Malte Hermann Schwerhoff

Advancing Automated, Permission-Based Program Verification Using Symbolic Execution

Diss. ETH 23977 | 2016

Diss. ETH No. 23977

Advancing Automated, Permission-Based Program Verification Using Symbolic Execution

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

Presented by

Malte Hermann Schwerhoff
MSc ETH CS, ETH Zurich
Born on 22nd March 1981
Citizen of Germany

Accepted on the recommendation of

Prof. Peter Müller (examiner)
Prof. Bart Jacobs (co-examiner)
Prof. Viktor Kuncak (co-examiner)
Prof. Martin Vechev (co-examiner)

2016

Für Birte. Für deine Liebe, dein Lachen und dein Vertrauen.

Abstract

Proving the correctness of programs by rigorously applying formal methods such as deductive verification has attracted substantial interest in the last two decades, from both the scientific community and industry. This has led to numerous important theoretical results and successful practical applications, benefiting greatly from two parallel developments: (1) the introduction and exploration of *permission logics* such as separation logic, which are particularly suitable for *modularly* reasoning about *concurrent, heap-manipulating* programs, and (2) the emergence of a de-facto standard architecture for reusable *verification infrastructures*, which significantly reduces the effort involved with developing *automated* verification tools. The existence of automated verification tools is a prerequisite for scaling verification to the size and complexity of real-world software, but it also makes verification more easily accessible and applicable in general: as a result, researchers are provided with valuable feedback, enabling them to identify remaining challenges. Existing verification infrastructures, however, are not well-suited for permission-based verification, which impedes the further development of the field. In particular, central aspects such as program heaps and permissions must be encoded, which substantially complicates the development of efficient and precise verification tools.

This thesis aims to remedy the situation by developing a verification infrastructure that facilitates the development of automated, permission-based verification tools for race-free, concurrent and heap-manipulating programs. To achieve this goal, we make the following three contributions: first, we design Viper, a novel *intermediate verification language*, carefully balanced to be sufficiently expressive to enable the encoding of different programming languages, program properties and specification languages, while also facilitating the development of efficient and precise verification tools. Second, we develop Silicon, an automated, symbolic-execution-based verifier for the Viper language that exhibits stable and good performance, and is more complete than comparable verifiers. Third, we extend Viper and Silicon with support for two challenging features of permission logics — iterated separating conjunctions and magic wands — that have shown to be useful in by-hand proofs of various properties, but for which no existing verifier provides comparably direct and automated support.

The *Viper verification infrastructure*, which includes the Viper language and the automated verifier Silicon, already had an impact on the field of automated, permission-based verification: it is actively being used at three different universities for building several verification tools, it won an award at a verification competition and it was used for teaching at a summer school.

Zusammenfassung

In zahlreichen universitären und industriellen Forschungsprojekten wurde in den letzten zwei Jahrzehnten die Möglichkeit untersucht, die Fehlerfreiheit von Programmen mittels formaler Methoden mathematisch präzise zu beweisen. Diese Anstrengungen führten zu einer Vielzahl bedeutsamer Ergebnisse, sowohl auf der theoretischen als auch auf der praktischen Seite; Erfolge, die durch zwei parallel stattfindende Entwicklungen ermöglicht wurden: (1) die Einführung und intensive Weiterentwicklung von Separation Logic und anderen *Permission-Logiken*, die sich als besonders hilfreiches Instrument zur *modularen* Verifikation von nebenläufigen, heap-manipulierenden Programmen erwiesen, sowie (2) das Herauskristallisieren einer De-facto-Standardarchitektur für wiederverwendbare bzw. vielseitig einsetzbare *Verifikationsinfrastrukturen*, welche den mit der Entwicklung von *automatisierten* Verifikationswerkzeugen verbundenen Aufwand erheblich reduzieren. Erst solche automatisierten *Verifier* ermöglichen die Verifikation von grossen, in der Praxis genutzten Softwaresystemen: ohne Automatisierung liessen sich die zur Programmverifikation eingesetzten Techniken nicht mit vertretbarem Aufwand auf die Komplexität dieser Systeme skalieren. Die Anwendbarkeit ihrer Techniken auf zahlreiche unterschiedliche Systeme ermöglicht es zudem Forschern, wertvolle Rückmeldungen von Anwendern zu erhalten, die dabei helfen können, noch offene Probleme zu identifizieren. Existierende Verifikationsinfrastrukturen sind jedoch nur unzureichend dazu geeignet als Grundlagen für die Entwicklung von permission-basierten Verifiern verwendet zu werden: zentrale Aspekte der Programmverifikation wie Heaps und Permissions können nicht direkt repräsentiert werden und müssen daher kodiert werden, was die Entwicklung von effizienten und präzisen Verifikationswerkzeugen substanziell erschwert.

Das Ziel dieser Arbeit ist es daher eine Infrastruktur bereitzustellen, die die Entwicklung von automatisierten und permission-basierten Werkzeugen für die Verifikation von nebenläufigen, interferenzfreien und heap-manipulierenden Programmen massgeblich vereinfacht. Den Kern dieser Arbeit bilden die folgenden drei Beiträge: (1) der Entwurf von Viper, einer neuartigen *Zwischensprache*, die als Grundlage für die Entwicklung von effizienten und präzisen Verifiern dient und die ausdrucksstark genug ist, um als Zwischensprache für unterschiedliche Programmiersprachen, Spezifikationssprachen und Programmeigenschaften genutzt zu werden; (2) die Entwicklung von Silicon, eines automatisierten, auf Symbolic Execution basierenden Verifiers für Viper, der vollständiger als vergleichbare Verifier ist und trotzdem gute und stabile Laufzeitleistung erbringt; (3) die Erweiterung von Viper und Silicon um die Unterstützung für zwei anspruchsvolle, permission-logische Junktoren — die iterierte trennende Konjunktion sowie die trennende Implikation — die sich in manuell geführten Beweisen als äusserst nützlich erwiesen haben, aber bisher nicht von automatisierten Verifiern unterstützt wurden.

Die im Rahmen dieser Arbeit entwickelte *Verifikationsinfrastruktur Viper*, welche die Viper-Zwischensprache sowie den automatisierten Verifier Silicon beinhaltet, wird bereits an drei Universitäten zur Entwicklung von Verifikationswerkzeugen genutzt, sie gewann einen Preis während eines Verifikationswettbewerbs und sie wurde in der Lehre an einer Sommerakademie eingesetzt.

Acknowledgements

Over the course of the last five years, many people have directly or indirectly influenced my work, my view on computer science, and also myself. To all of you: thank you for your advice, your help and your friendship.

I am deeply grateful to my supervisor, Peter Müller: your door was always open when I needed feedback and you made time when I needed advise, but you also gave me the freedom to set my own goals and to develop my own strategies for how to achieve them. Naturally, the latter involved a few setbacks, but you always affirmed your trust in me and gave me the feeling that I would eventually succeed. You have a remarkable talent for helping others to untangle their thoughts: I would often come to your office, my brain clogged with a ravel of different problems and solution attempts, and would leave it with the same web of thoughts, but so clearly laid out that I could see for myself which thread to follow next. I also consider myself lucky to have had a supervisor that is not only a brilliant researcher, but who truly cares about the people he works with, regardless of whether they are students, PhDs, research colleagues or general staff. I learned so incredibly much in the last five years — thank you, Peter, for making it all possible!

