Arshavir Ter-Gabrielyan

Compositional Verification of Rich Program Properties in Separation Logic

Diss. ETH 27994 | 2021

DISS. ETH № 27994

COMPOSITIONAL VERIFICATION OF RICH PROGRAM PROPERTIES IN SEPARATION LOGIC

A thesis submitted to attain the degree of
Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by

ARSHAVIR TER-GABRIELYAN
Master of Science in Applied Mathematics and Physics,
Moscow Institute of Physics and Technology

born April 24, 1991
citizen of Russia

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Martin Vechev, co-examiner
Prof. Dr. Thomas Wies, co-examiner

2021

Arshavir Ter-Gabrielyan

Compositional Verification of Rich Program Properties in Separation Logic

CONTACTS

☞ TerGabrielyan@gmail.com

Посвящается Светлане Туторской
1938 – 2005

ABSTRACT

Recent advances in deductive program verification correlate with the evolution of logics for modular reasoning about complex programs. Verification techniques built upon these logics require automation to help to verify practically essential *rich program properties* that summarize data structures via quantification or some form of abstraction. However, many such properties are *higher-order* (e. g. data structure comprehensions like sequence-fold), precluding automation. Furthermore, some rich properties (e. g. reachability between dynamically interlinked objects) are *non-compositional*; that is, one cannot compose the property of a data structure based solely on the properties of its disjoint sub-structures. However, compositionality is a prerequisite of automated modular reasoning (due to the problem commonly known as *framing*). If compositionality holds, the programmer can independently reason about each method in an application without considering the implementation of other methods, e. g. library code.

The goal of this thesis is to develop *compositional* — i. e. automated and modular — techniques for verifying rich program properties. We build on top of separation logic, a prominent program logic that enables modular reasoning about heap-transforming, concurrently executed programs. The specification language of separation logic expresses memory safety properties, but complementary techniques for handling rich properties are underdeveloped, especially in an automated setting. The two classes of rich properties that we consider are *data structure comprehensions* and *heap reachability properties*. An additional goal of the thesis is to develop techniques for *automated verification debugging*, aiding the programmer in authoring formal specifications and verified programs.

Our first contribution is a technique for reasoning about the class of (higher-order, compositional) comprehensive properties. These properties summarize data structures containing potentially unbounded (and statically unknown) object sets via a finite number of values. Our encoding reduces comprehensions to first-order logic by modeling them as uninterpreted functions and leveraging native features of separation logic, most notably, the *iterated separating conjunction* connective. We develop a first-order axiomatization of these functions, automating framing and the lemmas required for SMT-based verification of characteristic benchmark programs. Our technique supports comprehensions over data structures regardless of how their objects are accessed and ordered, e. g. general heap graphs in addition to index-based array structures.

Our second contribution is a technique for reasoning about the class of (non-compositional) heap reachability properties. These properties express the existence or the absence of directed paths connecting dynamically interlinked heap objects. For each method, we specify reachability only locally within the fragment of the heap on which the method operates. We identify *relative convexity*, a novel relation between the heap fragments of

a callee and its client, which enables (first-order) reachability framing, i. e. propagating reachability properties from the heap fragment of a callee to the larger heap fragment of its client, enabling precise, modular reasoning. Our technique supports practically important data structures, namely, acyclic graphs with a bounded outdegree and (potentially cyclic) graphs with at most one path (modulo cycles) between each pair of nodes.

Our third contribution is a technique for generating counterexample heap reachability models for verification debugging. We propose a general procedure for extracting heap models from partial SMT models. We then extend this procedure to extract (state-dependent) local heap reachability relations. To refer to relevant program states in a heap model, we employ lightweight instrumentation of the source program. Our technique is agnostic to the verifier implementation, supporting symbolic execution and verification condition generation with only minor adaptations. Our algorithm extracts first-class heap reachability relations needed for specifying complex heap configurations, e. g. acyclic or disjoint structures. We automatically visualize the output heap reachability models and demonstrate the practicality of our technique in two scenarios: debugging a failed verification and inspecting a verified heap-transforming program.

ZUSAMMENFASSUNG

Die jüngsten Fortschritte in der deduktiven Programmverifikation korrelieren mit der Evolution von Logiken für modulare Argumentation über komplexe Programme. Die auf diesen Logiken basierenden Verifikationsmethoden benötigen Automatisierung, um zu helfen, praktisch relevante *reiche Programmeigenschaften* zu verifizieren, die Datenstrukturen durch Quantifizierung oder gewisse Abstraktionen zusammenfassen. Allerdings sind viele solche Eigenschaften *höherer Stufe* (z.B. Datenstruktur-Comprehensions wie Sequence-Fold), was Automatisierung erschwert. Des Weiteren sind manche solche reiche Eigenschaften (z.B. Erreichbarkeit zwischen dynamisch miteinander verknüpften Objekten) *nicht-kompositionell*; das heißt, man kann die Eigenschaft der Datenstruktur nicht lediglich auf Basis der Eigenschaften ihrer disjunkten Substrukturen bilden. Jedoch ist Kompositionalität eine Voraussetzung der automatisierten modularen Argumentation (aufgrund des Problems allgemein bekannt als *Framing*). Gilt Kompositionalität, kann der/die Programmierer/-in unabhängig über jede Methode in der Applikation schlussfolgern, ohne die Implementierung anderer Methoden zu berücksichtigen, z.B. Library-Code.

Das Ziel dieser Dissertation ist, neue Techniken für Verifikation reicher Programm-Eigenschaften zu entwickeln, die *kompositionell* sind, das heißt automatisiert und modular. Wir bauen auf der Separationslogik auf, einer bekannten Programmlogik, die modulare Argumentation über Heap-transformierende, gleichzeitig ausgeführte Programme ermöglicht. Die Spezifikationssprache der Separationslogik drückt Speichersicherheitseigenschaften aus, doch ergänzende Techniken für Umgang mit reichen Eigenschaften sind unterentwickelt, insbesondere im Kontext von Automatisierung. Die zwei Klassen der reichen Eigenschaften, die wir betrachten, sind *Datenstruktur-Comprehensions* und *Heap-Erreichbarkeitseigenschaften*. Ein zusätzliches Ziel der Dissertation ist, Techniken für *automatisiertes Verifikations-Debugging* zu entwickeln, die Programmierer beim Verfassen von formalen Spezifikationen und verifizierten Programmen unterstützen.

Unser erster Beitrag ist eine Technik für Argumentation über die Klasse der (kompositionellen) zusammenfassenden Eigenschaften höherer Stufe. Diese Eigenschaften fassen Datenstrukturen, die potentiell unbegrenzte (und statisch unbekannte) Objektmengen enthalten, durch eine begrenzte Anzahl von Werten zusammen. Unsere Codierung reduziert Comprehensions auf die Prädikatenlogik erster Stufe, indem sie diese als uninterpretierte Funktionen modelliert und native Funktionen der Separationslogik wirksam einsetzt, insbesondere die Verknüpfung der *iterativen trennenden Konjunktion*. Wir entwickeln eine Axiomatisierung erster Stufe für diese Funktionen und automatisieren somit Framing und Lemmata, die für die SMT-basierte Verifikation der typischen Benchmark-Programmen erforderlich sind. Unsere Technik unterstützt Compre-

hensions von Datenstrukturen unabhängig davon, wie ihre Objekte geordnet sind und abgerufen werden, z.B. allgemeine Heap-Graphen zusätzlich zu indexbasierten Array-Strukturen.

Unser zweiter Beitrag ist eine Technik für Argumentation über die Klasse der (nicht-kompositionellen) Heap-Erreichbarkeitseigenschaften. Diese Eigenschaften drücken An- oder Abwesenheit der gerichteten Wegen aus, die dynamisch miteinander verknüpfte Heap-Objekte verbinden. Für jede Methode spezifizieren wir Erreichbarkeit nur lokal innerhalb des Heap-Fragments, auf dem die Methode operiert. Wir identifizieren das Konzept der *relativen Konvexität*, eine neuartige Beziehung zwischen den Heap-Fragmenten der aufgerufenen Methode und ihrem Client, welche Erreichbarkeits-Framing (erster Stufe) ermöglicht, das heißt, Erreichbarkeitseigenschaften vom Heap-Fragment der aufgerufenen Methode zum größeren Heap-Fragment ihres Clients zu propagieren, was präzise, modulare Argumentation ermöglicht. Unsere Technik unterstützt besonders wichtige Datenstrukturen, und zwar azyklische Graphen mit einem begrenzten Ausgangsgrad und (potenziell zyklische) Graphen mit höchstens einem Weg (modulo Zyklen) zwischen jeder Knotenpaar.

Unser dritter Beitrag ist eine Technik für Erzeugung von Heap-Erreichbarkeitsmodellen als Gegenbeispiele für Verifikations-Debugging. Wir schlagen ein allgemeines Verfahren für das Extrahieren der Heap-Modelle aus dem SMT-Teilmodell vor. Wir erweitern anschließend das Verfahren, um (zustandsabhängige) lokale Heap-Erreichbarkeitsbeziehungen zu extrahieren. Um auf relevante Programmzustände in einem Heap-Modell Bezug zu nehmen, verwenden wir eine einfache Instrumentierung des Quellprogramms. Unsere Technik ist nicht abhängig von der Implementierung des Verifiers und unterstützt Verifikation sowohl durch symbolische Ausführung als auch durch Überprüfung von Verifikationsbedingungen mit nur geringen Anpassungen. Unser Algorithmus extrahiert Heap-Erreichbarkeitsbeziehungen, die für die Spezifizierung von komplexen Heap-Konfigurationen, z.B. azyklischen oder disjunkten Strukturen, benötigt werden. Wir visualisieren automatisch die Ausgabemodelle der Heap-Erreichbarkeit und zeigen die Praktikabilität unserer Technik in zwei Szenarien: Debuggen der fehlgeschlagenen Verifikation und Inspektion des verifizierten Heap-transformierenden Programms.

ACKNOWLEDGMENTS

I am deeply grateful to the many people who helped me in my doctoral studies at ETH Zürich. Your support and advice enabled me to overcome all the difficulties along the way.

I thank my advisor, Prof. Peter Müller, who offered me the opportunity to work in his group. Thank you for the insightful research meetings. Thank you also for granting me the freedom to write this thesis in my way. It has been an honor to work with you.

I thank my co-advisor, Prof. Martin Vechev, and co-examiner, Prof. Thomas Wies, for taking the time to review this manuscript.

I am grateful to all the members and alumni of the Department of Computer Science who have played a vital role in the early days of my doctoral studies.

Huge thanks to Dr. Lucas Brutschy for selecting my application and recommending it for consideration. You have been an excellent fellow group member and friend. Thank you for introducing me to ETH, the city of Zurich, and climbing.

I thank Prof. Maria Christakis for a warm reception on my first visit to Zurich.

I thank Milos Novacek for helping me when I relocated to Switzerland. We have not worked together, but I am grateful for your contributions to the Viper system. Thank you also for being an early motorcycling mentor for me.

I thank Robert Khasanov for introducing me to the city of Zurich. You were the only friend I knew from before. Thank you for your support and friendship.

I thank Dr. Uri Juhasz for the many deep technical conversations about logic and formal verification. Those discussions have inspired my interest in transitive closures. Thank you also for being an excellent colleague and spreading your love for the magnificent Swiss outdoors.

I thank Marlies Weissert for her invaluable support and her exceptional organizational skills. Thank you for always going the extra mile.

I am grateful to my direct collaborators for their trust and help shape tough research topics into viable techniques.

I thank Prof. Alexander J. Summers for mentoring me in the early days of my doctoral studies and introducing me to formal verification. I will never forget some of those colorful whiteboards. Thank you for all the endless technical discussions, for encouraging me when I was stuck, and for introducing me to crayon-flavored beer.

I thank Alexandra Bugariu for trusting my intuition and investing her time to bring our ideas to life. Thank you for your thoughtful style of work and friendship.

I am grateful to all the members and alumni of the Programming Methodology group who have not only been excellent colleagues but have also become close friends of mine.

I thank Marco Eilers, with whom I shared an office for many years. Our views are known to be quite the opposite on so many things. This contrast has impacted the way I perceived the purpose of our work. Thank you for being open to discussions, be those about technical aspects, Ph.D.-related realizations or frustrations, or conceptual talks about life over a glass of beer. Thank you also for all the proof-readings.

I thank Jérôme Dohrau. While we never got to collaborate on research directly, I highly appreciate our friendship. Thank you for all your help throughout the years, be it in the context of teaching, writing papers, life in Switzerland, or motorcycles.

I thank Federico Poli for his professional and efficient approach to research. Thank you for all your help and advice throughout these years. Thank you also for being an excellent friend outside the office.

I thank Vytautas Astrauskas for his uncompromising focus on excellence. We worked side-by-side throughout some difficult years, and your ever-lasting sense of humor made this experience even more enjoyable.

I thank Linard Arquint, an excellent collaborator who has shared my enthusiasm for building things. Thank you for breathing life into our projects. Thank you also for proof-reading a draft of this thesis.

I thank Gaurav Parthasarathy for being an excellent colleague who has brought invaluable expertise to our group. Thank you for being an exemplar contributor and a genuinely kind person. Thank you also for proofreading an early draft of this thesis.

I thank Felix Wolf for his contributions to our group. Thank you for the brainstorming sessions. I admire your ability to dive into a complex problem and instantly come up with the right questions.

I thank Dr. Caterina Urban for her mentorship and dedication. Thank you for keeping your door open to me, for the encouraging discussions about life in academia, and for constantly radiating positive energy.

I thank Dr. Dimitar Asenov for exemplifying a disciplined and professional approach to research. Thank you also for your friendship and help throughout the years.

I thank Dr. Malte Schwerhoff for relentlessly contributing to the Programming Methodology group (and the entire department). Thank you for keeping your door always open to me throughout these years.

I thank Dr. Valentin Wüstholz for his research advice and friendship.

I thank Sandra Schneider for supporting me (and the group) during the trying years of the Covid pandemic. Thank you for all the help and advice, especially in the last phase of my doctoral studies.

I am enthusiastic about the new generation of members of the Programming Methodology group who are currently taking up the baton. Best of luck to you, João Carlos Mendes Pereira, Thibault Dardinier, and Jonás Fiala!

I thank my fellow (unfortunately, former) colleagues: Prof. Werner Dietl for his career advice; Dr. Martin Clochard for always being open to brainstorming obscure formal verification ideas; Dr. Wytse Oortwijn for his contributions to our projects and spreading his passion for board games; Prof. Christoph Matheja for exemplifying excellence in research; Dr. Hermann Lehner for his contributions to teaching at ETH; Fábio Pakk Selmi-Dei for supporting our group's engineering aspirations and sense of humor; Dr. Daniele Spampinato for the conceptual discussions and his contributions to the scientific staff association, VMI; Dr. Pavol Bielik for (informal, yet insightful) research discussions; Prof. Gagandeep Singh for the interesting hallway debates; Dr. Aristeidis Mastoras for the conversations over a beer; Prof. Pietro Ferrara for the research discussions.

I thank Bernadette Gianesi and the rest of the Department of Computer Science for the constant support and attention to detail.

I am grateful to the Swiss National Science Foundation for funding Project 200021-156980, supporting the bigger part of my doctoral studies.

I am grateful to all the students and interns whom I had the privilege to advise: Alessio Aurecchia, Tierry Hörmann, Ruben Kälin, Valentin Racine, Lukas Schär, Gishor Sivanrupan, Cédric Stoll, and Benjamin Weber.

I am grateful for the support of my other friends, who did not forget who I am, especially Vasily Vasilyev — looking forward to reading *your* thesis one day.

I am grateful to my family for their unconditional support throughout this journey. Distinctly, I thank my father, Gevorg, and my mother, Elena. I am most grateful to my dear fiancée, Sviatlana-Maryia Zdobnikava, for her constant support and love.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Formal verification of realistic heap-transforming programs requires techniques that are *modular*. One common target for modular verification is library code designed to be reused by multiple clients and applications. However, library developers typically do not know in advance what applications might rely on their code — let alone the implementation details of those applications — but their code should nonetheless be correct and secure in all cases. Modular verification solves this problem by proving that methods comply with their *modular specifications* regardless of the invocation context, as long as this context satisfies the formally specified requirements.

The *specification language* determines the extent to which one can verify safety- and security-critical software, e. g. concurrent data structure libraries. Therefore, a key aspect of the specification language is its *expressiveness*, i. e. the facts and logic that the programmers can refer to in specifications. *Separation logic* [22, 26] is a prominent specification language for specifying heap configurations, e. g. disjointness or object sharing; typically, these specifications express *crash freedom* of heap-transforming programs, e. g. the absence of null pointer dereferences and stack safety. Separation logic supports concurrent program specifications and is designed for modular verification as the programmer can specify and verify even concurrent methods independently from one another.

Practical verification techniques should provide *automation*. Programmers rely on automated modular verification tools that support *code-level specifications* [19, 49, 57]. Verification experts capable of *authoring logical proofs* also benefit from automation due to the potentially prohibitive overhead of writing formal proofs, compared to devising informal correctness arguments and testing. In particular, *refactoring* a manually verified program may require re-proving it from scratch. While proof assistants, e. g. Coq [29] and Isabelle/HOL [25] mechanize proof authoring, they still expose the programmer to low-level logical details, shifting their focus away from the essence of programming, and are thus not commonly applicable.

Separation logic-based techniques are often automated by encoding proof obligations as first-order formulas and discharging them via SMT solver [61, 71, 91]. Once a method is verified not to crash, the next question is whether its computation always yields intended results. Programmers thus specify more sophisticated properties that the methods must also respect. For example, one can specify that `Array.sort(a)`, indeed, results in the array being sorted: $\forall i, j \bullet 0 \leq i < j < \texttt{len(a)} \Rightarrow \texttt{a}[i]\texttt{.val} \leq \texttt{a}[j]\texttt{.val}$, where `a` is an array of object references, `len(a)` is the length of the array, and `val` is an integer

field.[1] Sortedness is a first-order property that a prover can check automatically as it relates array values simply via universal quantification.

Compositional program properties are properties of disjoint heap structures that naturally support modular verification. In modular verification, a client can adapt to the effect of a method call via *framing* [24, 37], i. e. composing the properties ensured by the callee and the properties of the remainder heap fragment, called the frame. Verifiers can frame compositional properties *automatically* through first-order reasoning. For example, the concatenation of the two disjoint, ascendingly sorted arrays `a` and `b` is also ascendingly sorted *iff* the maximal value in `a` is less than or equal to the minimal value in `b`. Since *minimal* and *maximal* array values are also compositional properties, sortedness is compositional, and the verifier can check this property modularly.

Rich program properties summarize heap structures in an abstract way. For instance, array sortedness is such a property, as it summarizes the relations between adjacent array values across the entire array. Unlike sortedness, some rich properties are *non-compositional*; framing non-compositional properties requires higher-order reasoning, which impedes automation. *Heap reachability* is a notable non-compositional program property that defines the existence (or absence) of heap paths connecting certain objects. For example, specifying structures that are *acyclic*, e. g. *binary-decision diagrams* [3, 10] and *version control histories* [73], requires heap reachability. Framing reachability properties is generally undecidable due to their higher-order nature. Nevertheless, compositional verification of reachability is possible in a restricted setting [76].

1.1 RESEARCH GOALS AND SCOPE

The main goal of this thesis is to advance modular deductive verification by enabling automated, compositional reasoning about rich program properties. For this purpose, we developed principles, axiomatizations, and logical encodings and evaluated them by verifying benchmarks from literature.

This thesis is dedicated to *static verification techniques*, i. e. those that do not rely on *concrete executions* but analyze programs only symbolically. In some cases, it is possible to combine static and dynamic information to achieve better results [28, 86]. However, a trend in modern verification infrastructures is to rely upon an *intermediate verification language* (IVL), a simple, non-executable language designed for expressing a spectrum of verification problems [46, 67, 91]. Verification *frontends* encode formally specified source programs from common languages, e. g. Java, into the IVL [49, 57, 98, 107, 126]. Verification *backends* apply generic techniques, e. g. symbolic execution [92] or verification condition generation [46, 68], to verify the IVL encodings. This infrastructure design sepa-

1 This specification of `sort` is still incomplete and is satisfied even by an (incorrect) implementation that assigns all the array elements to e. g. zero; to ensure that `sort` only *swaps* the values, one could specify that the resulting array has the same number of `0`s, `1`s, `-1`s, etc. — i. e. `sort` only permutes the values.

rates two orthogonal concerns: Frontend developers address only language-specific verification challenges and naturally benefit from improvements of the backends; Backend developers target only one, simple language, focusing on language-agnostic verification techniques. Static techniques support this trend as they are applicable to IVLs.

The logical foundation of the techniques developed in this thesis is separation logic, a prominent framework for reasoning about effectful programs. Most existing techniques based on separation logic leverage its *abstract predicates* to specify both memory layouts and value properties of heap structures. To define potentially unbounded structures, e. g. lists and trees, separation logic supports *recursive* abstract predicates. However, recursive predicates cannot define more general structures, e. g. random-access arrays and graphs of shared objects. In this thesis, we consider an alternative technique called *iterated separating conjunction* (ISC) [24, 90] that specifies memory footprints of methods and data structures without restricting order of access and sharing.

ISC offers a set-theoretical view of the framing problem. In this view, rich properties of sets of objects can be expressed via *state-dependent functions*. Therefore, framing amounts to establishing the relationship between a function over a *disjoint union* of two object sets and the same function over the individual sets, i. e.:

$$f(A \uplus B) \;=\; f(A) \oplus f(B)$$

where $f(X)$ expresses some property of the set X of heap objects, $\uplus$ is the disjoint union, and $\oplus$ is an operator that combines the properties of the two subsets A and B into the property of the entire set $A \uplus B$. The operator $\oplus$ enables modular reasoning about the property f. The complexity of $\oplus$, e. g. whether it is a *first-order* formula, determines the automation potential for modularly reasoning about the property f.

1.1.1 Problem statement

Automated modular deductive verification of heap-transforming programs is a research area with numerous techniques developed in the past two decades. Supporting modular reasoning is essential for a deductive verification technique as non-modular techniques typically do not scale well to real software. However, many important aspects of software correctness and reliability, including rich program properties discussed above, are currently not available in an automated setting. Thus, we identified the following gaps in the state of the art that are in the scope of this thesis:

1.1.1.1 Modular specification of unbounded structures. Most separation logic-based techniques specify data structures via recursive abstract predicates. In automated reasoning, recursive predicates often impede automation because they require the programmer to manually augment the program with ghost operations that *fold* and *unfold* predicate definitions, navigating through different abstract representations of a data structure. In

specifications, recursive predicates can define only a limited class of heap structures and decompositions thereof; it is not possible to specify practically via recursive predicates essential structures e. g. arrays or graphs of shared objects. How could one modularly specify and automatically verify *arbitrary* heap-allocated structures?

The key to a solution, namely, the *iterated separating conjunction* connective, was proposed already in a classical work on separation logic [24]. Theoretically, ISC-based specifications allow the programmer to disentangle memory layouts and functional properties of arbitrary heap structures. However, since ISC is a form of unbounded quantification, automated reasoning with ISC is complicated. To our knowledge, most separation logic-based verifiers do not support this connective. Fortunately, there are notable exceptions: Viper [90] and, to some extent, GRASShopper [78]. These projects show that ISC is useful not only in pen-and-paper proofs but also in an automated setting.

What is missing for practically realizing the generalization of recursive predicates via ISC is a suitable extension of the specification language. Since ISC provide a set-theoretical view over unbounded heap structures, one typically cannot exhaustively specify the properties of each individual object as methods can operate on *unbounded* object sets. A natural way to express properties of unbounded sets is via universal quantification. However, first-order quantifiers summarize only Boolean functions but not more advanced algebraic properties; even an intuitively simple property, e. g. the count of marked graph nodes, let alone more advanced ones, e. g. scalar vector products, cannot be expressed via universal quantification. Existing approaches partially tackle this problem, but either in a non-modular or non-automated setting.

In this thesis, we address the challenge of modular specification of unbounded heap structures by proposing *set comprehensions*, a novel class of compositional program properties that complement ISC. Set comprehensions generalize the concept of first-order quantification by summarizing potentially unbounded (and statically unknown) object sets via commutative, associative operators, e. g. addition, multiplication, and minimization. We develop a lightweight encoding of set comprehensions into separation logic and an efficient first-order axiomatization for automating relevant lemmas about comprehensions. We evaluate the technique based on examples from literature, including the benchmarks from closely-related prior work.

1.1.1.2 Composability of heap reachability.

Another class of practically essential program properties is heap reachability. Recursive predicate-based specifications specified reachability indirectly by imposing a (partial) ordering of objects; e. g. a method maintaining a recursive list predicate can rely on the acyclicity of this list. Conversely, in our ISC-based view, memory layouts are specified in terms of potentially unordered object sets, requiring a direct approach to specifying reachability.

The general heap reachability relation cannot be simply added to the specification language because this relation carries inherently non-local information, i. e. it is non-compositional. To modularly verify reachability properties, one needs to reason about

entire heap paths, not just their origins and destinations, as heap paths are generally not limited to method footprints defined through ISC. However, this is practically not feasible due to the potentially unbounded length of general heap paths. This challenge is of theoretical interest as it occurs not only in program verification but also in the general theory of flow networks. Existing modular approaches establish composability of heap reachability only for restricted classes of heap structures. Compared to recursive predicates, these techniques achieve better automation (as they do not require ghost operations), but the spectrum of supported heap structures is effectively the same.

In this thesis, we address the challenge of establishing composability of reachability in general graphs by proposing a novel relation of *local heap reachability* and developing a less restrictive setting of *relatively convex footprints*. We develop a lightweight encoding of local reachability relations into separation logic and an efficient first-order axiomatization for automating relevant reachability lemmas. We evaluate the technique based on examples from literature, including the benchmarks from closely-related prior work.

*1.1.1.3 **Verification debugging.*** Even with automated tools, deductive verification is still challenging as errors can occur in the implementation of the program, or its specification, or both at the same time. Thus, developing verified software is typically an *iterative* process, involving multiple stages of refinement before the end goal is reached. Each intermediate stage in this process consists of a verification failure (which the programmer must interpret and diagnose), a solution design (e. g. a hypothesis that the implementation is buggy or that the specification is inadequate), and a refinement (i. e. actually modifying the code and re-running the verifier).

We focus on the problem of diagnosing verification failures based on counterexamples. In case of a verification failure, SMT solvers powering state-of-the-art verifiers provide counterexample models that assign values to variables and provide interpretations to functions. However, *interpreting* raw counterexample models is challenging. First, the programmer might not be familiar with the *internal aspects* of the encoding, which are fully exposed in a raw model, e. g. *permission masks* or *field-value functions* used for encoding access permissions. Thus, interpreting counterexample models requires information about the logical encoding. For this reason, existing SMT-based verification debuggers are tailored to only one backend technology.

Second, raw SMT models are *partial* and *imprecise* in presence of undecidable logics, e. g. because the solver could not instantiate some quantifiers or find complete interpretations for challenging functions. Therefore, it is challenging to extract information from partial models that is *relevant* — that which helps programmers to understand verification failure causes. Existing SMT-based verification debuggers typically avoid undecidable logics [76, 78] or rely on code execution for filtering out spurious models [63, 102]. The former approach fundamentally limits expressiveness of the supported logic. The latter approach is not applicable to non-executable, intermediate verification languages.

In this thesis, we address the above challenges by proposing a novel procedure for extracting *heap configurations* from counterexample SMT models. Our procedure is *agnostic* to specific backend implementations (supporting both symbolic execution and verification condition generation) and is *extensible*, i. e. one can augment the produced models with information from a separation logic extension. We demonstrate the latter aspect by extending our models with arbitrary heap reachability relations. We evaluate our technique based on two scenarios: debugging inadequate specifications and learning typical behaviours of a verified algorithm implementation.

1.1.2 Problem domain

The experimental platform of this thesis is based on Viper [91], a state-of-the-art infrastructure for automated reasoning in separation logic. At the core of the infrastructure is the Viper intermediate verification language. Formally specified programs written in a source language are translated into Viper through frontend verifiers, most notably, VerCors for concurrent Java [72] and C [84], Viper-RSL for C11 [106], Nagini for (statically typed) Python [98], Prusti for Rust [107], and Gobra for Go [126]. These translations are then statically verified through the verification backends. Next, we summarize the reasons why Viper is an interesting research platform used for demonstrating the techniques of this thesis:

- Viper's logic natively supports ISC-based specifications and state-dependent functions. We discussed the main advantages of ISC-based specifications above. In addition to method footprints, ISC can also specify function footprints. Thus, state-dependent functions can *directly* axiomatize stateful aspects of separation logic extensions without requiring modification of the verification backends.
- Viper supports both symbolic execution [92] and verification condition generation [68]. To our knowledge, Viper is the only deductive verification infrastructure that supports both of these backend technologies. The availability of two alternative backends is essential in practice: For undecidable logics, the backends are incomparable in terms of completeness and hence complementary.
- Viper is simple yet expressive. This makes the language convenient for humans, e. g. researchers developing novel verification techniques and students learning about formal verification. In particular, we present the subset of Viper considered in this thesis in Sec. 1.2.

program	(*axiom* | *field* | *function* | *method*)*
axiom	**axiom** *idnt*? { *FOL* }
idnt	Alpha-numeric identifiers, e. g. `x1`, `y2`
FOL	First-order logical formulas over built-in theories and uninterpreted function symbols
field	**field** *idnt*: *type*
type	***Bool*** | ***Int*** | ***Ref*** | ***Seq***[*type*] | ***Set***[*type*] | ***Map***[*type*, *type*]
function	**function** *idnt* (*params*?): *type* (**requires** *exp*)* (**ensures** *pureExp*)* { *pureExp* }?
params	*idnt*: *type* (, *idnt*: *type*)*
exp	*pureExp* | *accExp* | *exp* && *exp* | *pureExp* $\implies$ *exp*
pureExp	*literal* | *idnt* | *funApp* | *pureExp*.*idnt* | *pureExp*[*pureExp*] | *FOL*(*pureExp*) | *oldExp*
literal	**null** | **true** | **false** | **result** | Integer | SetLiteral | SeqLiteral | MapLiteral
funApp	*idnt* (*args*)
args	*pureExp* (, *pureExp*)*
oldExp	**old** [*idnt*]? (*pureExp*)
accExp	*acc* | *ISC* | *accExp* && *accExp* | *pureExp* $\implies$ *accExp*
acc	**acc** (*pureExp*.*idnt* (, (**read** | **write**))?)
ISC	**forall** *params* • *accExp*
method	**method** *idnt* (*params*?) (**returns** (*params*))? (**requires** *exp*)* (**ensures** *exp*)* { *stmt* }?
stmt	*decl* | *assign* | *branch* | *loop* | *call* | *spec* | **label** *idnt* | *stmt*;*stmt*
decl	**var** *idnt*: *type* (:= *pureExp*)?
assign	*target* (, *target*)^N := *pureExp* (, *pureExp*)^N $\quad N \geq 0$
target	*idnt* | *pureExp*.*idnt*
branch	**if** (*pureExp*) { *stmt* } (**elseif** (*pureExp*) { *stmt* })* (**else** { *stmt* })?
loop	**while** (*pureExp*) (**invariant** *exp*)* { *stmt* }
call	(*target* (, *target*)* :=)? *idnt* (*args*?)
spec	**assume** *pureExp* | **assert** *pureExp* | **inhale** *exp* | **exhale** *exp*

Figure 1.1: Core language grammar.

This grammar corresponds to the *core language*, i. e. a subset of the Viper language used in this thesis. For simplicity, we omit some features of Viper, e. g. user-defined uninterpreted types. Refer to Sec. 1.2.1 for an overview of the core language features.

The reasoning techniques of this thesis are designed for a flavor of separation logic called *implicit dynamic frames* [66]. Conceptually, this logic separates specifications of access permissions for memory locations from specifications of the values stored in these locations. For instance, separation logic's points-to predicate $x.f \mapsto v$ is specified in implicit dynamic frames as a conjunction of the access permission and the field content: $\mathbf{acc}(x.f) * x.f = v$. Parkinson and Summers [64] study the relationship between classical separation logic and implicit dynamic frames for this purpose, a variation of separation logic [64].

1.2 CORE LANGUAGE

Our goal is to design a specification language that enables the programmer to specify rich properties. For this purpose, we start with a *core* language and then extend it with rich specification features. The core language combines a specification language with imperative constructs, reflecting the common subset of operations typically supported in modern imperative languages with similar semantics, e. g. assignments, loops, branches, and procedure calls.

For the core language, we choose a subset of the Viper intermediate language. Although Viper is a relatively small language, it still has more than we need. For example, Viper supports abstract separation-logic predicates that are not needed in our setting; instead, we assume that all methods operating over unbounded object sets are specified via ISC. Thus, we start with a *subset* of Viper and then *extend* it with novel specification ingredients.

1.2.1 Core language overview

The core language grammar is presented in Fig. 1.1. A program consists of four kinds of members: axioms, fields, functions, and methods.[2]

TYPES AND BUILT-IN THEORIES. The type system supports *primitive types* and *composite types*. The former are the following interpreted types: *Booleans* **`Bool`** (consisting of **`true`** and **`false`**), (unbounded) *integers* **`Int`** (e. g. `-1`, `0`, `42`), and *references* to heap objects **`Ref`** (including the special **`null`** value for uninitialized **`Ref`**-type variables). Boolean values can be combined via the propositional connectives, e. g. $\neg$, $\wedge$, $\vee$, and first-order quantifiers $\forall$ and $\exists$. Integer values support the *less-than* and the *greater-than* relations (e. g. $a < b \Leftrightarrow b > a$) and can be combined via the following binary operators: `+`, `-`, `*`, `/` (integer *division result*), `%` (integer *division remainder*). Values of all primitive types can be compared for *equality* ($=$) and disequality ($\neq$).

2 Although Viper also supports import directives, user-defined types, and macro definitions (for both expressions and statements), we omit those from the grammar for brevity.

The composite type **Seq**[T] represents finite *sequences* of elements of type T. The supported sequence operators are *sequence length* (e. g. |xs|), *sequence lookup* (e. g. xs[i] for some i in range 0 .. |xs|-1) *concatenation* ++ (e. g. xs ++ ys), *sequence member* (e. g. x **in** xs), *take-prefix* and *drop-prefix* (e. g. resp. xs[..i] and xs[i..]). Conversely, **Set**[T] represents unordered, finite *sets* of elements of type T, for which the list of supported operators is: *set cardinality* (e. g. |A|), *set member* (e. g. a **in** A), *set union* (e. g. A **union** B), *set intersection* (e. g. A **intersection** B), and *set exclusion* (e. g. A **setminus** B). Sets can be compared via the (non-strict) *subset relation* (e. g. A **subset** B). Finally, **Map**[T, S] represents *finite maps* from the *domain* set of values of type T (e. g. **domain**(M)) to the *range* set of values of type S (e. g. **range**(M)); another supported map operator is *map lookup* (e. g. M[t] for some t in set **domain**(M)).

Literals of composite types are written as **Seq**($x_1, \ldots, x_n$), **Set**($x_1, \ldots, x_n$), and **Map**(x_1:=v_1, ..., x_n:= v_n). Type parameters of the empty sequence **Seq**(), the empty set **Set**(), and the empty map **Map**() are inferred from the context. Like primitive types, values of composite types can be compared for *equality* (=) and disequality (≠). Unlike built-in types that are interpreted, composite types are *uninterpreted* and inherently incomplete; their semantics is merely *approximated* in first-order logic. In particular, this is the reason why the core language provides **Seq**[T] which could theoretically be modeled as **Map**[**Int**,T].

AXIOMS. Axioms consist of a name and a body (a first-order formula). Note that axioms cannot refer to object fields or program variables.

FOL denotes closed first-order formulas over uninterpreted function symbols and the built-in theories: Booleans, integers, references, as well as type-parametric sequences, sets, and maps, where the type parameters can be instantiated recursively.

Quantifiers in our language are additionally annotated with *triggering patterns* that syntactically restrict the situations in which an SMT solver can instantiate them with ground terms. We will discuss triggering patterns in more detail in Sec. 2.1.2.

EXPRESSIONS. Pure expressions (denoted *pureExp* in Fig. 1.1) can be used in both the specifications as well as the implementation of the program. These expressions may depend on the heap (e. g. x.f) and the store memory (e. g. x + 1, where x is a local program variable).

In contrast, *accExp* refers to separation-logic expressions that can be used only in specifications; these expressions specify *access permissions*, held by the current thread, to potentially shared heap locations. An example separation-logic expression is $x \neq y \Rightarrow$ **acc**(x.f) && **acc**(y.f) which specifies permissions to two disjoint heap locations, namely, the field f of the objects x and y, but only if these two objects are indeed disjoint (otherwise, no permissions are specified).

ACCESS PERMISSIONS. The keyword **acc** denotes access permissions to a single memory location, taking either two or three arguments. The first two arguments are the *re-*

ceiver expression (referring to a non-null object) and a field name, resp. The optional third argument is a *permission amount*.[3] For example, **acc**($x.f$, **write**) and **acc**($y.g$, **read**) express *write* and *read access* permissions to the memory locations for fields f and g of objects x and y, resp.[4]

To specify permissions to an arbitrary set of objects, the programmer can use iterated separating conjunction [26, 90] (denoted *ISC* in the grammar). For example, **forall** r**:Ref**• $r \in G \Rightarrow$ **acc**($r.f$) denotes write access to all fields f of all objects in G.

FUNCTIONS. A function is either an executable procedure that cannot modify the heap memory or, if the body is omitted, an uninterpreted function symbol. Function applications (denoted *funApp* in Fig. 1.1) yield pure values and can be used inside complex expressions. The programmer may apply functions inside specifications as functions cannot modify the state. Functions can be applied also from within statements, in which case they represent side-effect-free procedure calls.

Consider the following heap-dependent function:

```
field val: Int
function cmp(x: Ref, y: Ref): Int
  requires acc(x.val) && acc(y.val)
{ (x.val < y.val) ? 1 : (y.val < x.val ? -1 : 0) }
```

For example, one can use `cmp` inside a method's postcondition, e. g. $\forall x \in G \wedge y \in G \bullet x \neq y \implies \texttt{cmp}(x, y) \neq 0$. This expression uses `cmp` and says that all pairs of different objects in a set G must have different values in their `val` field. Note that evaluating this expression is possible only in a state in which there are sufficient permissions to access all required information, i. e. **forall** x**:Ref** • $x \in G \implies$ **acc**(x.val).

Functions *with* bodies are verification targets; to verify such a function, one must prove, under the assumption of its precondition, that any successful evaluation of the function's body results in a state that satisfies its postcondition.[5] Body-*less* functions that are specified via **requires** or **ensures** are *assumed* to have some implementation (an expression) that respects their specification.

The keyword **result** can be used in a function postcondition (i. e. inside an **ensures** clause) in order to refer to the function's hypothetical value. In the following example, the postcondition of the body-less function `size` ensures that its result is non-negative:

```
function size(A: Set[Ref]): Int
  ensures 0 ≤ result
```

3 If omitted, the permission amount of **acc** defaults to **write** in method specifications and **read** in function specifications, as function may never *write* to object fields.

4 The former is an equivalent for $x.f \mapsto \_$ in classical separation logic [26].

5 In this thesis, we focus on verifying *partial correctness* as opposed to *total correctness* which additionally requires formally proving *termination* of all statements in the program.

A body-less function is treated as a special kind of axiom as it potentially introduces new, unjustified information via its postcondition. In particular, the definition of `size` could be replaced with the following axiomatization (assuming `size` is uninterpreted):

```
axiom { ∀A:Set[Ref] • 0 ≤ size(A) }
```

A function can be rewritten via a first-order axiomatization only if all of its preconditions are *pure*, i. e. the evaluation of the function cannot depend on the program state. For example, such a rewriting does not exist for a function that checks if an array is sorted:

```
function is_sorted(a: Array): Bool
  requires forall i: Int • 0 ≤ i < len(a) ⟹ acc(a[i].val)
  ensures ∀i, j • 0 ≤ i < j < len(a) ⟹ a[i] ≤ a[j]
```

METHODS. A method is a procedure that can modify heap memory. Method calls (denoted *call* in Fig. 1.1) may return a tuple of pure output values.

Methods with bodies are verification targets; to verify such a method, one must prove, under the assumption of its precondition, that any (halting) execution of the method's body results in a state that satisfies its postcondition.[6]

Different statements inside method bodies must either occupy different lines or be separated with a semicolon. Methods without bodies are *assumed* to have some implementation, i. e. a statement that respects their specification.

The client of a method call may assume that the call eventually returns. Verifying its own termination is thus the responsibility of the callee.

MULTI-STATE CONSTRAINTS. The keyword **label** is used for labeling a program state in a method body in order to refer to it via an **old**-expression in consecutive parts of the control flow. For example, consider the following code snippet:

```
label l1
x.f := true
exhale old[l1](x.f) ⟹ acc(x.f)
```

Here, we *remove* the access permissions (held in the last state) to `x.f` if this field *used to be* set to true in the state marked l_1. If the label in an **old**-expression is omitted, then it refers to the state of the method's precondition. Note that **old** affects only the evaluation of heap values while local store values are not affected by it.

The programmer can use **old**-expressions inside method postconditions (i. e. inside an **ensures** clause) in order to refer to the state of the method's precondition (inside a **requires** clause). In the following example, the method `sqrt` reassigns the value stored in field `val` of the object `cell` with the integer square root of its original value:

```
method sqrt(cell: Ref)
```

6 Reasoning about non-terminating programs and programs that throw exceptions is beyond the scope of this thesis.

```
requires acc(cell.val)
ensures cell.val * cell.val = old(cell.val)
```

SYNTACTIC SIMPLIFICATIONS. Our language provides *syntactic sugar*. In particular, we extend our standard grammar of Fig. 1.1 with the following cases:

- Parallel assignments, e. g. `a, b, c := x, y, z`.
- Grouped type annotations, e. g. `var a, b: Int`.
- Implicit types, e. g. `x`'s type is inferred from the signature of `P` in `forall x ● P(x)`.
- Conjunctions: && is interpreted as the *separating conjunction* ($*$) if it occurs in between two impure expressions (*accExp*); otherwise, && is interpreted as the first-order conjunction ($\wedge$).
- Specification blocks: each line break within a *specification clause*[7] is treated as a *top-level* &&.
- Implications, e. g. $x \Rightarrow y \quad \equiv \quad \neg x \vee y$.
- Ternary expressions, e. g. $z = ((x < y)\ ?\ x : y) \quad \equiv \quad (x < y \Rightarrow z = x) \wedge (\neg(x < y) \Rightarrow z = y)$.
- Ligatures, e. g. $\texttt{a} \notin \texttt{A} \quad \equiv \quad \neg(\texttt{a}\ \texttt{in}\ \texttt{A})$; $\texttt{A} \setminus \texttt{B} \quad \equiv \quad \texttt{A}\ \texttt{setminus}\ \texttt{B}$, etc.
- Disjoint unions, e. g. $\texttt{C} = (\texttt{A} \uplus \texttt{B}) \quad \equiv \quad \texttt{C} = \texttt{A}\ \texttt{union}\ \texttt{B} \wedge \forall x \in A \bullet x \notin B$.

1.2.2 Language rules

Our language supports exactly one class of objects that can be allocated on the heap. Thus, each object *has* all the fields declared in the program; *accessing* a particular field requires the current thread to hold at least read permissions to the corresponding memory location.

WELL-FORMEDNESS OF FUNCTIONS. Functions must explicitly require access permissions to the parts of the heap memory that they depend on. If a function's precondition is a pure expression (*pureExp* in Fig. 1.1), we call it a *first-order* function; otherwise (if the precondition requires some access permissions), we call it a *state-dependent* function. Since all functions are side effect-free, only *read* permissions can be specified in a function's precondition, and the precondition must be *self-framing* (i. e. it must specify sufficient

7 The specification clauses are: **requires**, **ensures**, **invariant**, **assume**, **assert**, **inhale**, and **exhale**.

permissions to allow for evaluating its pure conjuncts). Conversely, function postconditions must be *pure expressions*: A function is assumed to return all of its access permissions back to the client.

Function applications in statements must represent terminating computations. Function applications in specifications must be represented by some interpretation. Since function bodies and postconditions can be recursive, the interpretation of a function is defined via its least fixpoint. For more details about verifying termination of functions in our setting, refer to Streun's Bachelor's thesis [111].

A function is said to be *well-formed* if the following conditions hold: (1) the precondition (if present) is self-framing, (2) the postcondition and the body (if present) are pure expressions which can be evaluated in the state of the precondition. For example, the following functions are not well-formed, and any program containing any of these should be rejected by the consistency checker:

```
function nonSelfFraming(x: Ref, y: Ref): Bool
  // Missing permissions to access y.val; not self-framing.
  requires acc(x.val) && x.val = y.val

function postNotPure(x: Ref): Bool
  requires acc(x.val)
  // Permissions should not appear in the postcondition
  ensures acc(x.val) && result

function illegalBodyExpression(x: Ref): Int
  requires acc(x.val)
{ // Division by zero
  x.val / 0 }
```

WELL-FORMEDNESS OF METHODS. Reasoning in separation logic has the advantage that one can modularly verify properties of a method, and reuse this verification for all calling contexts (and concurrently-running threads).

A method is said to be *well-formed* if the following conditions hold: (1) the precondition, the postcondition, and all invariants of the loops in the method's body are self-framing, i. e. they specify sufficient permissions to allow for evaluating their pure conjuncts; (2) all the callee methods in the implementation are well-formed;[8] (3) all the functions applied in the specification and the implementation are well-formed.

Verifying a (well-formed) method amounts to constructing a correctness argument about all of its possible *execution traces*, e. g. any type-correct instantiation of the parameters cannot lead to a crash. Note that we focus on partial correctness, so the correctness arguments apply under the assumption that all the methods eventually halt. The programmer (or a frontend verifier) may *encode* termination checks into the core language.

8 Except for bodies of the callees as they may not be available in a modular setting. We assume that callees without bodies have *some* well-formed implementation.

MODULAR SPECIFICATIONS. A method's modular specifications consist of its precondition (written in the **`requires`** clause) and postcondition (written in the **`ensures`** clause).[9] The precondition prunes some of the traces that otherwise would need to be considered in verification. The responsibility of reasoning about these pruned traces is shifted to the client. Conversely, the postcondition is our main verification target. Hence, a (partial) correctness argument is one that shows that all traces permitted by the precondition in the initial state of the method call satisfy the postcondition in the final state of the call.

LOOP INVARIANTS. To automate the verification, we focus on *first-order separation logic* because there exists a number of reasoning engines for this logic that are effective and predictable [61, 78, 91]. However, first-order provers cannot automatically construct inductive proofs that are required e. g. for programs with unbounded loops. To enable verification of such programs, the programmer can specify each loop with a *loop invariant* (written in the **`invariant`** clause).

A loop invariant specifies the properties of execution traces in four program points: (1) before the very first loop iteration, (2) in the initial state of an arbitrary iteration, (3) in the final state of an arbitrary iteration, and (4) after all the iterations. (1) and (3) are assertions (i. e. verification targets) while (2) and (4) are assumptions. Conceptually, a loop invariant provides an *induction hypothesis* for verifying a number of consecutive executions of the loop body. Loop invariants are modular in the sense that only the access permissions mentioned in the invariant of the loop are available in its body.

LOCAL SPECIFICATIONS. In addition to loop invariants, pre-, and postconditions, the programmer may write *local specifications* (*spec* in Fig. 1.1) in a method's body.

The programmer can *prune* execution traces by specifying local properties of the surviving traces via **`assume`** statements (for *pure conditions*) or via **`inhale`** statements (for *access permissions* possibly conjoined with pure conditions). Unlike method preconditions, **`assume`** and **`inhale`** do not shift any responsibility and are entirely invisible to the client. A verifier does not have to check whether all possible execution traces actually reach a state in which the assumption holds; hence, assumptions may lead to unsound verification results. However, **`assume`** and **`inhale`** can be very useful e. g. in verification debugging, or in cases when the soundness of an assumption is justified on a meta level.

Conversely, the programmer can *check* whether all execution traces reaching a given program point have certain properties via **`assert`** statements (for pure conditions) or via **`exhale`** statements (for *access permissions* possibly conjoined with pure conditions). Unlike method postconditions, **`assert`** and **`exhale`** do not affect a method's modular specifications and are entirely invisible to its clients. These statements can be used e. g. to *query* the verifier and observe what facts can be automatically established (or what permissions are held) in a given program state.

9 W.l.o.g. we assume here that each method has exactly one pre- and postcondition.

MEMORY MANAGEMENT. Memory management can be modeled via methods. For example, allocating a new object with fields `val` and `next` is modeled via a call to **`new`**:

```
method new(v: Int, n: Ref) returns (r: Ref)
  ensures acc(r.val, write) && acc(r.next, write)
  ensures r.val = v ∧ r.next = n
```

Thus, the client can create a new object by writing, e. g. **`var`** `x :=` **`new`**`(-1,` **`null`**`)`. Note that this simple approach does not take into account the possibility that a memory allocation operation may fail.

1.3 COMPREHENSIONS

In Chap. 2, we study automated verification of higher-order compositional properties, called *set comprehensions*, in an ISC-based setting. Set comprehensions can express *comprehensive program properties* that summarize stateful aspects of potentially unbounded *commutative semigroups*, i. e. object sets equipped with an associative, commutative binary operator. Examples of comprehensive properties include *minimal array value*, *square matrix trace*, and *sum of marked object values in a graph*. Framing of set comprehensions is straightforward as they are by-definition compositional properties. The main problems are thus (1) to decouple data structure representations and memory specifications and (2) to efficiently axiomatize set comprehensions, enabling automated verification.

The state of the art for automated verification of comprehensions is Spec# [32, 59]. This technique supports *sequence-based comprehensions* which are a special case of set comprehensions. To the best of our knowledge, there is no work to date that integrates comprehensions into an automated separation logic-based setting.

APPROACH. To express comprehensive specifications, we introduce a novel class of expressions called *set comprehensions*, denoted **`comp`**, that summarize potentially unbounded object sets within the current method's footprint. Conceptually, set comprehensions generalize first-order universal quantifiers. Intuitively, the quantifier $\forall x.\, P(x)$ summarizes the value of $P(x_1) \wedge \ldots \wedge P(x_N)$ for some x_i, while the set comprehension **`comp`**`[`$\oplus$`]` $Q(x)$ summarizes the value of $Q(x_1) \oplus \ldots \oplus Q(x_N)$, for the same elements x_i, but (1) $\oplus$ may be instantiated with any commutative, associative binary operator, e. g. `+`, and (2) Q may have *any type*, e. g. **`Int`**. The semantics of set comprehensions is bound to the current method's footprint in the state in which they are evaluated. To define comprehensive properties of *substructures*, the programmer can specify the exact set of objects that a comprehension summarizes by customizing its *filtering condition*. The value of a set comprehension is defined in terms of comprehensions over *any* two disjoint subsets of the original objects, which enables precise framing in presence of arbitrary method calls.

To uniformly support different heap-allocated structures, we require that data structure elements are accessed only through its *lifted representation*. The lifted representation

provides an abstract view over objects by mapping identifiers to memory locations. For example, a graph object is identified directly via its reference while an array element is identified via the array's name and an integer index. The availability of lifted representations allows for a uniform style of ISC-based memory specifications in all methods and state-dependent functions of our technique. In particular, the lifted representation's *domain* yields the set of all identifiers of currently accessible objects.

To encode comprehensions into ISC-based separation logic, we decouple a comprehension's stateful and stateless aspects. The former are encoded via a state-dependent function called *snapshot* that yields a mathematical map from object identifiers (leading to accessible objects) to the corresponding body term values. The latter are encoded as uninterpreted functions over snapshot maps. We axiomatize these functions only partially, as a complete first-order axiomatization of comprehensions cannot exist. Our axioms are enable modular reasoning about comprehensions in presence of arbitrary field updates and method calls and automate practically essential lemmas.

CONTRIBUTION 1. We propose a novel technique for automated verification of modular comprehensive specifications of heap-transforming programs. Our technique encodes comprehensions into separation logic, extending its specification language and providing compatibility with a spectrum of other separation logic-based techniques and tools.

1.4 REACHABILITY

In Chap. 3, we address the grand challenge of this thesis, namely, establishing composability of heap reachability. As we argue above, reachability is generally *not* a compositional property; establishing composability thus requires a tradeoff in terms of the generality of the setting.

The problem occurs is presence of a method call. If the postcondition of the callee ensures that there exists a heap path, then the client can rely on this information. Conversely, if the postcondition of the callee ensures that there *does not* exist a heap path, then the client *cannot* rely on this information as heap paths might traverse objects beyond footprint boundaries. Thus, an alternative formulation of the same problem is to *split* original heap paths at footprint boundaries, partitioned each original path into some number of sub-paths the existence of which can be checked modularly.

Foundational work on reasoning about reachability solves the framing problem but only for the simplest of operations, e. g. updates of individual heap edges [17, 97]. Operations modifying *unbounded* heap fragments, e. g. method calls, remain unsupported.

There are two existing approaches in the state of the art for *compositional* reasoning about reachability. The first approach restricts classes of supported heap structures and possible decompositions, reducing the framing problem to propositional logic in order to achieve automation [76, 78]. Consequently, these techniques support structures with

only *deterministic paths*, i. e. if there exists a path $x \dots y$, where x and y refer to some objects, then this path is unique. For example, structures with deterministic heap paths include various forms of linked lists and trees but do not include DAGs with unbounded sharing.

The second approach is to embrace higher-order reasoning, trading off automation [116]. Another higher-order technique that may be automated in future supports only operations that *preserve* their reachability properties [100], i. e. one cannot *propagate* new reachability information established by a callee to its client's context.

APPROACH. To enable reachability specifications in a modular setting, we introduce a novel relation called *local reachability* that expresses the existence of heap paths only within given heap fragments. Conceptually, local reachability generalizes classical heap reachability, which expresses the existence of *global* heap paths, but constraining global heap paths is typically undesirable and is generally not possible in a concurrent setting. Conversely, the semantics of the local reachability relation is bound to the current method's footprint in the state in which they are evaluated.

To encode local reachability into ISC-based separation logic, we follow the same approach as with comprehensions, decoupling reachability's stateful and stateless aspects. The former are encoded via a version of the snapshot function that yields *edge sets* of a mathematical graph representing the heap edges in a fixed state. The latter are encoded as an uninterpreted function over snapshots. We axiomatize this function only partially, as a complete first-order axiomatization of local reachability cannot exist; however, our axioms are sound, systematically derived, and tuned based on real-world benchmarks.

To enable precise, modular reasoning about local reachability, we develop the notion of *relatively convex footprint decompositions*. If a footprint of a method call is convex relatively to its client's footprint, then framing of local reachability information becomes a first-order problem and can be automated. On the one hand, nested footprints are often relative convex in practice, but one can check this automatically via first-order conditions. On the other hand, relative convexity is permissive enough to support highly-general heap structures, even those with cycles or non-unique paths, and decompositions thereof.

CONTRIBUTION 2. We propose a novel technique for automated verification of modular reachability specifications of heap-transforming programs. Our technique encodes reachability into separation logic, extending its specification language and providing compatibility with a spectrum of other separation logic-based techniques and tools.

This chapter is based on the following publication:

> [112] Arshavir Ter-Gabrielyan, Alexander J Summers, and Peter Müller. "Modular verification of heap reachability properties in separation logic." In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019)

1.5 VERIFICATION DEBUGGING

In Chap. 4, we develop a counterexample-based verification debugging technique for our setting. In case an SMT solver reports that a verification condition is invalid, the failure is typically accompanied by a *counterexample model* containing variable assignments and function interpretations that contradict the assumptions. In presence of undecidable logics, SMT solvers provide *partial models* that are imprecise, e. g. due to incomplete instantiations of the quantified formulas. Our hypothesis is that, despite their inherent incompleteness, counterexample models of undecidable proof obligations are helpful for understanding verification failure causes.

In our automated verification paradigm, the internal encoding of the verification technique is not exposed to the programmer. Therefore, the programmer cannot easily interpret raw counterexample models. Thus, there are three main problems of counterexample-based verification debugging: (1) to *systematize* extraction of state-dependent information from counterexample models in a way that is verification backend-agnostic, i. e. supporting both symbolic execution and verification condition generation; (2) to soundly resolve potentially *aliasing* objects; and (3) to find the right approach for *delivering* counterexample information to the programmer.

State of the art verification debuggers that rely on counterexample SMT models target either symbolic execution [56, 109] or verification condition generation [62, 63, 102], but not both of these complementary verification technologies. Propositional techniques for reachability verification [76, 78] provide counterexample heap models by leveraging SMT models that, in their setting, are guaranteed to be sound and complete. These heap models display only concrete heap edges but do not display *un*reachability relations that are needed to express e. g. acyclicity.

APPROACH. To generate heap models that help the programmer diagnosing verification failures, we start with *partial counterexample models* generated by the SMT solver. For systematizing the extraction of heap information, we fix the expected model shape using the notions that are (1) available in an ISC-based setting and (2) meaningful to programmers who might not be separation logic experts. In particular, our models contain objects (as heap nodes) and field values (as heap edges) in all states, as well as method footprints and non-reference constants.

We extract state-dependent information by employing *heap markers*, i. e. quasi-backend-agnostic model values that parameterize interpretations of stateful functions, e. g. those defining field values. We model only those objects that are reachable from local program variables. For transitively-reachable objects, we run a saturation-based procedure. To resolve aliasing, especially among transitively-reachable objects, we employ a custom protocol for managing equivalence classes.

Our approach naturally supports ISC-based separation logic extensions, such as set comprehensions and heap reachability (although we demonstrate this only for the lat-

ter case). To support local reachability relations, we extend our models with new kind of data representing known local reachability relations. While collecting this data is straightforward (which is an advantage of our technique), we dramatically simplify our heap reachability models by filtering out redundant information that is present in the original SMT model, e. g. paths that follow concrete heap edges.

CONTRIBUTION 3. We propose the first backend-agnostic algorithm for extracting heap reachability models from counterexample SMT models. Visualizations of the produced heap models supports verification debugging and are helpful even to programmers not familiar with separation logic. We demonstrate extensibility of our algorithm by augmenting concrete heap models with general heap reachability relations.

We implemented and integrated our algorithm into Viper IDE.

> [125] Arshavir Ter-Gabrielyan. *Lizard: The Visual Debugger for Viper*. 2021. URL: https://github.com/viperproject/lizard

1.6 FURTHER CONTRIBUTIONS DURING THE THESIS WORK

The author has made the following contributions that are not part of this thesis.

VIPER IDE is an interactive verification IDE for the Viper language. The core infrastructure of this project includes a *language server* that also powers the IDEs of two Viper frontends, namely, for Rust and Go. Additionally, Viper IDE supports the visual verification debugger discussed in Chap. 4.

> [121] Linard Arquint, Alessio Aurecchia, Ruben Kälin, Valentin Racine, and Arshavir Ter-Gabrielyan. *Viper IDE*. Aug. 2021. URL: https://github.com/viperproject/viper-ide

AXIOM TESTING is an automatic technique for identifying problematic patterns in SMT axiomatizations with quantifiers. This project has led to the following publication:

> [122] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. "Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers." In: *Formal Methods (FM)*. To appear. 2021

2 COMPREHENSIONS

In this chapter, we discuss reasoning about comprehensive properties of heap-transforming programs. Intuitively, these properties summarize data structures containing potentially unbounded (and statically unknown) sets of nodes via a finite number of values. For example, a universal quantifier in first-order logic can be used to specify comprehensive properties of a data structure, e. g. $\forall x \in \texttt{xs} \bullet P(x)$. This formula says that the Boolean function P holds for all nodes of some structure denoted `xs`.[1] This way, a first-order quantifier can combine the values of a Boolean function of individual nodes into a single value that characterizes the data structure as a whole.

EXAMPLES. Consider the *binary max-heap* [4] data structure instance `BH`, and its *heap property*: The key stored in a node `n` of `BH` is greater than or equal to the keys of all of `n`'s descendants. This is a comprehensive property that can be expressed as $\texttt{n.key} = \max_{x \in \texttt{des(BH, n)}} x\texttt{.key}$, or (equivalently) via the following quantified formula: $\forall x \in \texttt{des(BH, n)} \bullet x\texttt{.key} \leq \texttt{n.key}$; here, `des(BH, n)` is the subset of `BH` including all the descendants of `n`. Verifying the binary max-heap data structure requires proving that the heap property is *preserved* by all of its operations, e. g. insertion of a new node and deletion of the minimal node. However, *automating* such proofs is challenging because SMT solvers are inherently incomplete for reasoning with first-order quantification.

Furthermore, many verification problems require *higher-order* comprehensive properties that remain entirely unsupported by modern SMT solvers. For example, the specification of the DIJKSTRA *shortest-path finding algorithm* [2] includes the property that the resulting path's *distance*, i. e. the sum of distances between consecutive nodes along the path from the *source* node `s` to the *target* node `t`, is less than or equal to the distance of *any* of the possible paths `s` ... `t`; i. e. the resulting path is indeed the shortest: $\sum_{(p,q) \in \textbf{result}} \texttt{dist}(p, q) \leq \sum_{(p,q) \in \texttt{path}} \texttt{dist}(p, q)$. Here, **`result`** is the sequence of (predecessor, successor) node pairs computed by DIJKSTRA, `path` is some other path from `s` to `t` (also represented as a sequence of node pairs) connecting the source and destination nodes, and $\texttt{dist}(p, q)$ is the distance between nodes p and q. This comprehensive property cannot be expressed in first-order logic because it involves *summation* over (potentially) unbounded sets.

DEFINITION. We define *comprehensions* as (potentially, higher-order), scalar-valued functions over one first-order predicate (defining the set of considered nodes) and one first-

1 In slight notation abuse, we use the same symbol to denote a data structure and the set of its nodes.

order function (defining the values of particular nodes). For example, $\sum_{x \in \mathtt{xs}} f(x)$ is a comprehension. While reasoning about programs, we will refer to their *comprehensive properties*, i. e. logical assertions that syntactically contain comprehensions, and their *comprehensive specifications*, i. e. Hoare-style specifications, such as structural invariants, written in terms of comprehensive properties. Comprehensive properties can be used to summarize unbounded sets of nodes using a finite set of values. For example, $0 \leq \left(\sum_{x \in \mathtt{xs}} f(x)\right) + \left(\sum_{x \in \mathtt{xs}} g(x)\right) < 3$ is a comprehensive property over two comprehensions.

Next, we will identify the fundamental challenges in the context of reasoning about comprehensive properties of programs.

AUTOMATED REASONING. We are interested in *automated* verification of comprehensive properties. Comprehensive properties often can have multiple alternative representations. For example, consider Property (1): $42 = \sum_{x \in \mathtt{xs} \wedge f(x) > 0} 1$. This property says that there are exactly 42 nodes in `xs` for which the value of f is positive. Alternatively, one could write Property (2): $42 = \sum_{x \in \mathtt{xs}} f(x) > 0 \ ? \ 1 : 0$, which is mathematically equivalent to (1). Verification of comprehensive properties involves proving *entailments* such as $(1) \models (2)$ that may contain different representation of related comprehensive properties. Since our logic with comprehensions *extends* first-order logic, the problem of automatically proving such entailments is generally undecidable.

How can one automatically convert various possible representations of the relevant comprehensive properties? While this problem occurs even in stateless, purely functional settings, the next problem is specific to heap-transforming programs.

STATEFUL PROGRAMS. Further, we are interested in the verification of imperative, *heap-transforming programs*. The inherent complexity of these programs comes from the fact that they manipulate *stateful* data structures. In particular, reasoning about comprehensive specifications of stateful structures presents an additional challenge. Consider some operational semantics of the programming language (i. e. based on the known implementation of all the operations in our program), a fixed program state, and a set of comprehensive properties known in this state. The challenge is to infer what comprehensive properties hold in the *subsequent* states of the program.

To illustrate the problem, consider the operation `addAtEnd(xs,y)` that attaches a node referenced by `y` to an (acyclic) singly-linked list starting in `xs`, defined via the reference field `next`. One comprehensive property of acyclic lists is that the number of nodes that have a successor is *one less* than the total number of nodes in the list because only the tail

node of the list does not have a successor. We are interested in using the specification of `addAtEnd` to verify that it preserves the above-mentioned invariant of our list, i. e.:

$$\left\{ \begin{array}{l} \texttt{len(xs)}-1 = \sum\limits_{x \in \texttt{xs} \wedge x.\texttt{next} \neq \texttt{null}} 1 \\ \wedge\ \texttt{y.next} = \texttt{null} \\ \text{Precondition} \end{array} \right\} \ \texttt{xs:=addAtEnd(xs,y)} \ \left\{ \begin{array}{l} \texttt{len(xs)}-1 = \sum\limits_{x \in \texttt{xs} \wedge x.\texttt{next} \neq \texttt{null}} 1 \\ \wedge\ \texttt{len(xs)} = \texttt{old}(\texttt{len(xs)})+1 \\ \text{Postcondition} \end{array} \right\}$$

Here `len` denotes the list's length. We use the notation from Hoare logic [22]: Provided the *precondition* of this Hoare triple holds, the successful execution of the statement `xs:=addAtEnd(xs,y)` shall result in a new program state satisfying the *postcondition*.

Given the knowledge about all the comprehensive properties that hold in a given program state and the semantics of the operation invoked in this state, how can one automatically derive the comprehensive properties of the resulting state? Each heap structure can be characterized via an infinite number of different comprehensive properties.[2] Therefore, the general problem can be practically reduced to *checking* that the desired comprehensive properties in one program state follows from the known comprehensive properties in another state.

MODULAR REASONING. Another related challenge is *framing* of comprehensive properties. In a modular setting, a procedure's client needs information about the *frame* of the call, i. e. the parts of the heap that are not affected by it. Only properties that depend exclusively on the frame are guaranteed to be preserved by the call. Revisiting the example above, consider (in addition to `xs`) another heap-allocated list, say `zs`. Assuming the two lists do not overlap ($\texttt{xs} \perp \texttt{zs}$ and $\texttt{y} \notin \texttt{zs}$), the call to `addAtEnd(xs,y)` does not change the comprehensive properties of `zs`; e. g. the following Hoare triple holds (for some N):

$$\left\{ \begin{array}{l} N = \sum\limits_{z \in \texttt{zs} \wedge x.\texttt{next} \neq \texttt{null}} 1 \\ \text{Frame} \end{array} \right\} \ \texttt{xs:=addAtEnd(xs,y)} \ \left\{ \begin{array}{l} N = \sum\limits_{z \in \texttt{zs} \wedge x.\texttt{next} \neq \texttt{null}} 1 \\ \text{Frame} \end{array} \right\}$$

Given an operation's modular specification, how can one automatically derive the comprehensive properties that are *framed*, i. e. preserved by this operation's invocation?

OUTLINE OF THE CHAPTER. This chapter addresses the problem of automating deductive reasoning about comprehensive properties of heap-transforming programs, i. e. properties that can be specified via comprehensions and first-order connectives. We discuss the background of the problem of reasoning about higher-order comprehensions, as well as the related work, in Sec. 2.1. We present our set-based technique for automated modular specification and verification of comprehensions in Sec. 2.2. We proceed with a study of

2 For example, consider the property $3 < \sum_{x \in \texttt{xs}} x < c$ of a structure `xs`, for some constant c. There are infinitely many possible values for c, each leading to a unique comprehensive property.

two challenging cases in Sec. 2.3. We then describe the logical encoding of our technique in Sec. 2.4. The experimental evaluation is in Sec. 2.5. Finally, we conclude the chapter with a discussion of Sec. 2.6.

2.1 BACKGROUND

For the past forty years, automated verification of comprehensive properties of heap-transforming programs has been an ongoing research topic. Various approaches to this challenge have been supported by the results in *automatic theorem proving* and SMT solvers over the same course. In theorem proving, the established consensus is that a problem can be efficiently automated *iff* it can be simulated in first-order logic (with decidable theories) with sufficient precision. It is easy to check this claim for those problems that are decidable: For example, consider the spectrum of fully automatic verification techniques presented in Itzhaky's thesis [75]. In contrast, for the undecidable problems, e. g. reasoning with unbounded quantification that is essential for encoding comprehensions in first-order logic, it is much harder to define, let alone formalize in a useful way, the notions of *efficient automation* and *sufficient precision*.

Nevertheless, there exist many successful techniques that tackle such undecidable problems, including those in the context of formally verifying a program. These techniques typically operate in two steps. The first step involves identifying a suitable *first-order simulation* of the original problem: One replaces the original higher-order functions with uninterpreted function symbols constrained by a first-order axiomatization to respect the essential properties of the original functions. For example, one could apply this technique to encode the (undecidable) theories of sets, sequences, maps, and multisets, as is common in modern automated verifiers, e. g. Dafny [57], F* [71], and Viper [91].

The second step involves applying the first-order axiomatization to check the *verification conditions* the validity of which indicates that the program indeed adheres to its formal specification. One way to automate the checking is to employ a first-order prover, e. g. Vampire [70]. Such tools are efficient in finding refutational proofs, even if these proofs involve extensive quantified reasoning. However, first-order provers provide limited support for *theories*, e. g. arithmetic, which is essential in the context of program verification. Therefore, automated verifiers typically check verification conditions via *satisfiability modulo theories* (SMT) solvers that natively support common theories. To handle quantified formulas, SMT solvers employ (inherently incomplete) *quantifier instantiation strategies*. Although incomplete, the outcomes of these strategies are predictable, making them well-suited for program verification.[3]

3 A practically useful verification tool should be *responsive*, e. g. yield its best response within seconds, to allow the programmer to iteratively refine the specification and the implementation.

2.1.1 Verification as an SMT problem

The key idea behind using SMT solvers in formal verification is that a verification condition, C, follows from an axiomatization, A, *iff* the query formula $A \wedge \neg C$ is unsatisfiable. A first-order formula is said to be *satisfiable* if there exists an interpretation of all of its functions under which this formula is true (note that free and existentially quantified variables are interpreted as Skolem functions). If the query $A \wedge \neg C$ is unsatisfiable, then C is said to be a *valid* assertion under the assumption of A; otherwise, the assertion C is *invalid* at least for one interpretation of the query, called a *counterexample* to C under the assumption of A. For example, consider the following definitions of A and C:

$$A \;:=\; g(1) = 1 \wedge \forall x \bullet f(x) = g(x) \qquad\qquad C \;:=\; \forall y \bullet f(y) = y$$

Here f and g are two unary functions. Then the query $A \wedge \neg C$ is equivalent to $(g(1) = 1 \wedge \forall x \bullet f(x) = g(x)) \wedge (\exists y \bullet f(y) \neq y)$, which is a satisfiable formula because there exist interpretations of f and g that turn it to true, for instance, if both these functions are defined simply as the constant 1, in which case we can pick e. g. 0 for the existentially quantified variable, satisfying the query. Therefore, C is not a valid assertion under A, and $f(\ldots) = g(\ldots) = 1 \wedge y = 0$ is the counterexample.

Interestingly, establishing satisfiability and unsatisfiability of first-order formulas are different problems that generally cannot be reduced to one another. In our example above, A might still entail C for some other interpretation of f and g; for instance, this would be the case if both these functions are defined as the identity function. In particular, this shows that the negation of an invalid assertion is not necessarily a valid assertion, either.

2.1.2 Reasoning with unbounded quantification

Naturally, our SMT queries are expected to contain quantification (since axioms typically express properties of functions for arbitrary argument values), which is a major source of undecidability in a first-order setting. Hence, the challenge is to identify the set of concrete *instantiations* of the quantifiers of the query $A \wedge \neg C$ that would be sufficient for establishing whether it is satisfiable, yet small enough to avoid infinite instantiation chains and combinatorial explosions.

In his thesis, Nelson introduced *E-matching* [11], what has become the most prominent quantifier instantiation technique for program verification with undecidable theories [44].[4] The idea behind E-matching is to annotate all quantifiers with syntactic *patterns* that wrap each quantified variable into some terms of (exclusively uninterpreted) function symbols; when the solver reaches a term that *matches* a pattern, it instantiates

4 Reasoning about decidable theories requires a bounded number of quantifier instantiations, and the corresponding decidability problem can be efficiently solved via other techniques, e. g. MBQI [51] or enumerative instantiation [103].

the corresponding quantifier, using the matched *ground term* as the value for the respective quantified variable. For example, consider the assertion $C := R(P(42)) \wedge \neg Q(42)$ and the axiomatization A about three symbols, P, Q, and R:

$$\begin{aligned} A &:= A_1 \wedge A_2 \\ A_1 &:= \forall x \bullet \{ P(x) \}\ P(x) \implies R(Q(x)) \\ A_2 &:= \forall y \bullet \{ R(y) \}\ R(y) = y \end{aligned}$$

Here, the quantifiers' patterns are written between curly braces. To check whether C holds under A (i. e. whether $A \wedge \neg C$ is *unsatisfiable*), E-matching will start by identifying the possible instantiations triggered by the pre-existing ground terms of $A \wedge \neg C$:

Ground term	Triggered instantiation	Resulting term
$P(42)$	A_1 with 42 for x	$P(42) \implies R(Q(42))$
$R(P(42))$	A_2 with $P(42)$ for y	$R(P(42)) = P(42)$
$Q(42)$	Does not trigger anything.	

The implication obtained from the first instantiation above yields a new ground term, $R(Q(42))$, which will trigger one more instantiation:

$R(Q(42))$ A_2 with $Q(42)$ for y, resulting in $R(Q(42)) = Q(42)$

Note that the newly obtained terms, e. g. $R(Q(42))$, cannot trigger new instantiations as identical instances of these terms have already appeared before. Therefore, all available triggering terms have been used up, and E-matching saturates. The information obtained so far includes the negation of the assertion C, $\neg R(P(42)) \vee Q(42)$, as well as the formulas obtained through E-matching, in particular, $P(42) \Rightarrow R(Q(42))$ and $R(Q(42)) = Q(42)$. Now it is easy to check (by solving a propositional satisfiability problem) that all these quantifier-free formulas are satisfiable; in fact, that is the case for all interpretations of the three functions P, Q, and R. This means that the original assertion, C, is *invalid*, wheres, in this case, the negated assertion $\neg C$ *is valid*, under the assumption of A. Note that modern implementations used in state-of-the-art SMT solvers, e. g. Z3 [47] and CVC4 [60], apply E-matching lazily to maintain smaller sets of ground terms for as long as possible.

2.1.3 Obtaining the first-order axiomatization

Given an appropriate first-order axiomatization that simulates the essential properties of comprehensions, one could automatically and efficiently check programs against their comprehensive specifications using existing techniques, such as E-matching-powered

SMT solving discussed above. Obtaining this axiomatization is an orthogonal concern. Both of these challenges have been attacked from three main directions.

2.1.3.1 Program refinement. The first line of work concentrates on program refinement. Perfect Developer [32] has been one of the first languages and tools that enabled verifying comprehensive specifications of object-oriented programs. The programmer is free to define their own operations that can be iteratively applied to the elements of built-in collections, including lists, sets, and multisets; the resulting comprehension can be used both in the implementation and the specification of a program. The tool uses a custom theorem prover that attempts to statically check whether the specifications are satisfied by all possible inputs. While Perfect Developer has pioneered the methodology of formal refinement of object-oriented programs, the technique emphasizes the need for the programmer to refine both the implementation and the specification in order to mitigate its incompleteness.

2.1.3.2 Automatic first-order reduction of higher-order logic. The second line of work, initiated by Meng and Paulson, considers the general problem of automatically reducing proof obligations written in a higher-order logic, e. g. Isablelle/HOL [25], into first-order logic that is amenable to automation via SMT solvers. In particular, the focus of their work is on translating types [38] which are the foundation of Isabelle's formalism.

Since comprehensions are higher-order functions, one could attempt to use such general-purpose translators to obtain a first-order encoding for comprehensions that would soundly simulate their semantics. While in theory this approach can be viewed as a fully automatic way to solve the problem, its main limitation is the lack of predictability. An automatically generated first-order axiomatization would typically be sub-optimal for any specific domain, both in terms of its completeness and performance. Moreover, the programmer that experienced an incompleteness would have to resort to manual theorem proving techniques.

2.1.3.3 First-order simulation of undecidable theories. In contrast, Nelson, the author of E-matching, has initiated another line of work that is focused on developing specific first-order axiomatizations for automating reasoning about theories essential for program verification. Since many of these theories are undecidable, full theoretical completeness cannot be achieved in a general setting, e. g. with arbitrary comprehensions and data structures. However, as we discuss next, it *is* possible to obtain an axiomatization that performs well with modern SMT solvers and is complete under additional assumptions that do not limit the practicality of the overall verification technique, at least on a large set of characteristic benchmarks.

2.1.4 Comprehensions in Spec#

In the context of comprehensions, Leino and Monahan follow Nelson's approach by extending the Spec# program verifier [59] for automatically verifying programs with comprehensions [43]. We will next present this work in more detail.

2.1.4.1 Encoding into Boogie. To verify a Spec# program, the tool translates it into an *intermediate verification language* called Boogie [46], for which there already exists an automated SMT-based verifier. Consider a simple example that illustrates one aspect of this translation. Boogie does not natively support arrays, so the Spec# notation $\texttt{a}[i]$ for accessing the i-th element of the array `a` is translated as follows (the $\twoheadrightarrow$ arrow maps the Spec# notation to Boogie):

$$\texttt{a}[i] \quad \twoheadrightarrow \quad \mathit{ArrayGet}(\$\mathit{Heap}[\texttt{a}, \$\mathit{elements}],\ i) \tag{2.1}$$

Here $\$\mathit{Heap}$ is the map encoding the program heap into Boogie; in our example, it takes an array identifier and a field name (the only two available fields for the array type are $\$\mathit{elements}$ and `Length`) and returns a mathematical sequence of array elements (denoted `Elements`). The uninterpreted function symbol *ArrayGet* is then axiomatized to express some of its properties, e. g. injectivity. Using these axioms, Boogie can discharge its verification conditions to an SMT solver to verify specifications that depend on (a limited version of) the array theory.

Leino and Monahan have introduced the first notation for talking about comprehensions in a first-order setting. In particular, they propose a syntax and an intuitive definition of *comprehension expressions*. For example, the comprehension expression
`sum {int` k `in` $(0 : \texttt{a.Length}),\ k\%(M-1) = 0;\ \texttt{a}[k]\}$ corresponds to the sum of each M-th element ($x\%y$ denotes the remainder of integer division x/y). Using the common pen-and-paper notation, this *comprehension instance* can be written as:

$$\sum_{0 \le k < \texttt{a.Length} \,\wedge\, k\%(M-1)=0} \texttt{a}[k] \tag{2.2}$$

GENERAL PRINCIPLES. Generally, a comprehension instance supported in this technique has the form $\underset{L \le k < H \wedge F(k,\vec{a})}{\mathcal{Q}}\ T(k,\vec{a})$, where $\mathcal{Q}$ can be instantiated with one of the supported comprehensions: sum, count, product, min, or max. Here L and H are the lower and the upper bounds of the range of the index, k; $F(k,\vec{a})$ is the filter that may depend on k as well as a number of other subterms, denoted $\vec{a}$; and $T(k,\vec{a})$, called the *body* term of the comprehension, defines the value corresponding to particular indices k.

The key idea in this work is to axiomatize comprehension *templates* defining an entire class of comprehension instances, rather than supplying axioms for each individual instance. Generally, a template is a triple of the form:

$$(\mathcal{Q},\ \ F(k,\vec{a}),\ \ T(k,\vec{a})) \tag{2.3}$$

Using the template (2.3), the Spec# technique defines a class of comprehension instances in two steps. First, it generates a *comprehension function* representing the possible values of all comprehension instances from this class. Second, it instantiates a *generic axiomatization schema* with a concrete filter F and a concrete body T, generating one static axiomatization that defines a partial semantics of the comprehension function; we postpone the discussion of the axiomatization schema until Sec. 2.1.4.2. The comprehension function has the following signature:

$$\texttt{function}\ \mathcal{Q}\#n\texttt{(L:int, H:int, }\vec{a}\texttt{) returns (int);} \tag{2.4}$$

The numerical identifier n is an integer freshly picked for this particular comprehension function; the first two arguments, `L` and `H`, represent the lower (inclusive) and the upper (exclusive) bounds of the range of the internal index variable denoted k in (2.3); the symbol $\vec{a}$ represents all other arguments of this comprehension's filter and body (the translation would unpack this into a list of concrete arguments).

Finally, the technique uses the following translation of comprehension expressions via comprehension functions:

$$\mathcal{Q}\texttt{\{int }k\texttt{ in (L:H), }F(k,\vec{a})\texttt{; }T(k,\vec{a})\texttt{\}}\ \twoheadrightarrow\ \mathcal{Q}\#n\texttt{(L, H, }\vec{a}\texttt{)} \tag{2.5}$$

EXAMPLE. To find appropriate definitions for F and T in the comprehension template (2.3), the technique processes the comprehensive specifications written by the programmer, choosing the *least specific* terms that syntactically wrap the index, k. Then, the technique adds as many arguments to the signature of the comprehension function as there are *holes* in the selected term. For example, using the translation rule (2.1), the comprehension instance (2.2) belongs to the following template:

$$(\texttt{sum},\ \ k\%a_0 = a_1,\ \ \mathit{ArrayGet}(a_2,\, k)) \tag{2.6}$$

Here a_i denotes an occurrence of an arbitrary (appropriately typed) term. Using this template, the technique would generate the following comprehension function:

$$\texttt{function sum\#0(L:int, H:int, }a_0\texttt{:int, }a_1\texttt{:int, }a_2\texttt{:Elements) returns (int);} \tag{2.7}$$

Here a_0, a_1, and a_2 represent the values of some terms that fit into the holes of our template (2.6); in this case, the first two holes are of type `int` and belong to the filter while the third hole is of type `Elements` and belongs to the comprehension body.

The resulting comprehension function is then used for translating all comprehension expressions that match the same filter, e. g.:

$$\texttt{sum\{int } k \texttt{ in (L:H), } k\%(M-1) = 0\texttt{; a[}k\texttt{]\}} \twoheadrightarrow \texttt{sum\#0(L,H,}M-1\texttt{,0,}\mathit{Heap}\texttt{[a,\$}\mathit{elements}\texttt{])} \tag{2.8}$$

The main advantage of this encoding is that the programmer is free to use arbitrary filters and body terms in their comprehensions. In particular, this enables nested and multidimensional comprehensions, which are explored by the same authors in a follow-up paper [52]. However, an important limitation is that mathematically equivalent comprehension expressions might have different templates, in which case they will be translated using different comprehension functions and axiomatized using disjoint sets of axioms, resulting in incompleteness. For example, the following two comprehensions represent the same value, although this fact cannot be automatically verified because they have different templates:

$$\texttt{sum\{int } k \texttt{ in (0 : a.Length), L} \leq k < \texttt{H; a[}k\texttt{]\}} \twoheadrightarrow \texttt{sum\#1(...)}$$
$$\texttt{sum\{int } k \texttt{ in (L:H), true; a[}k\texttt{]\}} \twoheadrightarrow \texttt{sum\#2(...)}$$

2.1.4.2 Axiomatization. The Spec# technique for verifying comprehensive specifications generates a set of first-order axioms about each comprehension function, such as **sum#0** in (2.8). These axioms are generated according to the *axiom schemas* that are identified through manual inspection and generalize the properties that are required in characteristic proof outlines.

AVOIDING MATCHING LOOPS. To maintain a manageable search space for the SMT solver, the technique limits the number of instantiations of axiom quantifiers. This is achieved by using *synonym functions*, i. e. pairs of uninterpreted function symbols representing each comprehension function. The two functions have identical semantics: They are axiomatized to be equal over all possible arguments. While only the original symbol, e. g. **sum#0**, is exclusively used in (the translation of) the actual specifications and the triggering patterns of the quantifiers, the synonym, e. g. `s#0`, is exclusively used in the *bodies* of the quantified axioms. This avoids the possibility of *matching loops* i. e. infinite instantiation chains caused by quantifiers that continuously yield new triggering terms.

AXIOM SCHEMAS. In the following, we consider a single comprehension function, e. g. **sum#1**, and assume that the triggering patterns of all the quantified formulas respect the principle described above, i. e. there is one axiom that defines the semantics of the synonym function. To simplify the notation in all other axioms, we use just the function symbol `s` to refer to the comprehension function. For this comprehension function, the Spec# technique emits seven first-order axioms, as explained next.

Note that in all axiom schemas below, $\vec{a}$ and $\vec{b}$ are tuples containing the possibly captured values, as we explained in Sec. 2.1.4.1. Each of these axioms has the shape $\forall \ldots \bullet$ LHS $\Rightarrow$ RHS where LHS is the information required in order to obtain the new information from RHS.

The first four axioms are helpful in *induction proofs* in which the body term of a comprehension is independently computed for marginal values of the index. This typically happens in presence of programs with loops that use comprehensions to specify the loop invariant.

$$\begin{aligned}
&\forall lo, hi, \vec{a} \bullet lo < hi \wedge F(lo, \vec{a}) \implies \mathtt{s}(lo, hi, \vec{a}) = T(lo, \vec{a}) + \mathtt{s}(lo{+}1, hi, \vec{a}) \\
&\forall lo, hi, \vec{a} \bullet lo < hi \wedge \neg F(lo, \vec{a}) \implies \mathtt{s}(lo, hi, \vec{a}) = \mathtt{s}(lo{+}1, hi, \vec{a}) \\
&\forall lo, hi, \vec{a} \bullet lo < hi \wedge F(hi{-}1, \vec{a}) \implies \mathtt{s}(lo, hi, \vec{a}) = \mathtt{s}(lo, hi{-}1, \vec{a}) + T(hi{-}1, \vec{a}) \\
&\forall lo, hi, \vec{a} \bullet lo < hi \wedge \neg F(hi{-}1, \vec{a}) \implies \mathtt{s}(lo, hi, \vec{a}) = \mathtt{s}(lo, hi{-}1, \vec{a})
\end{aligned} \tag{Induct}$$

The fifth axiom enables splitting the comprehension range $(lo \ : \ hi)$ at an arbitrary index, *mid*. It permits deriving facts about the comprehension over a larger range based on known comprehensions over its adjacent subranges, and vice versa.

$$\forall lo, mid, hi, \vec{a} \bullet lo \leq mid \leq hi \implies \mathtt{s}(lo, mid, \vec{a}) + \mathtt{s}(mid, hi, \vec{a}) = \mathtt{s}(lo, hi, \vec{a}) \quad \text{(Split-Range)}$$

The sixth axiom allows one to derive equality between comprehension instances over the same range but with different instantiations of the holes in the filter and the body. For example, this axiom is useful for reasoning about array comprehensions across a heap-transforming operation; in this scenario, Spec# would encode the heap differently in the pre- and the post-states of the operation (see (2.1)), resulting in two different instantiations of the same comprehension template (2.6).

$$\begin{aligned}
\forall lo, hi, \vec{a}, \vec{b} \bullet \Big(\forall k \bullet lo \leq k < hi \Rightarrow F(k, \vec{a}) = F(k, \vec{b}) \wedge \Big(F(k, \vec{a}) \Rightarrow T(k, \vec{a}) = T(k, \vec{b})\Big)\Big) \\
\implies \mathtt{s}(lo, hi, \vec{a}) = \mathtt{s}(lo, hi, \vec{b})
\end{aligned}$$

(Same-Terms)

The seventh axiom is an evolution of (Same-Terms), adding the *distributivity* property of addition over the min and the max comprehensions. In particular, this axiom is designed for proving that **`min{int`** k **`in`** `(L:H); a[`k`]` + D`}` evaluates to the same value as **`min{int`** k **`in`** `(L:H); a[`k`]}` + D for any integer constant D. Note that the values of the min and the max comprehensions over the empty range, called the comprehension's *unit*, are generally problematic because, e. g. for the min comprehension, *all integers* must be less than or equal to the unit, so the unit itself cannot be an integer. Since Spec# uses

```
public static int minSegmentSum(int[] a)
  ensures result == min{int k in (0:a.Length); min{int j in (0:k);
                        sum{int i in (j:k); a[i] }}}
{
  int x = 0; int y = 0;
  for (int n = 0; n < n.Length; n++)
    invariant n <= a.Length;
    invariant x == min{int k in (0:n); min{int j in (0:k);
                       sum{int i in (j:k); a[i] }}};
    invariant y == min{int j in (0:n); sum{int i in (j:n); a[i] }};
  {
    y += a[n];
    if (0 < y) { y = 0; }
    else if (y < x) { x = y; }
  }
  return x; }
```

Figure 2.1: Example program and its comprehensive specifications written in Spec#.

finite integer arithmetic, the technique sets the units of min and max to be `MAX_INT` and `MIN_INT`, resp.; this approach cannot work in a setting with unbounded integers.

$$\begin{aligned}\forall lo, hi, D, \vec{a}, \vec{b} \bullet \Big(\forall k \bullet lo \le k < hi \Rightarrow F(k,\vec{a}) = F(k,\vec{b}) \wedge \Big(F(k,\vec{a}) \Rightarrow T(k,\vec{a}) + D = T(k,\vec{b})\Big)\Big) \wedge \\ \Big(\exists k \bullet lo \le k < hi \wedge F(k,\vec{a}) = F(k,\vec{b}) \wedge \Big(F(k,\vec{a}) \wedge T(k,\vec{a}) + D = T(k,\vec{b})\Big)\Big) \\ \Longrightarrow \mathtt{s}(lo, hi, \vec{a}) + D = \mathtt{s}(lo, hi, \vec{b})\end{aligned} \tag{Distrib}$$

The last axiom (Distrib) exposes a number of issues that were described in [52]. In particular, the quantified variable D does not occur as an argument of any uninterpreted function symbols, making it impossible for an E-matching-based SMT solver to trigger this axiom.

2.1.4.3 Running example. To illustrate some more complex comprehensive specifications supported by Spec#, consider the `minSegmentSum` program that computes the sum of the minimal array segment (Fig. 2.1).[5] For example, given the array $\mathtt{a} = [3, -2, 1, -2]$, the procedure returns -3, as this is the sum of `a`'s minimal segment, $[-2, 1, -2]$.

LIMITATIONS. One important limitation of the axiomatization schema in Spec# is that it relies on integer comprehension indices; in particular, this technique cannot support structures without a predefined total ordering over their nodes.

5 `minSegmentSum` is one of the benchmarks of the original Spec# comprehensions paper [43].

Second, the program logic of Spec# does not support concurrency as the encoding is inherently non-modular. Thus, the technique is not applicable in our setting with potentially concurrent programs.

Third, some of the axiomatized properties, e. g. (Distrib) cannot be used by modern SMT solvers like Z3 [47] that use the E-matching quantifier instantiation technique. Overcoming this issue is an open problem.

Finally, it is unknown which other potentially useful properties are missing in Spec#'s partial axiomatization. Generally, any inductive lemma that does not follow from the eight axioms above can be viewed as a missing property.

2.1.5 Open problems

In the remainder of this chapter, we will describe a technique that attacks two open problems that are not solved in the state-of-the-art comprehensions technique of Spec#.

MODULAR REASONING. We aim for automated verification of comprehensive specifications in a *modular* setting; to that end, we are interested in integrating into the framework of *separation logic* as it supports reasoning about concurrency and facilitates reusability of verified procedures. Conversely, Spec# comprehensions are not modular.

ARBITRARY NODESETS. Rather than exclusively targeting arrays and sequences, we aim to support comprehensions for *arbitrary structures* even those without a specific node ordering. Conversely, Spec# comprehensions do not support e. g. trees and DAGs.

2.2 SET COMPREHENSIONS

The comprehensions reasoning technique of Spec# discussed in Sec. 2.1.4 focuses on comprehensions over structures with totally ordered elements. In this section, we present a novel technique that addresses the more general problem of reasoning about *set comprehensions*, i. e. higher-order functions that summarize the values in *arbitrary* (unordered) sets of nodes.[6] While our comprehensions summarize values of node *sets*, these values do not have to be unique and generally form *multisets*. For example, $\sum_{x \in \texttt{xs}} x.\texttt{val}$ is a set comprehension that summarizes the set of nodes of `xs`, e. g. $\{\rho_0, \rho_1\}$, where ρ_0 and ρ_1 are some disjoint memory locations, while the values stored in these locations (e. g. $\rho_0.\texttt{val} = 42$ and $\rho_1.\texttt{val} = 42$) may form a multiset, e. g. $\{\!\{42 \rightarrowtail 2\}\!\}$.[7]

MOTIVATION. The motivation for the generalization discussed in this section is threefold. First, set comprehensions are suitable for specifying a *broader* spectrum of data structures, e. g. those without a total ordering over their nodes. Second, set comprehensions enable conceptually *simpler* decompositions of heap structures that typically lead to more concise proofs. Third, set comprehensions naturally integrate into separation logic, enabling verification of concurrent programs. We will demonstrate the first two advantages using the motivating example of Fig. 2.2; the third advantage will be explained later in Sec. 2.2.4.

OUTLINE. This section is organized as follows. We first discuss the motivation for our technique (Sec. 2.2.1). We then formally define set comprehensions (Sec. 2.2.2) and extend the specification language (Sec. 2.2.3). We explain the embedding of set comprehensions into separation logic (Sec. 2.2.4). Finally, we present our first-order axiomatization of set comprehensions (Sec. 2.2.5).

2.2.1 Motivating example

The comprehensions technique of Spec# enjoyed a conceptually simple encoding into first-order logic, but specifying *arbitrary* heap structures using this technique becomes challenging for the programmer. The specification is straightforward for array-manipulating — and, more generally, *tensor*-manipulating programs. However, to use the Spec# axioms with structures that do not enforce a total ordering, the programmer needs to specify and maintain additional sequences that represent (ordered) subsets of the nodes.

In the following, we will demonstrate the difference between the *sequence-based* and the *set-based* views on comprehensions, both of which are supported in our technique. We start with a high-level overview (Sec. 2.2.1.1), then introduce our specification (Sec. 2.2.1.2)

6 The initial design of this technique was developed by Hörmann [99] in his BSc. thesis.

7 We denote multiset literals via $\{\!\{\ldots\}\!\}$; each element is a pair $x \rightarrowtail N$ mapping x to its cardinality N.

```
field val: Int; field left: Ref; field right: Ref

method shortestPath(g: Set[Ref], node: Ref, target: Ref)
  returns (reachable: Bool, path: Seq[Ref], cost: Int)
  requires DAG(g, read) && node ∈ g ∧ target ∈ g
  ensures DAG(g, read)
          ∀x • x ∈ path ⟹ x ∈ g
          ∀i,j • 0 ≤ i < j < |path| ⟹ path[i] ≠ path[j]
          ¬reachable ⟹ path = Seq()
          reachable ⟹ 0 < |path| ∧ node = path[0] ∧ path[|path|-1] = target
  /*(6)*/ reachable ⟹ cost = 𝔰𝔲𝔪 n ⦃ n.val ‖ n ∈ path ⦄
  /*(7)*/ reachable ⟹ abs(cost) ≤ 𝔰𝔲𝔪 n ⦃ abs(n.val) ‖ n ∈ g ⦄
{
  if (node = target) {
    reachable, path, cost := true, Seq(node), node.val       // A
  } else {
    var reach_a, reach_b := false, false
    var path_a, path_b: Seq[Ref]
    var cost_a, cost_b: Int
    var g₁ := g setminus Set(node)

    label l₀
    if (node.left ≠ null) {
      reach_a, path_a, cost_a := shortestPath(g₁, node.left, target) }
    label l₁
    if (node.right ≠ null) {
      reach_b, path_b, cost_b := shortestPath(g₁, node.right, target) }
    label l₂
    // Return the aggregated results
    if (¬reach_a ∧ ¬reach_b) {
      reachable, path := false, Seq()
    } else {
      reachable := true
      if (reach_a ∧ ¬reach_b ∨ reach_a ∧ reach_b ∧ cost_a ≤ cost_b) {
        path := Seq(node) ++ path_a                          // B
        cost := node.val + cost_a                            // C
      } elseif (¬reach_a ∧ reach_b ∨ reach_a ∧ reach_b ∧ cost_b < cost_a)
        path := Seq(node) ++ path_b
        cost := node.val + cost_b
} } } }
```

Figure 2.2: Refining the specifications of complex data structures using set comprehensions.

