This enables our inference to be the first one that automatically infers all permission specifications required by a typical permission-based verifier; including method precondition and postconditions, loop invariants, and, crucially, also the definitions of recursive predicates. Moreover, we also extended our inference to support isorecursive verifiers – which is the norm for automated permission-based program verifiers – by automatically generating all ghost code required to manipulate isorecursive recursive predicates.

Furthermore, we have shown how our inference can be used to infer permission specifications for high-level concepts, such as thread preconditions and postconditions or resource invariants by encoding all related permission transfers as an inhale or exhale of a placeholder predicate. This underlines the generality of our approach and could be leveraged, for example, by a hypothetical verification frontend to use our inference (without the need for any retrofitting) to infer permission specifications for language features allowing such an encoding but not necessarily anticipated by us.

Compatibility. Although this thesis presents both of the aforementioned inference techniques separately, there are many cases in which they can be combined in an out-of-the-box manner to infer permission specifications for programs that operate on both, arrays and recursively defined data structures. This is possible as long as the parts of the specifications inferred by each technique do not depend on one another; typically, this the case when the program does *not* make use of nested structures, that is, does not operate on arrays of lists, trees with array-typed value fields, or alike.

For the sake of our elaborations below, let us assume that we want to infer permission specifications for a program admitting such independent permission specifications. As our permission inference for array programs is solely based on *syntactic* analyses and transformations, it is capable of inferring all array-related permission specifications by ignoring non-array heap accesses and without taking into account any other permission specifications. In contrast, our black-box inference is agnostic of heap accesses permitted by any existing partial permission specifications and can – assuming that the input program is already annotated with suitable array-related permission specifications – infer the remaining permission specifications.

In summary – and as illustrated below by Example 5.1.1 – there are many programs for which we can infer *all* permission specifications by first running our permission

```
method swap_combined(a: Int[], head: Node) {
  var i: Int
  var t: Int
  var node: Node
  i := 0
  node := head
  while (i < len(a) ∧ node ≠ null) {
    // swap values
    t := a[i]
    a[i] := node.val
    node.val := t
    // advance
    i := i + 1
    node := node.next
  }
}
```

Listing 5.1. A method swapping the values stored in an array and linked lists. We can infer all permission specifications required by a verifier by first running the permission inference presented in Chapter 3 and then the one from Chapter 4.

inference for array programs, followed by running the black-box permission inference. Note that running the inference techniques in reverse order does not work without further adjustments since the teacher of our black-box inference does not know how to handle array-related permission failures reported by the verifier, and would require some sort of program slicing [104].

Example 5.1.1. » Let us consider the swap_combined method shown in Listing 5.1 that swaps the values stored in an array and a linked lists. Running our permission inference for arrays on this program yields $\forall q_i \in \mathbb{Z}\colon \mathsf{acc}(a[q_i], (0 \leq q_i \wedge q_i < \mathsf{len}(a))\ ?\ 1 : 0)$ as method precondition and postcondition, as well as loop invariant. Annotating the input program with these specifications yields a program that, when verified, only causes permission failures related to the linked list. That is, we can run our black-box permission inference in this partially specified program to obtain the permission specifications related to the linked list. We experimentally verified that the resulting specifications are indeed sufficient to successfully verify the swap_combined method. «

In general, permission specifications and recursively defined data structures may depend on each other (as stated above, to specify permissions for arrays of lists, for example). In order to also handle such programs, there is probably no way around developing a unified permission inference capable of inferring all permission

specifications simultaneously; for example, by extending our black-box inference to also handle quantified permissions.

5.2 Future Directions

There are several ways in which the inference techniques presented in this thesis could be extended or complemented.