My PhD would not have been such a great experience without Alex Summers, my office mate and day-to-day supervisor: when I still was a Master's student and novice teaching assistant for Formal Methods, you kept on hinting at a realm beyond Hoare logic — with intriguing notions such as hypothetical worlds shaped by magic wands — and in the last five years, you continuously helped me to understand and explore this realm. What I admire the most about you, Alex, is your dedication to doing things right, and the inspiring and infectious joy you radiate when you do it. This, your strong desire of taking a holistic view on essentially every topic, and your gift of immediately spotting potential problems in anything from sketchy ideas to page-long explications, make you an exceptional scholar. However, you have been much more than an advisor and a work colleague to me: you introduced me to beer festivals and quirky dinosaur comics, and you got me to enjoy skiing and watching field hockey. Most importantly, though: you were a true friend. You cheered me up when things looked gloomy, and you celebrated with me when things went great. I will never forget our time together here in Zurich!

I would also like to thank my co-examiners Bart Jacobs, Viktor Kuncak and Martin Vechev, who invested a lot of time and effort into reviewing my thesis and ensuring that it has the expected quality. I am particularly thankful to Bart, with whom I had the pleasure of working during an internship at KU Leuven: seeing a different approach to identifying worthwhile scientific challenges and a different way of leading a research group made this visit both an insightful and fun experience. To Bart and his students Amin Timany, Gijs Vanspauwen and Willem Penninckx: thank you for including me in your team from the very first moment on.

Throughout the years, I had the good fortune of being part of a great research group and enjoying the company of many former and current colleagues: my own generation, Dimitar Asenov, Maria Christakis, Lucas Brutschy, Uri Juhasz, Milos Novacek and Valentin Wüstholz; the previous generation, Cédric Favre, Pietro Ferrara, Ioannis Kassios, Hermann Lehner, Arsenii Rudich and Joseph Ruskiewicz; the new generation, Alexandra Bugariu, Jérôme Dohrau, Marco Eilers, Arshavir Ter-Gabrielyan and Caterina Urban; the long-term visitors Pontus Boström and John Boyland; and the best secretary of the world and heart of the group, Marlies Weissert. To all of you: I am grateful that I had the opportunity to go a (large) part of the way together with you, and I would like to thank you for vivid discussions at the whiteboards and over coffee, and for numerous fun pub evenings and board game sessions! I wish you all the best for your future endeavours and I sincerely hope we stay in touch.

I would also like to thank the Bachelor's and Master's students that I had the pleasure of supervising: Mathias Birrer, Bernhard Brodowsky, Andres Bühlmann, Andreas

Buob, Ivo Colombo, Simon Fritsche, Flavio Goldener, Roger Koradi, Rokas Matulis and Roland Meyer. Each of you contributed a piece to this thesis — thank you for working with me!

If I hadn't met a few particular people, I would never have started this journey: my Bachelor's professors Marcel Luis and Heinrich Overhoff, who encouraged me to change over and continue my Master's at ETH Zurich; and my Master's mate and close friend Simon Hudon, whose unbridled excitement about the beauty of formal proofs ultimately awakened my own interest in formal methods. To all three of you: thank you!

Ultimately, this thesis would not exist without the ceaseless and invaluable support from my family: from my fiancée Birte, my siblings Mona and Simon, and my parents Ida and Rolf. You give me the courage to aim high, the strength to persevere and the necessary groundedness to never forget what really matters in life: you.

Contents

Chapter 1

Introduction

Software is everywhere: on thumbnail-sized embedded devices and on phones, it controls networks, household appliances, cars and planes, it transforms the way we communicate and suggests what we should read, watch and buy. Unfortunately, faulty software is ubiquitous as well, incurring financial losses but also causing loss of life: in 1992, 28 soldiers died as a consequence of a software bug in the Patriot missile system [131], and in 2002, the now-famous NIST report estimated that faulty software may cost up to $59.5 billion per year [128]; more recent examples include Nissan recalling over 900,000 cars due to a bug in the airbag control software [129], the TimSort bug [51] which affected the standard libraries of Java, Python and Android, and a spate of bugs such as Heartbleed, Shellshock and Stagefright that were widely reported on by the media and even had their own logos. Lots of effort has been invested into improving the quality of software, for example, by developing standard models for managing the software development process, by systematically integrating automated testing into the development process, and by applying formal methods that ensure the absence of certain errors through rigorous mathematical proofs.

Program verification, one such formal method with a research history of now over 50 years, is being increasingly applied in industry, for example, to prove the safety of third-party device drivers [4, 77], to verify the security, safety and liveness of apps in distributed systems [55, 56], and to detect bugs in Android and iOS apps [34]. Different program verification techniques exist, with different strengths and weaknesses, but of all these, *deductive* verification is arguably the most versatile (in particular with respect to tool support): it enables *rich specifications* and *modular reasoning*, it is suitable for *automation*, and it is applicable to sequential and *concurrent*, *heap-manipulating* programs.

Deductive program verification can guarantee a large variety of *safety properties*: the absence of common errors such as null-pointer dereferences and out-of-bounds array accesses; the absence of concurrency errors such as memory race conditions and synchronisation deadlocks; framing properties such as that a sorting algorithm modifies the input array, but not the elements in the array; and functional properties such as that the sorting algorithm actually sorts its input. Additionally, deductive program verification can also prove certain *liveness* properties which can be reduced to safety properties (through suitable annotations), such as termination and finite blocking. We use the term *full-functional verification*, encompassing all previously listed properties and potentially others, to emphasise the diversity of the properties deductive program verification is suitable for.

In addition to enabling the verification of such a variety of properties, deductive verification also enables *modular reasoning*, such that, for example, each method or thread can be verified in isolation. *Modularity* is essential for scaling full-functional verification to the size and complexity of real-world software, but also for verifying properties of libraries, and in general of modules potentially used in arbitrary contexts. Moreover, modularity can avoid the need for reverifying a module's clients if the implementation of the module changes, which is crucial for reducing the verification effort as software evolves.

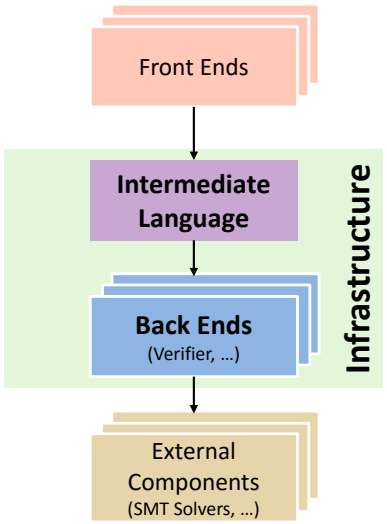

Figure 1.1: The de-facto standard architecture for reusable verification infrastructures, as implemented by Boogie [6] and Why3 [18]. The infrastructure comprises an intermediate verification language, which enables different front ends to encode their verification problems, and back ends that process the resulting programs, for example, verifiers or static analysers. Back ends are typically built on existing components such as SMT solvers or abstract domain libraries. Verification infrastructures significantly reduce the complexity of developing new front ends: a well-designed intermediate verification language simplifies the encoding task, and front ends benefit from improvements of the back ends.