The macro `DAG(g, read)` specifies access permissions to the fields `val`, `left`, and `right` of all objects within an acyclic heap fragment $\mathfrak{g}$ (see encoding details in Sec. 2.4.4). The postcondition marked `/*(6)*/` demonstrates the *sequence-based view* of a comprehensive property, specifying that the returned cost is equal to the sum of values stored in the `path` sequence. Conversely, `/*(7)*/` demonstrates the *set-based view*, specifying that the absolute value of the shortest path's cost must be less than or equal to the sum of absolute values of all the nodes' costs in the entire DAG represented by $\mathfrak{g}$; e. g. if there are no negative costs, this condition implies that the cost of the shortest path cannot exceed the total cost of the DAG. Our set comprehensions technique is the first to automatically verify this benchmark.

and verification (Sec. 2.2.1.3) techniques, and then discuss preliminary observations (Sec. 2.2.1.4).

2.2.1.1 high-level overview. The method `shortestPath` (Fig. 2.2) demonstrates the applicability of sequence-based comprehensions in the specification of a *shortest-path finding algorithm* operating over a DAG structure. The returned sequence `path` carries the main computation result; one could summarize the elements of this sequence (e. g. the overall cost of the shortest path) using a Spec#-style comprehension. However, it is harder to specify a custom comprehensive property, e. g. that (the absolute value of) the returned path's cost is *less that or equal to* (the absolute value of) the total cost of all the nodes in the entire DAG. To specify this property via Spec#-style comprehensions, the method would have to return an additional sequence containing all the reachable nodes (exactly once); such a sequence would need to be maintained via additional code which is specific to the custom property.

2.2.1.2 Specification. We propose a new kind of specifications, called *set comprehensions*, that enable *directly* specifying properties of arbitrary sets of nodes (within the method's footprint). We demonstrate these specifications based on the method `shortestPath` Fig. 2.2.

NOTATION. The expression $\mathfrak{sum}\, n{:}\mathtt{T}\ \{\!|\ \mathtt{body}(n)\ |\ \mathtt{filter}(n)\ |\!\}$ can be read as *the sum of* body terms *(`body(n)`) for all n that satisfy the* filter *(`filter(n)`)*, where `body` is a function $\mathtt{T} \to \mathbf{Int}$ and `filter` is a function $\mathtt{T} \to \mathbf{Bool}$. The part inside curly brackets is called the *structure of the comprehension*. We omit the type specification `:T` when it can be inferred from the structure.

For example, given a sequence `xs`, the expression $\mathfrak{sum}\, n{:}\mathbf{Int}\ \{\!|\ 1\ |\ 0 \le n < |\mathtt{xs}|\ |\!\}$ is equal to the length of `xs`, while for a node set `G` and an integer field `val`, the expression $\mathfrak{sum}\, n{:}\mathbf{Ref}\ \{\!|\ \mathtt{abs}(n.\mathtt{val})\ |\ n \in \mathtt{G}\ |\!\}$ is the sum of the absolute values in `G`.

Comprehensions specify properties of heap structures in particular states. To refer to the evaluation of a comprehension in a state σ, we use the subscript notation, e. g. $\mathfrak{sum}_\sigma\, n{:}\mathbf{Ref}\ \{\!|\ \mathtt{abs}(n.\mathtt{val})\ |\ n \in \mathtt{G}\ |\!\}$. To mark particular states in the program, we use state labels of the form $\mathbf{label}\ l_n$ where n is the integer identifier of that state. For convenience, we refer to the labeled states via their identifiers (rather than names). For example, the following code snippet demonstrates the usage of labeled states for specifying that an operation `oper` preserves a local comprehensive invariant:

$\mathbf{label}\ l_1\mathtt{;}\ \mathtt{oper();}\ \ \mathbf{assert}\ \mathfrak{sum}\, n\ \{\!|\ \mathtt{abs}(n.\mathtt{val})\ |\ n \in \mathtt{G}\ |\!\} = \mathfrak{sum}_1\, n\ \{\!|\ \mathtt{abs}(n.\mathtt{val})\ |\ n \in \mathtt{G}\ |\!\}$

Here, `oper()` is an operation that can modify the heap, and the state-dependent expression $\mathfrak{sum}_1\, n\{\!|\mathtt{abs}(n.\mathtt{val})|\ n \in \mathtt{G}\ |\!\}$ is rewritten into $\mathbf{old}[l_1](\mathfrak{sum}\, n\{\!|\mathtt{abs}(n.\mathtt{val})|\ n \in \mathtt{G}\ |\!\})$. Note that a comprehension without a subscript refers to the properties in the *current* state, and the subscript 0 always refers to the *initial* state of the current method, i. e. the state of its precondition.

SPECIFYING `shortestPath`. Recall that this method traverses a data structure in order to find, if it exists, the least costly heap path between two nodes (Fig. 2.2). The method takes as inputs a (ghost) set `g` containing the nodes of the data structure, as well as two references, `node` and `target`, which determine the start and the intended end of the path. The precondition requires (via `DAG(g, read)`) read permissions to the three relevant fields (`left`, `right`, and `val`) of each of the nodes in `g`, and that `node` and `target` are in `g`.

The outputs are a Boolean `reachable`, indicating the existence of a path from `node` to `target`, an auxiliary sequence `path` of nodes along the shortest path, and the computed cost of the shortest path; if such a path does not exist, `path` is set to the empty sequence and `cost` is left unspecified. The postcondition ensures: (1) the read permissions to the relevant fields of all nodes in `g` are not leaked; (2) the identified path may consist only of the nodes of the original structure represented by `g`; (3) the path is acyclic; (4) if `target` is unreachable from `node` then `path` is the empty sequence; the last three postconditions specify the property of an existent path, namely (5) a `path` consists of at least one node, with `node` and `target` as the first and the last sequence elements; (6) the cost of the path is equal to the sum of the values stored in the field `val` of all nodes of the path; and (7) the absolute value of the computed cost does not exceed the sum of absolute values stored in the field `val` of *all nodes in the current data structure* represented by `g`.

*2.2.1.3 **Verification.*** We proceed by introducing our proof outline notation. We will then demonstrate *reasoning with set comprehensions* by verifying the comprehensive part of `shortestPath`'s specifications, namely, bits (6) and (7).

PROOF OUTLINE NOTATION. We present the correctness arguments as *proof outlines* in the style of calculational proofs, i. e. derivation sequences of either values or conditions that we aim to justify. For each outline, we explain the assumptions under which it holds (e. g. induction hypothesis, loop guard, or branch condition) in its preceding paragraphs, and we justify each step of the outline in the subsequent paragraphs. Consider the following example outline:

Justification	Formula	
	$x = f(42) \;=$	(Outline-Name)
$x = f(42)$ follows from (123)		
Arithmetic	$f(40+2) \;=$	
Linearity of f; Def. y	$f(40)+f(2) \;=\; y+z$	

Each step of the outline occupies one row; we refer to the relevant facts justifying this step in the *left column*, while the derived formula is in the *right column*. Several minor steps may be inlined, occupying a single row, in which case the justification in the left column applies to all of the minor steps on this row. Our proof outlines often identify new types of reasoning steps that were not necessarily introduced before the outline;

we use unique symbols for each such step in the left column and explain these in the paragraphs after the outline. The end of a proof is marked with □

Why are proof outlines useful for presenting an automated verification technique? Although an SMT-based verifier does not necessarily *construct* proof outlines, automatic verification is predicated on the ability of the underlying SMT solver to (1) derive relevant lemmas from the supported theories and (2) instantiate first-order quantifiers. These are the two essential types of reasoning steps in our proof outlines. Hence, proof outlines provide intuition for understanding the essence of a verification problem.

OUTLINE 1: THEN-BRANCH. Entering the first branch of `shortestPath` under the condition `node = target`, we obtain the singleton `Seq(node)` for `path` and `node.val` for `cost`, which is sufficient to satisfy postcondition (6) via the following derivation:

	$\mathfrak{sum}\, n \,\{\!\{\, n.\texttt{val} \mid n \in \texttt{path} \,\}\!\} \;=$
`shortestPath-A`	$\mathfrak{sum}\, n \,\{\!\{\, n.\texttt{val} \mid n \in \texttt{Seq(node)} \,\}\!\} \;=$
†; `shortestPath-A`	$\texttt{node.val} \;=\; \texttt{cost}$

(Outline-1)

In the first step, we substitute `path` with its known value in the current branch, `Seq(node)`. In the second step (marked with †) we use the knowledge of the properties of a *singleton comprehension*, namely, that the sum over the value of a single node, indeed, yields the node's value. In the last step, we substitute the value `node.val` with the variable `cost`; this equality follows from the assignment of `shortestPath-A` in the current branch.

We continue exploring this branch, using the following derivation to justify postcondition (7):

Outline-1	$\texttt{abs(cost)} \;=\; \texttt{abs(node.val)} \;\leq$
Arith.	$\texttt{abs(node.val)} \;+\; \mathfrak{sum}\, n \,\{\!\{\, \texttt{abs}(n.\texttt{val}) \mid n \in \mathfrak{g} \setminus \texttt{Set(node)} \,\}\!\} \;=$
■	$\mathfrak{sum}\, n \,\{\!\{\, \texttt{abs}(n.\texttt{val}) \mid n \in \mathfrak{g} \,\}\!\}$

(Outline-2)

In the first step, we substitute `cost` with its value, `node.val`. The second step consists of two parts: We consider a comprehension over the filter $\mathfrak{g} \setminus$ `Set(node)` and then apply the following arithmetic lemma: *The sum of non-negative numbers is non-negative*; the incentive for this particular filter becomes clearer while reading the proof outline bottom-up: We aim to verify the lower bound of a comprehension over $\mathfrak{g}$ in which only `node` is modified. In the step ■, we select a *singleton decomposition* of the comprehension filter. □

OUTLINE 2: ELSE-BRANCH. Entering the second branch of `shortestPath` under the condition `node` $\neq$ `target`, we must consider the following three possible scenarios based on the intermediate results returned by the two recursive calls. First, if `target` is unreachable from both `node.left` and `node.right` ($\neg$`path_a` $\wedge$ $\neg$`path_b`), then `reachable` is set to false, trivially satisfying postconditions (6) and (7) in Fig. 2.2.

Otherwise, we learn that `target` is reachable from the current node, i. e. `reach_a` $\vee$ `reach_b` holds. The second and the third scenarios correspond to the two innermost branches of Fig. 2.2. The condition of the second scenario is that `path_b` either does not reach `target` at all ($\neg$`reach_b`) or it has a higher cost (`reach_b` $\wedge$ `cost_a` $\leq$ `cost_b`); in this case, we take `path_a` as the next node of `path`.

Since the third scenario is analogous to the second one, w.l.o.g. we present in the following a proof outline only for the second scenario. Note that the two innermost branches are exclusive, i. e. one could replace **`elseif...`** with **`else`**. The following derivation justifies postcondition (6), which is the first comprehensive property in question:

	$\mathfrak{sum}_2\, n \,\{\!\{\, n.\texttt{val} \mid n \in \texttt{path} \,\}\!\} \;=$
`shortestPath-B`	$\mathfrak{sum}_2\, n \,\{\!\{\, n.\texttt{val} \mid n \in \textbf{\textit{Seq}}\texttt{(node)++path\_a} \,\}\!\} \;=$
■	$\texttt{node.val} + \mathfrak{sum}_2\, n \,\{\!\{\, n.\texttt{val} \mid n \in \texttt{path\_a} \,\}\!\} \;=$
▣	$\texttt{node.val} + \mathfrak{sum}_1\, n \,\{\!\{\, n.\texttt{val} \mid n \in \texttt{path\_a} \,\}\!\} \;=$
Rec. 1	$\texttt{node.val} + \texttt{cost\_a} \;=$
`shortestPath-C`	`cost`

(Outline-3)

Here $\mathfrak{sum}_1$ and $\mathfrak{sum}_2$ denote the sum comprehension evaluated in the states l_1 and l_2, resp. In the first step, we substitute `path` with its value, ***`Seq`***`(node)++path_a` (`shortestPath-B`). In the step marked with ■, we select a singleton decomposition of the comprehension filter. The the step ▣ applies *separation-logic framing*, deriving the equality between the remaining comprehension in l_2 and l_1 (recall that `NODES(g,` **`read`**`)` specifies only read permissions). The last two steps apply the postcondition of the first recursive call and substitutes the value of `cost` according to `shortestPath-C`.

We proceed with the derivation for the second comprehensive property in question, namely, postcondition (7) of `shortestPath`:

	$\texttt{abs(cost)} \;=$
`shortestPath-C`	$\texttt{abs(node.val + cost\_a)} \;\leq$
Arith.	$\texttt{abs(node.val)} + \texttt{abs(cost\_a)} \;=$
Rec. 1	$\texttt{abs(node.val)} + \mathfrak{sum}_1\, n \,\{\!\!\{\, \texttt{abs}(n.\texttt{val}) \parallel n \in \mathfrak{g} \setminus \textbf{\textit{Set}}\texttt{(node)} \,\}\!\!\} \;=$
▣	$\texttt{abs(node.val)} + \mathfrak{sum}_2\, n \,\{\!\!\{\, \texttt{abs}(n.\texttt{val}) \parallel n \in \mathfrak{g} \setminus \textbf{\textit{Set}}\texttt{(node)} \,\}\!\!\} \;=$
▪	$\mathfrak{sum}_2\, n \,\{\!\!\{\, \texttt{abs}(n.\texttt{val}) \parallel n \in \mathfrak{g} \,\}\!\!\}$

(Outline-4)

In the first step of the above derivation, we substitute `cost` with its value (assigned on `shortestPath-C`). Second, we apply the *triangle inequality*: $\texttt{abs}(a{+}b) \leq \texttt{abs}(a){+}\texttt{abs}(b)$. Next, we apply the postcondition of the first recursive call, substituting `abs(cost_a)` with its upper bound in terms of the comprehension. We then apply separation-logic framing (▣), deriving the equality of the comprehension in states 1 and 2. The last step (▪) expresses the comprehension over our decomposition via the comprehension over the original set $\mathfrak{g}$. □

2.2.1.4 Discussion. The example of Fig. 2.2 demonstrates how set comprehensions enrich our specification language: While the penultimate postcondition can be specified in either sequence-based *or* set-based view, postcondition (7) expresses a comprehensive property of a DAG structure that cannot be phrased using sequence comprehensions.

Set comprehensions enable conceptually simple structural decompositions. In the sequence-based view, each operation fractured the array range into a *generally unknown number of segments*, e. g. a call to a procedure that swaps two elements indexed i and j of the array `a` results in up to *five* array segments in Spec# (Sec. 2.1.4): $[0,\ i)$, $[i,\ i)$, $[i{+}1,\ j)$, $[j,\ j)$, and $[j{+}1,\ \texttt{len(a)})$. In contrast, the specifications of `shortestPath` (Fig. 2.2) present an alternative view in which each operation partitions its client's footprint into *exactly two* heap fragments, i. e. the operation's footprint and its frame.

There are two main challenges of modular reasoning with set comprehensions. First, one needs to embed set comprehensions into a modular framework, e. g. separation logic. Second, one needs to automate the reasoning steps. Notice that the above proof outlines involved a number of recurring steps that were not yet explained; these were marked with †, ▪, and ▣. We will discuss each of these steps and their corresponding axiomatization in Sec. 2.2.5.

2.2.2 Formalization

2.2.2.1 Preliminaries. We begin by defining *node fields* $\mathcal{F} : \mathcal{I} \to \mathcal{T}$. Here, $\mathcal{I}$ are literal field names (cf. *idnt* in Fig. 1.1), e. g. `"val"` or `"next"`, and $\mathcal{T}$ are the field types (cf. *type* in Fig. 1.1), e. g. ***Int*** in **field** `val:` ***Int***. We assume that each program has a finite number of explicitly declared fields (i. e. the set $\mathcal{I}$ and the function $\mathcal{F}$ are always available).

Next, we define *program states* Σ as sets $\sigma = \left\{ \sigma^f \,\middle|\, f \in \mathcal{I} \right\}$ where $\sigma^f : \textbf{\textit{Ref}} \to \mathcal{F}(f)$ is the *projection of the state* σ over the field named f. For each field f in the program, the state σ defines the mapping from heap node references to values of the field's type.

Set comprehensions are members of the class **comp** of higher-order (scalar-valued) partial functions of the program state and one other functional argument (called the binary *step* operator $\oplus$). Intuitively, the step operator takes two values of some type, say S, and yields a result that is also of type S. We expect each step operator to respect the following properties:

$$\begin{aligned} &\oplus : \mathsf{S} \to \mathsf{S} \to \mathsf{S} \\ &\forall x, y \bullet x \oplus y = y \oplus x \\ &\forall x, y, z \bullet (x \oplus y) \oplus z = x \oplus (y \oplus z) \end{aligned} \tag{2.9}$$

Thus, $\mathcal{S} = (\mathsf{S}, \oplus)$ is a *commutative semigroup*. Some semigroups have a special *unit* element (denoted $\mathbb{1}$) that respects the following property:

$$\mathbb{1} : \mathsf{S}, \quad \forall x : \mathsf{S} \bullet \mathbb{1} \oplus x = x \tag{2.10}$$

If $\mathbb{1}$ is a unit of a semigroup $(\mathsf{S}, \oplus)$, then $(\mathsf{S}, \oplus, \mathbb{1})$ is a *monoid*. A monoid's unit is unique.

2.2.2.2 Main definitions. Rather than directly formalizing set comprehensions as partial functions, we define them in two stages, exclusively using total functions. The first stage introduces the notion of *comprehension kinds*, which are (total) higher-order functions. The second stage leverages this notion to define *comprehension instances*, i. e. scalar-typed expressions that define comprehensive properties.

COMPREHENSION KINDS. Fixing a commutative semigroup $\mathcal{S} = (\mathsf{S}, \oplus)$ as well as the name n and type T of an *iterated variable* defines a particular *comprehension kind*, denoted $\textbf{\textit{comp}}[\mathcal{S}]\, n{:}\mathsf{T}$:

$$\text{“}\textbf{\textit{comp}}[\mathcal{S}]\, n{:}\mathsf{T}\text{”} : \underbrace{(\Sigma \to \mathsf{T} \to F \to \mathsf{S})}_{\text{Body}} \to \underbrace{\textbf{\textit{Set}}[\mathsf{T}]}_{\text{Filter}} \to \mathsf{S} \tag{2.11}$$

The 1st functional argument of the comprehension is the *body*; this is a state-dependent expression over the (iterated) free variable n, mentioning the fields from the set F. Effec-

tively, the body maps program states, values of type T, and field names from some set $F \subseteq \mathcal{I}$ to values of type S, which is the same as the entire comprehension's result type.

The 2nd argument is the *filter* set; intuitively, the filter determines for which instantiations of n should the body term be evaluated. Note that we represent the filter as a set of accepted values of type T rather than a T → **Bool** function to simplify the notation for *updated filters*, that are used in the following.

For example, comprehensions of kind **comp**[(**Int**, *, 1)] n:**Ref** map references to integers, combine these integers via *, and have 1 as their unit. Instantiating the body term with n.val and the filter with xs (a set of nodes of a structure with the same name), we obtain a comprehension that represents the product of all values stored in the nodes of xs: **comp**[(**Int**, *, 1)] n:**Ref** {| n.val || xs |}. Note that (**Int**, *, 1) is a monoid.

To improve readability, we will write e. g. **comp**[*,1] ... for **comp**[(**Int**, *, 1)] ... since the semigroup's type is clear from the context.

COMPREHENSION INSTANCES. A *comprehension instance*[8] is an expression obtained by instantiating a particular comprehension kind (2.11) with a body and a filter (Fig. 2.3). Before a comprehension instance can be evaluated, we must check that, in the current state (in which we evaluate the body), there are sufficient permissions to evaluate the comprehension body for all instantiations of the iterated variable permitted by the filter.[9] For example, the evaluation of $\mathbf{comp}_\sigma$ n:**Ref** {| n.val || $n \in$ g |} from Fig. 2.2 requires read permissions to field val of all nodes in g. After checking permissions, we may treat the comprehension body as a *total function* as it is defined for all possible instantiations.

It is important to understand why filters are necessary. If $\mathcal{S}$ is a commutative monoid with unit $\mathbb{1}$, then we could rewrite any comprehension of the form **comp**[⊕, $\mathbb{1}$] n:T{| e(n)|| f |} using the *universe set* for the filter: **comp**[⊕, $\mathbb{1}$] n:T{| $n \in$ f ? e(n) : $\mathbb{1}$ || T |}. However, such a rewriting does not exist for comprehensions over general commutative semigroups, e. g. (**Int**, min), since the neutral element of min is the *non-integer* $+\infty$.

2.2.2.3 Evaluating set comprehensions. We will now define the evaluation of a set comprehension. Let $\mathbf{comp}_\sigma$[$\mathcal{S}$] n:T {| $\mathsf{e}^F_\sigma(n)$ || f |} be a comprehension over the term $\mathsf{e}^F_\sigma(n)$, where n is a free variable, f is the filter, and σ is the program state; we omit the type specification of n when it is clear from the context. Let $\mathcal{S}$ be a semigroup over the domain S with ⊕ as its step operator. The consistency assumption implies that ⊕ and $\mathsf{e}^F_\sigma(n)$ are of type S (as well as the unit $\mathbb{1}$, if $\mathcal{S}$ is a monoid).

8 We will use the terms *comprehension instance* and *comprehension* interchangeably.

9 For simplicity, we consider only heap-independent filters in our formalization; a heap-*dependent* filter can be obtained as a result of a function application, in which case additional permission checking is performed.

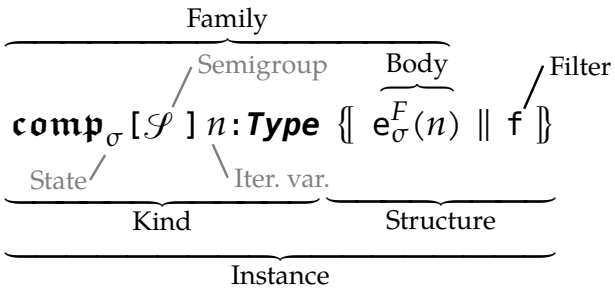


Figure 2.3: Anatomy of a set comprehension.

In a program state σ, the *comprehension instance* is the full instantiation of all the parameters of a comprehension: the *commutative semigroup* $\mathcal{S}$ (sometimes written as its operator and, in case of a monoid, unit), the name and the type of the *iterated variable* n, the *body* $\mathsf{e}^F(n)$, and the *filter set* $\mathtt{f}$; this is the necessary and sufficient information for evaluating the comprehension's value. Fixing all but the filter defines a *comprehension family*; the evaluation of a comprehension (2.12), as well as most of the axioms in our technique, are written in terms of one family. The comprehension's underlying *multiset structure* is defined by fixing its body and filter; this structure is the argument of the *comprehension kind* (2.11), e. g. $\mathbf{comp}[+,0]\,x{:}\mathbf{Ref}$ represents all instances of the $\mathbf{sum}$ comprehension, iterating over references, with $+$ and 0 as its step operator and unit, resp.

The *evaluation* of this comprehension is defined as a fixpoint, if it exists, of the following recursive equation:

$$\mathbf{comp}_\sigma[\mathcal{S}\,]\,n\,\{\!|\,\mathsf{e}^F_\sigma(n)\parallel \mathtt{f}\,|\!\} \;=\; \begin{cases} \mathcal{S}\equiv(\mathtt{S},\oplus,\mathbb{1})\;?\;\mathbb{1}\;:\text{undefined}, & \mathtt{f}=\emptyset \\ \mathsf{e}^F_\sigma(x), & \mathtt{f}=\mathtt{Set}(x) \\ \mathbf{comp}_\sigma[\mathcal{S}\,]\,n\,\{\!|\,\mathsf{e}^F_\sigma(n)\parallel \mathtt{f}_1\,|\!\} & \mathtt{f}=\mathtt{f}_1\uplus\mathtt{f}_2 \\ \quad\oplus\;\mathbf{comp}_\sigma[\mathcal{S}\,]\,n\,\{\!|\,\mathsf{e}^F_\sigma(n)\parallel \mathtt{f}_2\,|\!\}, & \text{where } \mathtt{f}_1,\mathtt{f}_2\neq\emptyset \end{cases} \tag{2.12}$$

The formula (2.12) provides three rules for evaluating a set comprehension. First, if the empty set is the filter (and $\mathcal{S}$ is a monoid), then the evaluation of the comprehension is equal to its unit. Second, if the filter is a singleton containing just the element x, then the evaluation of the comprehension is equal to the evaluation of its term with x for n, in the state σ. Third, for a decomposition of the original filter $\mathtt{f}$ into two disjoint, non-empty *sub-filters* $\mathtt{f}_1$ and $\mathtt{f}_2$, the evaluation of the original comprehension is equal to the $\oplus$-combination of the evaluations of two other comprehensions which are similar to the original one but for their respective sub-filters.

Set comprehensions specify data structures *regardless of their processing order*. For example, a method `mul` that uses two threads for concurrently multiplying the matrix `A` by the constant factor k may partition `A` either *row-wise* or *column-wise*. However, the thread specifications are *identical* in both cases; e. g. their postcondition can be set to $\mathbf{comp}[+,0]\,n\,\{\!|\,n.\mathtt{val}\parallel n\in\mathtt{nodes(X)}\,|\!\} \;=\; k * \mathbf{comp}_0[+,0]\,n\,\{\!|\,n.\mathtt{val}\parallel n\in\mathtt{nodes(X)}\,|\!\}$,

where `X` is this thread's sub-matrix (or just `A`, for `mul`'s postcondition); `nodes(X)` is a set of nodes of `X`, and the subscript 0 in **`comp`**$_0$ ... refers to `mul`'s initial state.

Coincidently, the formula (2.12) does not prescribe a single decomposition for the filter set. The variables x, $\mathtt{f}_1$, and $\mathtt{f}_2$ in the conditions of the 2^{nd} and the 3^{rd} cases are (implicitly) *existentially quantified*; hence, any (proper) two-way decomposition of the filter set `f` allows us to reduce the problem of evaluating the corresponding comprehension to the evaluation of at most two comprehensions with potentially smaller sub-filters.

The equation (2.12) has a fixpoint *iff* the filter `f` is a non-empty set or $\mathcal{S}$ is a monoid. Therefore, the evaluation of monoid-based comprehensions is a total function of filters. Conversely, the evaluation of comprehensions based on general semigroups is a partial function that is defined at least on all non-empty filters. Note that the fixpoint must be unique due to associativity and commutativity of the step operator (2.9).

2.2.3 Extending the specification language

We extend the grammar of our specification language (Fig. 1.1) to support set comprehensions. The augmented grammar is presented in Fig. 2.4. Set comprehensions extend pure expressions. *oper* denotes the step operator of the comprehension's semigroup and *unit* denotes the unit (in case of a monoid). Note that the prior can be instantiated either with one of the built-in binary operators or with a (binary) uninterpreted function symbol, denoted *customOper*. Analogously, the latter can be instantiated either with a built-in literal, denoted *literal* or with a (nullary) symbol, denoted *customUnit*.

The technique checks that the provided operator is commutative and associative (2.9). If the unit is provided, the technique also checks the properties of (2.10). For example, a comprehension instance of the form **`comp`**`[*,0]` n`:`***`Int`*** ... would not pass the check since any non-zero integer, say x, would violate the property x`*0` $= x$.

We omit from the grammar the ***`Bool`***-typed binary operators, e. g. $=$, $\neq$, $\wedge$, and $\vee$, as well as the units **`true`** and **`false`**, since they result in comprehensions that can be expressed using first-order quantification.

A comprehension's body and filter are both pure expressions. The former may mention the iterated variable and may capture by value any other variables available in the current scope. For example, the following is a valid comprehensive specification for a method called `foo`:

```
method foo(g: Set[Ref], x: Int)
  requires comp[+,0] n:Ref {| n.val = x ? 1 : 0 || g |}
  ensures  comp[+,0] n:Ref {| n.val = x ? 1 : 0 || n ∈ g |}
```

Note that the filter can be either a set-typed or a Boolean-typed expression. In the former case, the type of the set is expected to be ***`Set`***`[T]` where `T` is the type of the iterated variable (Fig. 2.3). In the latter case, the filter is a Boolean expression over the iterated variable. In particular, the precondition and the postcondition of the method `foo` above are equivalent.

pureExp	*literal* | *idnt* | *funApp* | *pureExp*`.`*idnt* | *pureExp*`[`*pureExp*`]` | *FOL*`(`*pureExp*`)` | *oldExp* | *setComp*
literal	**`null`** | **`true`** | **`false`** | **`result`** | Integer | SetLiteral | SeqLiteral | MapLiteral
idnt	Alpha-numeric identifiers, e. g. `x1`, `y2`
funApp	*idnt* `(`*args*`)`
args	*pureExp* `(, ` *pureExp*`)*`
FOL	First-order logical formulas over (built-in theories and uninterpreted function symbols)
oldExp	**`old`** `[`*idnt*`]`$^?$ `(` *pureExp* `)`
setComp	**`comp`**`[`*oper*`(,`*unit*`)`$^?$`]` *idnt*`(: ` *type*`)`$^?$ $\{\!\!\{$ *pureExp* $\|\|$ *pureExp* $\}\!\!\}$
oper	`+` | `*` | **`union`** | **`intersection`** | min | max | *customOper*
unit	*literal* | *customUnit*
type	***`Bool`*** | ***`Int`*** | ***`Ref`*** | ***`Seq`***`[`*type*`]` | ***`Set`***`[`*type*`]` | ***`Map`***`[`*type*`, `*type*`]`

Figure 2.4: Augmenting the grammar with set-comprehensive expressions.

MULTIDIMENSIONAL COMPREHENSIONS. To simplify the encoding in our technique, we permit only one iterated variable per comprehension instance. However, the programmer can overcome this limitation, either by *nesting* comprehension instances or by iterating over tuples. For example, the following expression represents the sum of all elements of a (non-empty) 2-dimensional matrix `m`:

$$\texttt{comp[+,0]}\, i\texttt{:}\mathbf{\mathit{Int}}\{\!\!\{\, \texttt{comp[+,0]}\, j\texttt{:}\mathbf{\mathit{Int}}\ \{\!\!\{\ \texttt{m[}i\texttt{][}j\texttt{]}\ \|\ \texttt{0} \le j < \texttt{len(m[0])}\ \}\!\!\} \|\ \texttt{0} \le i < \texttt{len(m)}\ \}\!\!\}$$

The same value can be represented via just one comprehension instance that iterates over pairs of integers:

$$\texttt{comp[+,0]}\, k\texttt{:Pair[}\mathbf{\mathit{Int}}\texttt{,}\mathbf{\mathit{Int}}\texttt{]}\ \{\!\!\{\ \texttt{m[}k_1\texttt{][}k_2\texttt{]}\ \|\ \texttt{0} \le k_1 < \texttt{len(m)} \wedge \texttt{0} \le k_2 < \texttt{len(m[0])}\ \}\!\!\}$$

Here, `Pair` is a (user-defined) algebraic data type and k is an instance of that type; k_1 and k_2 are the deconstructors that yield the 1st and the 2nd component of k, resp. The programmer may choose any of the two specification styles demonstrated above. However, automatically converting between these notations is beyond the scope of this thesis.

NAMED COMPREHENSIONS. We introduce the following notation for comprehensions over some frequently used semigroups (Fig. 2.3):

- **`sum`** is equivalent to **`comp`**`[+,0]`

- **prod** is equivalent to **comp**[*,1]
- **min** is equivalent to **comp**[min]
- **max** is equivalent to **comp**[max]

2.2.4 Encoding set comprehensions into separation logic

The main idea behind our encoding is to use a state-independent function that takes (1) the encoded representation of the body and (2) the filter set, yielding the comprehension's value. Recall that the body is a heap-dependent term; in a given state, we represent this term via a (stateless) mathematical object called *snapshot*.

For example, consider the array `xs` $= [\rho_0, \rho_1, \rho_2]$ of nodes ρ_i storing `3`, `4`, and `5`, resp. in their `val` fields and **`true`**, **`true`**, **`false`** in their `marked` fields. The snapshot of the comprehension **comp**`[+,0]` i`:`***Int*** $\{\!|$ `xs[`i`].marked ? xs[`i`].val : 0` $\|$ `0` $\leq i <$ `|xs|` $|\!\}$ is the map from array indices to the corresponding values of the body term: `0` $\rightarrow$ `3`, `1` $\rightarrow$ `4`, `2` $\rightarrow$ `0`.

In our encoding, we represent snapshots via instances of type ***Map***`[T,S]`, where `T` is the type of *index values*, e. g. the integer array indices in the above example, and `S` is the type of the comprehension.

2.2.4.1 Representing comprehensions via first-order functions. Consider the general form of a comprehension: **comp**$_\sigma$`[`$\mathcal{S}$`]` n`:T` $\{\!|$ $\mathsf{e}^F_\sigma(n)$ $\|$ `f` $|\!\}$; this notation was explained in Fig. 2.3. Assume that this is a consistent comprehension of type `S` (we will explain consistency checking in Sec. 2.2.4.7). We are searching for a viable first-order simulation of this higher-order comprehension. To that end, we follow the standard approach of modeling the (higher-order) comprehension via an uninterpreted function symbol and encoding its functional arguments via values of composite types, e. g. sets and maps.

To encode the above-mentioned kind, we introduce the *comprehension function* as a (hatted) uninterpreted function symbol, $\widehat{\textbf{comp}}$, representing our comprehension's value:

$$\widehat{\textbf{comp}} : \textbf{\textit{Map}}[\mathsf{T},\ \mathsf{S}] \rightarrow \textbf{\textit{Set}}[\mathsf{T}] \rightarrow \mathsf{S} \tag{2.13}$$

The signature of $\widehat{\textbf{comp}}$ resembles that in the definition of set comprehensions (2.11), except we left the program state unspecified and the functional argument is replaced with a first-class map. This design achieves two goals: On the one hand, we can decouple the state-dependent part of the encoding (represented by the first argument) from the state-*independent* encoding of the filter (Sec. 2.2.4.4); on the other hand, using an explicitly encoded map in the first argument of $\widehat{\textbf{comp}}$ (as opposed to a functional argument) allows us to write first-order axioms about this function symbol (Sec. 2.2.5).

2.2.4.2 Weakest filter. The first argument of $\widehat{\textbf{comp}}$ can be viewed as the comprehension structure (Fig. 2.3) with the *weakest possible filter* (i. e. the largest possible filter, if one

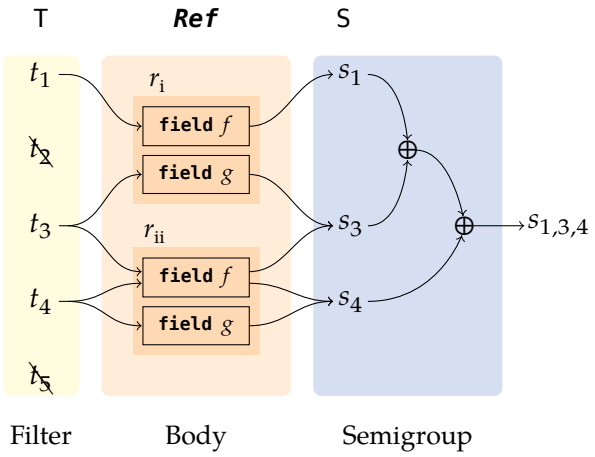


Figure 2.5: Model of a comprehensive computation.

Symbols represent values of the respective domains, as indicated at the top. Boxes represent heap objects. Arrows represent information flow. t-values are unique indices and r-values are unique node references; s-values may or may not be unique. Consider the comprehension instance `comp[Semigroup] t:T {| Body || Filter |}` where the semigroup is $(\mathsf{S}, \oplus)$, the body is $\mathsf{e}^F_\sigma(t)$ for some σ. The filter determines which `T`-values instantiate the body (t_1, t_3, t_4 in our model). The body is an expression that depends on the fields from the set F ($F = \{f, g\}$) of *multiple* heap nodes, as the body may traverse the heap to some limited depth (in our model, $\mathsf{e}^F_\sigma(t_1)$ and $\mathsf{e}^F_\sigma(t_4)$ depend on r_i and r_{ii}, resp., while $\mathsf{e}^F_\sigma(t_3)$ depends on both r_i and r_{ii}). However, an instantiation of the body yields a *single* value of type `S` ($t_1 \to s_1$, $t_3 \to s_3$, $t_4 \to s_4$). These values (s_1, s_3, s_4) are folded via a $\oplus$-combination into the resulting value ($s_{1,3,4}$).

represents the filter as a set). Intuitively, a comprehension's structure is a mapping from indices to values representing the properties of the individual nodes. In this view, the filter is a means of selecting *subsets* of the available nodes the properties of which are to be summarized. Thus, the weakest filter is precisely the set of indices (i. e. instantiations of the iterated variable) that lead to the available nodes (i. e. those the fields of which are accessible in the current state).

In separation logic, one can refer to only those memory locations that belong to the current footprint. Therefore, the weakest filter does not yield *all* the nodes on the heap, but only those that belong to the footprint. Concretely, the weakest filter permits only those instantiations of the iterated variable n of the comprehension body expression $\mathsf{e}^F_\sigma(n)$ that result in expressions that can be evaluated in the current state σ.

RUNNING EXAMPLE. Consider the family of comprehensions `comp[+,0] i:Int{|xs[i].val|| f |}`, summing the values in nodes of the array `xs` $= [\rho_0, \rho_1, \rho_2]$. First, we are interested in evaluating a comprehension of this family in a state (say, σ_1) with access permissions to

the entire array `xs`. Hence, the weakest filter is {`0, 1, 2`} as any array index from this set yields an instantiation of the body term `xs[`i`].val` that can be evaluated in this state.

Second, we are interested in evaluating a comprehension of the same family in a different state (say, σ_2) in which we have access permissions only to the subarray `xs[1..]` = $[\rho_1, \rho_2]$; this can happen in the specifications of e. g. a recursive procedure operating on a limited range of an array. Hence, the weakest filter is {`1, 2`} because, although index `0` represents some node in the original array `xs`, we do not have permissions to access the `val` field of this node that is requires for evaluating the comprehension's body term.

PURPOSE OF THE WEAKEST FILTER. Denoting the weakest filter through $N_\mathfrak{g}$, we conclude that the 1st argument of $\widehat{\texttt{comp}}$ must have the following shape:

$$\mathbb{R}(\mathsf{e}^F_\sigma) \;=\; \left\{\, (n : \mathtt{T}, v : \mathtt{S}) \;\middle|\; v = \mathsf{e}^F_\sigma(n) \wedge n \in N_\mathfrak{g} \,\right\} \tag{2.14}$$

Here, $\mathfrak{g}$ is the current *footprint*, i. e. the set of nodes to the fields $f \in F$ of which there are some access permissions. The map $\mathbb{R}(\mathsf{e}^F_\sigma)$ represents the multiset structure of the comprehension body `e` in the state σ (F is the set of fields through which `e` can access the heap memory). Since $N_\mathfrak{g}$ is the weakest filter, it contains precisely the indices through which one can access the nodes of our data structure within the current footprint.

Valid instantiations of the second argument of $\widehat{\texttt{comp}}$ (2.13) must be subsets of $N_\mathfrak{g}$; we will discuss the corresponding checks in Sec. 2.2.4.7. Recall that we allow the programmer to write arbitrary Boolean expressions (possibly referring to the iterated variable) for specifying comprehension filters. For example, **`comp`**`[`min`]` i`:`***`Int`***`{|xs[`i`].val| true |}` refers to the minimal value stored in the nodes of the *currently accessible* part(s) of the array `xs`; i. e. the filtering condition **`true`** is rewritten by our technique into $N_\mathfrak{g}$.

Revisiting our running example, assume that the nodes of `xs` store the values `5, 6, 7`, resp. in σ_1 and the values `10, 6, 7`, resp. in σ_2. Recall that in σ_1 there are permissions to access the entire array, and in σ_2 there are permissions to access only the subarray `xs[1..]`. Then, in σ_1 we have $\mathbb{R}$`(xs[`i`].val)` = {`(0, 5), (1, 6), (2, 7)`}, while in σ_2 we have $\mathbb{R}$`(xs[`i`].val)` = {`(1, 6), (2, 7)`}. This is precisely the information needed to evaluate e. g. **`comp`**`[+,0]` i`:`***`Int`*** {| `xs[`i`].val` | `f` |} in σ_1 and σ_2, for any *valid* filter `f`, e. g. {`0, 2`} for σ_1 and {`1, 2`} for σ_2.

ENCODING WEAKEST FILTERS. To express $N_\mathfrak{g}$ via the information available to the prover, we introduce the notion of the *lifted representation* of a heap structures. The lifted representation is a function `lift` : ***`Set`***`[`***`Ref`***`]` → ***`Map`***`[T, `***`Ref`***`]` of the current footprint, $\mathfrak{g}$, that yields a map from `T`-indices of our data structure to the actual nodes. We can now represent $N_\mathfrak{g}$ as the domain of `lift`:

$$N_\mathfrak{g} = \textbf{domain}(\texttt{lift}(\mathfrak{g})) \tag{2.15}$$

In our running example, $\mathfrak{g}_{\sigma_1} = \{\rho_0, \rho_1, \rho_2\}$ and $\mathfrak{g}_{\sigma_2} = \{\rho_1, \rho_2\}$. Hence, $\texttt{lift}(\mathfrak{g}_{\sigma_1})$ maps the indices of the original array (`xs`) to their corresponding memory locations: $\texttt{0} \to \rho_0$, $\texttt{1} \to \rho_1$, $\texttt{2} \to \rho_2$, while $\texttt{lift}(\mathfrak{g}_{\sigma_2})$ maps a subset of those indices: $\texttt{1} \to \rho_1$, $\texttt{2} \to \rho_2$.

DISCUSSION. The concept of weakest filters ensures that the semantics of set comprehensions is restricted to the current footprint, say $\mathfrak{g}$. As we will show in the following, this enables embedding set comprehensions into separation logic. However, the footprints in our setting can *change*, e. g. due to memory allocation operations. Our technique handles changing footprints by relating comprehensive information about different *heap decompositions*[10] using a number of first-order rules that will be discussed in Sec. 2.2.5.

To illustrate the relation between comprehensions in two states with different footprints, notice that the two comprehensions of our running example above are indeed related as they summarize the properties of *nested* subarrays (`xs` in σ_1 and `xs[1..]` in σ_2). Therefore, the knowledge about one of these comprehensions provides information about the other, e. g. $\texttt{comp}_{\sigma_1}\texttt{[+,0]}\, i\texttt{:}\mathbf{Int}\, \{\!|\, \texttt{xs[}i\texttt{].val} \,\|\, \{\texttt{0, 1, 2}\} \,|\!\} \;=\; \texttt{xs[0].val} + \texttt{comp}_{\sigma_2}\texttt{[+,0]}\, i\texttt{:}\mathbf{Int}\, \{\!|\, \texttt{xs[}i\texttt{].val} \,\|\, \{\texttt{1, 2}\} \,|\!\}$ follows from (2.12) due to $\{\texttt{0, 1, 2}\} = \{\texttt{0}\} \uplus \{\texttt{1, 2}\}$.

2.2.4.3 Lifted data structure representation. Recall that the node set $\mathfrak{g}$ is the *most-abstract* representation of the current footprint. In contrast, $\texttt{lift}(\mathfrak{g})$ lifts this representation to a slightly more concrete realm. While all heap structures, including arrays and graphs, have a conceptually equivalent node-based representation (namely, their footprint), the index-based representation of an array maps *integers* to nodes, while in graphs nodes are typically managed via direct references. Hence, the first case requires a version of `lift` that yields an injective $\mathbf{Int} \to \mathbf{Ref}$ mapping, while in the second case we simply use the identity function $\mathbf{Ref} \to \mathbf{Ref}$ (as each node is accessed directly via its reference).

The lifted representation incorporates the information about how particular nodes are accessed via their indices. However, this is still an abstract view over a data structure: `lift` does not depend on the values stored in the fields of the nodes. The following three axioms formalize the semantics of `lift`:

$$\begin{aligned}
&\forall \mathfrak{g}\texttt{:}\mathbf{Set}\texttt{[}\mathbf{Ref}\texttt{]}, n\texttt{:T} \bullet \texttt{indexOf}(\mathfrak{g}, \texttt{lift}(\mathfrak{g})[n]) = n \\
&\forall \mathfrak{g}\texttt{:}\mathbf{Set}\texttt{[}\mathbf{Ref}\texttt{]}, n\texttt{:T} \bullet n \in \mathbf{domain}(\texttt{lift}(\mathfrak{g})) \iff \texttt{lift}(\mathfrak{g})[n] \in \mathfrak{g} \\
&\forall \mathfrak{g}\texttt{:}\mathbf{Set}\texttt{[}\mathbf{Ref}\texttt{]} \bullet \mathbf{range}(\texttt{lift}(\mathfrak{g})) = \mathfrak{g}
\end{aligned} \tag{2.16}$$

The first axiom uses an auxiliary function, `indexOf`, ensuring that for a fixed $\mathfrak{g}$ the mapping $\texttt{lift}(\mathfrak{g})$ is injective. The second and the third axioms constrain the domain and the range of the mapping, resp. The former says that a node index n falls into the domain of the mapping *iff* it is mapped to a node in $\mathfrak{g}$; the latter — that the nodes of a structure accessible via the mapping comprise this structure's footprint; in other words, each node is accessible via some index.

10 Heap decompositions are typically isomorphic to *filter decompositions*.

2.2.4.4 Mapping data structures to their mathematical snapshots. The first step in our technique is to map a (stateful) data structure (or its part that the current method operates on) in a given program state to a mathematical, stateless object. Concretely, we are looking for a function that provides maps in the form of (2.14) that we could feed in as the 1st argument of our $\widehat{\texttt{comp}}$ function from (2.13). For this purpose, we employ a state-dependent function, `snap` (Fig. 2.6), that takes as an argument the current footprint $\mathfrak{g}$ (and a λ-argument, as will be explained below) and yields the map $\mathbb{R}$ (2.14), representing a fixed state of a heap structure.

```
function lift(g: Set[Ref]): Map[T, Ref] // axiomatized via (2.16)

function snap^F_σ(g: Set[Ref], λx · e^F_σ(x)): Map[T, S]
  requires ACCESS_NODES^F_σ(g)
  ensures domain(result) = domain(lift(g))
          ∀n: T • n ∈ domain(lift(g)) ⟹ result[n] = e^F_σ(n)
```

Figure 2.6: Mapping data structures to their mathematical snapshots.

The precondition of `snap` requires, for each node $n \in \mathfrak{g}$, the access permissions to read the fields from the set F. For example, given a comprehension $\mathbf{sum}\, n\texttt{:Ref}\{\!|\texttt{n.val * n.val}|\!|\, n \in \mathfrak{g}\,\}\!\}$, where $\mathfrak{g}$ is the current method's footprint and `val` is an integer field, we would instantiate the type parameters `T` and `S` with ***Ref*** and ***Int***, resp., and $F = \{\texttt{val}\}$. Hence, the precondition of `snap` would be translated to **forall** n • $n \in \mathfrak{g} \implies$ **acc**(n.val, **read**).

The postcondition of `snap` encodes the resulting value of this function, which is of the type ***Map***`[T,S]`, where `S` is the type of the comprehension body, $\mathsf{e}^F_\sigma(n)$. The keyword **result** represents the map `snap`$(\mathfrak{g}, \lambda x \cdot \mathsf{e}^F_\sigma(x))$. The first line of the postcondition specifies that the domain of this map is equal to the domain of the lifted representation, i. e. the indices. The second one specifies the *values* to which these indices are mapped.

The λ-argument of `snap` binds the comprehension's body s.t. it is available in `snap`'s postcondition. Internally, our technique defunctionalizes all the λ-arguments, generating separate versions of `snap` for each comprehension family (cf. Fig. 2.3).

EXAMPLE We demonstrate the encoding of `snap` for the comprehensions $\mathbf{comp}\, x\{\!|\, x\texttt{.val}\,|\!|\, \mathfrak{g}\,\}\!\}$ and $\mathbf{comp}\, x\,\{\!|\, x\texttt{.val+c}\,|\!|\, \mathfrak{g}\,\}\!\}$, where $\mathfrak{g}$ is the current footprint, `val` is an integer field, and `c` is a local variable. For these comprehensions, our technique generates two concrete versions of `snap` with fresh names, e. g.:

```
function snap_1σ(g: Set[Ref]): Map[Ref, Int]
  requires ACCESS_NODES^val_σ(g)
  ensures domain(result) = domain(lift(g))
          ∀x: T • x ∈ domain(lift(g)) ⟹ result[x] = x.val

function snap_2σ(g: Set[Ref], c: Int): Map[Ref, Int]
  requires ACCESS_NODES^val_σ(g)
  ensures domain(result) = domain(lift(g))
          ∀x: T • x ∈ domain(lift(g)) ⟹ result[x] = x.val+c
```

In this example, the iterated variable's type is ***Ref***, so the condition $x \in$ **domain**(`lift(g)`) can be written simply as $x \in$ `g` (since in this case `lift` is the identity function).

2.2.4.5 Selecting relevant fragments of data structures. To evaluate a set comprehension instance, the programmer has to define its *filter* (2.12), corresponding to the second argument of $\widehat{\texttt{comp}}$ (2.13). The filter is a set of possible values for instantiating the comprehension's iterated variable. Comprehensions are typically used for specifying the properties of the entire footprint, i. e. the fragment of a data structure that the current method operates on. In such scenarios, the filter should be set to the *weakest filter*, **domain**(`lift(g)`), where `g` is the current footprint. Recall that comprehension filters are subsets of the weakest filter, as explained in Sec. 2.2.4.1.

```
function filt^F_σ(g: Set[Ref], λx · P^F_σ(x)): Set[T]
  requires ACCESS_NODES^F_σ(g)
  ensures result ⊆ domain(lift(g))
          ∀n: T • n ∈ domain(lift(g)) ⟹ (n ∈ result ⟺ P^F_σ(n))
```

Figure 2.7: Selecting a data structure's fragment via a state-dependent filtering condition.

The programmer may decide to strengthen the filters in their comprehensive specifications, e. g. to independently specify the properties of multiple disjoint data structures or to specify exact fragments of a given data structure. For example, the filter of the comprehension **comp**`[+,0] x:Ref` $\{\!|\ x.\texttt{cost} \parallel x \in \texttt{g} \wedge x.\texttt{cost} > 0\ |\!\}$ is the set of references to nodes (in the current footprint `g`) whose `cost` is positive. This example shows that practically filters *may* depend on the program state.

To encode custom filters, one could employ a state-dependent function called `filt` (Fig. 2.7). For example, the filter for the above comprehension can be expressed as `filt(g,` $\lambda x \cdot$ `x.cost` > 0`)`. The signature of `filt` is similar to that of `snap`, except that the former yields a *set of values* of type `T` — the possible instantiations of a comprehension's iterated variable. The precondition of `filt` requires permissions to access precisely the nodes of the current footprint, `g`. The postcondition specifies two bits. First, `filt` yields a *subset* of the weakest filter. Second, this subset consists of the elements that satisfy the predicate P^F_σ, i. e. the filtering condition. Note that we generate a separate version of the `filt` function for each instantiation of the λ-argument occurring in the specifications.

EXAMPLE. We demonstrate the concrete encoding of `filt` for the above comprehension, **`comp`**`[+,0] x:`***`Ref`*** $\{\!\!\{\ x.\texttt{cost} \parallel x \in \mathfrak{g} \wedge x.\texttt{cost} > 0\ \}\!\!\}$, where $\mathfrak{g}$ is the current footprint and `cost` is an integer field (the encoding picks a fresh name for `filt_1`):

```
function filt_1_σ(g: Set[Ref]): Set[T]
  requires ACCESS_NODES^F_σ(g)
  ensures result ⊆ domain(lift(g))
          ∀x: T • x ∈ domain(lift(g)) ⟹ (x ∈ result ⟺ x.cost > 0)
```

Similar to the case of `snap`, the condition $x \in$ **`domain`**`(lift(g))` here can be written simply as $x \in \mathfrak{g}$ if the iterated variable's type is ***`Ref`***. Note that, due to the LHS in `filt`'s postcondition, the expression $x \in$ `filt`$(\mathfrak{g})$ is specified only for $x \in \mathfrak{g}$. Hence, the condition $x \in \mathit{GRAPH}$ from our comprehension's filter does not need to be included in the translation.

2.2.4.6 Encoding comprehensive specifications. The `snap` function introduces a layer of abstraction that helps separating two orthogonal concerns in our reasoning technique, namely, the inherent properties of the heap vs. the mathematical properties of comprehensions. However, this separation should not be exposed to the programmer. In particular, the language in which the programmer specifies the comprehensive properties of their programs should match their intuition. The baseline for an intuitive specification language is the mathematical notation introduced in Fig. 2.3. Internally, our technique translates this (higher-order) notation into one that is compatible with (first-order) separation logic with the following syntactic rewriting rule:

$$\begin{aligned}\textbf{comp}_\sigma[\mathscr{S}]\ n{:}\mathsf{T}\ \{\!\!\{\ \mathsf{e}^F_\sigma(n) \parallel P^F_\sigma(n)\ \}\!\!\} \twoheadrightarrow\ & \widehat{\textbf{comp}}\Big(\texttt{snap}^F_\sigma\left(\mathfrak{g},\ \lambda x \cdot\ \mathsf{e}^F_\sigma(x)\right),\\ & \texttt{filt}^F_\sigma\left(\mathfrak{g},\ \lambda x \cdot\ P^F_\sigma(x)\right)\Big)\end{aligned} \tag{2.17}$$

Recall the signature of $\widehat{\textbf{comp}}$ that was defined in (2.13); we generate separate versions of $\widehat{\textbf{comp}}$ for each comprehension semigroup (i. e. $\mathscr{S}$ in **`comp`**`[`$\mathscr{S}$`]`). This function takes two arguments; The 1st argument of this function is a map representing the current structure of the heap (yielded by `snap`). The 2nd argument is a filter (yielded by `filt`). The node set $\mathfrak{g}$ is the current method's footprint.[11]

Using the terminology of Fig. 2.3, our technique generates a separate version of `snap` for each comprehension body. Similarly, the technique generates a separate version of the (state-dependent) function `filt` for each filtering condition $P^F_\sigma(n)$.

11 We assume that the set representation of the current footprint ($\mathfrak{g}$) is always available in our encoding. This requirement could be dropped in a setting that supports *permission introspection*, a specification language feature providing permission amounts to particular fields held in a given state. For example, Viper supports permission introspection via the keyword **`perm`**, e. g. $\mathfrak{g} = \{n{:}$***`Ref`*** $\mid$ **`perm`**$(n.\texttt{val}) > 0\}$ for a single field `val`.

If all the comprehensive specifications are in terms of monoids, one can always rewrite them using exclusively the weakest filters. Under this assumption the rule (2.17) can be simplified as follows:

$$\mathbf{comp}_\sigma\,[\oplus, \mathbb{1}]\; n{:}\mathtt{T}\,\{\!|\, \mathsf{e}^F_\sigma(n) \| P^F_\sigma(n) \,|\!\}\twoheadrightarrow \widehat{\mathbf{comp}}\left(\mathtt{snap}^F_\sigma\left(\mathfrak{g},\ \lambda x\cdot\ P^F_\sigma(x)\ ?\ \mathsf{e}^F_\sigma(x) : \mathbb{1}\right),\ \mathfrak{g}\right) \quad (2.18)$$

2.2.4.7 Consistency checking. Our technique checks that all comprehension specifications in the program are *well-formed*. Comprehensive specifications are well-formed *iff* each of the comprehension instances is *consistent*. Consider the comprehension instance $\mathbf{comp}_\sigma\,[\mathscr{S}]\; n{:}\mathtt{T}\,\{\!|\, \mathsf{e}^F_\sigma(n) \| \mathtt{filter}^F_\sigma \,|\!\}$ of type `S` and its corresponding lifted representation, denoted $\mathtt{lift}(\mathfrak{g})$, where $\mathfrak{g}$ is the footprint in the state σ and F is a (possibly, empty) field set. This instance is well-formed *iff* all of the following conditions are satisfied:

1. **Semigroup.** If $\mathscr{S} = (\mathtt{S}, \oplus)$, then $\oplus$ satisfies (2.9). Otherwise, if $\mathscr{S} = (\mathtt{S}, \oplus, \mathbb{1})$, then $\oplus$ satisfies (2.9) and $\mathbb{1}$ satisfies (2.10).
2. **Filter.** If `filter` is of type ***Set***`[T]`, then $\mathtt{filter}^F_\sigma \subseteq \mathbf{domain}(\mathtt{lift}(\mathfrak{g}))$. Otherwise, if $\mathtt{filter}^F_\sigma \equiv P^F_\sigma(n)$ for some Boolean expression P, then the function $\mathtt{filt}(\mathfrak{g},\ \lambda x\cdot\ P^F_\sigma(x))$ is well-formed.[12]
3. **Body.** $\mathsf{e}^F_\sigma(x)$ is of type `S` and the function $\mathtt{snap}(\mathfrak{g},\ \lambda x\cdot\ \mathsf{e}^F_\sigma(x))$ is well-formed.[12]

Some remarks about the consistency checks are in order. First, the field set F is (as least) the *union* of the fields mentioned in the comprehension's body and its filter. Hence, it is possible to instantiate F with the entire field set of the current program, i. e. $\mathfrak{g} = \mathfrak{g}^{\mathscr{F}}_\sigma$ (cf. Sec. 2.2.2). Second, to type check a comprehension instance, one needs to check that the type of its semigroup matches the type of its body and that the type of its filter is either Boolean or ***Set***`[T]` where `T` is the type of its iterated variable. Third, the λ-arguments of `snap` and `filt` may or may not depend on the iterated variable. For instance, **comp**`[+,0]` n:***Ref*** $\{\!|$ `1` $\|$ **true** $|\!\}$ is a consistent comprehension representing the number of nodes in the current footprint.

2.2.5 Axiomatizing set comprehensions

Equipped with the function `snap`, we are now ready to present our axiomatization of set comprehensions of the form $\mathbf{comp}\,[\mathscr{S}]$, which we encode via the uninterpreted function symbol $\widehat{\mathbf{comp}}$ (2.13). We cannot axiomatize *all properties* of set comprehensions using first-order formulas, e. g. those that require inductive reasoning.[13] Nevertheless, it is possible to identify a small set of properties of set comprehensions that are sufficient for

12 We defined well-formedness requirements in Sec. 1.2.2.

13 One could axiomatize a limited number of inductive properties, e. g. *distributivity* of multiplication over summation: $\mathtt{d}* \sum_{x\in \mathtt{xs}} x.\mathtt{val} = \sum_{x\in \mathtt{xs}} \mathtt{d}* x.\mathtt{val}$. Yet, there are *infinitely many* different inductive properties, and it is not feasible to axiomatize all of them in first-order logic.

(a) Singleton — ■ (b) Subset — ■

Figure 2.8: Program heap (de)compositions as toy brick constructions.

Direct field updates and method calls are the two supported classes of heap-transforming operations. While the former modify singleton nodes at a time, they may indirectly affect the comprehensive properties of larger heap fragments to which they belong. In contrast, method calls affect arbitrary subsets of the set of currently accessible nodes. To illustrate singleton vs. subset compositions and decompositions, we use the constructions of toy bricks (a) and (b), resp., where the reddish bricks symbolize the footprint of the operation and the blueish ones — the remaining part of the heap, a.k.a. the frame.

the automatic modular verification of a number of benchmark examples. In particular, the benchmarks that we will use to demonstrate our technique require the consideration of various heap decompositions that are characteristic of a broad class of real-world heap-transforming programs.

We proceed as follows. First, we present an axiom for evaluating comprehensions over empty and singleton filters (Sec. 2.2.5.1). Second, we present an axiom for deriving equalities between comprehensions with a common filter (Sec. 2.2.5.2). Our third axiom supports singleton decompositions that occur in presence of individual field updates (Sec. 2.2.5.3). Conversely, our fourth axiom supports subset decompositions that occur in presence of individual method calls (Sec. 2.2.5.4). We then discuss the incompleteness of our axiomatization (Sec. 2.2.5.5) and demonstrate it in two example scenarios: a simple example (Sec. 2.2.5.6) and a more involved one (Sec. 2.2.5.7).

2.2.5.1 Handling the non-recursive cases. We start by considering the special cases in which the filter is an empty or a singleton set:

$$\begin{aligned} &\forall R\colon \mathbf{Map}[\mathsf{T},\mathsf{S}] \bullet \widehat{\mathfrak{comp}}(R,\emptyset) = \mathbb{1} \qquad \text{if } \widehat{\mathfrak{comp}} \text{ encodes } \mathfrak{comp}[\oplus,\mathbb{1}] \\ &\forall R\colon \mathbf{Map}[\mathsf{T},\mathsf{S}],\ a\colon\mathsf{T} \bullet a \in \mathbf{domain}(R) \implies \widehat{\mathfrak{comp}}(R,\mathbf{Set}(a)) = R[a] \end{aligned} \tag{2.19}$$

These two axioms directly corresponds to the 1st and the 2nd cases of (2.12). The first axiom is defined only for comprehensions over monoids. Most proof outlines can be organized in a way that avoids reasoning about comprehensions over empty filters. Conversely, the second axiom connects the value stored in the map R at a with the comprehension's value; this axiom is a crucial reasoning ingredient (marked ☮).

2.2.5.2 Equal comprehensions over two structures. A common reasoning step is to derive the equality between two set comprehensions based over heap structures that agree on a *subset* of their nodes. This pattern is particularly important for supporting separation logic framing as the frame of an operation can be represented in either of the states (i. e. before or after invoking the operation) while the comprehensive properties of the frame in these two states are the same. For example, recall that we applied separation-logic framing (marked with ▣) in our pen-and-paper proof outlines for `shortestPath` (Sec. 2.2.1). To automate this step, we introduce an axiom called *same-terms*:

$$\begin{aligned}\forall R,\, Q\colon \mathbf{Map}[\mathsf{T},\mathsf{S}],\, F\colon \mathbf{Set}[\mathsf{T}] \bullet \emptyset \subset F \subseteq \mathbf{domain}(R), \mathbf{domain}(Q) \wedge \\ (\forall n\colon \mathsf{T} \bullet n \in F \Rightarrow R[n] = Q[n]) \implies \\ \widehat{\mathbf{comp}}(R, F) = \widehat{\mathbf{comp}}(Q, F)\end{aligned} \tag{2.20}$$

The LHS of the outer implication of this axiom checks that F is a valid filter for either of the maps, R and Q, and says that these two maps must assign the same `S`-values to all (filtered) `T`-values; in other words, the two comprehensions must agree on the value of their terms over the filter F. Under this condition, we obtain the RHS that simply expresses the equality of the two comprehensions. This axiom is analogous to its namesake axiom (Same-Terms) proposed by Leino and Monahan [52] for Spec# comprehensions. However, since our encoding of set comprehensions relies upon first-class filter sets, (2.20) must *quantify* over filters (F) in addition to the snapshots of the two structures (R, Q).

Another important use case for *same-terms* proving the equality between comprehensions with the same filter over *nested snapshots*, i. e. if the domain of one snapshot is a subset of the domain of the other and the two snapshots agree over that subset. For example, it is often necessary to prove $\widehat{\mathbf{comp}}(\mathtt{snap}(A), A) = \widehat{\mathbf{comp}}(\mathtt{snap}(B), A)$ where $A \subseteq B$ and there is a bijection between references and indices (which is not always the case, as depicted in Fig. 2.5, but is the case e. g. in arrays), In this case, the two maps must agree over the entire filter A, satisfying the LHS of (2.20).

Note that the empty set is *excluded* from the possible values of F in (2.20). This is because a comprehension over an empty filter is generally undefined; cf. (2.12). Conversely, F may be R or Q's weakest filter (2.15) as long as it is *valid* for comprehensions over both maps, i. e. identical to or the subset of the domain of these maps.

2.2.5.3 Stepping through induction proofs. The second common pattern concerns the derivation of comprehensive properties of a structure based on its *singleton decomposition*, i. e. the concrete properties of one node of this structure and the comprehensive properties of all the remaining nodes. For example, recall the steps marked with ▪ in

the pen-and-paper proof outlines for `shortestPath`. Our technique automates this step using the axiom called *step*:

$$\forall R\colon \mathbf{Map}[\mathsf{T},\mathsf{S}],\ F\colon \mathbf{Set}[\mathsf{T}],\ n\colon \mathsf{T} \bullet \mathbf{Set}(n) \subset F \subseteq \mathbf{domain}(R) \implies \widehat{\mathbf{comp}}(R,F) = R[n] \oplus \widehat{\mathbf{comp}}(R, F \setminus \mathbf{Set}(n)) \tag{2.21}$$

The LHS says that n belongs to the (valid) filter F, which contains also some other elements. The RHS equates the comprehension over filter F and a $\oplus$-combination of the mapped value of n and the comprehension over the remainder filter, $F \setminus \mathbf{Set}(n)$ (which must be non-empty due to the LHS). There are four analogues of this axiom in the Spec# technique (Induct). For our set comprehensions, we require only one axiom to cover all possible singleton decompositions. Note that this axiom specifies only the case $n \in F$; the case of $n \notin F$ does not require a separate axiom since it trivially follows that $F = F \setminus \mathbf{Set}(n)$, and hence $\widehat{\mathbf{comp}}(R,F) = \widehat{\mathbf{comp}}(R, F \setminus \mathbf{Set}(n))$.

2.2.5.4 Supporting modular reasoning. It is important for a separation-logic based technique to support *modular reasoning*, i. e. the ability to derive the comprehensive properties of the entire client's footprint based on those of the frame and the footprint of the callee. Method calls split the client's heap into two potentially *unbounded* subheaps, introducing a conceptually different kind of decompositions (marked ◪) that cannot be reduced to a bounded number of singleton decompositions and are hence not covered by (2.21). Therefore, we add one more axiom, called *split-term*, to our set comprehension reasoning technique, enabling arbitrary two-way decompositions:

$$\forall R\colon \mathbf{Map}[\mathsf{T},\mathsf{S}],\ F_1,\ F_2\colon \mathbf{Set}[\mathsf{T}] \bullet \emptyset \subset F_1, F_2 \subset \mathbf{domain}(R) \implies \widehat{\mathbf{comp}}(R, F_1 \uplus F_2) = \widehat{\mathbf{comp}}(R,F_1) \oplus \widehat{\mathbf{comp}}(R,F_2) \tag{2.22}$$

The LHS checks that both F_1 and F_2 are valid filters of the comprehension over the map R. The RHS equates the comprehension over the composition $F_1 \uplus F_2$ and the $\oplus$-combination of the comprehensions over the two respective subfilters. This axiom is analogous to the *split-range* axiom from the Spec# technique (Split-Range). However, an important advantage of set-based comprehensions is that the components F_1 and F_2 of the decomposition $F_1 \uplus F_2$ do not have to represent adjacent parts of the original heap structure, whereas a split of an integer range always reduces it to two comprehensions over *adjacent* sub-ranges.

2.2.5.5 Incompleteness of axioms. Axiomatizing the properties of set comprehensions in first-order logic cannot be complete. The evaluation of a comprehension is defined via the recursive formula (2.12) with a potentially unbounded number of steps. There-

fore, deriving first-order properties from this definition (e. g. (2.20), (2.21), and (2.22)) requires *induction*, which is itself beyond first-order logic.

While inherently incomplete, our axioms are designed to automate the reasoning steps that frequently occur while verifying comprehensive properties of heap-transforming programs. On the one hand, our axioms support the two common operations that are used in such programs, namely, individual field updates and subprocedure calls. On the other hand, our axiomatization generalizes the state of the art, i. e. the axioms of Spec# (that are also inherently incomplete).

In practice, incompleteness of our technique also stems from the SMT solver's limited support for *quantifier instantiation*, as this problem is generally undecidable. However, our experiments show that the axiomatization presented above still provides good automation on a diverse set of characteristic benchmarks, as we will explain in Sec. 2.5.

2.2.5.6 Applying singleton decompositions. Consider the simple program `flipOne` with an alternative, set-based specification (Fig. 2.9). The goal of this example is to demonstrate that set comprehensions require conceptually simpler structure decompositions than Spec#-style comprehensions. In particular, we sketch the following proof outline to verify the assertion in `flipOne`:[14]

$$
\begin{aligned}
& & & \mathbf{min}\, n \,\{\!|\, n.\mathtt{val} \,|\!|\, n \in \mathtt{g} \,|\!|\}\, = \\
& (2.17) & & \widehat{\mathbf{min}}\,(\mathtt{snap(g)},\, \mathtt{g})\, = \\
& (2.19)\ \text{☮} : (2.21)\ \blacksquare & & \mathtt{g} = \mathbf{Set}(\mathtt{x})\ ?\ \mathtt{snap(g)[x]}\ :\ min(\mathtt{x.val},\ \widehat{\mathbf{min}}\,(\mathtt{snap(g)},\, \mathtt{g} \setminus \mathbf{Set}(\mathtt{x}))\,)\, = \\
& \mathtt{snap} : (2.20)\ \blacksquare & & \mathtt{g} = \mathbf{Set}(\mathtt{x})\ ?\ \mathtt{x.val}\ :\ min(\mathtt{x.val},\ \widehat{\mathbf{min}}\,(\mathbf{old}(\mathtt{snap(g)}),\, \mathtt{g} \setminus \mathbf{Set}(\mathtt{x}))\,)\, = \\
& \text{Assign.} & & \mathtt{g} = \mathbf{Set}(\mathtt{x})\ ?\ \mathtt{-m}\ :\ min(-\mathtt{m},\ \underbrace{\widehat{\mathbf{min}}\,(\mathbf{old}(\mathtt{snap(g)}),\, \mathtt{g} \setminus \mathbf{Set}(\mathtt{x}))}_{C})\, = \\
& \text{Outline-6} & & -\mathtt{m}
\end{aligned}
$$

(Outline-5)

In the first step of the derivation above, we rewrite the comprehension instance using the internal ingredients of our technique. We then consider a *singleton decomposition*, based on the node `x` that has been modified and the unchanged heap fragment, $\mathtt{g} \setminus \mathbf{Set}(\mathtt{x})$. We denote singleton decompositions via ■; see 2.8a. The abbreviation $\mathtt{snap(g)}$ denotes $\mathtt{snap}^{\{\mathtt{val}\}}(\mathtt{g},\, \lambda x \cdot\ x.\mathtt{val})$. If `x` is the only node in the footprint `g`, then $\mathtt{snap(g)}$ at `x` (i. e. `x.val`) *is* the minimal value. Otherwise, we apply separation-logic framing ▣, for the nodes from this unchanged fragment, to learn that the individual values of $n.\mathtt{val}$ have not changed after the field update operation. We then substitute `x.val` with the RHS of the assignment: $-\mathtt{m}$.

14 min denotes the binary operator defined as $min(x, y) \equiv x \le y\ ?\ x : y$. $\mathbf{min}$ denotes $\mathbf{comp}[min, +\infty]$, and $\widehat{\mathbf{min}}$ is its corresponding comprehension function (2.13).

```
field val: Int
method flipOne(g:Set[Ref], x:Ref, m:Int)
  requires forall n • n ∈ g ⇒ acc(n.val)
  requires x ∈ g
  requires 0 ≤ m = min n {| n.val | n ∈ g |}
{
  x.val := -1 * m
  assert min n {| n.val | n ∈ g |} = -m
}
```

Figure 2.9: Simple heap-transforming program with comprehensive specifications.

To prove that the resulting expression is equal to $-\mathtt{m}$, consider the following lemma:

Arith.	$-\mathtt{m} \leq 0 \leq \mathtt{m} \;=$
Precond.	$\mathbf{old}(\mathbf{min}\, n\, \{\!\mid n.\mathtt{val} \mid \mathtt{g} \mid\!\}) \;=$
(2.17)	$\mathbf{old}(\widehat{\mathbf{min}}(\mathtt{snap}(\mathtt{g}), \mathtt{g})) \;=$
Distrib. **old**	$\widehat{\mathbf{min}}(\mathbf{old}(\mathtt{snap}(\mathtt{g})), \mathtt{g}) \;=$
(2.21) ■	$min(\mathbf{old}(\mathtt{x.val}), C) \;\leq$
Arith.	C

(Outline-6)

On the one hand, the precondition of `flipOne` requires that `m` is non-negative; hence, $-\mathtt{m}$ must be a non-*positive*. On the other hand, the same precondition also specifies that `m` is the value of the **min** comprehension over the entire footprint `g` *in the pre-state of the method* `flipOne`. Applying distributivity of **old**, we obtain a comprehension with a new term, $\mathbf{old}(n.\mathtt{val})$, over the entire `g`. We then consider a singleton decomposition of `g`, again, based on the updated node `x` (the decomposition exists since we assumed $\mathtt{g} \neq \mathbf{Set}(\mathtt{x})$). The remainder comprehension, denoted C, has the filter $\mathtt{g} \setminus \mathbf{Set}(\mathtt{x})$; this is the instance that we have obtained at the end of our original derivation (Outline-5). Finally, we apply the definition of min, obtaining C as the upper bound for $-\mathtt{m}$. □

2.2.5.7 Applying complex decompositions. We have observed so far that the singleton decomposition pattern supported in our set comprehension technique enables concise proofs for programs that manipulate individual heap nodes. Yet, modular reasoning in separation logic relies also on the ability of our technique to reason about comprehensive properties of arbitrary *nested subheaps*, e. g. converting between the comprehensive properties of the footprints of a callee method and its client. We will now address this problem, demonstrating that our axiomatization of set comprehensions is general

```
field left, right: Ref

define count(g, f) comp[+,0] n {| n.f = null ? 1 : 0 || n ∈ g |}

method merge(g: Set[Ref], l, r: Ref, ldag, rdag: Set[Ref])
  returns (link: Ref)
  requires DAG(g) && l ∈ ldag ∧ r ∈ rdag
           g = ldag ⊎ rdag ∧ CLOSED_g(ldag)
  ensures DAG(g) && link ∈ ldag
          old(l.right) = null ⟹ l = link ∧ l.right = r
          count(g, right) = old(count(g, right)) − 1
{
  if (l.right ≠ null) {
    var nldag: Set[Ref] := subIn(g, ldag, l.right)
    var g1: Set[Ref] := nldag union rdag
    link := merge(l.right, r, g1, nldag, rdag)
  } else {
    l.right := r
    link := l
} }
```

Figure 2.10: Example recursive program and its modular comprehensive specifications.

Method `merge` attaches the DAG rooted in `r` to a node of the DAG rooted in `l`, and returns that node. The macros `DAG(g)` and `CLOSED_g(ldag)` will be defined in Chap. 3; intuitively, the former specifies acyclicity while the latter — that each heap edge *exiting* `ldag` ⊂ `g` (if it exists) cannot point to its compliment, `rdag` ⊂ `g`.

enough for supporting such scenarios. Therefore, we proceed by considering the recursive method `merge` (Fig. 2.10) specified in terms of set comprehensions.

We are interested in verifying the last bit in the postcondition of `merge`, expressing a comprehensive property of the DAG structure that this method operates on. Concretely, this postcondition says that, as a result of invoking this method, the number of nodes in its footprint, `g`, whose `right` field stores the value of **`null`** decreases by one. Intuitively, the property should hold as the method creates exactly one new heap edge and does not destroy any (Fig. 2.11 shows a typical run of `merge`). To verify this comprehensive property, we consider the two branches in the body of `merge`, starting from the then-branch and then proceeding to the else-branch.

VERIFYING THE THEN-BRANCH. We begin by assuming the branch condition `l.right` ≠ **`null`**. There are three operations in this branch. First, we declare a subheap `nldag`, representing the *new left-DAG* via a state-dependent function `subIn`. `subIn(g, ldag, l.right)` yields a subset of the left-DAG, `ldag`, reachable from the node `l.right` by following reference fields of nodes within `g`.[15] Second, we declare a new footprint, `g1`, as the union

15 We will discuss the notion of *local reachability* in Chap. 3. In particular, the logical encoding of `subIn` is presented in Sec. 3.3.3.

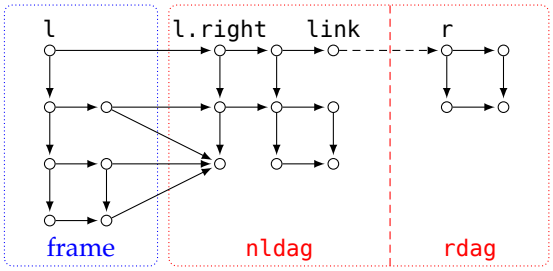


Figure 2.11: Example scenario of running `merge`.

The input structures are two DAGs rooted in `l` and `r`. Small circles correspond to heap objects; solid arrows represent fields initialized in the pre-state that are unchanged; the dashed arrow represents the new heap edge (created in the post-state by initializing a field). The frame of the recursive call is surrounded with blue; the footprint is surrounded with red.

of the new left-DAG `nldag` and the right-DAG `rdag`; note that (1) `g1` is a *subset* of the current footprint, `g`, as both components of the union are subsets of `g`, and (2) these components are *disjoint*, as `nldag` is a subset of `ldag` which is specified in the precondition to be disjoint with `rdag`. Third, we use the new footprint `g1` to invoke the callee instance of `merge`.

We proceed with the derivation for the value of `count(g, right)`:[16]

$$
\begin{array}{lll}
 & \texttt{count(g,right)} & = \\
\text{Syntax} & \mathbf{comp}\texttt{[+,0]}\ n\ \{\!|\ n.f = \mathbf{null}\ ?\ 1 : 0 \parallel n \in \mathtt{g}\ |\!\} & = \\
(2.17) & \widehat{\mathbf{comp}}\texttt{(snap(g),g)} & = \\
(2.22) & \widehat{\mathbf{comp}}\texttt{(snap(g),g \ g1)} + \widehat{\mathbf{comp}}\texttt{(snap(g),g1)} & = \\
(2.20) & \widehat{\mathbf{comp}}\texttt{(snap(g),g \ g1)} + \widehat{\mathbf{comp}}\texttt{(snap(g1),g1)} & = \\
\text{Recurse.} & \widehat{\mathbf{comp}}\texttt{(snap(g),g \ g1)} + \mathbf{old}(\widehat{\mathbf{comp}}\texttt{(snap(g1),g1)}) - 1 & = \\
(2.20) & \widehat{\mathbf{comp}}\texttt{(snap(g),g \ g1)} + \mathbf{old}(\widehat{\mathbf{comp}}\texttt{(snap(g),g1)}) - 1 & = \\
(2.20) & \mathbf{old}(\widehat{\mathbf{comp}}\texttt{(snap(g),g \ g1)}) + \mathbf{old}(\widehat{\mathbf{comp}}\texttt{(snap(g),g1)}) - 1 & = \\
(2.22) & \mathbf{old}(\widehat{\mathbf{comp}}\texttt{(snap(g),g)}) - 1 & = \\
(2.17) & \mathbf{old}(\mathbf{comp}\texttt{[+,0]}\ n\ \{\!|\ n.f = \mathbf{null}\ ?\ 1 : 0 \parallel n \in \mathtt{g}\ |\!\}) - 1 & = \\
\text{Syntax} & \mathbf{old}\texttt{(count(g,right))} - 1 &
\end{array}
$$

(Outline-7)

In the first step of the derivation, we rewrite the macro definition `count` according to the definition from Fig. 2.10, obtaining the comprehension instance in its canonical form. We then apply our encoding, which results in a (state-independent) function, $\widehat{\mathbf{comp}}$ over

16 The lower bounds in (2.22) and (2.20) are satisfied because neither $\mathtt{g}_1$ nor $\mathtt{g} \setminus \mathtt{g}_1$ are empty since they include `l.right` and `l`, resp. The upper bounds hold due to the specification of `snap`.

two arguments: a state-dependent map, `snap(g)`, and the current footprint, $\mathfrak{g}$, as the filter. Here `snap(g)` denotes $\texttt{snap}^{\{\texttt{left},\,\texttt{right}\}}(\mathfrak{g},\ \lambda x \cdot\ x.f = \textbf{null}\,?\,1:0)$.

Recall that the symbol ◪ denotes *arbitrary two-way decompositions*; see 2.8a. The third step involves such a decomposition, splitting the entire footprint $\mathfrak{g}$ into two disjoint parts based on the callee's footprint, $\mathfrak{g}_1$, and its frame, $\mathfrak{g} \setminus \mathfrak{g}_1$. We justify this step by instantiating our *split-term* axiom (2.22). Next, we apply the callee's postcondition, deriving the value of $\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}), \mathfrak{g}_1)$ in terms of the pre-state of the call. The equivalence of $\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}), \mathfrak{g}_1)$ and $\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}_1), \mathfrak{g}_1)$ follows from our *same-terms* axiom (2.20) because the maps `snap(g)` and $\texttt{snap}(\mathfrak{g}_1)$ agree over the domain $\mathfrak{g}_1$ (since $\mathfrak{g}_1 \subset \mathfrak{g}$).

We now apply our *same-terms* (2.20) again to derive that the comprehension over the frame has been preserved after the call. We then apply *split-term* (2.22) for the second time, composing a single comprehension in the pre-state over a larger filter, $\mathfrak{g}$, based on the two comprehensions over its disjoint subheaps. Finally, we conclude the derivation by rewriting the resulting comprehension via its canonical form, and then via the macro definition introduced by the programmer. □

VERIFYING THE ELSE-BRANCH. Proceeding to the else-branch of `merge`, we can justify the postcondition with quite a similar derivation sequence. The main difference here is that we apply our *step* axiom (2.21) while considering a singleton decomposition of the client's footprint $\mathfrak{g}$ based on the only node `l` modified in this branch. The following outline concludes our proof of the last postcondition in Fig. 2.10[merge]:[17]

	$\texttt{count}(\mathfrak{g}, \texttt{right}) \ =$
Syntax	$\textbf{comp}[+,0]\ n\ \{\!\!\{\ n.f = \textbf{null}\,?\,1:0 \parallel n \in \mathfrak{g}\ \}\!\!\} \ =$
(2.17)	$\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g}) \ =$
(2.21) ◪	$\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g} \setminus \textbf{\textit{Set}}(\texttt{l})) + \texttt{snap}(\mathfrak{g})[\texttt{l}] \ =$
Else-branch	$\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g} \setminus \textbf{\textit{Set}}(\texttt{l})) + 0 \ =$
(2.20) ▣	$\textbf{old}(\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g} \setminus \textbf{\textit{Set}}(\texttt{l}))) \ =$
Assign.	$\textbf{old}(\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g})) - \textbf{old}(\texttt{snap}(\mathfrak{g})[\texttt{l}]) \ =$
(2.21) ◪	$\textbf{old}(\widehat{\textbf{comp}}(\texttt{snap}(\mathfrak{g}),\ \mathfrak{g})) - 1 \ =$
(2.17)	$\textbf{old}(\textbf{comp}[+,0]\ n\ \{\!\!\{\ n.f = \textbf{null}\,?\,1:0 \parallel n \in \mathfrak{g}\ \}\!\!\}) - 1 \ =$
Syntax	$\textbf{old}(\texttt{count}(\mathfrak{g}, \texttt{right})) - 1$

(Outline-8)

As before, `snap(g)` denotes $\texttt{snap}^{\{\texttt{left},\,\texttt{right}\}}(\mathfrak{g},\ \lambda x \cdot\ x.f = \textbf{null}\,?\,1:0)$. In the final state, the evaluation of our comprehension over the singleton set $\textbf{\textit{Set}}(\texttt{l})$ yields the value 0; `snap(g)` holds the information that in this branch `l.right` has been assigned to a *non-null* value, namely, `r`. Conversely, the comprehension over $\textbf{\textit{Set}}(\texttt{l})$ yields 1 in the old

17 The premises of (2.21) and (2.20) are satisfied because $\textbf{\textit{Set}}(\texttt{l}) \subset \mathfrak{g}$ as the footprint also includes e. g. `r`.

state. As in the previous case, we apply our *same-terms* axiom (2.20) to prove that the comprehension over the frame, `g \ Set(l)`, is preserved across the state change. □

2.3 CASE STUDY

We have demonstrated in Sec. 2.2.5[18] that our technique is suitable for modular reasoning. In this section, we consider another dimension of complexity that arises in practice. Concretely, we revisit the running examples of prior work and explain how they can be specified and verified in our technique. We start by considering `minSubArraySum`, a classic algorithm for computing the sum of a minimal sub-array of an array; this case will demonstrate how our technique can handle *nested comprehensions* that are often useful in optimization problems (Sec. 2.3.1). We then consider `coincidenceCount`, a classic algorithm for computing the number of coinciding values in two arrays; this case demonstrates comprehensive specifications of a method that simultaneously operates on two disjoint data structures (Sec. 2.3.2).

2.3.1 Reasoning with nested comprehensions

2.3.1.1 Overview. The method `minSubArraySum` (Fig. 2.12) operates on a heap-allocated array whose footprint is specified via the node set `g`. The precondition of `minSubArraySum` requires that the array is non-empty. The postcondition ensures that the resulting value (stored in the output parameter `x`) is equal to either the sum of the minimal (non-empty) subarray or `0`, if there are no better solutions.

We first provide intuition for the algorithm and its specifications and then demonstrate the complete proof outlines.

ALGORITHM. To understand how the algorithm works, consider the scenario presented in Fig. 2.13. In this scenario, the input array is $[1, -2, -3, 3, -5, -1, 9, 2]$. The algorithm starts with `n`, `y`, `x` all set to `0`. Each loop iteration updates the three variables as follows. First, `y` gets the best[19] of `0` and $\texttt{y}_0\texttt{+array[n}_0\texttt{]}$. Second, `x` gets the best of `y` and $\texttt{x}_0$. Third, `n` is incremented. After each iteration, `y` stores the *locally best* solution and `x` stores the *globally best* solution over the processed array range.

SPECIFICATIONS. It is essential to understand why the specifications of Fig. 2.12 cannot be easily simplified. Verifying the ultimate postcondition relies on the verification of the two comprehensive properties in the loop invariant. The property marked `/*X*/` relies on the verification of `/*Y*/` as `x` depends on `y` in the loop body; conversely, `y` does not

18 In particular, Sec. 2.2.5.4 presents the axioms for reasoning about arbitrary two-way decompositions.
19 Here, *the best* refers to *the least* as we are considering a minimization problem.

```
field val: Int
define array lift(g)
define size(array) |domain(array)|
define sum(i, j) 𝔰𝔲𝔪 k ⦇ array[k].val ∥ i ≤ k < j ⦈

method minSubArraySum(g: Set[Ref]) returns (x: Int)
  requires ACCESS_NODES(g, read)
           size(array) > 0
  ensures ACCESS_NODES(g, read)
          x = min (0, min j ⦇ min i ⦇ sum(i,j) ∥ 0 ≤ i < j ⦈ ∥ 1 ≤ j < size(array)+1 ⦈)
{
  var n, y := 0, 0; x := 0
  while (n < size(array))
    invariant ACCESS_NODES(g, read)
              0 ≤ n ≤ size(array)
              n = 0 ⟹ y = 0 ∧ x = 0
      /* Y */ n > 0 ⟹ y = min (0, min i ⦇ sum(i,n) ∥ 0 ≤ i < n ⦈)
      /* X */ n > 0 ⟹ x = min (0, min j ⦇ min i ⦇ sum(i,j) ∥ 0 ≤ i < j ⦈ ∥ 1 ≤ j < n+1 ⦈)
  {
    var x₀, y₀, n₀ := x, y, n  // for the proof outlines
    y := y + array[n].val
    if (0 ≤ y) y := 0
    elseif (y < x) x := y
    n := n+1
} }
```

Figure 2.12: Implementation of `minSubArraySum` and its comprehensive specification.

The algorithm finds the cumulative value of the minimal sub-array of the input array. Each iteration updates `y` and then `x`; the former stores locally-optimal solutions (the sum of the minimal sub-array *ending* in `n`), and the latter stores the globally-optimal solution so far (the sum of the minimal sub-array ending *before* `n`). The special case in which all the array elements are positive (for which the best solution corresponds to the *empty* sub-array) is covered by $min(0,\ldots)$ in the invariant. The maximal value of i in `Y` is `n-1` because `n` is *the next node* to be processed. The maximal value of j in `X` is `n` as it determines the current sub-array's *excluded* upper bound.

Iteration	Array value	Local Best	Global Best	Comment
n = 0	array[n] = 1	y = 0	x = 0	
n = 1	array[n] = -2	y = -2	x = -2	
n = 2	array[n] = -3	y = -5	x = -5	
n = 3	array[n] = 3	y = -2	x = -5	x ← min(x₀, y)
n = 4	array[n] = -5	y = -7	x = -7	
n = 5	array[n] = -1	y = -8	x = -8	
n = 6	array[n] = 9	y = 0	x = -8	y ← min(0, y₀+array[n])
n = 7	array[n] = -2	y = -2	x = -8	

Figure 2.13: Typical run of `minSubArraySum`.

depend on x, so the property /*Y*/ can be verified independently. Note that (for $n > 0$) this property must be of the form $y = min(0, y_0 + array[n_0])$ as the algorithm may choose 0 over any (previously computed) local best; the 6th iteration in Fig. 2.13 shows such a situation.

We express property /*Y*/ via comprehensions (for $n > 0$ and some α):

$$y = min(0, \min_{\alpha \le i < n} sum(i, n))$$

Since α defines the lower bound of i, i. e. the leftmost array index, we conclude that $\alpha = 0$. Note that α must be *less than* n as otherwise the min-comprehension range would be empty (recall that min over $\emptyset$ is undefined).

We now express the dependent property /*X*/ (for $n > 0$ and some β):

$$x = min(0, \min_{\beta \le j < n+1} \min_{0 \le i < j} sum(i, j))$$

We will now justify that the above expression, indeed, represents x after each loop iteration. If $n = 1$, then we have just processed the first array element, so x must be the best of 0 and array[0]. We get $j \in [\beta, 2)$ and $i \in [0, j)$, i. e. the maximal value for j is 1 which gives the only possible value for $i = 0$. (The minimal value for j is also $\beta = 1$; otherwise, we would get an empty range in the inner min-comprehension.) Hence, the above expression for x yields the best of two: 0 and $sum(0, 1) = array[0]$, as expected.

We proceed under the alternative assumption, $n > 1$. On the one hand, substituting n with n_0+1 (due to the increment in the loop body) and $\beta = 1$ (we established this constant at $n = 1$), we can rewrite x using (2.21) as follows:

$$x = min(0, min(\min_{1 \le j < n_0+1} \min_{0 \le i < j} sum(i, j), \min_{0 \le i < n_0+1} sum(i, n_0+1)))$$

On the other hand, we have $x = min(x_0, y)$ where $y = \min_{0 \le i < n_0+1} sum(i, n_0+1)$ — from the above argument — and $x_0 = min(0, \min_{1 \le j < n_0+1} \min_{0 \le i < j} sum(i, j))$ — the hypothesis. Equating the two expressions for x and rewriting e. g. $min(a, min(b, c))$ as $min(a, b, c)$, we get the ultimate correctness condition:

$$\overbrace{min(0, \min_{1 \le j < n_0+1} \min_{0 \le i < j} sum(i, j), \min_{0 \le i < n_0+1} sum(i, n_0+1))}^{x}$$
$$= min(0, \underbrace{\min_{1 \le j < n_0+1} \min_{0 \le i < j} sum(i, j)}_{x_0}, \underbrace{\min_{0 \le i < n_0+1} sum(i, n_0+1)}_{y})$$

This condition is trivially valid. □

Note that one could refine the specifications of `minSubArraySum`, hiding the explicit $min(0, \ldots)$ cases. The idea is to use alternative invariants for `/*Y*/` and `/*X*/` that include empty ranges for the $\mathfrak{sum}$-comprehension:

$$\texttt{y} = \mathbf{min}\, i \;\{\!|\; \texttt{sum}(i, \texttt{n}) \parallel 0 \leq i \leq \texttt{n} \;|\!\}$$
$$\texttt{x} = \mathbf{min}\, j \;\{\!|\; \mathbf{min}\, i \;\{\!|\; \texttt{sum}(i, j) \parallel 0 \leq i \leq j \;|\!\} \parallel 1 \leq j < \texttt{n+1} \;|\!\}$$

In the above formulas, the ranges of `sum` can be empty. In the first expression, $\texttt{sum}(i, \texttt{n}) = 0$ if $i = \texttt{n}$. In the second expression, $\texttt{sum}(i, j) = 0$ if $i = j$. Arguably, this more concise specification version is harder to understand than the more verbose version of Fig. 2.12.

2.3.1.2 Encoding the nested comprehensions. Our technique automatically encodes comprehensions into separation logic. We will now demonstrate how this encoding can handle nested comprehensions. The encoding of a comprehension consists of three essential functions: `lift` that maps the node-based representation of a heap structure to an index-based representation (cf. Fig. 2.5), `snap` that encodes the evaluation of the comprehension body for all permitted indices, and `filt` that generates subsets of indices based on the conditions specified by the programmer.

The example of Fig. 2.12 features three comprehension families (cf. Fig. 2.3). As these comprehensions ultimately specify the same array structure, all three require only the standard `lift` function axiomatized via (2.16); we abbreviate `lift(g)` as `array` for readability. The `filt` function is also shared: All three comprehensions use filters that are defined as bounded sets of integer values: $\texttt{filt(lo, hi)} := \{k \mid \texttt{lo} \leq k < \texttt{hi}\}$. Here `lo` and `hi` are a semi-open interval's (included) lower and (excluded) upper bounds.