Magic Wands. In light of recent advances in automating magic wands [32], it would be interesting to explore inference techniques that infer specifications making use of them. A magic wand [86], written $A_1 \mathrel{-\!\!*} A_2$, is the separation logic counterpart of the logical implication and, intuitively, expresses the difference between the assertions A_2 and A_1. This makes them very useful to describe partial data structures; For example, let us recall the loop invariant seg(head, node) $*$ seg(node, null) from Example 4.5.5, where seg(head, node) captures the part of the linked list that has already been traversed and seg(node, null) the remaining part. Using magic wands, we can express an equivalent invariant rec(node) $*$ (rec(node) $\mathrel{-\!\!*}$ rec(head)) without the need for a segment predicate; here, the predicate instance rec(node) provides sufficient permissions to execute the remainder of the loop, whereas the magic wand list(node) $\mathrel{-\!\!*}$ list(head) allows us to regain the permission to the entire linked list once the loop terminates.

An advantage over an approach employing segmented versions predicates would be that magic wands do not only allow one to easily specify parts of list-like data structures but also naturally generalise to partial versions of arbitrary recursively defined data structures; that is, such an approach would likely also be able to support code that iteratively traverses tree-like data structures, for example. Similar to the fold and unfold operations for predicates, automated verifiers supporting magic wands require package and apply operations to establish a magic wand and apply it, respectively. Ideally, an inference could infer these ghost operations automatically.

We believe that our black-box permission inference from Chapter 4 could serve as a starting point, and be adapted or extended to support magic wands. For example, instead of allowing the learner to introduce segment predicate, we could try allowing it to introduce magic wands. One of the main challenges would be the automatic generation of suitable package and apply operations; it is conceivable that a strategy based on permission introspection – similar to the one we used for unfolding and folding predicate instances – could work.

Richer Specifications. Another possible future direction could be to extend the ghost code generation of our black-box inference approach to infer permission specifications

for a wider range of programs operating on a richer set of data structures. There are several directions one could pursue; examples include but are not limited to (i) inferring recursive predicate for nested data structures, for example n-ary trees that store the children of each node in a linked list, (ii) supporting overlaid data structures, where two or more data structures share a set of nodes, (iii) supporting additional lemmas, for example the counterpart of the concatenation lemma that splits a linked-lists segment in two parts or the counterpart of the append lemma that splits off the last element of a segment, or (iv) handling segmented versions of general tree-like data structures. Note that the first two directions would also require substantial changes in the part of the learner that is concerned with detecting potential instances of recursive data structures. As indicated above, a promising approach that could be used to tackle the last direction would be based on magic wands.

Graph-Like Structures. An important class of data structures not covered by either of our permission inference techniques presented in this thesis are linked data structures with less structure than, say a linked list or a binary tree, which cannot easily be described with a recursive predicate. In particular, this class includes more general graph-like data structures with multiple possible paths between nodes and potentially also consisting of loops and cycles. Such data structures are typically specified using quantified permissions ranging over a set of references R satisfying some closure property; hereby, the set R would be part of the ghost state and – in an interprocedural setting – be passed as a ghost parameter from one method to another. Maybe the simplest example of such a data structure is a cyclic list that can be described by the assertion $\forall \mathsf{node} \in \mathsf{Nodes} : \mathsf{acc}(\mathsf{node.next})$, where Nodes is a set of references satisfying $\mathsf{null} \notin \mathsf{Nodes}$ and $\forall \mathsf{node} \in \mathsf{Nodes} : \mathsf{node.next} \in \mathsf{Nodes}$.

It would be interesting to investigate permission inference techniques for such graph-like data structures. Possible starting points would be to explore whether either of the two inference techniques proposed in this thesis could be extended or adapted to do so. Regardless of the chosen approach, a challenging open question would be to automatically infer ghost code required to manipulate the set of references R for code that does not only traverse but also modify the data structure at hand.

Bibliography

[1] Martín Abadi and Marcelo P Fiore, "Syntactic considerations on recursive types," in *LICS 1996: Logic in Computer Science*, IEEE, year, pp. 242–252. DOI: 10.1109/LICS.1996.561324.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, Eds., *Deductive Software Verification – The KeY Book, From Theory to Practice*, ser. Lecture Notes in Computer Science 1. Springer, 2016. DOI: 10.1007/978-3-319-49812-6.

[3] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina, "Definability of accelerated relations in a theory of arrays and its applications," in *FroCoS 2013: Frontiers of Combining Systems*, ser. Lecture Notes in Computer Science, vol. 8152, Springer, 2013, pp. 23–39. DOI: 10.1007/978-3-642-40885-4_3.