For methods, modularity is typically achieved by providing specifications, such as pre- and postconditions, that summarise the behaviour of a method in a way that abstracts over the method's implementation; for threads, modular reasoning is often enabled by enforcing a protocol that restricts the concurrent behaviour.

In addition to modularity, applying full-functional verification to large, evolving software systems also requires techniques for (partially) *automating* the proof steps involved in verification. In the context of deductive verification, *automation* is commonly understood as verifying a program, once it has been annotated with specifications (such as pre-/postconditions), without any further manual intervention. Due to the expressiveness of the supported specifications, such *automated verifiers* commonly require additional assertions, which correspond to key steps in the (otherwise automated) proof, to guide the verification.

Reasoning modularly (and soundly) about sequential or concurrent programs with shared mutable state, that is, a program heap, requires addressing the *frame problem*: identifying the access set of an operation such as a method call, that is, the set of memory locations that an operation potentially reads from or writes to. Solving the frame problem is complicated by various challenges, including aliasing, concurrency, and dynamic (de)allocation of memory and threads.

In the last two decades, the research community has proposed several solutions to the frame problem: in *ownership*-based verification [93] and in *dynamic frames* [71], framing is essentially realised by quantifying over access sets, whereas in *separation logic* [99, 108] and related *permission logics*, which are particularly suitable for reasoning about concurrent programs, framing is realised by extending first-order logic with specialised logical connectives (more details follow in Section 1.2). Subsequent research and tooling efforts achieved significant progress towards scaling full-functional verification to real-world programs. The availability of automated verifiers has shown to be particularly important: not only, because it is essential for verifying large programs, but also because it makes program verification more easily accessible and applicable, which makes verification more appealing to practitioners and students, and facilitates teaching. Increasing the user base of their work in turn provides researchers with valuable feedback, which enables a better understanding and comparison of the strengths and weaknesses of individual approaches, and of the challenges that still need to be solved.

In parallel, a de-facto standard architecture for a reusable *verification infrastructure* has emerged (Figure 1.1), which substantially simplifies the development of new verifiers (and which has been influenced by analogous developments in the field of programming language compilers): the three-tier infrastructure is centred around an *intermediate verification language* into which (multiple) *front ends* can encode source

programs along with their specifications, and for which (multiple) *back ends* exist, for example, for verifying the resulting encoding by creating suitable verification conditions, or for inferring missing specifications. Back ends are often themselves built on existing components such as SMT solvers or abstract domain libraries.

Such a verification infrastructure reduces the effort involved with developing new front ends because the abstraction gap between the source language and the language used, for example, by an SMT solver is typically much larger than the gap between the source language and the intermediate verification language, which enables front-end developers to focus on the key ideas of the encoding. In addition, front-end developers can benefit from improvements made to the lower levels of the infrastructure, such as increased expressiveness of the intermediate verification language, improved performance of the back ends, or newly added back ends.

Two widely used verification infrastructures are Boogie [6] and Why3 [18]; their intermediate verification languages are a procedural, guarded-command-like language, respectively, a dialect of ML, both with support for specifications expressed in first-order logic. The intermediate verification languages do *not* have a built-in notion of a program heap, and thus require front-end developers to choose (and implement) a suitable encoding thereof; the loss of important structural information incurred by a heap encoding typically complicates the implementation of efficient and precise back ends such as verifiers and static analysers. Moreover, specifications that are not expressed in first-order logic, such as permission-logic-based specifications, need to be encoded as well, which aggravates the aforementioned problems. More details are provided in Section 1.2.

1.1 Problem Statement

The field of automated program verification has benefited greatly from the existence of verification infrastructures such as Boogie or Why3, but existing infrastructures are not well-suited for permission-based verification, an approach to verification that has shown to be particularly useful for reasoning about heap-manipulating concurrent programs. Developing a new permission-based verifier thus remains unnecessarily challenging and resource-consuming, which impedes experimenting with newly developed permission-based program logics beyond the size of programs for which by-hand proofs are tractable.

To overcome this situation, the goal of this thesis therefore is to *develop a Boogie-like verification infrastructure for automating the permission-based verification of race-free concurrent programs that manipulate shared data*. This goal can be divided down into two sub-goals: designing a suitable intermediate verification language, and developing an automated verifier for this language.

Language: A suitable intermediate verification language enables the development of different front ends that encode different procedural and object-oriented languages annotated with different (permission-based) specification styles, and of different, efficient and precise back ends. Several requirements can be derived from this characterisation:

- The level of abstraction provided by the intermediate verification language should be carefully chosen: the language should be expressive enough to support the encoding of different programming languages, control-flow statements and concurrency features, but it should preserve sufficient structural information to facilitate the implementation of efficient automated verifiers, and to make the encoding amenable to precise static analyses.
- The intermediate verification language should be expressive enough to support different specification styles: it should support idioms commonly used in separation logic, such as abstract predicates [101] and fractional permissions [25],

but also facilitate the combination of these idioms with idioms more commonly used by other verification approaches, for example, quantifiers and abstraction functions [60]; this will enable similar combinations in front ends. At the same time, the supported specification features should not prevent the development of efficient and precise back ends.

- The language should not restrict the kinds of (heap-implemented) data structures and (pure) mathematical structures that can be specified: it allows specifying arbitrary data structures, including recursively-defined data structures such as lists and trees, but also less directed data structures such as graphs, and random-access data structures such as arrays. Since properties of data structures and operations thereon are commonly specified using mathematical structures such as tuples, sets, sequences or maps, the intermediate verification language should support the definition of such structures as well.
- Extending the set of heap-implemented and purely mathematical structures that can be reasoned about should be directly possible on the level of the intermediate verification language. That is, it should suffice to add appropriate definitions (expressed in the intermediate verification language) to the input program, without having to make additional changes on other levels of the verification infrastructure, for example, to the back ends.
- In order to facilitate experimenting with potential encodings (rapid prototyping), but also for educational purposes and for participating in verification competitions, the intermediate verification language should make it convenient to manually encode examples. The language should therefore be designed with the aim of keeping the annotation overhead (comparably) low.

Verifier: A suitable automated verifier requires only few guiding assertions, and offers a predictable and good performance.

- The verifier should allow users to understand automation limitations on the level of the intermediate verification language, and not require detailed knowledge about the verifier's implementation. This is particularly important for user-provided annotations (other than specifications), which are commonly required to instruct the verifier to perform a step that has not been automated, such as unrolling a recursive definition or attempting an inductive proof. The need for such annotations should be explainable on the level of the intermediate verification language and motivated by a conceptual problem; it should not be the consequence of a particular detail of the verifier's implementation. This requirement is crucial for enabling the development of diverse front ends, since it can otherwise be difficult to anticipate when additional annotations are required.
- The verifier should enable an IDE-like experience: it should be sufficiently fast such that users can continuously work on verifying programs (of reasonable size), in particular since the verification is modular; and it should report verification failures (and related feedback) in a way that facilitates locating the source of the failure.
- The performance should be predictable, regardless of whether the verification succeeds or fails. Small modifications to the program under verification, in particular modifications that do not semantically change the program, should not significantly influence the performance. The performance should also not significantly vary across different possible encodings.
- Analogous to the case of automation limitations, the verifier should allow users to understand performance limitations on the level of the intermediate verification language, and not require detailed knowledge about the verifier's implementation.