In contrast, the three comprehensions require different versions of the `snap` function; indeed, nested comprehensions cannot share their body terms. Fig. 2.14 presents the encoding of the three snapshot functions. Applying these functions (as well as `filt`), we obtain the following encodings, as per (2.17):

$$\texttt{sum}(i,j) \;\equiv\; \mathfrak{sum}\, k \;\{\!|\; \texttt{array}[k].\texttt{val} \parallel i \leq k < j \;|\!\} \quad \twoheadrightarrow \quad \widehat{\mathfrak{sum}}\,(\texttt{snap1}(\mathfrak{g}), \texttt{filt}(i,j)) \tag{2.23}$$

$$\mathbf{min}\, i \;\{\!|\; \texttt{sum}(i,j) \parallel 0 \leq i < j \;|\!\} \quad \twoheadrightarrow \quad \widehat{\mathbf{min}}\,(\texttt{snap2}(\mathfrak{g}, j), \texttt{filt}(0,j)) \tag{2.24}$$

$$\mathbf{min}\, j \;\{\!|\; \mathbf{min}\, i \;\{\!|\; \texttt{sum}(i,j) \parallel 0 \leq i < j \;|\!\} \parallel 1 \leq j < \texttt{n+1} \;|\!\} \quad \twoheadrightarrow \quad \widehat{\mathbf{min}}\,(\texttt{snap3}(\mathfrak{g}), \texttt{filt}(1, \texttt{n+1})) \tag{2.25}$$

2.3.1.3 Verification of the `Y` invariant. The first step is to verify the invariant denoted `Y` in `minSubArraySum`. This invariant expresses the fact that `y` is a *local optimum*, i. e. it stores the sum of the minimal *postfix* of the sub-array processed so far, or `0`, if all those sums happen to be positive. While we present the proof only for the case of $\texttt{n} > 0$, the

```
function snap1(g: Set[Ref]): Map[Int, Int]
  requires ACCESS_NODES(g)
           range(array) = g
  ensures domain(result) = filt(0, size(array))
          ∀k ∈ domain(result) • result[k] = array[k].val

function snap2(g: Set[Ref], j: Int): Map[Int, Int]
  requires ACCESS_NODES(g)
           range(array) = g
  ensures domain(result) = filt(0, size(array))
          ∀i ∈ domain(result) • result[i] = sum(snap1(g), filt(i,j))
                                            sum(i,j)
function snap3(g: Set[Ref]): Map[Int, Int]
  requires ACCESS_NODES(g)
           range(array) = g
  ensures domain(result) = filt(1, size(array)+1)
          ∀j ∈ domain(result) • result[j] = min(snap2(g,j), filt(0,j))
```

Figure 2.14: Snapshot functions encoding nested comprehension bodies from Fig. 2.12.

case of `n = 0` is identical, except that all the variables are initialized with `0`. Note that the computation of `y` does not depend on the computation of `x`; hence, we can ignore the latter variable for now. We proceed with the following proof outline written in terms of the *penultimate state* of the loop body, i. e. right before incrementing `n`:

Update y	$\texttt{y} \;=\; \mathit{min}\,(\texttt{0},\, \texttt{y}_0 + \texttt{array}[\texttt{n}_0].\texttt{val}) \;=$
(2.19) ☮	$\mathit{min}\,(\texttt{0},\, \texttt{y}_0 + \texttt{sum}(\texttt{n}_0,\, \texttt{n}_0{+}\texttt{1})) \;=$
Spec. snap2	$\mathit{min}\,(\texttt{0},\, \texttt{y}_0 + \texttt{snap2}(\texttt{g},\texttt{n}_0{+}\texttt{1})\,[\texttt{n}_0]) \;=$
Def. R	$\mathit{min}\,(\texttt{0},\, \texttt{y}_0 + \texttt{R}[\texttt{n}_0]) \;=$
Ind. hyp. Y	$\mathit{min}\,(\texttt{0},\, \mathit{min}\,(\texttt{0},\, \mathbf{min}\, i\, \{\!\!\{\, \texttt{sum}(i,\texttt{n}) \parallel \texttt{0} \le i < \texttt{n} \,\}\!\!\}) + \texttt{R}[\texttt{n}_0]) \;=$
(2.24)	$\mathit{min}\,\big(\texttt{0},\, \mathit{min}\,\big(\texttt{0},\, \widehat{\mathbf{min}}(\texttt{snap2}(\texttt{g},\texttt{n}_0),\, \texttt{filt}(\texttt{0},\texttt{n}_0))\big) + \texttt{R}[\texttt{n}_0]\big) \;=$
Def. $\texttt{R}_0, \texttt{F}_0$	$\mathit{min}\,\big(\texttt{0},\, \mathit{min}\,\big(\texttt{0},\, \widehat{\mathbf{min}}(\texttt{R}_0,\, \texttt{F}_0)\big) + \texttt{R}[\texttt{n}_0]\big) \;=$
Symm. *min*	$\mathit{min}\,\big(\texttt{0},\, \mathit{min}\,\big(\widehat{\mathbf{min}}(\texttt{R}_0,\, \texttt{F}_0),\, \texttt{0}\big) + \texttt{R}[\texttt{n}_0]\big) \;=$
Distrib. *min*	$\mathit{min}\,\big(\texttt{0},\, \mathit{min}\,\big(\widehat{\mathbf{min}}(\texttt{R}_0,\, \texttt{F}_0) + \texttt{R}[\texttt{n}_0],\, \texttt{R}[\texttt{n}_0]\big)\big) \;=$
Distrib. **min**	$\mathit{min}\,\big(\texttt{0},\, \mathit{min}\,\big(\widehat{\mathbf{min}}(\texttt{R},\, \texttt{F}_0),\, \texttt{R}[\texttt{n}_0]\big)\big) \;=$
(2.21) ■	$\mathit{min}\,\big(\texttt{0},\, \widehat{\mathbf{min}}(\texttt{R},\, \texttt{F})\big) \;=$
Def. R, F	$\mathit{min}\,\big(\texttt{0},\, \widehat{\mathbf{min}}(\texttt{snap2}(\texttt{g},\texttt{n}_0{+}\texttt{1}),\, \texttt{filt}(\texttt{0},\texttt{n}_0{+}\texttt{1}))\big) \;=$
(2.24)	$\mathit{min}\,(\texttt{0},\, \mathbf{min}\, i\, \{\!\!\{\, \texttt{sum}(i,\texttt{n}_0{+}\texttt{1}) \parallel \texttt{0} \le i < \texttt{n}_0{+}\texttt{1} \,\}\!\!\}) \;=$
Incr. counter	$\mathit{min}\,(\texttt{0},\, \mathbf{min}\, i\, \{\!\!\{\, \texttt{sum}(i,\texttt{n}) \parallel \texttt{0} \le i < \texttt{n} \,\}\!\!\})$

(Outline-9)

Here, the subscripted symbols $\texttt{y}_0$ and $\texttt{n}_0$ denote the values of the respective variables at the *beginning* of the current iteration; e. g. the value $\texttt{y}_0$ is known from the loop invariant. The symbols $\texttt{F}_0 \equiv \texttt{filt}(\texttt{0},\texttt{n}_0)$ and $\texttt{F} \equiv \texttt{filt}(\texttt{0},\texttt{n}_0{+}\texttt{1})$ denote the *filters* of the (nested) **min** comprehension at the beginning and at the end of the current iteration, resp. Similarly, $\texttt{R}_0 \equiv \texttt{snap2}(\texttt{g},\texttt{n}_0)$ and $\texttt{R} \equiv \texttt{snap2}(\texttt{g},\texttt{n}_0{+}\texttt{1})$ are the *snapshots*, encoding the (nested) **min**'s body at the current iteration's beginning and its end, resp.

The very first step follows from the operational semantics of the assignments to y in Fig. 2.12. The second step replaces the value stored in the array under $\texttt{n}_0$ with the singleton sum comprehension. We then use the information from the postcondition of snap2, replacing $\texttt{sum}(\texttt{n}_0, \texttt{n})$ with the corresponding value stored at index $\texttt{n}_0$ in the snapshot R; this lookup can be justified as $\texttt{n}_0 \in \textbf{domain}(\texttt{R})$ follows from the loop invariant $\texttt{0} \le \texttt{n}_0 < \texttt{size}(\texttt{array})$ (recall that $\textbf{domain}(\texttt{R}) = \texttt{filt}(\texttt{0}, \texttt{size}(\texttt{array}))$).

We proceed by applying the *induction hypothesis* of Y, substituting $\texttt{y}_0$ with its value in terms of the **min** comprehension; we then translate this comprehension into separation logic, as per (2.17).

The following steps require knowledge from the theory of **min**: (1) *symmetry* (e. g. $\mathit{min}(a,b) = \mathit{min}(b,a)$); (2) *distributivity of addition over min* (e. g. $\mathit{min}(a,b) + c = \mathit{min}(a+c, b+c)$), and (3) *distributivity of addition over the* **min** *comprehension* (e. g. $\mathbf{min}\, i\, \{\!\!\{\, a_i \parallel \texttt{f} \,\}\!\!\} + c = \mathbf{min}\, i\, \{\!\!\{\, a_i + c \parallel \texttt{f} \,\}\!\!\}$). While the first two of these properties are *linear integer arithmetic*

(which is handled by SMT solvers completely), our technique's translation of the last property contains the uninterpreted function $\widehat{\mathbf{min}}$ that must be axiomatized. The missing distributivity property is conceptually the same as (Distrib) used by Spec#; however, the latter is problematic, as explained at the end of Sec. 2.1.4.2. We assume for now that this property *is* available to the prover; we will then demonstrate the existence of a viable encoding in Sec. 2.4.6.

We conclude the proof by considering a singleton composition ($\mathtt{F} = \mathtt{F}_0 \uplus \mathbf{Set}(\mathtt{n}_0)$). Finally, we translate the resulting comprehension back to the specification language and take into account the increment of the loop counter `n`, obtaining the target value for `y` at the end of the iteration. □

2.3.1.4 Verification of the `X` invariant. We are now prepared to verify the ultimate invariant of `minSubArraySum`, denoted `X` in the code. As before: (1) we assume $\mathtt{n} > \mathtt{0}$ (the case $\mathtt{n} = \mathtt{0}$ is straightforward); (2) $\mathtt{y}_0$ and $\mathtt{n}_0$ as well as $\mathtt{x}_0$ denote the values of the respective variables at the beginning of the current iteration; (3) we proceed with a proof outline written in terms of the loop body's *penultimate state*, i. e. right before incrementing `n`:

Update `x`	$\mathtt{x} \;=\; min\,(\mathtt{x}_0,\, \mathtt{y}) \;=$
Invariant `Y`	$min\,(\mathtt{x}_0,\, min\,(\mathtt{0},\, \mathbf{min}\, i\, \{\!\!\{\, \mathtt{sum}(i, \mathtt{n}_0\mathtt{+1}) \parallel \mathtt{0} \le i < \mathtt{n}_0\mathtt{+1} \,\}\!\!\}\,)) \;=$
Ind. hyp. `X`	$min\Big(min\,(\mathtt{0},\, \mathbf{min}\, j\, \{\!\!\{\, \mathbf{min}\, i\, \{\!\!\{\, \mathtt{sum}(i,j) \parallel \mathtt{0} \le i < j \,\}\!\!\} \parallel \mathtt{1} \le j < \mathtt{n}_0\mathtt{+1} \,\}\!\!\}\,),$ $min\,(\mathtt{0},\, \mathbf{min}\, i\, \{\!\!\{\, \mathtt{sum}(i, \mathtt{n}_0\mathtt{+1}) \parallel \mathtt{0} \le i < \mathtt{n}_0\mathtt{+1} \,\}\!\!\})\Big) \;=$
(2.25), (2.24)	$min\Big(min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{snap3}(\mathtt{g}),\, \mathtt{filt}(\mathtt{1}, \mathtt{n}_0\mathtt{+1}))\big),$ $min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{snap2}(\mathtt{g}, \mathtt{n}_0\mathtt{+1}),\, \mathtt{filt}(\mathtt{0}, \mathtt{n}_0\mathtt{+1}))\big)\Big) \;=$
Def. `Q`, $\mathtt{E}_0$, `R`, `F`	$min\,\big(min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{Q},\, \mathtt{E}_0)\big),\, min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{R},\, \mathtt{F})\big)\big) \;=$
Assoc. *min*	$min\,\big(\mathtt{0},\, min\,\big(\widehat{\mathbf{min}}\,(\mathtt{Q},\, \mathtt{E}_0),\, \widehat{\mathbf{min}}\,(\mathtt{R},\, \mathtt{F})\big)\big) \;=$
Spec. `snap3`	$min\,\big(\mathtt{0},\, min\,\big(\widehat{\mathbf{min}}\,(\mathtt{Q},\, \mathtt{E}_0),\, \mathtt{Q}[\mathtt{n}_0\mathtt{+1}]\big)\big) \;=$
(2.21) ■	$min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{Q},\, \mathtt{E})\big) \;=$
Def. `Q`, `E`	$min\,\big(\mathtt{0},\, \widehat{\mathbf{min}}\,(\mathtt{snap3}(\mathtt{g}),\, \mathtt{filt}(\mathtt{1}, \mathtt{n}_0\mathtt{+2}))\big) \;=$
(2.25)	$min\,(\mathtt{0},\, \mathbf{min}\, j\, \{\!\!\{\, \mathbf{min}\, i\, \{\!\!\{\, \mathtt{sum}(i,j) \parallel \mathtt{0} \le i < j \,\}\!\!\} \parallel \mathtt{1} \le j < \mathtt{n}_0\mathtt{+2} \,\}\!\!\}) \;=$
Incr. counter	$min\,(\mathtt{0},\, \mathbf{min}\, j\, \{\!\!\{\, \mathbf{min}\, i\, \{\!\!\{\, \mathtt{sum}(i,j) \parallel \mathtt{0} \le i < j \,\}\!\!\} \parallel \mathtt{1} \le j < \mathtt{n}\mathtt{+1} \,\}\!\!\})$

(Outline-10)

Here, $\mathtt{E}_0 \equiv \mathtt{filt}(\mathtt{1}, \mathtt{n}_0\mathtt{+1})$ and $\mathtt{E} \equiv \mathtt{filt}(\mathtt{1}, \mathtt{n}_0\mathtt{+2})$ denote the *filters* of the (outermost) comprehension at the beginning and at the end of the current iteration, resp.; $\mathtt{Q} \equiv \mathtt{snap3}(\mathtt{g})$

encodes this comprehension's body (which is the same in both states because `snap3(g)` does not depend on `n`).

The very first step follows from the operational semantics of the assignments to `x` in Fig. 2.12. In the second step, we apply the knowledge of the invariant `Y` (verified in Sec. 2.3.1.3). The third step applies the *induction hypothesis* of `X`, substituting x_0 with its value in terms of comprehensions; we then translate these comprehensions to separation logic, applying (2.24) and (2.25). We then rewrite the snapshots and filters using single-letter abbreviations and apply *associativity* of min: $min(min(0,a),min(0,b)) = min(0,min(a,b))$.

Next, we replace $\mathbf{min}$`(R, F)` with the map lookup `Q[`n_0`+1]`. To justify this step, one can instantiate the quantifier in the postcondition of `snap3` with ``$n_0$`+1` for $j$. Since (from the 2nd line of our loop invariant) ``n_0 $<$ `size(array)` and (from the first postcondition of `snap3`) **`domain`**`(Q)` $=$ `filt(1, size(array)+1)`, the condition ``n_0`+1` $\in$ **`domain`**`(Q)` is satisfied; hence, we learn `Q[`n_0`+1]` $=$ $\widehat{\mathbf{min}}$`(snap2(g, `n_0`+1), filt(0, `n_0`+1))`. Recalling that `R` $\equiv$ `snap2(g,`n_0`+1)` and `F` $\equiv$ `filt(0,`n_0`+1)`, we justify the claim.

We conclude the proof by considering a singleton composition (`E` $=$ `E`$_0$ $\uplus$ ***`Set`***`(`n_0`+1)`). Finally, we substitute `Q` and `E` their definitions and translate the resulting comprehension back to the specification language and take into account the increment of the loop counter `n`, obtaining the target value for `x` at the end of the iteration. The postcondition of `minSubArraySum` follows from instantiating `X` with the ultimate value of `n`, namely `size(array)-1`. □

2.3.1.5 Discussion. The case of `minSubArraySum` demonstrates how our technique handles the interaction of nested comprehensions in a single method's specification. Nesting is one of the two approaches for writing multidimensional comprehensions in our technique; the other approach involves *tupled iterated variables* (Sec. 2.3.2).

The encoding of nested comprehensions' bodies relies on the possibility of `snap` to capture values from its evaluation context, e. g. the iterated variables of the outer comprehensions. In the example of Fig. 2.12, capturing occurs in the *inner* $\mathbf{min}$'s body: the iterated variable j of the outer comprehension is captured in the invariant `X` and the variable `n` is captured in the invariant `Y`. Note that capturing in the *filters* of comprehensions (which does not happen in this case) would not affect the encoding.

All three comprehension families share a single (state-independent) version of the `filt` function. We leave open the problem of automatically detecting and caching the versions of `filt` and `snap` that can be reused for encoding multiple comprehension families. However, generating these independently would not require any conceptual changes in the proof.

This case also demonstrates the difference in working with monoid-based comprehensions, e. g. $\mathbf{sum}$ vs. general semigroup-based comprehensions for which a unit is not available. In the former case, the filtering conditions can be specified more liberally because these comprehensions are defined over empty filters. Conversely, preventing

empty filters is essential for e. g. **min**. This is reflected in our specifications of Fig. 2.12. In particular, our invariant for `minSubArraySum` is slightly different from that of the Spec# benchmark (cf. Fig. 2.1) as the latter relies on finite integers for which there exists a maximal one, whereas our type system supports unbounded (mathematical) integers.

We verified this benchmark in Viper. However, some manual assertions were needed for triggering the decompositions described above. Refer to Sec. 2.5 for more details.

2.3.2 Multi-structural comprehensions

2.3.2.1 Generalized representation. In order to encode multiple (disjoint) substructures that a single method can operate on, we use an extended form of our `lift` function that takes two arguments: the overall footprint and its *subset*, i. e. the set of nodes of a particular substructure. Our goal is to enable accessing nodes of *either* of the disjoint substructures via their (independent) indices. To that end, we axiomatize our extended `lift` function as follows:[20]

$$\begin{aligned}
&\forall \mathfrak{g}, \mathfrak{h}\texttt{:}\textbf{\textit{Set}}\texttt{[}\textbf{\textit{Ref}}\texttt{]}, n\texttt{:}\textbf{\textit{Int}} \bullet \mathfrak{h} \subseteq \mathfrak{g} \wedge n \in \texttt{domain(lift}(\mathfrak{g}, \mathfrak{h})) \implies n = \texttt{indexOf}(\mathfrak{g}, \mathfrak{h}, \texttt{lift}(\mathfrak{g}, \mathfrak{h})[n]) \\
&\forall \mathfrak{g}, \mathfrak{h}\texttt{:}\textbf{\textit{Set}}\texttt{[}\textbf{\textit{Ref}}\texttt{]}, n\texttt{:}\textbf{\textit{Int}} \bullet \mathfrak{h} \subseteq \mathfrak{g} \implies (0 \leq n < |\texttt{domain(lift}(\mathfrak{g}, \mathfrak{h}))| \Leftrightarrow n \in \texttt{domain(lift}(\mathfrak{g}, \mathfrak{h}))) \\
&\forall \mathfrak{g}, \mathfrak{h}\texttt{:}\textbf{\textit{Set}}\texttt{[}\textbf{\textit{Ref}}\texttt{]} \bullet \mathfrak{h} \subseteq \mathfrak{g} \implies \texttt{range(lift}(\mathfrak{g}, \mathfrak{h})) = \mathfrak{h}
\end{aligned} \tag{2.26}$$

The first axiom uses an auxiliary function `indexOf`, ensuring that for fixed $\mathfrak{g}$ and $\mathfrak{h}$ the mapping `lift(g,h)` is injective. The second and the third axioms constrain the domain and the range of the mapping, resp. The domain of the extended lifted representation is defined in terms of a range of non-negative integers; intuitively, this corresponds to conventional array indices. The range of a substructure's extended lifted representation is equal to this substructure's footprint; intuitively, this is because each substructure node must be accessible via *some* index.

The extended lifted representation presented above is suitable for specifying *multiple* disjoint substructures. The axioms of (2.26) generalize those presented earlier (cf. (2.16)); in particular, if a method with footprint $\mathfrak{g}$ operates on just one substructure, then its lifted representation can be obtained via the extended function simply as `lift(g,g)`. We present the axioms of (2.26) only for *integer indices* (as this is the domain used in our case study); the further generalization to an arbitrary index domain is straightforward.

2.3.2.2 Overview. The method `coincidenceCount` Fig. 2.15 operates on two disjoint heap-allocated *sorted sets* `A` and `B`. The precondition requires that the method's footprint (specified via the node set $\mathfrak{g}$) consists of the disjoint union of `A` and `B` and that the two structures are indeed sorted. The latter properties are specified (via a first-order quanti-

20 These are static axioms that do not depend a particular example.

fier of `sorted`) over the (extended) lifted representations of `A` and `B`, denoted `xs` and `ys`, resp. The postcondition ensures that the output parameter `res` contains all the coincidences in `xs` and `ys`. For example, if `xs` is $\{1,3,7,9\}$ and `ys` is $\{3,4,5,6,7\}$, then the result stored in `res` by the time the method terminates will contain the value of `2` as there are two coincidences, namely 3 and 7.

```
field val: Int
define xs lift(g, A)
define ys lift(g, B)
define size(zs) |domain(zs)|
define sorted(zs) ∀i,j ∈ domain(zs) • i < j ⟹ zs[i].val < zs[j].val

method coincidenceCount(g: Set[Ref], A: Set[Ref], B: Set[Ref]) returns (res: Int)
  requires ACCESS_NODES(g, read)
           g = A ⊎ B
           sorted(xs) ∧ sorted(ys)
  ensures ACCESS_NODES(g, read)
  res = 𝔰𝔲𝔪 p ⦃ (xs[p1].val = ys[p2].val) ? 1 : 0 ‖ 0 ≤ p1 < size(xs) ∧ 0 ≤ p2 < size(ys) ⦄
{
  var m, n := 0, 0; res := 0
  while (m < size(xs) ∧ n < size(ys))
    invariant ACCESS_NODES(g, read)
              sorted(xs) ∧ sorted(ys)
              0 ≤ m ≤ size(xs) ∧ 0 ≤ n ≤ size(ys)
              m ≠ size(xs) ⟹ ∀i • 0 ≤ i < m ⟹ xs[i] < ys[n]
              n ≠ size(ys) ⟹ ∀j • 0 ≤ j < n ⟹ ys[j] < xs[m]
              res = 𝔰𝔲𝔪 p ⦃ (xs[p1].val = ys[p2].val) ? 1 : 0 ‖ 0 ≤ p1 < m ∧ 0 ≤ p2 < n ⦄
  {
    if (xs[m].val < ys[n].val) m := m+1
    elseif (xs[m].val > ys[n].val) n := n+1
    else { res := res+1; m := m+1; n := n+1 }
} }
```

Figure 2.15: Implementation of `coincidenceCount` and its comprehensive specification.

The algorithm computes the number of coinciding values in two *sorted sets* (of possibly different sizes), denoted `xs` and `ys`. Each iteration compares the $\mathtt{m}^{\text{th}}$ value of `xs` with the $\mathtt{n}^{\text{th}}$ value of `ys`; if these values are not equal, the index of the least of the two is incremented; otherwise, both indices, as well as the coincidence counter `res`, are incremented. The comprehensive property of the loop invariant expresses that all the coincidences are indeed counted.

2.3.2.3 Encoding comprehensions with tupled iterated variables. To support the comprehension from Fig. 2.15, we augment the rules of our encoding technique to support *tupled iterated variables*. The main challenge is to identify a viable encoding design that can be applied *automatically*, without breaking any convenient abstractions.

```
function snap4(g: Set[Ref], lift1, lift2: Map[Int,Ref]): Map[Pair[Int,Int], Int]
  requires ACCESS_NODES(g)
           g = range(lift1) ⊎ range(lift2)
  ensures domain(result) = domain(lift1) × domain(lift2)
          ∀p ∈ domain(result) • result[p] = ((lift1[p1].val = lift2[p2].val) ? 1 : 0)

function filt4(g, lift1,lift2:Map[Int,Ref], a, b, c, d:Int):Set[Pair[Int,Int]]
  requires // ACCESS_NODES(g) -- filtering condition is state-independent
          g = range(lift1) ⊎ range(lift2)
  ensures result ⊆ domain(lift1) × domain(lift2)
          ∀p ∈ domain(lift1) × domain(lift2) • (p ∈ result ⟺ a ≤ p1 < b ∧ c ≤ p2 < d)
```

Figure 2.16: Encoding the snapshot and filter for a two-dimensional comprehension.

Recall that the main ingredients that our technique used for encoding comprehensions into separation logic are the lifted representation (for accessing heap-allocated nodes via indices) snapshots (for a comprehension's body) and filtering functions (for the filtering condition). The current generalization extends all three of these ingredients.[21] The extension of `lift` has been presented in Sec. 2.3.2.1; intuitively, if $\mathfrak{h}$ is a subset of the current footprint, denoted $\mathfrak{g}$, then the extended lifted representation `lift(g,h)` is the mapping from indices of $\mathfrak{h}$ to the actual nodes.

The corresponding extended encoding of snapshots and filters, called `snap4` and `filt4`, are presented in Fig. 2.16. For convenience, the snapshots are parameterized with extended lifted representations (denoted $\texttt{lift}_1$ and $\texttt{lift}_2$); in our example of `coincidenceCount`, these can be instantiated with `xs` and `ys`, resp. The precondition of `snap4` requires that $\mathfrak{g}$ is the footprint that consists of two disjoint parts, $\textbf{range}(\texttt{lift}_1)$ and $\textbf{range}(\texttt{lift}_2)$; i. e. $\mathfrak{g}$ consists of the nodes of both our substructures. The same preconditions are generally shared by `filt4` as well, although in Fig. 2.15 the filtering condition is state-*in*dependent, so `filt4` does not practically require any access permissions.

In the postconditions of `snap4` and `filt4`, the main novelty is that for the snapshot's domain we use a *Cartesian product* of all the extended lifted representation domains. Recall that the domain of the snapshot is the set of permitted instantiations of a comprehension's iterated variable. Since in this case our indices are *tuples* (pairs of integers), the permitted instantiations of an iterated variable, say $p \equiv (p_1, p_2)$, are those in which both its components correspond to some nodes of the respective substructures. Thus, p belongs to the snapshot domain *iff* p_1 and p_2 belong to the respective extended lifted representation domains; hence the Cartesian product.

21 However, we closely follow the design principles established in Sec. 2.2.4.

Applying these functions, we obtain the following encoding of our two-dimensional comprehension, i. e. a concrete variant of the general translation rule of (2.17):

$$\mathfrak{sum}\, p \,\{\!|\, (\texttt{xs}[p_1].\texttt{val} = \texttt{ys}[p_2].\texttt{val})\,?\,\texttt{1}:\texttt{0} \parallel \texttt{0} \le p_1 < \texttt{m} \wedge \texttt{0} \le p_2 < \texttt{n} \,|\!\}$$
$$\twoheadrightarrow\ \widehat{\mathfrak{sum}}\,(\texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}), \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}, \texttt{0}, \texttt{n})) \tag{2.27}$$

2.3.2.4 Verifying the first two branches. To verify the loop invariant of `coincidenceCount`, we must check that it is preserved in each of the three branches of the loop body. The first two possibilities are symmetric with respect to `m` and `n`; we therefore omit the proof outline for the second branch. We assume the branch condition $\texttt{xs[m].val} < \texttt{ys[n].val}$ and proceed with the following derivation written in terms of the *initial* state of the loop body, i. e. before the increments:

Then-Branch	$\texttt{res} \;=\; \texttt{res}_0 \;=$
Ind. hyp.	$\mathfrak{sum}\, p \,\{\!|\, (\texttt{xs}[p_1].\texttt{val} = \texttt{ys}[p_2].\texttt{val})\,?\,\texttt{1}:\texttt{0} \parallel \texttt{0} \le p_1 < \texttt{m}_0 \wedge \texttt{0} \le p_2 < \texttt{n}_0 \,|\!\} \;=$
Def. $\texttt{Q}, \texttt{F}_0$	$\widehat{\mathfrak{sum}}\,(\texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}), \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}_0, \texttt{0}, \texttt{n}_0)) \;=$
(2.27)	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{F}_0) \;=$
Arith.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{F}_0) + \texttt{0} \;=$
Monoton.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{F}_0) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{F} \setminus \texttt{F}_0) \;=$
(2.22) ◪	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{F}) \;=$
Def. $\texttt{Q}, \texttt{F}$	$\widehat{\mathfrak{sum}}\,(\texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}), \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}_0\texttt{+1}, \texttt{0}, \texttt{n}_0)) \;=$
(2.27)	$\mathfrak{sum}\, p \,\{\!|\, (\texttt{xs}[p_1].\texttt{val} = \texttt{ys}[p_2].\texttt{val})\,?\,\texttt{1}:\texttt{0} \parallel \texttt{0} \le p_1 < \texttt{m} \wedge \texttt{0} \le p_2 < \texttt{n} \,|\!\}$

(Outline-11)

Here, $\texttt{Q} \equiv \texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys})$ denotes the *snapshot* (note that it is independent of any of the variables modified in the loop); $\texttt{F}_0 \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}_0, \texttt{0}, \texttt{n}_0)$ and $\texttt{F} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}_0\texttt{+1}, \texttt{0}, \texttt{n}_0)$ are the filters at the beginning and at the end of the current iteration, resp. (see Fig. 2.17), and $\texttt{res}_0$ is the value of `res` at the beginning of this iteration. In this branch, the variable `res` remains unchanged; hence the very first step in the above derivation.

In the second step, we apply the *induction hypothesis* for the loop invariant, substituting $\texttt{res}_0$ with its value in terms of our $\mathfrak{sum}$ comprehension. We then translate the comprehension into separation logic via our rule of (2.27). We proceed by adding a zero-term and replacing it with a sum over the filter $\texttt{F} \setminus \texttt{F}_0$; the fact that this comprehension equals zero follows from *monotonicity* of $\mathfrak{sum}$ (e. g. $(\forall i \in F \bullet a_i = 0) \implies \mathfrak{sum}\, i \,\{\!|\, a_i \parallel F \,|\!\} = 0$).[22] Concretely, we have $\texttt{n}_0 < \texttt{size(ys)}$ (from the loop guard), so the implication from the 5th line of the loop invariant of Fig. 2.15 yields $\forall j \bullet \texttt{0} \le j < \texttt{n}_0 \Rightarrow \texttt{ys}[j] < \texttt{xs}[\texttt{m}_0]$. Recall

22 We assume for now that this property is available to the prover and postpone the corresponding axiomatization until Sec. 2.4.6.

that instantiating the comprehension's iterated variable p (i. e. evaluating a lookup in its snapshot Q) with any (permitted) pair (i, j) yields $(\texttt{xs}[i] = \texttt{ys}[j]\ ?\ 1 : 0)$. This expression evaluates to 0 for all pairs of the filter $\texttt{F} \setminus \texttt{F}_0 = \{(i,j) \mid i = \texttt{m}_0 \wedge 0 \leq j < \texttt{n}_0\}$. Hence, we apply monotonicity of $\mathfrak{sum}$ and learn $\widehat{\mathfrak{sum}}(\texttt{Q}, \texttt{F} \setminus \texttt{F}_0) = 0$.

We conclude the derivation by considering a composition of two disjoint filters, $\texttt{F}_0$ and $\texttt{F} \setminus \texttt{F}_0$. Finally, we translate the resulting comprehension back to the specification language and take into account the increment of m, obtaining the target value for res in this branch. □

2.3.2.5 Verifying the else-branch. We proceed by considering the else-branch in which we write the following derivation, as before, in terms of the *initial* state of the loop body:

Else-Branch	$\texttt{res} = \texttt{res}_0\texttt{+1} =$
Ind. hyp.	$\mathfrak{sum}\, p \,\{\!\{\, (\texttt{xs}[p_1].\texttt{val} = \texttt{ys}[p_2].\texttt{val})\ ?\ 1 : 0 \parallel 0 \leq p_1 < \texttt{m}_0 \wedge 0 \leq p_2 < \texttt{n}_0 \,\}\!\} + 1 =$
(2.27)	$\widehat{\mathfrak{sum}}\,(\texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}), \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, 0, \texttt{m}_0, 0, \texttt{n}_0)) + 1 =$
Def. $\texttt{Q}, \texttt{E}_{00}$	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + 1 =$
Else-Branch	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + (\texttt{xs}[\texttt{m}].\texttt{val} = \texttt{ys}[\texttt{n}].\texttt{val})\ ?\ 1 : 0 =$
snap4	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \texttt{Q}[(\texttt{m}, \texttt{n})] =$
(2.19) ☮	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{11}) =$
Arith.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + 0 + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{11}) =$
Monoton.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{10}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{11}) =$
(2.22) ▚	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{10} \uplus \texttt{E}_{11}) =$
Arith.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + 0 + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{10} \uplus \texttt{E}_{11}) =$
Monoton.	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{01}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{10} \uplus \texttt{E}_{11}) =$
(2.22) ▚	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{01} \uplus \texttt{E}_{10} \uplus \texttt{E}_{11}) =$
Def. $\texttt{E}_{01}, \texttt{E}_{10}, \texttt{E}_{11}$	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}_{00}) + \widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E} \setminus \texttt{E}_{00}) =$
(2.22) ▚	$\widehat{\mathfrak{sum}}\,(\texttt{Q}, \texttt{E}) =$
Def. Q, E	$\widehat{\mathfrak{sum}}\,(\texttt{snap4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}), \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, 0, \texttt{m}_0\texttt{+1}, 0, \texttt{n}_0\texttt{+1})) =$
(2.27)	$\mathfrak{sum}\, p \,\{\!\{\, (\texttt{xs}[p_1].\texttt{val} = \texttt{ys}[p_2].\texttt{val})\ ?\ 1 : 0 \parallel 0 \leq p_1 < \texttt{m} \wedge 0 \leq p_2 < \texttt{n} \,\}\!\}$

(Outline-12)

Here, $\texttt{E}_{00} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, 0, \texttt{m}_0, 0, \texttt{n}_0)$ and $\texttt{E} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, 0, \texttt{m}_0\texttt{+1}, 0, \texttt{n}_0\texttt{+1})$ are the *filters* of the comprehensions at the beginning and at the end of the current iteration, resp. The set difference between these filters is comprised of disjoint parts: $\texttt{E}_{01} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, 0, \texttt{m}_0\ \texttt{n}_0, \texttt{n}_0\texttt{+1})$, $\texttt{E}_{10} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{m}_0, \texttt{m}_0\texttt{+1}\ 0, \texttt{n}_0)$, and the singleton filter $\texttt{E}_{11} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{m}_0, \texttt{m}_0\texttt{+1}\ \texttt{n}_0, \texttt{n}_0\texttt{+1})$. We illustrate this four-way decomposition in Fig. 2.17.

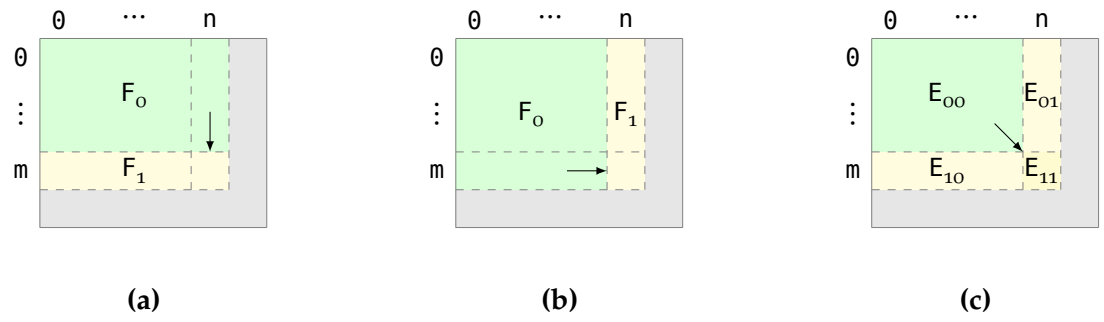


Figure 2.17: Three possible decompositions of a two-dimensional index domain of Fig. 2.15.

Rectangles represent partitions of the space of `(m, n)`-indices, i. e. the decompositions used in the proofs. The part that will be covered on this iteration is yellow (the south-eastern cell represents the indices that are actually being analyzed in the algorithm); the part already processed is green; the remainder is gray. Arrows denote the *direction of propagation* of the algorithm and correspond to the three branches in the loop of `coincidenceCount`.

The very first step accounts for the increment of `res` in the else-branch of `coincidenceCount`. We then apply the *induction hypothesis*, substituting the value of `res` at the beginning of the current iteration with its assumed value in terms of the $\mathfrak{sum}$ comprehension. We then apply the knowledge that the branch conditions did not hold (since this is the else-branch), learning `xs[m].val = ys[n].val`; hence, we justify the substitution of `+1` with the ternary expression, following the pattern in the comprehension's body. We then replace the ternary expression with the map lookup `Q[(m,n)]` (formally, by instantiating the quantifier in the postcondition of `snap4(g, xs, ys)` with `(m,n)` for p), which is justified since `(m,n)` $\in$ **`domain`**`(xs)` $\times$ **`domain`**`(ys)` follows from the 3rd line of the loop invariant (Fig. 2.12). We now instantiate (2.19), replacing the map lookup `Q[(m,n)]` with a comprehension over the *singleton filter* containing `(m,n)`, denoted $\mathtt{E}_{11}$.

Next, we add a zero-term and substitute it with the comprehension over $\mathtt{E}_{10}$; the fact that this comprehension evaluates to zero follows from the 4th and 5th lines of the loop invariant Fig. 2.15 and monotonicity of $\mathfrak{sum}$ (which we have assumed to be available to the prover). We then replace the sum of two comprehensions over the disjoint filters $\mathtt{E}_{10}$ and $\mathtt{E}_{11}$ with their composition, i. e. a single comprehension over the united filter $\mathtt{E}_{10} \uplus \mathtt{E}_{11}$. We repeat the same pattern for the filter $\mathtt{E}_{11}$, obtaining the comprehension over the composite filter $\mathtt{E}_{01} \uplus \mathtt{E}_{10} \uplus \mathtt{E}_{11}$. This filter is exactly the difference between `E` and $\mathtt{E}_{00}$, leading to the ultimate composition, namely, $\mathfrak{sum}$`(Q, E)`. Finally, we translate the resulting comprehension back to the specification language and take into account the increments of `m` and `n`, obtaining the target value for `res`. □

2.3.2.6 Deriving the postcondition. The last step in the proof is to use the verified loop invariant of Fig. 2.15 to justify the ultimate postcondition of the method. The required

derivation is similar to Outline-12 in that it involves another four-way decomposition of the target filter (which in this case is $\texttt{D} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{size(xs)}, \texttt{0}, \texttt{size(ys)})$). The four disjoint subfilters of D are: $\texttt{D}_{00} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}, \texttt{0}, \texttt{n})$, $\texttt{D}_{01} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{0}, \texttt{m}, \texttt{n}, \texttt{size(ys)})$, $\texttt{D}_{10} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{m}, \texttt{size(xs)}, \texttt{0}, \texttt{n})$, and $\texttt{D}_{11} \equiv \texttt{filt4}(\mathfrak{g}, \texttt{xs}, \texttt{ys}, \texttt{m}, \texttt{size(xs)}, \texttt{n}, \texttt{size(ys)})$; these subfilters represent the four corners of the rectangular filter D in the two-dimensional space of comprehension indices; Fig. 2.17 illustrates this decomposition (with all D-symbols replaced by E).

Intuitively, the reason we consider the decomposition $\texttt{D} = \texttt{D}_{00} \uplus \texttt{D}_{01} \uplus \texttt{D}_{10} \uplus \texttt{D}_{11}$ is that, after the loop terminates, only one of the two indices m, n must reach the end of its range, while the other may or may not. Hence, we consider both cases under which $\texttt{D}_{01}$ or $\texttt{D}_{10}$ might be non-empty filters, leading to the general four-way decomposition. However, even if either of these is a non-empty filter, they do not influence the overall sum of coincidences, which can be proven via the monotonicity lemma of Outline-11. Therefore, the sum computed after the loop is indeed equal to the sum in the postcondition. □

2.3.2.7 Discussion. This case shows that reasoning about multidimensional comprehensions may require complex two-way decompositions even if the implementation operates on individual nodes. This complexity stems from the fact that our decompositions are over the space of indices rather than the nodes of heap structures. In particular, we considered the `coincidenceCount` algorithm that is *efficient*, i. e. it traverses only a small part of the entire index space while leveraging monotonicity of the data structure to avoid missed coincidences. Conversely, the specification plainly summarizes the property of the entire two-dimensional space, which makes it intuitive yet harder to verify.

We demonstrated that the design principles of our technique can be generalized to support the case of multidimensional comprehensions through tupled iterated variables. This approach is different from the one used in Spec#, the latter encoding multidimensional comprehensions only as nested comprehensions. Our technique supports nesting as well, as we demonstrated in Sec. 2.3.1. However, the alternative of encoding multidimensional comprehensions via tupled indices opens new directions for e. g. performance optimization as well as domain-specific applications, e. g. a specification language dedicated for tensor-transforming algorithms, in which case tupled indices and the rectangular decompositions that they require (e. g. Fig. 2.17) would be a natural fit.

Our generalized design enables a conceptually simple encoding and has the following advantages: (1) a *single snapshot function* suffices per comprehension family, (2) the snapshot' postcondition *syntactically matches* the comprehension's body, and the filtering function's postcondition *syntactically matches* the comprehension's filter, (3) the design agrees with the rest of our comprehension reasoning technique presented so far.

We verified this benchmark in Viper. However, some manual assertions were needed for triggering the decompositions described above. Refer to Sec. 2.5 for more details.

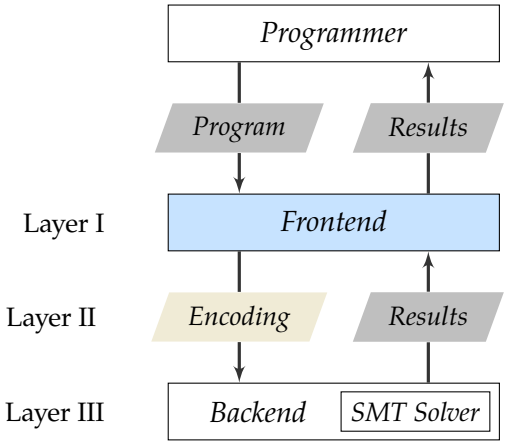


Figure 2.18: Overview of the tool stack.

Programmer submits a program in our source language and its comprehensive specifications. Three layers define the verifier: *Frontend* encodes comprehensions into separation logic; *Backend* checks the proof obligations via SMT and reports raw verification results. For each assertion (e. g. invariant or postcondition check), *Backend* reports either success or a potential violation, possibly with a counterexample. *Frontend* decodes these results, mapping locations of failed assertions and counterexamples to those in terms of the source program.

2.4 LOGICAL ENCODING

We will now demonstrate how to encode our technique into separation logic. We designed our technique for automation; thus, we begin with a high-level implementation overview of a possible frontend verifier (Sec. 2.4.1). We proceed by introducing the comprehension function (Sec. 2.4.2) and explain how it is encoded for various commutative semigroups (Sec. 2.4.3). We then present the encoding of snapshots (Sec. 2.4.4) and filters (Sec. 2.4.5). Next, we encode the first-order axiomatization of the comprehension function (Sec. 2.4.6). We then discuss our quantifier instantiation strategy (Sec. 2.4.7). Finally, we conclude this section with the encoding of consistency checks (Sec. 2.4.8).

2.4.1 Implementation overview

Our implementation consists of three layers (Fig. 2.18). Layer I: A lightweight frontend verifier that is responsible for defunctionalizing bound and program-specific parameters (e. g. rewriting the field set F and the λ-arguments of `snap` and `filt`), as well as lifting verification results in terms of the underlying tools back to the source language. Layer II: the static axiomatization, functions, macro definitions, and the translation of all the methods of the source program into the Viper language Fig. 1.1. Layer III: A backend separation-logic verifier that we treat as a black box. Our premier focus in this section is on the implementation aspects that are specific to comprehensions, corresponding to Layers II–III of the tool stack.

AXIOMATIZATION APPROACH. The example proof outlines in Sec. 2.2.5 demonstrate that a relatively small number of axioms is sufficient for verifying diverse program properties that can be expressed in terms of set comprehensions. We will now show that axioms can be encoded into separation logic and applied automatically. The key idea is to augment

the proof obligations generated by a separation-logic verifier with the first-order axioms that define the semantics of the function symbols such as $\widehat{\texttt{comp}}$.

To make our axioms applicable in an arbitrary context, we employ *universal quantification*. While universal quantifiers enable the desired level of generality of our axiomatization, they are challenging to automate; generally, the problem of *quantifier instantiation* in undecidable. Nevertheless, modern SMT solvers demonstrate that quantifier instantiation can be efficient and predictable in problems that require a limited number of instantiations, most notably, program verification [40, 57, 59, 71, 84, 98, 107].

In our experiments, we use the most prominent quantifier instantiation technique for program verification — *E-matching*. Intuitively, E-matching requires each quantifier to be annotated with *matching patterns* that restrict the syntactic shape of ground terms with which the SMT solver can instantiate the quantified variables (cf. Sec. 2.1.2). It is thus important to select these patterns wisely, s.t. multiple axioms can efficiently interact, while preventing diverging proofs.

2.4.2 Comprehension function

We encode the comprehension function (2.13) as an uninterpreted, ternary function $\widehat{\texttt{comp}}$ over three arguments: the *semigroup identifier* `t`, the *snapshot* `R`, and the *filter* `F`:

```
function comp(t: Int, R: Map[T, S], F: Set[T]): S
```

2.4.3 Encoding of semigroups

Using semigroup identifiers allows us to share the same generic axiomatization for different comprehension kinds. For example, our technique translates the comprehension instances of the form `comp[`min`]`... and `comp[+,0]`... as $\widehat{\texttt{comp}}$`(__min__, ...)` and $\widehat{\texttt{comp}}$`(__sum__, ...)`, resp., where the semigroup identifiers are defined as follows:

```
define __sum__     000   /* comp[+,0] */     axiom { __unit__(__sum__)     == 0 }
define __product__ 001   /* comp[*,1] */     axiom { __unit__(__product__) == 1 }
define __min__     002   /* comp[min] */
define __max__     003   /* comp[max] */
```

This list can be extended by specifying other (named) commutative semigroups. For example, to encode the comprehensions of the form `comp[union,Set()]`..., one can add the following definition:

```
define __union__   004  // comp[union,Set()]
```

This permits translating the above comprehensions as $\widehat{\texttt{comp}}$ `(__union__, ...)`.

For each semigroup, we encode its binary step operator $\oplus$ via a *ternary* uninterpreted function `__oper__`; the 1st argument is the semigroup's identifier, while the 2nd and the 3rd arguments correspond to the two operands of $\oplus$. If a semigroup additionally has a unit, i. e. it is a monoid, the unit is defined for the semigroup's identifier as the value of an uninterpreted function called `__unit__`:

```
8  function __unit__(t: Int): S
9  function __oper__(t: Int, left_arg: S, right_arg: S): S
```

We proceed with the axiomatization of semigroups. Recall from Fig. 2.3 that a comprehension kind is defined by a semigroup $\mathcal{S}$ and an iterated variable (a typed declaration). In our encoding, the latter instantiates the type parameter `T` (i. e. the index type) of our type-parametric axiomatization. All the other aspects of a comprehension kind follow from the semigroup. In particular, the type of the semigroup's step operator, $\oplus$, instantiates the second type parameter, `S`. Next, we axiomatize the semigroup.[23]

```
10 function $SemigroupName(R: Map[T, S], F: Set[T]): S
11 axiom $SemigroupNameComp { forall R:Map[T,S], F:Set[T] :: { $SemigroupName(R, F) }
12   $SemigroupName(R, F) == comp($SemigroupId, R, F) }

13 axiom $SemigroupNameOper { forall x: S, y: S :: { __oper__($SemigroupId, x, y) }
14   $SemigroupOperDef(__oper__($SemigroupId, x, y), x, y) }
```

To simplify the notation, we introduce *named comprehension functions* for each semigroup; e. g. `Sum(...)` denotes $\widehat{\texttt{comp}}$`(__sum__, ...)`. To refer to a semigroups's name, we use the meta-variable `$SemigroupName`. The corresponding identifiers (unique integer values) are denoted as `$SemigroupId`. The meta-variable `$SemigroupOperDef` is rewritten with the name of a macro definition that constrains the `$SemigroupName` semigroups's operator. For example, while translating the $\widehat{\texttt{sum}}$ comprehension, `$SemigroupOperDef` yields the string `"SUM_OPER_DEF"`, matching the corresponding definition from the list below.

```
15 define SUM_OPER_DEF(res, left, right) (res == left + right)
16 define PRODUCT_OPER_DEF(res, left, right) (res == left * right)
17 define MIN_OPER_DEF(res, left, right) (
18   (res == left || res == right) && (res <= left && res <= right)
19 define MAX_OPER_DEF(res, left, right) (
20   (res == left || res == right) && (res >= left && res >= right)
```

23 The code fragments highlighted in turquoise are rewritten via the frontend translation (cf. Fig. 2.18).

If the programmer uses comprehensions with custom semigroups, then the encoding generates analogous macro definitions for each of them.

To get a concrete version of $\oplus$, one has to instantiate the first argument of `__oper__` with the appropriate comprehension kind identifier. To illustrate this stage of the translation, consider a comprehension of the form **`comp`**`[+,0]` ... . First, the frontend generates the named comprehension (lines 10 to 12 with `"Sum"` and `"__sum__"` for `$SemigroupName` and `$SemigroupId`, resp.):

```
function sum(R: Map[T, S], F: Set[T]): S

axiom SumComp { forall R:Map[T,S], F:Set[T] :: { Sum(R, F) }
  Sum(R, F) == comp(__sum__, R, F)}
```

Second, the frontend generates the axiom that defines the + operator (lines 13 to 14 with `"SUM_OPER_DEF"` for `$SemigroupOperDef`). After expanding the macro (line 15), we obtain the following axiom:

```
axiom SumStep { forall x: Int, y: Int :: { __oper__(__sum__, x, y) }
  __oper__(__sum__, x, y) == x + y }
```

Abstracting the binary operations via an uninterpreted function helps in two ways. On the one hand, the programmer may use either uninterpreted (e. g. **`union`**) or *interpreted* (e. g. +) symbols as binary operators. Interpreted symbols cannot be used in quantifier patterns for instantiating universal quantifiers via E-matching; for example, if x and y are ground terms, then the expression x+y will not result in any new quantifier instantiations, while e. g. `__oper__(__sum__,`x`,`y`)` is, indeed, a valid triggering term. On the other hand, many crucial properties are shared among all comprehension kinds, so using the universal `__oper__` function allows us to keep our generic axiomatization *concise* as each generic axiom that needs to refer to the comprehension's binary operator can use the same universal function symbol.

2.4.4 Encoding of snapshots

The second argument of $\widehat{\texttt{comp}}$, `R`, is the *snapshot*, a map from values of type `T` to values of type `S`. `T` is the type of the comprehension's iterated variable, e. g. `T` in **`comp`**`[`$\mathscr{S}$`]` n`:T` ..., while `S` is the type of the comprehension *instance* and also the type of $\mathscr{S}$'s step operator.

Conceptually, snapshots (Sec. 2.2.4.4) represent the structure of the method footprints in fixed program states. What they *do not* represent is the actual folding of this structure into a single value; snapshots are thus independent of semigroups (as long as the types are respected). Therefore, the encoding of all comprehension instances with the same body (cf. Fig. 2.3) share a single version of the `snap` function, which is encoded according to the following template:

```
21 function snap_$T_$e_$S^F(g: Set[Ref], *captures_of_e): Map[T, S]
22   requires ACCESS_NODES^F(g)
23   ensures domain(result) == domain(lift(g))
24   ensures forall u:T :: { τ(e^F(u)) } { result[u] }
25     u in domain(lift(g)) ==> result[u] == e^F(u)
```

The name of a concrete version of the `snap` function is comprised of the prefix `snap`, followed by the name of the `T`-type, the `$`-encoding of the expression `e`, and the name of the `S`-type. For example, **`comp`**`[+,0]` i`:`***`Int`*** {‖ `xs[`i`].val` ‖ ... ‖} is a comprehension that represents the sum of the `val` fields of nodes of the sequence `xs:` ***`Seq`***`[`***`Ref`***`]`. Both types `T` and `S` are instantiated with ***`Int`***, while the `$`-encoding of the string `"xs[`i`].val"` is `"xs_i_val"`. Conceptually, the local variable `xs` is captured by value in the comprehension's body; in this case, our translation rewrites `*captures_of_e` with `xs:` ***`Seq`***`[`***`Ref`***`]`, making the value of `xs` available in `snap`'s postcondition. Hence, the corresponding version of the `snap` function will have the following signature:

```
function snap_Int_xs_i_val_Int(g: Seq[Ref], xs: Seq[Ref]): Map[Int, Int]
```

The precondition of `snap` requires access to the fields from the set F of objects in `g`. This footprint is encoded via the macro definition `ACCESS_NODES`^{F}`(g)` defined as follows:

```
26 define ACCESS_NODES^F(g, p=read) (
27   !(null in g) && ✳_{f∈F} (forall n:Ref :: n in g ==> acc(n.f, p)))
```

The big star is rewritten as a chain of separating conjunctions for all fields $f \in F$. If the second macro argument (i. e. the permission amount) is skipped, it is assumed to have the default value of **`read`**. We assume that the above macro definition is also used for specifying the footprints of all the *methods* with comprehensive specifications. Hence, the footprint of `snap` matches the current method's footprint, except it requires only read permissions to each node's field, while methods may have full, write access.

The postcondition of `snap` defines the mapping from indices (of type `T`) to state-dependent values (of type `S`). The latter are equal to the evaluation of the comprehension body, e^F, in the same context in which `snap` has been applied. e^F is a (state-dependent via the fields from F) expression of type `S` over a (named) free variable of type `T`. The name is declared as part of the current comprehension instance, as per Fig. 2.3. $e^F(x)$ denotes an instantiation of the free variable with x. Finally, the expression e^F can mention other free variables which are bound in the current context; these are made available in the postcondition of `snap` by rewriting `*captures_of_e` in the translation.

Note the following two details of our encoding of `snap`'s postconditions. First, the equality **`result`**`[u] == e`^{F}`(u)` holds for all `u` of the index domain, which (due to the first postcondition) is also the domain of the resulting map. Second, the meta clause `τ(e`^{F}`(u))` is rewritten by the translation as a list of (most-specific) triggering patterns

from the expression e^F(u). For example, τ(u.val + (u.next != **null** ? u.next.id : 0)) yields two patterns: u.val and u.next.

ENCODING OF lift. We employ the function lift in the postcondition of snap to refer to the *domain of indices*, i. e. the set of possible instantiations of the iterated variable. Generally, the semantics of lift depend on the data structure's nature. However, we focus on direct-access heap data structures (e. g. arrays and graphs) in which the stored resources can be deterministically accessed via unique keys. In such structures, the index domain must be *injective* into the heap. Therefore, we axiomatize lift as follows:

```
28 function indexOf(g: Set[Ref], x: Ref): T
29 function lift(g: Set[Ref]): Map[T, Ref]
30 axiom LiftRange { forall g:Set[Ref] :: { lift(g) } range(lift(g)) == g }
31 axiom LiftInjectivity { forall g:Set[Ref], i: T :: { lift(g)[i] }
32   indexOf(g, lift(g)[i]) == i }
33 axiom LiftDomain { forall g:Set[Ref], i:T :: {lift(g)[i]} {i in domain(lift(g))}
34   i in domain(lift(g)) <==> lift(g)[i] in g }
```

These axioms directly correspond to (2.16).

2.4.5 Encoding of filters

The last argument of $\widehat{\texttt{comp}}$, F, is the filter, i. e. the set of *intended instantiations* of the comprehension's iterated variable. Generally, the filter must be a subset of the *permitted instantiations* defined by the weakest filter, **domain**(lift(g)). The programmer can specify all monoid-based comprehensive properties using just the weakest filter. However, general comprehensions cannot be encoded this way and may require *custom filters* (Sec. 2.2.4.5). Even for monoids, custom filters allow for an overall more flexible specification language because the programmer may choose whether to constrain a comprehension via the filter or the body.

Since custom filters can be state-dependent expressions, the encoding should map their values to a state-*independent* structure — intuitively, a set of data structure indices — that can be featured in our first-order axiomatization. Conceptually, this idea is similar to that of our mapping in the specification of snap (although filters require somewhat less information). Hence, our encoding of custom filters is based on a state-dependent function called filt. Concretely, we encode a separate version of the filt function for each filtering condition (cf. Fig. 2.3) according to the following template:

```
35 define DISJOINT(A, B) (forall n :: { n in A, n in B } n in A ==> !(n in B))
36 define DISUNION(C, A, B) ((C == A union B) && DISJOINT(A, B))  // C = A ⊎ B
37 function filt_$T_P^F(g: Set[Ref], *captures_of_P): Set[T]
38   requires ACCESS_NODES^F(g)
39   ensures result subset domain(lift(g))
40   ensures forall u:T :: { τ(e^F(u)) } { result[u] }
41     u in domain(lift(g)) ==> (u in result <==> P^F(u))
42   ensures forall g_a: Set[Ref], g_b: Set[Ref] ::
43     { g_a union g_b, filt_$T_P^F(g_a) } { g_a union g_b, filt_$T_P^F(g_b) }
44       g_a != Set() && g_b != Set() &&
45       DISUNION(g, g_a, g_b) ==> DISUNION(result, filt_$T_P^F(g_a), filt_$T_P^F(g_b))
```

The translation of the above template into concrete `filt` functions is analogous to the translation of `snap`. The first postcondition ensures that the custom filter is a subset of the weakest filter, i. e. the index domain. The second one specifies that each value of the index domain belongs to **`result`** *iff* it satisfies the (potentially) state-dependent condition P^F. Note that the latter may capture by value some local variables from the context of the comprehension instance; as with the `snap` template, our translation of `filt` rewrites `*captures_of_`P into a list of function arguments to make those variables available in the postcondition of `filt`.

Custom filters encoded via the `filt` function are an additional challenge for compositional reasoning. We first illustrate this point and then explain how the last postcondition of `filt` plugs the gap.

Consider an operation that modifies the fragment $\mathfrak{h}$ of a heap structure with footprint $\mathfrak{g}$. Let the operation's pre-state be specified via a comprehension with some *custom filtering condition*. The filter is thus encoded as `filt(`$\mathfrak{h}$`)` (i. e. it cannot be simplified to just $\mathfrak{h}$). In the post-state of the operation, we aim to compose the comprehension over the original filter `filt(`$\mathfrak{g}$`)` using the two comprehensions over the filters `filt(`$\mathfrak{h}$`)` and `filt(`$\mathfrak{g} \setminus \mathfrak{h}$`)`. To that end, we need to apply our *split-term* axiom (2.22). Instantiating this axiom requires proving the following lemma about *disjointness of subfilters*: `filt(`$\mathfrak{g}$`)` $=$ `filt(`$\mathfrak{g} \setminus \mathfrak{h}$`)` $\uplus$ `filt(`$\mathfrak{h}$`)`. However, this property is not available.

The last postcondition of `filt` specifies the property that we call *compositionality of filters*. In principle, this property does follow from the second postcondition of `filt`, but the required inductive derivation cannot be automatically established by a first-order SMT solver. Hence, we add it as an explicit formula to our encoding. Note that compositionality of the *weakest filter*, i. e. of the form **`domain`**`(lift(g))`, follows directly from the properties of `lift`.

2.4.6 First-order axiomatization

Our axiomatization of the $\widehat{\texttt{comp}}$ function consists of three parts. First, we relate this function's values to snapshot values and encode a *synonym function* $\widehat{\texttt{cmp}}$ that has the same first-order semantics as $\widehat{\texttt{comp}}$ but helps controlling the instantiations of our axioms. Second, we encode the main axioms of our technique: *same-terms* (2.20), *step* (2.21), and *split-term* (2.22). Finally, we encode the properties of individual comprehension kinds.[24]

2.4.6.1 Preliminaries. We employ a new function symbol, $\widehat{\texttt{cmp}}$. This function has the same signature and semantics as $\widehat{\texttt{comp}}$ but has the advantage of not matching any of our axioms' patterns. Our patterns use exclusively $\widehat{\texttt{comp}}$, while the axiom bodies replace it with $\widehat{\texttt{cmp}}$ (unless the term in the body is identical to one of the patterns). Hence, the terms obtained while instantiating our axioms cannot cause any further instantiations. We will explain this mechanism in more detail in Sec. 2.4.7.

```
46 function cmp(t: Int, R: Map[T, S], F: Set[T]): S
47 axiom CompSynonym {
48   forall t: Int, R: Map[T, S], F: Set[T] :: { comp(t, R, F) }
49     cmp(t, R, F) == comp(t, R, F) }
```

We proceed with the axioms that specify the semantics of $\widehat{\texttt{comp}}$.

```
50 axiom Singleton {
51   forall t: Int, R: Map[T, S], a: T :: { comp(t, R, Set(a)) }
52     a in domain(R) ==> comp(t, R, Set(a)) == R[a] }
```

This axiom expresses the value of a comprehension over a singleton set containing the element `a` with the map lookup of `R` at `a`. Conceptually, `R` is the comprehension's snapshot, while the only instantiation of the iterated variable permitted by the filter is `a`. This axiom encodes the second formula of (2.19). Note that the first formula of (2.19) is defined only for monoid-based comprehensions; we will explain our encoding of monoid-specific properties below.

2.4.6.2 Compositionality axioms. We proceed with the encoding of the main axioms.

24 An axiomatization of set comprehensions cannot be complete. To devise our axioms, we explored the properties that are needed for verifying a diverse set of heap-transforming programs (see Sec. 2.2.5), axiomatizing them in a generic way. For example, our axioms support arbitrary *two-way* filter decompositions but not N-way decompositions that are theoretically possible but have not occurred in our experiments.

```
53 axiom SameTerm {
54   forall t: Int, R1: Map[T, S], R2: Map[T, S], F: Set[T] ::
55     { comp(t, R1, F), comp(t, R2, F) }
56     { F subset domain(R1), comp(t, R2, F) }
57       F != Set() && F subset domain(R1) && F subset domain(R2) &&
58       (forall a: T :: { R1[a] } { R2[a] } a in F ==> R1[a] == R2[a])
59       ==> cmp(t, R1, F) == cmp(t, R2, F) }
```

This axiom encodes (2.20): The nested quantifier checks that the two snapshots `R1` and `R2` agree at least on all elements in the filter `F`, implying that the comprehensions over this filter are equal for these two snapshots.

```
60 axiom DropOne {
61   forall t: Int, R: Map[T, S], a: T, F: Set[T] :: {comp(t, R, F setminus Set(a))}
62     F != Set(a) && F subset domain(R) &&
63     a in F ==> comp(t, R, F) ==
64                __oper__(t, cmp(t, R, F setminus Set(a)), R[a]) }

65 axiom TakeOne {
66   forall t: Int, R: Map[T, S], a: T, F: Set[T] :: { comp(t, R, Set(a) union F) }
67     F != Set() && F subset domain(R) && a in domain(R) &&
68     !(a in F) ==> comp(t, R, Set(a) union F) ==
69                   __oper__(t, cmp(t, R, F), R[a]) }
```

These two axioms encode (2.21). Unlike the original axiom, the LHS of the implication in the encoding of `TakeOne` is *negated*; hence, the sets `Set(a)` and `F` are guaranteed to be disjoint, and the equality on the RHS relates the comprehension over their union and over the comprehension over `F`. The two alternative formulations allow for more possibilities to instantiate the axiom, as can be seen from their corresponding patterns.

```
70 axiom DropMultiple {
71   forall t: Int, R: Map[T, S], A: Set[T], F: Set[T] :: {comp(t, R, F setminus A)}
72     A != F && A != Set() && F subset domain(R) &&
73     A subset F ==> comp(t, R, F) ==
74                    __oper__(t, cmp(t, R, F setminus A), cmp(t, R, A)) }
75 axiom TakeMultiple {
76   forall t: Int, R: Map[T, S], F1: Set[T], F2: Set[T] :: {comp(t,R,F1 union F2)}
77     F1 != Set() && F2 != Set() && F1 subset domain(R) && F2 subset domain(R) &&
78     F1 intersection F2 == Set() ==> comp(t, R, F1 union F2) ==
79                                     __oper__(t, cmp(t, R, F1), cmp(t, R, F2)) }
```

These two axioms encode (2.22). `TakeMultiple` expresses that, given two disjoint filters `F1` and `F2`, the comprehension over their union can be represented via the two comprehensions over these individual filters. Similar to singleton decomposition axioms, we introduce the additional axiom `DropMultiple` above to enhance our axiom instantiation strategy.

2.4.6.3 Encoding special properties. Some comprehension kinds have *special properties*, i. e. properties that stem from the nature of their semigroup that are essential in verification. These properties either do not follow from the previously discussed parts of the axiomatization or require inductive derivations that generally cannot be automatically established by an SMT solver.

Automatically extracting special properties from a limited number of first-order axioms is an instance of the general problem of entailment automation (Sec. 2). We identify and explicitly encode some of the most commonly useful special comprehensive properties, making them available to the verifier. This part of our axiomatization is inherently incomplete; however, it is easily extensible without affecting the rest of the technique[25] in case other special properties will be proven useful in the future.

EMPTY FILTERS. One class of special properties is the value of a comprehension over the empty filter. This value can be represented in our type system only if the comprehension's step operator (2.9) forms a *commutative monoid* with $\mathbb{1}$ as its unit (2.10).[26] Thus, we axiomatize the value of monoid-based comprehension instances over an empty filter:

```
80 IF $IsMonoid THEN
81 axiom Empty$SemigroupNameFilter {
82   forall R: Set[T, S] :: { $SemigroupName(R, Set()) }
83     $SemigroupName(R, Set()) == $Unit }
84 FI
```

The meta flag `$IsMonoid` indicates whether the translated semigroup is a monoid, in which case the axiom is emitted. While expanding the axiom, `$SemigroupName` is translated into the monoid's name, e. g. `"Product"`in case of the `(Int,*,1)` monoid; `$Unit` is translated into its unit, e. g. `1` in `comp[*,1]`.

MONOTONICITY. Another important class of properties is *monotonicity* of semigroups over integers. For example, if all the values stored in a data structure are greater than or equal to zero, then the sum (or the product) of all those values is also non-negative.

25 Provided there are no matching loops in quantifier instantiations. We will explain the problem of matching loops in Sec. 2.4.7.

26 We do not require that the operator has a unit to support a wider class of comprehensions, e. g. **min**.

```
85  axiom Int$SemigroupNamePositive {
86    forall R: Map[T, Int], F: Set[T] :: { $SemigroupName(R, F) }
87    (F != Set()) && (F subset domain(R)) &&
88      (forall a: T :: {a in F} a in F ==> R[a] > 0) ==> $SemigroupName(R, F) > 0 }
89  axiom Int$SemigroupNameNonNegative {
90    forall R: Map[T, Int], F: Set[T] :: { $SemigroupName(R, F) }
91    (F != Set()) && (F subset domain(R)) &&
92      (forall a: T :: {a in F} a in F ==> R[a] >= 0) ==> $SemigroupName(R, F) >= 0 }
93  axiom Int$SemigroupNameNegative {
94    forall R: Map[T, Int], F: Set[T] :: { $SemigroupName(R, F) }
95    (F != Set()) &&  (F subset domain(R))
96      (forall a: T :: {a in F} a in F ==> R[a] < 0) ==> $SemigroupName(R, F) < 0 }
97  axiom Int$SemigroupNameNonPositive {
98    forall R: Map[T, Int], F: Set[T] :: { $SemigroupName(R, F) }
99      (F != Set()) && (F subset domain(R)) &&
100     (forall a: T :: {a in F} a in F ==> R[a] <= 0) ==> $SemigroupName(R, F) <= 0 }
```

The first two axioms (`Positive` and `NonNegative`) are generated for all our built-in semigroups (based on +, *, min, max). The latter two axioms (`Negative` and `NonPositive`) are generated only for +, min, and max; e. g. these are needed for verifying `coincidenceCount` (Sec. 2.3.2). A special axiom for * axiomatizes that the product of multiple integer values is equal to zero *iff* one of the values is zero.

```
101 axiom IntProductZero {
102   (forall R: Map[T, Int], F: Set[T] :: { Product(R, F) }
103     F subset domain(R) &&
104     Product(R, F) == 0 ==> exists a: T :: { R[a] } a in F && R[a] == 0)
105   && (forall R: Map[T, Int], F: Set[T], a: T :: { Product(R, F), R[a] }
106     F subset domain(R) && a in F && R[a] == 0 ==> Product(R, F) == 0) }
```

DISTRIBUTIVITY. We conclude this section by encoding the last series of special properties, namely, *distributivity*. Recall that *addition* distributes over min and max, e. g. $min(a, b) + d = min(a + d,\ b + d)$. If a binary operator $\otimes$ distributes over $\oplus$, then the distributivity can be propagated to an *arbitrary number* of nested applications of $\oplus$, e. g.: $min\,(\ldots min\,(a_0, a_1)\,, \ldots,\ a_N) + d\ =\ min\,(\ldots min\,(a_0{+}d,\ a_1{+}d)\,, \ldots\ a_N{+}d)$. Here, $\{a_i\}$ and $\{a_i + d\}$ are two distributions of values and d is the *distance* between them. Generally, $\texttt{comp[}\oplus\texttt{]}\ n\ \{\!|\ \texttt{e}(n)\ \|\ \texttt{f}\ |\!\} \otimes d\ =\ \texttt{comp[}\oplus\texttt{]}\ n\ \{\!|\ \texttt{e}(n) \otimes d\ \|\ \texttt{f}\ |\!\}$. We showed the practical usefulness of distributivity of *addition* over min in Sec. 2.3.1. Overall, we generate three distributivity axioms for the following cases: + over min, + over max, and * over +.

```
107 function pick(F: Set[T]): Int
108 axiom PickIndex { forall F: Set[T] :: { pick(F) } F != Set() ==> pick(F) in F }
109 axiom Int$SemigroupNameDistributivity {
110   forall R1:Map[T,Int], R2:Map[T,Int], F:Set[T] ::
111     { $SemigroupName(R1, F), $SemigroupName(R2, F) }
112       R1 != R2 && F != Set() && F subset domain(R1) && F subset domain(R2) &&
113       ( forall a:T, b:T :: { R1[a], R2[a], R1[b], R2[b] }
114         a in F && b in F ==> R1[a] ⊗ R2[b] == R1[b] ⊗ R2[a] ) ==>
115       $SemigroupName(R1,F) ⊗ R2[pick(F)] == $SemigroupName(R2,F) ⊗ R1[pick(F)]
```

Here, `x⊗y` is translated to `__oper__($DistribMap($SemigroupId), x, y)` where `$DistribMap` is a meta-function on semigroup identifiers that maps `"__min__"`, `"__max__"`, `"__sum__"` to `"__sum__"`, `"__sum__"`, `"__product__"`, resp.

Our encoding of distributivity addresses the issues of a similar axiom (Distrib) from Spec#. The main problem is to enable instantiating the formula despite the fact that the distance cannot be added to the outermost pattern as we intend to use the formula even if this distance is written as an integer literal (i. e. not a suitable triggering term). Moreover, two distributions of integer values may have a well-defined distance that is *not an integer*. For example, consider *summation* over $\{2, 4, 6\}$ and $\{3, 6, 9\}$; the distance from the prior distribution to the latter one is $\frac{3}{2}$, which is not an integer and therefore cannot be directly represented in our lightweight type system (defined in Fig. 1.1).

We solve these problems by rephrasing the distributivity axiom. First, the inner quantifier expresses that the difference between `R1` and `R2` is *constant* at any two indices `a` and `b`. To illustrate, consider an instantiation of $+$ for $\otimes$ and assume that d is the distance between `R1` and `R2` over the subdomain defined by the filter, `F`. Then, the condition `R1[a]+ R2[b] = R1[b]+ R2[a]` is equivalent to `R2[a]- R1[a] = R2[b]- R1[b]` $= d$.

Second, if the inner quantifier holds, then we learn the RHS. In terms of the above example, we obtain the condition $\widehat{\mathbf{min}}$`(R1,F) + R2[pick(F)] =` $\widehat{\mathbf{min}}$`(R2,F) + R1[pick(F)]`, so $d =$ `R2[pick(F)]-R1[pick(F)]` (we choose $\widehat{\mathbf{min}}$ for this example as $+$ distributes over its step operator, min). The value `pick(F)` represents *some* index from `F` (which exists since `F` is non-empty) at which both `R1` and `R2` must be defined (due the outer implication's premise).[27] Generally, this formula provides the essential information, connecting the values of the two comprehensions over the common filter, `F`, via the values that indirectly specify the distance.

2.4.7 Preventing infinite instantiation chains

Our axioms constrain the values of multiple comprehension instances, e. g. relating the comprehensive properties of the footprints of a callee and its client. Such relational prop-

27 Alternatively, `pick(F)` can be viewed as the Skolem function for the quantifier $\exists p \bullet p \in$ `F`.

erties are inherently *recursive*. The recursive nature of our axioms (and the fact that they are universally quantified formulas) means that each instantiation can mention multiple different terms of the same syntactic shape. For example, consider a simplified version of our axiom *step* (2.21):

$$\forall R,\, F,\, n \bullet n \in F \implies \widehat{\mathbf{comp}}(R, F) = R[n] \oplus \widehat{\mathbf{comp}}(R, F \setminus \mathbf{Set}(n))$$

The naïve encoding of this formula could be written as follows:

```
forall t: Int, R: Map[T, S], F: Set[T], a: T ::
  { comp(t, R, F), R[a] }
    a in F ==> comp(t, R, F) == __oper__(t, comp(t, R, F setminus Set(a)), R[a])
```

Suppose the automatic proof search involving this axiom reaches a hypothesis about the values of the following two expressions: $\widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0)$ and $\mathtt{R}_0[\mathtt{a}_0]$, for some $\mathtt{t}_0$, $\mathtt{R}_0$, $\mathtt{F}_0$, and $\mathtt{a}_0$; consequently, these terms (and all of their sub-terms) are added to the *E-graph*, an internal data structure that helps the SMT solver to keep track of equivalence classes of terms.

Notice that $\widehat{\mathbf{comp}}(\mathtt{t}, \mathtt{R}, \mathtt{F})$ and $\mathtt{R}_0[\mathtt{a}_0]$ together form a viable *trigger* for the universal quantifier above because they syntactically wrap all quantified variables into uninterpreted functions and do not mention any interpreted functions. Hence, the solver will instantiate our quantifier with $\mathtt{t}_0$, $\mathtt{R}_0$, F, $\mathtt{a}_0$ for t, R, F, a, resp., learning the following formula:

$$\mathtt{a}_0 \ \mathbf{in}\ \mathtt{F}_0 \implies \widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0) = \mathtt{\_\_oper\_\_}(\mathtt{t}_0, \widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0 \ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)), \mathtt{R}_0[\mathtt{a}_0])$$

In particular, this instantiation yields the term $\widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0))$ that is also added to the E-graph; using a combination of this term with the priorly added term R[a], the solver will trigger a new instantiation of our quantifier with $\mathtt{t}_0$, $\mathtt{R}_0$, $\mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)$, $\mathtt{a}_0$ for t, R, F, $\mathtt{a}_0$, resp., learning a conceptually new formula:

$$\mathtt{a}_0 \ \mathbf{in}\ \mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0) \implies \widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)) = \mathtt{\_\_oper\_\_}(\mathtt{t}_0, \widehat{\mathbf{comp}}(\mathtt{t}_0, \mathtt{R}_0, \mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)), \mathtt{R}_0[\mathtt{a}_0])$$

Analogously, the solver is free to perform further instantiations using the triggering terms of the form $\mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)\ \ldots\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)$. Each of these terms can be combined with e. g. R[a] to produce a new term of the same shape. The proof search resulting from these instantiations diverges. The possibility of infinite instantiation chains in E-matching is called a *matching loop* [33]. Matching loops may occur whenever there is an infinite sequence of *syntactically distinct* triggering terms; e. g. the fact that $\mathtt{F}_0\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)\ \mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)$ is equal to $\mathtt{F}_0\mathbf{setminus}\ \mathbf{Set}(\mathtt{a}_0)$ alone does not prevent a matching loop, as we explain next.

Our technique does not assume that the SMT solver natively supports theories such as the theory of sets and maps; modern SMT-based verifiers tend to use a *partial axiomatization* of these theories involving multiple universally quantified formulas that themselves are designed to be used with E-matching. In particular, establishing the equality

between the terms F_0 **setminus** ***Set***(a_0) and F_0 **setminus** ***Set***(a_0) **setminus** ***Set***(a_0) would require instantiating the universally quantified axioms defining the semantics of **setminus**. Since our technique does not rely on a particular order of quantifier instantiation, the solver is free to explore the proof space depth-first, possibly leading to an infinite chain of instantiations of the above quantifier, i. e. a matching loop.

While it is generally possible to influence the order of quantifier instantiations in E-matching, e. g. by specifying each quantifier's weight, we opt for a more conservative approach. Concretely, we restrict the set of possible triggering terms by using a *synonym function* [52], $\widehat{\mathbf{cmp}}$; this function has the same signature as $\widehat{\mathbf{comp}}$, and both are axiomatized to be pointwise-equal via the following axiom:

```
function cmp^(t: Int, R: Map[T, S], F: Set[T]): S

axiom CompSynonym {
  forall t: Int, R: Map[T, Int], F: Set[T] :: { comp^(t, R, F) }
    cmp^(t, R, F) == comp^(t, R, F) }
```

Here, $\widehat{\mathbf{cmp}}$ is a fresh function that cannot be used by the programmer. Our encoding uses $\widehat{\mathbf{comp}}$ in the *patterns* of all quantified axioms; the quantifier bodies use only $\widehat{\mathbf{cmp}}$ (except for the cases in which the term in the body already appears in the pattern).

With the help of the synonym function, we obtain an encoding in which each triggering term may trigger only one instantiation of each axiom. While this approach reduces the theoretical completeness of our technique, our experiments show that this strategy is nonetheless sufficient for verifying a diverse set of benchmark programs.

2.4.8 Consistency checking

Our technique ensures that the encoding of each comprehension instance is applied soundly by following the consistency checking approach described in Sec. 2.2.4.7. To illustrate how the checks are encoded, consider a comprehension instance of the form $\mathbf{comp}[\mathscr{S}]\, n{:}\mathsf{T}\ \{\!|\ \mathsf{e}^F(n) \parallel \mathtt{filter}\ |\!\}$ in a state in which $\mathfrak{g}$ is the footprint.

2.4.8.1 Checking the semigroup. We aim to check the following properties.[28] If the programmer wrote a comprehension of the form $\mathbf{comp}[\oplus]\ldots$, we check that $\oplus$ forms a *semigroup*, i. e. that this is a binary operator with the signature $\mathsf{S} \to \mathsf{S} \to \mathsf{S}$ for some type S, and that it is commutative and associative (2.9). If the programmer instead wrote $\mathbf{comp}[\oplus, \mathbb{1}]\ldots$, we additionally check that $\mathbb{1}$ is of the same type, S, and is, indeed, the neutral element of the semigroup (2.10). These checks are emitted during the translation as a proof obligation inside a special method called `$SemigroupName_obligations`.

28 Recall that our technique permits the programmer to specify custom semigroups; thus, consistency cannot be established once for all possible semigroups.

```
116 method $SemigroupName_obligations() {
117   var x: T; var y: T; var z: T  // fresh variables
118   assert ⊕(x, y) == ⊕(y, x)
119   assert ⊕(⊕(x, y), z) == ⊕(x, ⊕(y, z))
120   IF $IsMonoid THEN
121     assert ⊕(𝟙, x) == x
122   FI }
```

Recall that `x⊗y` is translated to `__oper__($SemigroupId, x, y)`. The meta flag `$IsMonoid` is set *iff* the programmer has provided some value for the unit.

If the emitted assertions hold, then the programmer-specified operator (and, optionally, a value for the unit) indeed form a semigroup (or monoid). Otherwise, the error is mapped to a readable message by the frontend (Fig. 2.18) and displayed to the user. In particular, the mapping distinguishes the following cases:

- missing declaration errors → *semigroup is undefined*
- wrong number of arguments → *wrong arity*
- assertion failure, line 118 → *operator commutativity check failed*
- assertion failure, line 119 → *operator associativity check failed*
- assertion failure, line 121 → *unit neutrality check failed*
- type error → *the type of the binary operator does not match the type of its unit*

2.4.8.2 Checking the filter. We first check that `filter` is an expression of type ***Set***`[T]` or ***Bool***. If `filter` is of type ***Set***`[T]`, then it is translated to `filterCheck(g, filter)`, where `filterCheck` is the identity function with a precondition.

```
123 function filterCheck(g: Set[Ref], filter: Set[T]): Set[T]
124 IF T ≡ Ref THEN
125   requires filter subset g
126 ELSE
127   requires filter subset domain(lift(g))
128 FI
129 { filter }
```

If `filter` is of type ***Bool***, we ensure that the filtering condition is type-correct by checking that the `filt` function (Sec. 2.4.5) is well-formed.

2.4.8.3 Checking the body. The type of the comprehension body must match the type of the binary operator. Since we have already checked the semigroup, checking that the unit is also of the same type (in case of a monoid) is not needed. Afterwards, type checking the body amounts to checking that the `snap` function (Sec. 2.4.4) is well-formed.

2.4.8.4 On finiteness. Our encoding is sound only for reasoning about comprehensions over *finite structures*. The reason for this limitation is that we aim to encode comprehensions in a way that is compatible with a type system in which every value is finite. In particular, the universe set for unbounded types cannot be represented in our type system, e. g. because the cardinality of this set would have to be of type `Int` which does not include the value of $+\infty$. Hence, under our assumption that each method's footprint is represented as a value of type `Set[Ref]`, all footprints must be finite, too.[29]

We established that our setting precludes infinite structures. We will now show that adding our comprehension axiomatization does not violate finiteness. For example, if our axioms were to specify that some set must contain all `Ref`s or all `Int`s, that would render our technique unsound. To prove finiteness of a comprehension instance of the form $\texttt{comp}[\mathcal{S}]\ n\ \{\!|\ \mathsf{e}^F_\sigma(n) \parallel \mathtt{f}\ |\!\}$, assume that `f` is a finite set and each instantiation of the body with a value from `f` yields a finite value. Let $\oplus$ be the step operator of $\mathcal{S}$.

Lemma: *Any finite $\oplus$-combination of finite values represents a finite value.*

The number of recursive steps in the complete evaluation (2.12) of our instance is bounded by the cardinality of `f`, which is by assumption finite. Each step involves at most one application of $\oplus$. Therefore, the evaluation of our comprehension instance is a finite $\oplus$-combination of various instantiations of the body term (which are by assumption finite). Hence, the evaluation of this comprehension instance represents a *finite value*. □

As a corollary, we apply the above argument for each comprehension instance (via a *depth-first* strategy, in case there are nested comprehension instances), showing that all comprehension instances, under our assumptions, represent finite values. Hence (since our type system does not permit for infinite sets) all the comprehension instances must be finite, and no additional checks are required.

2.5 EVALUATION

We evaluated our technique on a variety of challenging example programs taken from the literature, illustrating the specification and the verification aspects for different types of data structures, including the running examples of closely-related work.

29 In contrast, our logic does support methods with infinite footprints; e. g. consider a method the footprint of which is defined via `forall r:Ref :: acc(r.val)`, for some field `val`.

2.5.1 Experimental setup

We encoded each example manually into the Viper verification language [91]; we have introduces our chosen subset of Viper in Sec. 1.2. Although manual, our encoding of each example was performed methodically, simulating the translation that a frontend verification tool could perform. Each example consists of a common set of background definitions and axioms, along with a translation of the code of the example. The background definitions common to our examples are organized in separately-included li-

Table 2.1: Experimental results.

The 1st column is the benchmark name, 2nd: benchmark variant, 3rd: data structure class (Graph = direct access to nodes; Array = nodes accessed via integer indices), 4th: number of comprehension kinds in specification, t_a and t_p: Carbon verification time with Boogie's "type encoding A" and "type encoding P", resp. t_s: Silicon verification time (t.o. = 1,000 s timeout; * = fail due to incompleteness). Benchmark source is indicated in the last column.

Example	Variant	Structure	#Comps	t_a	t_p	t_s	Notes
ArraySum	Spec 1	Array	1	0.8	1.1	1.2	From Spec#
(Fig. 2.20)	Spec 2			0.9	0.9	1.4	— " —
	Fail 1			0.8	0.9	1.1	— " —
	Fail 2			0.9	0.9	1.3	— " —
Factorial	Success	-	1	0.8	1.0	0.8	From Spec#
(Fig. 2.21)	Fail 1			0.8	1.1	0.9	(numerical comprehension)
	Fail 2			0.8	0.8	0.8	— " —
MinSubArraySum	Success	Array	3	3.5	8.2	17.1	From Spec#
(Fig. 2.12)	Fail 1			3.4	5.5	68.7	(by Joseph B. Kadane [12])
	Fail 2			3.9	4.2	53.0	— " —
Coincidence-Count	Success	Array	2	3.5	4.0	*77.4	From Spec#
(Fig. 2.15)	Fail 1			3.7	3.2	-	(by Dijkstra and Feijen [14])
	Fail 2			3.1	3.3	-	— " —
Flip	One node	Graph	1	0.8	0.9	4.0	Singleton decompositions
(Fig. 2.9)	Two nodes			0.9	1.0	6.8	— " —
	Three nodes			0.9	1.1	19.7	— " —
	Smoke test			1.3	1.2	6.7	— " —
Merge	Success	Graph	1	2.2	2.3	t.o.	With reachability (Chap. 3)
(Fig. 2.10)	Fail 1			1.6	1.8	t.o.	— " —
	Fail 2			1.5	1.6	t.o.	— " —
Shortest-Path	Success	Graph	2	16.0	96.8	185.3	Modulo reachability
(Fig. 2.2)	Fail 1			6.3	13.5	2.1	— " —
	Fail 2			8.3	66.3	137.5	— " —

brary files, and we make heavy use of Viper's macros to improve the readability of our encoded examples.

Our examples are verified using Viper's standard Boogie-based [46] verifier (Carbon [68]), using two configurations for both available *type encodings*[30] (argument-based as well as predicate-based). We also attempted to verify each example using Viper's symbolic execution engine (Silicon [92]), although our encoding is generally less complete with this backend. Internally, both Boogie and Silicon use the Z3 SMT solver [47] for checking verification conditions. We indicate each backend's run times for each example in Tab. 2.1. The experiments were performed on a laptop running macOS, with a 2.6 GHz Intel Core i7 CPU, with Z3 [version 4.8.6 - 64 bit].

2.5.2 Experiments

In this chapter, we have presented a modular technique for specifying and verifying comprehensions. To evaluate the technique, we selected (1) the benchmarks from closest-related existing work and (2) a number of benchmarks that feature the most challenging aspects of reasoning about comprehensive properties of heap structures.

2.5.2.1 Overview. Tab. 2.1 gives an overview of our experiments. The first four examples correspond to the benchmarks of the Spec# comprehensions technique [52]. ArraySum is a simple algorithm for computing the sum of array values for which we considered two alternative specifications. Factorial is a recursive algorithm specified via a *numerical comprehension* that summarizes the properties of a set of integers rather than data structure nodes. MinSubArraySum and Coincidence-Count are the most challenging benchmarks from prior work Sec. 2.1.4.

The other examples involve graph structures that cannot be verified in Spec# but are supported in our technique. Flip demonstrates that the relevant data structure decompositions required for reasoning about field updates are indeed triggered automatically based on the nodes that are modified in the current footprint. Merge and Shortest-Path demonstrate that our technique is indeed modular, which requires cooperation between our comprehensive specifications and separation-logic framing.

2.5.2.2 Details. Each example program from Tab. 2.1 is represented by a series of benchmarks, including correctly specified and verified variants as well as those in which a bug has been planted either into the specification or the implementation (marked "Fail 1" and "Fail 2", resp. in the second column). The figures indicated in the three penultimate columns t_a, t_p, t_s are wall clock times for the verifier's response (in seconds), while using either of the two type encoding strategies in Boogie or the Silicon verification backend, resp. Note that Silicon reports encountered verification results in real time; hence, we

30 A brief preview of Boogie's type encoding features is given in [123] (e. g. in Fig. 3). For full details about the Boogie type system and its SMT encoding, refer to [58].

indicate the time until the final response in case of multiple failures in one benchmark.

ArraySum. This method computes the sum of values stored in the nodes of an array (Fig. 2.20). The main interest of this example is in the two alternative comprehensive specifications: It is sufficient to keep only one of the two loop invariants (either `/*I1*/`, in blue, or `/*I2*/`, in purple font).[31] The former invariant (Spec 1 in Tab. 2.1) summarizes the part of the array that is processed by the loop so far; to verify this invariant, the required filter decomposition is of the form `filt(0, n+1)` = `filt(0, n)` ⊎ {n}. Conversely, the latter invariant (Spec 2 in Tab. 2.1) expresses that the intermediate sum stored in `res` plus the sum of the remaining part of the array equals the sum over the entire array; verifying this invariant requires a *different* decomposition, namely, `filt(n, size(array))` = {n} ⊎ `filt(n+1, size(array))`, which we added as a manual assertion at the beginning of the loop body.

To test some failing scenarios, we planted two bugs (into the version of ArraySum specified with `/*I1*/`). Fail 1: we removed the increment of the loop counter `n` in Fig. 2.20. Fail 2: we planted an off-by-one error in the loop invariant's comprehension, changing it to `res` = 𝔰𝔲𝔪 *i* ⦃ `array[`*i*`].val` ‖ `0` ≤ *i* < `n-1` ⦄ (the strict inequality should say *i* < `n`).

```
field val: Int
define array lift(g,g)
define size(a) |domain(a)|

method arraySum(g: Set[Ref]) returns (res: Int)
  requires ACCESS_NODES(g, read)
  ensures ACCESS_NODES(g, read) && res = 𝔰𝔲𝔪 i ⦃ array[i].val ‖ 0 ≤ i < size(array) ⦄
{
  var n: Int := 0; res := 0
  while (n < size(array))
    invariant ACCESS_NODES(g, read) && 0 ≤ n ≤ size(array)
    /* I1 */ res = 𝔰𝔲𝔪 i ⦃ array[i].val ‖ 0 ≤ i < n ⦄
    /* I2 */ res + 𝔰𝔲𝔪 i ⦃ array[i].val ‖ n ≤ i < size(array) ⦄ =
                 𝔰𝔲𝔪 i ⦃ array[i].val ‖ 0 ≤ i < size(array) ⦄
  {
    res := res + array[n].val
    n := n+1
} }
```

Figure 2.20: Implementation of ArraySum and its two alternative specifications.

Factorial. This method recursively computes the factorial of an integer number (Fig. 2.21). This example shows how our technique can be retrofitted to support *numerical comprehensions*, i. e. those do not depend on the heap but summarize some (potentially un-

31 These alternative specifications were proposed in [43].

bounded) sets of integer values. Directly applying our technique is not possible here since the method is pure and does not have a footprint. However, one can still use our axioms to verify this example. To that end, the comprehension's filter is translated as usual, i. e. we get a set of integers of the form $\{n \mid 1 \leq n \leq x\}$.[32] We replace the snapshot in our translation of the $\mathfrak{prod}\, n\, \{\!|\, n \,\|\, 1 \leq n \leq x \,|\!\}$ comprehension with a map of the form $\{n \to n \mid 1 \leq n \leq x\}$ (from filtered integers to corresponding values of the body term). The rest of the translation is standard.

```
method factorial(x: Int) returns (res: Int)
  requires 0 ≤ x
  ensures res = prod n {| n || 1 ≤ n ≤ x |}
{
  if (x ≤ 2) { res := 1 }
  else { res := x * factorial(x-1) }
}
```

Figure 2.21: Implementation of Factorial and its comprehensive specifications.

In order to verify `factorial`, one needs to add the following assertions to the two corresponding branches: `filt(x) = {1}` and `filt(x) = filt(x-1) ⊎ {x}`. The former assertion is due to an incompleteness in Viper's axiomatization of sets. The latter assertion triggers the decomposition required to complete the proof.[33]

To test some failing scenarios, we planted two bugs into Factorial. Fail 1: we changed the branch condition in Fig. 2.21 to $x \leq 1$. Fail 2: we planted an off-by-one error in the comprehension, changing it to $\mathfrak{prod}\, n\, \{\!|\, n \,\|\, 1 \leq n < x \,|\!\}$ (note the strict inequality $n < x$).

MinSubArraySum. This method is the optimal implementation for computing the sum of a minimal sub-array of the input array. The algorithm's invariant maintains the local minimum in `y` and the global minimum (up until the current iteration) in `x`. To express the minimization criteria in the invariant, we use *nested comprehensions* over the (dependent) lower bound and the (independent) upper bound of the sub-array (the lower bound cannot be greater than the upper). Hence, the benchmark demonstrates the support of nested, dependent comprehensions in our technique. Note that the verification required one manual assertion to trigger our distributivity axiom[34]; see the end of Sec. 2.4.6 for a discussion of distributivity. More details on this example were presented in Sec. 2.3.1.

32 Unlike our standard set comprehensions (Sec. 2.4.8.4), numerical set comprehensions require cardinality checking; e. g. if the filtering condition is an expression $Q(n)$, then $f = \{n \mid Q(n)\}$ must be a *finite set*. Automatically checking set finiteness is an open problem.

33 Note that our technique cannot automatically trigger this decomposition as there is no access to any parts of the heap. Automatically triggering filter decompositions for numerical comprehensions is future work.

34 The assertion is **assert** $n_o > 0 \Rightarrow \mathtt{min}(R, F_o) = \mathtt{min}(R_o, F_o) + \mathtt{array}[n_o]$; see Sec. 2.3.1.

To test some failing scenarios, we planted two bugs into MinSubArraySum. Fail 1: we replaced the range of the outermost **min**-comprehension's filter from `1 .. n+1` to `0 .. n` (both in the loop invariant and the postcondition). Fail 2: we replaced the second branch condition of Fig. 2.12 from $y < x$ to $x < y$.

This example demonstrates that our encoding works best with Viper's verification condition generator, especially with Boogie type encoding A. In contrast, symbolic execution performed significantly slower, especially in detecting the failures (Tab. 2.1). Optimizing our encoding for symbolic execution is future work.

Coincidence-Count. This method is the optimal implementation for computing the number of coinciding values in two sorted sets (i. e. disjoint, sorted arrays of unique elements). The method maintains a comprehensive invariant that requires multiple *independent* iterated variables. Instead of using nested comprehensions, our technique encodes independent iterated variables as tuples. This leads to a simpler proof due to fewer levels of indirection (compared to an encoding of nested comprehensions). More details on this example were presented in Sec. 2.3.2. The manual assertions required to complete the proof directly corresponds to the three decompositions depicted in Fig. 2.17.

To test some failing scenarios, we planted two bugs into Coincidence-Count. Fail 1: we decreased by one the upper bounds for both iterated variables of the **sum**-comprehension. Fail 2: we dropped `n:=n+1` in the last branch of Fig. 2.15.

While verification condition generation performed well on this example, symbolic execution reported a verification error after ca. 80 seconds (Tab. 2.1). Note that the two backends of Viper rely upon different (inherently incomplete) set axiomatizations, which may be the cause of the problem. However, we did not investigate the real cause; the first step would involve tuning our encoding for symbolic execution to make verification debugging practically feasible. We will discuss verification debugging in Chap. 4.

Flip. This series of synthetic methods demonstrates the interplay of separation-logic framing with comprehensive specifications in presence of field update operations. The first three methods feature updates to one, two, and three (non-aliasing) nodes, resp. Each field update is accompanied with a singleton decomposition, i. e. it is asserted that the current structure's footprint (which may contain *any* number of nodes in addition to the ones being modified) can be represented via a disjoint union of the currently modified node and the set of all other nodes of this footprint. These assertions are generated systematically using the translation rule for field updates.

The last benchmark in the series is a *smoke test*, i. e. we add **`assert false`** as the very last statement of the body of each of the three methods (including the one presented in Fig. 2.9) and measure the time it takes the verifier to report failure.

We observed that the performance of symbolic execution drops rapidly with each additional field update operation (Tab. 2.1). Such degradation might be caused by a suboptimal quantifier instantiation strategy in the encoding of separation logic framing;

investigating the real cause is future work. In contrast, there was no noticeable performance penalty for verification condition generation.