[4] ——, "Booster: An acceleration-based verification framework for array programs," in *ATVA 2014: Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, vol. 8837, Springer, 2014, pp. 18–23. DOI: 10.1007/978-3-319-11936-6_2.

[5] Rajeev Alur, Pavol Cerný, Parthasarathy Madhusudan, and Wonhong Nam, "Synthesis of interface specifications for Java classes," in *POPL 2005: Principles of Programming Languages*, ACM, 2005, pp. 98–109. DOI: 10.1145/1040305.1040314.

[6] Rajeev Alur, Parthasarathy Madhusudan, and Wonhong Nam, "Symbolic compositional verification by learning assumptions," in *CAV 2005: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 3576, Springer, 2005, pp. 548–562. DOI: 10.1007/11513988_52.

[7] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar, "Cvc5: A versatile and industrial-strength SMT solver," in *TACAS 2022: Tools and Algorithms for the Construction*

and Analysis of Systems, ser. Lecture Notes in Computer Science, vol. 13243, Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-642-22110-1_14.

[8] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO 2005: Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, vol. 4111, Springer, 2005, pp. 364–378. DOI: 10.1007/11804192_17.

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli, "CVC4," in *CAV 2011: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[10] Nils Becker, "Inference of progressive loop invariants for array programs," Master's Thesis, ETH Zurich, 2020.

[11] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang, "Shape analysis for composite data structures," in *CAV 2007: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 178–192. DOI: 10.1007/978-3-540-73368-3_22.

[12] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *FMCO 2005: Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, vol. 4111, Springer, 2005, pp. 115–137. DOI: 10.1007/11804192_6.

[13] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival, "Static analysis and verification of aerospace software by abstract interpretation," in *AIAA 2010: American Institute of Aeronautics and Astronautics*, AIAA, 2012, pp. 1–38. DOI: 10.2514/6.2010-3385.

[14] Stefan Blom, Saeed Darabi, and Marieke Huisman, "Verification of loop parallelisations," in *FASE 2015: Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, vol. 9033, Springer, 2015, pp. 202–217. DOI: 10.1007/978-3-662-46675-9_14.

[15] Stefan Blom and Marieke Huisman, "The VerCors tool for verification of concurrent programs," in *FM 2014: Formal Methods*, ser. Lecture Notes in Computer Science, vol. 8442, Springer, 2014, pp. 127–131. DOI: 10.1007/978-3-319-06410-9_9.

[16] John Boyland, "Checking interference with fractional permissions," in *SAS 2003: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 2694, Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4.

[17] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný, and Thomáš Vojnar, "Automatic verification of integer array programs," in *CAV 2009: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 157–172. DOI: 10.1007/978-3-642-02658-4_15.

[18] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet, "IKOS: A framework for static analysis based on abstract interpretation," in *SEFM 2014: Software Engineering and Formal Methods*, ser. Lecture Notes in Computer Science, vol. 8702, Springer, 2014, pp. 271–277. DOI: 10.1007/978-3-319-10431-7_20.

[19] Stephen Brookes, "A semantics for concurrent separation logic," *Theoretical Computer Science*, vol. 375, no. 1, pp. 227–270, 2007. DOI: 10.1016/j.tcs.2006.12.034.

[20] James Brotherston and Nikos Gorogiannis, "Cyclic abduction of inductively defined safety and termination preconditions," in *SAS 2007: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 8723, springer, 2007, pp. 68–84. DOI: 10.1007/978-3-319-10936-7_5.

[21] James Brotherston, Nikos Gorogiannis, and Max Kanovich, "Biabduction (and related problems) in array separation logic," in *CADE 2017: Automated Deduction*, ser. Lecture Notes in Computer Science, vol. 10395, Springer, 2017, pp. 472–490. DOI: 10.1007/978-3-319-63046-5_29.

[22] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang, "Compositional shape analysis by means of bi-abduction," *Journal of the ACM*, vol. 58, no. 6, pp. 1–66, 2011. DOI: 10.1145/2049697.2049700.