Since verification infrastructures such as Boogie and Why3 are not well-suited for permission-based verification, existing verifiers for permission logics have typically been developed from the ground up, with reuse, if any, only at the lowest level (SMT solvers). Such verifiers are not well-suited as the core of a verification infrastructure for permission-based reasoning either: some are too limited in their expressiveness to enable full-functional verification, but highly automated, whereas others are very expressive, but limited in their degree of automation, which complicates the encoding of verification approaches that aim at a higher degree of automation (in their front ends). In addition, the specification languages supported by existing verifiers have usually been developed for a specific programming language and with a certain style of specifications in mind, which complicates their reuse in different contexts. More details are provided in the next section.

1.2 State of the Art

This section gives an overview of the state of the art of modular, automated, full-functional verification of heap-manipulating concurrent programs. The overview is structured as follows: the existing verification infrastructures Boogie and Why3 are described first, followed by an overview of individual verifiers that do not form such an infrastructure. The latter overview is subdivided into framing approaches.

1.2.1 Infrastructures

The Boogie verification infrastructure [6] consists of the Boogie intermediate verification language and the Boogie verifier, which encodes a Boogie program as a verification condition that is passed on to an SMT solver. The language is imperative, and offers conditional and `goto` statements, local and global variables, loops and procedures. In addition to specifications such as pre-/postconditions and loop invariants, the language provides `assume` and `assert` statements, which can be used to encode the semantics and specifications of sequential and concurrent source programs. Termination checks (for loops and recursions) are not enforced, but can be encoded. Assertions are expressed in a first-order logic, and mathematical structures can be defined via uninterpreted functions and appropriate (quantified) axioms. Polymorphic maps are built into the language, and frequently used in encodings. Boogie does not have a built-in notion of a program heap (and thus of permissions), and its first-order-logic-based assertion language does not directly support specifications based on permission logics. Encoding program heaps and permissions is possible (as demonstrated by Chalice [84]), for example, via polymorphic maps, but tedious and error-prone; the loss of structural information also hampers precise static analyses, such as abstract interpretation. We interpret the fact that only one (namely Chalice) of the many tools encoding to Boogie is permission-logic-based as a direct consequence of the challenges arising from having to encode such logics into an intermediate verification language that does not have a built-in notation of a heap and of permissions.

The Why3 verification infrastructure [18] consists of the WhyML intermediate verification language, from which verification conditions, expressed in the Why3 language, are generated, which are then encoded to an SMT solver. The Why3 language is essentially a first-order logic enriched with algebraic data types and pattern matching. Why3's standard library includes numerous theories (expressed in the Why3 language), including mathematical structures such as sequences, and concepts such as orders; custom theories can be added as needed. The intermediate verification language WhyML is a sequential, first-order dialect of ML with records and functions, which also offers loops and exceptions. Similar to Boogie, Why3 supports `assert` and `assume` statements that can be used to encode a range of source program semantics

and specifications. Mutable references are possible, but a static alias control is imposed; in particular, recursive data types with mutable components are disallowed. Encoding programs with unrestricted aliasing requires, as in the case of Boogie, an encoding of the program heap, which entails aforementioned problems (and analogous for permissions). To our knowledge, no permission-logic-based front end for Why3 exists.

1.2.2 Individual Verifiers

Separation Logic

The key feature of specification languages based on separation logic [99] is the possibility of decomposing the (verification) heap into disjoint sub-heaps, which enables local reasoning about heap-manipulating programs and yields an immediate solution to the frame problem: if two operations work on disjoint sub-heaps, then the operations' executions cannot interfere. In specifications, a *points-to predicate* $o.f \mapsto V$ expresses that the specified operation may access location $o.f$ (whose value is V), and combining points-to predicates with *separating conjunctions*, as in $o_1.f \mapsto V_1 * o_2.f \mapsto V_2$, expresses that a heap can be partitioned into two disjoint sub-heaps, and thus, that o_1 and o_2 are not aliased. It is therefore save to assume that an operation that (only) works on $o_1.f$ cannot affect $o_2.f$.

A points-to predicate can intuitively be understood as denoting the *exclusive access permission* to a heap location, and the separating conjunction can be understood as adding up permissions. In this interpretation, the assertion $o_1.f \mapsto V_1 * o_2.f \mapsto V_2$ intuitively implies non-aliasing of o_1 and o_2 because the assertion would otherwise denote the exclusive access permission twice.

Various verifiers for specification languages based on separation logic exist. The first verifier that was developed for separation logic is Smallfoot [11], a verifier for programs written in a C-like imperative language. Smallfoot is based on a custom entailment prover for separation logic, which is limited to reasoning about properties of programs that manipulate linked lists and trees, whereas we aim at verifying arbitrary data structures.

jStar [42], a verifier for Java programs, is also based on a custom entailment prover for separation logic, which was subsequently released as coreStar [23]. coreStar can be extended by custom proof rules — and in practice must be since hardly any proof rules are predefined — which makes coreStar, and thus also jStar, very flexible. This flexibility comes at a cost, however: users have to leave the level of the specification language in order to provide additional proof rules, for example, in order to reason about custom data structures, whereas we strive to develop an infrastructure that is extensible on the level of the intermediate verification language. According to the authors, "... at the moment, jStar might be too complex to use by programmers" [42]. Moreover, the degree of flexibility that coreStar provides makes it unclear to which extent the implementation can be optimised for performance. Later work by Botinčan et al. [22] improved jStar by integrating an SMT solver into coreStar, but this did not in general address the need for extending the underlying entailment prover.

VeriFast [67], a verifier for C and Java programs, achieves remarkable and predictable performance, but at the cost of automation: VeriFast commonly requires non-negligible amounts of additional assertions to direct the verifier, which potentially complicates the encoding of approaches that aim to increase the level of automation — which is one of our goals — since it is in general difficult for front ends to anticipate when additional assertions are required. VeriFast's specification language offers several advanced features such as higher-order predicates and functions, but it only offers limited flexibility regarding specification styles, whereas we seek to enable combinations of different such styles.

The HIP/SLEEK system, based on work by Chin et al. [35], is a verifier (HIP) for a C-like imperative language which uses a custom entailment prover (SLEEK) to discharge proof obligations. The system supports custom predicates, provided that exactly one reference parameter is traversed in the predicate's definition; this conflicts with our goal of supporting arbitrary data structures. Values can be specified via (dis)equalities, Presburger arithmetic and bag constraints. The latter are handled by an interactive theorem prover, whereas we aim at automated verification.

GRASShopper [103], a verifier for a C-like imperative language, supports a subset of separation logic that ensures proof obligations which can be translated into a decidable fragment of first-order logic, and can then be discharged by an SMT solver. This approach allows highly-automated reasoning, but is currently limited to reasoning about programs that manipulate list- and tree-like structures and does not support arbitrary data structures, as is our goal.