MERGE. The benchmark demonstrates reasoning about comprehensions in presence of complex heap transformations, i. e. those over *subgraphs* (cf. the recursive call in the then-branch) and over *individual nodes* (cf. the field update in the else-branch). We omitted the reachability part of the specification of MERGE; this program is also the running example of our technique of Chap. 3 in which we focus precisely on the problem of modularly verifying reachability properties, in particular, acyclicity. The decompositions needed for completing the proof are $\texttt{filt}(\mathfrak{g}) = \texttt{filt}(\mathfrak{g} \setminus \mathfrak{h}) \uplus \texttt{filt}(\mathfrak{h})$, where $\mathfrak{h} \subset \mathfrak{g}$ is the callee footprint in the then-branch, and $\texttt{filt}(\mathfrak{g}) = \texttt{filt}(\mathfrak{g} \setminus \{\texttt{l}\}) \uplus \texttt{filt}(\{\texttt{l}\})$, where `l` is the node modified in the else-branch. More details on this example were presented in Sec. 2.2.5.

To test some failing scenarios, we planted two bugs into MERGE. Fail 1: we replaced `-1` with `+1` in the postcondition of Fig. 2.10. Fail 2: we inverted the branch condition.

We could not verify via example with symbolic execution as the verifier timed out after 1,000 seconds on all runs. This is expected as our reachability verification technique Chap. 3 is currently not optimized for Viper's symbolic execution backend.

SHORTEST-PATH. This method implements a depth-first search strategy for finding the shortest[35] path in a DAG. Refer to Sec. 2.2.1.1 for a detailed overview of this program.

To verify the comprehensive specifications, we assumed the acyclicity of the heap structure; this allowed us to verify the rest of the specifications via symbolic execution which practically cannot handle proof obligations with reachability.[36] Recall that the $\mathfrak{h} \subset \mathfrak{g}$ is the callee footprint and `node` is the currently processed node; `path_a` and `path_b` are the shortest paths to the `target` node from `node.left` and `node.right`, resp. The decompositions required for completing the proof are $\mathfrak{g} = \mathfrak{h} \uplus \{\texttt{node}\}$ for the set-based comprehension $(\mathfrak{sum}\, n \,\{\!|\, n.\texttt{val} \parallel n \in \mathfrak{g} \,|\!\})$ and $\texttt{filt}(\texttt{path}) = \texttt{filt}(\texttt{path\_a}) \uplus \{\texttt{node}\}$ and $\texttt{filt}(\texttt{path}) = \texttt{filt}(\texttt{path\_b}) \uplus \{\texttt{node}\}$ for the sequence-based comprehension $(\mathfrak{sum}\, n \,\{\!|\, \texttt{abs}(n.\texttt{val}) \parallel n \in \texttt{path} \,|\!\})$.

To test some failures, we planted two bugs into SHORTEST-PATH. Fail 1: we replaced the postcondition $\texttt{reachable} \Rightarrow \texttt{abs(cost)} \leq \mathfrak{sum}\, n \,\{\!|\, \texttt{abs}(n.\texttt{val}) \parallel n \in \mathfrak{g} \,|\!\}$ with $\texttt{reachable} \Rightarrow \texttt{cost} \leq \mathfrak{sum}\, n \,\{\!|\, n.\texttt{val} \parallel n \in \mathfrak{g} \,|\!\}$. A counterexample to this specification is a DAG $\mathfrak{g}$ with three nodes $a = \texttt{node}$, $b = \texttt{target}$, and c, storing the values e. g. `1`, `1`,`-10` , resp. and two heap edges $a.\texttt{left} = b$ and $a.\texttt{right} = c$. Then $\texttt{path} = [a, b]$ and $\texttt{cost} = a.\texttt{val} + b.\texttt{val} = 2$ which is greater than $\texttt{-8} = \mathfrak{sum}\, n \,\{\!|\, n.\texttt{val} \parallel n \in \mathfrak{g} \,|\!\}$. Fail 2: we replaced $\leq$ with $>$ in the first branch condition of Fig. 2.2.

Viper's verification condition-based backend performed significantly better on this example, especially with Boogie type encoding A. Symbolic execution also produced ex-

35 Shortest in terms of the cost, as explained next.

36 However, we independently verified the acyclicity invariant using verification condition generation, which in this case is straightforward as the method `shortestPath` only traverses but does not *modify* the heap.

pected results eventually, although much slower in scenarios labeled Success and Fail 2 (Tab. 2.1); however, we leave as future work understanding why symbolic execution performed the fastest in Fail 1.

2.5.3 Results

Our experiments show that comprehensive specifications enabled by our technique are amenable to *modular, SMT-based verification* for a broad class of heap-transforming programs. Our technique is a lightweight extension of separation logic; hence, it is easy to extend basic specifications (e. g. memory safety) with comprehensive specifications of functional correctness. Our technique automates the *verification* of comprehensive properties; devising suitable program invariants and specifications is conceptually still challenging as the correct specification of highly-optimized algorithms is sometimes more difficult to devise than the actual implementation (cf. Fig. 2.12).

ALGEBRAIC PROPERTIES OF COMPREHENSIONS. Our set-based comprehension reasoning technique is agnostic to the ordering of nodes in a data structure. In those cases in which the ordering matters, it can be encoded using an extra layer of abstraction, which we call the lifted representation of a data structure. MinSubArraySum and Coincidence Count demonstrate that our set-based view is general enough to support multidimensional comprehensions in combination with the lifted representation of arrays.

The fact that set-based comprehensions do not rely on the order of heap nodes enables straightforward applications of this technique to problems about graph structures with arbitrary in- and outdegrees. In particular, we used this technique to modularly specify Shortest-Path and Merge. These methods operate on heap-allocated *directed acyclic graphs* and cannot be easily specified in a sequential view of the Spec# technique.

INTERACTIVE VERIFICATION SESSIONS. Both time till success and time till failure are essential characteristics of a verification technique because programmers typically develop the implementation and the specification of each method *iteratively*, refining the code step-by-step, which may involve deliberate (temporary) violations or accidental bugs that must be detected and reported as quickly as possible. The experiments show that our technique's time till failure is on par with verification times for correctly specified versions of the corresponding benchmarks.

SYMBOLIC EXECUTION. Our encoding performed much slower (and less stable) while verifying the examples via symbolic execution. To optimize the encoding, one could investigate whether the problem is related to *matching loops* of quantifier instantiations. While our technique precludes obvious matching loops caused by our own axioms, e. g. by carefully selecting the triggering patterns and employing synonym functions (Sec. 2.4.7), matching loops may still occur due to the interplay of comprehension ax-

ioms, the set axiomatization, and the encoding of separation logic used internally by the verifier. Investigating matching loops is possible e. g. via the *Z3 Axiom Profiler* [108].

2.6 DISCUSSION

In this section, we summarize the strengths (Sec. 2.6.1) and limitations (Sec. 2.6.2) of our technique for compositional reasoning about comprehensive properties of heap-transforming programs, and conclude the chapter (Sec. 2.6.3).

2.6.1 Strengths

MODULARITY. Our reasoning technique for reachability properties is modular. For each method, the programmer specifies only local comprehensive properties, within the method's footprint. Our technique enables precise (first-order) framing of comprehensions, i. e. comprehensive properties of the callee footprint are automatically extended to the (larger) client footprint.

GENERALITY Our technique provides a general framework for reasoning about foldings of heap-dependent sub-expressions via *commutative*, *associative* binary operators. As a prerequisite, we require the programmer to confine accessing the heap via direct node references (which is common for graph structures) or via a suitable lifted representation. The lifted representation enables accessing heap nodes via some injective identifiers (e. g. array name, array index pairs). Programmers working on the level of an intermediate verification language typically use these abstractions already. For example, disjoint array partitions are typically encoded into Viper using an injective `loc` function that maps its two arguments (an array and an integer index) to a node reference; this abstraction is a special case of our `lift` function (which is more general in that e. g. it is suitable for encoding families of potentially-overlapping arrays).

SEPARATION LOGIC. Our technique integrates into first-order separation logic, enabling reasoning about comprehensions and other properties in a uniform way, verifying concurrent programs, and automating our technique via existing separation logic verifiers.

We will discuss generalizations of our techniques in more detail in Sec. 5.1.

2.6.2 Limitations

ENTAILMENTS. In addition to relating the properties of multiple program states, the verification of stateful programs often requires proving non-trivial entailments about

the properties of *individual states*. Conceptually, these are reasoning steps that transform the existing information about comprehensive properties from one representation to another. For example, if it is known that each node of an array stores a positive value, then we must be able to learn that the minimal and the maximal values, as well as the sum and the product of all the array elements, are all positive, too. Despite their intuitive simplicity, these entailments generally require *induction*, which precludes a first-order SMT solver from establishing them automatically.

There are two main avenues for entailments automation in our setting that are presented in this chapter. First, deductive verification requires the programmer to supply sufficiently precise loop invariants (and postconditions of recursive methods) that serve as *induction hypotheses* for some of the entailment proofs. Second, we identify and efficiently axiomatize some classes of inductive properties that are frequently needed for entailment automation, namely, *monotonicity* and *distributivity*. Based on our case studies, these properties are sufficient. We leave a thorough taxonomy of essential inductive lemmas about comprehensions as future work.

DECOMPOSITIONS It is hard to identify the best possible data structure decomposition, i. e. the one that would lead to a successful proof, from multiple alternatives. An obvious solution is to consider all possibilities, maximizing the completeness of the technique. However, this is practically infeasible as there are generally unboundedly many decompositions; ultimately, reasoning about the value of each individual node of the structure would break the level of abstraction that comprehensions grant. The next alternative is to define a family of *characteristic* decompositions that each operation can trigger. In case of a single-node update, i. e. an operation of the form `loc(a,k).val:=` ... for some array `a` and index `k` s.t. $\texttt{0} \leq k < \texttt{len(a)}$, it is intuitively clear that a useful decomposition to consider splits the array range into three segments, namely, $[\texttt{0} \mathinner{..} k)$, the singleton $[k \mathinner{..} k{+}1)$, and $[k{+}1 \mathinner{..} \texttt{len(a)})$. Although this three-way decomposition is very basic, it may still lead to a combinatorial explosion of the number of considered sub-ranges, e. g. in a scenario in which a method *subsequently* updates multiple potentially different nodes of the same array — each such operation would split one of the (currently, atomic) sub-ranges into up to three *smaller* sub-ranges, etc.

To mitigate the problem, we exploit the set-based nature of our technique. For instance, rather than splitting the integer range into three sub-ranges (in the above example) each operation triggers a *two-way decomposition* based on its footprint. Hence, in a method whose footprint is the node set $\mathfrak{g}$, the field update `x.val := v` triggers a decomposition of the shape $\mathfrak{g} = (\mathfrak{g} \setminus \{\texttt{x}\}) \uplus \{\texttt{x}\}$, while a method call whose footprint is $\mathfrak{h} \subset \mathfrak{g}$ triggers a decomposition of the shape $\mathfrak{g} = (\mathfrak{g} \setminus \mathfrak{h}) \uplus \mathfrak{h}$. However, node-based decompositions are generally not sufficient for reasoning about the most general abstractions that our technique supports. For example, the two-dimensional comprehension that represents the sum of all elements of a matrix. Matrix decompositions are naturally defined based on lower and upper bounds for the selected rows and columns, which in our tech-

nique would translate into filtering functions that define sets of permitted index pairs (i, j). To that end, we consider *multidimensional filter decompositions*, like Fig. 2.17. Automatically triggering the relevant multidimensional decomposition is future work.

NUMERICAL COMPREHENSIONS. Our set comprehensions are defined to express rich properties of heap structures which are guaranteed to have finite footprints. We nonetheless demonstrate that our technique can be retrofitted to support numerical comprehensions, i. e. the product of all integers between some values i and j (cf. FACTORIAL in Sec. 2.5). However, numerical comprehensions impose an additional challenge as they are generally *not* guaranteed to be finite. In future, we plan to extend our technique to support numerical comprehensions, perhaps following Dafny's template-based approach [57] that requires supported comprehensions to follow a known template for which automatic finiteness checking is possible.

ISC-BASED SETTING. We assumed a setting in which all method footprints are explicitly specified via nodesets. On the one hand, we argue that this general footprint representation supports diverse classes of heap structures, e. g. arrays and graphs. On the other hand, one could potentially lift this requirement if the nodeset-based footprint representations could be *encoded* or *inferred* rather that specified. Theoretically, this should be possible in a logic that supports *permission introspection*, which is the case e. g. in Viper [91]. However, investigating such a setting is future work.

We will discuss further generalizations of our technique in Sec. 5.1.

2.6.3 Conclusion

In this chapter, we have developed a compositional technique for reasoning about set comprehensions in separation logic. Our solution solves the open problem of modular reasoning about comprehensive properties of arbitrary heap structures. In particular, our technique lifts the requirement from the state of the art that the data structures must be ordered. Another novelty of our work is that we support two classes of comprehensions: those based on commutative monoids and (more general) commutative semigroups that may not have a unit. We allow the programmer to define custom binary operators and use them in comprehensions.

3 REACHABILITY

In this chapter, we discuss reasoning about reachability properties of heap-transforming programs.[1] A concrete discussion about reachability requires fixing some *imperative* programming language. We defined the programming language for our setting in Sec. 1.2.1; recall that our language is *strongly, statically typed*; otherwise, an expression e. g. $x.\mathtt{f} = y$ for two object references x and y cannot be unequivocally interpreted as a *heap edge* of the form (x, y), and reachability has no meaning.

The intuitive meaning of reachability corresponds to the existence or the absence of directed *paths* of reference fields that connect *nodes*, i. e. (dynamically allocated) heap objects. For example, the successful execution of the assignment operation `m.left.right := n`, where `m` and `n` are roots of disjoint and priorly disconnected binary trees, will create a heap path of the form `m` ... `n`. This path consists of two heap edges: (`m`, `m.left`) through the reference field `left` and (`m.left`, `n`) through the reference field `right` (Fig. 3.1).

In complex structures, multiple paths may exist between the same (ordered) pair of nodes. For instance, if instead of trees in the example above we considered two possibly overlapping *DAGs* (directed acyclic graphs) rooted in `m` and `n`, then the conditions, e. g. `m.right.left = n` and `m.left.right = n` could both hold in the same state. To indicate that through the discussed reference fields there exists *some* path connecting an (ordered) pair of nodes `m` and `n`, we will use the predicate notation $\mathtt{R}^F(\mathtt{m},\mathtt{n})$, where $F = \{\mathtt{left}, \mathtt{right}\}$ is the field set; we drop this parameter when it is clear from context.

(a) (b)

Figure 3.1: Illustration of the possible effects of a field assignment operation.

The heap configuration in presence of a field assignment operation `m.left.right := n`. The newly created heap edge is dashed. (a) `m` and `n` are roots of disjoint trees; the assignment created a *new* path `m` ... `n` that did not exist before. (b) `m` and `n` are roots of possibly overlapping DAGs; the assignment created an *alternative* path `m` ... `n`.

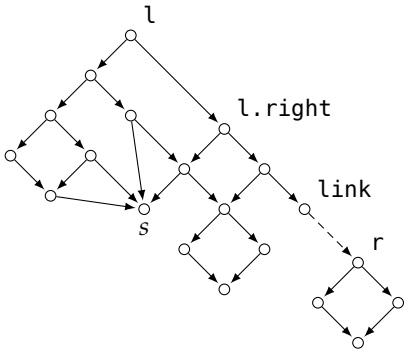
1 This chapter is based on the paper "Modular verification of heap reachability properties in separation logic" [112].

MOTIVATING EXAMPLE. Programmers employ the notion of reachability to informally describe heap structures and understand algorithms that operate on them. We will now show that reachability also naturally enables *formal specification* of programs.

Consider the method `merge` that merges two DAG structures rooted in `l` and `r` into a single DAG rooted in `l` (Fig. 3.2a).[2] The method takes the references `l` and `r` to the roots of the corresponding DAGs and returns the reference `link` to the (only) updated node. Fig. 3.2b illustrates a typical heap transformation while running `merge`; solid arrows depict preexisting heap edges and the dashed arrow — the newly created one.

```
field left:Ref       field right:Ref
method merge(l:Ref, r:Ref) returns (link:Ref)
 requires Structures rooted in l and r are disjoint.
          Structures rooted in l and r are acyclic.
 ensures Structure rooted in l is acyclic.
         l reaches link, which reaches r.
{
  if (l.right != null) {
    link := merge(l.right, r)
  } else {
    l.right := r; link := l
} }
```

(a) Annotated implementation



(b) Example scenario

Figure 3.2: Specifying reachability properties of complex data structures.

The first precondition of `merge` requires that the two structures reachable from `l` and `r` are *disjoint* in the heap. This is a *reachability property* that can be expressed in first-order logic with reachability predicates as follows:

$$\forall n{:}\textbf{\textit{Ref}} \bullet \neg \mathsf{R}(\mathtt{l}, n) \vee \neg \mathsf{R}(\mathtt{r}, n)$$

This formula says that each node *must not be reachable* from at least one of the two, `l` or `r`; i. e. the corresponding structures are disjoint.

The second precondition of `merge` requires that the two structures reachable from `l` and `r` are *acyclic*. Acyclicity is also a reachability property; for a heap structure rooted in `r`, one can express acyclicity as follows:

$$\forall x, y{:}\textbf{\textit{Ref}} \bullet x \neq y \wedge \mathsf{R}(\mathtt{r}, x) \wedge \mathsf{R}(\mathtt{r}, y) \implies \neg \mathsf{R}(x, y) \vee \neg \mathsf{R}(y, x)$$
$$\forall x{:}\textbf{\textit{Ref}} \bullet \mathsf{R}(\mathtt{r}, x) \implies x.\mathtt{left} \neq x \wedge x.\mathtt{right} \neq x$$

2 We previously considered this example as a benchmark in Chap. 2.

The 1st formula above says that two distinct nodes of the structure rooted in `r` *cannot be mutually reachable* from one another; i. e. this structure is acyclic. The 2nd formula says that the nodes of the same structure cannot have *self-edges*; i. e. `l.right` $\neq$ `l` in Fig. 3.2a.[3]

The first postcondition of `merge` ensures the acyclicity of the *merged* structure (which is rooted in `l`; cf. Fig. 3.2b). The second postcondition ensures that the newly created heap paths pass through `link`; i. e. there exists a heap path of the form `l` ... `link` ... `r`. This fact can also be simply expressed via reachability predicates: $\mathtt{R}(\mathtt{l}, \mathtt{link}) \wedge \mathtt{R}(\mathtt{link}, \mathtt{r})$. A possible refinement of this postcondition could specify that these paths follow only the `right` fields: $\mathtt{R}^{\{\mathtt{right}\}}(\mathtt{l}, \mathtt{link}) \wedge \mathtt{R}^{\{\mathtt{right}\}}(\mathtt{link}, \mathtt{r})$ if this detail is important to the client. Thus, the programmer can choose the right level of abstraction for their specifications.

Specifying (1) disjoint heap partitions, (2) acyclicity, or (3) the existence (and absence) of heap paths connecting referenced nodes in unbounded structures is generally *not possible without* reachability predicates. In particular, *recursive predicates* from separation logic [24, 26] can define acyclic and disjoint heap structures but *preclude sharing* (cf. s in Fig. 3.2b). The goal of this chapter is to overcome this limitation by enriching separation logic with reachability predicates.

OUTLINE. First, we explain the fundamental problems of reachability reasoning (Sec. 3.1). Second, we demonstrate the strengths and limitations of existing solutions (Sec. 3.2). Third, we introduce the notion of *local reachability* (Sec. 3.3). We then present our solution for reasoning in presence of individual field updates in acyclic graphs (Sec. 3.4) and method calls (Sec. 3.5). We then extend of our technique to potentially cyclic *o–1-path graphs* (Sec. 3.6). Next, we describe the logical encoding (Sec. 3.7), evaluate our technique (Sec. 3.8), and discuss the metatheory (Sec. 3.9). Finally, we summarize the results (Sec. 3.10).

3.1 THE PROBLEM

Reasoning with reachability information sets two main problems. (1) Automation: general graph reachability is *not supported* by SMT solvers that are typically used in automated program verification; hence, automated reasoning about heap reachability is challenging. (2) Modularity: reachability properties of the entire heap are potentially affected by *any* operation, which complicates framing. In this section, we first introduce the components of reachability information (Sec. 3.1.1). We then illustrate how the two problems above complicate verification (Sec. 3.1.2). Finally, we explain the depth of the general problem of modular reasoning about heap paths (Sec. 3.1.3).

3 We consider a *reflexive* reachability relation; hence $\mathtt{R}(x, x)$ holds for any x.

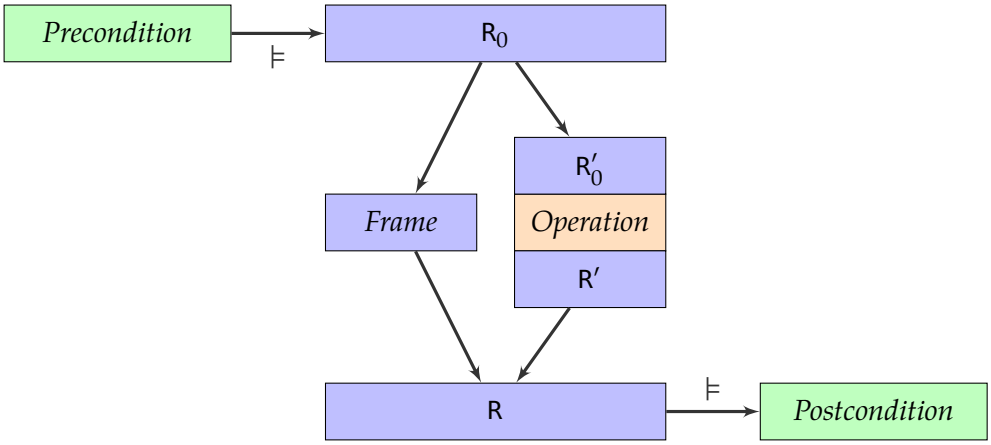


Figure 3.3: Flow of reachability information around a heap operation.

The original source of information (*Precondition* in the pre-state of our *Operation*, in the orange box) and the ultimate target (*Postcondition* in the post-state) are two assertions in our specification language. In the pre-state, *reachability information* R_0 is inferred from the precondition based on available *reachability inference rules*; this allows satisfying the precondition and *framing* the information that is not relevant to the operation. In the post-state, reachability information from the postcondition and from the frame form new reachability information that is then used to verify the operation's client. Lime boxes depict the specification of our operation; purple boxes depict modular reachability information; $\models$ depict the direction of logical entailments.

3.1.1 Components of reachability information

Lossless reasoning about the reachability relation involves combining multiple sources of information and targeting multiple proof obligations. We illustrate these in the information flow diagram of Fig. 3.3. The lime boxes represent the heap configurations before and after invoking a heap-transforming *Operation* (the central, orange box). The purple boxes represent information about the reachability relation: R_0 and R summarize the facts about reachability before and after the invocation, while R'_0 and R' are the operation's precondition and the postcondition, resp. Finally, the part of reachability information that is both unused and unchanged by the operation comprises its *Frame*.

Next, we will discuss the application of reachability information for reasoning about heap-transforming programs. First, we explain the two orthogonal problems: that of entailment automation (Sec. 3.1.1.1) and reachability specification (Sec. 3.1.1.2). We then illustrate the problems with a concrete example (Sec. 3.1.1.3).

*3.1.1.1 **Automating entailments.*** The first step is to verify *entailments* concerning the reachability relation ($\models$ in Fig. 3.3). In particular, reachability information must be *enclosed*, e. g. the knowledge that x reaches z and that z reaches y implies that x reaches y, due to transitivity of paths. In an automatic setting, applying the transitivity rule is problem-

atic because the number of possible applications may be infinite while the number of *necessary* applications may be statically unknown.[4]

Another problem is that the reachability relation R^F summarizes *different kinds* of information. For example, the condition $a.\texttt{next} = b$ implies that the reachability predicate $\mathsf{R}^F(a,b)$ must hold (where F can be any field set that includes `next`). Naturally, the reachability relation can also help inferring less abstract properties; e. g. the condition $\neg\mathsf{R}^F(b,a) \wedge \forall n \bullet \mathsf{R}^F(b,n) \implies \mathsf{R}^F(a,n)$ where $F = \{\texttt{left}, \texttt{right}\}$ implies $a \neq b$ (non-aliasing) *and* $a.\texttt{left} = b \vee a.\texttt{right} = b$ (existence of a heap edge). However, not all valid conversions of reachability information can be performed automatically as proving the corresponding entailments is generally undecidable.

3.1.1.2 Reachability specification. The second step is to infer the reachability relation in the *current state*, e. g. the post-state of *Operation*; see the purple boxes in Fig. 3.3. General heap operations, such as method calls and field updates, modify some heap properties while preserving others. The preserved heap properties of an operation are called its *frame*. For example, the frame of the operation `x.left := null` includes the entire relation $\mathsf{R}^{\{\texttt{right}\}}$ as all the paths expressed by this relation pass exclusively through the field `right` and are not affected by the changes of `left`.

The precondition and the postcondition of *Operation* are denoted as R'_0 and R', resp. These conditions determine the effect that *Operation* has on the global heap. There is a degree of freedom in splitting pre-state reachability information, R_0, into the frame and R'_0 and joining the frame and R' into post-state reachability information, R. On the one hand, the frame could be completely eliminated by conjoining it to the operation's specification, but that would render the specification unusable in a modular setting in which operations can be invoked many times in different contexts. On the other hand, deriving *precise* reachability specifications is a fundamental problem. In their general form, precise specifications are not unknown for arbitrary heap operations and problematic even for basic operations such as field updates, as will be explained in Sec. 3.2.1.

3.1.1.3 Example. Let us illustrate how the components of reachability information from Fig. 3.3 can be used in reasoning. Consider the creation of a new heap edge (u,v) in a heap configuration with (some number of) singly-linked lists. This operation changes the reachability relation according to the following rule:

$$\forall x,y \bullet \mathsf{R}(x,y) \;=\; \mathsf{R}_0(x,y) \vee (\mathsf{R}_0(x,u) \wedge \mathsf{R}_0(v,y))$$

This formula says that a path $x \ldots y$ exists in the current state *iff* it either already existed in the operation's pre-state (denoted $_0$) or the new edge has connected up a new path. In a setting in which singly-linked lists are *the only* possible heap structures, the above

4 It is unknown *what* is unknown; e. g., automatically checking that the predicate $\mathsf{R}(x,y)$ *can* be validated (or refuted) in a given program state is generally undecidable, just like our original verification problem.

formula can be used as a *precise reachability specification* for edge creation. This operation's frame is trivial because the above formula carries complete information about preserved heap paths. Alternatively, one could specify the frame as, e. g.:

$$\forall x, y, \bullet \neg (\mathsf{R}_0(x, u) \wedge \mathsf{R}_0(v, y)) \implies (\mathsf{R}(x, y) = \mathsf{R}_0(x, y))$$

This formula says that the paths $x \dots y$ that do not pass (in the pre-state) through the new edge are *preserved*, i. e. they coincide in both states. With this frame, a weaker specification of the edge creation operation is sufficient, i. e. only the *newly created paths* should be specified: $\forall x, y, \bullet \mathsf{R}_0(x, u) \wedge \mathsf{R}_0(v, y) \implies \mathsf{R}(x, y)$. Note that together the refined specification and the frame define the new reachability relation (in terms of the old relation) *for all pairs x and y*, allowing for lossless reasoning about our operation. It is important to identify, for each operation, a natural split of reachability information in the operation's specification and that of its frame.

3.1.2 Modular reachability specifications

Modular reasoning about reachability adds another dimension of complexity. In a modular setting, verification of each method may depend only on the information in its specification and must be *in*dependent of the global environment, e. g. static variables and concurrent threads. However, it is hard to manage reachability information modularly.

3.1.2.1 Modularizing reachability properties. Some reachability properties cannot be directly specified in a modular setting. For example, consider the method `findAndReplace`,[5] that operates on a segment $[a \dots b]$ along the `next` fields of a singly-linked list. Intuitively, this method should maintain the following acyclicity invariant:

$$\forall x \bullet \mathsf{R}(a, x) \wedge \mathsf{R}(x, b) \wedge x.\texttt{next} \neq \texttt{null} \implies \neg \mathsf{R}(x.\texttt{next}, x) \tag{3.1}$$

This formula says that if x belongs to the segment $[a \dots b]$ and has a successor node $x.\texttt{next}$, then this successor *cannot* lead back to x via a path of the form $x.\texttt{next} \dots x$.

The above property is problematic because it *cannot be modularly verified*. The example heap configuration of Fig. 3.4 shows a scenario in which the segment that `findAndReplace` operates on consists of the two nodes a and b, and the latter reaches an external node, c.

While modularly verifying `findAndReplace`, one *cannot rely* on the information outside the local segment $[a \dots b]$. However, the above acyclicity property relies on the R relation that is not bounded by the local heap fragment. In particular, even if the condition (3.1) holds before `findAndReplace` (*Thread* in Fig. 3.4) is invoked, acyclicity might be violated due to the concurrent context of this thread (*Context* in Fig. 3.4). Therefore,

5 The implementation of `findAndReplace` is not relevant for the discussion.

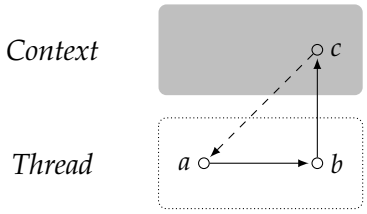


Figure 3.4: Creation of a non-local heap cycle.

The list segment $a \dots b$ comprises the *local* part of the heap that the current method (and *Thread*) operate on. c is a reachable *non-local* node (in *Context*). After the current method is invoked, the context creates a new (dashed) heap edge (c, a) that completes a heap cycle and thus violates acyclicity in the global heap.

there does not exist an implementation of `findAndReplace`, operating exclusively within $[a \dots b]$, that ensures the invariant (3.1); i. e. it is a *non-local property*.

LOCALIZED REACHABILITY. To support modularity, one could relax the non-local properties in a method's specification, constraining the heap configuration only *within the bounds* that this method operates on. Naturally, each method specification should have a pragmatic purpose, e. g. the acyclicity in the example above helps proving that `findAndReplace` eventually terminates. However, a weaker invariant, i. e. no heap cycles exist *within our local heap segment*, would suffice; other heap fragments, including those *reachable* from the local segment, may contain cycles without causing a problem. For instance, $a \to b \to c \to a$ in Fig. 3.4 may be permitted as c is not part of the current segment, while e. g. $a \to b \to a$ must not be permitted.

Therefore, the problem of modular reasoning with reachability properties requires a suitable *specification technique* that would allow expressing the existence (and the absence) of heap paths *within certain locality*, e. g. a particular heap fragment.

LOCAL VS. MODULAR. There is a subtle difference between *local* vs. *modular* specifications. While local specifications are always modular, global (i. e. non-local) information can sometimes comprise modular specifications, too.

We define a method's *local specifications* as those constraining *exclusively* the heap fragment that this method operates on. For example, the property $x.\texttt{next} \neq y$ of a method that operates on the two nodes `x` and `y` is a local specification. Conversely, the `R` relation carries *global reachability information*; e. g. if $\neg\texttt{R}(x, y)$ holds for some nodes x and y, then the path $x \dots y$ does not exist in the entire heap, not just within the local heap fragment.

Generally, a method cannot rely on global reachability information. For example, if $x.\texttt{next} = z$, where z is a *non-local* node, i. e. our method does not have permissions to access the fields of z, then our method *cannot rely* on the fact $\neg\texttt{R}(x, y)$ in its modular specification as other threads might create a path e. g. $x \dots z \dots y$ at any point in time.[6]

In this thesis, we focus on *local reasoning*, i. e. reasoning with exclusively local specifications. Not only do local specifications enable modular reasoning, they can be integrated into separation logic, supporting verification of concurrent programs.[7] Interest-

6 A special case in which one *can* rely on global unreachability is memory allocation. A freshly allocated node does not have any outgoing heap paths as none of its reference fields have been initialized yet.

7 In the following, we use the terms *local reasoning* and *modular reasoning* interchangeably.

ingly, modular reasoning about global reachability is possible in a restricted setting that we will discuss in Sec. 3.2.3.

3.1.2.2 Framing heap paths. We have established that reasoning about global heap reachability is not possible in our setting. Yet, modular verification requires *framing* [22]: a mechanism for adapting between two kinds of information, namely, the properties specified for the heap fragments that the *callee* and its *client* operate on. It is challenging to precisely frame information about heap paths, as we explain next.[8]

To illustrate, we revisit our motivating example of Fig. 3.2a. Recall that the method `merge` operates recursively on a heap structure comprised of two disjoint DAGs rooted in `l` and `r`, resp. An invocation of `merge` checks whether the `l` node is the *rightmost*.

ESTABLISHING NEW REACHABILITY INFORMATION. If there does not exist a right successor to `l`, i. e. it is indeed the rightmost node, then the heap is modified by creating a single new edge that attaches `l` to `r`, merging the two DAGs. Note that this operation creates a (potentially) unbounded number of *new heap paths* because all heap nodes reaching `l` are now (transitively) connected to all nodes reachable from `r`.

The problem of reachability framing is to *propagate* the reachability information, established by the current invocation, back to its *client*.

PROPAGATING REACHABILITY INFORMATION. If there exists a right successor to `l`, then a recursive call is performed. Notice that the *callee*, i. e. a new invocation of our method `merge`, will operate on a *different* heap fragment than the client: Since the callee parameters are `l.right` and `r` for `l` and `r`, resp., and the nodes reachable from `l.right` are a *proper subset* of the nodes of the DAG rooted in `l`, some nodes of the original DAG will be *unreachable* for the callee (cf. Fig. 3.2b). We call these the *frame nodes*.

To verify this branch, one must satisfy two proof obligations: the precondition of the callee (cf. $\mathtt{R}_0 \rightarrow \mathtt{R}'_0$ step in Fig. 3.3) and the (ultimate) postcondition of the client (cf. $\mathtt{R}' \rightarrow \mathtt{R}$ step in Fig. 3.3). Since `merge` is specified via reachability (and the set of frame nodes is non-empty), both of these proof obligations require *framing of reachability information*.

In the former case, framing involves the propagation of reachability information from a *larger* heap fragment (of the client's DAGs) to a *smaller* (nested) heap fragment (of the callee's DAGs). Preventing the loss of reachability information in this case is challenging. For example, if the client operates only on the four nodes `l`, `l.right`, *s*, and `r` while the callee operates on the subset with `l.right`, *s*, and `r` (cf. Fig. 3.2b), then propagating the fact that there exists a path of the form `l.right` ... *s* is challenging because this requires the knowledge that our path does not *depend* on the node `l` which is not available to the callee. Thus, the modular specifications of a callee method cannot depend on the existence of heap paths that traverse the nodes that are not available in these specifications.

8 We will introduce the general concept of *reachaility framing* in Sec. 3.5.

In the latter case, framing involves the propagation of reachability information from a *smaller* heap fragment (of the callee's DAGs) to *larger* heap fragment (of the client's DAGs). However, it is again challenging to prevent the loss of reachability information. For example, propagating the fact that there *does not exist* a path[9] of the form $\mathtt{r} \dots s$ (Fig. 3.2b) is challenging because this requires the knowledge that *none of the frame nodes* (e. g. $\mathtt{l}$), which are available to the client, connect up our path (e. g. via $\mathtt{r} \dots \mathtt{l} \dots s$).

To summarize, framing involves *decomposing* reachability information available to the client (cf. $\mathtt{R}_0 \to (\mathit{Frame}, \mathtt{R}'_0)$ in Fig. 3.3) and *recomposing* reachability information, in a potentially different state, from the two disjoint heap fragments of the callee and the frame (cf. $(\mathit{Frame}, \mathtt{R}') \to \mathtt{R}$). In the former case, it is challenging to preserve facts about *existing* paths, some of which might be lost due to the decomposition. In the latter case, it is challenging to preserve facts about *non-existing* paths, as some unconnected nodes might become connected in the composition.

3.1.3 Splitting and joining heap paths

We will now explain why framing reachability information is fundamentally challenging. Conceptually, framing involves decomposing and recomposing local reachability information, which can be viewed as *splitting* and *joining* heap paths, resp.

To characterize possible ways to split and join heap paths in the presence of a method call, we use the notion of *cutpoints* [36]. A cutpoint is a heap edge that crosses the boundary of the local heap fragment. If the cutpoint starts in the frame and ends in the local heap fragment, we call it an *entry point* (e. g. (c, a) in Fig. 3.4). If the cutpoint starts in the local heap fragment and ends in the frame, we call it an *exit point* (e. g. (b, c)). In separation logic, one can specify a method's exit points (as these are defined by local node fields) but not the entry points (which are *not defined* locally).

UNBOUNDED CUTPOINTS. Cutpoints define the possible splits and joins of heap paths because each path crossing the boundary of a heap fragment must traverse at least one cutpoint. Generally, a heap fragment may have an *unbounded* number of cutpoints. Moreover, an unbounded number of cutpoints may be traversed by a single heap path. The example of Fig. 3.5 illustrates this; e. g. the path $x \dots z_1 \dots y$ does not involve any cutpoints, while e. g. $x \dots z_2 \dots y$ traverses *four* cutpoints (alternating between exits and entries). Thus, the number of possible path splits is generally also *unbounded*.

In the presence of a method call, precise framing of reachability information requires that (1) in the state before the call, each *local path*, i. e. one that traverses only local heap edges, is split into some number of *sub-paths* based on the corresponding cutpoints that this path traverses, and (2) in the state after the call, the resulting sub-paths are joined, again, based on the cutpoints. However, since the number of splits (and joins) is un-

9 This is needed to prove e. g. that the client still operates on an *acyclic structure* after the callee terminates.

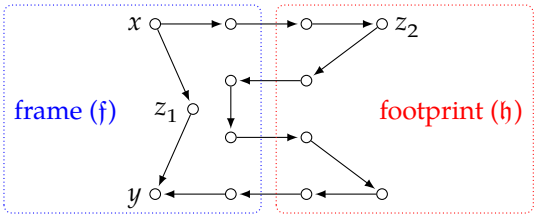


Figure 3.5: Example heap paths and their complex interaction with disjoint heap fragments.

The two heap fragments, denoted $\mathfrak{f}$ and $\mathfrak{h}$, are the frame and the local heap fragment of a method call, resp. Their disjoint union $\mathfrak{f} \uplus \mathfrak{h}$ comprises the local heap fragment of the client of this call. The nodes x and y of the frame are connected up via two distinct, alternative heap paths: $x \dots z_1 \dots y$, that is entirely inside the frame and $x \dots z_2 \dots y$, that crosses the boundaries of this heap fragment —multiple times. Statically inferring the existence or the absence of such paths, e. g. in the program state after the method call, is challenging because (1) there is generally an unbounded number of alternative paths that may connect up these nodes and (2) each possible path may cross the boundary between the two heap fragments an unbounded number of times; hence, it is generally impossible to precisely attribute the existence of a path such as $x \dots y$ to the reachability information that is statically known in a modular setting.

bounded, the general problem of reachability framing is undecidable and thus *cannot be solved* automatically.

3.2 EXISTING WORK

In this section, we give an overview of existing reachability reasoning techniques. Although the semantics of reachability is defined via transitive closure (of the heap adjacency relation), direct reasoning about reachability as transitive closure (a higher-order relation) is beyond the scope of this thesis. Instead, we mainly focus on techniques that can be efficiently automated (i. e. those based on first-order reasoning); we additionally discuss some higher-order techniques with relevant ideas. We then evaluate the existing work based on the following factors: automation, expressiveness of the specification language, modularity of reasoning, and the spectrum of supported data structures.

The discussion of existing work is structured as follows. First, we give an overview of the theory of first-order reachability reasoning in the context of graph database operations (Sec. 3.2.1). Second, we review a technique for *transitive closure simulation* that pioneered entailment proof automation for reachability properties (Sec. 3.2.2). The rest of the section is dedicated to modular techniques. The first modular reachability technique we discuss is based on *propositional reduction* (Sec. 3.2.3). The second technique combines reachability predicates and *separation logic* (Sec. 3.2.4). Next, we proceed with an overview of two other flavors of separation logic which blend reachability with local

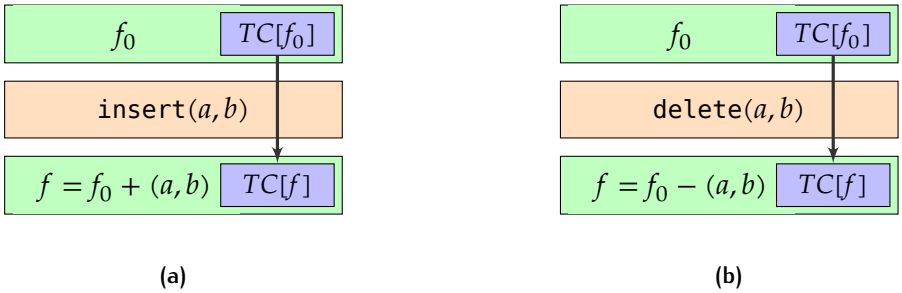


Figure 3.6: Dynamic query update diagram for incremental and decremental graph updates.

The operations (in the orange boxes) (a) insert the edge and (b) delete the edge (a,b); old states are subscripted; the green boxes represent graph configurations with f edges and the purple boxes represent queries *about* the states to which they belong, where $TC[f]$ denotes the transitive closure relation. The response to the queries can be *dynamically updated* by first-order formulas over the original query. Dynamically updating queries is typically fundamentally simpler than recomputing them from scratch.

reasoning but are not tailored for automation (Sec. 3.2.5, Sec. 3.2.6). We position *our work* in the space of the state of the art (Sec. 3.2.7). Tab. 3.1 summarizes the existing work.

3.2.1 Dynamic complexity theory

First-order reasoning about graph reachability properties has been originally studied in the context of database theory. Consider a database that stores the set of nodes V of a mathematical graph (V,E), where E is the edge relation. For the database query language, we choose *first-order logic with transitive closure* of the edge relation E. Statically computing the responses to the queries is typically inefficient; e. g. the asymptotic runtime complexity of the query $\exists m,n \in V \bullet E(m,n) \wedge E(n,m)$ that checks if double edges exist is $\mathcal{O}(|V|^2)$ as we quantify over pairs of nodes.

However, this query can be computed *dynamically*, resulting in constant amortized complexity. A possible dynamic approach requires (1) maintaining an auxiliary relation D of *mutually linked* node pairs (i. e. those that satisfy $E(m,n) \wedge E(n,m)$) and (2) updating this relation after each insert query. For example, inserting an edge (a,b), where $a \neq b$, to the (augmented) graph (V,E,D) will result in $(V \cup \{a,b\},\ E \cup \{(a,b)\},\ (b,a) \in E\ ?\ D \cup \{(a,b)\} : D)$. The simplicity of the *update formula* above for our mutually-linked relation D is not surprizing since D can be *defined* via a first-order formula over E.

3.2.1.1 Dynamic reachability. The idea of dynamically maintaining an auxiliary relation proved to be useful also in the context of the *reachability relation*. Although reachability cannot be *defined* in first-order logic, the *effect* that some operations have on this relation can be expressed using first-order formulas. Concretely, Dong and Su [17] have proposed using first-order reachability update formulas for reducing the dynamic com-

plexity of database queries, e. g. about the existence of graph cycles, pre- and post-dominators. [17] Beyond graph databases, these formulas can also be used for automating reasoning about heap-transforming programs, as demonstrated by several techniques that adopted them [45, 69, 78].

However, these update formulas are still problematic. First, they do not cover the case of a *general graph* but assume that it belongs to one of the supported classes, e. g. it is acyclic. Second, the structure of these reachability update formulas is complicated, in particular, involving *quantifier alternation*, e. g. $\forall x_1.\exists y_1.\forall x_2.\exists y_2$ .... Satisfiability of such formulas is known to be undecidable and hard to automate: Universal quantifiers can lead to infinite instantiations and existential quantifiers require witnesses that generally cannot be inferred automatically. Due to these problems, state-of-the-art automatic program verifiers tend to use simplified reachability update formulas that are within a *decidable* logic. This approach limits the class of supported data structures to various forms of link-lists and trees. Consequently, graphs in which alternative paths may occur remain unsupported [30].

3.2.1.2 ***Descriptive complexity.*** Recent work in descriptive complexity theory may lead to further generalization of first-order reachability reasoning. Patnaik and Immerman [18] have introduced a dynamic complexity class *DynFO* of graph queries that can be maintained by first-order formulas. They conjectured that the graph reachability query is in DynFO. Quarter century later, Datta et al. [97] have shown that this is indeed the case. Therefore, even if the pairwise reachability relation cannot be updated directly, there exists an *auxiliary relation* that carries more information about the graph, enabling first-order update formulas of the reachability query. For example, in the case of acyclic graphs, a binary relation is sufficient for capturing relevant reachability information; hence, there exist first-order update formulas for DAGs.

In contrast, maintaining reachability information in another important class called the *0–1 path graphs*, i. e. graphs with unique paths *modulo cycles* between each node pair, requires an auxiliary, *quaternary* relation, say, Q. Concretely, a quadruple (x, y, u, v) belongs to Q *iff* there exists a path $x \dots y$ that traverses the edge (u, v). After creating a new edge or deleting an edge, the auxiliary relation can be updated using first-order formulas. These update formulas exploit the fact that, given $Q(x, y, u, v)$, removing the edge (u, v) from our graph *must* destroy the path $x \dots y$ as the graph is 0–1 path [17]. Given our relation Q, one can obtain classical two-point reachability as follows:

$$\begin{aligned} \forall x, y, u, v \bullet Q(x, y, u, v) &\implies \mathsf{R}(x, y) \\ \forall x, y \bullet \mathsf{R}(x, y) &\implies \exists u, v \bullet Q(x, y, u, v) \end{aligned}$$

We will discuss reachability in 0–1 path graphs in more detail in Sec. 3.6.

3.2.1.3 Limitations of DynFO. Note that the DynFO complexity class is defined in the context of only two graph operations, namely *edge insertion* and *edge deletion*, as illustrated in Fig. 3.6. These operations are sufficient for modeling e. g. reference field updates. It remains open whether (and under what conditions) reachability update formulas can be defined in first-order logic for more complex operations that commonly occur in imperative programs, e. g. operations over *unbounded* number of nodes and edges. An alternative view of this problem is to understand the interplay of the reachability relations in *different* (potentially overlapping) graphs.

3.2.2 Reachability Simulation

In addition to the problem of updating the reachability relation after heap operations, an automatic reasoning technique must be capable of *converting* reachability information that can be expressed in various forms ($\models$ in Fig. 3.3). On the one hand, arbitrary conversions of reachability information cannot be performed fully automatically as proving the corresponding implications is generally undecidable (see Sec. 3.1.1). On the other hand, there exist *some* useful automatic conversions. For example, if we have established the existence of two paths of the form $x \dots z$ and $z \dots y$, then we could automatically derive the existence of a path $x \dots y$ due to *path transitivity*:

$$\forall x, y, z \bullet \mathsf{R}(x, z) \wedge \mathsf{R}(z, y) \implies \mathsf{R}(x, y)$$

3.2.2.1 Simulating transitive closures. Lev-Ami et al. [53] have proposed a technique for systematically adding (first-order) axioms that can enable a theorem prover to convert reachability information. First, they consider M uninterpreted relations f^k, where $k \in [0, M-1]$ and M equals the number of symbolic states in the program. For each such relation, they introduce a dual uninterpreted relation, denoted f^k_{tc}, called the *simulated* transitive closure relation. In particular, a simulated relation f^k_{tc} is different from the actual transitive closure of f^k in that the former *partially models* the latter. Now each program state k can be characterized via the pair (f^k, f^k_{tc}), corresponding to $(\mathsf{E}_k, \mathsf{R}_k)$ in our notation.

Next, they propose a sound, yet incomplete set of *axiom templates*, i. e. axioms with second-order quantification over (first-order) unary and binary relations. The former can be interpreted as *node sets* $\{A^i\}$, and the latter — as heap edge relations $\{f^k\}$. Hence, axiom templates are higher-order formulas that can be used to derive *partial* first-order constraints over the relations f^k and their respective (simulated) transitive closures, denoted f^k_{tc}. Recall that each f^k_{tc} was introduced as an *uninterpreted function symbol* and cannot express the complete semantics of the transitive closure of f^k.

Since the axiom templates are sound, any type-correct instantiation of the axiom templates is *permitted*. The authors prove that there *does not exist* a complete, recursively-enumerable axiomatization of transitive closure. Despite the theoretical incompleteness,

the techniques is tested to succeed in many practically important scenarios, including the verification of reachability properties in typical heap-transforming programs. The main problem is thus to establish *which* concrete instantiations of the available axiom templates are indeed sufficient for deriving each given reachability property.

3.2.2.2 Axiom templates. To pick the instantiations, the technique employs heuristics that iteratively instantiate the axiom templates with concrete relations based on the user-defined specifications. These instantiations yield *first-order* axioms that are then augmented with the verification conditions and discharged via an SMT solver. If the solver cannot verify the assertion, the heuristics continue with more complex instantiations. For unary relations, the technique uses TVLA-style reachability predicates, including reference-typed variables from the given program (defining *singleton* node sets) and *unary reachability predicates* (e. g. $r_{x,n}$) defining the set of nodes reachable from x via n edges, where x can be iteratively instantiated with concrete nodes.

One useful reachability axiom template is *NoExit*:

$$(\forall u, v \bullet A(u) \wedge \neg A(v) \Rightarrow \neg f(u,v)) \Rightarrow \forall u, v \bullet A(u) \wedge \neg A(v) \Rightarrow \neg f_{tc}(u,v) \quad \mathit{NoExit}[A,f]$$

This axiom template works as follows. We start by choosing a first-order predicate A and a binary relation f to obtain the concrete instantiation ($\mathit{NoExit}[A,f]$). Then, the obtained formula will partially constrain the uninterpreted function symbol f_{tc} that models the reflexive, transitive closure of f. This concrete axiom obtained from ($\mathit{NoExit}[A,f]$) says that, if no f-edge connects an inner node of the set defined via A with a node outside of that set, then the outside nodes *cannot* be reached from the inner nodes via f_{tc}-paths.

3.2.2.3 Example. To illustrate the technique, consider a simple procedure `listConcat` that concatenates two (acyclic) singly-linked lists (Fig. 3.7).[10] The specification language in the transitive closure simulation technique is first-order logic with one auxiliary reachability relation *per state*. Each reachability relation is defined as the reflexive, transitive closure of the current state's heap edge relation. `listConcat` modifies the heap only once (`last.next := y;`). Hence, we need only two edge relations to specify this example, namely n and n', denoting the edge relations in the pre- and post-states, resp.:

$$\forall x, y \bullet n(x,y) \;:\Longleftrightarrow\; \texttt{old}(x.\texttt{next}) = y$$
$$\forall x, y \bullet n'(x,y) \;:\Longleftrightarrow\; x.\texttt{next} = y$$

The precondition of `listConcat` requires that the edge relation n defines an acyclic singly-linked list (or a disconnected set thereof), denoted `alseg(`n`)`, that `x` is a non-null reference, and that each heap node v is reachable from either `x` ($r_{x,n}(v)$) or `y` ($r_{y,n}(v)$)

10 This example is derived from the second example of [53] by removing the specifications related to the memory model (as they are superseded by the type system) and slightly strengthening the precondition (to avoid dealing with the case of `x` = **`null`**).

```
method listConcat(x: Ref, y: Ref)
  requires alseg(n) ∧ x ≠ null
           ∀v • ¬r_{x,n}(v) ∨ ¬r_{y,n}(v)
           ∀v • r_{x,n}(v) ∨ r_{y,n}(v)
  ensures alseg(n')
          ∀v • r_{x,n'}(v) ⇔ (r_{x,n}(v) ∨ r_{y,n}(v))
          ∀u,v • n'(u,v) ⇔ n(u,v) ∨ (u = last ∧ v = y)
{
  var last: Ref := x
  while (last.next ≠ null) invariant last ≠ null ∧ r_{x,n}(last)
  { last := last.next }
  last.next := y;
}
```

Figure 3.7: Example procedure specified with simulated transitive closure relation.

The procedure `listConcat` traverses the heap via the `next` field starting from the head of the first list, `x`, reaches the last node, and connects it to the head of the second list, `y`. The edge relations n and n' represent heap edges in the states before and after the last assignment operation, resp. The macro `alseg`(n) specifies n to be *acyclic*, *functional*, and *unshared*.

exclusively. Note that this is not a modular technique, and so the precondition effectively restricts the configuration of the entire heap.

The postcondition ensures that (1) the new edge relation, n', still represents a set of acyclic lists, (2) any node v is reachable from `x` via the *new* edge relation, n', *iff* it has been reachable from either `x` or `y` via the *old* relation, n, and (3) the relations n' and n are identical except for the one newly created edge, (`last`, `y`), which is present only in n'. Finally, the *loop invariant* preserves reachability from `x` to the node referred to by `last` while traversing the heap via the `next` field.

The purpose of the unary reachability predicates (e. g. $r_{x,n}$ and $r_{y,n}$) is to help the heuristics in picking reasonable instantiations of the axiom templates. The unary reachability predicates are axiomatized as follows:

$$\forall x, y \bullet r_{x,n}(y) \;:\Longleftrightarrow\; n_{tc}(x, y)$$
$$\forall x, y \bullet r_{x,n'}(y) \;:\Longleftrightarrow\; n'_{tc}(x, y)$$

Here n_{tc}, and n'_{tc} denote the (simulated) reflexive, transitive closures of the binary relations n, n', resp. For our example of Fig. 3.7, the technique generates six instantiations of axiom templates, using the following predicates: `y`, `last`, $r_{x,n}$, $r_{y,n}$ (unary relations) and n, n' (binary relations).[11] These instantiations yield concrete reachability axioms. The SMT solver is then able to use the generated first-order partial reachability axiomatization to verify the postcondition of `listConcat`.

11 The technique selects subsets of the available relations to produce instantiations of axiom templates and e. g. `x` is another candidate that is not used in this case.

3.2.2.4 Limitations. The transitive closure simulation technique interacts with an SMT solver, attempting to find concrete instantiations of the template reachability axioms that are sufficient for proving a given reachability property. This approach works in a scenario in which the specified reachability properties of our program are respected by its implementation. However, this knowledge is generally not available; ideally, the tool should help the programmer in both certifying a correct implementation as well as finding bugs. Unfortunately, it is not possible to distinguish whether proving a reachability property requires more instantiations of the axiom templates or if the program, indeed, violates its specification.

3.2.3 Effectively propositional reduction

Effectively propositional reduction (EPR) is a technique and a tool for reasoning about reachability properties of programs that manipulate linked data structures [69]. The input program and its specification are translated into *the EPR logic*, a decidable fragment of first-order logic the assertions of which are automatically checked by an SMT solver. If an assertion is invalid, the tool produces a counterexample, i. e. a heap configuration that violates the assertion.

3.2.3.1 Key ideas. There are three main ideas in the EPR technique. First, in a restricted setting, the reachability update formulas for *bounded heap updates*, i. e. creation and deletion of a finite number of heap edges, can be expressed in propositional logic. Concretely, EPR restricts supported heap structures to *functional graphs* in which nodes have unique reference fields.

Second, each method is equipped with a footprint that must be explicitly specified by the programmer. To express footprints of an unbounded size, the specification language includes a reachability-based notion of *heap segments*, i. e. sets of nodes in an acyclic list segment. Thus, each footprint must be expressed as a union of a bounded number of heap segments. In the setting of EPR, heap segments are definable using only three parameters, e. g. $[a, b]_f$ denotes the segment via f-fields from a till b inclusive.

Third, EPR introduces a rule for *reachability framing*, i. e. adapting the reachability relation to the effects of a method call. The reachability framing rule of EPR is also expressible in propositional logic due to efficient cutpoint management (cf. Sec. 3.1.3). Since there cannot exist alternative paths connecting the same pair of nodes, the entry point of each node into the callee footprint is *uniquely defined*. To refer to these entry points, EPR axiomatizes an *idempotent* entry point function.

3.2.3.2 Example. We illustrate the EPR technique with the example of Fig. 3.8. The recursive method `find` of the Union-Find data structure [9] is specified in a modular way. In particular, EPR requires the method footprints to be specified via a dedicated **`mod`** clause. The argument of **`mod`** is a set of heap nodes defined as a union of a bounded

number of acyclic list segments. Each method may create or remove heap edges only if they start *and* end in nodes from **mod**.

In contrast to separation logic, this implies that the footprint and the frame of a method call can *overlap*. For instance, if **mod** of a callee consists of the segment $[a, b]$ of a (strictly longer) list $[a, c]$, then *both* **mod** and the frame of the call, i. e. $[b, c]$, would include the node b. While this design simplifies cutpoint management, it is not compatible with many existing program logics which assume *disjointness* of the footprint and its frame.

Returning to the specification of `find` Fig. 3.8. The precondition requires that `x` is not null. The **mod** clause specifies that `find` may change only the heap edges of the list segment starting in `x` and ending in some node referred to by the ghost variable r. The postcondition ensures that (1) the out parameter, `root`, is indeed equal to r, and (2) there exists a path $u \dots v$ via some number of `f` fields (denoted $u\langle \mathtt{f}^* \rangle v$), where $u, v \in$ **mod**, *iff* either $u = v$ or v is the root. The latter property specifies that our version of `find` performs the *path compression* optimization.

```
method find(x: Ref)
  returns (root: Ref)
  requires x ≠ null
  mod [x, r]_f
  ensures root = r ∧ ∀u, v ∈ mod • u⟨f*⟩v ⟺ u = v ∨ v = r
{
  var i = x.f
  if (i ≠ null) {
    i = find(i)
    x.f = i
  } else { i = x }
  return i
}
```

Figure 3.8: Example method specified according to the EPR technique.

The method `find` traverses the heap via the `f` field starting from `x` until the last *root* node is reached (which is the result of `find`) while performing *path compression*, i. e. connecting each node along the path directly to the root. The notation $[a, b]_\mathtt{f}$ denotes a list segment from the node a to the node b via `f`-edges (a must reach b); it is used in the **mod** to specify the method's footprint. The **mod** clause is evaluated in the state of the *precondition*; r_x allows to refer in **mod** to the node that `root` refers to in the postcondition. The ternary relation $u\langle \mathtt{f}^* \rangle v$ denotes the existence of a heap path from u to v via `f`-edges, i. e. $\mathtt{R}^{\{\mathtt{f}\}}(u, v)$ using our standard notation; note that EPR supports only transitive closure over *functions*, e. g. $u\langle (\mathtt{f}|\mathtt{e})^* \rangle v$ is not a valid predicate in EPR as the disjunctive relation `f|e` is not a function for two distinct fields `f`, `e`.

The most interesting step in reasoning about `find` is the recursive call. Note that this call (`find(i)`) takes place under the conditions `i = x.f` and `i ≠ null`; hence, the **mod** set

for the recursive call is equal to $[\texttt{x.f}, r]$. Since the technique works under the assumption of acyclic graphs,[12] the size of **mod** must strictly decrease at each recursion level.

After a recursive call returns, the reachability information from the callee's postcondition (specified for nodes of the callee's **mod**-set) must be extended to the larger **mod**-set of the client. The EPR technique employs elaborate, yet effectively decidable *reachability adaptation* formulas to solve this problem.

3.2.3.3 Memory management. EPR realizes explicit memory management with the help of a global predicate called *free*. The value $free(\texttt{x})$ indicates whether the node x is, indeed, *free*, i. e. it was never allocated via the **new** command or was freed via the **free** command; the specifications of these memory management commands are shown in Fig. 3.9.

```
retval = new()
  requires free(s)
  mod ∅
  ensures retval = s ∧ ¬free(s)
```

```
free(y)
  requires y ≠ null ∧ ¬free(y)
  mod ∅
  ensures free(y)
```

Figure 3.9: Specification of memory management operations in EPR.

3.2.3.4 Limitations. EPR requires each method's postcondition (the **ensures** clause) to be *complete*, i. e. specify the new reachability relation for all pairs of nodes within the **mod** clause. This practically restricts the class of supported programs: The effect of an arbitrary program on the reachability relation may or may not be precisely specifiable in the restricted logic of EPR.

Under the complete postcondition assumption, EPR supports reachability framing and propagates the local effect of a method call to the context of the client. Generally, reachability framing from a smaller context to a bigger one cannot be expressed in a restricted, decidable logic. However, reachability framing in EPR *can* be expressed in propositional logic because in this technique the number of *exit points* from the footprint is bounded.

In the EPR logic, transitive closure is eliminated by simulating each reachability relation with an uninterpreted, partially axiomatized relation. To support decidable reasoning about reachability, EPR assumes that it is a *total order*, i. e. that it is reflexive, transitive, acyclic, and linear. Note that the last two properties only apply to reachability in acyclic list segments. EPR can be extended to cyclic singly-linked lists by maintaining auxiliary information about *cycle-inducing heap edges*. However, the linearity property is crucial and the carefully designed technique of EPR cannot support non-functional graphs for multiple reasons, e. g. also because each frame node is assumed to have a *unique* entry point into the footprint.

12 The authors hint that the technique can be extended to reasoning with potentially-cyclic lists.

3.2.4 GRASShopper

GRASShopper [78] is a technique and a tool for verifying heap-transforming programs by translating separation-logic proof obligations into *GRASS*, the logic of graph reachability with stratified sets. GRASS is a decidable fragment of first-order logic, and its assertions can be automatically checked by an SMT solver. If an assertion is invalid, the tool produces a counterexample, i. e. a heap configuration that violates the assertion.

GRASShopper additionally supports more expressive yet undecidable specifications, in particular, those with unbounded quantification. The technique is robust under incomplete specifications and undecidable proof obligations. Unlike e. g. the EPR technique from Sec. 3.2.3, GRASShopper does not rely on complete reachability specifications, rather using reachability predicates in the definitions of separation-logic predicates.

3.2.4.1 Key ideas. The reachability technique of GRASShopper is based on two key ideas. First, the technique focuses on a restricted class of *functional* graphs, e. g. those with at most one outgoing heap edge per node. Hence, the entry point of a frame node x into the footprint (if it exists) is a *function* of x, denoted ep. To limit the discussion to total functions (which are natively supported by SMT solvers), ep is augmented s.t. if a frame node y does not reach any of the footprint nodes, then its entry point is defined as y. Finally, if z is a footprint node, then its entry point into this footprint is z, i. e. the entry point function is idempotent: $\forall x \bullet ep(ep(x)) = ep(x)$. The fact that entry points for all nodes can be encoded as functions dramatically simplifies the problem of cutpoint management (see Sec. 3.1.3).

Second, GRASShopper extends the classical frame rule from separation logic, which is insufficient for reachability properties (as discussed in Sec. 3.1.3). In addition to the information about which heap edges are unchanged, the extended frame rule provides the information about unchanged reachability facts. However, these facts are expressed using the *generalized dominator relation*, called `Btwn`, as opposed to the classical two-point reachability relation. For instance, `Btwn(f,x,z,y)` holds *iff* there exists an `f`-path `x` ... `y` and `z` is *between*, i. e. it occurs on all paths from `x` to `y`.

Based on the *ep* function and the `Btwn` relation, the technique encodes reachability framing formulas in a way that is sound, complete, and concise.

3.2.4.2 Specification language. The classical two-point reachability predicate called `Reach(f,x,y)` is desugared to `Btwn(f,x,y,y)`. `Reach(f,x,y)` is equivalent to $\mathsf{R}^{\{\mathsf{f}\}}(\mathsf{x}, \mathsf{y})$ in our standard notation, except that in the former case `y` can be null. Hence, the predicate `Reach(f,x,null)` says that `x` is the head of an *acyclic* list.

The `Btwn` relation is useful for constructing separation-logic predicates. For example, the following code defines a predicate for an acyclic singly-linked list segment `x` ... `y`:

```
struct Node { var next: Node; }
```

```
predicate lseg(x: Node, y: Node) {
  acc({ z: Node :: Btwn(next, x, z, y) ∧ z ≠ y }) * Reach(next, x, null) }
```

`acc`(S) grants access permissions to all nodes in the node set S. The node sets can be defined via the `Btwn` and the `Reach` relations and unbounded quantification. Due to the condition `z` $\neq$ `y`, an entire list starting in `x` can be expressed as `lseg(x,null)`. Excluding the end of a list segment also simplifies list decompositions, e. g. `lseg(x,y)*lseg(y,null)` (where `y` $\neq$ `null`) is equivalent to original assertion `lseg(x,null)`. If `y` is not reachable from `x`, then `lseg(x,y)` specifies an empty set of permissions.

GRASShopper distinguishes first-order conjunction, $\wedge$, and *separating conjunction*, $*$.[13] If P and Q are separation-logic assertions, then $P*Q$ expresses that these assertions apply to disjoint parts of the heap, while $P \wedge Q$ does not restrict the overlapping. For example, `lseg(a,b)*lseg(c,d)` says that the list segments `a` ... `b` and `c` ... `d` are disjoint in the heap. If P or Q is a pure (first-order) predicate, then $P * Q$ is equivalent to $P \wedge Q$.

3.2.4.3 Example. We illustrate GRASShopper with the example of Fig. 3.10. The recursive method `find` of the Union-Find data structure [9] is specified in a modular way (cf. Fig. 3.8). In particular, the footprint of `find` consists of the ghost set `X` plus one extra node, `root`. Since `lseg(x,root)` and `acc(X)` appear by the same side of a separating conjunction (in the precondition), these assertions *overlap*. Hence, `X` can be viewed as the *footprint of the predicate* `lseg(x,root)`. Using `X`, we can simply express in the postcondition of `find` that this method does not leak (or allocate) memory. Indeed, all original permissions will be returned to the client after the call, even though (due to path compression) we might no longer have a single list structure on the heap.

3.2.4.4 Memory management. To keep track of all currently allocated memory locations, GRASShopper internally uses a global variable called `Alloc`. That is, `Alloc` stores the set of allocated nodes in each program state. GRASShopper uses `Alloc` to maintain the invariant that footprints consist only of allocated nodes.

GRASShopper computes the frame of a call *at call site* as opposed to the client's context. Internally, the callee's footprint is specified via two implicit ghost variables, called `FP` and `FP_Caller`, representing the current footprints of the callee and its client, resp. For instance, `FP_Caller` in `find` (Fig. 3.10) is equal to $(\mathit{lseg} \cup \mathtt{X}) \uplus \{\mathtt{root}\}$, where *lseg* is the footprint of the predicate `lseg(f,x,root)`. In presence of memory allocation and deallocation (or a method call) the values of `FP` and `FP_Caller` are automatically updated, and the values of these variables in the final state of the call are returned to the client.

3.2.4.5 Framing. GRASShopper's reachability framing rule consists of two parts, (3.2) and (3.3), which we explain next; only the former in needed for fields of non-reference types. Consider the pre-state of a method call in which the callee's footprint `FP` equals

13 The tool uses && and &*& for $\wedge$ and $*$, but we adopt the latter notation in the interests of consistency.

```
procedure find(x: Node, ghost root: Node, implicit ghost X: Set<Node>)
  returns (res: Node)
  requires lseg(x, root) ∧ acc(X) ∗ root.f ↦ null
  ensures acc(X) ∗ root.f ↦ null ∗ res = root
{
  var n := x.f;
  if (n ≠ null) {
    res := find(n, root);
    x.f := res;
  } else {
    res := x;
} }
```

Figure 3.10: Example method specified in GRASShopper.

The method `find` traverses the heap via the `f` field starting from `x` until the last node, `root`, is reached (which is the result of `find`) while performing *path compression*, i. e. connecting each node along the path directly to `root`. `root.f` ↦ **`null`** is the *points-to* predicate that means **`acc`**`(root)` ∗ `root.f` = **`null`**. `lseg(x,y)` defines an acyclic list segment starting in `x`.

$\mathfrak{h}$ and the set of all allocated nodes `Alloc` equals A. Let f and f' be a field in its pre- and post-states, resp. Then, the separation-logic frame rule can be expressed as follows:

$$FrameSL(A,\mathfrak{h},f,f') \;:\Longleftrightarrow\; \forall n \in A \setminus \mathfrak{h} \;\bullet\; x.f = x.f' \tag{3.2}$$

This formula says that the value of the field f of any node in the frame is preserved.

The formula (3.2) alone is insufficient for preserving reachability properties (cf. Sec. 3.1.3). Hence, GRASShopper employs the following additional rule of reachability framing:

$$\begin{aligned} Frame(A,\mathfrak{h},f,f') \;:\Longleftrightarrow\; \big(&\forall x,y,z \in A \setminus \mathfrak{h} \;\bullet\; \mathtt{Btwn}\,(f,x,y,ep(\mathfrak{h},f,x)) \implies \\ &(\mathtt{Btwn}(f,x,z,y) \Leftrightarrow \mathtt{Btwn}(f',x,z,y))\big) \\ \wedge\, &\forall x,y,z \in A \;\bullet\; x \notin \mathfrak{h} \wedge x = ep(\mathfrak{h},f,x) \implies \\ &(\mathtt{Btwn}(f,x,z,y) \Leftrightarrow \mathtt{Btwn}(f',x,z,y)) \end{aligned} \tag{3.3}$$

The formula (3.3) provides two precise conditions under which triplets (x,z,y) of the `Btwn` relation are preserved as a result of the call that transforms the reference field f into f'. The first condition, $\mathtt{Btwn}(x,y,ep(\mathfrak{h},f,x))$, implies that y dominates t on the path $x \dots t$. Hence, the path $x \dots y$ is preserved. The second condition, $x \notin \mathfrak{h} \wedge x = ep(\mathfrak{h},f,x)$, implies that x is a frame node that does not reach the footprint at all. Hence, all heap paths starting in x are preserved.

3.2.4.6 Limitations. The main limitation of GRASShopper is that each supported reachability relation is tied to a single reference field. This design enables a simple reachability

framing formulation that falls into a decidable fragment of first-order logic. Similar to the EPR technique discussed in Sec. 3.2.3, GRASShopper encodes entry points via an idempotent, total, uninterpreted function. The axiomatization of reachability framing in GRASShopper relies on the fact that *ep* is a function, i. e. each node can have only one entry point into each footprint. Hence, one cannot easily generalize this design to support reachability over multiple reference fields, e. g. to support reasoning about general DAG structures.

3.2.5 Ramification techniques

Ramification techniques generalize classical separation logic to support reasoning about operations that have *non-local* effect [116]. The classical frame rule preserves only compositional properties of a heap fragment, i. e. those that depend only on the nodes *inside* this fragment. Under the compositionality assumption, modular reasoning is fundamentally simplified. In particular, a method call cannot change properties of its frame; thus all that is left to be done after the call is to compose the (preserved) properties of the frame and in the (updated) properties of the callee footprint into the property of the client footprint. Reachability is an example of a non-compositional property that violates this assumption because it talks about the existence of arbitrary heap paths, not necessarily those within the footprint or the frame.

$$\frac{G_1 \vdash L_1 * F \quad \{L_1\}\, c\, \{L_2\} \quad L_2 * F \vdash G_2}{\{G_1\}\, c\, \{G_2\}}\ \mathrm{Free}(F) \cap \mathrm{Mod}(c) = \emptyset \qquad (\textit{Frame})$$

To tackle the problem of *framing* non-local properties, ramification introduces a generalized, more expressive version of the *Frame* rule from classical separation logic [22]. This new rule lifts the requirement that the footprint is disjoint with the remainder heap fragment called the *ramification frame*. For example, Wang et al. [116] replace *Frame* with a generalized *Localize* rule with two novelties. First, they give up the quantifier-free form of the postconditions of the Hoare triples in the rule. Second, they add universal quantification to the third premise. To phrase the constraints set upon the ramification frame, R, the rule uses the *magic wand* [26] connective defined through $(P \vdash Q \mathrel{-\!\!*} R) \ :\Longleftrightarrow\ (P * Q \vdash R)$; note that the third premise of *Frame* can be written as $F \vdash L_2 \mathrel{-\!\!*} G_2$, closely resembling the structure in *Localize*, although sparing the universal quantifier. In both rules, the side condition ensures that free variables of the frame F (or the ramification frame R) cannot be modified by the operation c.

$$\frac{G_1 \vdash L_1 * R \quad \{L_1\}\, c\, \{\exists x \cdot L_2\} \quad R \vdash \forall x \cdot (L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\, c\, \{\exists x \cdot G_2\}}\ \mathrm{Free}(R) \cap \mathrm{Mod}(c) = \emptyset \quad (\textit{Localize})$$

The simplicity of the original frame rule allows for efficient frame inference, enabling modular reasoning techniques that can be *automated*. In contrast, ramification frames

were designed for expressivity, not automation. As a result, they are much harder to infer. However, ramification techniques are typically implemented as extensions to separation-logic reasoning frameworks for interactive proof assistants, e. g. Coq and VST [29] or Isabelle [25].

Both the classical frame rule as well as *Localize* can be efficiently combined with the concept of *iterated separating conjunction* (ISC). ISC is a generalization of the separating star conjunction that allows simultaneously specifying access permissions to an unbounded set of nodes. Even though ISC were already included in classical separation logic [26], the presence of ISC introduces additional challenges to automation.[14] Without ISC, footprints in a fixed state may only access bounded sets of nodes (and other resources). In contrast, ISC enable access to *unbounded* node sets that naturally represent footprints of *arbitrary* operations.

Reachability can be encoded in ramification via the idea of *local paths*. For example, in [116], local paths are an essential construct used in data structure specifications. In their definition, a local path $x \rightsquigarrow^{\gamma} y$ expresses that x reaches y in a *graph* γ that represents some heap fragment in a fixed state. Rather than axiomatizing graphs via a theory of sets, ramification techniques [116] built atop *type theory* that has better support in proof assistants. As discussed in Sec. 3.1.2, reasoning in separation logic requires the consideration of local paths because, generally, modular method specifications cannot precisely express global reachability properties.

Since ramification techniques do not provide any guidance for choosing ramification frames, the problem of managing *cutpoints* is open. Refer to Sec. 3.1.3 for a detailed discussion of the cutpoint management problem.

3.2.6 Flows

Flows is a framework for verification of *flow interfaces*: graph invariants about a rich spectrum of properties including reachability properties [100, 118]. Procedures can operate on subgraphs that correspond to their footprint. If the callee complies with its local flow interface, a view on the client's graph can be reconstructed. Reasoning about flows requires fixpoint computations that are generally hard to automate. An important limitation of Flows is that the framework does not permit propagating *side effects* of a local operation to the client's context. We first provide a quick introduction to the generic Flows framework, then illustrate how it can be applied to reachability reasoning, and then discuss limitations in more detail.

3.2.6.1 Concepts of the Flows framework. Consider a graph $G = (N, e)$ — e. g. representing a fragment of the program heap — where N is a node set and e is the set of *edge functions* specifying the connections between nodes. A *flow graph* $H = (N, e, \texttt{flow})$

14 Fortunately, an automatic separation logic reasoning technique with ISC has been recently proposed by Müller, Schwerhoff, and Summers [90] and has enabled the key contributions of this thesis.

is the augmentation of our graph G with a family of `flow` functions, specifying a measure called *flow* for each node of the graph. The flow of a node may depend on both local properties of this node (e. g. the value of its fields) as well as global properties of the entire graph. To incorporate the global properties of a graph into the flow of a given node, the Flows framework employs the following recursive equations:

$$\forall n \in N \bullet \texttt{flow}(n) = \texttt{inf}(n) + \sum_{m \in N} \texttt{flow}(m) \triangleright e(m,n) \tag{FlowEqn}$$

Here $\texttt{inf}(n)$ is the in-flow in node n, $e(m,n)$ is the edge function operating along the (potential) edge from m to n, and $x \triangleright e(m,n)$ denotes the application of the function $e(m,n)$ over x.

Intuitively, the flow equations (FlowEqn) describe, for each node n of our graph, how their flow is influenced by the flow coming from their *predecessors*. Note that the flow is propagated only along direct edges, i. e. from a predecessor to its successors; hence, the value of $\texttt{flow}(n)$ is defined as a *fixpoint* of the system of equations (FlowEqn). We omit the discussions about the conditions under which such fixpoints exist and assume that there exists a unique fixpoint for `flow`.The relevant formal proofs are presented in [118].

Carefully instantiating the domain of the Flows framework can enable expressing rich properties of heap graphs. The fixpoint computation described above propagates global graph properties to local nodes, supporting specifications on different abstraction levels. Since Flows build atop separation logic, procedure footprints can be specified in terms of node sets using iterated separating conjunction (ISC).

This approach supports modular verification of program invariants: A client needs only abstract information about the callee (its *flow interface*) to be able to reason about a call. Conversely, the concrete details (i. e. the actual flow graph) that do not affect the invariant are irrelevant for the client and thus can be hidden. Flow interfaces consist of two components, namely, the family of in-flow functions `inf` (introduced in (FlowEqn)) and the family of *out-flow* function `outf`, which are defined as follows:

$$\forall n \in \textbf{\textit{Ref}} \setminus N \bullet \texttt{outf}(n) = \sum_{m \in N} \texttt{flow}(m) \triangleright e(m,n)$$

Note that `outf` maps each node *outside* of our flow graph H to its out-flow. Using the notions of in-flow and out-flow, the Flows framework defines interfaces as the pair $(\texttt{inf}, \texttt{outf})$, providing a way to abstract concrete flow graphs via their flow interfaces. In particular, verifying the callee typically requires full information presented in its flow graph, but its public invariant can be written more abstractly in terms of the corresponding flow interface that is suitable for modular reasoning on the client's side.

3.2.6.2 Encoding reachability in Flows. We now demonstrate how the Flows framework can express reachability. First, we need to find an appropriate instantiation of the flow domain, restricting possible values of the `flow` function. Formally, the flow domain

is an augmentation of a commutative, cancellative (total) monoid $(M, +, 0)$ (where M is the domain of flow values) with a collection of edge functions $E \subseteq M \to M$. The cancellativity property (required in the meta-theory of Flows) can be used for deriving e. g. $a = b$ from $a + c = b + c$, if the latter equality is well-defined in $(M, +, 0)$.

We aim to express reachability from some node $r \in N$ to an arbitrary node n.[15] The key idea is to instantiate the Flows framework s.t. information from r is propagated to all nodes that are eventually reached by following heap paths starting in r. Then, a non-unit flow value ($\texttt{flow}(n) \neq 0$) would indicate that n is reachable from r.

We proceed by considering the following formal instantiation of the Flows framework. First, we select the domain of flow values to be *multisets of sets of nodes*, i. e. $\texttt{flow}(n)$ contains the sets of nodes representing each simple path from r to n. Hence, we have:

$$\mathit{FlowDomain} : \left(\mathbb{N}^{\mathbb{2}^{\mathbf{Ref}}}, \cup, \emptyset, E\right) \qquad H = (N, e, \texttt{flow}) \qquad E \;:=\; \{\lambda_n \mid n \in N\}$$

Here the first component of the flow domain is the set of all multisets of node sets. For example, a multiset of paths $r \ldots n$ in the graph of Fig. 3.11, denoted μ, must be of the form $\{p_1 \rightarrowtail c_1, p_2 \rightarrowtail c_2, p_3 \rightarrowtail c_3\}$, where $p_1 = \{r, a, b, n\}$, $p_2 = \{r, c, b, n\}$, and $p_3 = \{r, c, d, n\}$ and c_i are the counters of their corresponding elements p_i in our multiset. Then, $\mu \in \mathbb{N}^{\mathbb{2}^{\mathbf{Ref}}}$ is a valid flow value.[16]

Second, we initialize the flow functions with $\emptyset$ in all nodes of N *except* r; the latter should have a different initial flow value which can then be propagated to all *reachable* nodes via the fixpoint computation of (FlowEqn). For this initial value of $\texttt{flow}(r)$ we will use the singleton multiset $\{\emptyset \rightarrowtail 1\}$:

$$\forall n \in \mathbf{Ref} \bullet \texttt{inf}(n) \;=\; \left(\lambda n.\; n = r \;?\; \{\emptyset \rightarrowtail 1\} : \emptyset\right) \qquad \text{(In-Flow)}$$

To propagate the initial flow, we define the edge function $e(n, x) \;:=\; \lambda_n$; here $n \in N$ is the predecessor and $x \in \mathbf{Ref}$ is the successor and λ_n is defined as follows:

$$\forall n \in \mathbf{Ref} \bullet \lambda_n(S) \;:=\; \{p \rightarrowtail (n \in p \;?\; S(p \setminus \{n\}) : 0)\} \qquad \text{(Edge-Functions)}$$

Note that the edge functions depend only on the predecessor and not on the successor node; intuitively, this corresponds to the fact that heap nodes cannot control the set of other nodes that reference them. The parameter S in $\lambda_n(S)$ corresponds to the (multiset) flow value of the predecessor of n; hence, $S(p \setminus \{n\})$ yields the count of $p \setminus \{n\}$ in the multiset S (recall that node sets e. g. p represent heap paths, so $p \setminus \{n\}$ is effectively the intermediate path from r before we reached n).

15 This instantiation of the Flows framework is inspired by the inverse reachability example presented in [110].
16 Our instantiation leads to the following concrete signatures: $N \subseteq \mathbf{Ref}$, $e : N \times \mathbf{Ref} \to E$, $\texttt{flow} : N \to \mathbb{2}^{\mathbf{Ref}}$.

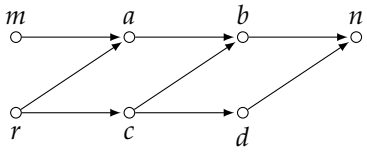


Figure 3.11: Example heap configuration with multiple paths connecting r to n.

COMPUTING THE FIXPOINT. We proceed by computing the flow value of our starting node, r. Instantiating (FlowEqn) with r for n, we obtain the general expression for $\texttt{flow}(r)$:

$$\texttt{flow}(r) \;=\; \texttt{inf}(r) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,r) \;=\; \{\emptyset \rightarrowtail 1\} \cup \{\{r\} \rightarrowtail 0\} \;=\; \{\emptyset \rightarrowtail 1\}$$

Only the in-flow affects the overall flow of r (cf. (In-Flow)). None of the other nodes add any flow to r because the edge function $e(x,r)$ yields an empty multiset if $x \notin \{r\}$ (cf. (Edge-Functions)). Analogously, we proceed by computing all other flow functions in a *breadth-first* manner.

$$\begin{aligned}
\texttt{flow}(c) &= \texttt{inf}(c) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,c) &&= \{\{r\} \rightarrowtail 1\} \\
\texttt{flow}(a) &= \texttt{inf}(a) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,a) &&= \{\{r\} \rightarrowtail 1\} \\
\texttt{flow}(b) &= \texttt{inf}(b) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,b) &&= \{\{r,a\} \rightarrowtail 1, \{r,c\} \rightarrowtail 1\} \\
\texttt{flow}(d) &= \texttt{inf}(d) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,d) &&= \{\{r,c\} \rightarrowtail 1\} \\
\texttt{flow}(n) &= \texttt{inf}(n) \cup \bigcup_{x \in N} \texttt{flow}(x) \rhd e(x,n) &&= \{\{r,a,b\} \rightarrowtail 1, \{r,c,b\} \rightarrowtail 1, \{r,c,d\} \rightarrowtail 1\}
\end{aligned}$$

In the five cases above, the in-flows are set to $\emptyset$. For node c, the only predecessor is r; hence, the big union yields $\texttt{flow}(r) \rhd e(r,c) = \lambda_r\,(\texttt{flow}(r)) = \lambda_r\left(\{\emptyset \rightarrowtail 1\}\right) = \{\{r\} \rightarrowtail 1\}$. Analogously, we compute the flow in all other nodes, including n.

Finally, we can now define the reachability predicate R in terms of flows:

$$\texttt{R}(r,n) \;:\Longleftrightarrow\; |\texttt{flow}(n)| > 0$$

Beyond point-wise reachability, one can also express some other properties using the same flow domain. For example, we could specify the *distance* between r and n (denoted dist(r,n)) as the length of the shortest path connecting these nodes:

$$\texttt{dist}(r,n) \;:\Longleftrightarrow\; \min_{p \in \texttt{flow}(n)} |p|$$

Note that the min comprehension above operates on a potentially unbounded set as the number of alternative paths of the form $r \dots n$ is typically not unknown statically; refer to Chap. 2 for the discussion of verification of unbounded comprehensions.

3.2.6.3 Limitations of Flows. The Flows framework supports local reasoning about global graph properties in separation logic. However, the possible *changes* to the properties in question (e. g. the creation or the destruction of heap paths) cannot be propagated to the client because that would violate the flow interface that defines the current reasoning context. Therefore, a Flows-based reasoning technique is limited to the verification of program invariants. In such a setting, the problem of cutpoint management is dramatically simplified as the cutpoints of an operation's footprint must be specified as part of its flow interface, rendering them immutable for the duration of the call.

Automated reasoning presents Flows with another challenge. In general, the fixpoint computation for the equations (FlowEqn) is a higher-order operation that is not natively supported by automatic theorem provers. Although it might be possible to mitigate the problem of automatically reasoning about fixpoints, this is not simpler than solving the original problem of reasoning about the reachability relation since reachability itself can be defined via fixpoints.

Finally, applying the Flows framework in practice requires selecting an appropriate instantiation of the flow domain. This by itself is a non-trivial problem. Recall that our example instantiation from Sec. 3.2.6.2 allowed us to express reachability only *from* a given node r. Specifying general two-point reachability would require a more complex flow domain that would propagate more information throughout the graph. Ultimately, any graph property could be expressed in *some* sufficiently sophisticated flow domain. However, overly detailed flow domains would break the essential abstractions granted by reachability in the first place.

3.2.7 This work

We present a specification and verification technique that allows one to reason about heap reachability properties modularly. The technique is integrated into separation logic and, thus, benefits immediately from the plurality of techniques and tools in this area. The key challenge of this integration is to specify reachability locally, within the footprint of a method. We solved this challenge by specifying reachability relatively to a given heap fragment and introducing a novel form of reachability framing to extend reachability properties in the footprint of a callee method to the larger footprint of the client. Even though reasoning about general reachability properties is difficult to automate, the proof obligations required by our technique are amenable to SMT solvers, which we will demonstrate in our experiments of Sec. 3.8. This section discusses the position of our work on the spectrum of existing techniques and concludes with an outlook into the possible future extensions.

3.2.7.1 Positioning our technique. Most work on separation logic focuses on data structures with limited sharing, with some notable exceptions. Iterated separating conjunction has been used to verify the Schorr-Waite graph marking algorithm [23], but without any tool support or automation. Recent work on Flows [100] (Sec. 3.2.6) allows one to prove the *preservation* of a rich variety of graph invariants including reachability properties, but requires fixpoint computations that are hard to automate. Methods can operate

Table 3.1: Summary of existing reachability verification techniques.

The techniques are grouped as follows. Top group includes foundational work in reasoning about transitive closures; Second group includes automated tools for modular verification of heap reachability properties; Third group includes higher-order separation logics with reachability support. **Auto** = *Automation*; **Modl** = *Modular reasoning*; **SepL** = *separation logic*.

Technique	Discussion	Citation	Auto	Modl	SepL	Structures
Dynamic TC	Sec. 3.2.1	Dong and Su [17]	✓			DAG, ZOPG
TC Simulation	Sec. 3.2.2	Lev-Ami et al. [53]	✓			List
EPR	Sec. 3.2.3	Itzhaky et al. [69]	✓	✓		List
GRASShopper	Sec. 3.2.4	Piskac, Wies, and Zufferey [78]	✓	✓	✓	Tree, List
Ramification	Sec. 3.2.5	Wang et al. [116]		✓	✓	Graph
Flows	Sec. 3.2.6	Krishna, Summers, and Wies [118]		✓	✓	Graph
Our Work	Sec. 3.3	Ter-Gabrielyan et al. [112]	✓	✓	✓	DAG, ZOPG

Table 3.2: Supported data structure categories.

A comparison of data structure categories supported by our technique vs. closest prior work. **"Itz."** denotes EPR (Sec. 3.2.3); **"Gr."** denotes GRASShopper (Sec. 3.2.4); **"Nv."** indicates if, to our knowledge, our work enables automated modular verification of reachability properties for the first time. * — conceptually supported but not evaluated (see Sec. 3.8).

Data structure	Class	Itz.	Gr.	Nv.	Author
General DAGs (e. g. Git hostory)	DAG			✓	Chacon and Straub [74]
Binary Decision Diagrams	DAG			✓*	Lee [3] and Akers Jr. [10]
Trees	ZOPG/DAG		✓		–
Acyclic list segments	ZOPG/DAG	✓	✓		–
Union-Find	ZOPG/DAG	✓	✓		Tarjan [9]
Arbitrary linked-list structures	ZOPG	✓*			–
Trees encoded via Java's LinkedList	ZOPG			✓*	Sun, Oracle
Priority inheritance protocol	ZOPG			✓*	Sha, Rajkumar, and Lehoczky [15]
Hierarchical rings	ZOPG			✓	Fredman et al. [13]
Traveling salesman optimization	ZOPG			✓*	Croes [1] and Lin [5]

on a subgraph; under the condition that *interfaces* [100] of these subgraphs are preserved, a view on the client's graph can be reconstructed. They make no convexity restriction, but the interface preservation conditions rule out the possibility of method calls adding or removing paths between nodes in subgraph boundaries. By contrast, our technique explicitly enables such side-effectful methods, and the reconstruction of appropriate changes in the client's footprint. For instance, we will present an example method `merge` (Fig. 3.13) in which new reachability relations are first *established* (by creating an edge from `link` to the root of `rdag`) and then *propagated* (by the enclosing method calls) to the larger context (the entirety of the client's footprint).

To support reasoning about reachability in presence of filed updates, we adapted the precise transitive-closure update formulas from Dong and Su [17] to program heaps and separation logic, rather than mathematical graphs. Their work also inspired our `DEP` relation that we will use for cyclic structures; however, our version of the `DEP` relation is compatible with the *reflexive* reachability relation and is used only in the internal encoding, whereas theirs is exposed to programmers. Reachability has been integrated into an automatic separation logic setting before; e. g. in GRASShopper [78] (Sec. 3.2.4), but only in a limited way that supports lists and trees but not heap structures with sharing.

Our work was inspired by Itzhaky et al. [69, 76] (Sec. 3.2.3). Their verification technique allows one to modularly prove reachability properties in various forms of list data structures. A focus of their work is to obtain decidable proof obligations. We sacrificed decidability in favor of supporting arbitrary acyclic graphs (with bounded outdegree) as well as 0–1-path graphs; our experiments show that we nevertheless achieve good automation. In contrast to Itzhaky et al., we integrated our work into separation logic, which allows us to verify concurrent programs and to reason about reachability and other properties in a uniform way. Moreover, we do not restrict method footprints in the number of entry and exit points or the number of strongly connected components in them. Tab. 3.2 summarizes the expressiveness of our technique and compares it with closely-related work.

3.3 LOCAL REACHABILITY

In this section, we introduce our automated reachability reasoning technique;[17] the foundation of this technique is a novel specification language for *local reachability properties*. This language is an extension of the core language described in Sec. 1.2.

We illustrate the technique using the same algorithm, `merge`, that served as our motivating example (Fig. 3.2). Recall that `merge` operates on two (disjoint) DAG structures

17 Our technique automatically checks that a program implementation adheres to its modular specifications with reachability; specification inference (in particular, inference of loop invariants) is beyond the scope of this thesis, but the abstract view of the heap provided by reachability is helpful, e. g. in abstract interpretation and shape analysis [20].

```
method merge(l: Node, r: Node,
             g: Graph, ldag: Graph, rdag: Graph)  // ghost parameters
returns link: Node                                // updated node
requires g = ldag ⊎ rdag ∧ l ∈ ldag ∧ r ∈ rdag
         ∀x,y ∈ g • ¬E(g,x,y) ∨ ¬P(g,y,x)         // acyclic invariant
         ∀n • n ∈ ldag ⇔ P(g,l,n)
         ∀n • n ∈ rdag ⇔ P(g,r,n)
ensures link ∈ ldag
        ∀x,y ∈ g • ¬E(g,x,y) ∨ ¬P(g,y,x)          // acyclic invariant
        ∀x,y • E(g,x,y) ⟺ E0(g,x,y) ∨ x = link ∧ y = r
        ∀x,y • P(g,x,y) ⟺ P0(g,x,y) ∨ P0(g,x,link) ∧ P0(g,r,y)
{
  if (l.right != null) {
    var nldag := sub(g, ldag, l.right)            // define new ghost parameter
    link := merge(l.right, r, nldag ⊎ rdag, nldag, rdag)
  } else {
    l.right := r
    link := l
} }
```

Figure 3.12: Example program and its reachability specification.

Method `merge` attaches the DAG rooted in `r` to a node of the DAG rooted in `l`, and returns that node. We use the edge predicate `E` and the path predicate `P` to specify reachability, within a set of objects `g`. The footprint `g` is closed due to the equivalences in the last two preconditions.

with nodes of type ***Node***[18] and reference fields `left` and `right`. Method `merge` in Fig. 3.12 takes as arguments references `l` and `r` to two nodes of the corresponding DAGs and attaches `r` as descendant of `l`. It returns `link`, a node of the first DAG, to which `r` was attached. The postcondition ensures that exactly one connection was created (via an edge from `link` to the root of the second DAG, `r`), and that heap paths exist in the post-state either if they existed in the pre-state or were connected up by the new edge `(link, r)`. We explain the specification of `merge` in full detail in the remainder of this section.

3.3.1 Footprints

The footprint of any method operating on linked heap structures, e. g. lists and DAGs, contains a statically unknown number of memory locations. To provide a convenient way to refer to a method's footprint, we equip each method with a distinct ghost parameter `g: Graph` to denote its footprint. For simplicity, instead of specifying the footprint as a set of object-field pairs, we let ***Graph*** denote sets of non-null objects and keep the fields implicit when they are clear from the context. The set stored in `g` is updated when-

18 Instances of type ***Node*** are *non-null references*, which is ensured in the generated proof obligations.

ever the footprint changes, e. g. due to allocation. In order to be able to refer to the final footprint of a method execution in its postcondition, we make $\mathfrak{g}$ an in-out parameter. For simplicity, we assume in the following that the footprint of a method remains unchanged, s.t. the value of $\mathfrak{g}$ is constant; an extension is straightforward.

We equip each method with implicit pre- and postconditions to require and ensure permissions to all locations in the footprint:

requires $\forall n \in \mathfrak{g} \bullet$ **acc**$(n.\texttt{left}) * $ **acc**$(n.\texttt{right})$
ensures $\forall n \in \mathfrak{g} \bullet$ **acc**$(n.\texttt{left}) * $ **acc**$(n.\texttt{right})$

Here, **acc**$(x.f)$ denotes an access permission to the *memory location* for field f of object x (like $x.f \mapsto \_$ in classical separation logic [26]), $*$ denotes separating conjunction, and the universal quantifier is an *iterated* separating conjunction (ISC) [26, 90], which (here) denotes permissions to all field locations of objects in the footprint $\mathfrak{g}$. In contrast to using recursive definitions to specify unbounded heap structures (e. g. separation logic predicates [24, 34]), ISC permits arbitrary sharing within the set $\mathfrak{g}$ (many field *values* may alias the same node) and does not prescribe a traversal order within the data structure. We assume for simplicity that a method specification expresses *all* required and returned permissions via these implicit contracts with respect to $\mathfrak{g}$, but it is easy to also support other permission specifications, e. g. points-to predicates and recursive predicates.

In our example of Fig. 3.12, we use two additional ghost parameters `ldag` and `rdag` to allow our specification to simply denote the sets of objects constituting the first and second DAG, resp. The first precondition expresses that the method footprint is the disjoint union of these two DAGs.