[23] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis, "Bi-abductive resource invariant synthesis," in *APLAS 2009: Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 5904, Springer, 2009, pp. 259–274. DOI: 10.1007/978-3-642-10672-9_19.

[24] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang, "Local action and abstract separation logic," in *LICS 2007: Logic in Computer Science*, IEEE, 2007, pp. 366–378. DOI: 10.1109/LICS.2007.30.

[25] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat, "Verifying array manipulating programs by tiling," in *SAS 2017: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 10422, Springer, 2017, pp. 428–449. DOI: 10.1007/978-3-319-66706-5_21.

[26] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu, "Learning assumptions for compositional verification," in *TACAS 2003: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 2619, Springer, 2003, pp. 331–346. DOI: 10.1007/3-540-36577-X_24.

[27] David C. Cooper, "Theorem proving in arithmetic without multiplication," *Machine Intelligence*, vol. 7, pp. 91–99, 1972.

[28] Patrick Cousot and Radhia Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL 1977: Principles of Programming Languages*, ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[29] Patrick Cousot, Radhia Cousot, and Francesco Logozzo, "A parametric segmentation functor for fully automatic and scalable array content analysis," in *POPL 2011: Principles of Programming Langauges*, ACM, 2011, pp. 105–118. DOI: 10.1145/1925844.1926399.

[30] Patrick Cousot and Nicolas Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL 1978: Principles of Programming Languages*, ACM, 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[31] George Bernard Dantzig, *Linear Programming and Extensions: A Report Prepared for US Air Force Project Rand*. Rand Corporation, 1963.

[32] Thibault Dardinier, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller, "Sound automation of magic wands (extended version)," ETH Zurich, Technical Report, 2022. DOI: 10.48550/arXiv.2205.11325.

[33] David Detlefs, Greg Nelson, and James B. Saxe, "Simplify: A theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, pp. 365–473, 2005. DOI: 10.1145/1066100.1066102.

[34] Edsger W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972. DOI: 10.1145/355604.361591.

[35] Işil Dillig, Thomas Dillig, and Alex Aiken, "Fluid updates: Beyond strong vs. weak updates," in *ESOP 2010: Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 246–266. DOI: 10.1007/978-3-642-11957-6_14.

[36] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang, "A local shape analysis based on separation logic," in *TACAS 2006: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 287–302. DOI: 10.1007/11691372_19.

[37] Dino Distefano and Matthew J. Parkinson, "jStar: Towards practical verification for Java," in *OOPSLA 2008: Object-Oriented Programming, Sytems, Languages, and Applications*, ACM, 2008, pp. 213–226. DOI: 10.1145/1449764.1449782.

[38] Jérôme Dohrau, Alexender J. Summers, Caterina Urban, Severin Münger, and Peter Müller, "Permission inference for array programs," in *CAV 2018: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 10982, Springer, 2018, pp. 55–74. DOI: 10.1007/978-3-319-96142-2_7.

[39] Marco Eilers and Peter Müller, "Nagini: A static verifier for python," in *CAV 2018: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 10981, Springer, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33.

[40] Manuel Fähndrich and Francesco Logozzo, "Static contract checking with abstract interpretation," in *FoVeOOS 2010: Formal Verification of Object-Oriented Software*, ser. Lecture Notes in Computer Science, vol. 6528, Springer, 2010, pp. 10–30. DOI: 10.1007/978-3-642-18070-5_2.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its use in optimization," in *International Symposium on Programming*, ser. Lecture Notes in Computer Science, vol. 167, Springer, 1984, pp. 125–132. DOI: 10.1007/3-540-12925-1_33.

[42] Pietro Ferrara and Peter Müller, "Automatic inference of access permissions," in *VMCAI 2012: Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, vol. 7148, Springer, 2012, pp. 202–218. DOI: 10.1007/978-3-642-27940-9_14.

[43] Pietro Ferrara, Peter Müller, and Milos Novacek, "Automatic inference of heap properties exploiting value domains," in *VMCAI 2015: Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, vol. 8931, Springer, 2015, pp. 393–411. DOI: 10.1007/978-3-662-46081-8_22.