Chu et al. [36] report on a specification language with explicit heaps: in every assertion, the global program heap has to be explicitly decomposed into named sub-heaps that can, but do not have to be, disjoint. Predicate definitions can be parameterised by sub-heaps, which enables structure sharing across predicate instances. The report mentions an (unreleased) implementation prototype for a sequential C-like language without loops, but with recursive functions. However, the report does not give many details; in particular, the expressiveness of the supported specification language is not described and does not become apparent from the few given examples. An evaluation of the prototype is also not available.

Implicit Dynamic Frames

Implicit dynamic frames [121] is a program logic closely related to separation logic [102]: the logic provides *accessibility predicates* $\texttt{acc}(x.f)$ to denote access permissions to location $x.f$, and heap-dependent expressions such as $x.f > 0$ are used to constrain heap values; the corresponding separation logic assertion is $x.f \mapsto V \wedge V > 0$. Implicit dynamic frames also provides a separating conjunction: the assertion $\texttt{acc}(x.f) * \texttt{acc}(y.f)$ requires $x.f$ and $y.f$ to denote two disjoint heap locations, and thus forbids aliasing between x and y.

The first verifier that was developed for implicit dynamic frames is VeriCool [121], a verifier for a Java-like language. The specification language supported by VeriCool permits, among other things, arbitrary recursive predicates and heap-dependent abstraction functions. In their presence, the verifier generates verification conditions that are likely to cause unstable SMT solver performance (small changes to the program or the specifications can significantly affect the performance), including non-termination. In contrast, we strive for predictable and good performance. As a consequence of the unstable behaviour of the verifier [118], an alternative back end, based on symbolic execution instead of verification condition generation, was developed, which was originally released under the name SpecCheck [119], but later on integrated into VeriCool. The symbolic execution back end supports only a subset of VeriCool's specification language, however, which excludes specification features such as fractional permissions and quantifiers, which are important for full-functional verification and which we aim to support in order to allow a wide range of front ends.

Chalice [84], a verifier for a simple object-based language (without inheritance) with fork-join concurrency and message-passing channels, encodes a given program and its specifications into Boogie. The VerCors project [16] used Chalice as an intermediate verification language in the context of their work on magic wands [17], but the resulting encoding, complicated by the fact that Chalice was not designed as an expressive and flexible intermediate verification language, did not perform well. VerCors subsequently migrated from Chalice to Viper [96], the intermediate verification language presented in Chapter 2 of this thesis, which resulted in a significant performance gain of up to 10x (according to personal communication with the authors); more

details are provided in Section 2.8. In recent work [2], VerCors was extended with support for the specification and verification of GPGPU programs, which commonly manipulate multi-dimensional arrays. It is unlikely that an encoding of their work into Chalice would have been possible because Chalice — similar to most existing verifiers for permission logics, but unlike Viper (see Section 4) — lacks adequate support for specifying programs that manipulate random-access data structures; support that we seek to provide.

Dynamic Frames

In dynamic frames [71], each operation specifies the set of heap locations it reads and writes by means of a *read*, respectively, a *write frame* (or *region*), often represented (in tools) as a set-typed specification variable. The variable's value can change dynamically, which accounts for dynamic changes of data structures, for example, by adding or removing objects, including the potential (de)allocation of the involved objects.

In contrast to separation logic, dynamic frames does not enforce separation; instead, specifications need to explicitly constrain frames to be (and to remain) disjoint. This approach makes dynamic frames well-suited for specifying examples which require structure sharing, for example, lists with shared tails, and iterators. It is unclear, however, how to extend the theory of dynamic frames to reasoning about concurrent programs — which we aim to verify — exactly because frames are not necessarily disjoint. Consequently, verifiers based on dynamic frames such as Dafny [82] and KeY [1] support reasoning only about sequential programs. Another potential disadvantage of automated verifiers for dynamic frames is their strong dependency on solvers for set constraints.

Region logic [5] is a variant of dynamic frames, and as such also limited to sequential programs. In an experiment, several examples of programs with region-logic-based specifications have been verified via a manual encoding into Boogie. In other work, (semi)-decision procedures for fragments of region logic have been devised and implemented as an extension of an SMT solver [110]. To the best of our knowledge, however, no automated verifier based on region logic exists.

Jahob [137] is a verification system for sequential Java programs (whereas we aim at verifying concurrent programs) annotated with specifications in higher-order logic. As in dynamic frames, operations need to specify frames of locations they read and write via specification variables. Jahob achieves integrated reasoning in the spirit of Nelson-Oppen [97] by splitting the resulting verification conditions into simpler sub-formulas that fall into specific logical fragments. Jahob then attempts to prove each sub-formula by applying a number of specialised reasoning engines such as individual decision procedures, SMT solvers, automated theorem provers, and ultimately interactive theorem provers. The need for manual interaction constitutes an automation limitation that we essay to overcome.

Ownership

In ownership-based verification [93], each object in the heap is notionally *owned* by at most one other object, its *owner*; this restricts the object graph to a forest of ownership trees. Each owner indirectly represents the set of objects it transitively owns, and two such sets are guaranteed to be disjoint if their respective owners are different, and if neither (transitively) owns the other. To enable reasoning about the effects of operations (on the heap), each operation lists the objects it potentially reads, respectively, writes; and to support information hiding (and to specify operations that potentially affect a statically unbounded number of objects), each listed object abstracts over all objects it transitively owns. Framing can now be achieved by proving that operations affect disjoint sets of objects. Ownership can also be used to address

other common verification tasks, for example, proving termination; a good overview of the field of ownership-based verification is given in [40].

The main disadvantage of ownership-based verification is its restriction to hierarchical data structures, which significantly complicates reasoning about non-hierarchical structures such as doubly-linked or cyclic lists, arrays and graphs. Due to the lack of a strict ownership hierarchy, such structures can only be represented as a "flat" set of peers having the same owner, in which case ownership no longer provides (noteworthy) support for framing. Another challenge for ownership-based verification is ownership transfer, which arises, for example, when two linked lists are concatenated, or when a list is split into sub-lists. Ownership transfer can be supported [85], but imposes visibility restrictions on objects for which ownership transfer should be allowed, which can, for example, complicate the verification of library code and thus impedes modular verification. A third challenge for ownership systems are the specification of partial data structures: for example, a tree that has a "hole" because the corresponding sub-tree is currently owned by another object, but whose original shape can be restored by "plugging" the missing part back in. To avoid the need for expressing reachability, ownership systems typically do not provide a way of denoting the set of all objects *transitively* owned by a given object, which complicates the specification of aforementioned partial structures, where it would be necessary to state that the node pointing into the "hole" is transitively owned by the root.

Existing ownership-based verification systems usually implement an object invariant methodology that builds on top of ownership, and that uses the enforced hierarchical object structure to determine the objects whose invariant is potentially broken by an operation, respectively, when it is sound to assume an object's invariant. A potential advantage of using invariant methodologies for front-end specifications is that the invariant protocols can be tailored towards specific classes of problems, which can reduce the necessary specification overhead. A conceptual disadvantage of invariant methodologies, however, is the strict enforcement of the respective invariant protocol, which complicates the specification of programs that do not (always) adhere to the protocol.