3.3.2 Reachability predicates

Reasoning in separation logic has the advantage that one can modularly verify properties of a method, and reuse this verification for all calling contexts (and concurrently-running threads). Enforcing that properties verified for the method depend *only* on its footprint, guarantees that they hold independently of the context; we refer to these as the *local* properties of the footprint. However, classical reachability in the heap is *not* a local property of this form. Hence, combining reachability and separation logic requires us to refine the notion of reachability to one that is local, as we explain next.

Our technique provides two predicates to express reachability properties in specifications. We generalize classical reachability by adding an extra footprint parameter, $\mathfrak{g}$ to make the property local. The *edge predicate* $\texttt{E}^F(\mathfrak{g}, x, y)$ expresses that object x is in the set $\mathfrak{g}$ and has a field from the set of fields F storing a non-null object y (which need not be in $\mathfrak{g}$). The *path predicate* `P` denotes, for a fixed $\mathfrak{g}$ and F, the reflexive, transitive closure of `E`, that is, $\texttt{P}^F(\mathfrak{g}, x, y)$ expresses that either $x = y$, or there is a path of field references from x to y s.t. all objects on the path (except possibly y) are in $\mathfrak{g}$ and all fields are in F; in particular `P` may denote reachability via multiple fields. We omit the parameter F when the set of fields is clear from the context; for instance, in our example, F consists

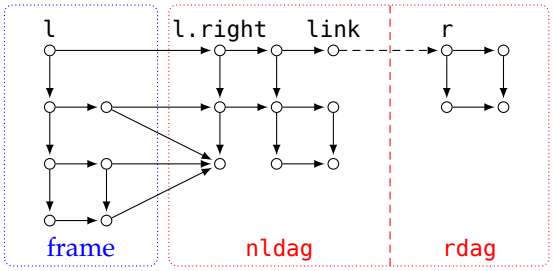


Figure 3.13: Example scenario of running `merge`.

The input structures are two DAGs rooted in `l` and `r`. Small circles correspond to heap objects; solid arrows represent fields initialized in the pre-state that are unchanged; the dashed arrow represents the new heap edge (created in the post-state by initializing a field). The frame of the recursive call is surrounded with blue; the footprint is surrounded with red.

of the (only) reference-typed fields `left` and `right`. We say that a path $x \dots y$ is $\mathtt{g}$-*local* if $\mathtt{P}(\mathtt{g}, x, y)$ holds. Both our edge and path predicates are defined over a *mathematical abstraction* of the current heap graph (cf. Sec. 3.3.5), and are *pure* in the separation logic sense, allowing us to freely repeat them in specifications.

Our edge and path predicates enable rich reachability specifications within a method's footprint. The preconditions of `merge` express that the method footprint is acyclic and closed under the edge relation (due to the second and the last two preconditions), and that `ldag` and `rdag` contain exactly the objects reachable from `l` and `r`, resp. In general, method specifications are checked to only employ edge and path predicates whose first parameter is the method's footprint or a subset thereof.

Method postconditions typically express how reachability *changes* within this footprint. In our example, the first postcondition specifies that the result `link` is part of the first DAG and its `right`-field was initially null. The **`old`**-expression allows postconditions to refer to pre-state values; we write $\mathtt{E}_0(\dots)$ to abbreviate **`old`**$(\mathtt{E}(\dots))$, and analogously for `P`. We can freely mix reachability specifications with specifications in terms of the program heap (e. g. the `link.right` expression). The other postconditions illustrate how we can specify the new edge and path relations in terms of their originals, summarizing the method's effect. In particular, the last postcondition expresses that an object x reaches an object y in the post-state *iff* it reached y already in the pre-state, or if x reaches `link` in the first DAG and y is in the second DAG. Our method specification leaves `link` underspecified, whereas the implementation chooses the rightmost node in the first DAG. We could easily provide a less abstract specification by using path predicates over (only) the `right`-field.

3.3.3 Footprint selectors

Method calls in our technique, such as the recursive call in method `merge`, need to supply values for their ghost parameters; each method must have at least the ghost parameter $\mathfrak{g}$ that specifies its footprint. To provide such values, we use *footprint selectors*, i. e. functions defining new ***Graph***s. For instance in `merge`, we use the predefined function `sub(`$\mathfrak{g}$`:`***Graph***`,`$\mathfrak{h}$`:`***Graph***`,root:`***Node***`)` that yields the subset of $\mathfrak{h}$ reachable from `root` via $\mathfrak{g}$-local paths. The properties known for the resulting set are summarized by the following *heap-dependent function* [91] declaration:[19]

```
function sub(g: Graph, h: Graph, root: Node): Graph
  requires root ∈ h ∧ h ⊆ g
  ensures CLOSED_h(result) ∧ ∀n • n ∈ result ⇔ n ∈ h ∧ P(g, root, n)
```

where **result** refers to the value of the function application; $\texttt{CLOSED}_{\mathfrak{h}}(\mathfrak{r})$ denotes that an edge that exits $\mathfrak{r}$ must not end in $\mathfrak{h}$:

$$\texttt{CLOSED}_{\mathfrak{h}}(\mathfrak{r}) \;:\Longleftrightarrow\; \forall x \in \mathfrak{r},\, y \bullet \texttt{E}(\mathfrak{r}, x, y) \implies y \notin \mathfrak{h} \setminus \mathfrak{r} \tag{3.4}$$

Note that $\texttt{CLOSED}_{\mathfrak{h}}(\mathfrak{r})$ is permissive enough to allow selecting new footprints for method calls even if the current footprint is *open*, i. e. if there exist edges that exit the current footprint. To specify that a subheap is closed *in the global heap*, we would use a stronger condition:

$$\texttt{CLOSED}(\mathfrak{g}) \;:\Longleftrightarrow\; \forall x \in \mathfrak{g}, y \notin \mathfrak{g} \;\bullet\; \neg\texttt{E}(\mathfrak{g}, x, y) \tag{3.5}$$

In addition to `sub`, our technique can be used with other footprint selectors, and footprints may be under-constrained. The function `some` defines the weakest (deterministic) footprint selector:

```
function some(g: Graph, schema: Int): Graph
  ensures result ⊆ g
```

where the first argument, $\mathfrak{g}$, is the footprint of the current method and the second argument, `schema`, specifies *which* of the possible subsets of $\mathfrak{g}$ should be yielded; it is used for differentiating multiple applications of `some`. For example, in the following code snippet, the verifier will consider both branches:

```
var i, j: Int
if (some(g, i) == some(g, j)) { /* Case 1: identical footprints */ }
else { /* Case 2: possibly different footprints */ }
```

because neither the equality nor the disequality of `i` and `j` is known. A frontend tool could provide a ghost statement $\mathfrak{h}$`:= fresh(`$\mathfrak{g}$`)` that applies `some` to the current footprint $\mathfrak{g}$, and an implicit fresh integer `schema`, returning the freshly selected sub-footprints, $\mathfrak{h}$.

19 Similar to methods, functions in our language have implicit footprints specified via ISC over all elements in their ghost parameter $\mathfrak{g}$. However, functions are guaranteed to be side-effect free. Hence, we do not distinguish between P and $\texttt{P}_0$ in the postcondition of `sub` (similar for E and $\texttt{E}_0$).

All new footprints can be selected via the function `some`, possibly with additional arguments and in combination with additional constraints on the shape of the resulting sub-footprint, e. g. the ones in the postcondition of `sub`. The latter must be independently verified to guarantee soundness of the verification of the entire method. Concretely, one must prove that a sub-footprint is *well-defined*, i. e. it must represent a subset of $\mathfrak{g}$.

In practice, it is sometimes convenient to specify sub-footprints *compositionally*[20]; e. g., one can define a sub-footprint containing two disjoint sub-trees, rooted in r_1 and r_2, resp., as $\texttt{sub}(\mathfrak{g},\mathfrak{g},r_1) \uplus \texttt{sub}(\mathfrak{g},\mathfrak{g},r_2)$. However, the frontend tool must check that each individual component in the composition is independently well-defined. Other supported set operations that may be used for composing sub-footprints are **`union`**, **`setminus`**, **`intersection`**, as well as ***`Set`***`(x)` that yields a singleton set containing the node `x`.

*3.3.3.1 **Soundness proof sketch.*** We show that our footprint selector function `sub` is *well-formed* by assuming its specification and proving $\textbf{result} \subseteq \mathfrak{g}$. First, the quantified postcondition implies $\forall n \bullet n \in \textbf{result} \Rightarrow n \in \mathfrak{h}$, which is the definition of $\textbf{result} \subseteq \mathfrak{h}$. Second, due to $\mathfrak{h} \subseteq \mathfrak{g}$ from the precondition and the transitivity of $\subseteq$, we conclude $\textbf{result} \subseteq \mathfrak{g}$. Note that the property $\texttt{root} \in \textbf{result}$ follows from the same postcondition: Instantiating the quantifier with `root` for n, we get $\texttt{root} \in \textbf{result} \Leftrightarrow \texttt{root} \in \mathfrak{h} \wedge \texttt{P}(\mathfrak{g}, \texttt{root}, \texttt{root})$. The last bit in this formula is trivially true, due to the definition of `P`. Since $\texttt{root} \in \mathfrak{h}$ from the precondition, we conclude $\texttt{root} \in \textbf{result}$. □

We now show that the postcondition of $\texttt{sub}(\mathfrak{g}, \mathfrak{h}, \texttt{root})$ is *satisfiable*. Consider the following (precondition-satisfying) model: the universe of nodes $\textbf{\textit{Node}} \leftarrow \{\alpha, \texttt{root}\}$, $\mathfrak{g} = \mathfrak{h} \leftarrow \{\alpha, \texttt{root}\}$, $\textbf{result} = \texttt{sub}(\mathfrak{g}, \mathfrak{g}, \texttt{root}) \leftarrow \{\texttt{root}\}$, and $\alpha.\texttt{right} = \texttt{root}$ (let all other fields be assigned to **`null`**). The 1st postcondition conjunct yields $\forall x \in \mathfrak{g}, y \bullet \texttt{E}(\textbf{result}, x, y) \Rightarrow y \notin \mathfrak{g} \setminus \textbf{result}$; this quantifier is satisfied in our model as there are no heap edges originating in **`result`**. The 2nd postcondition conjunct yields $\forall n \bullet n \in \textbf{result} \Leftrightarrow n \in \mathfrak{h} \wedge \texttt{P}(\mathfrak{g}, \texttt{root}, n)$; both α and `root` satisfy this quantifier (α belongs neither to **`result`** nor to $\mathfrak{h}$; `root` belongs to both sets and is trivially reachable from itself). □

3.3.4 Local reasoning

In this section, we introduce the core ingredients of our verification technique for combining reachability information with separation logic style reasoning. Reasoning about a method starts with assuming its precondition. The precondition provides permissions to access the objects (i. e. nodes) in its footprint and the reachability constraints that guarantee the existence or the absence of heap paths connecting some objects from the footprint. As the program performs modifications to some parts of the heap, our goal is to determine a precise way of checking any (local) reachability query (e. g. in the method's

20 An example with a compositional sub-footprint will be given towards the end of this chapter, in Fig. 3.22.

postcondition) after these changes. Hence, it is important to identify the paths that were unchanged and those that were created or destroyed by each operation.

Heap modifications are performed either directly by field updates or indirectly through method calls; we postpone the discussion of the latter case until Sec. 3.5. In the former case of direct field updates, the reachability properties known to hold before the update need to be adjusted to reflect the change of heap references. For a field update, the local reachability properties before and after the update can be expressed within the same (enclosing method's) footprint.

In this section, we explain how our reachability reasoning technique is integrated with separation logic (Sec. 3.3.5) and present our technique for direct field updates (Sec. 3.4). We require the current method's footprint to satisfy the special conditions under which a precise update formula can exist for the reachability relation; in particular, cyclic and non-functional structures require special treatment.

3.3.5 Encoding of edge and path predicates

Our specification technique supports reachability via the edge predicate `E` and the path predicate `P`. In order to verify such specifications, we encode them into a flavor of separation logic and use an existing verification tool to construct proofs in that logic. We use Implicit Dynamic Frames [66] for this purpose, a variation of separation logic [64] that separates specifications of access permissions for memory locations from specifications of the values stored in these locations. For instance, separation logic's points-to predicate $x.f \mapsto v$ is specified in implicit dynamic frames as a conjunction of the access permission and the field content: $\mathbf{acc}(x.f) * x.f = v$. This separation of permissions and value properties allows us to conveniently express additional value properties, e. g. sortedness, in addition to reachability properties, without having to define a new graph-abstraction that exposes the values of interest.

Our edge predicates could be defined directly, e. g. as $(x.f_1 = v \vee x.f_2 = v)$ for two fields f_1 and f_2; conceptually, `E` is a first-order abstraction over this property, which may, in particular, be used in the *syntactic triggering patterns* [33, 47, 54, 94] that the SMT solver requires to control quantifier instantiations (and which cannot include logical operations such as $\vee$ above).

Unlike the edge predicate `E`, directly defining the path predicate `P` would compromise automation. A definition would involve transitive closure, which is notoriously difficult to handle for SMT solvers. Therefore, we take a different approach here. We leave the path predicate undefined and axiomatize its essential properties, for instance, how it is affected by heap updates. We specify these axioms over mathematical graphs and not directly over the heap-dependent edge and path predicates. Therefore, our encoding first abstracts the heap within a footprint to a set of edges (ordered pairs of nodes) and then expresses reachability over those. This abstraction is defined by a predefined function called `rsnap`. For simplicity, we define `rsnap` using the notation of our source language,

but it is only used internally by our encoding. In particular, the function implicitly depends on the heap and requires permissions to all objects in its footprint $\mathfrak{g}$:

```
function rsnap^F(g: Graph): Set[Edge]
  ensures  ∀x, y • x ∈ g ∧ y ≠ null ∧ (x.f_1 = y ∨ ... ∨ x.f_n = y)  ⟺  (x, y) ∈ result
```

Here, the return type `Set[Edge]` is the type of sets of pairs of nodes, and $F = \{f_1, \dots, f_n\}$; we omit this parameter when it is clear from the context. The postcondition can be thought of as an axiom over an uninterpreted function that defines its semantics. Note that `rsnap` also collapses edges between two objects for different field names (duplicate edges are not needed to keep track of reachability).

This abstraction function lets us define the edge predicate in a straightforward way:

$$\mathsf{E}^F(\mathfrak{g}, x, y) \;:\Longleftrightarrow\; (x, y) \in \texttt{rsnap}^F(\mathfrak{g}) \tag{3.6}$$

To avoid the issues with transitive closure mentioned above, we do not define the path predicate directly, but axiomatize the properties we need for verification. In fact, we define the path relation in terms of a function $\hat{P}$ over graphs and then axiomatize the latter, state-independent function:

$$\mathsf{P}^F(\mathfrak{g}, x, y) \;:\Longleftrightarrow\; \hat{P}(\texttt{rsnap}^F(\mathfrak{g}), x, y) \tag{3.7}$$

For this axiomatization, we carefully control the quantifier instantiation performed by SMT solvers to avoid diverging proof search. For instance, we include the axiom below, but let the solver instantiate it only to a fixed depth of unrolling $\hat{P}$ [52]. We provide the full axiomatization in Sec. 3.7.2.

$$\hat{P}(G, x, y) \;:\Longleftrightarrow\; x = y \vee \exists z \bullet (x, z) \in G \wedge \hat{P}(G, z, y) \tag{3.8}$$

3.4 REASONING ABOUT FIELD UPDATES

A field update `x.f := v` may affect reachability properties in the heap and, thus, both edge and path predicates. Since our encoding contains a precise definition of the edge predicate in terms of the underlying heap (via (3.6) and the definition of `rsnap`), the verifier can determine which edge predicates hold after a field update.

However, determining the effect of a single field update on the *path* relation is more intricate as its partial axiomatization is not sufficient to determine which predicates hold after a field assignment (e. g. because this reasoning step would require induction proofs, which SMT solvers cannot find automatically). We solve this problem by adapting an existing approach: for *acyclic* graphs (which we focus on in this section; we will discuss cyclic structures in Sec. 3.6), one can provide first-order *update formulas* that express precisely how adding or deleting a single edge affects reachability [17, 53]. For example,

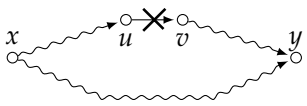


Figure 3.14: Reachability update problem in presence of alternative paths.

Direct heap edges are depicted by straight arrows, while (possibly, zero-length) heap paths are depicted by wavy arrows. The upper path $x \dots y$ *depends* on the edge (u, v); removing this edge would destroy the path, but x may still reach y after deleting (u, v), here, via the lower path. Alternative paths may occur in our setting because we permit multiple reference fields per object.

the following update formula characterizes the effect of adding an edge between nodes a and b (e. g. by initializing a field of a):

$$\forall x, y \bullet \mathsf{P}^F(\mathfrak{g}, x, y) \iff \mathsf{P}_0^F(\mathfrak{g}, x, y) \vee \mathsf{P}_0^F(\mathfrak{g}, x, a) \wedge \mathsf{P}_0^F(\mathfrak{g}, b, y) \tag{3.9}$$

where P and P_0 denote the path predicate in the states before and after the update.

The update formula for removing an edge is more complex. Since we allow for an arbitrary outdegree of nodes (via multiple reference fields), it is possible for there to exist multiple paths between two different nodes (Fig. 3.14). When adding an edge between two nodes, the new `P` relation can be updated relatively simply, e. g. via (3.9); no paths have been lost, and only paths connected by this new edge are created. On *removal* of an edge, no paths are created, but, for node pairs previously connected by a path using this edge, it is unclear whether or not they belong to the new `P` relation, due to the possibility of *alternative paths*. This entails a more-complex update formula for the edge-removal case (due to Dong and Su [17]). A general field update entails removing and then adding an edge, as we demonstrate for our `merge` example in Fig. 3.18.

Our verification technique rewrites each field update `x.`f`:= v` with a method call to an internal `update` method with the same footprint `g` as for the current method. The postconditions of `update` make the reachability update formulas available to the SMT solver. This way, we assume the update formulas for each field set F that is used in the current method specification and that contains the updated field f (reachability for other field sets is not affected by the update). We will discuss the logical encoding of reference field updates in full generality in Sec. 3.7.3.

The else-branch in the example from Fig. 3.12 modifies the heap through a single field update. The second postcondition describes the effect on the edge relation; it follows directly from the definition of the edge predicate. The third postcondition, about the path relation, is exactly the update formula (3.9), with `link` and `r` for a and b, resp.

3.5 REASONING ABOUT METHOD CALLS

In this section, we focus on modular aspects of reachability reasoning, i. e., supporting method calls. Similar to the case of direct field updates discussed in Sec. 3.3.4, a method call requires that the reachability properties known to hold before the call are adjusted. However, for method calls the situation is more complex: To determine the reachability properties after a call (the reachability framing problem), one needs to combine reachability properties before the call that are known to be outside of the call's footprint (hence, unaffected by the call) with reachability properties guaranteed by the callee method (as expressed in the callee's postcondition). These two sets of properties are expressed within the footprints of the client and the callee, respectively. If these footprints are not equal, then the reachability properties guaranteed by the callee need to be re-interpreted in the client's footprint.

In this section, we will present our techniques for tackling these challenges. We begin by introducing *relative convexity*, a novel relation between nested heap fragments that is strong enough to enable efficient re-interpretation of their reachability properties, e. g., reachability framing, but permissive enough to embrace a broad spectrum of challenging data structures (Sec. 3.5.1). Under the condition that the footprints of the callee and the client are relatively convex (which is the only requirement we impose in this section), it becomes possible to support modular reasoning about reachability in a technique that can be efficiently automated (Sec. 3.5.2). Despite the fact that the frame of the call may be *non-convex* — even if the corresponding footprints *are* relatively convex — we present a more subtle argument that our reachability framing technique is overall mathematically sound and complete (Sec. 3.5.1).

3.5.1 Method calls and relatively-convex footprints

Update formulas allow us to precisely capture the effect of adding or removing *individual* edges, which is sufficient to reason about field updates. However, reasoning modularly about method calls requires us to determine the effect of *multiple* heap updates. According to the callee's specification, we can partition the footprint $\mathfrak{g}$ of the client into the footprint $\mathfrak{h}$ of the callee and the remainder $\mathfrak{f}$ ($\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$). This remainder is the *frame* of the call and cannot be modified by the callee method. The postcondition of a callee method provides a specification of reachability information *within* its footprint $\mathfrak{h}$. The challenge is to determine the effect of the call on reachability within the (generally larger) footprint $\mathfrak{g}$ of its client. For the edge relation, this extrapolation is straightforward:

$$\forall x \in \mathfrak{f} \uplus \mathfrak{h}, y \bullet \mathsf{E}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{E}(\mathfrak{f}, x, y) \vee \mathsf{E}(\mathfrak{h}, x, y) \tag{3.10}$$

In separation logic, a method may modify any heap edges that *originate* in its footprint; hence, the predicate $\mathsf{E}(\mathfrak{h}, x, y)$ implies that x is in the footprint $\mathfrak{h}$ and $\mathsf{E}(\mathfrak{f}, x, y)$ implies

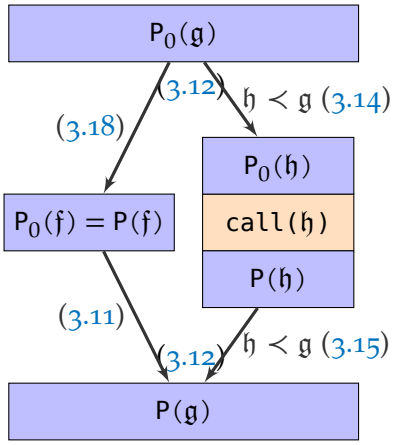


Figure 3.15: Flow of reachability information in the presence of a method call.

Reachability information in the client's footprint $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ is split into reachability information within $\mathfrak{h}$ and $\mathfrak{f}$ before the call, the effect of the call on $\mathfrak{h}$ is accounted for, and then the information is recombined to paths in $\mathfrak{g}$. The numbers in parentheses indicate ingredients of our technique explained in this section.

that x is in the frame $\mathfrak{f}$. We refer to edges that cross the boundary of the footprint as *cutpoints*:[21] if $x \in \mathfrak{h}, y \notin \mathfrak{h}$, then (x, y) is an *exit point* of the footprint, and if $x \notin \mathfrak{h}, y \in \mathfrak{h}$, then (x, y) is an *entry point* into the footprint.

Unfortunately, a simple rule such as (3.10) does not exist for relating *paths* in $\mathfrak{f} \uplus \mathfrak{h}$ to those in $\mathfrak{f}$ and $\mathfrak{h}$. A path can span fields from both heap partitions, and, in general, could cross the boundary between the two unboundedly many times. It is known that, in full generality, a first-order reachability framing formula for our path predicate cannot exist (see e. g. [76]). The key insight behind our technique for handling method calls is that this intractable situation becomes tractable if the footprint of the callee is *relatively convex* in the composed heap.

Definition 1 (Relatively Convex Footprints)**.** *In a given program state and for a given set of reference fields F, footprint $\mathfrak{h}$ defines a* relatively convex sub-footprint *of footprint $\mathfrak{g}$ (written $\mathfrak{h} \prec \mathfrak{g}$) iff $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ for some footprint $\mathfrak{f}$, and no paths within $\mathfrak{g}$ leave $\mathfrak{h}$ and then return:*

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \bullet \neg \mathsf{P}^F(\mathfrak{g}, x, u) \vee \neg \mathsf{P}^F(\mathfrak{g}, u, y)$$

We show, in the remainder of this section, how we exploit this property to enable precise, first-order, and modular reasoning about reachability in presence of method calls. In particular, we are able to make tractable the problem of framing reachability information when a method footprint $\mathfrak{h}$ is relatively convex in its client's footprint $\mathfrak{g}$. This requirement is checked by our technique at the call site, but is typically naturally the case. For

21 We introduced cutpoints in Sec. 3.1.3.

example, any method operating on a recursively-defined data type, its sub-structures or portions thereof (such as linked list segments), a strongly connected component of a potentially-cyclic structure, or combinations of these will have a relatively convex footprint. DAG traversals also have relatively convex footprints. For instance, the recursive call to `merge` in our running example of Fig. 3.12 and the corresponding illustration in Fig. 3.13 demonstrate a method call with a relatively convex footprint. Note that both acyclicity and relative convexity are defined relatively to a field set F. Therefore, even operations on data structures with back-pointers (such as parent-pointers in a tree) typically have relatively convex footprints as long as the path predicates are defined in terms of only forward-references or only back-pointers.

3.5.1.1 Method call overview A high-level overview of our solution is illustrated in Fig. 3.15. We use P_0 to represent the paths before the call, and P for those afterwards. According to standard separation logic reasoning, method calls are allowed only if the callee's footprint $\mathfrak{h}$ is a subset of the client's ($\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$, for some frame $\mathfrak{f}$). Under the additional requirement that $\mathfrak{h} \prec \mathfrak{g}$, the technique we present in this section shows how to decompose reachability information before the call (i. e. expressed in terms of $\mathsf{P}_0(\mathfrak{g}, \ldots)$) into paths in the callee's footprint ($\mathsf{P}_0(\mathfrak{h}, \ldots)$) and paths in the frame ($\mathsf{P}_0(\mathfrak{f}, \ldots)$). The callee's specification is responsible for relating $\mathsf{P}(\mathfrak{h}, \ldots)$ information to $\mathsf{P}_0(\mathfrak{h}, \ldots)$ information, i. e. specifying how reachability changes *within* the callee's footprint. Conversely, reachability purely in the *frame* $\mathfrak{f}$ cannot be changed by a call, since it does not have the permissions to do so. Indeed, based on consideration of the permissions *not passed* to the method call, we know that the following formula holds (which we call *separation-logic framing*):

$$\forall x \in \mathfrak{f}, y \bullet \mathsf{P}(\mathfrak{f}, x, y) \iff \mathsf{P}_0(\mathfrak{f}, x, y) \tag{3.11}$$

Our technique then provides means of reconstructing reachability in the client's footprint ($\mathsf{P}(\mathfrak{g}, \ldots)$) from the information we have after the call in terms of $\mathsf{P}(\mathfrak{h}, \ldots)$ and $\mathsf{P}(\mathfrak{f}, \ldots)$.

3.5.1.2 Path partitioning The first key step of our solution is *path partitioning*. We exploit relative convexity of the callee's footprint to define formulas for soundly and precisely relating reachability in a client's footprint to reachability in the callee and its frame, and vice versa. Fig. 3.16 illustrates the possibilities for a path in the client's footprint $\mathfrak{g}$ to interact with a relatively convex footprint $\mathfrak{h}$. We proceed by analyzing Fig. 3.16 by cases, deriving formulas one of which must hold in each possible case.

Crucially, our relative convexity assumption $\mathfrak{h} \prec \mathfrak{g}$ guarantees that no paths $x \ldots y$ in $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ enter or leave $\mathfrak{h}$ more than once. We summarize the five cases for paths from x to y based on the distribution of these nodes between the footprint, $\mathfrak{h}$, and the frame of the call, $\mathfrak{f}$: (i) $x, y \in \mathfrak{h}$ as is the whole path, (ii) $x \in \mathfrak{f}, y \in \mathfrak{h}$; the path crosses the boundary *once*, (iii) $x \in \mathfrak{h}, y \in \mathfrak{f}$ again crossing *once*, (iv) $x, y \in \mathfrak{f}$ with a path entering

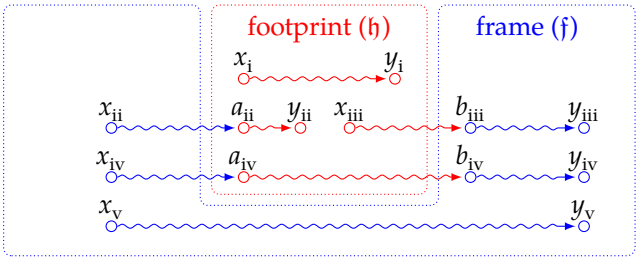


Figure 3.16: Path partitioning in presence of a relatively convex footprint.

No paths originating and ending inside a relatively convex footprint may go though nodes of its frame. Therefore, the paths that originate in the frame may enter and exit the footprint at most once. This gives five possibilities for a path to interact with a relatively convex footprint.

and leaving $\mathfrak{h}$ *once*, and (v) $x, y \in \mathfrak{f}$ with a path entirely in $\mathfrak{f}$. Note that these cases are exhaustive for a path between $x, y \in \mathfrak{g}$, due to our convexity restriction.

These five cases translate to the following formulas, allowing us to relate reachability in $\mathfrak{f} \uplus \mathfrak{h}$ and reachability in the two subheaps $\mathfrak{f}$ and $\mathfrak{h}$ individually, which we call *path partitioning formulas*:

$$
\begin{aligned}
&\text{(i)} && \forall x \in \mathfrak{h}, y \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{P}(\mathfrak{h}, x, y) \\
&\text{(ii)} && \forall x \in \mathfrak{f}, y \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \exists a \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, y) \\
&\text{(iii)} && \forall x \in \mathfrak{h}, y \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \exists b \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{h}, x, b) \wedge \mathsf{P}(\mathfrak{f}, b, y) \\
&\text{(iv)–(v)} && \forall x \in \mathfrak{f}, y \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{P}(\mathfrak{f}, x, y) \vee \\
& && \qquad \exists a \in \mathfrak{h}, b \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, b) \wedge \mathsf{P}(\mathfrak{f}, b, y)
\end{aligned}
\tag{3.12}
$$

These formulas can be used left-to-right or right-to-left. In the former case, we obtain a canonical means of *decomposing* information about paths in a composed footprint of the client into information about paths in the callee's footprint and paths in the frame. In the latter case, we obtain means of *reassembling* reachability information in the composed footprint from that in the constituent parts. In practice, we add separate **`assume`** statements for both directions of each formula, so that we can clearly specify to the underlying SMT solver when to instantiate the formula in which direction.

It is due to our relative convexity assumption that there exist simple first-order path partitioning formulas (3.12). Without this assumption, the number of cutpoints of $\mathfrak{h}$ could be *arbitrary*, and localization of reachability information would require considering an unbounded set of cases (or higher-order reasoning), preventing automatic verification.

For simplicity, formulas (3.12) cover only the cases $x, y \in \mathfrak{g}$: however, paths that are local to a particular footprint may leave that footprint by a single edge (and so case (i) above, for example, does not provide information about such paths in $\mathfrak{h}$). The following

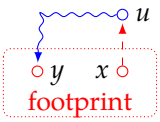


Figure 3.17: Violation of relative convexity due to the effect of an operation.

The footprint (surrounded with red) is relatively convex *before* a method call, satisfying (3.14). The heap edge created by the call is represented by a dashed arrow. The new edge (x, u) is an *exit point* of the footprint into the frame; since its end node u reaches the footprint via some path $u \dots y$, adding (x, u) violates the relative convexity property of the footprint (3.15).

formulas reduce the case $y \notin \mathfrak{g}$ to the cases already covered by introducing a node $u \in \mathfrak{g}$ with an edge to y:

$$\begin{aligned} \forall x \in \mathfrak{h}, y \notin \mathfrak{h} \bullet \mathsf{P}(\mathfrak{h}, x, y) &\iff \exists u \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{h}, x, u) \wedge \mathsf{E}(\mathfrak{h}, u, y) \\ \forall x \in \mathfrak{f}, y \notin \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f}, x, y) &\iff \exists u \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f}, x, u) \wedge \mathsf{E}(\mathfrak{f}, u, y) \end{aligned} \tag{3.13}$$

3.5.1.3 Checking relative convexity of footprints. In terms of reasoning about calls, we emit **`assume`** statements for our path partitioning formulas *both* before and after a method call (to decompose paths into those matching the callee's footprint and frame before the call, and to reconstruct information from these sources back to the client's footprint, afterwards; cf. Fig. 3.15). In both cases, before assuming our path-partitioning formulas, we first check that the footprint is relatively convex (since this property justifies their soundness); as we show in Fig. 3.17, a method's footprint could be relatively convex before the call but non-convex in the client's footprint afterwards. The two checks employed by our technique must be expressed in slightly different terms. Before the call (and without yet emitting our path-partitioning formulas) the $\mathfrak{g}$-local reachability information is available, and we directly use the formula from Def. 1:

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \bullet \neg \mathsf{P}(\mathfrak{g}, x, u) \vee \neg \mathsf{P}(\mathfrak{g}, u, y) \tag{3.14}$$

However, after the call we obtain the $\mathfrak{h}$-local reachability from the postcondition of the callee, while the $\mathfrak{f}$-local reachability is preserved. We cannot use $\mathfrak{g}$-local reachability in the post-state of the call; the aim of our path-partitioning formulas is to *deduce* information in this form, and these are only justified *after* making the convexity check. Therefore, after the method call, we use the following alternative formulation:

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \bullet \neg \mathsf{P}(\mathfrak{h}, x, u) \vee \neg \mathsf{P}(\mathfrak{f}, u, y) \tag{3.15}$$

3.5.1.4 Relating reachability information before and after a method call. Based solely on our assumption of relatively convex footprints, we now have a rich set of formulas

available for precisely relating reachability information before and after a method call. To illustrate how our formulas can be used in practice, we consider one of the verification conditions needed for verifying the postcondition in Fig. 3.12 after the recursive call to `merge`. Concretely, we consider the following Hoare triple:

$$\left\{ \texttt{l.right} \neq \textbf{null} \wedge \texttt{P}_0(\mathfrak{g}, \texttt{r}, n) \right\} \ \texttt{link := merge(l.right, r, } \mathfrak{h}\texttt{,...)} \ \left\{ \texttt{P}(\mathfrak{g}, \texttt{l}, n) \right\}$$

Here, $\mathfrak{g}$ and $\mathfrak{h}$ are the footprints of the client and the callee, resp., `l` and `r` are the roots of the left and right DAGs, resp. (Fig. 3.13), and n is *some* node reachable from `r` in the pre-state; we omit the rest of the arguments of `merge` for brevity. The condition $\texttt{l.right} \neq \textbf{null}$ comes from the **if** statement in Fig. 3.12 and holds before (and after) the recursive call; we enter this branch *iff* we have not yet found the `link` node and must keep recursively traversing the current structure. We proceed with a proof sketch for the postcondition of this Hoare triple. Other checks needed to verify the Hoare triple include relative convexity checks (3.14) and (3.15), and the precondition check before the recursive call; these require similar reasoning steps and are omitted for brevity.

The postcondition $\texttt{P}(\mathfrak{g}, \texttt{l}, n)$ expresses the existence of a path $\texttt{l} \dots n$. We justify this postcondition by showing the existence of a (single-edge) frame-local path $\texttt{l} \dots \texttt{l.right}$ and an $\mathfrak{h}$-local path $\texttt{l.right} \dots n$ (where $\mathfrak{h}$ is the footprint of the call). The former sub-path starts in $\texttt{l} \notin \mathfrak{h}$ and ends in $\texttt{l.right} \in \mathfrak{h}$ (since $\mathfrak{h} = \texttt{nldag} \uplus \texttt{rdag}$, where `nldag` was constructed via `sub`), and the latter sub-path starts in $\texttt{l.right} \in \mathfrak{h}$ and ends in $n \in \mathfrak{h}$ ($n \in \texttt{rdag}$ follows from the precondition $\texttt{P}_0(\mathfrak{g}, \texttt{r}, n)$ of the Hoare triple and the last precondition of `merge`, while $\texttt{rdag} \subseteq \mathfrak{h}$ because $\mathfrak{h} = \texttt{nldag} \uplus \texttt{rdag}$). The distribution of the starting and ending nodes of these sub-paths enables an instantiation of (ii) from (3.12) with `l`, `l.right`, n for x, a, y, resp., reducing the overall proof goal to the two predicates $\texttt{P}(\mathfrak{f}, \texttt{l}, \texttt{l.right})$ and $\texttt{P}(\mathfrak{h}, \texttt{l.right}, n)$. First, since $\texttt{l.right} \neq \textbf{null}$, the former predicate can be justified by (3.8) and the postcondition of `rsnap`. Second, we instantiate the last postcondition of `merge` with `l.right`, n for x, y, resp. in order to reduce the latter predicate to $\texttt{P}_0(\mathfrak{h}, \texttt{l.right}, \texttt{link})$ and $\texttt{P}_0(\mathfrak{h}, \texttt{r}, n)$. Note that, since the path $\texttt{l.right} \dots \texttt{link}$ starts in the root of `nldag`, the former predicate is implied by the last postcondition of `merge`. We can justify the latter predicate, $\texttt{P}_0(\mathfrak{h}, \texttt{r}, n)$, with an instantiation of (i) from (3.12) with `r`, n for x, y, resp., since (as we argued above) $\texttt{r}, n \in \mathfrak{h}$. □

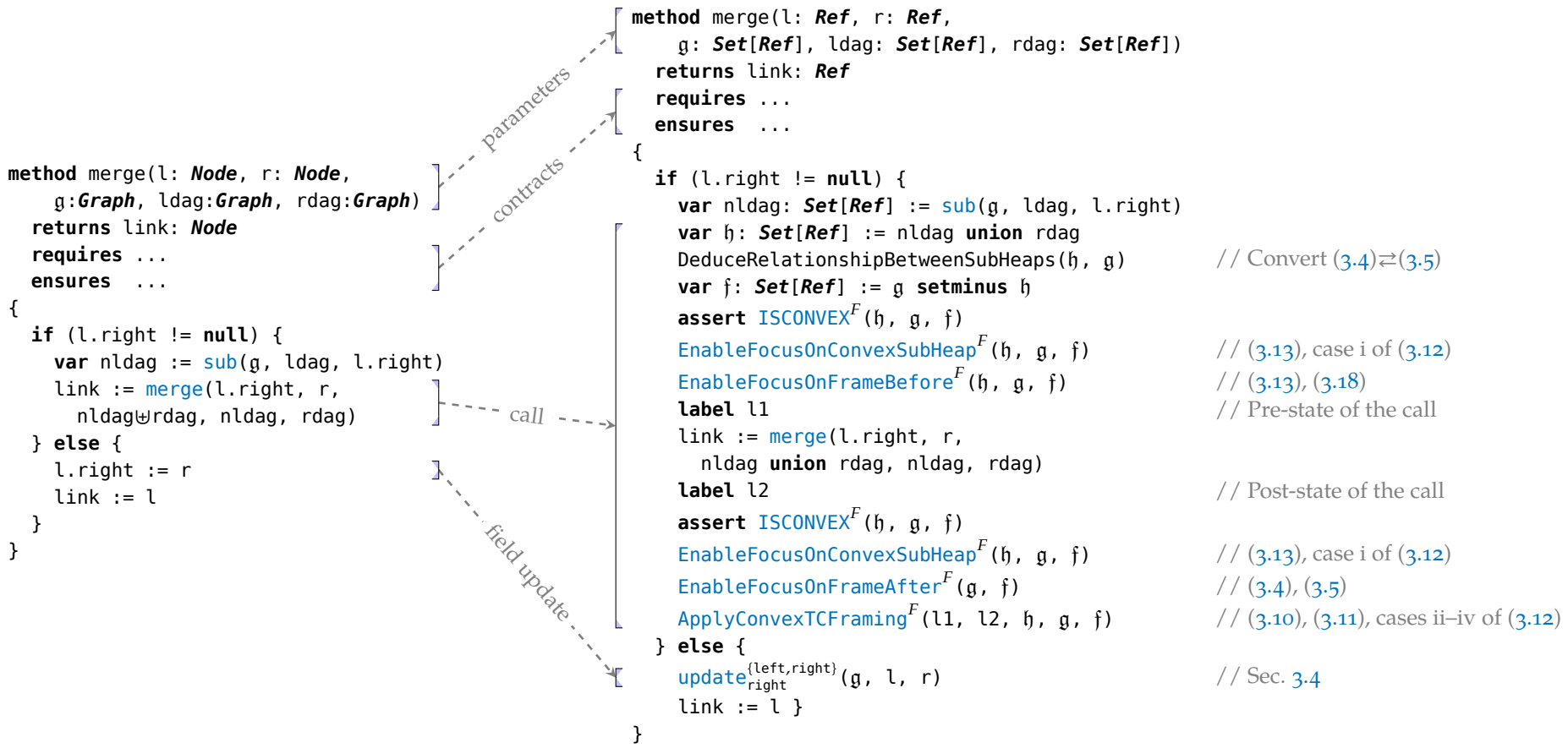


Figure 3.18: Encoding `merge` into Viper.

Types are translated directly. The specifications (Sec. 3.3.1) are omitted for brevity. The reference field update is translated via `unlinkDAG`, `linkDAG` (Sec. 3.4). The method call is augmented with *local assumptions* in the form of macros (lines starting with capital letters), constraining the states `l1`, `l2`. The macros with infix `Convex` also check relative convexity of corresponding footprints (Sec. 3.5.1). `DeduceRelationshipBetweenSubHeaps` adds terms for triggering set-based properties based on reachability predicates. The complete encoding is part of the publicly available artifact [114].

3.5.2 Frame-localized reachability

The ingredients presented thus far form the core of our solution for handling method calls, but are not yet sufficient to preserve reachability information in all cases, as we explain next. In some cases, we need to be able to localize reachability information in the *frame* of the call (cf. the left branch in Fig. 3.15). But precise frame-local reachability information cannot be obtained the same way as footprint-local reachability because, unlike method footprints, our technique permits the frame to be *non-convex* in the client's footprint. For example, consider a call to a method that operates on an acyclic list segment; the footprint of this call must be convex, while the frame would generally be non-convex in the entire list. Since the issue is subtle, we illustrate how information can be lost with a concrete example, and then show how to plug the gap.

3.5.2.1 The problematic scenario. The program in Fig. 3.19 consists of two methods: the client, `joinAndModify`, and the callee, `disconnectAll`. This program[22] concerns a particular shape of DAG structure, which we call a *hammock* between two nodes. We say that a (closed) DAG $\mathfrak{h}$ is a hammock between two (distinct) nodes s and t *iff* it consists of all nodes reachable from its node s (called the *source*) that reach its distinct node t (called the *sink*):

$$\begin{aligned}\mathtt{HAMMOCK}_{\mathfrak{g}}(\mathfrak{h},s,t) \;:\Longleftrightarrow\; & s \in \mathfrak{h} \wedge t \in \mathfrak{h} \wedge \mathtt{CLOSED}(\mathfrak{h}) \wedge \mathtt{ACYCLIC}_{\mathfrak{g}}(\mathfrak{h}) \wedge s \neq t \wedge \\ & \forall n \in \mathfrak{h} \;\bullet\; \mathtt{P}(\mathfrak{g},s,n) \wedge \mathtt{P}(\mathfrak{g},n,t)\end{aligned} \tag{3.16}$$

$$\mathtt{ACYCLIC}_{\mathfrak{g}}(\mathfrak{h}) \;:\Longleftrightarrow\; \mathfrak{h} \subseteq \mathfrak{g} \wedge \forall x,y \in \mathfrak{h} \;\bullet\; \neg\mathtt{E}(\mathfrak{g},x,y) \vee \neg\mathtt{P}(\mathfrak{g},y,x) \tag{3.17}$$

We start reasoning about `joinAndModify` in state 0, with the footprint being comprised of two (disjoint) hammocks, $\mathfrak{f}$ and $\mathfrak{h}$, where s_1 and s_2 are their sources and t_1 and t_2 are their sinks, resp. The first two operations are field updates, resulting in state 1. They join the two hammocks into one by creating exactly two edges: (s_1, s_2) and (t_2, t_1). Hence, there must exist at least two distinct paths from s_1 to t_1 in state 1: one path through the nested hammock, $\mathfrak{h}$, and one inside $\mathfrak{f}$. Note that this makes the subheap $\mathfrak{f}$ *non-convex* in $\mathfrak{g}$, even though $\mathfrak{h}$ is still relatively convex in $\mathfrak{g}$. The last operation in `joinAndModify` is a method call with a (relatively convex) footprint, $\mathfrak{h}$, which results in state 2. The callee method, `disconnectAll`, destroys all heap paths *inside* its footprint (first conjunct of the postcondition), while preserving all of its exit points (second conjunct of the postcondition). We omit the callee's implementation because the problem that we are about to explain occurs exclusively at the call site.

22 The essential property in Fig. 3.19 is that the client method *creates* heap edges inside the footprint of the callee (and not just in the frame of the call), whereas the callee *destroys* some of the paths in its footprint.

```
method joinAndModify(g: Graph,
      f: Graph, s1: Node, t1: Node,
      h: Graph, s2: Node, t2: Node)
  requires g = f ⊎ h ∧
    HAMMOCK_g(f, s1, t1) ∧ HAMMOCK_g(h, s2, t2)
    ∧ s1.left = null ∧ t2.right = null
  ensures P(g, s1, t1)
{ /* state 0 */ s1.left := s2; t2.right := t1
  /* state 1 */ disconnectAll(h)
  /* state 2 */ }
```

```
method disconnectAll(g: Graph)
  ensures (∀x, y ∈ g • P(g, x, y) ⇔ x = y) ∧
    ∀x ∈ g, y ∉ g • E(g, x, y) ⇔ E0(g, x, y)
```

The method `disconnectAll` destroys all non-trivial paths inside $\mathfrak{h}$ (exemplifying a possible destructive update to the heap structure).

Figure 3.19: Example of a non-convex frame situation.

The method `joinAndModify` first attaches the hammock $\mathfrak{h}$ to the hammock $\mathfrak{f}$, creating a larger hammock, and then calls the method `disconnectAll`, creating a frame that is non-convex in $\mathfrak{g}$. Verification of the postcondition is challenging, as it requires localizing reachability in the frame, $\mathfrak{f}$, of the call to `disconnectAll`. Fig. 3.21 illustrates a typical run of `joinAndModify`.

The postcondition of `joinAndModify` says that there *still* exists a $\mathfrak{g}$-local path $s_1 \dots t_1$ in state 2. Intuitively, this claim should hold, as these two nodes *were*, before the call to `disconnectAll`, reachable via at least one $\mathfrak{f}$-local path that could not have been destroyed as a result of the method call (because $\mathfrak{f}$ is the frame of that call). However, our path-partitioning formulas (3.12) do not capture that such a frame-local path definitely existed; we learn from cases (iv)–(v) of (3.12) only the *disjunction* describing that at least one of the two paths from s_1 to t_1, labeled "iv" and "v" in Fig. 3.16, must have existed before the call, but we do not know which. Since the call is known to destroy the paths corresponding to one disjunct, we cannot deduce $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$ after the call unless we can precisely derive frame-local reachability.

3.5.2.2 Localizing reachability in the frame of a relatively convex footprint. Fig. 3.20 demonstrates the general problem of localizing reachability information in the frame of a method call. Consider a method call with a relatively convex footprint $\mathfrak{h}$ and the frame $\mathfrak{f}$; the client's footprint $\mathfrak{g}$ is their disjoint union $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$. Our path partitioning formulas (3.12) allow us to precisely define $\mathfrak{h}$-*local* reachability based solely on $\mathfrak{g}$-local reachability. As demonstrated by our `joinAndModify` example of Fig. 3.19, we additionally need a complementary formula that would precisely define $\mathfrak{f}$-*local* reachability (again, based solely on $\mathfrak{g}$-local reachability). In other words, we are looking for a first-order formula over the relation P (with the first parameter fixed to $\mathfrak{g}$) that, for a given pair of nodes $x, y \in \mathfrak{f}$, precisely defines the existence of an $\mathfrak{f}$-local path $x \dots y$. Fortunately, such

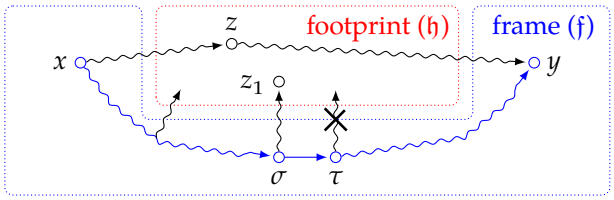


Figure 3.20: Localization of alternative paths in a non-convex frame.

Paths starting and ending in the frame of a relatively convex footprint may either be local to the frame or go through the footprint. Relative convexity of the footprint guarantees that there must be at most one entry and one exit point *per path*. Note that x and σ, as well as τ and y may possibly alias each other, but $x \neq y$ and $\sigma \neq \tau$ are guaranteed.

an *in-frame reachability localization formula* exists if $\mathfrak{f}$ is a *frame of a relatively convex footprint* $\mathfrak{h}$ (even if $\mathfrak{f}$ itself is non-convex in $\mathfrak{g}$):

$$\begin{aligned}
&\forall x, y \in \mathfrak{f} \bullet (\forall z \in \mathfrak{h} \bullet \neg \mathsf{P}(\mathfrak{g}, x, z) \vee \neg \mathsf{P}(\mathfrak{g}, z, y)) \implies (\mathsf{P}(\mathfrak{f}, x, y) \Leftrightarrow \mathsf{P}(\mathfrak{g}, x, y)) \\
&\forall x, y \in \mathfrak{f} \bullet \quad (\exists z \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, x, z) \wedge \mathsf{P}(\mathfrak{g}, z, y)) \implies \big(\mathsf{P}(\mathfrak{f}, x, y) \Leftrightarrow \exists \sigma, \tau \in \mathfrak{f} \bullet \\
&\qquad\qquad \mathsf{P}(\mathfrak{g}, x, \sigma) \wedge \mathsf{E}(\mathfrak{g}, \sigma, \tau) \wedge \mathsf{P}(\mathfrak{g}, \tau, y) \wedge \\
&\qquad\qquad (\exists z_1 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \sigma, z_1)) \wedge \neg(\exists z_2 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \tau, z_2))\big)
\end{aligned} \tag{3.18}$$

We explain and justify (3.18) using the diagram of Fig. 3.20. Generally, since $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$, we can case split on whether there exists a path $x \ldots y$ that goes through $\mathfrak{h}$, allowing us to obtain the required $\mathfrak{f}$-local reachability formula. The first formula above covers the case in which such a path *does not* exist; thus, the following must hold: $\forall z \in \mathfrak{h}.\, \neg\mathsf{P}(\mathfrak{g}, x, z) \vee \neg\mathsf{P}(\mathfrak{g}, z, y)$ (which trivially holds for all $x, y \in \mathfrak{f}$ in the special case when $\mathfrak{f}$ *is* convex in $\mathfrak{g}$). The second formula above says that, if there exists a path through $\mathfrak{h}$ (the upper kind of path in Fig. 3.20), it must pass through some node $z \in \mathfrak{h}$; hence, the following condition must hold: $\exists z \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, x, z) \wedge \mathsf{P}(\mathfrak{g}, z, y)$. Under this condition, we must define the existence of an $\mathfrak{f}$-local path that *also* connects $x \ldots y$. The key idea that we exploit to justify the second formula in (3.18) is to use our relative convexity assumption to justify a *three-way* split of the (hypothetical) $\mathfrak{f}$-local path $x \ldots y$ into three segments: a path $\mathsf{P}(\mathfrak{f}, x, \sigma)$, an edge $\mathsf{E}(\mathfrak{f}, \sigma, \tau)$, and a path $\mathsf{P}(\mathfrak{f}, \tau, y)$ (the lower kind of path in Fig. 3.20). Furthermore, we choose (σ, τ) such that σ is *the last node that reaches* $\mathfrak{h}$ ($\exists z_1 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \sigma, z_1)$) and τ is *the first node that does not reach* $\mathfrak{h}$ ($\neg\exists z_2 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \tau, z_2)$). Under our assumptions about x and y, this requirement can always be satisfied because the footprint of the callee, $\mathfrak{h}$, is reachable from (at least) the node x and is unreachable from (at least) the node y.

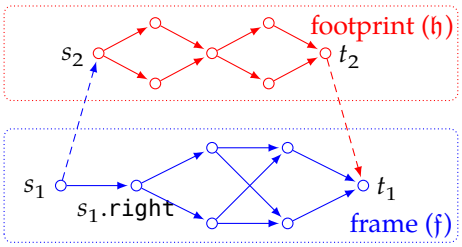


Figure 3.21: Example scenario of running the method `joinAndModify`.

In state 0, only the solid edges exist. In state 1, the client has created two new edges: (s_1, s_2) and (t_2, t_1). In state 2, the callee has destroyed all solid-red edges, but there still exists a path $s_1 \dots t_1$ via solid-blue edges. However, we cannot deduce the existence of this path using just the path partitioning formulas alone due to the disjunction in case (iv)–(v) of (3.12) that does not allow us to distinguish whether *all* paths between s_1 and t_1 were passing through the footprint in state 1. Therefore, recovering this bit after the call to `disconnectAll` requires precise localization of reachability information in the frame (3.18).

We summarize the conditions under which the predicates defining the three-way split of our hypothetical path $x \dots (\sigma, \tau) \dots y$ can be rewritten with $\mathfrak{g}$ instead of $\mathfrak{f}$, without losing precision:

- The footprint of the call, $\mathfrak{h}$, is convex in the client's footprint, $\mathfrak{g}$.
- Both nodes x and y are in the frame of the call, $\mathfrak{f}$.
- We picked σ that reaches $\mathfrak{h}$ and τ that does not reach $\mathfrak{h}$ s.t. (σ, τ) is on the path $x \dots y$.

For the first predicate, we need to argue by contradiction: suppose that $\mathsf{P}(\mathfrak{g}, x, \sigma)$ were the case, but $\mathsf{P}(\mathfrak{f}, x, \sigma)$ were not (the opposite implication is direct, since $\mathfrak{f} \subseteq \mathfrak{g}$). Then, the path from x to σ must visit the callee's footprint, $\mathfrak{h}$. However, by construction, σ is known to have a path to some node in the callee's footprint, and this violates the assumption that this footprint is relatively convex. Hence, we get $\mathsf{P}(\mathfrak{f}, x, \sigma) = \mathsf{P}(\mathfrak{g}, x, \sigma)$. The second predicate is easiest: a single edge between two nodes in the frame ($\mathfrak{f}$) can only depend on the frame itself; therefore, we get $\mathsf{E}(\mathfrak{f}, \sigma, \tau) = \mathsf{E}(\mathfrak{g}, \sigma, \tau)$. The third predicate expresses the existence of a path $\tau \dots y$; since we have picked τ s.t. it does not reach the footprint, such a path exists in this case exactly when it exists in the frame, giving $\mathsf{P}(\mathfrak{f}, \tau, y) = \mathsf{P}(\mathfrak{g}, \tau, y)$. Thus, our construction of σ and τ, along with our relative convexity property for method footprints, allows us to justify the formulation in (3.18). These formulas now provide the missing ingredient for our technique that complements our path-partitioning formulas of (3.12). □

*3.5.2.3 **Revisiting the problematic scenario.*** We return to our `joinAndModify` method, and show that we can now verify the last conjunct of its postcondition. Previously, we

were unable to verify $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$ after the call to `disconnectAll`, while intuitively, a $\mathfrak{g}$-local path $s_1 \dots t_1$ exists in state 2, because the method call could not have destroyed the existing *frame*-local path $s_1 \dots t_1$ that existed in state 1 (Fig. 3.21). Thus, if we could deduce $\mathsf{P}_1(\mathfrak{f}, s_1, t_1)$ (before the call to `disconnectAll`), we would obtain our proof goal. This is now possible using the second equation from (3.18): instantiating s_1 for x and t_1 for y, we deduce the hypothesis of the implication, since before the call to `disconnectAll` we can deduce that paths from s_1 to t_1 exist passing through the footprint. To deduce $\mathsf{P}_1(\mathfrak{f}, s_1, t_1)$ from our formula, we need to obtain the following property (recall that $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$):

$$\exists \sigma, \tau \in \mathfrak{f}.\, \mathsf{P}(\mathfrak{g}, s_1, \sigma) \wedge \mathsf{E}(\mathfrak{g}, \sigma, \tau) \wedge \mathsf{P}(\mathfrak{g}, \tau, t_1) \wedge (\exists z_1 \in \mathfrak{h}.\mathsf{P}(\mathfrak{g}, \sigma, z_1)) \wedge \neg(\exists z_2 \in \mathfrak{h}.\mathsf{P}(\mathfrak{g}, \tau, z_2))$$

The existentially-quantified pair (σ, τ) can be witnessed by $(s_1, s_1.\texttt{right})$. From this, and our hammock properties (3.16), all conditions above follow directly, allowing us to deduce our intermediate proof goals, $\mathsf{P}_{1,2}(\mathfrak{f}, s_1, t_1)$, and use the last case of (3.12) to deduce the ultimate proof goal, $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$. □

Together with (3.18), the ingredients of our technique presented in Fig. 3.15 empower *completely general, precise* reasoning about reachability in the presence of method calls with relatively convex footprints. Note that, in examples where stronger properties are known about a method's footprint, our formulas from (3.18) reduce to much simpler criteria. In particular, if a method operates on a *closed* data structure (no paths leave the footprint), we can always apply the first of our formulas; the full expressiveness of our conditions is required only in the presence of potential paths crossing the callee's footprint (e. g. Fig. 3.21). Our technique is complete, provided that callee postconditions specify sufficient information about reachability within their footprint. However, even in examples where this information is incomplete, our technique is applicable and provides useful information at the call site, for instance, by deducing which frame-local paths will be preserved across a method call. It is the restriction to method calls with relatively convex footprints which enables us to express appropriate formulas to preserve this information; without this restriction, we would not be able to precisely define the existence of frame-local paths exclusively via coarser reachability information. Finally, we note on the efficiency of formulas (3.18): in our encoding (demonstrated in Fig. 3.18 and validated in Sec. 3.8), we supply appropriate triggers for the universal quantifiers and Skolemize the existential quantifiers.

This completes our treatment of acyclic graphs and method calls; the latter is the most complex part of our technique, and applies equally to the cyclic case, which we tackle in the next section.

3.6 REACHABILITY IN CYCLIC STRUCTURES

In the previous section, we presented our technique for enabling modular reasoning about heap reachability in combination with first-order separation logic. The presented technique operates under two key restrictions: (1) that method footprints are always relatively convex in their client's footprint and (2) that all footprints used contain acyclic graphs. These are two *independent* criteria, which our technique checks where necessary. Restriction (1) alone enables our handling of method calls. In this section, we show that we can adapt our technique to a particular setting in which restriction (2) is dropped: that of general *0–1-path graphs*. A graph is called a 0–1-path graph (hereafter, ZOPG) if there exists *at most one* (non-trivial) path (modulo cycles) between all pairs of nodes in the graph; for instance, $\{(a,b),(b,c),(c,a),(c,d)\}$ is a ZOPG, but $\{(a,b),(b,a),(a,c),(c,a)\}$ is not since there are *two* distinct (non-trivial) paths from a to itself [17]. Although this notion does not permit *arbitrary* cyclic graphs, the technique presented in this section allows us to adapt our work to reason about reachability in the presence of potentially-cyclic lists in the heap or more-complex data structures consisting of these, including, for example: trees where the children of each node are stored in a cyclic list (e. g. using Java LinkedList), generalized tree-like structures in which some nodes consist of rings, and the ring representation of heap-ordered trees [13]. Therefore, the ZOPG class is an important generalization of (potentially-cyclic) singly-linked lists, which is the class handled in the closest prior work [76].

Extending our technique to ZOPGs requires a new way of handling direct field updates (Sec. 3.6.1); our handling in Sec. 3.5 depended on acyclicity, and a way to retain that certain graphs in the program *are* ZOPGs; modifying a ZOPG by adding an edge could violate the ZOPG invariant (Sec. 3.6.2). Note that our requirement of relatively convex footprints (Def. 1) is again crucial, enabling an efficient solution of the latter problem.

3.6.1 Field updates in ZOPGs

To support direct field updates, we adapt prior work [17] that shows how to precisely update a more-refined reachability relation called DEP for ZOPGs. There are few changes in our adaptation: our DEP relation is compatible with the *reflexive* reachability relation (P), whereas Dong and Su work with irreflexive reachability, and we parameterize our DEP relation with two extra parameters, F and $\mathfrak{g}$, supporting separation-logic reasoning, as we did for P in Sec. 3.3.2. The predicate $\mathtt{DEP}^F(\mathfrak{g},x,y,u,v)$ expresses the existence of a (non-trivial) path of field references from x to y such that all objects on the path (except possibly y) are in $\mathfrak{g}$, all fields are in F, and the path *depends on* the edge (u,v). Intuitively, this means that removing (u,v) from the graph will destroy the path $x \dots y$ (which is the unique path from x to y in a ZOPG). We will omit the parameter F when it is clear from the context. Note that $\mathtt{DEP}(\mathfrak{g},x,y,u,v) \Rightarrow u \neq v$ since an edge (v,v) cannot be a

dependency of any path: deleting such an edge would not affect reachability. Note also that $\texttt{DEP}(\mathfrak{g}, x, y, u, v) \Rightarrow x \neq y$ since a trivial path $x \ldots x$ does not depend on any edges.

Although precisely updating the classical reachability relation P in potentially-cyclic graphs after destructive heap operations is beyond first-order logic (and cannot be efficiently automated), the information about the DEP relation *can* be updated precisely and efficiently after such destructive operations [17]. For example, if the edge (s, t) is deleted by executing the statement `s.adj := null` in a method with footprint $\mathfrak{g}$, then the new relation, DEP, can be simply expressed via the old relation $\texttt{DEP}_0$ as follows:

$$\forall x, y, u, v \bullet \texttt{DEP}^F(\mathfrak{g}, x, y, u, v) \iff \texttt{DEP}_0^F(\mathfrak{g}, x, y, u, v) \wedge \neg \texttt{DEP}_0^F(\mathfrak{g}, x, y, \texttt{s}, \texttt{t}) \tag{3.19}$$

For fixed F and $\mathfrak{g}$, the intuition for (3.19) is this: (x, y, u, v) is in the new relation *iff* it was in the old relation and the deleted edge (s, t) was not a dependency of the path $x \ldots y$ before the update.

Precisely updating DEP after an operation that only creates an edge (e. g. by executing the statement `s.adj := t`) is also possible, provided one additionally checks that the newly-created edge does not violate the ZOPG invariant; we describe how this check is enforced in Sec. 3.6.2. As before, a general field update entails removing and then adding an edge (see Sec. 3.7.3). Our treatment of the DEP relation is similar to the treatment of the P relation described in Sec. 3.4: since the mathematical definitions of these relations are beyond first-order logic, we provide the verifier with a *partial* axiomatization (see [113, App. C]). We rewrite each field update with a call to an internal `updateZOPG` method with the same footprint $\mathfrak{g}$ as for the current method; the postconditions of `updateZOPG` make the DEP update formulas (e. g. (3.19)) available to the SMT solver.

A technical difference between our reachability relation P and the DEP relation is that the latter carries richer information (in particular, knowledge of every edge on which each path depends). Conversely, it seems unlikely that having to enumerate all edge facts in a graph would be suitable for a method specification; the abstraction provided by P is typically desirable. Thus, we do not provide DEP as a primitive in our specifications, and instead provide a means of converting between information in one relation and the other, while losing as little information as possible. Our conversion rules are based on the following main axiom:

$$\forall \mathfrak{h}, x, y \bullet \texttt{P}^F(\mathfrak{h}, x, y) \wedge x \neq y \iff \exists u, v \bullet \texttt{DEP}^F(\mathfrak{h}, x, y, u, v) \tag{3.20}$$

Unlike the update formulas that are emitted for concrete method footprints, our conversion axioms (e. g. (3.20)) *quantify* over the footprint ($\mathfrak{h}$); as before, we carefully select the triggers for these axioms to guide the SMT solver's quantifier instantiation procedure.

In general, formula (3.20) does not capture full information in principle expressible with the DEP relation; intuitively, this is because a single path $x \ldots y$ (described by the LHS) may depend on multiple edges, all of which match the RHS existential quantifier. To partially mitigate this fact, we augment our axiomatization with a number of addi-

tional properties. For instance, one can easily prove the following formula (an axiom in our technique) about ZOPGs, providing (for fixed F and $\mathfrak{g}$) some quadruples which *do not* belong to the DEP relation:

$$\forall \mathfrak{h}, u, v, w \bullet \neg \mathtt{DEP}^F(\mathfrak{h}, v, w, u, v) \tag{3.21}$$

Note that if $v = w$, $\neg\mathtt{DEP}(\mathfrak{h}, v, v, u, v)$ holds because a trivial path $v \dots v$ does not depend on any edges. Assume $v \neq w$. There can be at most one (cycle-free) path from v to w in a ZOPG. If there are no paths from v to w, then we get $\neg\mathtt{DEP}(\mathfrak{h}, v, w, u, v)$ from (3.20). Otherwise, the edge (u, v) is not part of the (cycle-free) path $v \dots w$ and cannot be one of its dependencies. □

Our ZOPG axiomatization is based on a set of formulas like (3.21) that, together with (3.20), help reasoning about the DEP relation (we provide the full axiomatization in [113, App. C]). Equipped with this conversion between relations P and DEP, precise reachability information is preserved in all cases that we have observed. This is interesting because the DEP relation carries more information than the transitive relation P, so (for fixed F and $\mathfrak{g}$) not all quadruples (x, y, u, v) in DEP can be extracted precisely from P, even if all pairs (x, y) in P are known. Intuitively, these missing quadruples appear not to be needed in practice because the overall proof goals are phrased in terms of just P (and not DEP).

We illustrate how reachability information is preserved throughout the transformations between P and DEP with a concrete example. Consider the following Hoare triple that describes a heap update in a ZOPG with footprint $\mathfrak{g}$ and a single reference field next:

$$\left\{ \begin{array}{l} \mathtt{x}, \mathtt{y} \in \mathfrak{g} \wedge \mathtt{x.next} = \mathtt{y} \wedge \\ \forall n, m \in \mathfrak{g} \bullet \mathtt{P}_0(\mathfrak{g}, n, m) \end{array} \right\} \ \mathtt{x.next := null} \ \left\{ \ \forall m \in \mathfrak{g} \bullet \mathtt{P}(\mathfrak{g}, \mathtt{y}, m) \ \right\}$$

We can justify the postcondition assertion as follows. Consider an arbitrary node $m \in \mathfrak{g}$. If $m = \mathtt{y}$, then we trivially get $\mathtt{P}(\mathfrak{g}, \mathtt{y}, \mathtt{y})$. Otherwise, we assume $m \neq \mathtt{y}$, and the remaining proof obligation is $\mathtt{P}(\mathfrak{g}, \mathtt{y}, m)$; to justify this, we need to exploit information from the pre-state. Since by (3.20), we can reduce the current proof obligation to $\mathtt{DEP}(\mathfrak{g}, \mathtt{y}, m, \mathtt{y}, \mathtt{y.next})$, we can instantiate the DEP update formula (3.19), obtaining two pre-state conditions: $\mathtt{DEP}_0(\mathfrak{g}, \mathtt{y}, m, \mathtt{y}, \mathtt{y.next})$ and $\neg\mathtt{DEP}_0(\mathfrak{g}, \mathtt{y}, m, \mathtt{x}, \mathtt{y})$. The former is justified by the precondition quantifier (providing $\mathtt{P}_0(\mathfrak{g}, \mathtt{y}, m)$) and the main conversion axiom (3.20), whereas the latter can be obtained directly from the additional conversion axiom (3.21). □

This example, as well as our evaluation (Sec. 3.8), show that necessary reachability information can be fully recovered after the following steps: first, conversion from P to DEP, second, application of update formulas for the DEP relation, and third, conversion from DEP to P. We plan to investigate as future work the extent to which this approach is always precise for preserving reachability information of this kind.

3.6.2 Preservation of the ZOPG invariant

To justify the handling of field updates from the previous subsection, we require knowledge that the graph being updated is a ZOPG. Since this fact can be violated by changes to the heap, an important question is how we can know if the ZOPG invariant holds. It can be expressed in first-order logic with the combination of edge and path predicates as follows:

$$\begin{aligned}\mathtt{ZOPG}(\mathfrak{h}) :\Longleftrightarrow (\forall x_1, x_2, a, b \in \mathfrak{h}, y \ \bullet \ & (x_1 \neq x_2 \vee a \neq b) \wedge \mathsf{P}(\mathfrak{h}, x_1, x_2) \wedge \mathsf{P}(\mathfrak{h}, x_2, x_1) \wedge \\ & \mathsf{E}(\mathfrak{h}, x_1, a) \wedge \neg \mathsf{P}(\mathfrak{h}, a, x_1) \wedge \\ & \mathsf{E}(\mathfrak{h}, x_2, b) \wedge \neg \mathsf{P}(\mathfrak{h}, b, x_2) \implies \neg \mathsf{P}(\mathfrak{h}, a, y) \vee \neg \mathsf{P}(\mathfrak{h}, b, y)) \ \wedge \\ \forall x, a, b \in \mathfrak{h} \ \bullet \ & a \neq x \wedge b \neq x \wedge \mathsf{E}(\mathfrak{h}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, x) \wedge \\ & \mathsf{E}(\mathfrak{h}, x, b) \wedge \mathsf{P}(\mathfrak{h}, b, x) \implies a = b\end{aligned} \tag{3.22}$$

The first conjunct of the formula expresses a situation in which two (potentially aliasing) nodes x_1 and x_2 are on the same strongly-connected component (SCC), and two edges (starting in x_1 and x_2) that are *different*—at least by source or target—end in nodes a and b, resp., outside of the SCC (a and b may alias unless $x_1 = x_2$). In such a case, it is forbidden that any node y is reachable from both a and b (this would form two different paths from the SCC to y). The second conjunct restricts the structure of SCCs themselves: no two *different* edges may start in x and stay within the same SCC.

Intuitively, formula (3.22) is hard to automate because it uses a non-trivial combination of edge and reachability predicates. Establishing $\mathtt{ZOPG}(\mathfrak{h})$ would require, for example, the information about all path splits in $\mathfrak{h}$, i. e. all nodes $x \in \mathfrak{h}$ s.t. $\exists a, b \in \mathfrak{h} \ \bullet \ a \neq b \wedge \mathsf{E}(\mathfrak{h}, x, a) \wedge \mathsf{E}(\mathfrak{h}, x, b)$. Such details ultimately require specifications to enumerate edges in the graph, which is impractical, and breaks the abstraction that reachability specifications grant. Even if the full information about the edge relation were present, establishing $\mathtt{ZOPG}(\mathfrak{h})$ would require an induction proof that is beyond the power of modern SMT solvers. Instead of checking this invariant from scratch, we design a mechanism for checking that the ZOPG invariant is *preserved* across changes to the heap.

3.6.2.1 Extending the specification language for potentially-cyclic footprints. As a first step, we introduce an additional annotation in our specification language, so that we can label certain method footprints as ZOPGs. In addition to general graphs (whose structure is only constrained by other specifications), such as $\mathfrak{h}$: ***Graph***, we allow the footprints of some methods to be more specifically marked as ZOPGs, using the syntax $\mathfrak{g}$: ***Zopg***. For method footprints declared this way, we will explain the additional proof obligations necessary to check that we *maintain* the ZOPG invariant. In particular, a method with footprint $\mathfrak{g}$: ***Zopg*** can be translated to a method with footprint $\mathfrak{g}$: ***Graph*** with additional *ZOPG proof obligations*.

```
method testZopgObligations(g: Zopg, R: Graph, r: Node, u: Node)
  requires {u} ⊎ R ⊆ g ∧ CLOSED({u}) ∧ RING_g(R) ∧ r ∈ R ∧
           ∀x ∈ g,y • P(g,x,y) ∧ P(g,x,u) ⟹ ¬P(g,r,y)   // (Pre)
{
    var h: Zopg := {u} ⊎ R
    ringInsert(h, R, r, u)
}

method ringInsert(g: Zopg, R: Graph, r: Node, u: Node)
  requires g = {u} ⊎ R ∧ CLOSED({u}) ∧ RING_g(R) ∧ r ∈ R
  ensures RING_g(g) ∧ (∀n ∉ g • P(g,u,n) ⇔ P_0(g,r,n)) ∧
          ∀x ∈ g,y • x ≠ u ∧ y ≠ u ⟹ (P(g,x,y) ⇔ P_0(g,x,y))
{
    u.next := r.next
    r.next := u
}
```

$$
\begin{aligned}
\mathtt{RING}_g(h) &:\Longleftrightarrow \mathtt{FUNCTIONAL}(h) \wedge \mathtt{UNSHARED}(h) \wedge \mathtt{SCC}_g(h) \\
\mathtt{SCC}_g(h) &:\Longleftrightarrow \forall x,y \in h \bullet \mathtt{P}(g,x,y) \\
\mathtt{FUNCTIONAL}(h) &:\Longleftrightarrow \forall a,b,c \in h \bullet \mathtt{E}(h,a,b) \wedge \mathtt{E}(h,a,c) \implies b = c \\
\mathtt{UNSHARED}(h) &:\Longleftrightarrow \forall a,b,c \in h \bullet \mathtt{E}(h,a,c) \wedge \mathtt{E}(h,b,c) \implies a = b
\end{aligned}
\tag{3.23}
$$

Figure 3.22: Example client with a ZOPG footprint.

The example demonstrates modular reasoning about reachability properties in presence of heap cycles. The client method `testZopgObligations` has no postconditions, for simplicity. In order to verify the client, we must nonetheless prove that the ZOPG invariant is maintained after the call to `ringInsert`. The definition of `RING` is given in (3.23). Note the different meaning of footprint parameters written in subscripts vs. those written in parentheses (e. g. g and h, resp., in the definition of $\mathtt{SCC}_g(h)$): the former are used as arguments for the `E` and `P` predicates, whereas the latter are used for restricting the domain of quantification.

We illustrate the generation of ZOPG proof obligations based on the example in Fig. 3.22. The client, `testZopgObligations`, operates on a ZOPG g that includes two disjoint parts: a ring R and a (closed) singleton graph consisting of just one node, u. The extra node r denotes an arbitrary node of the ring. The only operation performed by the client is a call to `ringInsert`. To verify that g remains a ZOPG by the end of `testZopgObligations`, we need to check that the callee does not create alternative paths—not just in its footprint, h (which is guaranteed to remain a ZOPG, as the methods with footprints marked by ***Zopg*** are locally checked to preserve this property), but also in the larger subheap, g. The callee `ringInsert`, operates on a ZOPG that *equals* the union of two disjoint parts: a closed singleton graph u and a ring R (these two parts must be mutually-unreachable in the pre-state). The callee attaches u to the ring R, resulting in a larger ring, $u \uplus R$. The

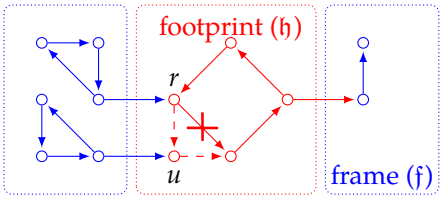


Figure 3.23: Typical scenario of running `testZopgObligations`.

An example heap configuration before the call in depicted via the solid edges, where r is an arbitrary node of the ring, and u is added to the ring after the call to `ringInsert`. The diagram demonstrates a data structure with nodes that can have two reference fields (for simplicity, the implementation of `ringInsert` shows a single field, `next`). The second conjunct in the client's precondition says that no (non-trivial) paths may originate from u (there may be paths ending in u). The footprint may have both entry and exit points, but (as required by the last conjunct in the client's precondition) each connected component of the frame may have *at most one* entry or exit point into the footprint; otherwise, the ZOPG invariant would be violated by the call.

callee's postcondition says that (in the post-state) its entire footprint is a ring (thus, all pairs of footprint nodes are mutually reachable), and precisely defines its local reachability. Local paths that end *outside* of the callee's footprint (i. e. *outgoing paths*) are defined by the last two conjuncts: the former says that all exit points reachable from the ring in the pre-state are exactly the exit points reachable from u in the post-state, whereas the latter preserves all outgoing paths of the initial ring, i. e. *all* exit points of the footprint in the pre-state where $\{u\}$ was closed. Fig. 3.23 illustrates the client's footprint in a state after `ringInsert` has executed.

3.6.2.2 Maintaining the ZOPG invariant after a field update. The knowledge that a subheap was a ZOPG in the pre-state of an operation helps checking that that subheap is still a ZOPG in the post-state, as we show next. We translate a general field update `u.next := v` in a method with the footprint `g: Zopg` to `u.next := null; u.next := v`, where (assuming `v` is not null) the first update deletes an edge and the second one creates a new edge. Deleting edges does not alter the graph class of `g`. However, a newly added edge may create an alternative path between some nodes of the graph. Concretely, new paths will be created between all pairs of nodes (x, y) s.t. there exist two paths: $x \dots \mathsf{u}$ and $\mathsf{v} \dots y$. Therefore, we get the following soundness criterion (emitted as a proof obligation before the second update) for a field update in a ZOPG:

$$\mathsf{u} \neq \mathsf{v} \implies \forall x \in \mathsf{g}, y \bullet \mathsf{P}(\mathsf{g}, x, \mathsf{u}) \wedge \mathsf{P}(\mathsf{g}, \mathsf{v}, y) \implies \neg \mathsf{P}(\mathsf{g}, x, y) \tag{3.24}$$

Note that y may be outside of the current method's footprint because a g-local path may leave g (*iff* its last edge leaves that subheap). The formula (3.24) is much simpler than, e. g. (3.22) because it is (a) an *incremental* condition (we used the knowledge that the subheap *was* a ZOPG before the update; otherwise, we would need to consider alterna-

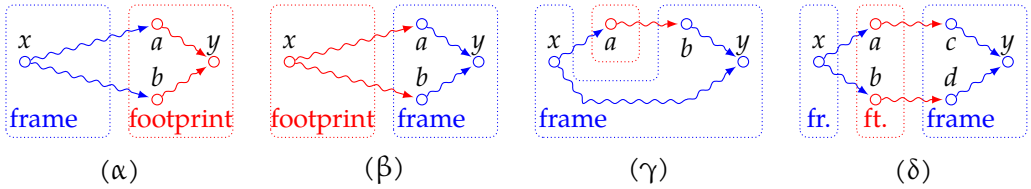


Figure 3.24: Enumeration of heap configurations violating the ZOPG invariant.

There are four possible heap configurations that violate the ZOPG invariant after a method call with a relatively convex ZOPG footprint (in red) and a ZOPG frame (in blue). Violation of this invariant means the existence of at least two distinct, alternative paths, say $x \dots y$. (α) Both alternative paths start in the frame and end in the footprint, entering the footprint via distinct entry points: $a \neq b$. (β) Both alternative paths start in the footprint and end in the frame, exiting into the frame via distinct exit points: $a \neq b$. (γ) Both paths start and end in the frame; the path above passes through the footprint while the path below stays entirely inside the frame. (δ) Both paths start and end in the frame, both passing through the footprint; however, they enter the footprint via distinct entry points and exit into the frame again via distinct exit points: $a \neq b \wedge c \neq d$.

tive paths other than those introduced by the new creation) and (b) the operation is a field update (hence, only one edge has been added to the graph). Keeping track of graph classes in the presence of method calls is more involved, but the idea (a) is again helpful for tackling this problem.

3.6.2.3 Maintaining the ZOPG invariant after a method call. A method call may violate the ZOPG invariant at call site even if the footprint of the call remains a ZOPG (a condition which is checked locally for the callee). The condition that a method call *does not* violate the ZOPG invariant at call site is generally as hard to check as the formula (3.22) itself. Fortunately, this condition can be drastically simplified if the footprints or the callee and the client are *relatively convex*.

We proceed as follows. First, we enumerate the ways in which a method call that preserves the ZOPG invariant on its own footprint, could potentially *violate* that invariant for its client's footprint. In particular, this must be by the creation of at least one new path. A call to a method with a *convex* footprint may result in one of the four *bad heap configurations* (violating the ZOPG invariant) depicted in Fig. 3.24. Second, we conjoin the *negated* formulas (3.25), (3.26), (3.27), (3.28) characterizing these four bad configurations, comprising an efficient criterion for preserving the ZOPG invariant. Checking this criterion can be easily automated: unlike formula (3.22), our criterion requires no information about the edge relation whatsoever. Our technique encodes this criterion as a proof obligation for the client. Finally, we sketch a proof of completeness for the four cases in Fig. 3.24.

The bad configuration in Fig. 3.24 (α) corresponds to a scenario in which the method call has created an alternative path from x to y, where the former node does not belong to the callee's footprint. We can describe this configuration via the following formula:

$$\begin{aligned}\exists x \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{f} \bullet\ & a \neq b \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, x, b) \\ & \wedge \mathsf{P}(\mathfrak{h}, a, y) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, y) \wedge \mathsf{P}(\mathfrak{h}, b, y)\end{aligned} \tag{3.25}$$

The symbols P_0 and P denote the reachability relation before and after the method call; $\mathfrak{h}$ is the callee's footprint; $\mathfrak{f}$ is the frame of the call. We evaluate the first two reachability predicates in the old state because frame-local reachability is not affected by the call. The information about the last three predicates comes from the postcondition of the callee[23]. We assume w.l.o.g. that $a \dots y$ has been *newly created* by the call (whereas $b \dots y$ may have existed before the call). Both paths could not have existed before the call, as that would contradict our assumption that $\mathfrak{g}$ was a ZOPG.

Returning to our example of Fig. 3.22, we observe that the precondition of `testZopgObligations` is strong enough to prevent the bad configuration (α) after the call to `ringInsert`; we prove this by contradiction. Assume that, while preserving its local ZOPG invariant, the call results in the bad configuration (α) for some $x \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{f}$; thus, we learn the conjuncts (say, #1 to #6) from the body of (3.25). Note that a and b must be distinct (due to #1) and cannot *both* be in R (due to #4 and #6; otherwise, there would be alternative paths, violating the ZOPG invariant of the callee's footprint in the post-state). We draw the contradiction by instantiating the last conjunct, (Pre), of the precondition of `testZopgObligations`: $\forall x \in \mathfrak{g}, y.\, \mathsf{P}(\mathfrak{g}, x, y) \wedge \mathsf{P}(\mathfrak{g}, x, u) \Rightarrow \neg \mathsf{P}(\mathfrak{g}, r, y)$. With our path partitioning formulas (3.12), #2 and #3 imply $\mathsf{P}_0(\mathfrak{g}, x, a)$ and $\mathsf{P}_0(\mathfrak{g}, x, b)$, resp. Together, #4 and #5 express that $a \dots y$ is a *newly created path*; hence either $a = u$ or $y = u$ (see Fig. 3.23). If $a = u$, $y \neq u$, then $b \in R$; we draw the contradiction by instantiating (Pre) with x, b for x, y. Otherwise, $a \neq u$, $y = u = b$, then $a \in R$; we draw the contradiction by instantiating (Pre) with x, a for x, y. □

Similarly, we can describe the bad configuration in Fig. 3.24 (β) using the following formula:

$$\begin{aligned}\exists x \in \mathfrak{h}, a, b \in \mathfrak{f}, y \notin \mathfrak{h} \bullet\ & a \neq b \wedge \mathsf{P}_0(\mathfrak{f}, a, y) \wedge \mathsf{P}_0(\mathfrak{f}, a, y) \\ & \wedge \mathsf{P}(\mathfrak{h}, x, a) \wedge \neg \mathsf{P}_0(\mathfrak{h}, x, a) \wedge \mathsf{P}(\mathfrak{h}, x, b)\end{aligned} \tag{3.26}$$

In this configuration, the source of the alternative paths falls into the callee footprint, and their end into the frame; this results in alternative paths $x \dots a \dots y$ and $x \dots b \dots y$. In our example of Fig. 3.22, the new outgoing paths that `ringInsert` creates originate in u; all other outgoing paths also existed *before* the call (due to the last postcondition). In order to avoid the bad configuration (β), `testZopgObligations` requires that *no paths* may originate in the attached node u. Thus, any *new outgoing path* must pass through R

23 It is also possible to get the information about the old reachability relation from a modified version of (3.25) where $\neg \mathsf{P}_0(\mathfrak{h}, a, y)$ is dropped and all other path predicates are evaluated in the pre-state.

before it reaches the callee's footprint. Since `ringInsert` preserves the paths that start in R and end in the frame (due to its last postcondition), and we assumed that the callee's footprint is a ZOPG before and after the call, the last three conjuncts in (3.26) cannot be satisfied. Hence, our specification is strong enough to prevent (β). □

The scenario depicted in Fig. 3.24 (γ) illustrates that *any new path* $a \dots b$ created by the method call, combined with suitable frame paths, may violate the ZOPG invariant:

$$\exists x, b \in \mathfrak{f}, a \in \mathfrak{h}, y \notin \mathfrak{h} \bullet \mathsf{P}_0(\mathfrak{f}, x, y) \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, b, y) \wedge \mathsf{P}(\mathfrak{h}, a, b) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, b). \tag{3.27}$$

In order to avoid the bad configuration (γ), we must ensure that an arbitrary frame node x that reaches the footprint node a does not reach any of the frame nodes (e. g. y) that will be reachable from a after the call. In our example of Fig. 3.22, the precondition of `testZopgObligations` is strong enough to prevent (γ) after the call to `ringInsert`. The nodes x and y in the last conjunct of this precondition can be thought of as those in (3.27) and Fig. 3.24 (γ); the condition rules out the possibility that the effect of the call will connect up such alternative path. □

The most subtle bad configuration is Fig. 3.24 (δ), where both alternative paths $x \dots y$ go via the footprint of the method call. This heap configuration can be expressed via the following formula:

$$\exists x, c, d \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{h} \bullet a \neq b \wedge c \neq d \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, x, b) \wedge \mathsf{P}_0(\mathfrak{f}, c, y) \wedge \mathsf{P}_0(\mathfrak{f}, d, y) \wedge \mathsf{P}(\mathfrak{h}, a, c) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, c) \wedge \mathsf{P}(\mathfrak{h}, b, d). \tag{3.28}$$

This configuration can be realized when a and b are *mutually unreachable in both states* (otherwise, the configuration is covered by (α) and (β)). This configuration cannot occur in the post-state of our example because after the method call u is attached to the ring. □

3.6.2.4 Completeness proof sketch. To derive the four cases in Fig. 3.24, consider a ZOPG subheap $\mathfrak{g}$ comprised of the ZOPG frame $\mathfrak{f}$ and the (relatively convex) ZOPG footprint $\mathfrak{h}$ of a method call. Assume that the method call creates at least one new path s.t. the ZOPG invariant of its footprint is maintained while the ZOPG invariant of the (larger) client's footprint is violated. Consider as well two nodes x and y that are connected (in the state after the call) by *multiple* (at least two) $\mathfrak{g}$-local paths $x \dots y$. We assume that x and y are both in $\mathfrak{g}$; if y is outside $\mathfrak{g}$, we first apply (3.13), providing some node $u \in \mathfrak{g}$ s.t. $\mathsf{P}(\mathfrak{g}, x, u) \wedge \mathsf{E}(\mathfrak{g}, u, y)$; we then continue the argument for $x \dots u$ instead of $x \dots y$.

Multiple paths $x \dots y$ may not be entirely inside just one of the two subheaps $\mathfrak{f}$ or $\mathfrak{h}$ because that would violate our assumption that these are ZOPG subheaps. Therefore, *at least one of these paths must cross the border between* $\mathfrak{f}$ *and* $\mathfrak{h}$. We proceed with a case

analysis based on the distribution of the nodes x and y between (disjoint) subheaps $\mathfrak{h}$ and $\mathfrak{f}$:

- The case $x, y \in \mathfrak{h}$ cannot be realized because a path starting in x may leave $\mathfrak{h}$ just once and *may never come back* to reach y (otherwise our convexity assumption would be violated).
- If $x \in \mathfrak{h}, y \in \mathfrak{f}$, then again, the paths starting in x may leave $\mathfrak{h}$ just once and may never come back due to $\mathfrak{h} \prec \mathfrak{g}$. Since two different paths starting in x may not merge in $\mathfrak{h}$ (otherwise, alternative paths will exist within $\mathfrak{h}$, contradicting our assumption that it is a ZOPG), such paths must reach *two different frame nodes* $a, b \in \mathfrak{f}$, creating alternative paths of the form $x \dots a \dots y$ and $x \dots b \dots y$, as covered by case (β) of Fig. 3.24.
- If $x \in \mathfrak{f}, y \in \mathfrak{h}$, then no path that starts in the frame node x can enter the footprint more than once due to $\mathfrak{h} \prec \mathfrak{g}$. Next, since in this case these paths must end in y, they cannot leave the (relatively convex) footprint $\mathfrak{h}$ at all. Finally, these paths *may not merge until at least one of them enters* $\mathfrak{h}$ (otherwise, alternative paths will exist within $\mathfrak{f}$, contradicting our assumption that it is a ZOPG). This gives us *two different footprint nodes* $a, b \in \mathfrak{h}$, creating alternative paths of the form $x \dots a \dots y$ and $x \dots b \dots y$, as covered by case (α) of Fig. 3.24.
- In the most subtle case of $x, y \in \mathfrak{f}$, each pair of alternative paths of the form $x \dots y$ is s.t. either *just one* or *each of the two* alternative paths *enters and exits* the footprint exactly once, as covered by cases (γ) and (δ) of Fig. 3.24, resp. □

The simplicity of our formulas (3.25), (3.26), (3.27), (3.28) is due to the fact that, in our technique, footprints of method calls must be relatively convex, limiting the number of bad configurations to just four. The bad configurations that we have identified are helpful for deriving weakest preconditions for method calls that operate on ZOPGs, like in our `testZopgObligations` example. In combination with local heap updates (for which (3.24) is the efficient ZOPG preservation criterion), we have explained how our technique is generalized for modular reasoning about ZOPGs.

3.7 LOGICAL ENCODING

In this section, we demonstrate how to encode our technique into separation logic. We designed our technique for automation; thus, we begin with a high-level implementation overview of a possible frontend verifier (Sec. 3.7.1). We factor our encoding into three components. The first component consists of a static preamble, i. e. the axioms that are emitted for any input program, as well as the encoding of `rsnap` and our heap selector functions (Sec. 3.7.2). The second and the third components are the encodings of field update operations (Sec. 3.7.3) and method calls (Sec. 3.7.4), resp.

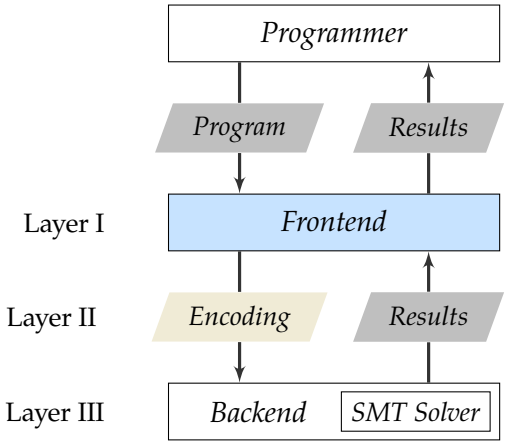


Figure 3.25: Overview of the tool stack.

Programmer submits a program in our source language and its reachability-powered specifications. Three layers define the verifier: *Frontend* encodes reachability specifications into separation logic; *Backend* checks the proof obligations via SMT and reports raw verification results. For each assertion (e. g. invariant or postcondition check), *Backend* reports either success or a potential violation, possibly with a counterexample. *Frontend* decodes these results, mapping locations of failed assertions and counterexamples to those in terms of the source program.

3.7.1 Implementation overview

The implementation consists of three layers (Fig. 3.25). Layer I: A lightweight frontend verifier that is responsible for defunctionalizing program-specific parameters (e. g. rewriting the field set F), as well as lifting verification results in terms of the underlying tools back to the source language. Additionally, the frontend is responsible for keeping track of the ***Zopg*** type annotations from Sec. 3.6. Layer II is the most essential part of the implementation; it consists of a static axiomatization, functions, and macros, as well as the encoding of field update operations (instantiated for concrete values of F) and the translation of all methods from the source program. Layer III: A backend separation-logic verifier that we treat as a black box. Our premier focus in this section is on the encoding technique, corresponding to Layers II–III of the tool stack. The prototype encoding for a variety of benchmarks is publicly available [114].

Our encoding requires a logic with two main features. (1) There should be a means for treating method footprints and frames as first-class node sets, e. g. iterated separating conjunctions (ISC) [90]. (2) The logic should support heap-dependent functions [91]. The former enable generic, set-based memory specifications that are crucial for reasoning about structures that can have unbounded sharing and branching, as well as cycles. The latter serves two purposes in our technique: First, we rely on heap-dependent snapshot functions that provide us with the mathematical edge relation (cf. Sec. 3.3.5); we use this explicit edge relation to axiomatize the heap graph for a fixed state and a fixed footprint. This allows us to disentangle reachability-related axioms from other axioms emitted by the verifier. Second, we rely on (heap-dependent) footprint selectors to relate the footprints of a callee to the footprint of its client (cf. Sec. 3.3.3); this design allows the user to extend the existing selectors and actually *verify* that the new ghost functions are well-formed by implementing them as a heap-dependent fixpoint expression.

We demonstrate our encoding into Viper, a state-of-the art verification language and verification infrastructure that satisfies the above requirements.

3.7.2 Encoding of the static preamble

We present the *static* part of our encoding, consisting of function declarations and axioms that are independent of a particular source program. Assume that the types ***Graph*** and ***Edgeset*** are rewritten as ***Set***[***Ref***] and ***Set***[***Edge***], resp., where ***Ref*** is a type of references to heap objects and ***Edge*** is a tuple of two ***Ref***s. The encoding of the ***Zopg*** domain is presented in [113, App. C].[24]

3.7.2.1 Partial reachability axiomatization. We encode the reachability relation as an uninterpreted Boolean function $\hat{P}$ over an edge set `EG` and two references `from` and `to` corresponding to the origin and the destination of a heap path `from` ... `to`:

```
1  function P̂(EG: Set[Edge], from: Ref, to: Ref): Bool
```

For a fixed value of `EG`, the intended semantics of $\hat{P}$ is the reflexive, transitive closure of the relation $f \equiv$ ***Edge***$(\cdot,\cdot) \in$ `EG`. Since there does not exist a finite, complete set of first-order axioms that defines this semantics, we use a partial axiomatization that includes the most practically essential properties of $\hat{P}$ that we identified based on the scenarios that we aim to support. Note that a complimentary line of work explores the possibility of continuously generating axioms for gradually increasing completeness of supported reachability properties [80].

To justify soundness of our partial reachability axioms, we derive our concrete axioms from the canonical *axiom templates* of the transitive closure simulation technique [53]. We discussed the axiom templates in Sec. 3.2.2.

Heap traversals present the first scenario that we intend to support. For example:

```
var start: Ref := node
while (node != null) { node := node.next }
```

Assuming the loop terminates, we should be able to prove that there exists a path `start` ... `node`. Hence, we add an axiom called `PathStep`:

```
2  axiom PathStep { forall EG: Set[Edge], u: Ref, v: Ref ::
3      P̂^F(EG, u, v) <==> (u == v || (exists w :: Edge(u, w) in EG && P̂(EG, w, v))) }
```

24 The implementation of our support for 0–1-path graphs is in part based on the contributions of Sivanrupan's Bachelor's thesis [104].

This axiom can be obtained from the axiom template $T_1[f]$:

$$\forall u, v \bullet f_{tc}(u, v) \iff u = v \vee \exists w \bullet f(u, w) \wedge f_{tc}(w, v) \qquad T_1[f]$$

We instantiate ***Edge***$(\cdot,\cdot) \in$ `EG` for f and $\hat{P}$(`EG`$,\cdot,\cdot)$ for f_{tc} and get `PathStep`. □

Another important scenario arises due to *transitivity* of the reachability relation. For example, if we establish the existence of two consecutive heap paths, $u \ldots w$ and $w \ldots v$, then we should be able to prove that the path $u \ldots v$ exists as well (cf. Fig. 3.13). To support this case, we add an axiom called `PathJump`:

```
4 axiom PathJump { forall EG: Set[Edge], u: Ref, w: Ref v: Ref ::
5     P̂(EG, u, w) && P̂(EG, w, v) ==> P̂(EG, u, v) }
```

This axiom follows from the template $T_1[f]$ and the *induction principle* $IND[Z, P, f]$ below. The fact that the transitivity property of reachability is a *consequence* of the templates has been claimed by Lev-Ami et al. [53], but the original paper does not seem to present a concrete instantiation that proves the claim, which we will do next. The induction principle for the relation f and its reflexive, transitive closure f_{tc} is given as:

$$\begin{aligned}(\forall w \bullet Z(w) \Rightarrow P(w)) \wedge (\forall w, v \bullet P(w) \wedge f(w, v) \Rightarrow P(v)) \\ \Longrightarrow \forall v, w \bullet Z(w) \wedge f_{tc}(w, v) \Rightarrow P(v)\end{aligned} \qquad IND[Z, P, f]$$

> The intuitive meaning of the induction principle is that if every …[node satisfying Z *also*] satisfies P, and P is preserved when following f-edges, then every …[node] f_{tc}-reachable from a …[node satisfying Z] satisfies P.[a] Lev-Ami et al. [53]
>
> a Ellipsis and brackets indicate omissions and paraphrasing, resp. of the original text.