[44] Cormac Flanagan and K. Rustan M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, vol. 2021, Springer, 2001, pp. 500–517. DOI: 10.1007/3-540-45251-6_29.

[45] Robert W. Floyd, "Assigning meaning to programs," in *American Mathematical Society Symposia on Applied Mathematics*, vol. 19, 1967, pp. 19–32.

[46] Pranav Garg, "Learning universally quantified invariants of linear data structures," in *CAV 2013: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 813–829. DOI: 10.1007/978-3-642-39799-8_57.

[47] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider, "ICE: A robust framework for learning invariants," in *CAV 2018: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 69–87. DOI: 10.1007/978-3-319-08867-9_5.

[48] Tobias Gedell and Reiner Hähnle, "Automating verification of loops by parallelization," in *LPAR 2006: Logic for Programming, Artificial Intelligence, and Reasoning*, ser. Lecture Notes in Computer Science, vol. 4246, Springer, 2006, pp. 332–346. DOI: 10.1007/11916277_23.

[49] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv, "A framework for numeric analysis of array operations," in *POPL 2005: Principles of Programming Languages*, ACM, 2005, pp. 338–350. DOI: 10.1145/1040305.1040333.

[50] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori, "Bottom-up shape analysis," in *SAS 2009: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 5673, Springer, 2009, pp. 188–204. DOI: 10.1007/978-3-642-03237-0_14.

[51] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari, "Lifting abstract interpreters to quantified logical domains," in *POPL 2008: Principles of Programming Langauges*, ACM, 2008, pp. 235–246. DOI: 10.1145/1328897.1328468.

[52] Bolei Guo, Neil Vachharajani, and David I. August, "Shape analysis with inductive recursion synthesis," in *PLDI 2007: Programming Language Design and Implementation*, ACM, 2007, pp. 256–265. DOI: 10.1145/1250734.1250764.

[53] Ashutosh Gupta and Andry Rybalchenko, "InvGen: An efficient invariant generator," in *CAV 2009: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 634–640. DOI: 10.1007/978-3-642-02658-4_48.

[54] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas, "The SeaHorn verification framework," in *CAV 2015: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 9206, Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[55] Reiner Hähnle, Nathan Wasser, and Richard Bubel, "Array abstraction with symbolic pivots," in *Theory and Practice of Formal Methods*, ser. Lecture Notes in Computer Science, vol. 9660, Springer, 2016, pp. 104–121. DOI: 10.1007/978-3-319-30734-3_9.

[56] Nicolas Halbwachs and Mathias Péron, "Discovering properties about arrays in simple programs," in *PLDI 2008: Programming Language Design and Implementation*, ACM, 2008, pp. 339–348. DOI: 10.1145/1379022.1375623.

[57] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259.

[58] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar, "Fully automated shape analysis based on forest automata," in *CAV 2013: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 740–755. DOI: 10.1007/978-3-642-39799-8_52.

[59] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *NFM 2011: NASA Formal Methods*, ser. Lecture Notes in Computer Science, vol. 6617, Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[60] Bertrand Jeannet and Antoine Miné, "Apron: A library of numerical abstract domains for static analysis," in *CAV 2009: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[61] Ranjit Jhala and Kenneth L. McMillan, "Array abstractions in proofs," in *CAV 2007: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 193–206. DOI: 10.1007/978-3-540-73368-3_23.

[62] Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August, "A collaborative dependence analysis framework," in *CGO 2017: Code Generation and Optimization*, IEEE, 2017, pp. 148–159. DOI: 10.1109/CGO.2017.7863736.

[63] Ioannis T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM 2006: Formal Methods*, ser. Lecture Notes in Computer Science, vol. 4085, Springer, 2006, pp. 268–283. DOI: 10.1007/11813040_19.

[64] Daisuke Kimura and Makoto Tatsuta, "Decision procedure for entailment of symbolic heaps with arrays," in *APLAS 2017: Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 10695, Springer, 2017, pp. 169–189. DOI: 10.1007/978-3-319-71237-6_9.