The first verifier that implemented ownership-based verification is Spec# [7, 85], which was designed for reasoning about sequential C# programs that modify hierarchical data structures; in follow-up work, SpecLeuven [66] extended Spec# to support synchronised concurrency. VCC [38] and AutoProof [104] subsequently improved over Spec# in terms of expressiveness, but specialised in orthogonal directions: VCC was designed for reasoning about low-level C code with fine-grained concurrency such as lock-free data structures built on top of primitive atomic operations, whereas AutoProof focuses on *considerate* inter-object collaboration [127], for example, as used in the composite design pattern, where other objects' invariants can be broken temporarily if the objects are notified appropriately.

Other Approaches

Matching logic [111, 113] is a logic for reasoning about program configurations by specifying structural patterns that are matched against configurations. The logic is parametric with respect to a programming language and its operational semantics, in particular, with a model for program configurations. By choosing an appropriate configuration model, one can, for example, obtain an instance of matching logic which corresponds to separation logic. Pairs of patterns express *reachability rules*, which are used to specify reachability between configurations matching the involved patterns, which in turn allows the encoding of Hoare-style pre-/postconditions and invariants.

MatchC [125], a verifier for a subset of C, is based on matching logic and implemented on top of a rewrite system for the operational semantics and their symbolic execution, and another rewrite system for entailment checking that can also query an SMT solver. MatchC can reason about sequential programs that manipulate linked lists, trees and

graphs, but support for these data structures, as well as for mathematical structures such as sequences, is built into the underlying rewriting systems, and reasoning about additional structures requires adding rules to the rewriting systems. In contrast, we aim to develop a verification infrastructure that is extensible on the level of the intermediate verification language. To the best of our knowledge, MatchC does not support concurrent programs, and it is unclear if a verification approach based on operational semantics can be extended to reason about important properties of concurrent programs such as deadlock freedom, which is one of our goals. Fractional permissions, an important feature of permission logics which enables shared read access, are also not supported by MatchC.

Numerous automated tools for reasoning about heap-manipulating programs are based on static analyses [124, 115, 46, 13, 32, 34, 69]: the degree of automation that such tools achieve is in general very high, but none can modularly reason about the kind of complex properties that are necessary for full-functional verification of heap-manipulating concurrent programs.

Finally, various automated verifiers for reasoning about functional programs exist [73, 135, 123, 15], but they deliberately exclude heap-manipulating programs.

1.3 Contributions

The main contribution of this thesis to the field of modular, automated and full-functional verification of heap manipulating concurrent programs is the introduction of Viper [96], the first verification infrastructure tailored to permission-based reasoning. The thesis spans the core of the infrastructure: an expressive intermediate verification language that facilitates the encoding of different programming languages and specification styles, and a verifier for the intermediate verification language that exhibits predictable and good performance. Specifically, the thesis makes the following four contributions:

1. Viper: An intermediate verification language for permission-based reasoning

The Viper language, described in Chapter 2, is an expressive intermediate verification language in the spirit of the Boogie language [6], specifically designed with the goal of facilitating the development of new front ends that use permission-based specifications to reason about race-free concurrent programs. To achieve this goal, Viper offers (1) a unique combination of features that have been proven useful in existing verification approaches and tools, (2) support for advanced permission-logic-features that have been successfully used in by-hand proofs, but that so far were not supported by automated verifiers; and (3) several novel verification constructs that further increase the expressiveness of the language.

The language offers a level of abstraction that is expressive enough to encode and specify a wide range of programming language features, in particular, concurrency synchronisation features, but that preserves sufficient structural and control flow information to permit efficient and precise back ends, and to be well-suited for manually encoding examples. The language is based on implicit dynamic frames [121] and integrates typical permission logic features, such as arbitrary recursive predicates, with specification features more commonly used in other verification approaches, for example, quantifiers and heap-dependent abstraction functions. Mathematical structures that are often used in specifications (sets, sequences and multisets) are natively supported, additional structures can be defined via uninterpreted functions and appropriate axioms. Viper's permission model is rich enough to express classical fractional permissions [25], but also approaches that are based on constraining symbolic read permissions [58, 28].

In addition to features of permission logics commonly supported in automated verification (such as predicates and fractional permissions), the Viper language also provides direct support for general *iterated separating conjunctions* and *magic wands* (described below), which enables the encoding of different (permission-based) specification styles.

Viper also offers several novel language features that facilitate the development of front ends: aforementioned rich permission model; permission-aware analogues of `assume`/`assert` statements, which enable the encoding of various sequential and concurrent programming features, and a generalisation of `old` expressions for encoding two-state invariants (particularly useful for specifying concurrency synchronisation); and a generalisation of asymmetric specifications such as free pre- or postconditions, which facilitates the integration of properties obtained by additional meta-reasoning, for example, soundness proofs, into the encoding.

2. Silicon: Symbolic execution for Viper

The Silicon verifier, described in Chapter 3, is a symbolic execution engine for Viper programs. Symbolic execution is the predominant implementation technique for permission logic verifiers, and Silicon extends work by Berdine et al. [11] (Smallfoot), respectively, Smans et al. [119] (VeriCool) on symbolically executing programs with permission-based specifications.

Silicon includes contributions that arise from the need of having to support Viper's rich set of features, which includes, for the first time (in a symbolic execution engine), the handling of quantifiers over heap-dependent expressions, sound support for permission introspection features, and a novel technique for handling heap-dependent abstraction functions that enables important completeness improvements (with respect to VeriCool).

Another contribution of the work on Silicon is a novel classification of aliasing-related incompletenesses commonly exhibited by automated verifiers based on Smallfoot-style symbolic execution, and a technique for systematically reducing the number of situations in which such incompletenesses can arise, without noticeably degrading performance.

Silicon generally exhibits good performance, which is stable across (small) modifications of the input, and independent of the verification result (success or failure).

3. The first symbolic execution technique for iterated separating conjunctions

Recursive predicates are an adequate means of specifying statically-unbounded data structures that can be traversed in only one direction, such as singly-linked lists and trees, but they substantially complicate the specification of data structures that admit different access patterns, for example, doubly-linked lists, graphs and arrays. Various by-hand proofs involving such structures therefore used iterated separating conjunctions [108] instead of recursive predicates to specify the properties of interest.

This thesis presents (in Chapter 4) the first symbolic execution technique for general iterated separating conjunctions, which includes: an innovative representation of iterated separating conjunctions during the symbolic execution, an approach to framing expressions that depend on unbounded sets of heap locations as denoted by iterated separating conjunctions, and an implementation of the technique in Silicon that achieves good performance.

Iterated separating conjunctions are integrated with other important features of the Viper language, such as abstraction functions, predicates and permission introspection. This integration, and in general, support for iterated separating conjunctions in an intermediate verification infrastructure, enables new front ends that combine specification features in ways beyond the current state of the art.

4. The first automated support for magic wands

The fourth contribution of this thesis, described in Chapter 5, is the first automated support for magic wands [64], a feature of permission logics that expresses properties about hypothetical (future) developments of the program state. Magic wands have been used in various by-hand proofs, for example, to specify partial versions of data structures during ongoing traversals, to enforce their orderly modification, and to reason about synchronisation barriers.