Let u be the inductive parameter. Consider the following instantiations for the three parameter relations in $IND[Z, P, f]$: $Z(v) \equiv P(v) \equiv f_{tc}(u, v) \equiv \hat{P}$(`EG`$, u, v)$ and $f \equiv$ ***Edge***$(\cdot,\cdot) \in$ `EG`. The first conjunct of the LHS of $\Longrightarrow$ is trivially satisfied. Consider the seconds conjunct: $\forall u, w, v \bullet f_{tc}(u, w) \wedge f(w, v) \Rightarrow f_{tc}(u, v)$. This part of the premise directly follows from the $\Longleftarrow$ direction of the axiom template ($T_2[f]$):

$$\forall u, v \bullet f_{tc}(u, v) \iff u = v \vee \exists w \bullet f_{tc}(u, w) \wedge f(w, v) \qquad T_2[f]$$

This formula is symmetric to $T_1[f]$, with the difference that here the single-edge step is taken at the very end of the path rather than the beginning. $T_2[f]$ follows from $T_1[f]$ and $IND[Z, P, f]$ [53]. Hence, we obtain the RHS:

$$\forall v, w \bullet f_{tc}(u, w) \wedge f_{tc}(w, v) \Rightarrow f_{tc}(u, v)$$

Since u is arbitrary, the formula above matches our proof objective. □

3.7.2.2 Automatically instantiating quantifiers. An important aspect of the implementation of our partial reachability axioms is controlling the instantiations of these axioms. In our prototype implementation, we annotate the quantifiers with *syntactic triggers* that specify the class of terms for which the SMT solver is permitted to produce an instantiation. Triggers are used together with a quantifier instantiation mechanism called *E-matching* [33, 47, 54, 94]. For example, to instantiate `PathJump`, we require two terms from the LHS of its body, which is specified via the trigger: {$\hat{P}$`(EG,u,w)`,$\hat{P}$`(EG,w,v)`}. The trigger (written in curly brackets) must mention (at least once) all quantified variables of its quantifier as arguments of some uninterpreted functions, such as $\hat{P}$. Specifying this trigger (and not, for example, {$\hat{P}$`(EG,u,v)`,$\hat{P}$`(EG,w,v)`}) directs the solver's proof search: Intuitively, whenever there exist two smaller paths, `u`…`w` and `w`…`v`, we instantiate `PathJump` to assemble a larger path, `u`…`v`.

Formalizing the principles behind writing good triggers for quantified axiomatizations is an open problem. However, the general idea (and the main guiding principle used in our encoding) is to enable the solver to traverse the space of possible quantifier instantiations efficiently yet reaching useful proof goals. The main issue is to prevent the proof search from diverging in presence of quantifiers, e. g. due to the possibility of infinite chains of instantiations called *matching loops* [108]. Even though we do not prove that the solver can never diverge while processing our axioms, the experimental results presented in Sec. 3.8 demonstrate that our axiomatization is useful for verifying real-world programs with reachability-powered specifications. Note that the triggers cannot affect soundness of an axiom.

3.7.2.3 The snapshot function. The snapshot function was introduced in Sec. 3.3.5. The purpose of this heap-dependent function is to map a footprint (i. e. set of node references) in a fixed program state to an edge set, providing the first argument for $\hat{P}$:

```
6 function rsnap^F(g: Set[Ref]): Set[Edge]
7 requires ACCESS_NODES^F(g)
8 ensures forall x, y :: x in g && y != null && hedge^F(x,y) <==> Edge(x,y) in result
```

Here `hedge`^{F} is the *heap edge relation* defined by the frontend verifier, for $F = \{f_1, \ldots, f_N\}$, as **`define`** `hedge`^{F}`(x, y) (x.`f_1`== y ||...|| x.`f_N`== y)`. The `rsnap` function plays a special role in our encoding as it is *the only connection* between (the constraints set by our static axiomatization of) the mathematical function $\hat{P}$ and (the facts known about) the actual program states via the field values of the nodes in `g`. Note that `rsnap` does not have a body as it is not *implemented*; hence, this function is treated by the verifier as part of the axiomatic encoding. As an abbreviation for $\hat{P}$`(rsnap`^{F}`(g),x,y)`, we keep the macro

notation $\texttt{P}^F(\texttt{g},\texttt{x},\texttt{y})$ used throughout this chapter; the frontend (Fig. 3.25) desugars all macros, in particular, rewriting the macro variable F, as explained above.

The precondition of `rsnap` requires access to the (***Ref***-type) fields from the set F of objects in `g`. This footprint is encoded via the macro $\texttt{ACCESS\_NODES}^F(\texttt{g})$:

```
9  define ACCESS_NODES^F(g, p=read) (
10   !(null in g) && ✱_{f∈F} (forall n:Ref :: n in g ==> acc(n.f, p)))
```

The big star is rewritten as a chain of separating conjunctions for all fields $f \in F$. If the second macro argument (i. e. the permission amount) is skipped, it is assumed to have the default value of **read**. We assume that the above macro definition is also used for specifying the footprints of all the *methods* with comprehensive specifications. Hence, the footprint of `snap` matches the current method's footprint, except it requires only read permissions to each node's field, while methods may have full, write access.

3.7.2.4 Footprint selector functions. Footprint selector functions were introduced in Sec. 3.3.3. Their purpose is to provide new footprints for method calls, based on the current method's footprint. Intuitively, we want to be able to define these new footprints by selecting subsets of all the nodes accessible in a given program state. While reachability provides a natural way of selecting such subsets, e. g. by specifying all the nodes reachable from a given root node (like in `sub` from Sec. 3.3.3), generally, any predicate $\pi(n)$ over a node n can be used as a footprint selector, e. g. $\texttt{g}_1 = \{n \mid n \in \texttt{g} \wedge \pi(n)\}$, where $\texttt{g}_1$ and `g` are the footprints of the callee and the client, resp. Since our host logic may not natively support set comprehensions, we provide the following function:

```
11 function sub^π(g: Set[Ref]): Set[Ref]
12 requires ACCESS_NODES^F(g)
13 ensures forall n :: n in result <==> n in g && π(n)
```

The symbol π is syntactically rewritten by the frontend (Fig. 3.25). Note that $\texttt{sub}^\pi$ (and π) are heap-dependent [91] functions as the constraints set upon new footprints may depend on the current heap configuration, e. g. via our local reachability predicates.

Our technique does not require for footprint selectors to specify node sets completely; the only required constraint is that the new footprint must be a subset of the current one, as per the specification of the most-generic footprint selector, `some` (cf. Sec. 3.3.3):

```
14 function some(g: Set[Ref]): Set[Ref]
15 requires ACCESS_NODES^F(g)
16 ensures result subset g
```

Using the functions $\texttt{sub}^{\pi}$ and `some`, one could extend the set of supported footprint selectors, as discussed in Sec. 3.3.3. We conclude this part by demonstrating the encoding our footprint selector $\texttt{sub}^{F}$ previously shown in Fig. 3.13 for the special case $F = \{\texttt{left}, \texttt{right}\}$:

```
17 define CLOSED_g^F(h) // (3.4)
18 forall x, y :: x in h && Edge(x,y) in rsnap^F(h) ==> !(y in g setminus h)
19 function sub^F(g: Set[Ref], h: Set[Ref], root: Ref): Set[Ref]
20 requires ACCESS_NODES^F(g)
21 requires root in h && h subset g
22 ensures CLOSED_h^F(result)
23 ensures forall n :: n in result <==> n in h && P̂(rsnap^F(g),root,n)
```

As before, the parameter F is rewritten by the frontend (Fig. 3.25). Note that `Edge(x,y) in` $\texttt{rsnap}^{F}$`(result)` corresponds to the condition that there exists a heap edge (along one of the fields from F) connecting `x` to `y` in the current configuration of the heap fragment specified by **`result`**; the solver can prove that, for a fixed F, heap edges exist in a larger heap fragment (e. g. $\mathfrak{g}$) *iff* they exist in any of its nested heap fragments (e. g. **`result`**).

3.7.3 Encoding of field updates

We are now ready to present the encoding of field updates in our technique. Our goal is to rewrite operations of the form `from.`f `:= to` in a way that will introduce appropriate reachability update formulas to the verifier (cf. Sec. 3.4). We denote the footprint of the method enclosing this field update as $\mathfrak{g}$; we assume that the method's implementation and specification mention only reference fields from the set F; in particular, $f \in F$.

General field updates may result in a deletion of an old heap edge followed by the insertion of a new heap edge. Hence, we encode a field update as a sequential composition of two (conceptually simpler) operations, where the first operation (possibly) deletes a single heap edge and the second one (possibly) adds a new heap edge. The possible models of the resulting heap transition are presented in Fig. 3.26.

3.7.3.1 Field updates in DAGs and ZOPGs. Precise, first-order reachability update formulas for the singleton edge removal and the singleton edge insertion are not known for general graphs (see discussion in Sec. 3.2.1). However, such update formulas are known for two broad classes of graphs, namely acyclic graphs and 0–1-path graphs (introduced in Sec. 3.6). To ensure that the update formulas are applied soundly, all operations, including field updates, must maintain the corresponding structural invariants (i. e. either acyclicity or the absence of alternative heap paths modulo cycles) for the current method's footprint.

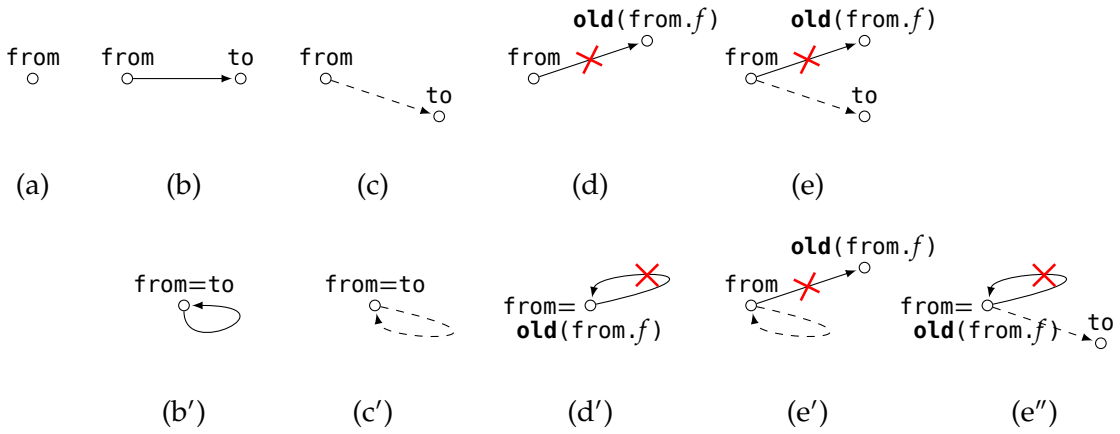


Figure 3.26: Enumeration of heap transitions caused by a field update operation.

The ten possible heap transitions caused by a field assignment operation `from.`f`:=to`. As before, newly added heap edges are dashed, while deleted heap edges are crossed out; **`old`**`(from.`f`)` denotes the value of the f field of `from` in the pre-state of the assignment. The models *without* aliasing are shown on the top row; the models with aliasing are on the bottom row. Consider the top row. The first two models represent the trivial transitions in which the new heap configuration is identical to the old one; the following conditions define their corresponding models: (a) `from.`f`=to=`**`null`**; (b) `from.`f`=to≠`**`null`**. The remaining transitions are non-trivial (`from.`f $\neq$ `to`); the following conditions define their corresponding models: (c) `from.`f $=$ **`null`** $\wedge$ `to` $\neq$ **`null`** (d) `from.`f $\neq$ **`null`** $\wedge$ `to` $=$ **`null`**; (e) `from.`f $\neq$ **`null`** $\wedge$ `to` $\neq$ **`null`**.

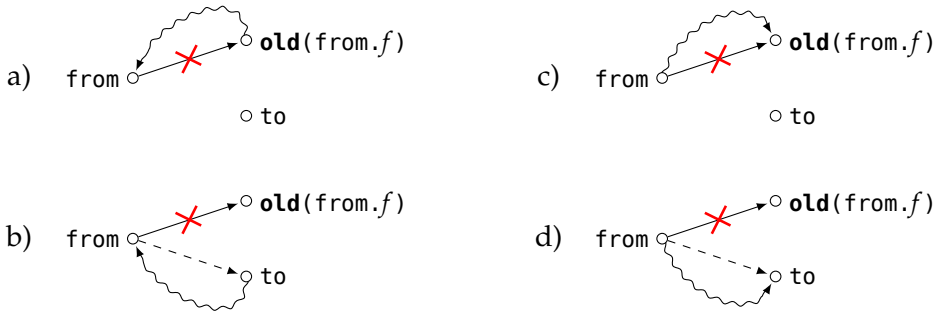


Figure 3.27: Heap configurations that violate graph invariants required for field updates.

The bad heap configurations in presence of a field update operation. The left column corresponds to the DAG case and the right column corresponds to the ZOPG case. The rows show violation of the respective invariants in the pre-state and the post-state, resp. (a), (b) Violation of the acyclicity invariant can be most-efficiently checked in the *intermediate* state in which the node `from` must be unreachable from both `to` and **`old`**`(from.`f`)`. (b), (c) Again, violation of the ZOPG invariant can be most-efficiently checked in the intermediate state, except the condition is that neither `to` nor **`old`**`(from.`f`)` can be reached from the node `from` in this state.

The choice to represent general field updates via a pair of operations enables an efficient way to check that these operations maintain their respective structural invariants. In general, we need to check that, in the case of DAG, adding a new heap edge $(\texttt{from}, \texttt{to})$ to $\mathfrak{g}$ does not complete a cycle of heap references:

$$\forall x \in \mathfrak{g}, y \bullet x \neq y \wedge \mathsf{P}(\mathfrak{g}, x, \texttt{from}) \wedge \mathsf{P}(\mathfrak{g}, \texttt{to}, y) \implies \neg \mathsf{P}(\mathfrak{g}, y, x) \tag{3.29}$$

In the case of ZOPG, we need to check that adding such a heap edge does not create an alternative path (modulo cycles):

$$\forall x \in \mathfrak{g}, y \bullet x \neq y \wedge \mathsf{P}(\mathfrak{g}, x, \texttt{from}) \wedge \mathsf{P}(\mathfrak{g}, \texttt{to}, y) \implies \neg \mathsf{P}(\mathfrak{g}, x, y) \tag{3.30}$$

In practice, checking quantified assertions, such as the ones above, is computationally expensive in SMT because each universally quantified formula may cause a large number of new quantifier instantiations. It is possible to eliminate the quantifiers by pre-instantiating them with fresh variables, but this approach is still inefficient as the underlying SMT solver would have to verify the resulting conditions for all possible (hypothetical) aliasing combinations, the number of which grows exponentially with the number of references. Fortunately, we can use a much simpler (quantifier-free) assertion to check these structural invariants. This can be done by rephrasing the assertions in terms of the *intermediate* state, i. e., after deleting the old edge and before inserting the new edge (Fig. 3.27). The resulting encoding for field updates with efficient structural invariant checks is shown in Fig. 3.28.

It is interesting to observe that, while inserting the *new* edge $(\texttt{from}, \texttt{to})$ can violate the acyclicity invariant (or the ZOPG invariant), removing the *old* edge $(\texttt{from}, \texttt{to}_o)$ cannot result in a more complex structure. However, we still use *two* assertions to check each of these invariants in Fig. 3.28. Note that the first assertion indirectly checks that the old edge did not belong to an SCC (or did not support an alternative path) *in the pre-state* of our field update; this condition already follows from our assumption that, in the pre-state, the structure was a DAG (ZOPG). Indeed, if we ensure that all operations in the current method maintain the required structural invariant, then the same invariants must hold in the pre-state of each field update. The reason why we need both checks is more subtle, as we explain next.

3.7.3.2 Relating reachability via update formulas. We focus on the problem of enabling precise conversion of reachability information in the program states before and after a field update. To this end, we use update formulas, as introduced in Sec. 3.4. Given a graph transforming operation, a reachability update formula for this operation represents reachability information about the new graph in terms of reachability information about the old graph, i. e.:

$$\forall x, y \bullet \mathsf{R}(x, y) \iff Q[\mathsf{R}_0, x, y] \tag{3.31}$$

```
24 define update_f^F(g: Graph, from: Node, to: Node) {
25   if (to != from.f) {    /* pre-state */
26     var to_o := from.f
27     if (from.f != null) {
28       unlink_f^F(g, from)
29       match [g] {
30       case DAG:  assert to_o != from ==> !P^F(g,to_o,from)
31       case ZOPG: assert to_o != from ==> !P^F(g,from,to_o) }
32     }          /* intermediate state */
33     if (to != null) {
34       match [g] {
35       case DAG:  assert to != from ==> !P^F(g,to,from)
36       case ZOPG: assert to != from ==> !P^F(g,from,to) }
37       link_f^F(g, from, to)
38 }}}                    /* post-state */

39 method unlink_f^F(g:Graph, node:Node)
40 requires MEMORY_SPECS_f^F(g,node) &&
41          node.f != null
42 ensures MEMORY_SPECS_f^F(g,node) &&
43         node.f == null &&
44         ( ⋀_{e∈F, e≠f} old(node.e) ≠ old(node.f) )
45         ? UNLINK_[g]^F(g,node,old(node.f))
46         : rsnap^F(g) == old(rsnap^F(g))

47 method link_f^F(g:Graph,from:Node,to:Node)
48 requires MEMORY_SPECS_f^F(g,from) &&
49           from.f == null && to != null
50 ensures MEMORY_SPECS_f^F(g,from) &&
51         from.f == to &&
52         ( ⋀_{e∈F, e≠f} old(from.e) ≠ to )
53         ? LINK_[g]^F(g,from,to)
54         : rsnap^F(g) == old(rsnap^F(g))

55 define MEMORY_SPECS_f^F(g,node) (
56   !(null in g) && node in g && acc(node.f, write)
57   && (forall n :: n in g && n != node ==> acc(n.f, read))
58   && ✱_{e∈F, e≠f} (forall n :: n in g ==> acc(n.e, read)))
```

Figure 3.28: Reachability-aware encoding of field updates into separation logic.

The macro $\texttt{update}_f^F(\texttt{g},\texttt{from},\texttt{to})$ encodes a field update operation $\texttt{from}.f\texttt{:=to}$ of a method with footprint g over the set of reference fields F. $\boxed{\texttt{g}}$ denotes the type annotation of g. The first **if** handles the trivial case, and the nested **if**s handle the remaining special cases; see the enumeration of possible models in Fig. 3.26. $\texttt{unlink}_f^F$ and $\texttt{link}_f^F$ encode the deletion and the creation of a single heap edge, resp., providing (via the postconditions) the update formulas for our relations E and P. $\texttt{to}_o$ is the value of $\texttt{from}.f$ in the pre-state. The assertions check that an update will not result in a bad heap configuration (cf. Fig. 3.27). Lines 23,28: Since the structure is assumed to be acyclic in the terminal (pre- and post-) states, the DAG invariant of our field update is checked more efficiently in the *intermediate* state; checking this invariant in the terminal states would require asserting the quantified formula (3.29), which we spare. Lines 24,29: Similarly, we check the ***Zopg*** invariant (avoiding alternative paths) in the *intermediate* state, sparring the quantified formula (3.30); this is possible due to our assumption that the structure is already a ZOPG in the terminal states. We illustrate MEMORY_SPECS(g, node) in Fig. 3.29.

where R_0 and R are the reachability relations of the old and the new graphs, resp. and Q is some first-order Boolean formula over R_0 with two free variables, namely x and y. This template shows, e. g., that an update formula cannot define R recursively. In the following, we first demonstrate the insufficiency of standard update formulas for preserving all reachability information, and then explain that this limitation can be overcome by adding additional complementary update formulas of the shape:

$$\forall x, y \bullet \mathsf{R}_0(x, y) \iff Q^{\dagger}[\mathsf{R}, x, y] \tag{3.32}$$

where $Q^{\dagger}$ is *the complement of* Q, i. e. some first-order Boolean formula over R with two free variables, namely x and y.

DIRECTED UPDATE FORMULAS. Update formulas in the form of (3.31) generally enable only one direction for converting reachability information: While we can express any element of the $\mathsf{R}(\cdot,\cdot)$ relation — given full information about $\mathsf{R}_0(\cdot,\cdot)$ — we might not be able to use such an update formula to convert reachability information in the opposite direction. Conceptually, update formulas provide the best possible conversion in applications in which the next operation is not known upfront; e. g. a stream of database update queries. However, converting reachability information about the new heap *into* the reachability information about the old heap is a perfectly natural reasoning step when all operations of the current method are known upfront, e. g. in program verification and static analysis, in particular, for computing weakest preconditions.

EXAMPLE. Consider a Hoare triple that demonstrates loss of reachability information in presence of a field update operation that removes a single heap edge `(u, v)`:

$$\left\{ \begin{array}{l} \mathtt{u.next} = \mathtt{v} \wedge \\ \quad \mathsf{R}_0(\mathtt{u}, \mathtt{n}) \end{array} \right\} \ \mathtt{u.next := null} \ \left\{ \begin{array}{l} \mathtt{u.next} = \mathtt{null} \wedge \\ \mathsf{R}(\mathtt{v}, \mathtt{n}) \wedge \neg \mathsf{R}(\mathtt{u}, \mathtt{n}) \end{array} \right\}$$

Here, `n` be some node reachable from `v` in the new state.

Our goal is to check the validity of the *precondition*. For simplicity, we assume that the entire graph is *acyclic* and that `next` is the only reference field. In this scenario, the precise update formula is give by:

$$\forall x, y \bullet \mathsf{R}(x, y) \iff \mathsf{R}_0(x, y) \wedge (\neg \mathsf{R}_0(x, \mathtt{u}) \vee \neg \mathsf{R}_0(\mathtt{v}, y))$$

While this formula connects the reachability information in the two states, it does not provide the *weakest* precondition for the above operation and postcondition; e. g. one cannot use it to justify the above precondition.

In particular, instantiating the update formula with `u` and `n` for x and y, resp. yields $\mathsf{R}(\mathtt{u}, \mathtt{n}) \iff \mathsf{R}_0(\mathtt{u}, \mathtt{n}) \wedge (\neg \mathsf{R}_0(\mathtt{u}, \mathtt{u}) \vee \neg \mathsf{R}_0(\mathtt{v}, \mathtt{n}))$. Simplifying this formula using reflexivity of R and applying the knowledge that $\neg \mathsf{R}(\mathtt{u}, \mathtt{n})$ holds (from the postcondition), we

learn: $\neg \mathsf{R}_0(\mathsf{u},\mathsf{n}) \vee \mathsf{R}_0(\mathsf{v},\mathsf{n})$. This formula *does not follow* from the precondition of the above Hoare tripe. Hence, the (precise) reachability update formula alone is insufficient, showing that reachability information can be lost while reasoning backwards. □

The loss of information via update formulas can be sometimes mitigated by using the information about the *edge relation*. For example, instantiating the update formula above with v and n for x and y implies $\mathsf{R}_0(\mathsf{v},\mathsf{n})$ which can be used in combination with the precondition $\mathsf{u.next} = \mathsf{v}$ (implying $\mathsf{R}_0(\mathsf{u},\mathsf{v})$) to validate $\mathsf{R}_0(\mathsf{r},\mathsf{v})$. However, conversion of the information about the edge relation into reachability information generally requires transitive closure and cannot be fully automated; therefore, we cannot rely on the edge relation to recover the missing bits of the reachability relation.

SOLUTION. To overcome the loss of reachability information in presence of field update operations, we emit *two* update formulas for each edge deletion or insertion. Consider again the Hoare triple above; in this scenario, the operation `u.next := null` reduces the size of the edge set by one edge, namely, $(\mathsf{u.next},\mathsf{v})$. Let $\Delta = \{(\mathsf{u.next},\mathsf{v})\}$; we abbreviate the corresponding transition of the edge relation as $\mathsf{E}_0 \to \mathsf{E}_0 \setminus \Delta$, where $\mathsf{E}_0 \setminus \Delta = \mathsf{E}$. For this transition, the precise update formula is given in the postcondition of our Hoare triple.[25] Intuitively, this update formula provides canonical means of expressing R via elements of R_0. Next, we consider the *reverse transition*: $\mathsf{E} \to \mathsf{E}_0$, where $\mathsf{E}_0 = \mathsf{E} \uplus \Delta$. Because this reverse transition is an incremental update of the edge relation, the precise update formula is given by:

$$\forall x, y \bullet \mathsf{R}_0(x,y) \iff \mathsf{R}(x,y) \vee (\mathsf{R}(x,\mathsf{u}) \wedge \mathsf{R}(\mathsf{v},y))$$

Hence, we obtain the canonical means of expressing R_0 via elements of R.

Returning to the Hoare triple above, we can now simply instantiate our *complementary update formula* with u and n for x and y, resp. to learn $\mathsf{R}_0(\mathsf{u},\mathsf{n}) \iff \mathsf{R}(\mathsf{u},\mathsf{n}) \vee (\mathsf{R}(\mathsf{u},\mathsf{u}) \wedge \mathsf{R}(\mathsf{v},\mathsf{n}))$. Due to the postcondition of the Hoare triple, we have $\neg \mathsf{R}(\mathsf{u},\mathsf{n}) \wedge \mathsf{R}(\mathsf{v},\mathsf{n})$; due to reflexivity, $\mathsf{R}(\mathsf{u},\mathsf{u})$ holds. We learn $\mathsf{R}_0(\mathsf{u},\mathsf{n})$, validating the Hoare triple. □

In the example above, we demonstrate how a combination of a standard (3.31) and a complementary (3.32) update formulas can be used to enable precise, bidirectional conversion of reachability information across an edge removal operation. Naturally, an edge insertion $\mathsf{E}_0 \to \mathsf{E}_0 \uplus \Delta$ and its standard update formula induce a complementary update formula as well, for which the reverse transition $\mathsf{E} \to \mathsf{E} \setminus \Delta$ is a *decremental* update of the edge relation. Therefore, compared to an edge deleting, the update formula and its complement for an edge insertion are simply *swapped* and equal to $Q^\dagger$ and Q, resp.

Since we encode general field updates via a sequence of two operations (Fig. 3.28), namely, an edge deletion and an edge insertion, we combine the standard and the com-

25 Recall that we present our argument for the special case of acyclic list segments.

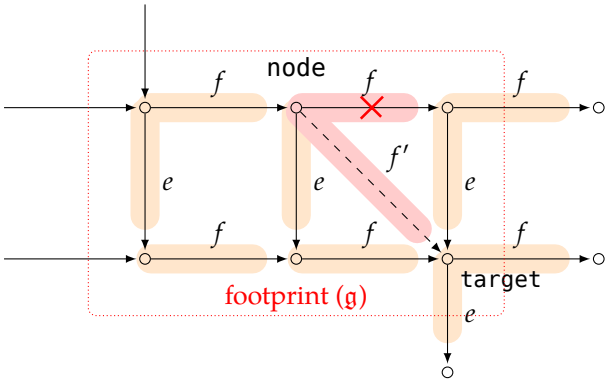


Figure 3.29: Example field update and the required access permissions.

The red frame represents the current footprint, $\mathtt{g}$, containing six nodes with fields f and e (null-valued fields not depicted). The field update `node.`f `:= target` rotates the corresponding edge $f \to f'$. *Write permissions* are highlighted in red. *Read permissions* are highlighted in orange.

plement update formulas in the postconditions of both of these operations, enabling precise, bidirectional conversion of reachability information *across the entire field update*. Note that, in the context of local heap reachability (Sec. 3.3.4), we replace the relations $\mathtt{R}_0$ and $\mathtt{R}$ in (3.31) with the local reachability relations $\mathtt{P}_0^F(\mathtt{g},\cdot,\cdot)$ and $\mathtt{P}^F(\mathtt{g},\cdot,\cdot)$, resp., where $\mathtt{g}$ is the current method's footprint and F is the set of reference fields used by this method. In the case of DAGs, we use a local-reachability adaptation of the standard update formulas for acyclic graphs [17]. In the case of ZOPG, the update formulas are written in terms of the auxiliary relations $\mathtt{DEP}_0$ and $\mathtt{DEP}$ (as explained in Sec. 3.6.1), but our idea of complementary update formulas works exactly the same.

3.7.3.3 Encoding update formulas into separation logic. We return to the discussion of our encoding for update formulas presented in Fig. 3.28. Recall that we encode a general field update as a sequential composition of *two operations*, `unlink` and `link`. Both of these operations have the same memory footprint. *Write access* is needed exclusively for the updated node's reference field (f) in order to permit the required modification; note the first **acc** in `MEMORY_SPECS`. Additionally, all other reference fields, within the current footprint, also require *read access* in order to reason about local reachability along all available fields in F.[26] The last two conjuncts of `MEMORY_SPECS` cover the corresponding access permissions. Fig. 3.29 illustrates the access permissions required for a field update.

The actual reachability update formulas, denoted by $\mathtt{UNLINK}^F_{[\mathtt{g}]}$ and $\mathtt{LINK}^F_{[\mathtt{g}]}$, are assumed (in the postconditions of `unlink` and `link`, resp.) only for operations resulting

26 Recall that evaluating the expression e. g. $\mathtt{P}^F(\mathtt{g},x,y)$ requires at least read-level access permissions to the fields from F of all nodes in $\mathtt{g}$, intuitively, because any of these fields may support the heap path $x \dots y$.

in *non-trivial transitions*, i. e. if the local reachability relations in the pre-state and the post-state are not identical. For example, an assignment operation `a.`f`:= b` may result in a trivial transition even if `a`, `b`, and **`old`**`(a.`f`)` are all non-aliasing, non-null references because the *mathematical edges* `(a, a.`f`)` and `(a, b)` can be supported by reference fields distinct from f (say, e_1 and e_2 in $F = \{f, e_1, e_2\}$), so that the *heap edges* newly deleted and newly inserted by our assignment operation would not affect the reachability relation. For the case of a trivial transition, we merely assume the equality between snapshot functions evaluated in the pre-state and the post-state, from which follows the equality of the local path predicates, namely, $\hat{P}_1^F(\mathtt{g}, x, y) = \hat{P}_2^F(\mathtt{g}, x, y)$, for arbitrary $x \in \mathtt{g}$ and y. Finally, we present the encoding of the update formulas that we assume for the case of a non-trivial transition in Fig. 3.30 for the DAG case and in Fig. 3.31 for the case of ZOPG.

$$
\begin{aligned}
\texttt{LINK}^F_{\texttt{DAG}}(\mathfrak{g},\alpha,\beta) &:\Longleftrightarrow \texttt{G\_PLUS\_DELTA}(\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\texttt{rsnap}^F(\mathfrak{g}),\alpha,\beta)\wedge\\
&\quad\ \texttt{TC\_G\_MINUS\_DELTA}(\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\texttt{rsnap}^F(\mathfrak{g}),\alpha,\beta)\wedge\\
&\quad\ \texttt{TC\_G\_PLUS\_DELTA}(\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\texttt{rsnap}^F(\mathfrak{g}),\alpha,\beta),\\
\texttt{UNLINK}^F_{\texttt{DAG}}(\mathfrak{g},\alpha,\beta) &:\Longleftrightarrow \texttt{G\_MINUS\_DELTA}(\texttt{rsnap}^F(\mathfrak{g}),\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\alpha,\beta)\wedge\\
&\quad\ \texttt{TC\_G\_MINUS\_DELTA}(\texttt{rsnap}^F(\mathfrak{g}),\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\alpha,\beta)\wedge\\
&\quad\ \texttt{TC\_G\_PLUS\_DELTA}(\texttt{rsnap}^F(\mathfrak{g}),\textbf{old}(\texttt{rsnap}^F(\mathfrak{g})),\alpha,\beta),\\
\texttt{G\_PLUS\_DELTA}(g,G,\alpha,\beta) &:\Longleftrightarrow \forall x,y \bullet (x,y)\in G == (x,y)\in g \vee x=\alpha\wedge y=\beta,\\
\texttt{G\_MINUS\_DELTA}(g,G,\alpha,\beta) &:\Longleftrightarrow \forall x,y \bullet (x,y)\in g == (x,y)\in G \wedge (x\neq\alpha\vee y\neq\beta),\\
\texttt{TC\_G\_PLUS\_DELTA}(g,G,\alpha,\beta) &:\Longleftrightarrow \forall x,y \bullet \hat{P}(G,x,y) == \hat{P}(g,x,y)\vee\hat{P}(g,x,\alpha)\wedge\hat{P}(g,\beta,y),\\
\texttt{TC\_G\_MINUS\_DELTA}(g,G,\alpha,\beta) &:\Longleftrightarrow (\forall x,y \bullet \neg\hat{P}(G,x,\alpha)\vee\neg\hat{P}(G,\beta,y) \implies \hat{P}(g,x,y)==\hat{P}(G,x,y))\\
&\quad\ \wedge\forall x,y\bullet \hat{P}(G,x,\alpha)\wedge\hat{P}(G,\beta,y)\implies\\
&\qquad\qquad \hat{P}(g,x,y)==\texttt{case}_i(G,x,\alpha,\beta,y)\vee\\
&\qquad\qquad\qquad \texttt{case}_{ii}(G,x,\alpha,\beta,y)\vee\\
&\qquad\qquad\qquad \texttt{case}_{iii}(G,x,\alpha,\beta,y),\\
\texttt{case}_i(G,x,\alpha,\beta,y) &:\Longleftrightarrow \exists u\bullet u\neq\alpha\wedge u\neq\beta\wedge\hat{P}(G,\alpha,u)\wedge\hat{P}(G,u,\beta),\\
\texttt{case}_{ii}(G,x,\alpha,\beta,y) &:\Longleftrightarrow \exists u\bullet \hat{P}(G,x,u)\wedge\hat{P}(G,u,y)\ \wedge\\
&\quad\ (\neg\hat{P}(G,u,\alpha)\wedge\neg\hat{P}(G,\alpha,u)\vee\neg\hat{P}(G,u,\beta)\wedge\neg\hat{P}(G,\beta,u)),\\
\texttt{case}_{iii}(G,x,\alpha,\beta,y) &:\Longleftrightarrow \exists u,v\bullet (u\neq\alpha\vee v\neq\beta)\wedge\hat{E}(G,u,v)\ \wedge\\
&\quad\ \hat{P}(G,x,u)\wedge\hat{P}(G,u,\alpha)\wedge\hat{P}(G,\beta,v)\wedge\hat{P}(G,v,y).
\end{aligned}
$$

Figure 3.30: Encoding of update formulas for DAGs.

The ultimate formula combines an update formula for an *incremental* and a *decremental* updates of a directed acyclic graph. In all subformulas, $G = g \uplus \{(\alpha, \beta)\}$ holds. The purpose of `TC_G_PLUS_DELTA` is to express the *larger* binary relation $\hat{P}(G, \cdot, \cdot)$ in terms of the *smaller* binary relation $\hat{P}(g, \cdot, \cdot)$. This formula is straightforward: A path $x \ldots y$ exists in the larger relation *iff* it already existed in the smaller relation or it is connected up by the edge (α, β). The purpose of `TC_G_MINUS_DELTA` is to express the *smaller* binary relation $\hat{P}(g, \cdot, \cdot)$ in terms of the *larger* binary relation $\hat{P}(G, \cdot, \cdot)$. This formula has two complementary conjuncts. The first conjunct covers the situation in which the path $x \ldots y$ cannot traverse the edge (α, β); in this case, the smaller relation simply coincides with the larger one. The second conjunct covers the remaining three possibilities in which the path $x \ldots y$ exists in the smaller relation. In the *i*-case, there exists a (distinct) node u that connects up a path of the form $x \ldots \alpha \ldots u \ldots \beta \ldots y$. In the *ii*-case, the node u is *disconnected* from both α and β, ensuring that the detour path $x \ldots u \ldots y$ could not have been dependent on the edge (α, β) and is thus preserved. In the *iii*-case, there exists a detour path of the form $x \ldots (u, v) \ldots y$ which cannot have been affected by removing the edge (α, β) because u *precedes* α or β *succeeds* v in our DAG. The completeness of this formula is due to [17].

$$
\begin{aligned}
\mathtt{LINK}^{F}_{\mathtt{ZOPG}}(\mathfrak{g},\alpha,\beta) \;:\Longleftrightarrow\; & \mathtt{G\_PLUS\_DELTA}(\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\mathtt{rsnap}^{F}(\mathfrak{g}),\alpha,\beta)\wedge\\
& \mathtt{DEP\_G\_MINUS\_DELTA}(\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\mathtt{rsnap}^{F}(\mathfrak{g}),\alpha,\beta)\wedge\\
& \mathtt{DEP\_G\_PLUS\_DELTA}(\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\mathtt{rsnap}^{F}(\mathfrak{g}),\alpha,\beta),\\
\mathtt{UNLINK}^{F}_{\mathtt{ZOPG}}(\mathfrak{g},\alpha,\beta) \;:\Longleftrightarrow\; & \mathtt{G\_MINUS\_DELTA}(\mathtt{rsnap}^{F}(\mathfrak{g}),\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\alpha,\beta)\wedge\\
& \mathtt{DEP\_G\_MINUS\_DELTA}(\mathtt{rsnap}^{F}(\mathfrak{g}),\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\alpha,\beta)\wedge\\
& \mathtt{DEP\_G\_PLUS\_DELTA}(\mathtt{rsnap}^{F}(\mathfrak{g}),\mathbf{old}(\mathtt{rsnap}^{F}(\mathfrak{g})),\alpha,\beta),\\
\mathtt{G\_PLUS\_DELTA}(g,G,\alpha,\beta) \;:\Longleftrightarrow\; & \forall x,y \bullet (x,y)\in G \;==\; (x,y)\in g \;\vee\; x=\alpha\wedge y=\beta,\\
\mathtt{G\_MINUS\_DELTA}(g,G,\alpha,\beta) \;:\Longleftrightarrow\; & \forall x,y \bullet (x,y)\in g \;==\; (x,y)\in G \;\wedge\; (x\neq\alpha\vee y\neq\beta),\\
\mathtt{DEP\_G\_MINUS\_DELTA}(g,G,\alpha,\beta) \;:\Longleftrightarrow\; & \forall x,y,u,v \bullet \widehat{DEP}(g,x,y,u,v) \;==\; \widehat{DEP}(G,x,y,u,v)\wedge\\
& \qquad\qquad \neg\widehat{DEP}(G,x,y,\alpha,\beta),\\
\mathtt{DEP\_G\_PLUS\_DELTA}(g,G,\alpha,\beta) \;:\Longleftrightarrow\; & \alpha\neq\beta\wedge\forall x,y,u,v \bullet u\neq v\wedge x\neq y \implies\\
& \Big(\widehat{DEP}(G,x,y,u,v) \;==\; \widehat{DEP}(g,x,y,u,v)\vee\\
& \qquad\qquad (x=u=\alpha\wedge y=v=\beta)\vee\\
& \qquad\qquad (\exists w,z \;\bullet\; \Psi(g,\,\alpha,\beta,\,x,y,\,u,v,\,w,z))\vee\\
& \qquad\qquad (\exists u',v' \;\bullet\; \Psi(g,\,\alpha,\beta,\,x,y,\,u',v',\,u,v))\vee\\
& u=\alpha\wedge v=\beta\wedge\exists u',v',w,z \;\bullet\; \Psi(g,\,\alpha,\beta,\,x,y,\,u',v',\,w,z)\Big),\\
\Psi(g,\alpha,\beta,x,y,U,V,w,z) \;:\Longleftrightarrow\; & (\forall u'',v'' \;\bullet\; \neg\widehat{DEP}(g,x,y,u'',v''))\wedge\\
& (\widehat{DEP}(g,x,\alpha,u,v)\wedge x\neq\alpha\vee x=u=\alpha\wedge v=\beta)\\
& (\widehat{DEP}(g,\beta,y,w,z)\wedge y\neq\beta\vee\alpha=w\wedge\beta=z=y).
\end{aligned}
$$

Figure 3.31: Encoding of update formulas for ZOPGs.

The ultimate formula combines an update formula for a *decremental* and an *incremental* updates of a 0–1-path graph. In all subformulas, $G = g \uplus \{(\alpha,\ \beta)\}$ holds. The purpose of `DEP_G_MINUS_DELTA` is to express the *smaller* quaternary relation $\widehat{DEP}(g,\ \cdot,\ \cdot,\ \cdot,\ \cdot)$ in terms of the *larger* quaternary relation $\widehat{DEP}(G,\ \cdot,\ \cdot,\ \cdot,\ \cdot)$. This formula is straightforward: A quadruple (x,y,u,v) belongs to the smaller relation *iff* this quadruple belong to the larger relation and the path $x \ldots y$ does not depend on the missing edge $(\alpha,\ \beta)$. The purpose of `DEP_G_PLUS_DELTA` is to express the *larger* quaternary relation $\widehat{DEP}(G,\ \cdot,\ \cdot,\ \cdot,\ \cdot)$ in terms of the *smaller* quaternary relation $\widehat{DEP}(g,\ \cdot,\ \cdot,\ \cdot,\ \cdot)$. There are five possible cases in which a quadruple $(x,\ y,\ u,\ v)$ belongs to the larger relation. In the first case, this quadruple exists even in the smaller relation, so it is definitely preserved in the larger one. The second case is that $(x,\ y)$ is exactly the edge that we added to g to obtain G; since the path $x \ldots y$ could not have existed in g (due to the ZOPG invariant), it trivially depends on the new edge, so our quadruple, indeed, belongs to the larger relation. The last three cases employ an additional formula, Ψ. Intuitively, this formula expresses that adding the edge $(\alpha,\ \beta)$ to a graph with the edge relation g would connect up a new path of the form $x \ldots \alpha \ldots \beta \ldots y$. The last four arguments of Ψ define the two dependency edges of the subpaths $x \ldots \alpha$ and $\beta \ldots y$, resp. Hence, the third and the fourth cases of `DEP_G_PLUS_DELTA` represent situations in which $(u,\ v)$ is a dependency edge of one of the subpaths (hence, it must also be a dependency edge of a larger path in a ZOPG, but only if the other subpath has a dependency edge, too). Finally, the fifth case represents the situation in which $(u,\ v)$ is exactly $(\alpha,\ \beta)$; thus, our quadruple belongs to the larger relation if there exist two dependency edges for *both* our subpaths, $x \ldots \alpha$ and $\beta \ldots y$. The completeness of this formula is due to [17].

3.7.4 Encoding of method calls

We are now ready to present the encoding of method calls in our technique. Our goal now is to rewrite operations of the form `*outs := m(ℌ, *ins)`, where `*ins` and `*outs` denote tuples of input and output parameters, resp. of the method `m`, and ℌ is the footprint of the call. The purpose of our method call rewriting is similar to that of our encoding of field updates: It introduces appropriate formulas to the verifier, enabling modular reasoning about the call and automating entailment proofs in its pre-state and post-state. Fig. 3.3 explains the relationship between reachability framing and entailment proofs. As before, we denote through 𝔊 the enclosing method's footprint and assume that the method's implementation and specification mentions only reference fields from the set F; in particular, $f \in F$. The frontend replaces the original method call with an application $\texttt{call}^F_{\texttt{m}}(\mathfrak{G},\mathfrak{H},\texttt{*ins},\texttt{*outs})$ of the macro `call` defined as follows (the macro parameters `m` and F are expanded by the frontend):

```
59 define call^F_m(𝔊, ℌ, *ins, *outs) {
60   var 𝔣: Set[Ref] := 𝔊 setminus ℌ
61   assert ISCONVEX^F(ℌ, 𝔊, 𝔣)                    // Check via (3.14)
62   EnableFocusOnConvexSubHeap^F(ℌ, 𝔊, 𝔣)          // (3.13), case i of (3.12)
63   EnableFocusOnFrameBefore^F(ℌ, 𝔊, 𝔣)            // (3.13), (3.18)
64   label l1                                       // Pre-state of the call
65   *outs := m(ℌ, *ins)
66   label l2                                       // Post-state of the call
67   assert ISCONVEX^F(ℌ, 𝔊, 𝔣)                    // Check via (3.15)
68   EnableFocusOnConvexSubHeap^F(ℌ, 𝔊, 𝔣)          // (3.13), case i of (3.12)
69   EnableFocusOnFrameAfter^F(l1, l2, 𝔣, ℌ)        // (3.4), (3.5)
70   ApplyConvexTCFraming^F(l1, l2, ℌ, 𝔊, 𝔣)        // (3.10), (3.11), cases ii–iv of (3.12)
71 }
```

3.7.4.1 Overview. Recall from Sec. 3.5 that the main problem in modular reasoning with reachability is *reachability framing*, i. e. the ability to automatically frame reachability properties unchanged by the call and extrapolate the properties of disjoint heap fragments (the frame and the callee's footprint) over their union (the caller's footprint). Our technique solves this problem by employing two groups of formulas: (1) for localization of paths that are entirely inside the relatively convex footprint or its compliment, i. e. the frame (cases i and v in Fig. 3.16) and (2) for splitting and joining heap paths that span the boundary between those two heap fragments (cases ii, iii, iv in Fig. 3.16). Soundly emitting these formulas requires the *checking* relative convexity conditions according to Def. 1.

3.7.4.2 Relative convexity checks. The checks are performed for both states, before and after the call, as heap operations can potentially violate relative convexity (see Fig. 3.17).

As discussed in the end of Sec. 3.5.1, we must use different formulas for checking relative convexity in the state before the call, when we have available only $\mathfrak{g}$-local reachability information (3.14), and in the state after the call, when we have only $\mathfrak{f}$-local and $\mathfrak{h}$-local reachability information (3.15). We encode these assertions as follows:

```
72 define ISCONVEX^F(ℌ, 𝔤, 𝔣) (CONVEX_BEFORE^F(ℌ,𝔤) || CONVEX_AFTER^F(ℌ, 𝔣))
73 define CONVEX_ADD^F(ℌ, 𝔤, 𝔣) (CONVEX_BEFORE^F(ℌ, 𝔤) && CONVEX_AFTER^F(ℌ, 𝔣))

74 define CONVEX_BEFORE^F(ℌ, 𝔤) (ℌ subset 𝔤 && forall x, y, z ::
75   x in ℌ && y in ℌ && !(z in ℌ) && z in 𝔤 ==> !(P^F(𝔤,x,z) && P^F(𝔤,z,y)))

76 define CONVEX_AFTER^F(ℌ, 𝔣) (DISJOINT(ℌ, 𝔣) && forall x, y, z ::
77   x in ℌ && y in ℌ && z in 𝔣 ==> !(P^F(ℌ,x,z) && P^F(𝔣,z,y)))

78 define DISJOINT(ℌ_1, ℌ_2) (forall n :: n in ℌ_1 ==> !(n in ℌ_2))
```

The encoding uses $\texttt{ISCONVEX}^F(\mathfrak{h},\mathfrak{g})$ to *check* that $\mathfrak{h}$ is relatively convex in $\mathfrak{g}$ — the SMT solver automatically identifies which of the two (mathematically equivalent) representations can be satisfied, CONVEX_BEFIRE or CONVEX_AFTER; $\texttt{CONVEX\_ADD}^F(\mathfrak{h},\mathfrak{g})$ can be used in the encoding to *add* the relative convexity constraint over $\mathfrak{h}$ and $\mathfrak{g}$.

3.7.4.3 Path localization. Next, we add the formulas for converting between $\mathfrak{h}$-local and $\mathfrak{g}$-local reachability information (for $\mathfrak{h} \prec \mathfrak{g}$):

```
79 define EnableFocusOnConvexSubHeap^F(ℌ, 𝔤, 𝔣) {
80   assume forall x, y :: x in ℌ && y in ℌ && P^F(𝔤,x,y) ==> P^F(ℌ,x,y)
81   assume forall x, y :: x in ℌ && P^F(ℌ,x,y) ==> P^F(𝔤,x,y)
82   assume CLOSED^F_𝔤(ℌ) ==> forall x, y :: x in ℌ && !(y in ℌ) ==> !P^F(𝔤,x,y)
83 }
```

The first two assumptions follow from to case i of (3.12) and (3.13), enabling localization of $\mathfrak{h}$-local paths that may leave $\mathfrak{h}$ by at most one node. The relative convexity condition allows expressing the existence of a path $\mathtt{x} \ldots \mathtt{y}$, where *both* x and y are in $\mathfrak{h}$, via any of the two predicates: $\mathtt{P}(\mathfrak{h},\mathtt{x},\mathtt{y})$ or $\mathtt{P}(\mathfrak{g},\mathtt{x},\mathtt{y})$. If the condition $\mathtt{y} \in \mathfrak{h}$ is dropped, then the existence of an $\mathfrak{h}$-local path still implies the existence of a $\mathfrak{g}$-local one, but not vice versa because an $\mathfrak{h}$-local path may exit its local heap fragment only by a single node; we will treat the case of paths that span the boundaries of heap fragments in the next part of this section.

The third assumption covers the special case in which $\mathfrak{h}$ is a *relatively closed* subheap of $\mathfrak{g}$ (see line 17), and so all heap paths originating in $\mathfrak{h}$ must not cross its boundary. This formula follows from ($NoExit[A,f]$) with $\cdot \in \mathfrak{h}$ for A and $\textbf{\textit{Edge}}(\cdot,\cdot) \in \texttt{rsnap}^F(\mathfrak{g})$ for f.

Note that the second and the third assumptions can be used together for constraining both $\mathfrak{h}$-local and $\mathfrak{g}$-local paths in a configuration in which $\mathfrak{h}$ is relatively closed.

We now encode the complimentary set of formulas for converting between $\mathfrak{f}$-local paths and $\mathfrak{g}$-local paths. These formulas are more complex due to the fact that $\mathfrak{f}$ is not necessarily relatively convex in $\mathfrak{g}$, as we explained in Sec. 3.5.2. In particular, we use two different macro definitions for constraining $\mathfrak{f}$-local paths in the pre- and the post-states of the call:

```
84  define EnableFocusOnFrameBefore^F(h, g, f) {
85    assume forall x,y :: x in f && y in f &&
86      (forall z :: z in h ==> !(P^F(g,x,z) && P^F(g,z,y))
87        ==> (P^F(f,x,y) <==> P^F(g,x,y))
88    assume forall x,y :: x in f && y in f &&
89      (exists z :: z in h && P^F(g,x,z) && P^F(g,z,y))
90        ==> (P^F(f,x,y) <==> exists σ, τ ::
91          σ in f && (exists z: Ref :: z in h ==> P^F(g,σ,z)) &&
92          τ in f && (forall z: Ref :: z in h ==> !P^F(g,τ,z)) &&
93          P^F(g,x,σ) && E(g,σ,τ) && P^F(g,τ,y)) }

94  define EnableFocusOnFrameAfter^F(l1, l2, g, f) {
95    assume old[l1](rsnap^F(f)) == old[l2](rsnap^F(f))
96    assume forall x,y :: x in f && P^F(f,x,y) ==> P_1^F(g,x,y) && P_2^F(g,x,y)
97    assume CLOSED_g^F(f) ==> forall x,y :: x in f && !(y in f) ==> !P_1^F(g,x,y) && !P_2^F(g,x,y)
98  }
```

The two assumptions on lines 85 to 93 enable the SMT solver to convert $\mathfrak{g}$-local reachability information into $\mathfrak{f}$-local information. The first assumption is simpler: if x does not reach $\mathfrak{h}$ *or* y is not reachable from $\mathfrak{h}$ (line 86), then the path x ... y simply coincides in $\mathfrak{f}$ and $\mathfrak{g}$. Conversely, if the stated condition does not hold (line 89), the existence of an $\mathfrak{f}$-local path can be expressed via a *three-way split* of the hypothetical path x ... y into x ... (σ, τ) ... y, where σ and τ are two frame nodes, reaching and not reaching $\mathfrak{h}$, resp., such that there exists an edge connecting the former to the latter. This fact corresponds to (3.18) and is encoded as the second assumption.

It is sufficient to constrain the frame-local reachability information in the pre-state of the call because, in the post-state, the properties of the frame (in particular, the existence or the absence of frame-local paths) are *unchanged*. Hence, the assumption on line 95 simply states that the evaluation of the `rsnap` function is the same in both states, enabling separation-logic framing (3.11). The two assumptions on lines 96 to 97 are similar to those on lines 81 to 82, except with $\mathfrak{f}$ for $\mathfrak{h}$, but they constrain both states at the same time. Intuitively, since all properties of the frame are unchanged, the properties of a bigger heap fragment that they imply must also remain unchanged.

Note that our encoding has some mathematical redundancy; in particular, since the existence of a $\mathfrak{f}$-local path, say, $x \dots y$ always implies the existence of a $\mathfrak{g}$-local path $x \dots y$

(lines 96 to 97), we could avoid using double implications in our assumptions on lines 85 to 93, leaving only the <== direction in the encoding that corresponds to converting $\mathfrak{g}$-local reachability information back to $\mathfrak{f}$-local information. However, the three-way split of our $\mathfrak{f}$-local path is useful *independently* of the mere knowledge that this path exists because it introduces candidate witnesses that can be used for recovering this information in future states. A concrete example of a situation in which this problem arises is given in Fig. 3.19.

3.7.4.4 Splitting and joining the paths. Under the condition $\mathfrak{h} = \mathfrak{g}$ framing is trivial; we do not add any quantified formulas for this case to avoid matching loops [108] in the encoding. Otherwise, we add the three formulas (lines 101 to 103) corresponding to cases ii, iii, and iv of Fig. 3.16, resp.; note that cases i and v do not require splitting or joining paths and were already covered in lines 79 to 98.

```
99  define ApplyConvexTCFraming^F(l1, l2, ℌ, 𝔤, 𝔣) {
100   if (ℌ != 𝔤) {
101     assume OutInPaths^F(l1, l2, ℌ, 𝔤, 𝔣)
102     assume InOutPaths^F(l1, l2, ℌ, 𝔤, 𝔣)
103     assume OutOutPaths^F(l1, l2, ℌ, 𝔤, 𝔣)
104 } }
```

Our formulas for splitting and joining heap paths are *two-state constraints*. For example, it is possible to join an $\mathfrak{f}$-local path $x \dots a$ (where $x \in \mathfrak{f}$ and $a \in \mathfrak{h}$) with an $\mathfrak{h}$-local path $a \dots y$ in the *post-state* of the call (see "ii" in Fig. 3.16); since the former sub-path is identical in both states, the existence of the path $x \dots y$ can be expressed as the two-state formula $\mathsf{P}_1(\mathfrak{f}, x, a) \wedge \mathsf{P}_2(\mathfrak{h}, a, y)$, where the subscripts 1 and 2 indicate the pre-state (`l1`) and the post-state (`l2`) of the call, resp. Since $\mathfrak{f}$-local paths remain unchanged, we always encode them in the pre-state. Accordingly, our encoding adds constraints over the pre-state of the call using only $\mathsf{P}_1(\mathfrak{f}, \cdot, \cdot)$, $\mathsf{P}_1(\mathfrak{h}, \cdot, \cdot)$, $\mathsf{P}_1(\mathfrak{g}, \cdot, \cdot)$ and constraints over the post-state $\mathsf{P}_1(\mathfrak{f}, \cdot, \cdot)$, $\mathsf{P}_2(\mathfrak{h}, \cdot, \cdot)$, $\mathsf{P}_2(\mathfrak{g}, \cdot, \cdot)$. For readability, we make the F parameter implicit in P and use α and β to denote entry and exit points of the footprint $\mathfrak{h}$ — these are freshly declared Skolem functions with their parameters in the subscripts.

```
105 define OutInPaths^F(l1, l2, ℌ, 𝔤, 𝔣) (
106    (forall x,y :: x in 𝔣 && !(y in 𝔣) && P1(𝔤,x,y) ==> // Split in pre-state
107     α¹x,y in ℌ && P1(𝔣,x,α¹x,y) && P1(ℌ,α¹x,y,y))
108 && (forall x,y :: x in 𝔣 && !(y in 𝔣) && P2(𝔤,x,y) ==> // Split in post-state
109     α²x,y in ℌ && P1(𝔣,x,α²x,y) && P2(ℌ,α²x,y,y))
110 && (forall x,a,y :: x in 𝔣 && a in ℌ && !(y in 𝔣) &&  // Join in pre-state
111     P1(𝔣,x,a) && P1(ℌ,a,y) ==> P1(𝔤,x,y))
112 && (forall x,a,y :: x in 𝔣 && a in ℌ && !(y in 𝔣) &&  // Join in post-state
113     P1(𝔣,x,a) && P2(ℌ,a,y) ==> P2(𝔤,x,y)))
```

The formula on lines 106 to 113 corresponds to case "ii" of (3.12). The only difference is that we incorporate the information from (3.13) in order to support the case of paths exiting the entire heap fragment of $\mathfrak{g}$ (note the conditions $\mathtt{y} \notin \mathfrak{h}$ on the LHS of the implications that we use in the encoding instead of $\mathtt{y} \in \mathfrak{f}$ in the original formula). In our encoding, we explicitly Skolemize the existential quantifier and rewrite the double implication via two implications to guarantee full control over the direction of instantiations.

```
114 define InOutPaths^F(l1, l2, h, g, f) (
115    (forall x,y :: x in h && !(y in h) && P1(g,x,y) ==> // Split in pre-state
116     β^3_{x,y} in f && P1(h,x,β^3_{x,y}) && P1(f,β^3_{x,y},y))
117 && (forall x,y :: x in h && !(y in h) && P2(g,x,y) ==> // Split in post-state
118     β^4_{x,y} in f && P2(h,x,β^4_{x,y}) && P1(f,β^4_{x,y},y))
119 && (forall x,b,y :: x in h && b in f && !(y in h) &&  // Join in pre-state
120     P1(h,x,b) && P1(f,b,y) ==> P1(g,x,y))
121 && (forall x,b,y :: x in h && b in f && !(y in h) &&  // Join in post-state
122     P2(h,x,b) && P1(f,b,y) ==> P2(g,x,y)))
```

Similarly, lines 115 to 122 correspond to case "iii" of (3.12) augmented with the information from (3.13). The above comments apply.

```
123 define OutOutPaths^F(l1, l2, h, g, f) (
124    (forall x,y :: x in f && !(y in h) && P1(g,x,y) ==> P1(f,x,y) ||
125     (α^5_{x,y} in h && β^5_{x,y} in f && P1(f,x,α^5_{x,y}) && P1(h,α^5_{x,y},β^5_{x,y}) && P1(f,β^5_{x,y},y)))
126 && (forall x,y :: x in f && !(y in h) && P2(g,x,y) ==> P1(f,x,y) ||
127     (α^6_{x,y} in h && β^6_{x,y} in f && P1(f,x,α^6_{x,y}) && P2(h,α^6_{x,y},β^6_{x,y}) && P1(f,β^6_{x,y},y)))
128 && (forall x,a,b,y :: x in f && a in h && b in f && !(y in h) &&
129     P1(f,x,a) && P1(h,a,b) && P1(f,b,y) ==> P1(g,x,y))
130 && (forall x,a,b,y :: x in f && a in h && b in f && !(y in h) &&
131     P1(f,x,a) && P2(h,a,b) && P1(f,b,y) ==> P2(g,x,y)))
```

Finally, lines 124 to 131 correspond to the information from case "iv" of (3.12) induced by *the second disjunct*; again, we augment the original formula with the information from (3.13) and the above comments apply. While the left-to-right direction of the implication is encoded directly, the *right-to-left* direction provides two implications, one of which was already covered in our encoding on lines 96 to 97; hence, we omit from the encoding the second copy of formula $\forall x \in \mathfrak{f}, y \notin \mathfrak{h}.\, \mathtt{P}_1(\mathfrak{f}, x, y) \Rightarrow \mathtt{P}_1(\mathfrak{g}, x, y) \wedge \mathtt{P}_2(\mathfrak{g}, x, y)$.

3.8 EVALUATION

We have evaluated our technique on a variety of challenging example programs taken from the literature, illustrating our technique for different classes of graphs and data structures (including the running examples of closely-related work).

3.8.1 Experimental setup

We encoded each example by-hand into the Viper verification language [91]: an intermediate verification language designed for expressing heap-based verification problems, and with native support for separation logic reasoning. Although manual, our encoding of each example was performed methodically, simulating the translation that a frontend verification tool could perform. Each example consists of a common set of background definitions and axioms, along with a translation of the code of the example, statement by statement, according to the technique presented in Sec. 3.5 and Sec. 3.6. For instance, a source-level method call is encoded with additional **`assume`** and **`assert`** statements be-

Table 3.3: Experimental results.

We indicate example features via ✓ where ↗ and ↠ denote examples with *greater-than-one outdegree* and *with sharing*, resp.; ≺ means *convex framing*.

Example	Variant	Class	↗	↠	≺	Time [s]	Notes
Merge (Fig. 3.12)	Tree	DAG	✓		✓	16.1	Path-partitioning
	DAG	DAG	✓	✓	✓	14.5	Unbounded cutpoints
	Fail 1	DAG	✓	✓	✓	13.2	Bug in code
	Fail 2	DAG	✓	✓	✓	33.9	Bug in spec.
Left-Child-Right-Sibling	Tree, add sibl.	DAG	✓		✓	10.5	Encodes n-ary tree as binary
	Tree, add child	DAG	✓		✓	15.0	— ″ —
	DAG, add sibl.	DAG	✓	✓	✓	10.1	Unbounded cutpoints
	DAG, add child	DAG	✓	✓	✓	17.1	— ″ —
Harris List	Original	DAG		✓		14.5	From [100]
Acyclic List	Reverse	DAG				7.9	From [53]
	Append	DAG				6.9	— ″ —
Ring-Insert: Impl.	Sorted	ZOPG	✓	✓		87.2	Functional spec.
	Anywhere	ZOPG	✓	✓		10.1	— ″ —
Ring-Insert: Client (Fig. 3.22)	Closed $\{u\}$	ZOPG	✓	✓	✓	11.5	Non-convex frame
	Open $\{u\}$	ZOPG	✓	✓	✓	10.8	ZOPG obligations
	Fail 1	ZOPG	✓	✓	✓	12.4	Failure due to (β)
	Fail 2	ZOPG	✓	✓	✓	10.7	Failure due to (α), (γ)

fore and after the call which enable reachability framing on relatively convex method footprints, as defined in Sec. 3.5.2.

The background definitions common to our examples are organized in separately-included library files, and we make heavy use of Viper's macros to improve the readability of our encoded examples. Our examples are verified with Viper's standard Boogie-based [46] verifier, which uses the Z3 SMT solver [47] for checking verification conditions. We indicate Viper's run time for each example in Tab. 3.3. The experiments were performed on a laptop running macOS, with a 2.8 GHz Intel Core i7 CPU, with Z3 version 4.8.5 - 64 bit. The Viper files used in our experiments are available online [114].

An important practical issue arising in the successful use of SMT-based verification tools is controlling the instantiation of quantifiers; our technique employs a large number of quantified formulas, and we have carefully selected appropriate *triggering patterns* [33, 47, 54, 94] for these, guided by the intended situations in which these formulas are relevant; for the rich reachability properties expressed by our technique, such triggers are essential for performance. Since our source-level specifications can also contain quantified formulas, we require these to be annotated with appropriate triggers (for simple cases, Viper can also infer appropriate choices if omitted).

3.8.2 Experiments

Tab. 3.3 gives an overview of our experiments. The "Merge" example is our first running example of Fig. 3.12, in variants with both tree and DAG structures for the underlying graphs (obtaining the DAG variant simply requires dropping the tree requirements throughout; no other changes are necessary). "Left-Child-Right-Sibling" is a technique for encoding trees with arbitrary multiplicities using only two fields (representing a list of children at each node), as employed in binomial heaps [50], and recently proposed as a verification challenge [101]. We again show a DAG variant (directly obtained by removing tree requirements), and verify adding sibling and child structures. As with the running example, these are non-functional graphs with (in the DAG case) sharing and requiring our convex framing to frame reachability across sub-calls; to our knowledge, they are beyond reach for all existing automated graph-verification techniques.

The "Fail" variants of Merge are buggy, with the bug being (1) negation of the branch condition in the body of `merge` and (2) missing `merge`'s last precondition. We have observed that the failure time does not diverge from the time of a successful verification attempt. This is important in practice if a program's implementation and specification are developed iteratively, with multiple invocations of the verifier.

Lev-Ami et al. verify reachability for linked-list reverse and append methods [53]; the recent Flows framework [100] uses the Harris List as running example. In both cases, we prove the same invariants and reachability specifications, simply encoded in our language. In the latter case, we use two reachability relations based on different edges.

"Ring-Insert" is a series of six 0–1-path graph examples. We wrote two variants of the Ring-Insert method. "Sorted" is an implementation that traverses a sorted ring and inserts a newly allocated node into the right place. We can prove both reachability (the ring remains a ring) and sortedness; our connection to separation-logic reasoning makes layering additional functional specifications of this kind straightforward. "Anywhere" is the version discussed in Sec. 3.6, where the insertion happens at an arbitrary point in the ring. We also verified two types of clients of Ring-Insert. "Closed $\{u\}$" is the example of Fig. 3.22, where the attached node does not have outgoing paths, whereas "Open $\{u\}$" permits the attached node to be both reachable from the frame and have outgoing paths. The latter requires a more subtle precondition to satisfy the 0–1-path preservation criteria. In the final two cases, we show that our technique allows us to automatically identify the type of bad configurations that may violate the 0–1-path invariant in cases where the heap is under-constraint before a method call Sec. 3.6.2.

3.8.3 Results

Our experiments show that reachability properties are amenable to SMT-based verification for a broad class of heap-transforming programs. In particular, we have observed that our technique is well-suited for this task despite heavy usage of quantified formulas. While developing the specifications, we have experienced that our technique helps the programmer to better understand the subtle effects of heap operations on data structure invariants. Even with good tool support, writing consistent preconditions and postconditions requires particular craftsmanship, especially for recursive methods, like `merge`. Additionally, SMT-based verification with quantifiers requires the programmer to annotate the specifications with triggers.

3.9 METATHEORY

In this section, we first sketch an argument that our overall verification technique is *sound* (Sec. 3.9.1) and then expand the foundation of *relatively convex framing* (Sec. 3.9.2).

3.9.1 Soundness

Our technique encodes *separation logic with local reachability relations* (SL^2R^2) into the *host logic*, i. e. a first-order separation logic equipped with iterated separating conjunction and heap-dependent functions, e. g. Viper. We assume that the host logic is sound and its assertions are automatically translated (by our technique) into first-order proof obligations that are subsequently discharged by an automatic prover, e. g. an SMT solver.

Our technique is sound if a successful verification of the encoding of a program and its local reachability specifications into the host logic implies that there exists a proof of correctness of the specified program using SL^2R^2. To establish soundness, the following three conditions must hold. First, the *prerequisites* of the technique are met (as will be defined below). Second, we can establish a (Hoare-style) SL^2R^2 proof outline that relies on a mapping from encoded program states to SL^2R^2 assertions (e. g. `exists_path(rsnap(g),x,y) → P(g,x,y)`). For each encoded statement `s`, we map its pre- and post-states to the corresponding SL^2R^2 assertions *pre*, *post* and check that the Hoare triple {*pre*} `s` {*post*} is indeed provable in SL^2R^2. Third, all lemmas automatically established by the prover coincide with SL^2R^2 lemmas. These three requirements imply *soundness* of our technique, i. e. the encoding of a specified program will verify only if there exists an SL^2R^2 proof that justifies it.

Our technique has the following three soundness prerequisites:

- Each *footprint selector* is well-defined (Sec. 3.3.3.1).
- Each new footprint is *relatively convex* in its client's footprint, both before and after the call (Sec. 3.5.1.3).
- All heap-transforming operations in methods with DAG and 0–1-path footprints preserve their corresponding *footprint invariant* (Sec. 3.6.2).

The first prerequisite is automatically met if the programmer uses (a combination of) one of the standard footprint selector functions, as described in Sec. 3.3.3. The programmer must manually prove that each custom footprint selector function is well-defined. The latter two prerequisites are automatically checked by our technique; in case a check fails, the technique reports an error. A future frontend could parse such errors and report the corresponding readable message to the programmer; in particular, counterexample SMT models can be useful for understanding such failures, as we will discuss in Chap. 4.

3.9.2 Relatively Convex Subgraphs

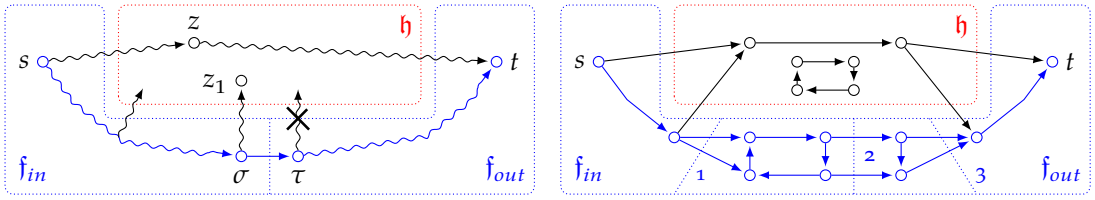
Relative convexity of procedure footprints is the key assumption that enabled efficient and precise reachability framing via simple, first-order formulas (3.12) because in our setting path partitioning is based solely on the origins and the destinations of paths. In this partitioning schema, only the callee footprint is required to be relatively convex while the frame of the call may be non-convex, complicating reachability information localization in the frame. Information about the existence of frame-local paths is a prerequisite of a precise reasoning technique, as we explained in Sec. 3.5.2. Our technique seamlessly solves this problem with a generalized formula (3.18) that provides precise frame-local reachability information with the help of *edge witnesses* (i. e. (σ, τ) edges from Fig. 3.20). However, the solution that we demonstrated so far is not unique. Therefore, we proceed with the observations and propositions that motivated our design.

SETTING. We revisit the configuration of Fig. 3.16, with two procedures, a callee and its client with $\mathfrak{h}$ and $\mathfrak{g}$ as their footprints, resp. Let these footprints be relatively convex ($\mathfrak{h} \prec \mathfrak{g}$, see Def. 1) in some program state (e. g. before or after the call). Let $\mathfrak{f}$ be the frame of the call ($\mathfrak{f} = \mathfrak{g} \setminus \mathfrak{h}$). Due to relative convexity, following a $\mathfrak{g}$-local path that originates in and exits $\mathfrak{h}$, one cannot enter $\mathfrak{h}$ again; hence, all $\mathfrak{g}$-local paths are distributed in five cases based on the distribution of their origins and destinations in $\mathfrak{h}$ and $\mathfrak{f}$.

PROBLEM. To support reachability framing, our goal is to convert $\mathfrak{g}$-local reachability information into a combination of $\mathfrak{h}$-local and $\mathfrak{f}$-local information, and vice versa. We explained in Sec. 3.5 that the simple path partitioning schema powered by relative convexity enables such conversion for all cases except the case of paths that start and end in the frame. In this last case, an additional complication arises: Unlike the callee footprint, the frame of the call is not necessarily relatively convex in the client footprint, and paths that start and end in $\mathfrak{f}$ are not guaranteed to be $\mathfrak{f}$-local, as following such paths one can exit the frame and then enter it again, although, no more than once.

We first summarize the key observations and then formalize our claim in Th. 1. While partitioning the client's footprint into *two* relatively convex partitions (i. e. the footprint and the frame of a call) is not always possible, it is always possible to further partition the (generally, non-convex) frame into two relatively convex *sub-frames*. Moreover, one sub-frame ($\mathfrak{f}_{in}$) can include all entry points (footprint nodes directly pointed to from the frame) whereas the other ($\mathfrak{f}_{out}$) can include all exit points (footprint nodes directly pointing to the frame). The resulting three-way partitioning schema of the form $\mathfrak{g} = \mathfrak{f}_{in} \uplus \mathfrak{h} \uplus \mathfrak{f}_{out}$, where $\mathfrak{f}_{in}, \mathfrak{h}, \mathfrak{f}_{out} \prec \mathfrak{g}$, enables precise conversion of $\mathfrak{g}$-local reachability into $\mathfrak{f}$-local reachability (where $\mathfrak{f} = \mathfrak{f}_{in} \uplus \mathfrak{f}_{out}$ is the frame partition from our standard path partitioning formulas (3.12)):

$$
\begin{aligned}
\forall x, y \in \mathfrak{f}_{in} \bullet \mathsf{P}(\mathfrak{f}, x, y) &\iff \mathsf{P}(\mathfrak{g}, x, y) \\
\forall x, y \in \mathfrak{f}_{out} \bullet \mathsf{P}(\mathfrak{f}, x, y) &\iff \mathsf{P}(\mathfrak{g}, x, y) \\
\forall x \in \mathfrak{f}_{out}, y \in \mathfrak{f}_{in} \bullet \neg \mathsf{P}(\mathfrak{f}, x, y) & \\
\forall x \in \mathfrak{f}_{in}, y \in \mathfrak{f}_{out} \bullet \mathsf{P}(\mathfrak{f}, x, y) &\iff \exists \sigma \in \mathfrak{f}_{in}, \tau \in \mathfrak{f}_{out} \bullet \mathsf{P}(\mathfrak{g}, x, \sigma) \wedge \mathsf{E}(\mathfrak{g}, \sigma, \tau) \wedge \mathsf{P}(\mathfrak{g}, \tau, y)
\end{aligned}
\tag{3.33}
$$



(a) Abstract path diagram. **(b)** Example with alternative partitioning schemes.

Figure 3.32: Relatively convex three-way partitioning.

The partition $\mathfrak{h}$ (surrounded with red) is a relatively convex partition (of some larger heap fragment $\mathfrak{g}$). (a): The compliment partition (surrounded with blue) can be split into two new relatively convex partitions of $\mathfrak{g}$ — $\mathfrak{f}_{in}$ and $\mathfrak{f}_{out}$ — via an s–t-cut, e. g. where the former is the set of all nodes from the compliment *reaching* $\mathfrak{h}$ and the latter is the set of all other nodes from the compliment *reachable from* $\mathfrak{h}$. (b) There can be many possible s–t-cuts that split the compliment into relatively convex partitions (e. g. 1, 2, 3 on the diagram). The three partitions are not necessarily acyclic, although, due to relative convexity, heap cycles cannot span the boundaries of these partitions.

We will now justify our claim that selecting a pair of relatively convex footprints $\mathfrak{h}, \mathfrak{g}$ defines a triplet of relatively convex partitions.

Theorem 1 (Relatively Convex Three-Way Partitioning)**.** *Let $\mathfrak{g}$ be a heap fragment in some program state and $\mathfrak{h}$ be its relatively convex partition ($\mathfrak{h} \prec \mathfrak{g}$, see Def. 1). Then the compliment $\mathfrak{g} \setminus \mathfrak{h}$ can be split into two relatively convex partitions, $\mathfrak{f}_{in}$ and $\mathfrak{f}_{out}$, i. e.:*

$$\forall \mathfrak{h} \prec \mathfrak{g} \bullet \exists \mathfrak{f}_{in}, \mathfrak{f}_{out} \prec \mathfrak{g} \bullet \mathfrak{g} = \mathfrak{f}_{in} \uplus \mathfrak{h} \uplus \mathfrak{f}_{out}$$

Proof. We analyze nodes $n \in \mathfrak{f}_{in} \uplus \mathfrak{f}_{out}$ based on their reachability *to* and *from* $\mathfrak{h}$, namely, the values of the following predicates:

$$\mathsf{P}(\mathfrak{g}, n, \mathfrak{h}) \;:\Longleftrightarrow\; \exists z_1 \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, n, z_1)$$

$$\mathsf{P}(\mathfrak{g}, \mathfrak{h}, n) \;:\Longleftrightarrow\; \exists z_2 \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, z_2, n)$$

First, both $\mathsf{P}(\mathfrak{g}, \mathfrak{h}, n)$ and $\mathsf{P}(\mathfrak{g}, n, \mathfrak{h})$ cannot hold at the same time as that would violate our assumption that $\mathfrak{h}$ is relatively convex in $\mathfrak{g}$. Second, if $\neg\mathsf{P}(\mathfrak{g}, \mathfrak{h}, n)$ and $\neg\mathsf{P}(\mathfrak{g}, n, \mathfrak{h})$, then n can be included in any one of the sub-frames, e. g. in $\mathfrak{f}_{out}$, ensuring their relative convexity (such nodes do not affect the rest of the discussion). Finally, we include all nodes reaching $\mathfrak{h}$ into $\mathfrak{f}_{in}$ and all other nodes into $\mathfrak{f}_{out}$; since in our setting we assume no additional information about $\mathfrak{g}$-local paths, the condition $\mathsf{P}(\mathfrak{g}, n, \mathfrak{h})$ is a precise criterion. Hence, we construct the following relatively convex partitions for $\mathfrak{g} \setminus \mathfrak{h}$:

$$\begin{aligned} \mathfrak{f}_{in} &= \{n \in \mathfrak{g} \setminus \mathfrak{h} \mid \exists a \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, n, a)\} \\ \mathfrak{f}_{out} &= (\mathfrak{g} \setminus \mathfrak{h}) \setminus \mathfrak{f}_{in} \end{aligned}$$

It is easy to check that these partitions are relatively convex in $\mathfrak{g}$. $\mathfrak{f}_{out} \prec \mathfrak{g}$: Paths that originate in this partition reach neither $\mathfrak{h}$ (by definition of $\mathfrak{f}_{out}$) nor $\mathfrak{f}_{in}$ (as that would imply that they transitively reach $\mathfrak{h}$, violating the same definition). $\mathfrak{f}_{in} \prec \mathfrak{g}$: Paths originating in $\mathfrak{f}_{in}$ that reach $\mathfrak{f}_{out}$ cannot reach $\mathfrak{f}_{in}$ again due to our last argument; since all nodes in $\mathfrak{f}_{in}$ reach $\mathfrak{h}$ (by definition), paths originating in $\mathfrak{f}_{in}$ that reach $\mathfrak{h}$ cannot reach $\mathfrak{f}_{in}$ again as, due to our assumption that $\mathfrak{h}$ is relatively convex, there cannot exist paths that originate in and exit $\mathfrak{h}$ that reach $\mathfrak{h}$ again. □

Based on Th. 1, we can derive a *static* frame partitioning schema ($\mathfrak{f} = \mathfrak{f}_{in} \uplus \mathfrak{f}_{out}$), i. e. a partitioning that can be defined once for all nodes in a relatively convex decomposition of $\mathfrak{h} \prec \mathfrak{g}$, as opposed to *dynamic* frame partitioning,[27] such as the kind used in our automatic technique of Sec. 3.5.2. In both schemes, the sub-frames are defined for a fixed footprint pair $\mathfrak{h}, \mathfrak{g}$ (with the condition $\mathfrak{h} \prec \mathfrak{g}$), but the dynamic partitioning schema additionally fixes a pair of frame nodes, s and t, for which the paths $s \dots t$ are distributed among the footprint of the call and the two sub-frames (Fig. 3.32). We choose to split the path into $s \dots (\sigma, \tau) \dots t$ where σ is the last node reaching $\mathfrak{h}$ on the frame-local path from s to t; as we argued in Sec. 3.5.2, this increases the chances that a witness for the edge (σ, τ) will naturally occur, e. g. in the form $(n, n.f)$ for some field f and frame node n, as in our example of Fig. 3.19. However, any *s–t-cut modulo cycles* (i. e. a split of the frame's acyclic part that separates s and t) could be used instead. Note that we do not assume that the heap can be represented by finite graphs; the existence of s–t-cuts for infinite graphs follows from the Menger-Erdős's theorem [48].

3.10 DISCUSSION

In this section, we summarize the strengths (Sec. 3.10.1) and limitations (Sec. 3.10.2) of our technique for compositional reasoning about heap reachability properties,[28] and conclude the chapter (Sec. 3.10.3).

3.10.1 Strengths

MODULARITY. Our reasoning technique for reachability properties is modular. For each method, the programmer specifies reachability only locally, within the method's footprint. Our technique enables precise (first-order) framing of reachability information, i. e. reachability properties of the callee footprint are automatically extended to the (larger) client footprint, as long as these footprints are relative convexity.

27 Our technique does not require specifying the sub-frames.
28 We have presented a detailed comparison of our work vs. the state of the art in Sec. 3.2.7.

GENERALITY. Our technique supports practically important heap structures, significantly generalizing over the structures supported in prior work. In particular, we support arbitrary acyclic structures with a bounded outdegree, e. g. binary decision diagrams and version control histories, but also (potentially cyclic) 0–1-path graphs which generalize linked-lists and trees.

SEPARATION LOGIC. Our technique integrates into first-order separation logic, enabling reasoning about reachability and other properties in a uniform way, verifying concurrent programs, and automating our technique via existing separation logic verifiers.

3.10.2 Limitations

UNBOUNDED QUANTIFICATION. The generality of our axiomatization is due to unbounded first-order quantification. However, instantiating quantifiers with appropriate ground terms is an undecidable problem. This makes our technique inherently incomplete. We mitigated this incompleteness by formulating our axioms in a generic way. This required trading off some efficiency as there may be spurious axiom instantiations.

In our experience, the incompleteness caused by quantifier instantiation mostly concerns entailment proofs, e. g. converting between reachability information defined in terms of concrete heap edges and reachability predicates, and — to a lesser extent — field update formulas. In practice, automating a proof might require adding manual assertions or slightly refining the program's specification to expose useful ground terms to the solver. Conversely, our encoding for reachability framing did not cause any automation problems; this is expected as *splitting* heap paths before a method call introduces explicit cutpoints (via Skolemization) that the solver can then use as existential witnesses for *joining* heap paths after the call.

SYMBOLIC EXECUTION. Our encoding performs poorly with Viper's symbolic execution engine, practically limiting the current version of the technique to Viper's verification condition generator. We discussed the usefulness of supporting two alternative backends in Sec. 1.1.2. A preliminary investigation showed that extremely wide symbolic execution trees occur while symbolically evaluating reachability update formulas. A thorough analysis and remedy are future work.

COMBINING DAGS AND ZOPGS. To support field updates, our technique maintains one of the two structural invariants of the current method's footprint: either acyclicity or uniqueness of paths. As an effect of an operation, acyclic structures must not be changed into 0–1-path graphs, and vice versa. Supported data structure decompositions are thus limited to those in which both the callee and the client share their structural invariant. A generalization is future work.

DECOMPOSITIONS OF CYCLIC STRUCTURES. Our technique requires that a callee's footprint is convex relatively to its client's footprint. This requirement is typically naturally satisfied for most data structure decompositions. However, our technique does not support some two-way decompositions of cyclic structures that could occur in practice. For example, if the client with the footprint $\mathfrak{g}$ operates on three-node ring structure $a \to b \to c \to a$, then it cannot invoke a callee with the footprint $\mathfrak{h} = \{a, b\}$ as this would violate relative convexity (since there exists a path of the form $b \in \mathfrak{h} \quad \ldots \quad c \in (\mathfrak{g} \setminus \mathfrak{h}) \quad \ldots \quad a \in \mathfrak{h}$).

BOUNDED OUTDEGREE. Our technique is designed for heap graphs with bounded outdegree, i. e. nodes may have only a bounded number of reference fields. This assumption allows us to avoid some existential quantifiers in the encoding, simplifying the logical complexity of our technique (which is important for automation). However, this limitation can be mitigated, e. g. by employing the left-child-right-sibling representation [13] that encodes graphs with arbitrary outdegree via graphs with an outdegree of 2.

ISC-BASED SETTING. We assumed a setting in which all method footprints are explicitly specified via nodesets. One could potentially lift this requirement if the nodeset-based footprint representations could be *encoded* or *inferred* rather that specified. Theoretically, this should be possible in a logic that supports *permission introspection*, which is the case e. g. in Viper [91]. However, investigating such a setting is future work.

We will discuss generalizations of our techniques in more detail in Sec. 5.1.

3.10.3 Conclusion

In this chapter, we have developed a compositional technique for reasoning about reachability properties in separation logic. Our solution is based on the novel notions of local reachability and relatively convex heap fragments. Relative convexity enables precise, first-order reachability framing by restricting only the possible decompositions — but not the shapes — of supported linked heap structures.

The compositional techniques of Chap. 2 and Chap. 3 complement each other. The former supports comprehensive specifications that allow the programmer to summarize data structure *values* in an abstract but expressive way. The latter supports local reachability properties that are essential for defining data structure *shapes*.

4 VERIFICATION DEBUGGING

In this chapter, we address the problem of *verification debugging* for heap-transforming programs and their separation-logic specifications. Generally, verification debugging involves techniques and tools that aid the programmer in understanding the *reasons* why certain verification attempts are unsuccessful. In particular, verification failures may be caused by *bugs in the implementation* of a program, e. g. due to off-by-one errors or typos in variable names. *Inadequate specifications* are another source of failure, e. g. a loop invariant that lacks information may be too weak to entail the overall proof goal, while an overly strong invariant cannot be established at loop entry. Note that these failure causes are not necessarily exclusive, e. g. in case of a buggy program with inadequate specification.

DEBUGGING IN PRESENCE OF UNDECIDABILITY. Understanding the scope of a verification failure is especially complicated for programs with rich specifications, e. g. those expressed with set comprehensions (Chap. 2) or local reachability predicates (Chap. 3). Richer specifications express more program properties that are harder to automatically verify as this requires discharging proof obligations that are *undecidable*. Even the best automatic theorem provers used by modern program verifiers (e. g. Z3 [47]) are fundamentally incomplete while reasoning with undecidable logics. Therefore, automatically verifying a program against its rich specifications may result in *spurious verification failures*.[1] It is crucial for the programmer to be able to understand whether a particular failure is spurious as this may determine the best solution strategy.

DEBUGGING WITH COUNTEREXAMPLES. To understand why an *execution* of a program fails, the programmer typically analyzes the test inputs that fail. Conceptually, a test input is a counterexample to the hypothesis that the program is correct. In deductive verification, a failure is not immediately accompanied by a concrete counterexample. Yet, the programmer needs to be able to synthesize such counterexamples, e. g. to decide which parts of the specification are inadequate and how they should be fixed.

SMT MODELS. An important advantage of SMT-automated verification is that the solver provides *counterexample models* that can help understanding the contradiction that caused the verification to fail. Recall that SMT-based verifiers encode the problem of verifying an assertion, say Q, about a program under some assumptions, say P, as the formula

1 In practice, spurious failures may occur even with decidable proof obligations due to the verifier's limited resources (e. g. out-of-time or out-of-memory events).

$F \equiv P \wedge \neg Q$ (called *query*). If F is *unsatisfiable*, then Q is valid under P. Otherwise, Q may be violated, i. e. we have a verification failure; in this case, there exists a model of F that can be interpreted as a counterexample.

To illustrate, let $P \equiv$ (`x.next = y` $\wedge$ `y` $\neq$ **`null`**) and $Q \equiv$ (`x.next` $\neq$ `y.next`); then $F \equiv$ (`x.next = y` $\wedge$ `y` $\neq$ **`null`** $\wedge$ `x.next = y.next`). This formula is satisfiable, e. g. for the following model:

```
null ↦ v0, x ↦ v1, y ↦ v2, next ↦ { v1 ↦ v2, v2 ↦ v2, else ↦ unspecified}
```

Here, `v0`–`v2` are some (non-aliasing) values. For simplicity, we assume that there is only one program state, and `next` is a (total) function of one argument (hence the redundant **`else`** case). It is easy to see that this model encodes the counterexample to F that can be summarizes as `x.next = y = y.next`.

Generally, the assumptions P may include global axioms (e. g. the fact that the length of an empty list equals zero) as well as assumptions along a particular execution path.

APPROXIMATE SMT MODELS. In presence of rich program specifications (that lead to undecidable proof obligations), SMT models are generally *approximate*. A common approach to encoding such specifications is to employ uninterpreted functions with quantified axioms that specify some of their essential properties. Ideally, SMT models should have concrete interpretations for all such functions, but (due to the undecidability) the interpretations are typically *partial*, i. e. the function values are not specified for some inputs, or even *unsound*, i. e. the solver did not instantiate some of the axioms that theoretically contradict the model.

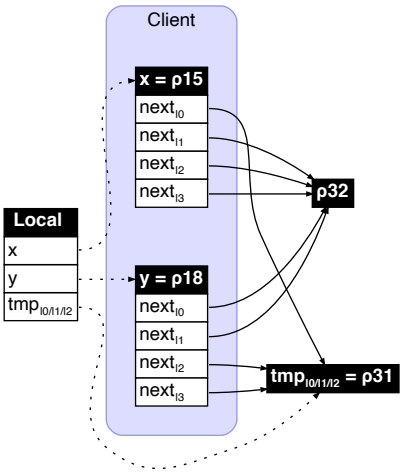
It is sometimes possible to filter out spurious models via a validation procedure, e. g. the Z3 SMT solver [47] supports the `model_validate` option. In a possible validation approach, one could incorporate information from the model into the original query and re-run the solver. Since concrete model values yield ground terms that potentially trigger new quantifier instantiations, the solver may conclude that the new query is *unsatisfiable*, implying that the model certainly does not represent a counterexample to the original proof obligation. However, model validation is generally also undecidable.

There are two possible approaches to verification debugging with approximate models. First, one could filter out the information that is not guaranteed to be sound, like the partial SMT models, or even generate counterexamples to verification failures *independently* of the SMT solver. This approach has the advantage that the programmer gets reliable information about which counterexamples would actually cause a runtime failure. Second, one could use *exactly* the information that the partial SMT model provides to explain why *this particular solver* failed the verification. This approach is especially useful for debugging verification failures caused by the solver's incompleteness.

REFINEMENT. After understanding the cause of a verification failure, the next step for the programmer is to design and implement *the fix*. Fixing a program's verification failure consists of refining (or simply correcting) the program's specification and implemen-

```
method swap(g: Set[Ref], x: Ref, y: Ref)
  requires NODES(g) && x ∈ g ∧ y ∈ g
  ensures NODES(g)
          x.next = old(x.next)  // l3
{
  var tmp := x.next             // l0
  x.next := y.next              // l1
  y.next := tmp                 // l2
}
```

(a)



(b)

Figure 4.1: Motivating example of visual verification debugging.

(a) Method `swap` swaps the `next`-field values of `x` and `y`, two nodes in the footprint `g`. Postcondition marked `//l3` has a typo; it should say `x.next = old(y.next)`. (b) Counterexample diagram that helps understanding the issue. *Local* is the local store and *Client* depicts the footprint.[2] Stateful variables (`tmp`) and fields (`next`) are subscripted with their respective state labels (`l0`–`l3` in comments). Solid black arrows are heap edges and dotted arrows are local (non-null) references.

tation. While we acknowledge this problem, our focus is on aiding the programmer's understanding (by providing helpful counterexamples), rather than actually fixing verification failures.

SETTING. Further, we focus on verification debugging of heap-transforming programs specified in the style of our techniques of Chap. 2 and Chap. 3. Hence, our setting is again based on separation logic with potentially unbounded method footprints that are explicitly specified as sets of non-null references.

MOTIVATING EXAMPLE. To motivate our visual debugging approach, consider the example of Fig. 4.1. The method `swap` has a typo in its specification (Fig. 4.1a). The counterexample diagram (Fig. 4.1b) helps the programmer to understand that the implementation transforms the heap *correctly*, while the issue is in `swap`'s postcondition (`//l3`). To fully understand the problem, the programmer could use the visual diagram in two ways. First, they could check how the failed postcondition reflects on the visual diagram: `x.next` in `l3` refers to ρ_{32}, while `old(x.next)` refers to a different node, ρ_{31}. Second, they could simulate the algorithmic steps: the triplet (`x.next`, `y.next`, `tmp`) is (ρ_{31}, ρ_{32}, ρ_{31}) in `l0`; it then changes to (ρ_{32}, ρ_{32}, ρ_{31}) in `l1`; finally, we get (ρ_{32}, ρ_{31}, ρ_{31}) in `l2`, i. e. indeed, the algorithm swaps `x.next` and `y.next` while the postcondition is inadequate.

The purpose of this chapter is to establish a technique that could automatically produce *counterexample heap diagrams* (like Fig. 4.1b) from raw SMT models, aiding the programmer in debugging verification failures. Automation is an essential prerequisite of a practically applicable verification debugging technique because manual inspection of raw SMT models is cumbersome and prone to errors; e. g. even for the simple case of Fig. 4.1a, the raw SMT model contains over 500 assignments.

PROBLEM STATEMENT. First, our technique should be *agnostic* to particular verifier implementations. Second, our technique should be *scalable*, e. g. debugging should not significantly increase verification time. The technique should succeed in producing counterexamples even for highly complex input programs. Third, our technique should visualize the counterexamples with minimal user input. The output should be intelligible yet informative evidence for the verification failure causes. Fourth, our technique should be *extensible*, i. e. adding new specification features should be a systematic process.

OUTLINE. In the remainder of the chapter, we first present and compare the most relevant existing verification debugging techniques (Sec. 4.1). We then present an approach to *manually* constructing counterexample heap models to verification failures that occur in our setting (Sec. 4.2); this includes our running example (Sec. 4.2.1).

The following four sections present our algorithm that automates those ideas. We introduce our instrumentation for the source program (Sec. 4.3); we then present the general algorithm for *automatically* producing counterexample heap models (Sec. 4.4) and the extended algorithm for debugging reachability-powered specifications (Sec. 4.5).

We then describe the implementation of our technique (Sec. 4.6) and present a case study (Sec. 4.7). We conclude the chapter with a discussion (Sec. 4.8).

4.1 EXISTING WORK

Two major lines of work target the problem of verification debugging. First, we will discuss the line concerning *dynamic verification debuggers* (Sec. 4.1.1). We will then discuss the alternative line — and the main focus of this chapter — techniques concerning *static verification debuggers* (Sec. 4.1.2). We summarize the state of the art in Tab. 4.1.

4.1.1 Dynamic verification debugging

A verification failure that is not spurious implies that there exists a set of program inputs exposing the problem, either by crashing the program or by reaching a program state in which the specified conditions are violated. If the specification can be implemented as

2 We assume for now that a method's footprint is the same in all states (e. g. there is no allocation); we will discuss how to overcome this practical limitation in Sec. 4.8.2.

runtime assertions, then violating these assertions will also crash the program, making crashes (e. g. unhandled exceptions) the one and only problem indicator.

This idea has led to *dynamic* verification techniques that leverage information about failed assertions to generate test cases. If the original specifications and their executable implementation can be trusted, then a crashing test provides concrete evidence for a bug in the source program. Conversely, if a test does not crash, then the verification error is spurious, e. g. it may be caused by an incompleteness or a bug in the verifier itself.

4.1.1.1 Spec# debugger. Müller and Ruskiewicz [63] propose a verification debugging technique (and a tool integrated into the Spec# system) that leverages counterexample SMT models to generate runnable test cases that can be analyzed by the programmer via a traditional debugger like GDB. The tool takes a (formally specified) Spec# program and the counterexample SMT model as input; it then produces an *executable* program that reproduces the semantics of the original in the states from the counterexample. The programmer then runs the output program in a traditional debugger, exploring a concrete execution via the common debug actions, e. g. *single-step* and *step-over*.