[65] Laura Kovács and Andrei Voronkov, "Finding loop invariants for programs over arrays using a theorem prover," in *FASE 2009: Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, vol. 5503, Springer, 2009, pp. 470–485. DOI: 10.1007/978-3-642-00593-0_33.

[66] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin, "Shape analysis via second-order bi-abductino," in *CAV 2014: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 52–68. DOI: 10.1007/978-3-319-08867-9_4.

[67] Ton Chanh and Le, "SLING: Using dynamic analysis to infer program invariants in separation logic," in *PLDI 2019: Programming Language Design and Implementation*, ACM, 2019, pp. 788–801. DOI: 10.1145/3314221.3314634.

[68] Oukseh Lee, Hongseok Yang, and Rasmus Petersen, "Program analysis for overlaid data structures," in *CAV 2011: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 591–608. DOI: 10.1007/978-3-642-22110-1_48.

[69] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi, "Automatic verification of pointer programs using grammar-based shape analysis," in *ESOP 2005: European Symposium on Programming*, ser. Lecture Notes in Computer Science, vol. 3444, Springer, 2005, pp. 124–140. DOI: 10.1007/978-3-540-31987-0_10.

[70] K. Rustan M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR 201: Logic for Prigramming Artificial Intelligence and Reasoning*, ser. Lecture Notes in Computer Science, vol. 6355, Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[71] K. Rustan M. Leino and Peter Müller, "A basis for verifying multi-threaded programs," in *ESOP 2009: Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 5502, Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27.

[72] K. Rustan M. Leino, Peter Müller, and Jan Smans, "Deadlock-free channels and locks," in *ESOP 2010: Programming Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 407–426. DOI: 10.1007/978-3-642-11957-6_22.

[73] Sorin Lerner, David Grove, and Craig Chambers, "Composing dataflow analyses and transformations," in *POPL 2002: Principles of Programming Languages*, ACM, 2002, pp. 270–282. DOI: 10.1145/503272.503298.

[74] Jangchao Liu and Xavier Rival, "An array content static analysis based on non-contigous partitions," *Computer Languages, Systems & Structures*, vol. 47, pp. 104–129, 2017. DOI: 10.1016/j.cl.2016.01.005.

[75] Roman Manevich, Josh Berdine, Byron Cook, G Ramalingam, and Mooly Sagiv, "Shape analysis by graph decomposition," in *TACAS 2007: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 3–18. DOI: 10.1007/978-3-540-71209-1_3.

[76] John McCarthy and Patrick J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence*, vol. 4, Edinburgh University Press, 1969, pp. 463–502.

[77] Antoine Miné, "Inferring sufficient conditions with backward polyhedral underapproximations," *Electronical Notes in Theoretical Computer Science*, vol. 287, pp. 89–100, 2012. DOI: 10.1016/j.entcs.2012.09.009.

[78] David Monniaux and Laure Gonnord, "Cell morphing: From array programs to array-free horn clauses," in *SAS 2016: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 9837, Springer, 2016, pp. 361–382. DOI: 10.1007/978-3-662-53413-7_18.

[79] Leonardo de Moura and Nikolaj Bjørner, "Z3: An efficient SMT solver," in *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[80] Yannick Moy and Claude Marché, "Modular inference of subprogram contracts for safety checking," *Journal of Symbolic Computation*, vol. 45, pp. 1184–1211, 11 2010. DOI: 10.1016/j.jsc.2010.06.004.

[81] Jan Tobias Mühlberg, David H. White, Mike Dodds, Gerald Lüttgen, and Frank Piessens, "Learning assertions to verify linked-list programs," in *SEFM 2015: Software Engineering and Formal Methods*, ser. Lecture Notes in Computer Science, vol. 9276, Springer, 2015, pp. 37–52. DOI: 10.1007/978-3-319-22969-0_3.

[82] Peter Müller, "Modular specification and verification of object-oriented programs," ser. Lecture Notes in Computer Science 2262, Springer, 2002. DOI: 10.1007/3-540-45651-1.