The technical contributions that enable the first automated support for magic wands include: a novel representation of magic wands during the symbolic execution; algorithms that automate the tedious choice of a wand's footprint, which significantly reduces the user's annotation overhead; a language design that provides concise user annotations suitable for using magic wands (reasoning about which is undecidable without user guidance), and a set of heuristics that aim to infer these annotations; and an implementation of the technique in Silicon that achieves good performance.

Magic wands are integrated with all other important features of the Viper language, including iterated separating conjunctions. The possibility of practical tool support, enabled by this thesis, facilitates the development of new front ends, and enables the exploration of further applications of magic wands.

Chapter 2

The Viper Language

The vision behind the Viper verification infrastructure [96] is to establish a verification infrastructure similar to the successful Boogie infrastructure [6], but tailored to permission-based reasoning about race-free concurrent programs. Achieving this goal requires an *intermediate verification language* that, as discussed in Section 1.1, is flexible and expressive enough such that different programming languages and their features, as well as different (permission-based) specification styles, can be automatically encoded. In addition, the intermediate verification language should make it convenient to manually encode examples, which is useful for experimenting with potential encodings (rapid-prototyping), for educational purposes and for participating in verification competitions. In order to facilitate the integration of verification and static analyses, the intermediate verification language should also be suitable for common static analysis techniques such as abstract interpretation.

Figure 2.1 gives an overview of the Viper verification infrastructure. The top layer shows front ends that encode different programming and specification languages into the Viper intermediate verification language: a Chalice front end that encodes a substantial subset of the Chalice language [75]; a Python front end that is part of the SCION project [8] for developing a secure, next-generation internet architecture; and a Java and an OpenCL front end that are part of the VerCors project [16], developed at the University of Twente. Viper includes two automated verifiers (middle layer), one is implemented as a symbolic execution engine, the other one generates verification conditions via Boogie. Both verifiers ultimately use the Z3 SMT solver [92] to discharge proof obligations. The verifiers expect fully-specified programs annotated with, for example, pre- and postconditions, and loop invariants. While we expect that the majority of these specifications are generated by front ends, we also believe that it is possible to infer certain specifications, in particular permission-related ones. To this end, Viper includes an inference engine that can already infer basic, permission-related specifications. Inferring stronger properties is work in progress, with the ultimate goal of mutually integrating verification and inference, for example, to enable the inference to learn from a failed verification attempt.

Chapter Overview

The work described in this chapter has in parts been published at VMCAI 2016, in the paper *Viper: A Verification Infrastructure for Permission-Based Reasoning* by Müller, Schwerhoff and Summers [96].

The chapter is structured as follows: Section 2.1 provides an overview of the Viper language and briefly introduces its language features. Afterwards, Section 2.2 motivates and illustrates permission-based reasoning, introduces a running example that is further developed in subsequent chapters, and demonstrates how a potential front end could encode high-level verification problems into Viper. Section 2.3 then demonstrates Viper features that enable reasoning about unbounded heap structures: recursive predicates, magic wands and quantified permissions (iterated separating conjunctions). The section focuses on the permission-related specifications, whereas

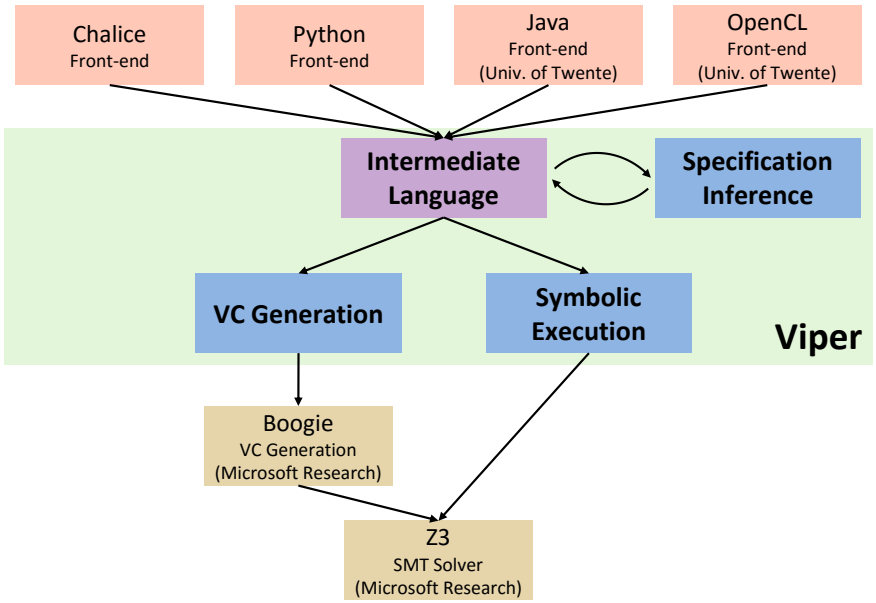

Figure 2.1: The Viper verification infrastructure: front ends (top layer) encode verification problems into the Viper intermediate verification language, the result of which can be verified using each of the two available automated verifiers (one based on symbolic execution, one on verification condition generation); both verifiers build upon existing reasoning engines (bottom layer). The infrastructure includes a basic specification inference, based on the abstract interpretation of Viper programs. The Viper intermediate verification language and the symbolic-execution-based verifier (Silicon, introduced in Chapter 3) constitute the applied results of this thesis.

Section 2.4 focuses on functional properties. Section 2.5 demonstrates Viper's support for axiomatising custom first-order theories, and Section 2.6 subsequently discusses intricacies of automatically reasoning about theories with quantifiers. Section 2.7 presents further details on Viper's permission model, before the chapter concludes with an empirical evaluation of Viper as an intermediate verification language and with a discussion of related work in Section 2.8, respectively, Section 2.9.

This chapter introduces the Viper language and demonstrates its suitability as an intermediate verification language. The formal treatment of Viper is postponed until Chapter 3, which presents symbolic execution rules for Viper (that can be understood as an implementation of a strongest postcondition calculus) and discusses well-formedness conditions (with the exception of *self-framingness*, discussed in Section 2.2.1, an important property of specifications based on implicit dynamic frames [121]).

2.1 An Overview of the Viper Language

The Viper language, whose grammar is shown in Figure 2.2, is an expressive intermediate verification language whose design has been influenced by a variety of existing languages, most notably, by Boogie [6], Chalice [84] and Dafny [82]. The language is imperative and has a built-in notion of a program heap and of permissions to memory locations, and front ends therefore do not need to encode these key concepts. This substantially simplifies the development of front ends that use permission-based specifications, and facilitates the development of efficient and precise back ends. Viper's assertion language is based on implicit dynamic frames [121], and thus provides accessibility predicates of the shape `acc(`e_1`.`f`,` e_2`)`, denoting e_2 (fractional) permissions to the memory location identified by $e_1.f$; more details about Viper's heap and permission model follow.