The executable program *simulates* Spec#'s modular verification semantics, e. g. a method call is replaced with a *program stub* that transforms the call's post-state as per the counterexample. Specifications of the original program that are relevant to the failure are translated into *runtime checks*. If a runtime check passes while the corresponding specification caused a verification failure, then the failure is spurious. Otherwise, the programmer can step through the executable program in the debugger, spotting implementation errors or unexpected state changes (which imply that the original specification is incomplete, or simply wrong).

Some specifications generally cannot be executed, e. g. the *modifies clause* of the method `IntList.Sort` that sorts a list of integers is encoded in Spec# via unbounded quantification since the set of all the list elements is potentially unbounded. To translate such specifications, the technique identifies finite sets of relevant objects based on the counterexample model and effectively pre-instantiates the universal quantifiers. The runtime checks are then generated only for the (executable) ground terms.

LIMITATIONS. This work provides programmers, who might not be verification experts, with a familiar interface for debugging verification failures in Spec#. The main limitation of this technique is that the specifications must be executable s.t. one can simulate them in a runnable program; hence, we cannot simply reuse the Spec# technique for an *intermediate verification language*, e. g. Viper.

4.1.1.2 StaDy (Frama-C). Petiot et al. [102] propose a technique for automatically classifying verification failures and generating counterexamples to formally specified C programs. The corresponding tool is integrated into the Frama-C platform [82]. If a verification fails, the technique first checks whether this is a *non-compliance* failure, i. e. the

implementation violates the specifications. For this purpose, the input programs is augmented with runtime checks corresponding to the program specifications. If a runtime check fails, then the technique extracts a concrete counterexample from the execution trace and reports it to the programmer.

Otherwise, each function of the original (specified) program is translated into a new executable program in which the specifications are rewritten via runtime checks. In this program, each *callee* is rewritten with a mock implementation that respects the callee's modular specification: A runtime check is added to check the precondition, and the values computed by the callee (which are listed in the `assigns` clause) are assigned to potentially *non-deterministic values* that satisfy the postcondition.

The resulting program is submitted to the *PathCrawler* test case generator [35] that yields counterexamples in the form of test input data.[3] To increase precision, the technique first runs the above procedure on individual specifications, which may result in detecting a *single contract weakness*. In case this strategy does not succeed, the technique proceeds with its best effort by repeating the procedure with all present specifications at once, which may result in detecting a *global contract weakness*. In case the procedure fails, the tool reports either *prover incapacity* (e. g. if all generated tests passed) or *unknown* (e. g. in case of a timeout). The programmer may annotate their functions with the `typically` clause that helps reducing the test generation input domain; e. g. a function that computes the factorial of `n` could be specified with `typically n` $\leq$ `10`.

LIMITATIONS. The focus of this work is on precise classification of verification failure causes. The technique does not require the programmer to have formal verification-specific expertise. The technique might not scale to larger software as the potentially high number of concrete executions of an input program impedes test coverage. Since the technique translates specifications to runtime checks, it is limited to executable specification languages and cannot be simply reused for intermediate verification languages.

4.1.2 Static verification debugging

Complementary to the dynamic techniques discussed above, *static verification debugging* techniques do not rely on execution of test cases. Static techniques apply not only to executable specification languages but also *intermediate verification languages*, e. g. Boogie [46], Viper [91], or WhyML [67], which are typically not intended to be executed. In the remainder of this section, we thus explore the extent to which one can leverage static information for understanding of verification failures.

In SMT-based verification, static debug information can come from two (potentially overlapping) sources: the verifier (e. g. information about types or inferred specifications) and the solver (e. g. an SMT model that represents a counterexample to an as-

3 PathCrawler is based on concolic execution and is built on top of the Colibri solver, supporting e. g. unbounded integer arithmetic and floating point theory.

sertion). The former source can be easily interpreted by the programmer, which allows directly displaying e. g. type errors. However, in case of a verification error, the feedback from the solver represents a counterexample model in terms of the *encoded* verification problem. Recall that the internal encoding of the verification technique is not exposed to the programmer in the paradigm of automated deductive verification. Therefore, the programmer cannot easily interpret raw counterexample models.

Verification debuggers are tools that automatically decode SMT-level counterexample models into information in terms of the program semantics. This decoding is typically complicated by the many transformations that a verifier performs before the ultimate proof obligations are discharged. We proceed with an overview of existing verification debugging projects.

4.1.2.1 Boogie verification debugger. The Boogie verification debugger (BVD) [62] aims at supporting the programmer's understanding of verification failures by mapping SMT models to the source language level. BVD's user interface displays program variables and their corresponding values in a table format, next to a menu in which the programmer can select a program state. Each table row displays the entry name and its value in both states: the current and the previous. Object fields are represented as (drop-down) list hierarchies; in case of reference-type fields, the programmer can explore the list to the desired depth, or until the null value is reached. Analogously, sets and function applications are also represented as drop-down lists, with entry names containing set elements and function argument instantiations, resp.

The debugging capabilities of BVD can be reused by a frontend verifier that translates source programs into the Boogie intermediate verification language [46]. The debugger maps the information from the SMT models obtained by the Boogie verifier to a counterexample in terms of the intermediate language, which is then translated to the source level by a language-specific plugin. The list of language plugins includes one for Dafny [57] and one for the VCC verifier [49] (the authors claim that implementing new plugins is not difficult). Both of these debuggers are integrated into the Visual Studio IDE, combining BVD with other useful features, e. g. dynamic test case generation, the axiom profiler [108], and procedures for diagnosing verification timeouts [85].

LIMITATIONS. While BVD provides a convenient interface for browsing verification counterexamples (which are practically too large to grasp as a whole), it does not try to pinpoint the cause of the problem. In particular, understanding bugs in a complex heap-transforming program requires manually constructing heap models: Navigating through the tree hierarchies does not directly present e. g. shared nodes or heap cycles.

4.1.2.2 Static debugging for WhyML. Hauzar, Marché, and Moy [87] present a verification debugger integrated into the SPARK 2014 system, featuring a verification condition generator and an SMT-based verifier (called GNATprove [81]) for a subset of Ada 2012.

Similar to BVD, this technique maps SMT models to the source language by looking up the values of model terms that match the tokens relevant at the source level, i. e. variable names. To explain verification failures, the debugger interleaves the program text with comments that represent an error trace with the values of program variables leading to a contradiction. The tool supports record and array types but does not support heap structures (which are not available in SPARK 2014); hence, pointer aliasing, framing, and heap reachability are out of scope of this work. Dailler et al. [96] generalize this technique, enabling verification debugging for several WhyML-based systems, e. g. the Java verifier Krakatoa [42] and the C verifier Frama-C/Jessie [89].

LIMITATIONS. This work demonstrates the benefit of developing language-agnostic verification debuggers that can be adapted to many different projects, avoiding the need to reimplement similar ideas for each language. However, the limitations of this technique arise from its two main assumptions. First, information about the relevant parts of a counterexample is assumed to be available at the program logic level (by labeling relevant terms in WhyML); it is unclear if one can obtain this information automatically. Second, the technique is tailored to verification condition generation, and one cannot simply apply it to verifiers based on symbolic execution.

4.1.2.3 Verification condition visualizer. The Verification Condition Visualizer [77] is a technique for visualizing verification conditions arising from array-transforming programs. The corresponding tool, called Auto-VCV, targets the SPARK language [27], a subset of Ada 95. The visualization procedure works in four steps. First, the procedure identifies the arrays that are referenced in the verification conditions. Second, it extracts the constraints over the index variables and array bounds. Third, the procedure orders the array elements and subsegments that correspond to the previously extracted indices. Fourth, the procedure derives the disjoint and the partially overlapping array fragments and *symbolically* calculates the gaps and overlaps among them.

The resulting information is then visualized as two schematic array diagrams. The top diagram summarizes the hypothesis of the verification condition, while the bottom diagram summarizes its conclusion (if there are multiple verification conditions, the tool first merges all the hypotheses and, separately, all the conclusions, and then renders the two array diagrams). In addition to the actual array segments, the diagrams depict *functions* (e. g. all elements in the segment $[0, x)$ are less than or equal to the element under x; depicted via a curly brace over the range leading to a $\leqslant$-labeled arrow to x) and operations (e. g. the elements under x and $x+1$ are swapped; depicted via a double arrow between the corresponding array elements).

LIMITATIONS. The main novelty of the technique is that it symbolically represents unbounded array structures. However, it is unclear if one can easily generalize the procedure to support more complex structures, e. g. multidimentional arrays, or graphs.

4.1.2.4 Effectively propositional reduction. *Effectively propositional reduction* (EPR) is a technique and a tool for reasoning about the existence and absence of paths in linked heap structures [69, 76]. The input program and its specification are translated into *the EPR logic*, a decidable fragment of first-order logic the assertions of which are automatically checked by an SMT solver. If an assertion is invalid, the tool produces a visual counterexample, i. e. a heap configuration that violates the assertion. We have discussed this work in more detail in Sec. 3.2.3.

LIMITATIONS. This technique is dedicated to reasoning about heap reachability properties in a sequential setting. While decidability of proof obligations guarantees automation, it comes at a cost: The class of supported heap structures is limited to various forms of linked lists. To our knowledge, the EPR tool does not have IDE support.

4.1.2.5 GRASShopper. *GRASShopper* [78] is a technique and a tool for verifying heap-transforming programs by translating separation-logic proof obligations into *GRASS*, the logic of graph reachability with stratified sets. GRASS is a decidable fragment of first-order logic, and its assertions can be automatically checked by an SMT solver. If an assertion is invalid, the tool produces a visual counterexample, i. e. a concrete heap configuration that violates the assertion. Unlike EPR, potentially undecidable specifications, e. g. unbounded first-order quantification, are also supported in GRASShopper; however, if the solver fails to validate such obligations, GRASShopper might not be able to provide a counterexample. We have discussed this work in more detail in Sec. 3.2.4.

LIMITATIONS. GRASShopper's visual counterexamples effectively aid the programmer in understanding verification failures. However, the debug information is limited to concrete heap edges and field values, which might be insufficient for understanding spurious failures in presence of complex specifications often leading to undecidable proof obligations. The Emacs mode for GRASShopper provides some tool integration, e. g. on-the-fly checking and highlighting of error locations from failed verifications.

4.1.2.6 VeriFast symbolic debugger. VeriFast [61] is a symbolic execution-based verification engine for C and Java. The project includes an IDE that integrates standard features, e. g. code editing and verification status reporting, with some verification debugging features [56]. The programmer can explore the symbolic execution trace with the path conditions accumulated so far, and the symbolic state. Information about the symbolic state consists of the local store variables with their symbolic values in the current state (e. g. `n=n0` where `n` is a local variable and `n0` is an unspecified constant) and a list of disjoint *heap chunks*, i. e. a symbolic representation of the currently held resources, (e. g. a singleton chunk `n0->next`, where `next` is a reference field, and an abstract chunk `list(n0->next)`, where `list(x)` is a separation-logic predicate defining an acyclic singly-linked list starting in `x`).

LIMITATIONS. While the VeriFast symbolic debugger provides useful information for verification experts, it assumes that the programmer understands both separation logic and symbolic execution. Additionally, the debugger does not seem to incorporate any information from the SMT models and does not display counterexamples.

4.1.2.7 Visual symbolic execution debugger. The Visual Symbolic Execution Debugger (VSDB) [55] is the first-generation visual verification debugger for the KeY verification tool [39]. VSDB targets the Java Card subset of Java; the tool is integrated into the Eclipse IDE [31]. Similar to the VeriFast IDE, VSDB enables the programmer to explore symbolic execution traces. One novelty in VSDB is that the tool visualizes symbolic traces as a tree graph, demonstrating more aspects of the symbolic execution, e. g. the degree of branching. Another novelty is that VSDB also visualizes the *symbolic state*, rendering (symbolic) heap configurations as diagrams. The user can specify the number of symbolic execution steps (e. g. unrolling a loop traversing a singly-linked list to five iterations), while the tools enumerates the possible heap models (e. g. an acyclic list segment of length five: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$, a ring structure with only three nodes: $a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow \ldots$, or a *lasso*: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow b \rightarrow \ldots$, etc.)

LIMITATIONS. VSDB was one of the first tools to visualize symbolic heap configurations that are fully helpful to programmers without verification expertise. However, the tool does not incorporate counterexample information from SMT models. Therefore, the programmer can only *manually* explore the symbolic execution tree and identify the symbolic states relevant to a verification failure, *manually* filtering the possible heap configurations to find those that showcase the problem.

4.1.2.8 Symbolic execution debugger. Symbolic Execution Debugger (SED) [109] is the second-generation visual verification debugger for the KeY verification tool [39]. SED targets sequential Java programs specified via JML [19]; the tool is part of the KeY IDE and is powered by Eclipse [31]. Similar to VSDB, this tool also visualizes the symbolic execution information in two parts, namely, the symbolic execution tree and the symbolic heap configurations. The former part is generalized to depict the execution tree modularly: The sub-trees corresponding to the symbolic execution of each statement are *graphically framed* and can be independently collapsed to improve readability. For example, each method call is depicted as a subtree with the invocation statement as its root (the point of checking the precondition), while all the return statements and exception handlers as its leaves (the points of checking the postcondition), conveniently matching the method's modular specifications. The heap visualization is analogous to that of VSDB. For example, the programmer can fix the path conditions and visually inspect the various possible configurations of aliasing heap nodes.

LIMITATIONS. SED is a powerful tool for inspecting symbolic execution traces of heap-transforming Java programs within the KeY verification tool set. SED helps understanding the code and why proofs succeed. However, its visualizations of the symbolic heap do not pinpoint a single problematic configuration that causes a proof to fail. In particular, SED does not show counterexamples to the failed proofs.

4.1.2.9 Viper IDE. Viper IDE [88] is a platform that integrates the Viper tool stack [91] into Visual Studio Code. In addition to the standard IDE features, Viper IDE supports interactive verification sessions that work with either of the verification backends available in Viper: the symbolic execution engine called Silicon [92] and the verification condition generator called Carbon [68].

Kälin [88] proposed the first automatic visual debugger that integrates into Viper IDE and leverages its symbolic execution backend. This debugger supports two modes: simple and advanced. In simple mode, the main window of the IDE is split into three columns. The first column contains the source code of a Viper program, interleaved with *state markers*, i. e. contrasting text decorators enumerating all symbolic states of the program. The second and third columns display information about *two program states*, e. g. the state of the verification failure and its predecessor state. These two vertical panels have the same structure, as described next. Similar to VSDB (Sec. 4.1.2.7), Viper IDE displays the heap configuration of the symbolic states graphically. However, the programmer does not need to traverse multiple possible configurations to find interesting cases, as Viper IDE is capable of rendering both concrete (e. g. an object referenced by `x` with a non-null field `next`) and abstract heap chunks (e. g. separation-logic predicates like `list(x)`) uniformly. Other displayed information about states includes the path conditions and the partial symbolic execution tree.

The advanced mode offers two major differences. First, the programmer may now simultaneously inspect and compare *any* two symbolic states, not necessarily subsequent ones. This enables many new debugging scenarios, e. g. comparing the symbolic states before and after two elements of a singly-linked list were swapped. Second, in case of a verification failure, the debugger in advanced mode incorporates information from the SMT model into the state diagram. This allows for displaying only the problematic scenario to the programmer, e. g. indicating which references are aliasing, which heap objects are disjoint, and whether the objects store the same values in their fields.

LIMITATIONS. This project laid the groundwork for Viper IDE and demonstrated the synergy of several verification debugging features combined in a coherent workflow. However, the visualization approach used for rendering the symbolic state supports only a special subset of the Viper language. The debugger heavily relies on symbolic execution traces and cannot be used with Viper's verification condition generator.

Although the support of two alternative verification backends is a key feature of Viper, there has not been a verification debugger to date that worked with Viper's verification condition generator, Carbon.

4.1.2.10 Viper to Alloy.

Aurecchia [95] developed an alternative debugger for Viper IDE that avoids the need to interpret SMT models for constructing counterexamples to verification failures by modeling the symbolic state in Alloy [65]. The motivation for this approach is twofold. First, SMT models are incomplete in presence of quantified formulas which frequently occur in practice. Hence, one needs to enumerate a potentially large space of possible model instances, some of which are equivalent (e. g. due to symmetry). Alloy applies symmetry breaking rules to generates conceptually different models. If a verification failure is caused by the solver's incompleteness, then the SMT model leads to spurious, confusing counterexamples. In contrast, Alloy produces the exact set of possible model instances (within bounds). Second, it is easy to extend the Alloy model of symbolic states to support new relations, or even to allow the programmer to submit their own custom queries, narrowing the space of possible concrete models.

LIMITATIONS. The Alloy-powered Viper debugger facilitates verification debugging and inspection of Viper programs by automatically visualizing essential concrete heap configurations. However, this approach has a number of limitations. First, Alloy models typically over-approximate the actual SMT constraints. This imprecision may lead to spurious models. Second, Alloy's support for integer reasoning is extremely limited; numeric counterexamples are thus highly imprecise. Third, the approach does not directly help understanding why the *solver* failed to validate an assertion as the technique does not model the solver's incompleteness, e. g. limited quantifier instantiation. However, a more urgent question is often *why* the tool rejects a specification (and not e. g. *which* heap configurations could satisfy or violate them).

4.1.2.11 Visual Ssmbolic execution models.

Stoll [105] proposed another approach called Visual Symbolic Execution Models (VSEM) for debugging verification failures in Viper. The main idea in this project is to embrace incomplete SMT models from failed symbolic executions, translating them to the verifier's abstraction level. Unlike the original Viper debugger, VSEM supports a broader subset of the Viper language. In particular, it is capable of displaying *quantified heap chucks* that occur in presence of iterated separating conjunctions (ISC) [90].[4] The corresponding tool produces diagrams that depict counterexample heap layouts in terms of heap chunks.

LIMITATIONS. While this project demonstrated the usefulness of partial SMT models in verification debugging, the produced output diagrams are low-level and cannot be fully

4 ISC is useful for specifying the memory footprint of an unbounded heap structure, e. g. arrays and graphs.

understood by non-experts. Another problem with this approach is that a backwards SMT translation is tailored to a particular verifier implementation.

4.2 MANUAL COUNTEREXAMPLE EXTRACTION

In this section, we present a novel approach to verification debugging that tries to overcome the key limitations of prior work. Like in the previous chapters, we demonstrate our technique in the context of (a subset of) Viper [91]. However, the key ideas of our technique are not specific to Viper. We proceed by defining the guiding principles for our technique.

VERIFIER-AGNOSTIC DEBUGGING. To our knowledge, all existing verification debuggers target either symbolic execution or verification condition generation, but not both. The support of both kinds of verification backends is a hallmark of the Viper ecosystem. Each backend has its pros and cons — both fundamental restrictions and implementation aspects — but, often, the ability to switch between them allows overcoming their (individual) limitations. Thus, the programmer should ideally get the same level of support for debugging verification failures with each backend.

Table 4.1: Summary of existing verification debugging techniques.

The techniques are grouped as follows. 1st — includes dynamic verification debugging techniques. 2nd — static VCG-debuggers. 3rd — concrete counterexample heap configurations. 4th — static SE-dubuggers. 5th — Viper debuggers. **Gr** = *group*; **SE** = *Symbolic Execution* (*dyn* = *dynamic SE*); **VCG** = *Verification Condition Generation*; **Vis** = *Visual debuggers*; **IDE** = *Tool stack integration*.

Gr	Technique	Discussion	Citation	SE	VCG	Vis	IDE
1	Spec#	Sec. 4.1.1.1	Müller and Ruskiewicz [63]		✓		✓
	StaDy (Frama-C)	Sec. 4.1.1.2	Petiot et al. [102]	*dyn*			✓
2	BVD (Boogie)	Sec. 4.1.2.1	Le Goues, Leino, and Moskal [62]		✓		✓
	SPARK / WhyML	Sec. 4.1.2.2	Hoang et al. [81]		✓		
	Auto-VCV	Sec. 4.1.2.3	Jami and Ireland [77]		✓	✓	
3	EPR	Sec. 4.1.2.4	Itzhaky et al. [76]		✓	✓	
	GRASShopper	Sec. 4.1.2.5	Piskac, Wies, and Zufferey [78]		✓	✓	
4	VeriFast IDE	Sec. 4.1.2.6	Jacobs, Smans, and Piessens [56]	✓			✓
	VSDB (KeY)	Sec. 4.1.2.7	Hähnle et al. [55]	✓		✓	✓
	SED (KeY)	Sec. 4.1.2.8	Hentschel, Bubel, and Hähnle [109]	✓		✓	✓
5	Viper IDE	Sec. 4.1.2.9	Kälin [88]	✓		✓	✓
	Viper-to-Alloy	Sec. 4.1.2.10	Aurecchia [95]	✓		✓	✓
	Visual SE Models	Sec. 4.1.2.11	Stoll [105]	✓		✓	

VISUAL DEBUGGING. We focus on verification debugging for heap-transforming programs. One of the hardest aspects of reasoning about such programs is understanding which subtle heap configurations are possible under specified assumptions. Typically, a verification expert needs to manually sketch such configurations in a pen-and-paper approach, possibly using the raw values from an SMT model, which is cumbersome and error-prone. However, existing visual debuggers demonstrate that this task can be automated, e. g. SED (Sec. 4.1.2.8) and Viper-to-Alloy (Sec. 4.1.2.10) for symbolic execution and Auto-VCV (Sec. 4.1.2.3) for verification conditions about unbounded structures (e. g. arrays). Our approach follows the lead, prioritizing diagrams over other possible representations of the mutable program state.

OBJECT-ORIENTED DEBUGGING. Rather than focusing on the low-level, backend-specific details of the verification backend, we opt for a debugger that is helpful to all programmers, even those who do not fully understand the intricacies of our program logic. For example, symbolic execution traces can be useful for a verification expert but might be counter-intuitive to programmers without special knowledge about the verifier. In contrast, we assume that all programmers understand essential concepts e. g. *local store*, *program heap*, *heap objects*, and *object fields*. Thus, in this chapter, we will build a debugger that highlights as few aspects of the program logic as possible,[5] while still helping to understand verification failures.

INTEGRATION. We are interested in a tool that seamlessly integrates into an existing IDE infrastructure that supports basic yet essential features, e. g. code navigation and editing, syntax highlighting, and interactive verification sessions. Thus, our implementation must be lightweight, leveraging common interfaces supported by modern IDEs.

4.2.1 Running example

To demonstrate our approach, consider the simple heap-transforming method `insert` that inserts the node `n` after the head node `hd` of a singly-linked list with footprint `g` (Fig. 4.2).[6] In the following, we will construct a counterexample heap model that helps understanding why this method fails to verify.

4.2.1.1 Overview. Recall that our setting requires each method to specify its footprint as a set of non-null nodes. Hence, the first precondition of `insert` requires permissions to access (the `next` fields of) all nodes inside the footprint `g` (via the macro `NODES`). The second precondition requires that the nodes `hd` and `n` belong to the footprint.

5 Note that the verification techniques presented in Chap. 2 and Chap. 3 involve a very simple subset of separation logic, i. e. no predicate abstractions.

6 This example was used as a running example for the Boogie verification debugger [62].

```
field next: Ref
define NODES(g) (null ∉ g && forall n ∈ g • acc(n.next))

method insert(g: Set[Ref], hd: Ref, n: Ref)
    requires NODES(g)
             hd ∈ g ∧ n ∈ g
    ensures NODES(g)
            n.next = old(hd.next) // fails to verify
{
    label l0
    n.next := hd.next
    label l1
    hd.next := n
    label l2
}
```

Figure 4.2: Example program that fails to verify due to incorrect specifications.

The first postcondition (which verifies) ensures that all the permissions held by `insert` are returned to the client (we assume that there are no memory leaks). The second postcondition ensures that the successor of `n` after the call is the successor of `hd` before the call. However, this postcondition fails to verify with the following report: *Postcondition of `insert` might not hold. Assertion `n.next = old(hd.next)` might not hold.*

What is the reason for this postcondition to fail? An experienced programmer would understand the issue by carefully reading the code and its specifications, concluding that this is an aliasing problem. However, significantly more information than the above error report should support this conclusion. In particular, we are interested in a counterexample that would demonstrate which initial heap configuration leads to a state violating our failed postcondition.

4.2.1.2 Structure of the SMT model.

To obtain a counterexample, we rely on the model provided by the SMT solver. Fig. 4.3 presents an example model containing full information about a counterexample. We start by introducing the structure of the original SMT model. Generally, models contain *entries* of tree kinds, as explained next.

CONSTANT ENTRIES assign literal values, called *inner values*, to variables. For example, `i ↦ 42` assigns the (integer) inner value `42` to the variable `i`, while `r ↦ v0` assigns the (uninterpreted) inner value `"v0"` to `r`. Since we consider partial models, some constant entries may be *unspecified*, e. g. `j ↦ unspecified`.

APPLICATION ENTRIES combine several argument literals into a single value by applying *interpreted functions*. For example, `p ↦ (/ 1.0 2.0)` assigns `p` to the result of an appli-

```
[3] ↦ {
    v8 v17 v2 ↦ v24
    v18 v17 v2 ↦ v30
    v25 v17 v2 ↦ v17
    v27 v17 v2 ↦ v26
    else ↦ unspecified
}
```

```
g ↦ v7

[2] ↦ {
    v7 v3 ↦ false
    v7 v17 ↦ true
    else ↦ unspecified
}
```

```
n ↦ v17
hd ↦ v17
null ↦ v3
next ↦ v2
Heap@1 ↦ v25
Heap@0 ↦ v27
Heap@@18 ↦ v18
PostHeap@0 ↦ v8
```

Figure 4.3: Partial SMT model for the failure of Fig. 4.2 (only relevant entries are shown).

cation of the function `/` over the literals `1.0` and `2.0`. Note that uninterpreted functions are *not* applied this way in the model.

MAP ENTRIES are lookup tables that assign values to function interpretations for some concrete cases of input arguments. For example, if the original program has the following declaration: `function equals(a:Ref,b:Ref): Bool {a = b}`, then the SMT model may contain the following interpretation: `equals ↦ { v0 v0 ↦ true, else ↦ false }`. Here, `equals` is a map entry that specifies two cases: The concrete cases map a list of arguments (`["v0","v0"]`, where `"v0"` is the inner value of the instantiations of `a` and `b`) to their corresponding values (e. g. `true`), while the keyword `else` maps all other argument lists (e. g. `["v1","v0"]` for some `"v1"`) to the *default value* — `false`, in the example above. The default value in a map entry may be unspecified (since we are dealing with partial models); e. g. `inv ↦ { v0 ↦ v1, else ↦ unspecified }`.

TYPES. Most verifiers *encode* the type system of a programming language via uninterpreted functions, rather than exclusively relying on built-in SMT *sorts* [94]. For example, Boogie declares two (uninterpreted) SMT sorts called `"T@U"` and `"T@T"`; a program value is translated to some value of sort `"T@U"`, while its program type is encoded via the (uninterpreted) function `"type"` that maps `"T@U"` to `"T@T"`. The solver models the `"type"` function using a *partial interpretation* which may assign the same `"T@T"`-value to program variable types which are incompatible at the level of the source program. For example, the inner value `"v1"` may be assigned to (the SMT translation of) the two variables `b:Bool` and `r:Ref`.[7] Therefore, we cannot rely on the type information from the SMT model and must obtain it directly from the verifier.

4.2.1.3 Modeling local variables. We will now demonstrate how the SMT model can be used to manually construct the counterexample for the verification failure in Fig. 4.2. We start by looking up the values that the SMT solver assigned to the (encoded versions of)

7 Partial SMT models may not satisfy *all* axioms. If e. g. the disjointness of the types of `b:Bool` and `r:Ref` is not needed to draw a contradiction, the solver may ignore this information while generating a model.

our *local program variables*. In our scenario, the three local variables are `g`, `hd`, and `n`, and their corresponding model values are resp. `"v7"`, `"v17"`, and `"v17"` (Fig. 4.3). Note that the indices `7` and `17` are picked arbitrarily by the solver and are not important. However, it *is* important that the model assigns *the same value* to `hd` and `n`; i. e., these variables will be aliasing in the model.

Since we are dealing with ***`Ref`***-type variables, it is also important to check whether the model assigns some of them to **`null`**. Hence, we perform an additional lookup for the value of **`null`** in the model, obtaining the value of `"v3"`. From this, we conclude that all our local variables are non-null in the model. This is an expected consequence of the precondition of `insert`.

Note that *mutable* local variables may have different values in different program states. The verifiers encode such variables in the *static single assignment* (SSA) form, i. e. each assignment of a variable `x` in the program, called the *prototype variable*, results in a fresh SMT variable, e. g. `x@1` and `x@2`. The information about prototype variables is maintained by the verifiers and used e. g. while reporting verification errors. However, our running example of Fig. 4.2 does not involve assignments to local variables.

To summarize the counterexample information obtained so far, we sketch the simple diagram of Fig. 4.4. This diagram shows the local store that contains our two (aliasing) ***`Ref`***-type variables, `hd` and `n`. Note that, for now, we omit the footprint `g` (all methods in our setting involve an explicit footprint). In the following, we will extract more information from the SMT model, refining our sketch of Fig. 4.4 to show the *heap configuration*.

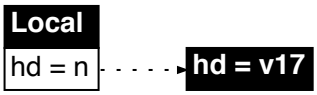


Figure 4.4: Sketching the local store of a counterexample to Fig. 4.2.

4.2.1.4 Modeling field values.

The next step towards building a complete counterexample is to extract the information about the *field values* from the model. The SMT encoding of fields — and, more generally, program states — varies in symbolic execution (SE) verifiers and verification condition generators (VCG). However, the information contained in the SMT models from either kind of backend is conceptually equivalent, i. e. one can extract the value of a given receiver's field in a given program state using the partial function interpretations provided in the models. To concretize the discussion, we demonstrate next how this can be done for Viper's *VCG* backend.

EXTRACTING FIELD VALUES FROM THE SMT MODEL. In Viper's VCG, each field is represented with a constant entry, e. g. `next` ↦ `v2` that assigns the field name `"next"` to the inner value `"v2"`. The *values* of the fields can be obtained by instantiating the *field-value* map entry.[8] The field-value map assigns triplets (corresponding to the program state,

8 For example, running Viper's VCG with the default configuration of Boogie results in models that specify field-values via a map entry called `"[3]"`, based on the arity of the corresponding Boogie map type (Fig. 4.3).

the receiver, and the field) to field values. For example, since our method `insert` has the line `hd.next := n`, we are interested in the inner value of `hd.next` in the program state after the assignment. We already have two out of three required arguments for instantiating the field-value map (recall that `hd` ↦ `v17` and `next` ↦ `v2`).

The missing argument for instantiating the field-value map is called the *state marker*, i. e. an inner value corresponding to a program state. In Viper's VCG, state markers are also encoded as constant entries. For example, our model contains the following entries that define various program states: `"Heap@0"`, `"Heap@1"`, `"Heap@@18"`, and `"PostHeap@0"`. In both cases, these state markers are formed in several stages of the translation (from Viper, to Boogie, to SMT). Each stage may have its own transformations, rewritings, and optimizations that can affect the number and the effective names of the state markers. Hence, the programmer needs knowledge about each transformation stage in order to map the state markers (as they appear in the SMT model) back to the program states.

Assume that the following mapping from state markers to program states is known: `"Heap@@18"` corresponds to `l0`, `"Heap@0"` to `l1`, and `"Heap@1"` to `l2`; the state marker `"PostHeap@0"` corresponds to the (unlabeled) state of the postcondition of `insert`.[9] Since we are interested in obtaining a counterexample in the program state labeled `l2` (i. e. after the second assignment), we look up the inner value of `"Heap@1"` in the model (Fig. 4.3), which assigns `"v25"` to this state marker. Finally, the lookup in the field-value map for the triplet `["v25", "v17", "v2"]` yields the inner value `"v17"`. Hence, we learn that `"v17"` is the value of `hd.next` in `l2`, i. e. it is a self-reference.

We apply the information about field values obtained so far to extend the counterexample diagram (Fig. 4.5). We visualize the object referenced by `hd` and `n` as *tables*; the table header contains the *name* of this object (i. e. one of the variables that references it) and its inner value from the model (in this case, `"v17"`). The rows of the table correspond to the fields of this object; since fields may store different information in different states, we annotate them with the state label (e. g. `l2` in `next[l2]`). The dotted arrow connects the local ***Ref***-type variables to the corresponding heap objects (since they are known to be non-null). The solid arrow depicts the *heap edge*, connecting `hd`'s only ***Ref***-type field to its destination.

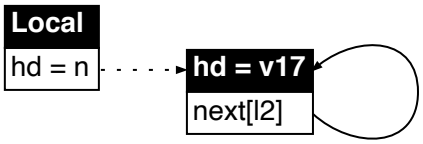


Figure 4.5: Extending the counterexample of Fig. 4.4 with intermediate heap information.

4.2.1.5 Handling multiple states. To illustrate how the information about multiple program states can be incorporated into a single counterexample, consider the state labeled `l1` in Fig. 4.2. This is the resulting state of the `insert`'s first assignment, `n.next := hd.next`. Hence, we are interested in the value of `n.next` in `l1`. To obtain this value, we perform a

9 This information can be obtained via the instrumentation that we will present in Sec. 4.3.

field-value lookup for the triplet ["v27", "v17", "v2"], representing "Heap@0", the node n, and the field next, resp. (recall that Heap@0 ↦ v27 is the state marker that corresponds to l1). This lookup yields the inner value "v26". We learn that the next field of the corresponding object changes its value from "v26" (in l1) to "v17" (in l2).

Next, by performing an analogous lookup for the value of n.next in the initial state of insert (labeled l0, corresponding to the state marker Heap@@18 ↦ v18), we learn that the value of n.next in l0 is *unspecified*.[10] We treat this case conservatively, concluding that the value of n.next in the initial state l0 is not relevant for the counterexample.

We apply the newly gathered information, extending our counterexample with the relevant field value information in multiple program states (Fig. 4.6).

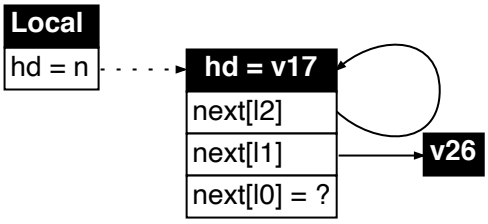


Figure 4.6: Extending the counterexample of Fig. 4.5 with information about old states.

4.2.1.6 Modeling method footprints. Recall that the SMT model assigns the inner value "v26" to n.next in state l1. Using the knowledge about the type of the field next, we conclude that "v26" must represent some ***Ref***-typed value. Since this value is non-null (as null ↦ v3), "v26" represents an object which itself may have a next field.

Recall that, in our setting, one can reason only about fields that belong to the nodes of the current method's footprint. In particular, the fields of the object represented by "v26" are relevant to the counterexample only if this node belongs to the set g (Fig. 4.2). Intuitively, we need to check whether the model implies v26 ∈ g.

The set-in relation that we are interested in is encoded into SMT in a conceptually equivalent way by both backends, SE and VCG. Generally, both backends use uninterpreted functions to encode the theory of sets. Concretely, the VCG backend translates Viper sets to Boogie maps, which in turn are translated to (partially axiomatized) uninterpreted functions that represent the operations and relations over instances of these types.[11] Analogously, the SE backend encodes its own theory of sets, but in the same style, i. e. employing uninterpreted function symbols for each set-related operation and relation, including the set-in relation, and supplies axioms that specify their properties.

To check whether hd belongs to the current footprint (according to the SMT model), we proceed with a new lookup in the interpretation of the set-in relation (which is a map entry of two arguments) with the values representing the footprint g and the node hd;

10 Although it is possible to force an SMT solver, e. g. Z3, to produce only complete models, we opt for partial models which provide the information that is likely needed for building helpful counterexamples. Conversely, it is not possible to identify what parts of a complete model are indeed relevant.

11 The resulting (binary) set-in relation is called "[2]".

recall that their respective inner values are `"v7"` and `"v17"` (Fig. 4.3). The lookup yields the value ***true***; hence, we observe that the node `hd`, indeed, belongs to the footprint `g`.

4.2.1.7 Transitive heap analysis. Other than `"v17"`, the only other non-null node in our counterexample is the one referenced by `hd.next` in `l1`, i. e. `"v26"` (Fig. 4.6). However, the lookup for the pair `["v7", "v26"]` in the set-in map entry is unspecified, so we cannot conclude that this node belongs to the footprint and treat its fields as irrelevant for the model, i. e. we do not proceed with analyzing its field values. Otherwise, we would transitively analyze the fields, repeating the previous steps for newly encountered nodes.

4.2.1.8 Final visualization. Finally, we apply the gathered information about the footprint to further refine our counterexample (Fig. 4.7). The resulting diagram is helpful: Indeed, the postcondition `n.next = old(hd.next)` (where **`old`** refers to `l0`) is violated if `n` and `hd` alias. The diagram also shows the *effect* that `insert` has on the heap configuration (cf. `next[l1]`, `next[l2]`), helping to understand that the implementation is not the problem of this scenario.

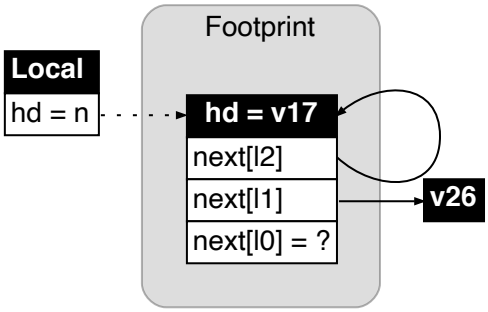


Figure 4.7: Final counterexample to Fig. 4.2.

This diagram shows a heap configuration contradicting the postcondition `n.next = old(hd.next)` of `insert`. We merge information from different states into one diagram. However, only *current* footprints are depicted (in some selected state); we will discuss footprints that change due to allocation in Sec. 4.8.2.

4.2.2 Systematic approach

We have manually extracted a visual counterexample to a verification failure using the information from the SMT model as well as some internal knowledge about the verifier (e. g. the types of local definitions). We will now summarize our findings, systematizing the assumptions (Sec. 4.2.2.1) and the stages (Sec. 4.2.2.2) of our technique.

4.2.2.1 Assumptions. We summarize the assumptions that enable our verification debugging technique in the setting introduced in Sec. 1.2.

PROGRAM LOGIC. We expect the footprint of each method to be explicitly specified (via iterated separating conjunction, ISC [90]). This allows us to process SMT models obtained from different verification backends *uniformly*. The backend-specific differences are minor, e. g. differing function signatures and field value lookups. However, the fact that we

extract counterexamples of a pre-defined shape drastically simplifies the problem: Our procedure for extracting counterexamples is agnostic to the most intricate differences in separation logic engines, e. g. permission accounting and framing.

MULTIPLE FAILURES. We focus on debugging one verification failure at a time. We therefore assume that both the verifier and the SMT solver do not mix information from several verification failures. In practice, verification errors often occur in groups, and modifying one line of the specifications can sometimes fix multiple errors. Each counterexample produced by a verification debugger should ideally focus on the cause of a single problem.

SMT MODEL. The raw SMT model is the main source of information that we used for building the counterexample of Fig. 4.7. Hence, whether our counterexamples can help the programmer in understanding verification failures is predicated on the precision of information in the SMT model. To generate *precise* counterexamples, e. g. those that ideally show only relevant information, we rely on partial models that should specify only the information that is necessary to draw contradiction. For the uninterpreted values in the model, we assume that the inner values do not alias (i. e. $\forall u, v \bullet u \not\equiv v \Rightarrow u \neq v$).

VERIFIER. Extracting relevant information from SMT models requires some internal knowledge (and assumptions) about the *verifier*. In particular, we assume that relevant parts of the program are *not optimized out* by the translation. For example, the variable `bar` does not affect the execution of the following program (which fails to verify):

```
method foo() { var bar: Int := 42; assert bar = 0 }
```

If the verifier performs inline expansion, the produced SMT model corresponding to the optimized failing assertion `42 = 0` will no longer contain any information about `bar`. Because this kind of information loss complicates debugging, we assume that optimizations like inline expansion are disabled in the verifier.

NAMING CONVENTION. We rely on the knowledge of the *naming convention* used internally by the verifier in its SMT encoding. For example, to extract the set-in relation, we relied on the name of the corresponding predicate. Note that, in the following, we do not assume the availability of the mapping from state markers to program states; although we *have* assumed the availability of this mapping for demonstrating our running example of Sec. 4.2.1; this information can be encoded in a verifier-agnostic manner, making our technique portable, as will be explained in Sec. 4.3.

4.2.2.2 Processing stages.

We will now summarize each of the model processing steps used for building the counterexample for Fig. 4.2. Although our running example is very simple, the concepts and decisions that were involved in producing the corresponding counterexample establish the basis of our technique, which we will further generalize.

1. Collect local definitions (Sec. 4.2.1.1)
2. Obtain the SMT model (Sec. 4.2.1.2)
3. Model local variables (Sec. 4.2.1.3)
4. Model field values for current heap nodes (Sec. 4.2.1.4)
5. Collect state markers (Sec. 4.2.1.5)
6. Model the current footprint (Sec. 4.2.1.6)
7. Model transitively reachable nodes (Sec. 4.2.1.7)
8. Visualize of the produced counterexample heap model (Sec. 4.2.1.8)

4.3 INSTRUMENTATION

This section introduces our verification debugging technique's instrumentation of the source program. The purpose of instrumentation is to slightly modify the original program, adding useful information that facilitates producing meaningful counterexample models. Theoretically, a verification debugger could obtain *full information* about the program and the verification outcome from the verification backend (if the verifier provides the required interfaces). However, we are interested in a verification debugging technique that does not rely on a particular verification backend implementation; ideally, our instrumentation should work uniformly with all backends, e. g. those based on symbolic execution as well as verification condition generation.

The information that our instrumentation provides is the mapping from (the interpretations of) stateful functions to program locations. This information is crucial because debugging heap-transforming programs and their verification failures requires *state-aware* counterexamples. We will first explain how handling states complicates counterexample production (Sec. 4.3.1), then present our instrumentation approach (Sec. 4.3.2), and then demonstrate how it helps processing the SMT model (Sec. 4.3.3).

4.3.1 Specifying the problem

Often, counterexamples to verification failures carry the information about multiple program states. In particular, this was the case for our running example, `insert` (Fig. 4.2): the postcondition `n.next == old(hd.next)` is a *two-state assertion*, depending on the configuration of two, potentially different versions of the heap (in this case, the versions of the post- and the precondition of `insert`). In terms of an SMT model, state-dependent functions (and their interpretations), e. g. the value of the field `next`, are parameterized with *heap markers*, i. e. inner values of the model specifying particular heap versions.

Hence, it is important to identify the heap markers and map them to the relevant program states of a counterexample.

Mapping from heap markers to program states is practically complicated. First, not all heap markers may be *relevant* for understanding a verification failure, e. g. some of them may encode auxiliary states used for checking permission amounts or that a state-dependent expression is well-formed. Second, SMT model may assign *different inner values* to two markers that represent effectively equal states. For example, it is possible that instantiating *all* heap-dependent functions in a model with the two heap marker $h_1 \mapsto$ `v1` and $h_2 \mapsto$ `v2` yields *the same* inner values. Third, heap markers do not *fully define* program states as they do not contain information about the *local store*, as discussed next.

TRACKING LOCAL STORE VERSIONS. If a counterexample carries the information about two program states, then it is important to distinguish the values that local variable have in either of them, as they may be different. Hence, a local variable may have multiple *versions*. Most verifiers rewrite local variables into the static single assignment (SSA) form, so each variable ends up having only one version in the SMT model. For example, a local variable declaration `var x:Int := 0` can be translated (e. g. in Boogie) into two statements (`var x:Int` and `x:=0`), each introducing a separate version of `x`, which are translated to SMT as follows:

```
(declare-const x@0 Int)
(declare-const x@1 Int)
(assert (= x@1 0))
```

The SMT model corresponding to the translation above would be, e. g. `x@0`$\mapsto$`17, x@1`$\mapsto$`0`. Here, the both translation and the model reflect the fact that there are two versions of `x`. In this case, we are able to deduce that `x@1` is the version of `x` after it is initialized, but `x@0` is *not relevant* for the source program. Next, we will discuss a simple yet general solution for identifying the relevant versions of local store variables and mapping them to particular locations in the source program.

4.3.2 Labeling program states

We have established that producing counterexamples to verification failures in our setting requires (1) filtering relevant heap markers (and mapping them to observable program locations), and (2) filtering relevant local store variables (and mapping them to observable program locations). To that end, one needs to instrument the debugged program with *state labels*, where each label has a name and identifies a location in the source program. The Viper intermediate verification language includes label declarations that enable *labeled old expression*, i. e. a generalization of **`old`** that allows referring to previous heap versions (cf. Fig. 1.1). However, Viper labels (and labeled old expressions) cannot be used to refer to previous versions of *local store* variables. Another limitation of Viper

```
method swap(g: Set[Ref], x: Ref, y: Ref)
requires NODES(g)
  requires x ∈ g ∧ y ∈ g ∧ x ≠ y
  ensures NODES(g)
          THIS_STATE_IS__l3 ⟹ x.next ≠ old(x.next)
{
  var tmp: Ref := x.next; assume THIS_STATE_IS__l0(tmp)
  x.next := y.next;       assume THIS_STATE_IS__l1(tmp)
  y.next := tmp;          assume THIS_STATE_IS__l2(tmp)
}
```

Figure 4.8: Instrumentation of an (incorrectly specified) program with state labels.

labels is that a label declaration must be a separate statement, e. g. a programmer cannot easily instrument the state of a loop invariant.

To overcome these limitations, we opt for an alternative way of instrumenting the program states. Our labels are pure expressions that can be soundly *assumed* at any program location, or within any expression, as long as our instrumentation requirements are met. Intuitively, a state label is merely an assumption that the program state (including both the heap and the local store) at a given program location has a given name. This immediately sets the first requirement: each label may be assumed only once (just like Viper labels that must have unique names).

EXAMPLE. To illustrate our instrumentation, consider the example of Fig. 4.8. The method `swap` swaps the nodes referenced by `x` and `y` of a singly-linked list (represented by the node set `g`). The precondition requires permissions to access the list's nodes, and that `x` and `y` are (non-aliasing) members of this list. The postcondition ensures that all permissions are returned to the client, and tries, but fails to ensure that `swap` preserves the value of `x.next`.[12] The body of `swap` uses a temporary ***Ref*** variable, `tmp`, to (correctly) perform the actual swap.

We instrument the program states after each statement in the body of `swap`. Each instrumentation starts with **`assume`** where the assumed predicate is a macro definition named `"THIS_STATE_IS__li"`, where $i \in \{0, 1, 2\}$ is the current state's specified index. These macros are defined in Fig. 4.9. Since the verifier *fails to verify* the last bit of `swap`'s postcondition, we also instrument the postcondition, this time — by adding the macro `"THIS_STATE_IS__l4"` as the LHS of an implication; the implication's RHS is the original postcondition that we wish to debug.

The auxiliary definitions used in our instrumentation technique are presented in Fig. 4.9. Each state labels' names have the form `"l`i`"` where i is this state's index. The corresponding macro definition is rewritten into a conjunction in which the first conjunct

12 Such an error could easily happen in practice due to a typo; e. g. `x.next = old(y.next)` differs in only one character and would be a correct specification for `swap`.

```
function @heap(g: Set[Ref]): Map[Ref, Ref]
  requires NODES(g)
  ensures domain(result) = g
          ∀u:Ref • {u.next} u ∈ domain(result) ⟹ result[u] = u.next

function l0(): Map[Ref, Ref]    function @local$tmp$l0(): Ref
function l1(): Map[Ref, Ref]    function @local$tmp$l1(): Ref
function l2(): Map[Ref, Ref]    function @local$tmp$l2(): Ref
function l3(): Map[Ref, Ref]

define THIS_STATE_IS__l0(tmp) (@heap(g) = l0() ∧ tmp = @local$tmp$l0())
define THIS_STATE_IS__l1(tmp) (@heap(g) = l1() ∧ tmp = @local$tmp$l1())
define THIS_STATE_IS__l2(tmp) (@heap(g) = l2() ∧ tmp = @local$tmp$l2())
define THIS_STATE_IS__l3      (@heap(g) = l3())
```

Figure 4.9: Auxiliary instrumentation definitions.

is the equality of the form `@heap(g) = l`i, and the subsequent conjuncts have the form $\bigwedge$ `v = @local$v$`i, where the conjunction iterates over all (mutable) local variable names (`v`) and the labels instrumenting the program locations in which the corresponding variables are defined (i). For example, the local variable `tmp` is defined in `l0`, `l1`, and `l2` but is not in scope of `l4` (Fig. 4.8); hence, the instrumentation macro for `l4` does not depend on the value of `tmp`.

INSTRUMENTING THE HEAP. In each instrumented program location, we assume that the value of the `@heap(g)` function over the current footprint equals some freshly selected constant; these constants are represented via nullary functions, e. g. `l0()`–`l3()`. To ensure that *any* modification of the footprint will result in a new value of `@heap(g)`, we axiomatize the function's (***Map***-type) value in its postcondition. We first consider the simple case in which all nodes within the current footprint have only one field, and then discuss how to generalize.

An assignment to a field (say, `next`) of a node (say, `u`) in the current footprint (`g`) should trigger the quantifier in `@heap(g)`'s postcondition (e. g. this should happen after `x.next := ...` and `y.next := ...` in Fig. 4.8). The instantiation results in the formula `result[u] = u.next`, connecting the field value to the value of `@heap(g)` at `u`; intuitively, modifying the `next` field potentially modified the resulting map. Since this map depends on each object's field, any modification of the heap inside `g` will result in a state in which `@heap(g)` yields a fresh value.

Recall the *snapshot functions*, `rsnap`, used internally in our set comprehensions technique of Chap. 2. The argument of `rsnap` is the current method's footprint, `g`, and the result is a map from ***Ref*** to a polymorphic type `S` (i. e. the type of a comprehension expression). Consider an instantiation of `S` with the field type, e. g. ***Ref*** in the example of `swap`. Notice that the postcondition of `@heap` exactly matches that of `rsnap`. Indeed:

our heap instrumentation function is a special case of a set comprehension. Therefore, to generalize our instrumentation to the case of multiple fields per node, one can combine multiple snapshots as follows:

$$\texttt{define THIS\_STATE\_IS\_\_}li(F, L)\ \Big(\bigwedge_{f \in F} \texttt{@heap\$}f(\texttt{g}) = li\texttt{\$}f\Big) \wedge \Big(\bigwedge_{v \in L} v = \texttt{@local\$}v\texttt{\$}li()\Big)$$

Here, i is the index of the state that is to be instrumented,[13] F is the set of names of all the field modified by this method, and L is the set of local variable names.

4.3.3 Processing instrumented SMT models

We will now revisit our example of Fig. 4.8 to explain how one can extract relevant information from an SMT model of an instrumented program. Recall that the method `swap` in this example has a buggy specification; concretely, its postcondition causes a verification error. This error is accompanied by an SMT model that combines our instrumentation with information about a counterexample to the failed postcondition (Fig. 4.10).

```
[3] ↦ {
    v20 v15 v2 ↦ v33
...}

@heap ↦ {
    v20 v7 ↦ v39
    v30 v7 ↦ v31
    v34 v7 ↦ v44
    else ↦ unspecified
}
```

```
Heap@0 ↦ v34
Heap@1 ↦ v30
Heap@@14 ↦ v20
PostHeap@0 ↦ v8

@local$tmp$0 ↦ v32
@local$tmp$1 ↦ v32
@local$tmp$2 ↦ v32
```

```
g ↦ v7
x ↦ v15
y ↦ v18
l0 ↦ v39
l1 ↦ v44
l2 ↦ v31
l3 ↦ v31
next ↦ v2
tmp ↦ v19
tmp@0 ↦ v32
```

Figure 4.10: Partial SMT model for the failure of the instrumented method `swap`.

PROCESSING THE MODEL OF THE HEAP. The name templates of heap markers depend on the verification backend. In our example, we run the verification via Viper's VCG backend, so the expected heap markers contain `"Heap"` in their names, i. e. `Heap@0`, `Heap@1`, `Heap@@14`, `PostHeap@0` in the model of Fig. 4.10. Recall that the heap markers parameterize each state-dependent function. In particular, a heap marker is the 1st argument of `$heap` in the SMT encoding (the 2nd argument is the current footprint). Therefore, we obtain the following mapping from heap markers to inner values of `$heap`: `Heap@0` ↔ `v44`, `Heap@1` ↔ `v31`, `Heap@@14` ↔ `v39`. Note that the value of `$heap` for `PostHeap@0` is *unspecified*; this is because the corresponding heap is not instrumented and should therefore be ignored.

13 The indices can be selected by the programmer (as long as they are unique), labeling the states that are relevant for debugging.

Recall that the value of `$heap` applied over the current footprint equals its corresponding label's value (Fig. 4.9). Therefore, we can use the the model to obtain the mapping from heap markers to *sets of labels*:

Heap val.		Marker		Label val.		Labels
`v20`	⟷	`Heap@@14`	⟷	`v39`	⟷	{ `l0` }
`v34`	⟷	`Heap@0`	⟷	`v44`	⟷	{ `l1` }
`v30`	⟷	`Heap@1`	⟷	`v31`	⟷	{ `l2`, `l3` }

(HeapsToLabels)

Here, we added the information about the *heap markers'* inner values (at the very left), followed by the heap marker names, the inner values of the corresponding cases of `$heap`, and the sets of labels.

We can now use the information of (HeapsToLabels) to extract the values of heap-dependent functions from the model. For example, we obtain the value of `x.next` in the program state labeled `l0` by performing a lookup into the field-value function, `[3]`, by instantiating its three arguments with `v20` for the heap of `l0`, `v15` for the receiver `x`, and `v2` for the `next` field (obtaining the inner value of a fresh node `v33`). Note that multiple labels might be *merged* in the model, effectively specifying the same heap; in our example, this *aliasing* situation occurs between `l2` and `l3`.

PROCESSING THE MODEL OF THE LOCAL STORE. Finally, we demonstrate the mapping from labels to the local store versions. To that end, we collect and parse all the *local variable snapshots*, i. e. `@local` constants in the model. Each such constant provides the information about the inner value of a local store variable in a labeled state. In our example of Fig. 4.10, the only local variable is `tmp`, and its value is `v32` in the states labeled `l0`, `l1`, and `l2` and is *unspecified* in `l3`. Indeed, this variable is assigned only once in the body of `swap` (Fig. 4.8), and it is not available in its postcondition.

Similar to heaps, store versions may alias, too. While, in our example, the first three labeled local stores *are* aliasing, the store labeled `l3` is unique (as it lacks `tmp`). Hence, in the SMT model obtained for `swap`, the actual *program states* (consisting of a heap and a local store) that are labeled `l0`–`l3` are all pairwise distinct.

This example demonstrates that our instrumentation provides a precise mapping from labeled program locations to the corresponding versions of the heap and the local store. Additionally, the instrumentation allows us to filter out the redundant heap markers from the SMT model. In particular, one can see that the model entry `tmp`↦`v19` is *irrelevant* for the counterexample, while `tmp@0` *is* relevant.[14]

LABELING LOOP INVARIANTS. Conceptually, loop invariants are assertions that are *checked* before entering the loop and after a loop iteration; they are also *assumed* at the beginning of a loop iteration and after the loop terminates [7, 8]. Since checking a loop invariant

14 The extra `tmp` symbol is an artifact of Boogie's SSA transformation; see Sec. 4.3.1.

may fail, the programmer should be able to label and debug the corresponding program states. A straightforward approach is to label the state before the loop (for debugging failed invariant *entry* checks) and the final state of the loop body (for debugging failed invariant *preservation* checks). However, the latter state is not exactly the same as the one in which the verifier checks the invariant, e. g. due to the limited lexical scope of local variables. To guarantee that the right state is labeled, we propose an alternative, more precise approach.

This following approach requires *inhale-exhale* assertions, which are available e. g. in Viper [91]. An inhale-exhale assertion `[`A`, `B`]` behaves as A when the assertion is assumed, while the same assertion `[`A`, `B`]` behaves as B when it is checked. If the programmer wishes to label a loop invariant, say A, with some label, e. g. `"l1"`, then our instrumentation rewrites this invariant as `[`A`, THIS_STATE_IS__l1`$(F, L) \implies A$`]`. As before, F is the set of names of all the field modified by this method, and L is the set of local variable names. The new invariant behaves as the original one in all places in which the verifier assumes it. Conversely, the verifier *checks* a modified version of the invariant that has our state label assumption as the LHS of the implication (and the original invariant as the RHS). Thus, we label the state of a possible failure of the invariant's *preservation* check.

4.4 GENERAL ALGORITHM

In this section, we introduce our algorithm (Alg. 1) for producing counterexample heap models. This algorithm generalizes and further systematizes the ideas and observations introduced in Sec. 4.2. In particular, our algorithm automates the steps listed in Sec. 4.2.2.2.

4.4.1 Algorithm overview

The entry point of Alg. 1 is ProduceHeapModel.[15] This procedure takes two arguments: the *program definitions* (`pds`) and the *raw SMT model* (`model`). The former maps symbol names to *definitions*, i. e. instances of type `ProgDef`, each storing the name and the type of the corresponding program definitions. We assume that these definitions are exactly the ones that are relevant to the current verification failure for which the algorithm intends to produce counterexample models.

RAW SMT MODELS. The structure of `model` is slightly more involved, mapping model names to *entries*, i. e. instances of type `Entry`. Its two subtypes define *constant entries* (`ConstEntry`) and *map entries* (`MapEntry`), resp. In addition to entry names, constant en-

15 Our pseudocode is statically typed with subtyping. For brevity, we infrequently write type annotations to improve readability and in pattern matching. E. g., `ConstEntry` is a subtype of `Entry`, so the λ-function in `model.`*`collect`*`(λe: ConstEntry · ...)` is *partial*, applying only to `model` elements of type `ConstEntry`.

Algorithm 1 Produce Heap Model

```
1:  interface Entry(name)
2:  struct ConstEntry(name, val) <: Entry
3:  struct MapEntry(name, apply) <: Entry
4:  struct ProgDef(name, type)
5:  procedure ProduceHeapModel(pds:{str ↦ ProgDef},      ▷ Program definitions
                               model:{str ↦ Entry})      ▷ SMT model
6:      states ← CollectProgStates(pds, model)
7:      eqcls ← CollectInitialEquivClasses(pds, model, states)
8:      graphs, nodes, scalars, fields ← Saturate(pds, model, states, eqcls))
9:      ftprints ← ConnectGraphsAndNodes(model, graphs, nodes)
10:     return states, ftprints, graphs, nodes, scalars, fields
```

tries store (concrete) values, i. e. the inner value that the SMT solver assigns to a variable. In contrast, map entries model function interpretations; in order to obtain a concrete value from a map entry, one needs to perform a lookup based on some arguments. `MapEntry` provides the method *`apply`* that abstracts such lookups; this method takes a list of arguments and returns a value corresponding to the map lookup for these arguments. The size of the argument list corresponds to the arity of the modeled function. For example, if our source program declares **`function`** `bar(i:`***`Int`***`, b:`***`Bool`***`):` ***`Ref`***, then the interpretation of `bar` in the SMT model is a map entry with the name `bar` that assigns cases e. g. `i0 b0` ↦ `r0`, where `i0`, `b0`, and `r0` are some constants the types of which can be interpreted as ***`Int`***, ***`Bool`***, and ***`Ref`***, resp.

For simplicity of the presentation, we do not consider the case of *application entries*. Application entries *without* arguments, e. g. `(f 1)` for an uninterpreted function `f` in the SMT notation, can be serialized in a pre-processing step and treated as constant entries by the rest of our algorithm. Application entries *with* arguments, e. g. `(= (:var 0) 42)`, sometimes occur inside map entries, specifying conditions over function arguments under which a particular case should be taken (here, `(:var 0)` refers to the value of the first argument of a map entry). Again, a preprocessing step can rewrite such map entries without the use of application entries. Refer to Sec. 4.2.1.2 for an overview of the structure of SMT models.

HIGH-LEVEL STEPS. The algorithm produces heap models in four steps. First (line 6), the procedure CollectProgStates identifies and returns the relevant *program states*. A program state is an instance of type `State` (Alg. 2, line 12), containing the versions of the program heap, the local store, and the corresponding state labels.[16]

16 Although our algorithm produces counterexample heap models for one verification failure at a time, the cause of each failure may involve the information about multiple program states.

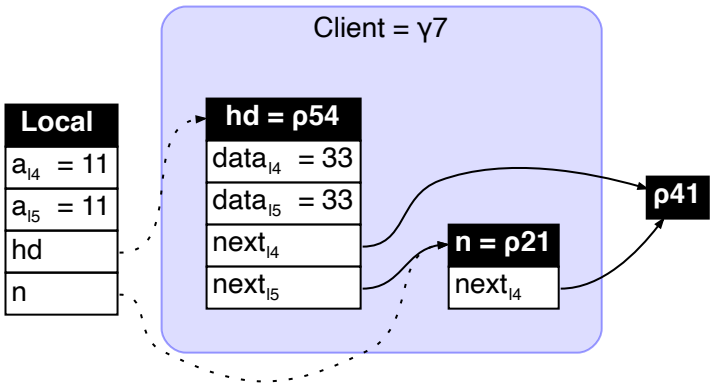


Figure 4.11: Rendering of a heap model.

The diagram shows our visualization approach for the outputs generated by Alg. 1. This diagram depicts a superposition of two program states (`l4` and `l5`). The local store contains a variable `a` and two constants, `hd` and `n`, that reference distinct objects of the client's footprint (purple box). The object referenced by `hd` has two fields, `data` and `next` (depicted in either of the two states), while the object referenced by `n` depicts only the field `next` in the state `l4` (the information about other fields and states is not relevant for in this example). The diagram shows that `hd.next` and `n.next` reference the same object (denoted ρ40) in `l4`, yet `hd.next` references `n` in `l5`.

Second (line 7), the procedure CollectInitialEquivClasses identifies and returns the initial *equivalence classes*, i. e. instances of `EquivClass` (Alg. 2, line 49) modeling the relevant (potentially aliasing) program variables. For example, the formal arguments of the current method (including its footprint), local variables, as well as `null` (which has its own inner value in the model) all form the initial equivalence classes.

Third (line 8), the procedure Saturate transitively analyzes the ***`Ref`***-fields of all reachable objects (within the current method's footprint). This procedure returns four objects: (1) `graphs` is the set of all (equivalence classes modeling) ***`Set`***`[`***`Ref`***`]`-type variables; (2) `nodes` is the set of all relevant heap objects; (3) `scalars` is the set of all *other* objects, e. g. local integer variables, and (4) `fields` is the list of all relevant *heap adjacency relations*, i. e. instances of type `Relation`. Since the input model is finite, this analysis saturates after a finite number of steps (Sec. 4.4.4).

Fourth (line 9), the procedure ConnectGraphsAndNodes identifies which nodes belong to which graphs, and returns `ftprints`, containing the graphs that model the footprints (both for the caller and the callee methods).

Rendering visual heap diagrams based on the returned objects (line 10) is straightforward. Fig. 4.11 shows an example of the rendering of our algorithm's end result (via a layout interface implemented on top of Graphviz [16]).

4.4.2 Modeling program states

INSTRUMENTATION. Generally, the SMT model contains information about *all* program states needed for the verification of a program, including some that are not observed by the programmer, e. g. auxiliary states for checking that state-dependent expressions are well-formed. To produce a meaningful counterexample that can be understood by any programmer, and not just the experts familiar with the verifier's implementation, our algorithm requires *labeling* all program locations that should be considered for the counterexample. We described our instrumentation technique in Sec. 4.3; we will now explain how program states are treated algorithmically.

HIGH-LEVEL OVERVIEW. The procedure CollectProgStates (Alg. 2) takes the program definitions and the SMT model and returns a list of *states*. Each state is an instance of `State` with four fields: `val` is the inner value of heap markers, `aliases` is the set of heap markers, e. g. `{"Heap@0", "Heap@0"}` (the model may assign the same value to multiple heap markers), `labels` is the set of state labels, e. g. `{"l0", "l1"}` (multiple labels may end up referring to the same state in the counterexample), and `storeHash` is a value that uniquely represents each store version, e. g. `"x=1;y=2"` for a local store with two integer variables `x` and `y`.

The procedure works in three phases. Phase A extracts the information about state labels from the instrumentation (Sec. 4.4.2.1). Phase B extracts the raw heap markers from the model and maps them to the corresponding state labels (Sec. 4.4.2.2). Finally, Phase C consolidates potentially equivalent states (Sec. 4.4.2.3).

4.4.2.1 Extracting state labels. The first step of this phase declares and populates the map `labelVarVal` from label names to store variables to their corresponding inner values (lines 14 to 17). To that end, we first iterate through the model entries (line 15), filtering constant entries whose name starts with `"@local"`; recall that this is the prefix of local variable snapshots in our instrumentation technique Sec. 4.3.2. Second, we decode the labels, extracting a variable name and the corresponding label (line 16). For example, `"@local$x$l7"` is `x`'s snapshot in a state labeled `l7`. Finally, we store this information in the `labelVarVal` map (line 17).

Second, the procedure maps label names to the corresponding *store hashes* (lines 18 to 21). This mapping is store in `storeHash` (line 18). We iterate through the value/key pairs of `labelVarVal`; each pair defines a mapping from variable names to values and a state label, resp. (line 19). We then compute the *store hash* for this label's state (line 20). The store hash is simply a string representation of pairs of variable names with their corresponding values; e. g. if `varToVal` is `{"x" ↦ "1", "y" ↦ "2"}`, then we obtain `"x=1;y=2"` for the store hash. We then save the store hash into the `storeHashes` map under the current label's name (line 21).

Third, we collect the model entries that assign values to labels (lines 22 to 23). If the source program is correctly instrumented, then state labels should be constant entries in the model, their names should be present in the program definitions (`pds`), and their type should be `"Label"`.

Fourth, we create the lookup table `storeHashes` from (label value/store hash) pairs to label names (lines 24 to 28). The purpose of step is twofold: (1) Hashing labels by their inner values and store hashes ensures that equivalent labels are consolidated. Recall from Sec. 4.3.3 that multiple labels may end up referring to equivalent states in a particular model, e. g. due to an effectless field update `x.next := y` where `x.next` already equals `y` before the assignment. Note, however, that versions of both the heap and the local store are state equivalence factors, hence the pair structure of keys (line 26). (2) Additionally, the `storeHashes` map helps labeling heap markers with consolidated state labels in Phase B (Sec. 4.4.2.2).

Fifth, we declare and populate the auxiliary map `valToStores`, this time — from label names to the corresponding store hashes (lines 29 to 32). To that end, we iterate through (label value/store hash) pairs, i. e. the keys of `storeHashes` (line 30), aggregating store hashes per label inner values (lines 31 to 32). This map enables a concise yet efficient implementation of the next phase of CollectProgStates, as presented next.

4.4.2.2 Combining stores and heaps. Equipped with the state labels, we are now prepared for Phase B: combining stores and heaps into labeled states. We first collect the heap markers from the model (line 33). Recall that heap markers are constant entries named according to some backend-specific convention. To abstract over this convention, we use a predicate called `isHeapSnapshot`. For example, `isHeapSnapshot(e)`, where `e` is a constant entry, can be rewritten as `e.name.includes("Heap")` for Viper's VCG, or as `e.name.includes("$FVF")` for Viper's SE engine.[17] [18] We then use the entry to construct a new *heap*, i. e. instance of type `Heap` (line 11), storing the heap marker's inner value and initializing the set of its aliases with the marker itself (line 34).

We then declare the `states` map from (heap inner value/store hash) pairs to actual states (line 35). Note that states are instances of type `State` (line 12) which extends `Heap` with two fields: `labels` (a set of labels referring to this state) and `storeHash` (the string representation of the local store in this state). To populate `states`, we iterate through `heaps` (line 36). For each heap `h` value `h.val`, we perform a lookup into the `"@heap"` function, obtaining the inner value of a state label, `labelVal` (line 37). We then check if this value corresponds to some store hashes in the `valToStores` map; if this is the case, we *iterate* through these store hashes (line 38).

Next, we combine the current heap's value (`h.val`) with the current store hash (`storeHash`) into a single key (line 39); we proceed only if this key is not yet present in our `states`

17 `str.includes(sub)` evaluates to true *iff* `sub` is a substring of `str`.
18 In Viper's SE engine, FVF stands for *field-value function* [92].

Algorithm 2 Collect Program States

```
11: struct Heap(val, aliases)
12: struct State(val, aliases, labels, storeHash,
                    hash ← λ() · [labels.join(), storeHash].join()) <: Heap
13: procedure CollectProgStates(pds, model)
                    ..... Phase A: Extract State Labels .....
14:   labelVarVal ←{}                          ▷ Maps labels to store variables to values
15:   model.foreach(λentry: ConstEntry · entry.name.startswith("@local"))
16:     prefix, var, label ← entry.name.split("$")          ▷ E.g. "@local$x$l7"
17:     localVarVal[label][var] ← entry.val
18:   storeHashes ←{}                              ▷ Maps labels to store hashes
19:   labelVarVal.foreach(λ(varToVal, label) ·          ▷ Compute store hashes
20:     storeHash ← varToVal.map(λ(val, var) · var + "=" + val).join(";")
21:     storeHashes[label] ←storeHash )
22:   labelEntries ← model.filter(λe: ConstEntry ·          ▷ Collect label entries
23:     e.name ∈ pds ∧ pds[e.name].type = "Label")
24:   valStoreToLabels ←{}          ▷ Maps (label value/store hash) pairs to labels
25:   labelEntries.foreach(λe ·    ▷ Merge labels by (label value/store hash) pairs
26:     key ← (e.val, storeHashes[e.name])                  ▷ Keys are ADT pairs
27:     key ∈ valStoreToLabels ? valStoreToLabels[key].extend(e.name)
28:                            : valStoreToLabels[key] ← [e.name])
29:   valToStores ←{}                          ▷ Maps label values to store hashes
30:   valStoreToLabels.keys().foreach(λ(storeHash, labelVal) ·
31:     labelVal ∈ valToStores ? valToStores[labelVal].extend(storeHash)
32:                            : valToStores[labelVal] ← [storeHash])
                    ..... Phase B: Combine Stores and Heaps .....
33:   heaps ← model.collect(λe: ConstEntry ·          ▷ Collect heaps from model
34:     isHeapSnapshot(e) ⇒ Heap(e.val, aliases ←{e.name} ))
35:   states ←{}                              ▷ Maps heap/store pairs to states
36:   heaps.foreach(λh ·                                ▷ Collect labeled states
37:     labelVal ← model["@heap"].apply([h.val])                ▷ Model lookup
38:     labelVal ∈ valToStores ⇒ valToStores[labelVal].foreach(λstoreHash·
39:       key ← (h.val, storeHash)
40:       key ∉ states ⇒                              ▷ Drop redundant labels
41:         labels ← valStoreToLabels[(labelVal, storeHash)].sorted()
42:         states[key] ← State(h.val, h.aliases, labels, storeHash)))
                    ..... Phase C: Consolidate Equivalent States .....
43:   unique ←{}                              ▷ Maps label hashes to states
44:   states.foreach(λs · key ← s.labels.join("/")      ▷ Merge states by labels
45:     key ∈ unique ? unique[key].aliases.extend(s.aliases)
46:                  : unique[key] ← s)
47:   return unique.values()
```

map[19] (line 40). If this is a *fresh* key, i. e. we have not already encountered for a state with these exact heap marker value and store hash, then we retrieve the corresponding state labels from `valStoreToLabels` (note that the key here is different, combining the *label's* inner value with this store hash), ascendingly sort, and store them into `labels` (line 41). Finally, we construct a new `State` instance (resembling our heap, `h`, but also storing the corresponding labels and the store hash), and save it into `states` under this `key` (line 42).

4.4.2.3 Consolidating equivalent states. The final phase of CollectProgStates consolidates equivalent states (lines 43 to 46). First, we declare the map `unique` from *label hashes* to states (line 43). Label hashes are string representations of multiple labels, e. g. the hash of `{"l0", "l1", "l2"}` is the string `"l0/l1/l2"` (recall that we store *sorted sets* of labels). We then iterate through each state `s`, computing and saving its hash into `key` (line 44). If `key` is already present in `unique`, we consolidate the previously saved state (`unique[key]`) with the new state (`s`) by merging their aliases (line 45). Otherwise, we save the fresh state `s` under `key` in `unique` (line 46).

In the end, CollectProgStates returns the consolidated states, i. e. the values of the `unique` map (line 47).

4.4.3 Modeling initial equivalence classes

This step is implemented via the procedure CollectInitialEquivClasses (Alg. 3) that takes the program definitions (`pds`), the raw SMT model (`model`), and the list of relevant program states (`states`), returning a list of initial equivalence classes. The algorithm works in three phases: Phase A collects constants (Sec. 4.4.3.1), Phase B collects mutable variables (Sec. 4.4.3.2), and then Phase C forms the equivalence classes amongst them (Sec. 4.4.3.3). We will first describe the structures used by CollectInitialEquivClasses and then explain its three phases.

STRUCTURE OF ATOMS. Each equivalence class summarizes a set of *atoms* — instances of type `Atom` (line 48) — that have the same inner value (`val`) and type (`type`). In addition to `val` and `type`, these objects have the following three fields: (1) The field `proto` stores the name of this atom's *prototype* variable; (2) `state` stores either the state in which this atom is *active* (i. e. relevant to the counterexample), or the special value `None` (if this atom models a constant value); (3) The `isLocal` flag indicates whether this atom models a local store variable (and, hence, should be depicted as a member of the program store in the resulting model).

19 Otherwise, one could warn the user that multiple labels refer to the same state (not shown in Alg. 2).

Algorithm 3 Collect Initial Equivalence Classes

```
48: struct Atom(val, type, proto, state ← None, isLocal ← true,
                    hash ← λ() · [val, type].join())      ▷ Hash by value/type
49: struct EquivClass(val, type, aliases,
                    hash ← λ() · [val, type].join())      ▷ Hash by value/type
50: struct GraphNode(val, aliases, fields ← []) <: EquivClass(type ← "Ref")
51: struct Graph(val, aliases, nodes ←{} ) <: EquivClass(type ← "Set[Ref]")
52: procedure CollectInitialEquivClasses(pds, model, states)
                ..... Phase A: Collect constants .....
53:     immut ← model.collect(λe: ConstEntry · p ← e.name
54:         p ∈ pds ∧ pds[p].type = "Argument" ⇒ Atom(e.val, pds[p].type, p))
        ..... Phase B: Collect mutable local store variables .....
55:     labelsToStates ←{}                                ▷ Maps labels to states
56:     states.foreach(λstate · state.labels.foreach(λlabel ·
57:         labelsToStates[label] ← state))
58:     mut ←{}                  ▷ Maps (variable name/state hash) pairs to atoms
59:     model.foreach(λentry · entry.name.startswith("@local") ⇒
60:         prefix, var, label ← entry.name.split("$")      ▷ E.g. "@local$x$l7"
61:         state ← labelsToStates[label]
62:         type ← pds[var].type
63:         key ← (var, state.hash())
64:         key ∉ mut ⇒ mut[key] ← Atom(entry.val, type, var, Some(state)))
        ..... Phase C: Group atoms into equivalence classes .....
65:     eqcls ←{}                ▷ Maps (value/type) pairs to equivalence classes
66:     immut.extend(mut.values()).foreach(λatom · key ← (atom.val, atom.type)
67:         key ∈ eqcls ? eqcls[key].aliases.extend(atom)
68:                     : eqcls[key] ← EquivClass(atom.val, atom.type, {atom} ))
69:     return eqcls.values()
```

STRUCTURE OF EQUIVALENCE CLASSES. Equivalence classes (line 49) have the following fields: `val` and `type` are the inner value and the type (e. g. `"Int"` or `"Ref"`) of the atoms forming this equivalence class; the field `aliases` is an (ordered) set of aliasing atoms.

The two extensions of `EquivClass` are *graph nodes*, i. e. `GraphNode` (line 50) and *graphs* i. e. `Graph` (line 51). Graph nodes model variables of type ***`Ref`***; since non-null references correspond to heap objects, graph nodes may have *fields* which are stored in `GraphNode.fields`. Graphs model variables of type ***`Set`***`[`***`Ref`***`]`, i. e. the type used for representing method footprints in our setting; these have an (ordered set) field `Graph.nodes`, indicating which nodes belong to a particular graph.

4.4.3.1 Immutable variables. The algorithm first collects *immutable variables*, i. e. those that must have the same value in all states, e. g. formal method arguments (lines 53 to 54). We start by collecting constant model entries `e`, storing their names into `p` (line 53). We then check that `e` has a corresponding program definition whose type is `"Argument"`, in which case we construct a new atom for each such entry (line 54).

4.4.3.2 Mutable variables. We collect mutable variables in two steps. First, we create and populate the map `labelsToStates` from label names (e. g. `"l1"`) to lists of states corresponding to those labels (lines 55 to 57). Recall from Sec. 4.4.2 that each label uniquely identifies a state, but our consolidated states may have multiple labels.

Second, we collect model entries whose name starts with the reserved prefix `"@local"`; these are special instrumentation constants in the source program that mark particular variable versions with state labels (lines 58 to 64). The full name of these constants is e. g. `"@local$x$l7"` where `x` is the name of a local variable and `l7` is the state label in which `x` is marked; these values are stored in `var` and `label`, resp. (line 60).

We proceed by retrieving the state corresponding to `label` from the `labelsToStates` map; we obtain the state in which the current atom is active in the counterexample (line 61). Next, we obtain the variable's type from the program definitions `pds` (line 62). We then form a key of two factors: the variable name and the state hash[20] (line 63). Finally, we check that the key is not already present in the `mut` map, in which case we construct a new atom, storing all information about this variable version, and saving it into `mut` under `key` (line 64).

AVOIDING DUPLICATE ATOMS. Two distinct program states may have identical local stores if they differ only in their heaps, e. g. in the following scenario (in which we assume that `t` and `x` are two nodes in the current method's footprint, $\mathfrak{g}$):

```
t := null;   THIS_STATE_IS__l1(t)  // yields (@heap(g) = l1() ∧ t = @local$t$l1())
x.next := t; THIS_STATE_IS__l2(t)  // yields (@heap(g) = l2() ∧ t = @local$t$l2())
```

On the one hand, the two assignments result in states labeled `l1` and `l2`, resp. Assume that the model includes the entry `null` ↦`v11`. The local store hash is then *the same* in both `l1` and `l2`, namely, the string `"t=v11"` (cf. line 20).

On the other hand, our instrumentation yields two distinct model entries `"@local$t$l1"` and `"@local$t$l2"` that define the local variable snapshots of `t` in `l1` and `l2`; both of these entries will be processed on line 59. Therefore, our two-factor key (cf. line 63) depends on both the name of a local variable and the local store hash in the state in which this variable is labeled, avoiding the creation of duplicate atoms.

4.4.3.3 Forming equivalence classes. The last phase of CollectInitialEquivClasses iterates through all the atoms (both immutable and mutable), consolidating them by their

20 We explain why this particular key structure is used shortly after.

Algorithm 4 Transitive heap analysis

```
70: struct Data(eqcls ← {}, gs ← {}, ns ← {}, cs ← {}, fs ← [])

71: procedure Saturate(pds, model, states, eqcls)
72:     d ← Data()
73:     repeat
74:         d.eqcls.extend(eqcls)
75:         nodes ← GroupEquivClassesByType(eqcls, d)   ▷ Extends d.gs, d.ns, d.cs
76:         eqcls ← ExtractFields(pds, model, states, nodes, d)   ▷ Extends d.fs
77:     until eqcls = []
78:     return d.gs, d.ns, d.cs, d.fs
```

inner value and type (lines 65 to 68). First, we declare the map `eqcls` from (inner value/-type) pairs to equivalent classes (line 65). We iterate through all the atoms; for each `atom`, we create a two-factor key with its inner value and type and store it in `key` (line 66).

We then check if this key is already present in the `eqcls` map, in which case we add the current atom to the set of aliases of the existing equivalence class (line 67). Otherwise, we create a new equivalence class with this atom's inner value and type, and initialize its aliases set with a singleton set of the current atom; we save the new equivalence class into `eqcls` under `key` (line 68).

The procedure CollectInitialEquivClasses returns `eqcls.values()`, i. e. the set of all equivalence classes (modeling both immutable and mutable variables) (line 69).

4.4.4 Transitive heap analysis

After collecting the relevant program states (Sec. 4.4.2) and the initial equivalence classes (Sec. 4.4.3), the algorithm proceeds with its main step of transitive heap analysis. This step is implemented via the procedure Saturate (Alg. 4).

The procedure takes as input the program definitions (`pds`), the raw SMT model (`model`), and the relevant program states and starting equivalence classes (resp. `states`, `eqcls`). Next, we discuss the four collections that are returned at the end of Saturate. Graphs (`d.gs`) are *arbitrary* node sets (including footprints) i. e. all variables of type ***Set*[*Ref*]**. Correspondingly, graph nodes (`d.ns`) are (non-null) references, i. e. those referring to actual heap objects, while *scalar atoms* (`d.cs`) contain variables of all other types.

Finally, Saturate returns the *heap adjacency relations* (`d.fs`). Each relation, i. e. an instance of type `Relation` (line 94) has a name (corresponding to some ***Ref***-field's name), a state in which it is active, as well as this relation's *predecessor* and *successor*. For example, if the source program has the line `x.next := y` for some ***Ref***-field `next` and two references `x` and `y`, then we will model the corresponding relation as `Relation("next", s, x, y)`, where `s` models the state after this assignment. Note that we will not need `Relation`'s method *`hash`* until Sec. 4.5.

Algorithm 5 Group Atoms by Type

```
79: procedure GroupEquivClassesByType(eqcls, data)
80:     nodes ← []
81:     eqcls.foreach(λe · match e.type
82:         case "Set[Ref]" ⇒ data.gs.extend(Graph(e))
83:         case "Ref" ⇒ nodes.extend(GraphNode(e))
84:         case _ ⇒ data.cs.extend(e))
85:     data.ns.extend(nodes)
86:     return nodes
```

The body of Saturate first initializes a singleton object, `d`, containing data across all iterations (line 72), then repeats the three steps (discussed below), until `atoms` becomes empty (lines 73 to 77).

saturation. We now discuss the body of the repeat-until loop of Saturate, which consists of three steps. First, the current equivalent classes are added to the global collection `d.eqcls` (line 74).[21] Then, GroupEquivClassesByType forms three groups: graphs, nodes, and all others, accumulating them in the corresponding fields of `d` (line 75). This procedure returns `nodes`, i. e. the subset of the new ***Ref***-type atoms (Sec. 4.4.4.1).

Third, ExtractFields extracts *new heap adjacency relations* by analyzing the field values of new graph nodes and accumulates them into `d.fs` (line 76). This procedure returns the set of fresh, *transitive* equivalence classes, i. e. those modeling newly reached nodes (via one heap edge) that were not already processed on previous iterations (Sec. 4.4.4.2).

The loop repeats until there are no new transitive equivalence classes. We will next describe the saturation steps in more detail and then present a typical scenario of running Saturate and an informal analysis of this algorithm (Sec. 4.4.4.3).

4.4.4.1 Grouping equivalence classes by type. The procedure GroupEquivClassesByType is straightforward (Alg. 5). At a high-level, the procedure initializes a new list (line 80), `nodes`, then populates it with atoms whose type is ***Ref*** (lines 81 to 84), and then returns the list (line 86). In addition, we accumulate graphs, i. e. atoms with type ***Set*****[*****Ref*****]** (line 84) and scalar nodes (line 84) into the corresponding fields of `data`; graph nodes are accumulated into `data` as well (line 85). Note that each of the three cases (lines 82 to 84) upcasts the current atom, `a`, to the corresponding subtype.

4.4.4.2 Extracting field values. The last step of an iteration in Alg. 4 is the procedure ExtractFields that collects fields of the new nodes. This procedure uses the program definitions (`pds`), the model (`model`), the relevant program states (`states`), and the new graph nodes (`ns`); it then accumulates new *fields* into `data`; and then returns the list `trans`

21 We will explain why the two sets of equivalence classes are always disjoint in Sec. 4.4.4.2.

of *transitive*, i. e. newly reached, atoms. We model fields as relations, i. e. instances of type `Relation` (line 94), that specify a field's name, a program state, as well as the predecessor (which must be a graph node) and the successor (which may be any equivalence class).

ACTIVE RECEIVERS. The first step in ExtractFields is to obtain a mapping from each state to the set of *field receivers*, i. e. non-null nodes with active atoms in that state; we achieve this by involving the sub-procedure GetActiveObjects (line 97).

GetActiveObjects takes a set of nodes `nodes` as input and returns a function that maps each state to the set of nodes that are active in that state. We start by initializing the map `vObjs` from states to nodes and the set `cObjs` for collecting immutable objects (line 88).

We then iterate through (non-null) `nodes` (line 89).[22] For each node `n`, we then traverse its aliasing atoms `n.aliases`; and for each atom `a`, we pattern match on the type of `a.state` (line 90). If some state is associated with this atom, then we add its equivalence class, i. e. the node `n`, to the set of nodes in `vObjs` for the state `s` (line 91). Otherwise, this is an immutable object that is active in all states, and we add it to the set `cObjs` (line 92).

After grouping the input nodes between `vObjs` and `cObjs`, we return a λ-function of one argument, `state`, that yields exactly the objects that are active in `state` (line 93).

Algorithm 6 Map States to Active Objects

```
87: procedure GetActiveObjects(nodes)
88:     vObjs, cObjs ← {}, {}        ▷ v: maps states to active objs.; c: set of constant objs.
89:     nodes.foreach(λn · isNotNull(n) ⇒        ▷ Only non-null Refs are objects
90:         n.aliases.foreach(λa · match a.state        ▷ Check if atom has state
91:             case Some(s) ⇒ s ∈ vObjs ? vObjs[s].extend(n) : vObjs[s] ←{n}
92:             case None ⇒ cObjs.extend(n)))        ▷ No associated state ⇒ constant
93:     return λstate · state ∈ vObjs ? vObjs[state].union(cObjs) : cObjs
```

FIELD-RELATED MODEL ENTRIES. Returning to Alg. 7, we collect all field-related model entries that provide information about field values (lines 98 to 98). These are the entries that correspond to program definitions with type `"Field"`.[23]

SETTING UP THE HEAP ANALYSIS. We proceed by initializing the set for transitive nodes (line 99) and iterating through the states `state`; we then obtain the set `objs ← stateToNodes(states)` of all active nodes of this state (line 100). We proceed by iterating through these relevant nodes and all field-related entries (line 101).

For each (state/receiver object/field entry) triplet, we first perform a lookup into the field entry, passing as arguments the inner values of this state, object, and field, and stor-

22 The function `isNotNull(node)` can be implemented as `model["null"].val ≠ node.val`.

23 For simplicity of the presentation, we assume in Alg. 7 a particular organization of field-related entries in the model (i. e. as in Viper's VCG). Some aspects of ExtractFields require (simple) customization to support other verification backends, e. g. the condition `pd.name ∈ model` (line 98) and the signature of `fent.apply`.

ing the resulting value in `val` (line 102).[24] Then, we check that `val` is specified, i. e. the model assigns some field value (line 103). Recall that we use partial SMT models that typically leave some unspecified cases in function interpretations, e. g. because the solver could not find a suitable assignment or because this value is not needed for drawing the contradiction. If the value *is* specified in the model, we proceed.

HEAP ANALYSIS. The algorithm proceeds as follows. First, we obtain this field's type from the program definitions, `pds` (line 104). Second, we form a two-factor key based on `val` and `type` (line 105).

Third, we create a set of aliasing atoms, one for each alias of the field's receiver object, and save them into `aliases` (lines 106 to 109). For example, if `obj` has two aliases, say `a` and `b`, the current field's name is *next*, and `state` is labeled as `l1`, then the algorithm will create two new aliasing atoms, naming them `"a.next[l1]"` and `"b.next[l1]"` (cf. lines 107 to 108); these atoms will be active in `state` and marked as *non-local* as they are reached via a heap edge (cf. line 109).

Fourth, we consider whether to reuse an equivalence class created on *previous* iterations of SATURATE (lines 110 to 112) or earlier on *this* iteration, e. g. for another state or field (lines 113 to 115); in both cases, we extend the set of aliases of the existing equivalence class with the newly created atoms from `aliases`. Alternatively, if no equivalence classes have been created so far for these value and type, then we create a new equivalence class that represents our newly collected set of aliasing atoms; we save this new class into the `trans` map under `key` (lines 116 to 118).

Fifth, we use the obtained class `succ` that models the field value in order to construct a new relation, storing it in `field` (line 119); we then associate the this field with the receiver object (line 120) and accumulate it in `data.fs` (line 121).

The EXTRACTFIELDS procedure accumulates each new field in `data.fs` (line 121). After all fields are created, the procedure returns the list of transitive atoms (line 122).

4.4.4.3 Managing equivalence classes.

Recall that atoms represent variables and constants of our source program.[25] While generating a model, the SMT solver may choose whether to assign a fresh value to a particular atom or to reuse an existing one. Hence, some atoms may end up with the same inner value. Equivalence classes consolidate aliasing atoms, enabling us to reason about unique heap objects in a counterexample.

EXAMPLE. Assume we have a model of two non-null references `a` $\mapsto$`v1`, `b` $\mapsto$`v1`; this indicates that `a` = `b`, i. e. `a` and `b` reference the same object. Let the corresponding initial atoms be `a` $\leftarrow$ `Atom(val` $\leftarrow$ `"v1", type` $\leftarrow$ `"Ref", proto` $\leftarrow$ `"a", isLocal` $\leftarrow$ *`true`*`)` and `b` $\leftarrow$ `Atom(val` $\leftarrow$ `"v1", type` $\leftarrow$ `"Ref", proto` $\leftarrow$ `"b", isLocal` $\leftarrow$ *`true`*`)`. These two local

24 Viper's SE uses separate functions for each field's values; an adaptation of EXTRACTFIELDS is straightforward.
25 We use the terms *atom* and *constant model entry* interchangeably as they are isomorphic.

Algorithm 7 Extract Field Values

```
94:  struct Relation(name, state, pred, succ, hash ← λ() ·  ▷ Hash all but the name
95:              [state.labels.join(), pred.hash(), succ.hash()].join())
96:  procedure ExtractFields(pds, model, states, nodes, data)
97:      stateToNodes ← GetActiveObjects(nodes)
98:      fents ← pds.collect(λpd ·                    ▷ Collect field-related model entries
           pd.type = "Field" ∧ pd.name ∈ model ⇒ model[pd.name])
99:      trans ← {}                ▷ Maps (value/type) pairs to new transitive equiv. classes
100:     states.foreach( λstate · objs ← stateToNodes(state)
101:         objs.foreach(λobj · fents.foreach(λfent ·
102:             val ← fent.apply([state.val, obj.val, fent.val])  ▷ Model lookup
103:             val ≠ "unspecified"  ⇒                ▷ Skip unspecified field values
104:                 type ← pds[fent.name].type                ▷ Get this field's type
105:                 key ← [val, type].join()
106:                 aliases ← obj.aliases.map(λrec ·   ▷ E.g. if a = b then a.f = b.f
107:                     sid ← state.labels.join("/")          ▷ E.g. "l1" or "l2/l3"
108:                     name ← rec.name + "." + fent.name + "[" + sid + "]"
109:                     return Atom(val, type, name, Some(state), isLocal ← false))
110:                 if key ∈ data.eqcls then   ▷ Reuse eq.cl. from previous iterations
111:                     succ ← data.eqcls[key]
112:                     succ.aliases.extend(aliases)  ▷ Merge in new aliasing atoms
113:                 else if key ∈ trans              ▷ Reuse eq.cl. from this iteration
114:                     succ ← trans[key]
115:                     succ.aliases.extend(aliases)  ▷ Merge in new aliasing atoms
116:                 else                          ▷ Found eq.cl. with fresh value/type
117:                     succ ← EquivClass(val, type, aliases)      ▷ New new eq.cl.
118:                     trans.extend(succ)                         ▷ Save fresh eq.cl.
119:                 field ← Relation(fent.name, state, obj, succ)  ▷ New relation
120:                 obj.fields.extend(field)          ▷ Extend this receiver's fields
121:                 data.fs.extend(field))))                       ▷ Extend global fields
122:     return trans.values()              ▷ Return new transitive equiv. classes
```

atoms are merged in Phase C of CollectInitialEquivClasses into one equivalence class of the form `ab ← EquivClass(val ← "v1", type ← "Ref", aliases ←{a, b})`.

Further, assume that our method assigns to `a.next`. This may lead ExtractFields to discover a fresh atom (e. g., `Atom(val ← "v3", ...)`), modeling `a.next`'s value in some state (say, `s`). Since the algorithm maintains equivalent classes of atoms, it will consider `a` along with all of its potential aliases (in this case, `b`); the newly discovered field relation will thus be represented as `Relation("next", s, ab, cd)`. Here `cd` is the transitive equivalence class that models the atoms `a.next` and `b.next` which alias so long as $a = b$.

If the given inner value (`"v3"`) and type (`"Ref"`) were used on previous iterations of Saturate, then we would not have created `cd`, reusing the existing equivalence class from `data.eqcls` (lines 110 to 112). Analogously, if the given inner value and type were already used on *this* iteration of Saturate (e. g. for a different program state on line 100), then we would also not have created `cd`, this time, reusing the existing equivalence class from `trans` (lines 113 to 115). Only if neither of the two cases apply do we create a new equivalence class, using a fresh value and type combination, and adding all the aliasing atoms, i. e. `a.next` and `b.next` in the scenario above (lines 116 to 118).

In all three cases, the considered equivalence class can be *uniquely identified* via the inner value and type. Hence, the equivalence classes collected in `trans` and in `data.eqcls` are *always disjoint*; merging disjoint sets of equivalence classes is trivial (line 74).

TYPE INFORMATION. The type information from the SMT model can be imprecise; e. g. the model may assign the same inner value to atoms representing program values of incompatible types. Hence, we rely on the type information only from the verifier (via `pds`), incorporating it into the hash of atoms and equivalence classes (line 49). This guarantees that atoms with incompatible types are not merged by one equivalence class.

TERMINATION. Recall that for the procedure Saturate to terminate, ExtractFields must return an empty list, indicating that there are no fresh, transitively reachable equivalence classes to consider. This condition is eventually fulfilled because (1) The set of all possible equivalence classes for a given SMT model (say, EC), is *finite* as there are finitely-many (inner value/type) pairs; (2) The set of equivalence classes produced by ExtractFields (say, TR_0) is guaranteed to be *disjoint* with the set of equivalence classes collected in all previous iterations of Saturate (say, DT_0), as we informally argue above. We have that $(TR_0 \uplus DT_0) \subseteq EC$. If $TR_0 = \emptyset$, then the algorithm terminates. Otherwise, we proceed to the next iteration, replacing DT_0 with $DT_1 = (TR_0 \uplus DT_0)$ and repeating the argument for TR_1. Because $DT_0 \subset DT_1 \subset \ldots$ are all subsets of EC, we can see that the number of iterations must be finite; hence, Alg. 4 terminates.