[83] Peter Müller, Malte Schwerhoff, and Alexander J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *VMCAI 2016: Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, vol. 9583, Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[84] Daniel Neider, Parthasarathy Madhusudan, Shambwaditya Saha, and Daejun Park, "A learning-based approach to synthesizing invariants for incomplete verification engines," *Journal of Automated Reasoning*, vol. 64, no. 7, pp. 1523–1552, 2020. DOI: 10.1007/s10817-020-09570-z.

[85] Peter W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1 – 3, pp. 271–307, 2007. DOI: 10.1016/j.tcs.2006.12.035.

[86] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang, "Local reasoning about programs that alter data structures," in *CSL 2001: Computer Science Logic*, ser. Lecture Notes in Computer Science, vol. 2142, Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

[87] Matthew J. Parkinson and Gavin M. Bierman, "Separation logic and abstraction," in *POPL 2005: Principles of Programming Languages*, ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326.

[88] Matthew J. Parkinson and Alexander J. Summers, "The relationship between separation logic and implicit dynamic frames," *Logical Methods in Computer Science*, vol. 8, no. 3, 2012. DOI: 10.2168/LMCS-8(3:1)2012.

[89] Ruzica Piskac and Thomas Wies, "GRASShopper - complete heap verification with mixed specifications," in *TACAS 2014: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 124–139. DOI: 10.1007/978-3-642-54862-8_9.

[90] Shengchao Qin, Guanhua He, Wei-Ngan Chin, Florin Craciun, Mengda He, and Zhong Ming, "Automated specification inference in a combined domain via user-defined predicates," *Science of Computer Programming*, vol. 148, pp. 189–212, 2017. DOI: 10.1016/j.scico.2017.05.007.

[91] Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan, "Natural proofs for structure, data, and separation," in *PLDI 2013: Programming Language Design and Implementation*, ACM, 2013, pp. 231–242. DOI: 10.1145/2491956.2462169.

[92] John C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS 2002: Logic in Computer Science*, IEEE, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[93] Reuben N. S. Rowe and James Brotherston, "Automatic cyclic termination proofs for recursive procedures in separation logic," in *CPP 2017: Certified Programs and Proofs*, ACM, 2017, pp. 53–65. DOI: 10.1145/3018610.3018623.

[94] Malte Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution," PhD Thesis, ETH Zurich, 2016.

[95] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori, "Verification as learning geometric concepts," in *SAS 2013: Static Analysis*, ser. Lecture Notes in Computer Science, vol. 7935, 2013, pp. 388–411. DOI: 10.1007/978-3-642-38856-9_21.

[96] Rahul Sharma, Aditya V. Nori, and Alex Aiken, "Interpolants as classifiers," in *CAV 2012: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 71–87. DOI: 10.1007/978-3-642-31424-7_11.

[97] Jan Smans, Bart Jacobs, and Frank Piessens, "VeriCool: An automatic verifier fro a concurrent object-oriented language," in *FMOODS 2008: Formal Methods for Open Object-Based Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 5051, Springer, 2008, pp. 220–239. DOI: 10.1007/978-3-540-68863-1_14.

[98] ——, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *ECOOP 2009: Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 148–172. DOI: 10.1007/978-3-642-03013-0_8.

[99] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík, "Sketching concurrent data structures," in *PLDI 2008: Programming Language Design and Implementation*, ACM, 2008, pp. 136–148. DOI: 10.1145/1375581.1375599.

[100] Alexander J. Summers and Sophia Drossopoulou, "A formal semantics for isorecursive and equirecursive state abstractions," in *ECOOP 2013: Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 7920, Springer, 2013, pp. 129–153. DOI: 10.1007/978-3-642-39038-8_6.

[101] Alexander J. Summers and Peter Müller, "Automating deductive verification for weak-memory programs (extended version)," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 6, pp. 709–728, 2020. DOI: 10.1007/s10009-020-00559-y.

[102] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans, "Annotation inference for separation logic based verifiers," in *FMOODS 2001, FORTE 2011: Formal Techniques for Distributed Systems*, ser. Lecture Notes in Computer Science, vol. 6722, Springer, 2011, pp. 319–333. DOI: 10.1007/978-3-642-21461-5_21.

[103] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner, "RELAY: Static race detection on millions of lines of code," in *SIGSOFT 2007: Symposium on the Foundations of Software Engineering*, ACM, 2007, pp. 205–214. DOI: 10.1145/1287624.1287654.

[104] Mark D. Weiser, "Program slicing," *Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984. DOI: 10.1109/TSE.1984.5010248.

[105] Felix A. Wolf, Linard Arquint, Martin Clochard, Ortwijn Wytse, João C. Pereira, and Peter Müller, "Gobra: Modular specification and verification of Go programs," in *CAV 2021: Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 12759, Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17.

[106] Felix A. Wolf, Malte Schwerhoff, and Peter Müller, "Concise outlines for a complex logic: A proof outline chekcer for TaDA," in *FM 2021: Formal Methods*, ser. Lecture Notes in Computer Science, vol. 13047, Springer, 2021, pp. 407–426. DOI: 10.1007/978-3-030-90870-6_22.

[107] Hongseok Yang and Peter W. O'Hearn, "A semantic basic for local reasoning," in *FOSSACS 2002: Foundations of Software Science and Computation Structures*, ser. Lecture Notes in Computer Science, vol. 2303, Springer, 2002. DOI: 10.1007/3-540-45931-6_28.

Appendix

A.1 Lemmas for Isorecursive Predicates

Methods proving the append lemma and the concatenation lemma from Section 4.5.5 are shown in Listing A.1 and Listing A.2, respectively.

```
1  method append_lemma(x: Node, y: Ref, z: Ref)
2    requires seg(x, y)
3    requires y ≠ z ⇒ rec′(y) ∗ (b′ ⇒ s(y) = z)
4    ensures seg(x, z)
5  {
6    if (y ≠ z) {
7      if (x = y) {
8        // base case
9        if (b′) {
10         fold seg(s(y), z)
11       }
12       fold seg(x, z)
13     } else {
14       // step case
15       unfold seg(x, y)
16       if (b) {
17         append_lemma(s(x), y, z)
18       }
19       fold seg(x, z)
20     }
21   }
22 }
```

Listing A.1. A method proving the append lemma for a generic list-like predicate segment $\mathsf{seg}(x, y) \triangleq x \neq y \Rightarrow \mathsf{rec}'(x) * (b \Rightarrow \mathsf{seg}(s(x), y))$. We use $b' :\equiv b[y\backslash x]$ to denote the condition b adapted to its context.

```
1  method concat_lemma(x: Node, y: Ref, z: Ref)
2    requires seg(x, y) ∗ seg(y, z)
3    ensures seg(x, z)
4  {
5    if (x = y) {
6      // base case
7    } else {
8      // step case
9      unfold seg(x, y)
10     if (b) {
11       concat_lemma(s(x), y, z)
12     }
13     fold seg(x, z)
14   }
15 }
```

Listing A.2. A method proving the concatenation lemma for a generic list-like predicate segment $\mathsf{seg}(\mathsf{x}, \mathsf{y}) \triangleq \mathsf{x} \neq \mathsf{y} \Rightarrow \mathsf{rec}'(\mathsf{x}) * (b \Rightarrow \mathsf{seg}(s(\mathsf{x}), \mathsf{y}))$.

Curriculum Vitae

Personal Information

Name:	Jérôme Dohrau
Date of Birth:	March 15, 1989
Place of Birth:	Basel, Switzerland
Nationality:	Swiss
Contact:	jerome@dohrau.ch

Education

2016–2022	**Ph.D. in Computer Science**, ETH Zurich Advisor: Prof. Dr. Peter Müller Thesis: “Automatic Inference of Permission Specifications”.
2013–2016	**Master in Computer Science**, ETH Zurich Thesis: “Edge Flips in Combinatorial Triangulations”.
2010–2014	**Bachelor in Computer Science**, ETH Zurich Thesis: “Online Makespan Scheduling with Sublinear Advice”.

Employment

2022–today	**Software Engineer**, Google Switzerland.
2016–2021	**Research Assistant**, ETH Zurich.