Algorithm 8 Connect Graphs and Nodes

```
123: procedure ConnectGraphsAndNodes(model, graphs, nodes)
124:     footprints ← {}
125:     graphs.foreach(λgraph ·
126:         graph.name = "𝔤" ⇒ footprints["client"] ← graph
127:         graph.name = "𝔥" ⇒ footprints["callee"] ← graph
128:         nodes.foreach(λnode ·
129:             res ← model["Set.in"].apply([graph, node])
130:             res = "true" ⇒ graph.nodes.add(node)))
131:     return footprints
```

4.4.5 Connecting graphs and nodes

ConnectGraphsAndNodes (Alg. 8) is the final procedure of our model-generating algorithm of Alg. 1, providing the information about the heap node's membership (e. g. *does this node belong to the client's footprint?*). Recall that footprints are node sets with special names (each method specifies its footprint via the node set $\mathfrak{g}$ and its callee's footprint as $\mathfrak{h}$). We assume for simplicity that the method being debugged does not change its footprint (there is no allocation or deallocation of memory); generalizing our algorithm to handle multiple footprint versions is straightforward since the algorithm collects the information about all node sets.[26]

The procedure starts by declaring the `footprints` object (line 124). It then iterates through the graphs (line 125). For each graph, the procedure performs three steps: (1) Checking if this graph is the client's (line 126) or the callee's (line 127) footprint, in which case they are saved into the `footprint` object.(2) Iterating through the nodes (line 128), performing a lookup in the model's set-in entry[27] whether each node belongs to this graph (line 129). (3) If the interpretation of the set-in relation for this graph-node pair evaluates to `"true"`, then the procedure adds this node to this graph's `nodes` set (line 130). Finally, the procedure returns the `footprints` object (line 131).

4.4.6 Summary

We presented an algorithm that produces counterexample models to verification failures of heap-transforming programs. We assumed that the program is specified in separation logic; in particular, each method's footprint must be declared as a set of heap nodes ($\mathfrak{g}$), while the permissions to access the fields of these nodes should be specified via iterated separating conjunctions.[28] Correspondingly, before a client method can in-

26 However, it is hard to *visualize* superpositions of program states with non-constant footprints.
27 The exact name of the model entry for the set-in relation is verifier-specific, e. g. it is `"[2]"` in Viper's VCG.
28 In case method footprints are not available, our technique *gracefully degrades* to modeling heap nodes and edges without footprint affiliation by simply skipping ConnectGraphsAndNodes (Alg. 8).

voke a callee, it should declare the callee's footprint as a set of heap nodes (ℌ). In order for the counterexamples to refer to states in an understandable way, we require the programmer to instrument the program with state labels (Sec. 4.3).

The inputs of our algorithm are the raw SMT model obtained after a verification failure and the program definitions. While the SMT model contains (almost) all the information needed to produce a counterexample, filtering the relevant bits out of a large number of model entries requires a number of choices that make this process not trivial. For example, it is crucial to merge aliasing atoms (since aliasing is often the cause of verification failures); yet, it is not obvious at what stages to merge them into equivalence classes as *fresh* atoms may occur while traversing the heap's model. Our approach makes merging atoms trivial (Alg. 4, line 74) as we form explicit equivalence classes from the beginning and then update them as soon as a fresh atom is reached.

In addition to merging aliasing atoms, our algorithm overcomes the problem of aliasing *states*. Our state consolidation procedure (Alg. 2) merges states if and only if both the store and the heap are equivalent, automating an otherwise cumbersome (and error-prone) manual task.

4.5 HEAP REACHABILITY MODELS

Our general algorithm of Sec. 4.4 leverages partial SMT models and precise type information to produce counterexample heap models. These counterexamples contain the basic information needed for visualizing and understanding heap configurations, possibly in multiple states; in addition to the heap edges, we keep track of the *in-set* relation defining which nodes are inside the current method's footprint.

In this section, we will show that an important advantage of our algorithm is its *extensibility*, i. e. one can easily extend this algorithm to enrich the produced counterexamples with other useful information from the program specifications. Concretely, we will demonstrate this for the *local reachability* relation used in our modular heap reachability verification technique of Chap. 3.[29]

We implement or extended algorithm in Alg. 9.

4.5.1 Purpose of heap reachability models

The purpose of Alg. 9 is to augment counterexample heap models produced by the general algorithm with *local reachability information*, indicating the transitive heap paths that *exist* or *do not exist* (according to the SMT solver). This information is collected as a list of relations, i. e. instances of type `LocalRelation` (line 137), and returned alongside the components of the general model (line 136).

29 We assume that the reader is familiar with our local reachability notation described in Sec. 3.3.

Algorithm 9 Produce Reachability Model

```
132: procedure ProduceReachabilityModel(pds:{str ↦ ProgDef},   ▷ Program def-s
                                        model:{str ↦ Entry})   ▷ SMT model
133:     states, ftprints, graphs, nodes, fields ← ProduceHeapModel(pds, model)
134:     reach ← ExtractReachability(pds, model, graphs, nodes)
135:     reach ← Reduce(reach, fields, ftprints)
136:     return states, ftprints, graphs, nodes, fields, reach

137: struct LocalRelation(name, state, graph, pred, succ) <: Relation

138: procedure ExtractReachability(pds, model, states, graphs, nodes)
139:     pathEntry ← model["exists_path"]          ▷ Local reachability predicate
140:     snapEntry ← model["rsnap"]                ▷ Snapshot function
141:     stateToNodes ← GetActiveObjects(nodes)    ▷ Maps states to actve nodes
142:     stateToGraphs ← GetActiveObjects(graphs)  ▷ Maps states to actve graphs
143:     return states.flatmap( λstate · stateToGraphs(state).collect(λgraph ·
144:         rsnap ← snapEntry.apply([state.val, graph.val]) ▷ 1st lookup in model
145:         rsnap ≠ "unspecified" ⇒                ▷ Check if value is relevant
146:             stateToNodes(state).flatmap(λpred· stateToNodes(state).collect(λsucc·
147:                 reach ← pathEntry.apply([rsnap, pred, succ])   ▷ 2nd lookup
148:                 reach ≠ "unspecified" ⇒        ▷ Check if value is relevant
149:                     name ← (reach = "true") ? "P" : "¬P"
150:                     return LocalRelation(name, state, graph, pred, succ) ) ) ) )
```

LOCAL REACHABILITY. We recap on the semantics of local reachability relations that were defined in Chap. 3. The local reachability relation $\mathtt{P}^F_\sigma(\mathfrak{g}, x, y)$ holds *iff*, in the state σ, there exists a (possibly, zero-length) path $x \ldots y$ of fields from the set F, s.t. all nodes on this path are inside the footprint $\mathfrak{g}$ (except, possibly, the last node, which may or may not be in $\mathfrak{g}$). For example, $x.\mathtt{left.right.left} = y$ implies $\mathtt{P}^{\{\mathtt{left},\,\mathtt{right}\}}_\sigma(\mathfrak{g}, x, y)$.

In modular verification, local reachability relations serve two main purposes: (1) they provide an *abstract view* over concrete heap edges and (2) they specify *absence of reachability*, e. g. $\neg\mathtt{P}^{\{\mathtt{left},\,\mathtt{right}\}}_\sigma(\mathfrak{g}, y, x)$ expresses that, in the state σ, y cannot reach x by following `left` or `right` fields of nodes from $\mathfrak{g}$. Therefore, local reachability relations provide essential information for understanding complex heap configurations.

MOTIVATING EXAMPLE. To motivate the algorithm described in this section, we illustrate two versions of a counterexample heap model (Fig. 4.12). These models are constructed for a buggy method `reverse` of an acyclic list.[30] The first model corresponds to the output generated by our general algorithm of Alg. 1, showing only direct heap edges. In

30 We will discuss `reverse` in full detail in Sec. 4.7.1.

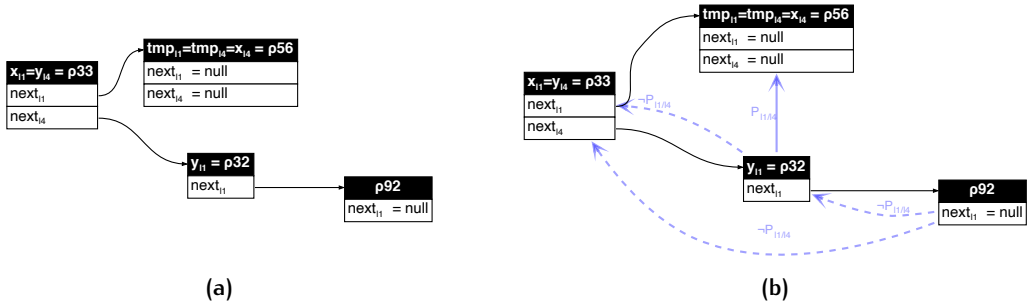


Figure 4.12: Heap model and its augmentation with local reachability information.

(a) A regular heap model. Concrete heap edges are depicted via black arrows. (b) Heap reachability model. Facts about local reachability are depicted via blue arrows; *solid* and *dashed* arrows denote *existing* and *non-existing* local heap paths, resp. (according to the solver).

contrast, the second model (generated by Alg. 9) additionally contains local reachability relations essential for fully understanding the heap configuration. In particular, the *un*reachability facts are needed to see that the structure is *acyclic*, which direct heap edges cannot express. Additionally, the heap reachability model of Fig. 4.12b shows that ρ32 reaches ρ56, which contradicts the intuition since the only heap edge originating in ρ32 leads to ρ92, which itself has no outgoing edges. While spurious relations are a typical artifact of an incomplete SMT model, visualizing them helps the programmer understanding what facts the solver can or cannot establish.

4.5.2 Overview of the extended algorithm

The entry point of Alg. 9 is ProduceReachabilityModel. This procedure takes the same inputs (the program definitions `pds` and the raw SMT model `model`) as our general algorithm of Alg. 1, which is invoked at the very first step (line 133). Refer to Sec. 4.4.1 for an overview of the input structures.

We collect relevant reachability information in two steps. First, we directly obtain all the reachability relations from the raw SMT model (line 134; details in Sec. 4.5.3). Second, we reduce the relations to an intelligible set (line 135; details in Sec. 4.5.4).

4.5.3 Extracting reachability relations

We implement this step via ExtractReachability. This procedure takes the program definitions (`pds`), the SMT model (`model`), the program states (`states`), the graphs (`graphs`), and the nodes (`nodes`) and returns a list of local relations. Each local relation (instance of `LocalRelation`) contains a name, a state, a *graph* (defining its *locality*), as well as a predecessor and a successor node.

For collecting local reachability information from the SMT model, we will use only two relation names: "P" and "¬P". The prior relation corresponds to *existing* heap paths, while the latter relation corresponds to *non-existing* heap paths. For example, recall that if a subgraph of a graph g is an acyclic list segment with x as its head and y as its last node, then P(g, x, y) and ¬P(g, y, x) must hold in our technique.

We collect local reachability relations as follows. First, we obtain the reachability-defining model entries, i. e. the interpretations of the functions named "exists_path" and "rsnap" (lines 139 to 140). Recall from Sec. 3.3.4 that the former function is uninterpreted and state-independent,[31] while the latter is the (state-dependent) *snapshot* function. In our local reachability encoding, the snapshot function takes the current footprint and yields an *edge set*, i. e. a mathematical representation of the heap graph in the current state (in the SMT translation, the snapshot function explicitly depends on a heap marker). This edge set can be used as the first argument of exists_path; the other two arguments are the predecessor and the successor nodes.

Second, we obtain two mapping functions: stateToNodes maps each state to the *graph nodes* that are active in that state; stateToGraphs maps each state to the active *graphs* (lines 141 to 142). Active objects are those that have some aliases relevant to a particular state; e. g. atoms modeling immutable variables are active in all states, but atoms modeling local variables are *not* active in the states in which the corresponding variables have different values. Sec. 4.4.4.2 discusses active nodes in more detail and outlines GetActiveObjects (Alg. 6).

Third, we iterate through each program state (state) and each graph (graph) that is active in that state (line 143). Using these two parameters, we perform a lookup into snapEntry, storing the result in rsnap (line 144). If rsnap is specified, we proceed; otherwise, the model does not have reachability information for this state and graph (line 145).

Next, since rsnap *is* a specified snapshot, we proceed by iterating through all pairs of nodes (pred, succ) that are active in the current state (line 146). We use rsnap, pred, and succ to perform the second lookup into the model, this time into pathEntry, storing the result in reach (line 147). If reach is specified, we proceed; otherwise, the model does not provide the information about the value of $\mathtt{P}_{\mathtt{state}}(\mathtt{graph}, \mathtt{pred}, \mathtt{succ})$ (line 148).

If reach *is* specified, then it is a Boolean literal that determines our relation's name (line 149). Finally, we create a new local relation instance with this name, storing the current state, graph, and the predecessor and successor nodes (line 150). We collect all such relations into a list that is then returned as the result of ExtractReachability.

4.5.4 Reducing reachability relations

Our procedure ExtractReachability provides us with an over-approximation of the reachability relations that are relevant for understanding a counterexample heap config-

31 We used the name $\hat{P}$ for exists_path in Chap. 3.

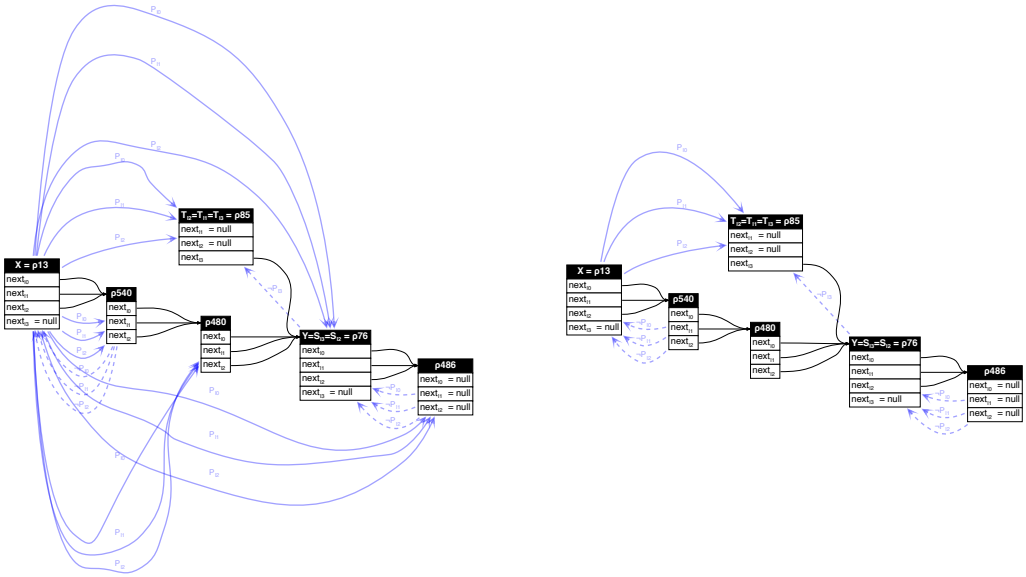


Figure 4.13: Comparison of unfiltered (a) and filtered (b) heap reachability models.

The two heap models represent the same counterexample obtained for the method **`union`** of the Union-Find data structure. Solid blue edges represent local heap reachability and dashed blue edges represent local *un*reachability relations (based on the SMT model's information).

uration. We will now discuss how these relations can be *reduced* to simplify the overall counterexample models. To motivate this step, consider a heap reachability model in two versions: before and after applying our reduction steps (Fig. 4.13); we will present other heap reachability models in Sec. 4.7.

REDUCTION OVERVIEW. We implement the reduction step via REDUCE (Alg. 10). This procedure takes the initial reachability relations `reach`, the field relations `fields`, and the current footprints (of the client and the callee) stored in `footprints`. The relevant subset of the original reachability relations is returned.

We reduce the set of reachability relations as follows. First, we filter out the *spurious* reachability relations (Sec. 4.5.4.1). Second, we collect the heap edge relations and compute their transitive closure, which we then use to synchronize our reachability relations (Sec. 4.5.4.2). Finally, we supplant *weak reachability* information by *strong reachability* information that is available (Sec. 4.5.4.3).

4.5.4.1 Filtering out spurious reachability. Spurious reachability relations are those that originate in nodes outside of the local graph. Recall that if $x \dots y$ is a local path in a graph $\mathfrak{g}$, then $x \in \mathfrak{g}$, but y may or may not be in $\mathfrak{g}$ (Fig. 3.16). Hence, if $x \notin \mathfrak{g}$, then $x \dots y$ cannot

Algorithm 10 Reduce Reachability Relations

```
151: procedure REDUCE(reach, fields, footprints)
        ..... Phase A: Drop spurious reachability relations .....
152:    reach ← reach.filter(λrel · rel.pred ∈ rel.graph.nodes)
        ..... Phase B: Synchronize reachability with transitive closure .....
153:    edges ← fields.filter(λf · f.type = "Ref" ∧ isNotNull(f.succ))
        ▷ Compute transitive closure over heap edges per state
154:    tc ← TransClosInStates(edges)
        ▷ Synchronize local reachability with the TC relation
155:    reach ← reach.filter(λrel ·
156:       rel.state ∉ tc ⇒ true                          ▷ No trans. rel-s for this state
157:       rel.hash() ∉ tc[rel.state] ⇒ true              ▷ No trans. rel. for pred/succ
158:       rel.name = "P" ⇒ false                         ▷ Path follows heap edges; redundant
159:       rel.name = "¬P" ⇒ true)                        ▷ Desynchronization
        ..... Phase C: Drop reachability framing-relation redundancies .....
160:    clientPos, clientNeg, calleePos, calleeNeg ← {},{},{},{}   ▷ hash to rel
161:    reach.foreach(λrel ·  ▷ Group (pos-ve, nag-ve) client- and callee-local reach.
162:       rel.graph = footprints.client ⇒
163:          rel.name = "P" ⇒ clientPos[rel.hash()] ← rel
164:          rel.name = "¬P" ⇒ clientNeg[rel.hash()] ← rel
165:       rel.graph = footprints.callee ⇒
166:          rel.name = "P" ⇒ calleePos[rel.hash()] ← rel
167:          rel.name = "¬P" ⇒ calleeNeg[rel.hash()] ← rel)
168:    final ← []                                        ▷ Stores final reach. relations
169:    final.extend(clientNeg)                           ▷ Keep strong reachability information
170:    final.extend(calleePos)                           ▷ Filter weak reachability information
171:    clientPos.foreach(λgPos·gPos.hash() ∉ calleePos ⇒ final.extend(gPos))
172:    calleeNeg.foreach(λhNeg·hNeg.hash() ∉ clientNeg ⇒ final.extend(hNeg))
173:    return final
```

exist as a local path in $\mathtt{g}$, and $\mathtt{P}(\mathtt{g}, x, y)$ is then spurious. The outline of this filtering step is straightforward (line 152).

Note that the SMT model may contain such spurious relations because the solver cannot exhaustively instantiate all the quantified constraints, and the models that it generates are approximate. Generally, understanding which facts can the solver establish is essential for debugging spurious verification failures that are caused by the solver's incompleteness. However, establishing the above property always requires exactly one quantifier instantiation (i. e. the postcondition of `rsnap`; see Sec. 3.3.5); *lacking* this property indicates that the corresponding heap path is irrelevant for deriving a contradiction.

4.5.4.2 Synchronizing reachability with transitive closure. Our goal is to decide whether a particular reachability relation is helpful for understanding the verification failure. We

will now explain and illustrate the reasons why we keep or drop certain reachability relations, and then present our filtering procedure.

SYNCHRONIZATION. Information about *existing* paths is redundant if it follows from the direct heap edges in the model. For example, if the model has three direct heap edges (a, b), (b, c), (c, d), where a, b, c are all different nodes of the current footprint, say $\mathtt{g}$, then the hypothetical local reachability predicate $\mathtt{P}(\mathtt{g}, a, d)$ is redundant whereas $\mathtt{P}(\mathtt{g}, d, a)$ is *not* redundant.

DESYNCHRONIZATION. Since our local reachability verification technique is inherently incomplete, some facts about reachability may be *out of sync* with the transitive closure of the direct heap edge relations. For example, the local *un*reachability predicate $\neg\mathtt{P}(\mathtt{g}, a, d)$ contradicts the direct heap edges (a, b), (b, c), (c, d). However, this unreachability relation is *not redundant* because the desynchronization of reachability and direct heap information may be crucial for understanding the verification failure.[32]

Desynchronization can also occur for *positive* local reachability relations, e. g. $\mathtt{P}(\mathtt{g}, \mathtt{m}, \mathtt{n})$, where (1) $\mathtt{n} \neq \mathtt{m}$, (2) the model contains the information that $\mathtt{n.next} = \mathbf{\mathtt{null}}$, and (3) there are no other reference fields except `next`. Again, we keep this relation as it demonstrates an important incompleteness and is therefore *not redundant*.[33]

FILTERING PROCEDURE OVERVIEW. Based on the above insights, we will now develop a systematic procedure for filtering local reachability relations based on transitive closure of direct heap edges. First, we have to compute this transitive closure. To that end, we collect all heap edges, i. e. reference field relations with non-null successors, saving them into `edges` (line 153). We then delegate the computation of the transitive closure of `edges` to a sub-procedure called TransClosInStates (Alg. 11), obtaining the map `tc` from states to sets of *transitive relations* (line 154). These transitive relations are instances of type `Relation` (not `LocalRelation`) named `"TC"`.

COMPUTING STATE-AWARE TRANSITIVE CLOSURE. We briefly discuss the outline of TransClosInStates (Alg. 11). This procedure starts by declaring a map `tc` from states to (hashed) relations (line 175). Recall that the relation hash depends on its state, predecessor, and successor, but does not depend on the relation name (line 94). Hence, `tc` is well-suited for looking up matches by other relations, such as our reachability relations, which is important for Alg. 10.

To populate `tc`, we first create an auxiliary map `edgesInStates` from states to edge relations (lines 176 to 179) and then iterate through each (value/key) pair of this map: `sedges`, `state` (line 180). We then compute the mathematical transitive closure of `sedges`

32 Transitively expanding reachability information (e. g. concluding $\mathtt{P}(\mathtt{g}, x_0, x_N)$ from $\mathtt{P}(\mathtt{g}, x_0, x_1) \wedge \mathtt{P}(\mathtt{g}, x_1, x_2) \wedge \ldots \wedge \mathtt{P}(\mathtt{g}, x_{N-1}, x_N)$) is as hard of a problem as reasoning with general transitive closures.

33 In practice, most incompletenesses of this kind are caused by inadequate triggering patterns in the program's *specifications*, i. e. it is likely that the programmer has control over mitigating the issue.

Algorithm 11 Compute Transitive Closure per Program State

```
174: procedure TransClosInStates(edges)
175:     tc ← {}                                          ▷ Maps states to hashes to relations
         ..... Phase A .....
176:     edgesInState ← {}                                ▷ Maps states to relations
177:     edges.foreach(λedge·                             ▷ Hash heap edges by state
178:         edge.state ∈ edgesInState ? edgesInState[edge.state] ← [edge]
179:                                   : edgesInState[edge.state].extend(edge))
         ..... Phase B .....
180:     edgesInState.foreach(λ(sedges, state) ·           ▷ Process each state's edges
181:         tcInState ← TC(sedges)
182:         relHashMap ← {}                              ▷ Maps relation hashes to relations
183:         tcInState.foreach(λrel · relHashMap[rel.hash()] ← rel)
184:         tc[state] ← relHashMap)
185:     return tc
```

using an off-the-shelf procedure TC (Alg. 12); we save the resulting list of transitive relations into `tcInState` (line 181). We then hash the obtained relations by all their fields except for the names, simplifying future lookups (lines 182 to 183). Finally, we save `relHashMap` in `tc` under `state` (line 184).

After computing transitive closures for all states, TransClosInStates returns `tc`.

FILTERING CRITERION. We proceed by filtering the reachability relations (line 155). For each local reachability relation `rel`, we first check that its state is missing in `tc`, in which case we keep this local reachability relation as it is definitely not redundant (line 156). If `tc` *does* have some relations under `rel.state`, then we check that none of them match `rel`, in which case we also keep this local reachability relation as non-redundant (line 157).

If there *is* a match between `rel` and some relation from the transitive closure, then this relation is either *redundant* (as it merely follows direct heap edges) or it is *desynchronized*, e. g. if a negative local reachability predicate of the form $\neg\mathtt{P}(\mathfrak{g}, x, y)$ matched a relation from the transitive closure of the form $x \dots y$.[34] As explained above, we decide to drop `rel` in the former case and keep it in the later case (lines 158 to 159).

4.5.4.3 Supplanting weak reachability by strong reachability We will now define another tactic for filtering redundant local reachability relations from the model. To that end, recall that local reachability predicates are characterized by their *local footprints* for which they are defined (e. g. $\mathfrak{g}$ in $\mathtt{P}(\mathfrak{g}, x, y)$ for the $\mathfrak{g}$-local path $x \dots y$). The idea is to supplant relations modeling *weak* local reachability predicates by their *strong* analogues (if the latter are available) without losing any information.

34 Unlike local reachability, the transitive closure always carries positive reachability information, i. e. facts about the *existence* but never about the *absence* of heap paths.

Algorithm 12 Transitive Closure

```
186: procedure TC(edges)
187:     clos ← {}                                          ▷ Maps hashes to relations
                                  Phase A
188:     edges.foreach(λedge ·
189:         root ← Relation("TC", edge.state, edge.pred, edge.succ)
190:         clos[root.hash()] ← root)
191:     while true do
                                  Phase B
192:         newRels ← {}                                   ▷ Maps hashes to relations
193:         clos.foreach(λa · clos.foreach(λb ·            ▷ Construct transitive relations
194:             a.succ = b.pred ⇒                          ▷ Assert a.state = b.state
195:                 trans ← Relation("TC", a.state, a.pred, b.succ)
196:                 newRels[trans.hash()] ← trans))
                                  Phase C
197:         foundNewRels ← false                           ▷ Indicates saturation
198:         newRels.foreach(λ(rel, rhash) ·  ▷ Merge fresh relations into the closure
199:             rhash ∉ clos ⇒ foundNewRels ← true
200:                            clos[rhash] ← rel)
201:         if ¬foundNewRels then return clos
202:     end
```

Definition 2 (Relatively Weak and Relatively Strong Reachability)**.** *In a given program state, for a given set of reference fields F, and for two footprints $\mathfrak{g}$ and $\mathfrak{h} \subset \mathfrak{g}$, the predicates $\mathsf{P}^F(\mathfrak{g}, x, y)$ and $\neg\mathsf{P}^F(\mathfrak{h}, x, y)$ are called* relatively weak reachability predicates *and the predicates $\neg\mathsf{P}^F(\mathfrak{g}, x, y)$ and $\mathsf{P}^F(\mathfrak{h}, x, y)$ are called* relatively strong reachability predicates *iff $x \in \mathfrak{h}$ and $y \neq$* `null`.

Intuitively, the knowledge that a path exists in $\mathfrak{h}$ is *stronger* that the knowledge of the same fact but in the context of $\mathfrak{g}$, which has *at least* all the edges of $\mathfrak{h}$. Analogously, the knowledge that $\mathfrak{g}$ *does not* contain a some path, say, $x \ldots y$, is *stronger* than the knowledge that the same path does not exist in its subgraph $\mathfrak{h}$.[35]

FILTERING. The outline of this tactic consists of two steps (lines 160 to 172).

First, we group all the available local reachability relations into four sets: the *client-local* (positive and negative) reachability relations (`clientPos`, `clientNeg`) and the *callee-local* (positive and negative) reachability relations (`calleePos`, `calleeNeg`); these sets are realized as maps from relation hashes to relations (lines 160 to 167).

Second, we collect the relevant relations into the `final` list (line 168). Strong reachability relations, i. e. those modeling the client-local *negative* and the callee-local *posi-*

35 The notions of relatively weak and relatively strong reachability are aligned with the problem of reachability framing that we discussed in Sec. 3.5.

tive predicates, are added completely (lines 169 to 170). Conversely, weak reachability relations are added to `final` only if there is no *strong* counterpart, i. e. we add *positive* client-local relations that do not match any *positive* callee-local ones; and we add *negative* callee-local relations that do not match any *negative* client-local ones (lines 171 to 172).[36]

Finally, REDUCE returns `final`, i. e. the list of filtered local reachability relations (line 173).

4.6 IMPLEMENTATION

We have implemented algorithm Alg. 9 and applied it in Viper IDE's verification debugger called *Lizard*.[37] Lizard is written in TypeScript, closely following the listings of Sec. 4.4 and Sec. 4.5.

Lizard is a Visual Studio Code extension that enhances the functionality of Viper IDE with an intuitive verification debugging experience. Beyond the core algorithm, the extension features (1) a renderer that translates heap reachability models into dot diagrams [16] and (2) a WebView-based debugger panel that displays these diagrams and offers some control over the rendering process. Fig. 4.15 is a screenshot from a typical verification debugging session using Viper IDE with Lizard. Fig. 4.14 shows a high-level overview of the extension. The complete source code of the algorithm and the debugger extension is publicly available [125].

TYPICAL SCENARIO. When verification fails for a source program, the debugger panel opens automatically, displaying a heap model that represents a counterexample to the (last) verification failure. The user can then select any other verification failure from the menu (although only one failure at a time). The displayed counterexample heap model depicts a superposition of all selected states; this is a scalable graphic, allowing the user to zoom in, zoom out, or scroll without any restrictions.

The debugger panel is interactive, allowing the user to select arbitrary *subsets* of the source program's instrumented states; selecting a new subset triggers the debugger to render a new diagram of the corresponding *projection*, i. e. a model that contains only information that is meaningful for the selected states.

INTERMEDIATE MODELS. For IDE developers, the debugger panel can be expanded to show three intermediate representations of the counterexample model: (1) the raw model produced by the SMT solver, (2) the heap reachability model produced by Alg. 9, and (3) the dot diagram description produced by the renderer. All of these are represented as in interactively-expandable JSON tree. The entire debugger panel supports the common text search feature that simplifies navigation through large models.

36 Our relations are matched via a three-factor hash over their states, predecessors, and successors (line 94).

37 Although the motivation behind the name is primarily to avoid confusion with previous Viper debugger projects, it does follow Viper ecosystem's reptile-based naming convention.

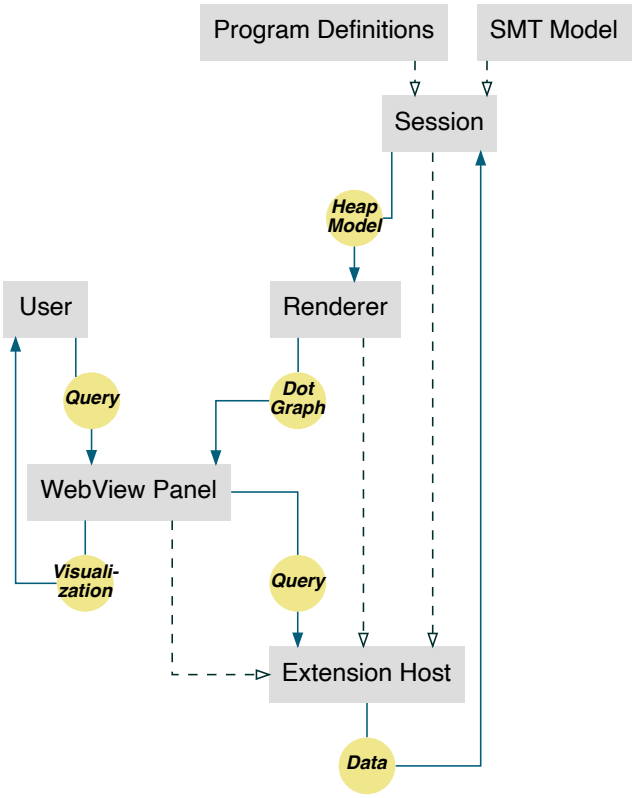


Figure 4.14: Implementation overview.

The diagram shows the main components (grey boxes) in our verification debugger implementation. Dashed arrows are aggregation relations; solid arrows with khaki labels indicate data flow. *Extension host* is the entry point; this is a VS Code extension that receives messages of two kinds from Viper IDE (program definitions, verification failures). If a failure message contains a counterexample SMT model, the extension host creates a *session* instance, initializing it with the current *program definitions* and *SMT model*. The session applies Alg. 9, producing a *heap model*. The heap model is then passed to the *renderer* that returns a *dot graph* that is then visualized by the *WebView panel* and displayed to the user. The WebView panel is interactive, e. g. the user can submit a *query* to project the model over selected states. The query is then processed by the extension host.

OPTIMAL LAYOUTS. The diversity of possible heap configurations that occurs in verification debugging makes it impossible to find a perfect layout that is optimal in all cases. The debugger panel provides only two buttons for adjusting the layout, so the user can switch between Graphviz's *rankdir* options, i. e. top-down vs. left-right layouts, as well as between our two heap node notations. We developed two intuitive visual notations with complementary advantages.

In the first notation (used in this chapter), each object is depicted as a *table*, with the head row specifying the (symbolic) memory location and the possible variable names. Other rows correspond to the fields. Reference fields are black arrows originating *from the corresponding row* in the receiver's table and pointing to the *header row* of the target's table. The field arrows thus do not require labels to disambiguate which field name, state, and receiver they belong to.

In the second notation, each object is depicted as a box specifying the (symbolic) memory location and the possible names of variables that refer to this location. Since fields are not depicted as table rows, heap edges are instead *labeled* black arrows, i. e. ordinary edges in a directed multigraph. Labels carry information about field names and states. This layout requires significantly less rendering per node, leaving more space for depicting relations. Hence, this layout better suits dense heap reachability models.

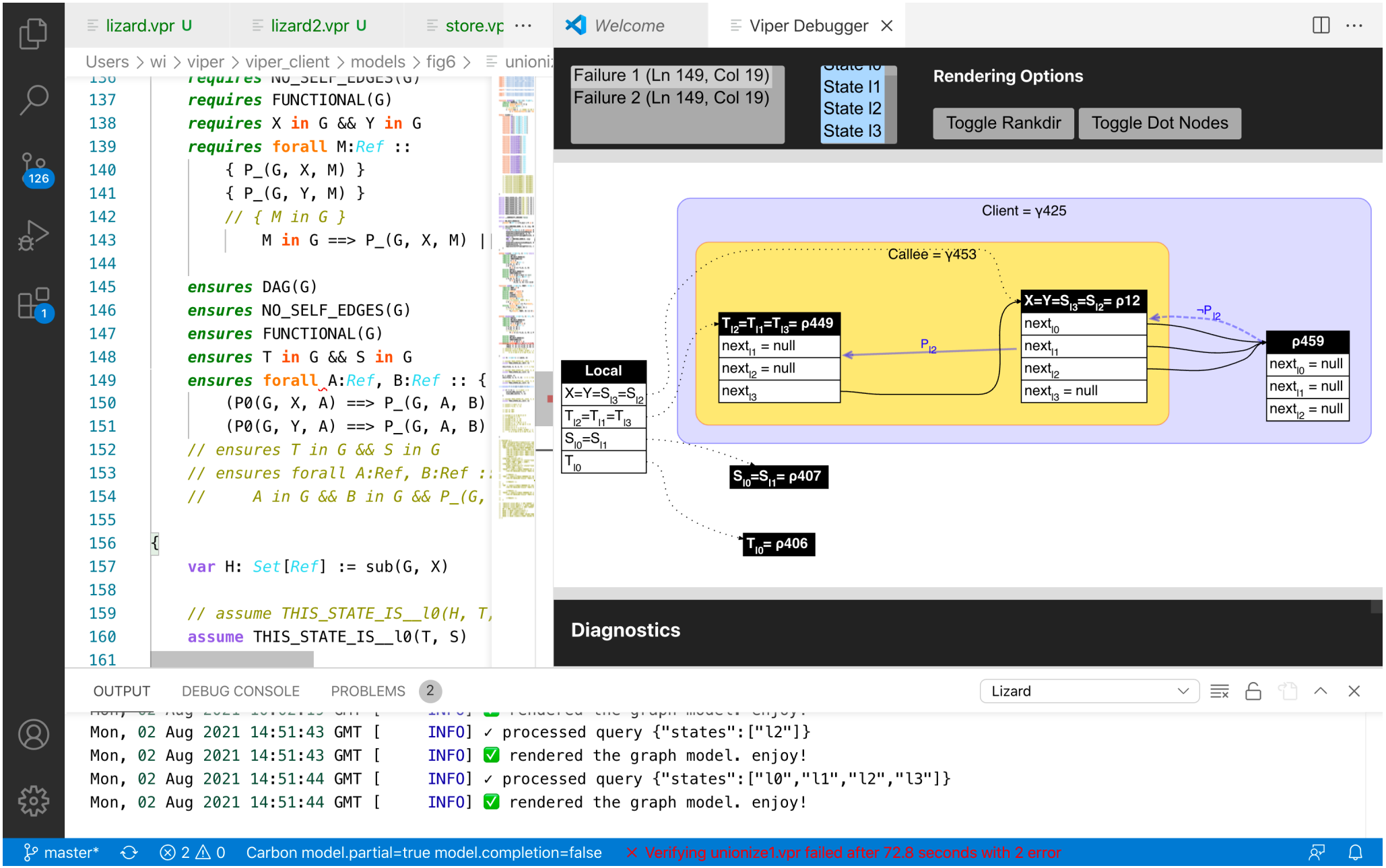


Figure 4.15: Screenshot of a verification debugging session in Viper IDE with Lizard.

4.7 CASE STUDY

In this section, we present a study of our algorithm's applicability in two scenarios. In the first scenario, the programmer specifies and verifies an in-place list reversing algorithm implementation, and the debugger is used interactively for *understanding the verification failures* caused by incomplete specifications (Sec. 4.7.1). In the second scenario, the programmer is given a correct implementation of the method `find` from the Union-Find (disjoint set) data structure [9], and the debugger is used for *inspecting* the possible heap configurations (Sec. 4.7.2).

4.7.1 Specifying and verifying List-Reverse

In this scenario, the programmer attempts to specify and verify local reachability properties of the method `reverse` that reverses a singly-linked list in place (Fig. 4.16). While this is a classical programming problem that most programmers can solve easily, correctly writing its formal specifications requires deep understanding of the algorithm and the consideration of its corner cases. Typically, devising the final version of such specifications takes several verification attempts. Therefore, we consider a scenario in which the programmer refines `reverse`'s loop invariant in three steps.

OVERVIEW. The method `reverse` takes the footprint `g`, consisting of all the nodes of the list, and `xe`, referring to the head of the list. The method returns `y`, referring to the head of the *reversed* list. The precondition of `reverse` requires that `g` is a directed acyclic graph (*functional*, since there is only one reference field per node, `next`).The last precondition specifies that `xe` is indeed the head of the list, i. e. all other nodes in `g` are reachable from it. The postcondition of `reverse` ensures that the resulting heap configuration is also a DAG, and that the returned variable `y` indeed refers to the new list head.

4.7.1.1 Initial verification attempt. Assume that the programmer has developed some intuition for the algorithm and proceeds with sketching its specifications, which involve devising a suitable loop invariant. The very first loop iteration starts by processing the entire list initially headed in `xe` (which is also the initial value of `x`); in this state, `y` is initialized with **`null`**. This situation is summarizes by the two bits of the loop invariant marked `/*A*/` (Fig. 4.16). Analogously, the very last iteration ends with the entire (reversed) list headed in `y`, in which state `x` becomes **`null`**, as there are no more nodes to process. The corresponding two bits are marked `/*B*/`.

Next, the programmer specifies the *intermediate* heap configuration that occurs at all other iterations of the loop. Namely, we have *two* list segments: one segment (headed in `y`) consists of the nodes that are already processed, and the other segment (headed in `x`) consists of the nodes that are yet to be processed. Hence, the programmer writes the condition marked `/*C*/`, specifying each node in `g` as reachable from `x` or from `y`.

```
method reverse(g: Set[Ref], xe: Ref) returns (y: Ref)
  requires DAG(g) && xe ∈ g
           ∀n ∈ g • n ∈ g ⟹ P(g,xe,n)
  ensures  DAG(g)
           ∀n ∈ g • n ∈ g ⟹ P(g,y,n)
{
  var x: Ref := xe; y := null;  assume THIS_STATE_IS__l0(x, y)
  while (x ≠ null) invariant DAG(G)
  /*A */ x ≠ null ⟹ x ∈ g
         y = null ⟹ ∀n ∈ g • P(g,x,n)
  /*B */ y ≠ null ⟹ y ∈ g
         x = null ⟹ ∀n ∈ g • P(g,y,n)            /* R2 */
  /*C */ x ≠ null ∧ y ≠ null ⟹ ∀n ∈ g • {P(g,x,n)}{P(g,y,n)} P(g,x,n) ∨ P(g,y,n)
  /*R1*/ x ≠ null ∧ y ≠ null ⟹ ∀n ∈ g • {P(g,x,n)}{P(g,y,n)} ¬P(g,x,n) ∨ ¬P(g,y,n)
  {
    var tmp := x.next;          assume THIS_STATE_IS__l1(x, y, tmp)
    x.next := y;                assume THIS_STATE_IS__l2(x, y, tmp)
    y := x;                     assume THIS_STATE_IS__l3(x, y, tmp)
    x := tmp;                   assume THIS_STATE_IS__l4(x, y, tmp)
} }
```

Figure 4.16: Method `reverse` with its local reachability specifications and instrumentation.

Three refinements of the loop invariant are depicted. First, the programmer attempts to verify the invariant in black font (inc. A, B, C). Then, they add the missing condition, in blue font, i. e. each node is *exclusively* reachable from x or y in intermediate states of the loop (R1) Finally, they add the missing triggering patterns, in purple font (R2). The assumed instrumentation macros mark the five program states (`l0`–`l4`) that are relevant for debugging.

However, attempting to verify the program (specified using only black font in Fig. 4.16) results in a verification failure with the following message:

> *Loop invariant `DAG(g)` might not be preserved; `ACYCLIC(g)` might not hold.*

To understand what heap configuration can violate the acyclicity property, the programmer refers to the counterexample model. The raw SMT model for the above verification failure includes ca. 800 assignments;[38] rather than manually processing the model, the programmer applies ProduceReachabilityModel (Alg. 9) to automatically produce a relevant heap reachability model that can be easily visualized. This visualization is presented in Fig. 4.17.

The resulting diagram shows that acyclicity can be violated in `l4`, e. g. due to a self-edge of the form `y.next = y`. Indeed, this cyclic heap configuration can occur if y and x *alias* at the beginning of a loop iteration (note $x_{l1} = y_{l1}$ in the model). Then, the assignment `x.next := y` (Fig. 4.16) will result in a new heap edge that completes a cycle.

38 We are using Viper's VCG backend for this experiment.

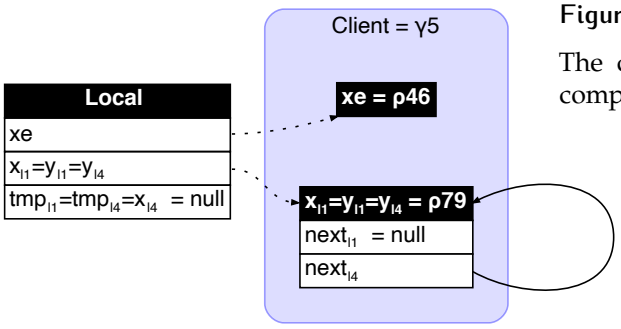


Figure 4.17: First counterexample to Fig. 4.16.

The diagram visualizes the projection of the complete model over states {l1, l4}.

4.7.1.2 First refinement. The counterexample heap model of Fig. 4.17 shows that the aliasing of x and y is a symptom of the general problem that the acyclicity invariant can be violated. On the one hand, this possibility contradicts the programmer's intuition since they expect x and y to reference heads of list segments that should be *disjoint*. On the other hand, the programmer is aware that acyclicity is a reachability property (recall that $\texttt{ACYCLIC}(\mathfrak{g}) \equiv \forall u, v \in \mathfrak{g} \bullet v.\texttt{next} \neq u \vee \neg \texttt{P}(\mathfrak{g}, u, v)$).

The programmer concludes that the missing condition must be a reachability property that, together with the rest of the invariant, ensures the disjointness of our two intermediate acyclic list segments. Concretely, they *strengthen* the invariant by adding the condition marked /*R1*/ in Fig. 4.16. Together with /*C*/, these two conjuncts of the loop invariant specify that each node must be *exclusively* reachable from either x or y (but not both at the same time).

However, attempting to verify the program (specified using black and blue font in Fig. 4.16) results in a new verification failure with the following message:

Loop invariant R1 might not be preserved; $\neg \texttt{P}(\mathfrak{g}, \texttt{x}, \texttt{n}) \vee \neg \texttt{P}(\mathfrak{g}, \texttt{y}, \texttt{n})$ *might not hold.*

To understand the problem, the programmer again refers to the counterexample model. This time, the raw SMT model includes over 1,500 assignments; hence, the programmer applies ProduceReachabilityModel (Alg. 9) to automatically produce a relevant heap reachability model. The corresponding visualization is presented in Fig. 4.18.

This diagram shows the node ρ56 that is reachable from *both* x and y in the states l1 and l4, contradicting the formula marked /*R1*/ in Fig. 4.16.

4.7.1.3 Second refinement. The fact that the contradiction depicted in Fig. 4.18 occurs in l1, i. e. even before the field assignment takes place in the loop body, hints that some information from the loop invariant cannot be used by the solver.

The incompleteness is due to *quantifier instantiation*, which is an undecidable problem; recall that in our setting, SMT solvers employ an incomplete instantiation mechanism called E-matching [11, 44].[39] To work with E-matching, the quantifiers in our program

39 We have discussed E-matching in Sec. 2.1.2.

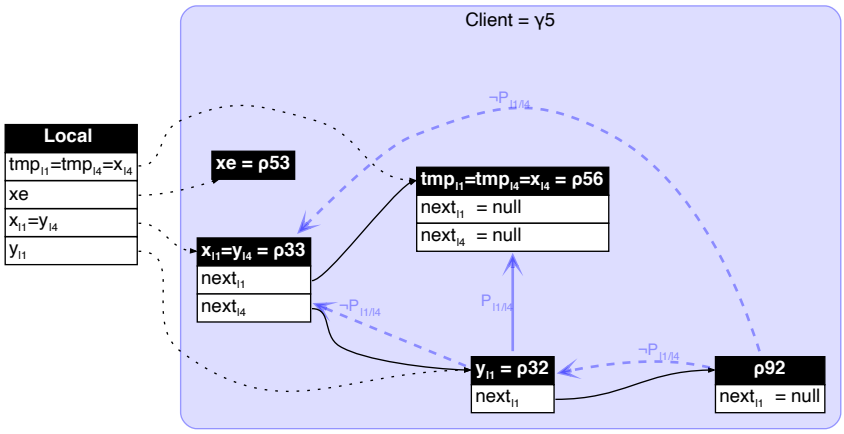


Figure 4.18: Second counterexample to Fig. 4.16.

The diagram visualizes the projection of the complete model over states {`l1`, `l4`}.

specifications must be annotated with *triggering patterns*, i. e. annotations that specify which syntactic terms should trigger a particular quantifier's instantiation.[40]

Thus, the programmer decides to revisit the patterns, discovering that they are limited for the quantifiers marked `/*C*/` and `/*R1*/` (Fig. 4.16). Concretely, these quantifiers have only one triggering pattern `{P(g,x,n)}`, while their bodies are formulas over *two* local reachability predicates (`P(g,x,n)` and `P(g,y,n)`). For both quantifiers, the programmer adds the missing triggering pattern `{P(g,y,n)}`. This enables the solver to instantiate the quantifiers based on any of the two patterns.

After this refinement, the verification of `reverse` succeeds. □

4.7.1.4 Discussion This case demonstrated that visual counterexample heap reachability diagrams can be helpful for debugging verification failures of two practically important kinds: (1) failures due to incomplete specifications and (2) failures due to inadequate quantifier triggering patterns. Manually reconstructing counterexample models based on raw SMT models is time consuming and prone to errors, e. g. because the models contain hundreds of assignments and because mapping the state-dependent relations to known program locations requires additional information. In contrast, our technique is automatic, modulo the low overhead from source program instrumentation.

4.7.2 Inspecting Union-Find

In this scenario, the programmer is given a correct implementation of the method `find` of the Union-Find disjoint set data structure, as well as its local reachability specification (Fig. 4.16). The recursive method `find` has two branches; the programmer's task is to

40 Triggering patterns are specified between { and } preceding the quantifier's body (Fig. 4.16).

```
method find(g: Set[Ref], x: Ref) returns (r: Ref)
  requires DAG(g) && x ∈ g ∧ ∀n ∈ g • P(g,x,n)
  ensures DAG(g) && r ∈ g
          ∀a,b ∈ g • P(g,a,b) ⟺ a = b ∨ b = r
{                                  assume THIS_STATE_IS__l0(r)
  if (x.next = null) {
    r := x;                        assume THIS_STATE_IS__l1(r)
    assert false                   // I
  } else {
    var h := g setminus Set(x)     // Declare new footprint
    r := find(h, x.next);
    x.next := r;                   assume THIS_STATE_IS__l2(r, h)
    assert false                   // II
} }
```

Figure 4.19: Method `find` with its local reachability specifications and instrumentation.

The original program written in black font *verifies*. To inspect possible heap configurations, the programmer adds an **`assert false`** statement at the end of each branch (first, only the line in blue font; second, only the line in purple font), causing the verifier to fail at those locations. The models obtained for each of the caused failures are *valid heap configurations* that occur along the corresponding execution paths.

construct some heap configurations that may occur after executing each of the branches in order to better understand the corresponding data structure and what information is available to the SMT solver.

OVERVIEW. The method `find` takes the footprint `g` and the starting node `x`. The method returns the *root* node, `r`, which is the last node on the heap path from `x` along `next`-fields. The precondition requires that `g` is a DAG, that `x` is indeed in `g`, and specifies that *all* nodes in `g` are reachable from `x`, i. e. the footprint is precise. The postcondition ensures that (1) all held access permissions to footprint nodes are returned to the client, while the heap configuration remains a DAG; (2) the returned node `r` is within the footprint; and (3) a `g`-local heap path $a \dots b$ exists *iff* $a = b$ (i. e. it is trivial) or $b =$ `r` (i. e. the path ends in the returned node `r`).

4.7.2.1 Inspecting the Then-branch. Assume that the programmer is not yet familiar with the Union-Find data structure. To develop understanding, they begin by considering the implementation of `find`, in particular, the first branch. This branch is taken under the condition `x.next = null`, leading to a single assignment, `r := x`. What would be a possible heap configuration in the resulting state?

To answer this question, the programmer may add an assertion contradicting the method's specifications, thus leading to a verification failure at the location of interest. In our scenario, the programmer adds the statement **`assert false`** after the assignment

in the then-branch of `find` (marked `//I`). Attempting to verify the resulting program (specified in black and blue font in Fig. 4.19) results in an (expected) verification failure with the following message:

Assertion might fail: ***`false`*** *might not hold.*

To obtain information about a possible heap configuration at the point of the failing assertion, the programmer refers to the counterexample model.[41] The raw SMT model includes 442 assignments; rather than manually processing the model, the programmer applies ProduceReachabilityModel (Alg. 9) to automatically produce a relevant heap model that can be easily visualized (Fig. 4.20).

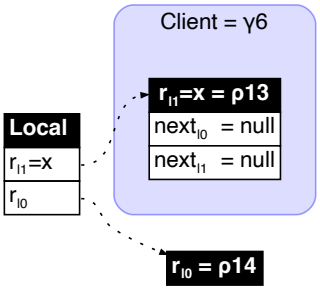


Figure 4.20: Inspecting the Then-branch to Fig. 4.19.

The diagram shows a superposition of two states, `l0` and `l1`. In `l0`, `i` refers to some node ρ14; the model does not provide any information about this node as it is outside of the current footprint (γ6). Local variable `x` is the (immutable) input parameter of `find`; its value is the same in both states (hence, no subscript is needed). In both states, the field values of ρ13 (referenced by `x`) store **`null`**. Due to `r := x`, the same object ρ13 is also referenced by `r` in `l1`.

4.7.2.2 Inspecting the Else-branch. The programmer proceeds with the second branch. This branch is taken under the condition `x.next` $\neq$ **`null`**, leading to (1) the declaration of a sub-footprint, ɧ, (2) the recursive call to `find`, and (3) an assignment to `x.next`. To develop understanding of this branch, we ask the same question again: What would be a possible heap configuration in the resulting state?

To answer this question, the programmer may repeat the inspection steps performed for the first branch. Concretely, they add an assertion contradicting `find`'s specifications, thus leading the verifier to a verification failure at the location of interest. In our scenario, the programmer adds **`assert false`** as the last statement (marked `//II`) of the else-branch. As before, attempting to verify the resulting program (specified in black and purple font in Fig. 4.19) results in an (expected) verification failure with the message:

Assertion might fail: ***`false`*** *might not hold.*

To obtain information about a possible heap configuration at the point of the failing assertion, the programmer again refers to the counterexample model. The raw SMT model includes ca. 1,600 assignments; the programmer applies ProduceReachabilityModel (Alg. 9) to automatically produce a relevant heap reachability model (Fig. 4.21).

41 A counterexample to a trivial failure is effectively an *example of a valid run* of the original program.

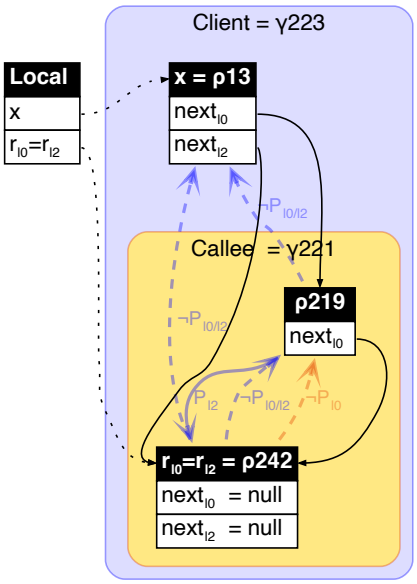


Figure 4.21: Inspecting the Else-branch to Fig. 4.19.

The diagram shows a possible heap configuration in superposition of two states, `l0` and `l2`. The heap configuration in `l0` includes the singly linked list `[x, x.next=ρ219, r]` comprising the client's footprint (𝔤=γ, i. e. the blue box). The recursive call processes the sublist `[x.next=ρ219, r]` comprising the *callee*'s footprint (𝔥=γ221, i. e. the orange box), returning `r`. Finally, the assignment `x.next:=r` *compresses* the path `x` ... `r` s.t. in `l2`, each node of the original list is directly attached to `r`. Blue (dashed) arrows depict known *client*-local (un)reachability information; the orange dashed arrow depicts the only available *callee*-local unreachability information: $\neg \mathtt{P}_{l0}(\mathfrak{h}, \mathtt{r}, \mathtt{x.next})$. Note that $\mathtt{P}_{l2}(\mathfrak{g}, \mathtt{r}, \mathtt{x.next})$ is a *spurious* bit in our SMT model.

4.7.2.3 Discussion. The problem of understanding existing algorithm implementations occurs frequently in practice e. g. among academic reviewers and formal verification students. The case that we presented demonstrates that visual counterexample heap reachability diagrams can be helpful for supporting the understanding implementations of such algorithms that are already specified and verified. The efforts required from the programmer are minimal (i. e. adding an **`assert false`** statement it the location of interest), while the resulting heap configurations are produced automatically and are sometimes non-trivial (e. g. Fig. 4.21). Beyond the information about the algorithm and its specifications, our models expose the aspects that are caused by the SMT solver's potentially incomplete reasoning, e. g. field value information and local reachability information that are out of sync.

4.8 DISCUSSION

In this section, we summarize the strengths (Sec. 4.8.1) and limitations (Sec. 4.8.2) of our verification debugging technique[42] and conclude the chapter (Sec. 4.8.3).

4.8.1 Strengths

BACKEND-AGNOSTIC DEBUGGING. Our verification debugging technique is conceptually agnostic to particular separation-logic engines. To our knowledge, this is the first static technique that works with *both* symbolic execution and verification condition-based

42 The technique is also suitable for inspecting programs that *verify*, as demonstrated in Sec. 4.7.2.

backends. The core algorithm Alg. 1 requires only minor adaptations (e. g. the arity of field-value functions must be specified) to account for the differences in SMT models used in different backends. The extended algorithm Alg. 9 requires no adaptation because our local reachability encoding is backend-agnostic.

HEAP REACHABILITY MODELS. To our knowledge, our technique is the first to produce heap reachability models for arbitrary DAGs and ZOPGs, i. e. the rich family of heap structures supported by the reachability reasoning technique of Chap. 3. Modeling heap graphs that are *beyond functional*, with multiple reference fields per node, is a fundamental challenge because the reachability relations in such structures generally cannot be described in a decidable logic, requiring unbounded quantification. In practice, this leads to models that are *partial* and, possibly, *imprecise*, e. g. in case some reachability information was not available to the solver through sufficient quantifier instantiations. Despite the incompleteness, our case study of Sec. 4.7 demonstrates that the heap reachability models produced by our algorithm are helpful for understanding verification failures.

DEBUGGING SPURIOUS FAILURES. Models produced by our technique help *understanding* the solver's imprecision rather than hiding it from the programmer. For example, the reduction step in Alg. 9 effectively compares reachability information from the model with transitive closure of direct heap edges; if the former source contradicts the latter, we still include the corresponding relations in the resulting model because such desynchronization may be the cause of the verification failure. Conversely, we declutter the model by removing reachability relations that are implied by direct edges as such relations cannot cause spurious contradictions. This way, our approach enables the programmer to understand verification failures from the perspective of the verifier.

VISUALIZATION. Our technique supports *heap diagrams*: a natural representation of counterexamples for heap-transforming programs. In fact, programmers typically sketch heap diagrams manually while studying a new program or a verification failure; our technique automates this process. Although our (independently developed) visualization approach and GRASShopper counterexamples are alike, they differ in visualizing e. g. local store variables, and our technique is capable of visualizing nested method footprints and local reachability relations.

The key principle behind our visualization approach is to present the programmer with *a single diagram per counterexample*. Since counterexamples often contain information about multiple program states, it is challenging to visualize them as a state superposition. Arguably, our layout approach results in *intelligible* state superposition diagrams in most cases, even without user input.

The programmer may select any subset of the instrumented states. Our approach to projecting the original model over user-selected states is inspired by Alloy Analyzer [65].

We deliberately limit the possible user interactions as our goal is to provide as much automation as possible.

INTEGRATION. Often, tool support and IDE integration are the practical bottlenecks of a novel debugging technique. We implemented our algorithm in a way that seamlessly integrates into the existing Viper IDE [121] workflow. Our implementation is lightweight; it reuses Viper's existing interfaces that provide program definitions and forward verification failure messages and counterexample SMT models. We provided more details about the implementation in Sec. 4.6.

4.8.2 Limitations

PARTIAL MODELS. Our technique relies on information from the SMT solver (through counterexample models). In our setting, the models are partial, and so are the counterexample heap models that our algorithm produces. We embrace partial models because *revealing* the solvers imprecision can help the programmer better understand the true cause of a verification failure. However, the programmer has to *interpret* our counterexamples in order to classify them as implementation bug, prover incapacity, or specification inadequacy. In future, we plan to augment our technique with existing approaches for automatic counterexample classification [102].

INSTRUMENTATION. To label stateful relations (e. g. fields or reachability) in a backend-agnostic manner, our technique relies on an instrumentation of the source program (Sec. 4.3). This instrumentation reuses an idea from our modular comprehensions technique of Chap. 2, allowing the programmer to label versions of the program state. Labeling program heaps is conceptually challenging because the label depends on a potentially unbounded set of memory locations, i. e. labeling functions are a special case of (heap-dependent) set comprehensions.

Our instrumentation imposes a performance overhead on the verifier. In our experiments, this overhead averaged around 25% of the original verification time, which allows debugging most verification failures that occur in practice but can theoretically lead to spurious timeouts on very complex examples. In one instance (while debugging method Union of Union-Find), we have observed that, for unknown reasons, the verification of an instrumented program took magnitudes longer than the verification of the original program. However, automatic debugging information is especially important for such complex examples that cannot be interactively re-verified. Thus, our technique could significantly benefit from a verifier's API that would supplant the need for our instrumentation's quantifier-heavy encoding.

SCALABILITY. Our counterexample-extracting algorithms Alg. 1 and Alg. 9 are guaranteed to terminate and are *worst-case polynomial* (both in terms of space and time) to the

size of the input SMT model. This makes *verification time*, rather than model extraction, the sole bottleneck of a debugging process. Compared to the scenario in which counterexamples are not extracted at all, the verification time might increase since model generation sometimes involves less optimized algorithms in SMT solver implementations. However, the most notable difference in verification times is practically due to our instrumentation that relies on universal quantification, as discussed above.

CHANGING FOOTPRINTS. The example methods presented throughout this chapter have constant footprints. However, methods can generally change their footprint, e. g. by allocating memory, which is not supported by our state-superposition visualization approach. One can nonetheless visualize changing footprints by rendering one state at a time (state projections are already available in our implementation). In future, we plan to generalize our visualization approach to support state superpositions in presence of changing footprints. A possible approach would be to symbolically label nodes that belong to the same nodeset, rather than clustering the nodes. Extending our algorithms to *collect* information about potentially changing footprints is straightforward.

ISC-BASED SETTING. We emphasized that our verification debugging technique complements the compositional verification techniques of Chap. 2 and Chap. 3. In particular, most of our benchmark methods explicitly specify their footprints as nodesets. However, our technique can support other flavors of separation logic. It is due to the instrumentation that we require explicit footprints, but our algorithm produces useful counterexamples even if the program is not instrumented. The outputs generated without an instrumentation might display auxiliary program states that are hard to understand for a non-expert. One approach is to provide a better debugging API from the verifiers (see above). Another approach is to *encode* the footprint nodeset as part of the instrumentation, if the logic supports *permission introspection*, which is the case e. g. in Viper [91].

We will discuss generalizations of our techniques in more detail in Sec. 5.1.

4.8.3 Conclusion

In this chapter, we have presented a visual counterexample-based verification debugging technique for separation logic. Our technique abstracts concrete SMT representations of stateful relations used in different verification backends. The technique enables visual verification debugging of local heap reachability properties in an ISC-based setting. Our technique is implemented as an automatic, publicly available tool [125] and is integrated into Viper IDE.

5 CONCLUSION

In this thesis, we have explored the problems related to compositional reasoning about rich properties of heap-transforming programs in separation logic. Concretely, our setting is based on the *iterated separating conjunction* (ISC) connective which allows the programmer to disentangle specifications of memory layouts and rich structural properties. The main challenge in this setting is to establish suitable conditions under which these properties are, indeed, compositional.

Set comprehensions comprise the first class of properties that we have considered. These rich properties generalize the concept of first-order quantification by summarizing potentially unbounded heap structures via commutative, associative operators, e. g. addition, multiplication, and minimization. For these properties, we proposed a novel technique based on a lightweight separation logic encoding that supports modular reasoning and enables automation. Our technique abstracts over particular object access models, supporting ordered structures, e. g. arrays, and unordered ones, e. g. graphs. To support framing of comprehensive properties, we leveraged the compositional nature of set comprehensions and developed a first-order axiomatization that automates framing for the cases of field updates and method calls. We also proposed an axiomatization that automates entailment proofs required for the verification of comprehensive properties; although such proofs generally require inductive reasoning and cannot be fully automated, our axioms improve over those used in the state of the art and are suitable to verify a diverse set of challenging benchmarks.

Heap reachability is the second class of properties that we have considered. Reachability properties complement set comprehensions, allowing to express connectivity or detachment between object pairs needed for specifying cyclic and acyclic data structures, disjointness of heap fragments, etc. We have proposed a novel technique for modular reachability specifications based on the generalized notion of local heap reachability and developed a lightweight encoding of local reachability into separation logic. We have proposed a new, highly permissive condition — relative convexity of method footprints — under which local reachability is compositional. Relative convexity enables a precise, first-order solution of the framing problem in general heap structures (with bounded outdegree). Based on relative convexity, we developed a modular verification technique for local reachability properties. We adapted existing precise update formulas for the global reachability relation to our modular setting, supporting two broad classes of heap structures: arbitrary acyclic graphs (with bounded outdegree) and (potentially, cyclic) 0–1-path graphs. We devised a first-order axiomatization that automates reachability-related entailment proofs in a diverse set of challenging benchmarks.

Finally, we have proposed a novel technique for counterexample-based verification debugging of heap-transforming programs. This technique extracts counterexample heap models from raw models produced by an SMT solver when verification fails. The technique is based on a verifier-agnostic procedure, supporting both symbolic execution and verification condition generation through only minor customization of function signatures. The technique supports an intuitive (optional) instrumentation that enables the programmer to select and label program states via ghost annotations. We demonstrated the extensibility of our procedure by proposing an algorithm that augments concrete heap models with relevant local reachability relations. To our knowledge, our approach is the first to produce counterexamples with heap reachability relations in complex structures, e. g. acyclic or 0–1-path graphs. We have demonstrated that automatically generated visualizations of our procedure's outputs are effective for diagnosing verification failure causes and for example-based understanding of typical program behaviors.

5.1 FUTURE RESEARCH DIRECTIONS

The work presented in this thesis inspires several new research directions; we will now discuss the five most promising ones. The first direction could improve completeness of our technique (Sec. 5.1.1). The second direction is related to verification debugging (Sec. 5.1.2). The third direction could explore the applications of our compositional verification techniques to a concrete imperative programming language (Sec. 5.1.3). The fourth direction could further explore algebraic properties of heaps and graphs (Sec. 5.1.4). The fifth direction could shift the focus from heap-transforming programs and towards distributed applications (Sec. 5.1.5).

5.1.1 Reducing incompleteness

Generally, a verifier targeting our setting cannot be complete without trading off some of the advantages of our techniques. Thus, some valid assertions that require incomplete reasoning might not automatically verify. Incomplete reasoning in our setting is typically due to unbounded first-order quantification or undecidable theories, e. g. nonlinear arithmetic. Although inherently incomplete, the techniques presented in this thesis are based on precise first-order reasoning, i. e. information cannot be lost as a result of decomposing or recomposing comprehensive properties or reachability properties.

The benchmarks for an inherently incomplete deductive verification technique are (formally specified) real-world programs that require nontrivial pen-and-paper proofs (by size or complexity) with a significant overhead for the programmer. Often, there are multiple alternative, mathematically valid specifications for a given program. Sometimes an automated verifier cannot verify the program against one valid specification but succeeds with a refined specification. Such issues are typically related to quantifier

instantiation, i. e. different specification versions may lead the underlying SMT solver to establish potentially different sets of lemmas.

Even without changing the specification, the programmer can aid the verifier by asserting ground facts at an intermediate state of a program that fails to verify. Although manual, these assertions are still fundamentally easier to write than formal proofs: Our specification language is seamlessly integrated into the programming language and is familiar to programmers. For example, a complete proof of Union-Find is over 600 lines of Coq [41], while in our technique the programmer needs to write only 16 lines of code-level specifications, i. e. 12 method contracts, 3 footprint declarations for the method calls, and 1 manual assertion.

It is interesting to study the potential for improving completeness of the first-order axiomatizations powering our techniques, further reducing the number of cases in which manual assertions are needed. A possible approach is to devise better triggering strategies for instantiating the axioms. Fortunately, there are existing automatic tools for diagnosing both types of triggering issues, namely, detecting overly permissive [108] and overly restrictive triggering patterns [122].

Additionally, when there will be found reachability update formulas for edge updates in graphs that fit neither the DAG nor the ZOPG classes, one could study the corresponding extension of our technique. In particular, this would simplify and further generalize our technique by potentially lifting the requirement that footprints must be either DAGs or ZOPGs. The related problem of descriptive complexity of field updates was discussed in Sec. 3.2.1.2.

5.1.2 Verification debugging

Our work on verification debugging could be continued in four orthogonal directions.

First, one could improve the existing technique's instrumentation and visualization. The visualization could be optimized, e. g. by automatically merging multiple reachability relations connecting the same pair of nodes. Generally, the standard Graphviz algorithm for strict graphs is superior to the one for multigraphs (that we use); theoretically, the layout engine could be improved, and our tool would then produce smoother diagrams. The instrumentation could be applied automatically, e. g. via a Viper plugin. Further, the instrumentation could be optimized if the verifier natively supported first-class state labels; Viper's current labels lack some needed functionality, e. g. they cannot be used to differentiate local store versions or used as function arguments.

Second, one could extend our verification debugging procedure to support compositional program properties encoded via ISC. In particular, set comprehensions are an immediate candidate. Visualizing comprehensive properties is an open problem.

Third, one could study a backend-agnostic approach to verification debugging in presence of other specification styles supported in separation logic, e. g. recursive predicates and magic wands [83]. While each verification backend may translate these specifica-

tions in a fundamentally different way, our experience with debugging ISC suggests that extracting relevant information for generating counterexample models might not require the knowledge about particular verification backend implementations. Thus, one could likely identify common abstractions, e. g. heap markers, in the case of ISC debugging, and devise minimal requirements for debugger-compatible backends. For this line of work, practically useful visualizations have been studied in the context of recursive predicate definitions [88, 95], but visualization of magic wands is an open problem.

Fourth, one could apply our procedure for generating counterexample heap reachability models to a concrete imperative programming language. In this case, it would be practically useful to combine our static counterexample generation technique with a dynamic technique for classifying verification failures [63, 102].

5.1.3 Compositional frontend verifiers

The verification techniques developed in the scope of this thesis are designed for a intermediate verification language. In future, we plan to implement a lightweight Viper frontend to apply our compositional verification techniques fully automatically.

A possible direction of future research is to also apply our techniques to a *programming* language, e. g. Go or Rust. In particular, one could support rich specifications in one of the existing Viper frontends, supporting modular comprehensive specifications and reachability properties.

An interesting setting to consider is one that lifts the requirement of *specifying* method footprints. Recall that, in an ISC-based setting, the programmer must specify footprints via an extra method parameter `g:Set[Ref]`. However, the semantics of this set can be inferred from other specifications. For example, one can typically use local reachability relations to specify whether an object belongs to a heap structure, e. g. objects in a tree must be reachable from its root. This idea has been already exploited in the EPR technique for reasoning about deterministic heap paths in linked list structures [76]. However, EPR requires the programmer to define footprints via reachability relations rather than inferring these footprints automatically. One open problem that we anticipate is automatically identifying required permission amounts, e. g. to distinguish read and write permissions, especially in concurrent programs.

5.1.4 Algebraic properties of heaps and graphs

Our results inspire the exploration of further extensions of our setting. First, the fact that relative convexity implies compositionality[1] of reachability in program heaps suggests that there may be developed (or discovered) conditions that enable compositionality of other properties that are generally non-compositional; one could thus investigate fur-

1 A nice introduction to the principle of compositionality is given in [117].

ther the algebraic properties of graphs that are useful in the context of program verification. An algebraic view of the problem of reasoning about graphs has been recently proposed in the context of effect propagation-free operations [100]. However, this work focuses on invariant preservation; to our knowledge, general algebraic frameworks and abstractions for *compositional effect propagation* are underdeveloped.

5.1.5 Compositional verification of distributed applications

The theoretical results of our work, while only scratching the surface, suggest that there may be other applications, even beyond deductive program verification, that can benefit from ideas similar to the ones powering our technique. In particular, the scope of Th. 1, and the relative convexity principle, are not limited to program heaps, and the formulation of our theorem about a three-way relatively convex partitioning carries over to the general theory of networks.

Therefore, we conclude this dissertation by considering a line of future work that targets higher abstraction levels than that of heap-transforming programs. Many compositionality-related challenges that we have faced in the current thesis exist on multiple abstraction levels, e. g. reachability, partitioning and summarization of graphs, etc. It is therefore interesting to study the extent to which our techniques and observations could be carried over to reasoning about *dapps* (decentralized applications) rather than heap-transforming programs.

On the other hand, the emerging new generation of Internet technologies, known colloquially as Web3, raises the challenge of building novel ecosystems of dapps. Typically, the novelty of dapps comes from the stronger guarantees that they provide, e. g. fault tolerance, resource sharing, or closedness, rather than from conceptually new functionality. Dapp programmers need formal verification to ensure that their implementation actually satisfies these properties, but are existing tools up to the challenge? We believe that there are open challenges that make verification of dapps an essential research area.

To support our intuition that the ideas of this thesis might help in compositional reasoning about dapps, we draw the following analogies between the two settings.

- Abstractions — Modular reasoning about classical programs is done at the method level. In contrast, Dapps consist of *actors* that interact via *APIs* [21]. Dapp developers can change an actor's implementation or run the dapp with a different network configuration, but the API is typically guaranteed to be backwards-compatible to support existing clients. Thus, it is convenient to reason about the behavior of dapps at the API level. Existing approaches based on *session types* [79, 120, 124] can express local API properties, e. g. the reaction of an actor to particular incoming messages. Compositional reasoning about rich properties of *whole* dapps, e. g. those related to network topology, is an open problem.

- Shared state — Concurrent programs often use heap memory as a shared global resource; program logics impose access protocols (e. g. permission-based access in the case of separation logic), ensuring that these resources are managed correctly. Similarly, dapps can operate on shared *distributed* resources, e. g. distributed ledgers [93]. Accessing such resources also requires carefully designed protocols, but the problem is further complicated since untrusted parties may violate these protocols. Existing techniques based on *capabilities* provide a low-level abstraction for specifying shared memory protocols in dapps [6, 115]. In future, higher-level compositional abstractions for specifying stateful API-level properties could be developed.
- Effect boundaries — A prerequisite of modular reasoning about programs is that each method's specification contains full information about its behavior. For example, a method should not access global variables in an unspecified way and should not leak memory. Similarly, dapp actors should not interact with the global environment except via their formally specified API. In particular, formal reasoning is practically intractable in case actors capture or leak capabilities [119].

BIBLIOGRAPHY

[1] G. A. Croes. "A Method for Solving Traveling-Salesman Problems." In: *Operations Research* 6.6 (1958), pp. 791–812 (cit. on p. 130).

[2] Edsger W Dijkstra et al. "A note on two problems in connexion with graphs." In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on p. 21).

[3] C. Y. Lee. "Representation of switching circuits by binary-decision programs." In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999 (cit. on pp. 2, 130).

[4] John William Joseph Williams. "Algorithm 232: heapsort." In: *Communications of the ACM* 7 (1964), pp. 347–348 (cit. on p. 21).

[5] Shen Lin. "Computer solutions of the traveling salesman problem." In: *The Bell System Technical Journal* 44.10 (1965), pp. 2245–2269 (cit. on p. 130).

[6] Jack B Dennis and Earl C Van Horn. "Programming semantics for multiprogrammed computations." In: *Communications of the ACM* 9.3 (1966), pp. 143–155 (cit. on p. 262).

[7] Robert W. Floyd. "Assigning meaning to programs." In: *American Mathematical Society Symposia on Applied Mathematics*. Vol. 19. 1967 (cit. on p. 217).

[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* 12.10 (1969) (cit. on p. 217).

[9] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm." In: *Journal of the ACM* 22.2 (1975), pp. 215–225 (cit. on pp. 118, 122, 130, 246).

[10] Sheldon B. Akers Jr. "Binary Decision Diagrams." In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516 (cit. on pp. 2, 130).

[11] Charles Gregory Nelson. "Techniques for Program Verification." PhD thesis. Stanford University, 1980 (cit. on pp. 25, 27, 28, 248).

[12] Jon Bentley. "Programming Pearls: Algorithm Design Techniques." In: *Communications of the ACM* (1984), pp. 865–873 (cit. on p. 93).

[13] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert E. Tarjan. "The pairing heap: A new form of self-adjusting heap." In: *Algorithmica* 1 (1986), pp. 111–129 (cit. on pp. 130, 152, 190).

[14] Edsger W. Dijkstra and W. H. Feijen. *A Method of Programming*. 1988 (cit. on p. 93).

[15] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. "Priority inheritance protocols: an approach to real-time synchronization." In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185 (cit. on p. 130).

[16] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-phong Vo. "A Technique for Drawing Directed Graphs." In: *IEEE Transactions on Software Engineering* 19.3 (1993), pp. 214–230 (cit. on pp. 220, 243).

[17] Guozhu Dong and Jianwen Su. "Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries." In: *Information and Computation* 120 (1995), pp. 101–106 (cit. on pp. 16, 113, 114, 130, 131, 138, 139, 152, 153, 173, 175, 176).

[18] Sushant Patnaik and Neil Immerman. "Dyn-FO: A Parallel, Dynamic Complexity Class." In: *Journal of Computer and System Sciences* 55.2 (1997), pp. 199–209 (cit. on p. 114).

[19] Gary T Leavens, Albert L Baker, and Clyde Ruby. "JML: A notation for detailed design." In: *behavioral specifications of Businesses and Systems*. 1999 (cit. on pp. 1, 200).

[20] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'99. 1999, pp. 105–118 (cit. on p. 131).

[21] Roy Thomas Fielding. "REST: architectural styles and the design of network-based software architectures." PhD thesis. University of California, 2000 (cit. on p. 261).

[22] Peter O'Hearn, John Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures." In: *Computer Science Logic*. Ed. by Laurent Fribourg. 2001, pp. 1–19 (cit. on pp. 1, 23, 110, 124).

[23] Hongseok Yang. "An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm." In: *Proceedings of the SPACE Workshop*. 2001 (cit. on p. 130).

[24] Hongseok Yang. "Local Reasoning for Stateful Programs." PhD thesis. University of Illinois at Urbana-Champaign, 2001 (cit. on pp. 2, 3, 4, 105, 133).

[25] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. 2002 (cit. on pp. 1, 27, 125).

[26] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *IEEE Symposium on Logic in Computer Science*. LICS'02. 2002, pp. 55–74 (cit. on pp. 1, 10, 105, 124, 125, 133).

[27] John Gilbert Presslie Barnes. *High integrity software: the spark approach to safety and security: sample chapters*. 2003 (cit. on p. 198).

[28] Eric Larson and Todd M Austin. "High Coverage Detection of Input-Related Security Faults." In: *USENIX Security Symposium*. 2003 (cit. on p. 2).

[29] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. 2004 (cit. on pp. 1, 125).

[30] Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. "The Boundary Between Decidability and Undecidability for Transitive-Closure Logics." In: *Computer Science Logic*. Ed. by Jerzy Marcinkowski and Andrzej Tarlecki. 2004, pp. 160–174 (cit. on p. 114).

[31] Jim des Rivières and John Wiegand. "Eclipse: A platform for integrating development tools." In: *IBM Systems Journal* 43.2 (2004) (cit. on p. 200).

[32] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. "Software refinement with Perfect Developer." In: *International Conference on Software Engineering and Formal Methods*. SEFM'11. 2005, pp. 363–372 (cit. on pp. 15, 27).

[33] David Detlefs, Greg Nelson, and James B. Saxe. "Simplify: A Theorem Prover for Program Checking." In: *Journal of the ACM* 52.3 (2005), pp. 365–473 (cit. on pp. 89, 137, 165, 183).

[34] Matthew J. Parkinson and Gavin M. Bierman. "Separation logic and abstraction." In: *POPL*. Ed. by Jens Palsberg and Martín Abadi. 2005, pp. 247–258 (cit. on p. 133).

[35] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. "PathCrawler: automatic generation of path tests by combining static and dynamic analysis." In: *European Dependable Computing Conference*. Springer. 2005, pp. 281–292 (cit. on p. 196).

[36] Alexey Gotsman, Josh Berdine, and Byron Cook. "Interprocedural Shape Analysis with Separated Heap Abstractions." In: *Static Analysis*. Ed. by Kwangkeun Yi. 2006, pp. 240–260 (cit. on p. 111).

[37] Ioannis T. Kassios. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions." In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. 2006, pp. 268–283 (cit. on p. 2).

[38] Jia Meng and Lawrence C. Paulson. "Translating higher-order problems to first-order clauses." In: *Proceedings of the 2006 Workshop on Empirically Successful Computerized Reasoning*. ESCoR'06. 2006, pp. 70–80 (cit. on p. 27).

[39] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. 2007 (cit. on p. 200).

[40] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. "A Reachability Predicate for Analyzing Low-Level Software." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. 2007, pp. 19–33 (cit. on p. 78).

[41] Sylvain Conchon and Jean-Christophe Filliâtre. "A persistent union-find data structure." In: *Proceedings of the 2007 workshop on Workshop on ML*. 2007, pp. 37–46 (cit. on p. 259).

[42] Jean-Christophe Filliâtre and Claude Marché. "The Why/Krakatoa/Caduceus platform for deductive program verification." In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 173–177 (cit. on p. 198).

[43] K. Rustan M. Leino and Rosemary Monahan. "Automatic Verification of Textbook Programs that Use Comprehensions." In: *Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs*. FTfJP'07. 2007 (cit. on pp. 28, 32, 95).

[44] Leonardo de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers." In: *Automated Deduction*. Ed. by Frank Pfenning. CADE'21. Springer Berlin Heidelberg, 2007, pp. 183–198 (cit. on pp. 25, 248).

[45] Shuvendu K. Lahiri and Shaz Qadeer. "Back to the future: revisiting precise program verification using SMT solvers." In: *POPL*. 2008, pp. 171–182 (cit. on p. 114).

[46] K. Rustan M. Leino. "This is Boogie 2." 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/ (cit. on pp. 2, 28, 94, 183, 196, 197).

[47] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *TACAS*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. 2008, pp. 337–340 (cit. on pp. 26, 33, 94, 137, 165, 183, 191, 192).

[48] Ron Aharoni and Eli Berger. "Menger's theorem for infinite graphs." In: *Inventiones mathematicae* 176 (2009), pp. 1–62 (cit. on p. 188).

[49] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A practical system for verifying concurrent C." In: *International Conference on Theorem Proving in Higher Order Logics*. 2009 (cit. on pp. 1, 2, 197).

[50] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. 2009 (cit. on p. 183).

[51] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories." In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Springer Berlin Heidelberg, 2009, pp. 306–320 (cit. on p. 25).

[52] K. Rustan M. Leino and Rosemary Monahan. "Reasoning about comprehensions with first-order SMT solvers." In: *Proceedings of the 2009 ACM symposium on Applied Computing*. Ed. by Sung Y. Shin and Sascha Ossowski. SAC'09. 2009, pp. 615–622 (cit. on pp. 30, 32, 55, 90, 94, 138).

[53] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. "Simulating reachability using first-order logic with applications to verification of linked data structures." In: *Logical Methods in Computer Science* 5.2 (2009) (cit. on pp. 115, 116, 130, 138, 163, 164, 182, 183).

[54] Michal Moskal. "Programming with triggers." In: *ACM International Conference Proceeding Series* (2009) (cit. on pp. 137, 165, 183).

[55] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. "A visual interactive debugger based on symbolic execution." In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010 (cit. on pp. 200, 203).

[56] Bart Jacobs, Jan Smans, and Frank Piessens. "A quick tour of the VeriFast program verifier." In: *Asian Symposium on Programming Languages And Systems*. 2010 (cit. on pp. 18, 199, 203).

[57] K. Rustan M. Leino. "Dafny: An automatic program verifier for functional correctness." In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. LPAR'10. Springer. 2010, pp. 348–370 (cit. on pp. 1, 2, 24, 78, 102, 197).

[58] K. Rustan M. Leino and Philipp Rümmer. "A Polymorphic Intermediate Verification Language: Design and Logical Encoding." In: *TACAS*. Ed. by Javier Esparza and Rupak Majumdar. 2010, pp. 312–327 (cit. on p. 94).

[59] Michael Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. "Specification and Verification: The Spec# Experience." In: *Communications of the ACM* 54.6 (2011), pp. 81–91 (cit. on pp. 15, 28, 78).

[60] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. 2011 (cit. on p. 26).

[61] 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: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. 2011, pp. 41–55 (cit. on pp. 1, 14, 199).

[62] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. "The Boogie verification debugger (tool paper)." In: *International Conference on Software Engineering and Formal Methods*. SEFM'11. 2011, pp. 407–414 (cit. on pp. 18, 197, 203, 204).

[63] Peter Müller and J. N. Ruskiewicz. "Using Debuggers to Understand Failed Verification Attempts." In: *Formal Methods (FM)*. Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. 2011 (cit. on pp. 5, 18, 195, 203, 260).

[64] Matthew J. Parkinson and Alexander J. Summers. "The Relationship between Separation Logic and Implicit Dynamic Frames." In: *ESOP*. Ed. by Gilles Barthe. 2011, pp. 439–458 (cit. on pp. 8, 137).

[65] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. 2012 (cit. on pp. 202, 253).

[66] Jan Smans, Bart Jacobs, and Frank Piessens. "Implicit Dynamic Frames." In: *ACM Transactions on Programming Languages and Systems* 34.1 (2012) (cit. on pp. 8, 137).

[67] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—where programs meet provers." In: *European symposium on programming*. 2013 (cit. on pp. 2, 196).

[68] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions." In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by Giuseppe Castagna. Lecture Notes in Computer Science. 2013 (cit. on pp. 2, 6, 94, 201).

[69] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. "Effectively-Propositional Reasoning about Reachability in Linked Data Structures." In: *CAV*. Ed. by Natasha Sharygina and Helmut Veith. 2013, pp. 756–772 (cit. on pp. 114, 118, 130, 131, 199).

[70] Laura Kovács and Andrei Voronkov. "First-Order Theorem Proving and Vampire." In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. 2013, pp. 1–35 (cit. on p. 24).

[71] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. "Verifying Higher-order Programs with the Dijkstra Monad." In: *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI'13. 2013, pp. 387–398 (cit. on pp. 1, 24, 78).

[72] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. "Verification of Concurrent Systems with VerCors." In: *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Ed. by Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer. 2014, pp. 172–216 (cit. on p. 6).

[73] Petr Baudiš. "Current Concepts in Version Control Systems." In: *CoRR* (2014) (cit. on p. 2).

[74] Scott Chacon and Ben Straub. *Pro git*. 2014 (cit. on p. 130).

[75] Shachar Itzhaky. "Automatic reasoning for pointer programs using decidable logics." PhD thesis. Tel Aviv University, 2014 (cit. on p. 24).

[76] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. "Modular Reasoning About Heap Paths via Effectively Propositional Formulas." In: *POPL*. Ed. by Suresh Jagannathan and Peter Sewell. 2014, pp. 385–396 (cit. on pp. 2, 5, 16, 18, 131, 141, 152, 199, 203, 260).

[77] Madiha Jami and Andrew Ireland. "A verification condition visualizer." In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2014 (cit. on pp. 198, 203).

[78] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "GRASShopper - Complete Heap Verification with Mixed Specifications." In: *TACAS*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 124–139 (cit. on pp. 4, 5, 14, 16, 18, 114, 121, 130, 131, 199, 203).

[79] Philip Wadler. "Propositions as sessions." In: *Journal of Functional Programming* 24.2-3 (2014), pp. 384–418 (cit. on p. 261).

[80] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. "First-Order Transitive Closure Axiomatization via Iterative Invariant Injections." In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. 2015, pp. 143–157 (cit. on p. 163).

[81] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. "SPARK 2014 and GNATprove." In: *International Journal on Software Tools for Technology Transfer* 17.6 (2015), pp. 695–707 (cit. on pp. 197, 203).

[82] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective." In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609 (cit. on p. 195).

[83] Malte Schwerhoff and Alexander J. Summers. "Lightweight Support for Magic Wands in an Automatic Verifier." In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by John Tang Boyland. LIPIcs. Schloss Dagstuhl, 2015 (cit. on p. 259).

[84] Afshin Amighi, Stefan Blom, and Marieke Huisman. "VerCors: A Layered Approach to Practical Verification of Concurrent Software." In: *PDP*. 2016, pp. 495–503 (cit. on pp. 6, 78).

[85] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. "Integrated environment for diagnosing verification errors." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2016 (cit. on p. 197).

[86] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Guiding Dynamic Symbolic Execution toward Unverified Program Executions." In: *International Conference on Software Engineering (ICSE)*. Ed. by Laura K. Dillon, Willem Visser, and Laurie Williams. 2016 (cit. on p. 2).

[87] David Hauzar, Claude Marché, and Yannick Moy. "Counterexamples from proof failures in SPARK." In: *International Conference on Software Engineering and Formal Methods*. 2016, pp. 215–233 (cit. on p. 197).

[88] Ruben Kälin. "Advanced Features for an Integrated Verification Environment." Master's Thesis. ETH Zurich, 2016 (cit. on pp. 201, 203, 260).

[89] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. "Static versus dynamic verification in Why3, Frama-C and SPARK 2014." In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2016, pp. 461–478 (cit. on p. 198).

[90] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution." In: *CAV*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. LNCS. 2016, pp. 405–425 (cit. on pp. 3, 4, 10, 125, 133, 162, 202, 210).

[91] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *VMCAI*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. LNCS. 2016 (cit. on pp. 1, 2, 6, 14, 24, 93, 102, 135, 162, 166, 182, 190, 196, 201, 203, 218, 255).

[92] Malte Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution." PhD thesis. ETH Zurich, 2016 (cit. on pp. 2, 6, 94, 201, 222).

[93] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. "Bitcoin as a transaction ledger: A composable treatment." In: *Annual international cryptology conference*. Springer. 2017, pp. 324–356 (cit. on p. 262).

[94] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017 (cit. on pp. 137, 165, 183, 206).

[95] Alessio Aurecchia. "Visual Debugging for Symbolic Execution." Master's Thesis. ETH Zurich, 2018 (cit. on pp. 202, 203, 260).

[96] Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. "Instrumenting a weakest precondition calculus for counterexample generation." In: *Journal of logical and algebraic methods in programming* 99 (2018) (cit. on p. 198).

[97] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. "Reachability Is in DynFO." In: *Journal of the ACM* 65.5 (2018). URL: https://doi.org/10.1145/3212685 (cit. on pp. 16, 114).

[98] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python." In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. 2018, pp. 596–603 (cit. on pp. 2, 6, 78).

[99] Tierry Hörmann. "Specification and Automated Reasoning for Datastructure Comprehensions." Bachelor's Thesis. ETH Zurich, 2018 (cit. on p. 34).

[100] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. "Go with the flow: compositional abstractions for concurrent data structures." In: *PACMPL* 2.POPL (2018), 37:1–37:31 (cit. on pp. 17, 125, 130, 131, 182, 183, 261).

[101] Peter Müller. "The Binomial Heap Verification Challenge in Viper." In: *Principled Software Development*. Ed. by Peter Müller and Ina Schaefer. 2018, pp. 203–219 (cit. on p. 183).

[102] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. "How testing helps to diagnose proof failures." In: *Formal Aspects of Computing* 30.6 (2018) (cit. on pp. 5, 18, 195, 203, 254, 260).

[103] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. "Revisiting Enumerative Instantiation." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. 2018, pp. 112–131 (cit. on p. 25).

[104] Gishor Sivanrupan. "Deductive Verification of Imperative Graph Algorithms." Bachelor's Thesis. ETH Zurich, 2018 (cit. on p. 163).

[105] Cédric Stoll. "SMT models for verification debugging." Master's Thesis. ETH Zurich, 2018 (cit. on pp. 202, 203).

[106] Alexander J. Summers and Peter Müller. "Automating Deductive Verification for Weak-Memory Programs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. 2018, pp. 190–209 (cit. on p. 6).

[107] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. "Leveraging Rust Types for Modular Specification and Verification." In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019) (cit. on pp. 2, 6, 78).

[108] Nils Becker, Peter Müller, and Alexander J. Summers. "The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Tomás Vojnar and Lijun Zhang. LNCS. 2019, pp. 99–116 (cit. on pp. 100, 165, 180, 197, 259).

[109] Martin Hentschel, Richard Bubel, and Reiner Hähnle. "The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more." In: *International Journal on Software Tools for Technology Transfer* 21.5 (2019) (cit. on pp. 18, 200, 203).

[110] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. *Local Reasoning for Global Graph Properties*. 2019. arXiv: 1911.08632 [cs.LO] (cit. on p. 127).

[111] Fabio Streun. "Tool Support for Termination Proofs." Bachelor's Thesis. ETH Zurich, 2019 (cit. on p. 13).

[112] Arshavir Ter-Gabrielyan, Alexander J Summers, and Peter Müller. "Modular verification of heap reachability properties in separation logic." In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019) (cit. on pp. 17, 103, 130).

[113] Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. *Modular Verification of Heap Reachability Properties in Separation Logic*. Tech. rep. Department of Computer Science, ETH Zurich, Switzerland, 2019. arXiv: 1908.05799 (cit. on pp. 153, 154, 163).

[114] Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. *Modular Verification of Heap Reachability Properties in Separation Logic (Artifact)*. 2019. URL: https://doi.org/10.5281/zenodo.3367478 (cit. on pp. 146, 162, 183).

[115] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. "Linear capabilities for fully abstract compilation of separation-logic-verified code." In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29 (cit. on p. 262).

[116] Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. "Certifying Graph-Manipulating C Programs via Localizations within Data Structures." In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019) (cit. on pp. 17, 124, 125, 130).

[117] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Jannis Limperg, and Johannes Hölzl. *The Hitchhiker's Guide to Logical Verification*. 2020 (cit. on p. 260).

[118] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. "Local Reasoning for Global Graph Properties." In: *Programming Languages and Systems*. Ed. by Peter Müller. 2020, pp. 308–335 (cit. on pp. 125, 126, 130).

[119] Fengyun Liu, Sandro Stucki, Nada Amin, Paolo Giosuè Giarrusso, and Martin Odersky. *Stoic: Towards Disciplined Capabilities*. Tech. rep. 2020 (cit. on p. 262).

[120] Anson Miu. "Type-safe Web Programming Using Routed Multiparty Session Types in TypeScript." PhD thesis. Imperial College London, 2020 (cit. on p. 261).

[121] Linard Arquint, Alessio Aurecchia, Ruben Kälin, Valentin Racine, and Arshavir Ter-Gabrielyan. *Viper IDE*. Aug. 2021. URL: https://github.com/viperproject/viper-ide (cit. on pp. 19, 254).

[122] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. "Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers." In: *Formal Methods (FM)*. To appear. 2021 (cit. on pp. 19, 259).

[123] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. *Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers*. Tech. rep. Department of Computer Science, ETH Zurich, Switzerland, 2021. arXiv: 2105.04385 (cit. on p. 94).

[124] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. "Machine-checked semantic session typing." In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2021, pp. 178–198 (cit. on p. 261).

[125] Arshavir Ter-Gabrielyan. *Lizard: The Visual Debugger for Viper*. 2021. URL: https://github.com/viperproject/lizard (cit. on pp. 19, 243, 255).

[126] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. *Gobra: Modular Specification and Verification of Go Programs (extended version)*. Tech. rep. 2021. arXiv: 2105.13840 (cit. on pp. 2, 6).

CURRICULUM VITAE

Name: Arshavir Ter-Gabrielyan

Date of birth: April 24, 1991

Place of birth: Yerevan, Armenia

Citizenship: Russia

Contacts:

aterga.github.io

linked.in/Arshavir

tergabrielyan@gmail.com

EDUCATION

2015–2021	**Doctor of Sciences** ETH Zurich, Switzerland Thesis: *Compositional Verification of Rich Program Properties in Separation Logic* Adviser: Prof. Peter Müller
2013–2015	**Master of Science in Applied Mathematics and Physics (de facto Computer Science), *Red Diploma*** Moscow Institute of Physics and Technology, Russia Thesis: *A Machine Learning Approach to Static Code Analysis*
2009–2013	**Bachelor of Science in Applied Mathematics and Physics** Moscow Institute of Physics and Technology, Russia Thesis: *A Control Flow Optimization for Multi-Strand Architectures*

EMPLOYMENT

2021–now	**Software Engineer** DFINITY Foundation, Zurich, Switzerland
2015–2020	**Research and Teaching Assistant** Programming Methodology Group, ETH Zurich, Switzerland
2014–2015	**Research Scientist** Strategic CAD Labs, Intel Corporation, Moscow, Russia
2011–2014	**Software Engineering Intern (Long-Term)** Intel Corporation, Moscow, Russia