program	::=	$\overline{decl}$
decl	::=	*fielddecl* | *preddecl* | *funcdecl* | *methdecl* | *domdecl*
fielddecl	::=	`field` f`:` T
preddecl	::=	`predicate` *pred*`(`$\overline{x\texttt{:}\ T}$`)` `{`$a$`}`
funcdecl	::=	`function` *func*`(`$\overline{x\texttt{:}\ T}$`):` T `requires` a `ensures` e `{`e`}`

methdecl	::=	`method` *meth*`(`$\overline{x\texttt{:}\ T}$`)` `returns` `(`$\overline{y\texttt{:}\ T}$`)`
		`requires` a
		`ensures` a
		`{`*stmt*`}`

stmt	::=	`var` x`:` T | x `:=` e | x `:=` `new(`$\overline{f}$`)` | $\overline{x}$ `:=` *meth*`(`$\overline{e}$`)` | x`.`f `:=` e | *stmt*`;` *stmt* | `label` *lb* | `goto` *lb* | `inhale` a | `exhale` a | `assert` a | `fold acc(`*pred*`(`$\overline{e}$`),` e`)` | `unfold acc(`*pred*`(`$\overline{e}$`),` e`)` | `if (`e`) {`*stmt*`} else {`*stmt*`}` | `while (`e`) invariant` a `{`*stmt*`}` | `constraining (`$\overline{x}$`) {`*stmt*`}` | `package` a `--*` g | `apply` a `--*` a
a	::=	e | `acc(`e`.`f`,` e`)` | `acc(`*pred*`(`$\overline{e}$`),` e`)` | a `&&` a | e `?` a `:` a | `[`a`,` a`]` | a `--*` a | `forall` x`:` T `::` e `==>` `acc(`e`.`f`,` e`)` | `forall` x`:` T `::` e `==>` `acc(`*pred*`(`$\overline{e}$`),` e`)`
e	::=	*op*`(`$\overline{e}$`)` | e`.`f | *func*`(`$\overline{e}$`)` | e `?` e `:` e | `old[`*lb*`](`e`)` | `unfolding acc(`*pred*`(`$\overline{e}$`),` e`)` `in` e | `perm(`e`.`f`)` | `perm(`*pred*`(`$\overline{e}$`))` | `forall` $\overline{x\texttt{:}\ T}$ `::` `{`e`}` e | `exists` $\overline{x\texttt{:}\ T}$ `::` `{`e`}` e | `forperm[`f`]` `x:` T `::` e
g	::=	a | `folding acc(`*pred*`(`$\overline{e}$`),` e`)` `in` g | `unfolding acc(`*pred*`(`$\overline{e}$`),` e`)` `in` g | `packaging` a `--*` g `in` g | `applying` a `--*` a `in` g

domdecl	::=	`domain` *dom*`[`$\overline{T}$`]` `{`
		$\overline{dfndecl}$
		$\overline{daxdecl}$
		`}`

dfndecl	::=	`function` *func*`(`$\overline{x\texttt{:}\ T}$`):` T
daxdecl	::=	`axiom` *ax* `{`e`}`

Figure 2.2: The Viper intermediate verification language. Overlining denotes repetition. Metavariable f ranges over fields, T over types, x, y over variables, *pred* over predicates, *func* over functions, *meth* over methods, *lb* over labels, *dom* over domains, and *ax* over axiom names. Types T are either the built-in types `Int`, `Bool`, `Ref`, `Perm`, `Set[`T'`]`, `Seq[`T'`]` and `Multiset[`T'`]`, or *dom*$[\overline{T'}]$, the type induced by the corresponding domain declaration. All top-level declarations (fields, predicates, abstraction functions, methods and domains) are global, as are domain functions and axioms (albeit nested inside domain definitions). The bodies of predicates, functions and methods are optional. *op*$(\overline{e})$ denotes any heap-independent function of arbitrary arity, including literals: (dis)equality; arithmetic, relational and boolean functions; set, sequence and multiset functions; and domain functions.

Assertions in Viper are either *spatial* or *pure*: a *spatial* assertion [99, 108] describes the shape of the heap, whereas a *pure* assertion constrains values. Note that pure assertions may be heap-dependent; this distinguishes implicit dynamic frames from separation logic [99, 108]. In Viper, accessibility predicates (to locations $e.f$ or predicate instances *pred*$(\overline{e})$), magic wands (a `--*` a) and quantified permission assertions

(`forall` x`:` T `::` e `==> acc(...)`) are *spatial* assertions, and compound assertions (for example, conjunctions or conditionals) that nest a spatial assertion are themselves spatial. All other assertions are pure, and Viper does not differentiate between pure assertions and expressions (shortly discussed in more detail): all expressions (syntactic category e in Figure 2.2) can be used in assertions and in code (for example, on the right of an assignment or as the condition of an `if` statement).

To achieve modular verification, Viper provides the declaration of methods and (heap-dependent) functions with pre- and postconditions; each method and function is verified in isolation (with respect to its specification). In addition to methods and functions, the language supports the unrestricted declaration of (recursive) predicates for specifying heap-implemented data structures, and of custom first-order theories via *domains*. The latter enable front ends to provide their own mathematical vocabulary, for example, to encode algebraic data types, or to preserve type system information. Viper does not have a notion of classes: methods, functions and predicates do not take implicit receiver arguments — these have to be declared explicitly (as in for example, Python) — which makes the language suitable for encoding procedural and object-oriented languages.

Viper differentiates between an instance, respectively, application of a (recursive) predicate, respectively, function, and the corresponding body: that is, it implements an *iso-recursive* semantics [126]. This avoids non-terminating SMT solver queries (unlike VeriCool [121]), but requires additional annotations that direct the verifier accordingly: `unfold` and `fold` statements exchange a predicate instance for its body, and vice versa; `unfolding` expressions perform a temporary unfold within in the scope of the nested expression. Other verifiers, for example, VeriFast [67], require similar annotations. To avoid the need for similar annotations for (un)rolling function definitions, Viper's verifiers implement a technique by Heule et al. [59] that links the (un)rolling of functions to the (un)folding of predicates.

A similar problem arises in the context of universal quantifiers (which Viper supports), an important and concise means of expressing an unbounded number of constraints, which is often necessary for achieving full-functional verification. To prevent infinite instantiation chains (known as *matching loops*), SMT solvers typically support syntactic *matching triggers* [39] to control possible instantiations. Viper therefore provides syntax that allows users to specify such triggers (as does, for example, Boogie), but it also attempts to infer suitable triggers (more details about triggers are provided in Section 2.6; a comparison with VeriFast, which takes a different approach, is provided in Section 3.7.3).

Viper, being an intermediate *verification* language, does not necessarily have to be executable, and thus does not distinguish between real code and *ghost code* (for example, methods that encode lemmas, or conditionals that depend on permission values). Enforcing a strict separation of real and ghost code can potentially complicate encodings (see also [47]), for example, because data types or code needs to be duplicated (to exist as a real and as a ghost version). In contrast, verification systems for programs that are intended to be executed typically enforce such a separation, so that ghost code can be erased during compilation without affecting the operational semantics. The erased program — but not the ghost code — can then be regularly tested and debugged. Examples of such systems are VeriFast, which reasons about C and Java programs, and Dafny, whose programs can be compiled to C# code.

To enable the encoding of a wide range of imperative source languages with different type systems and potentially different notions of objects and classes, the Viper language supports object references, and globally declared fields that can be accessed from each object, but no notion of classes, inheritance or sub-typing; these concepts need to be encoded (for example, via custom domains), if necessary. In such an encoding, it is the responsibility of the front end that generates a Viper program to ensure that fields are only accessed from appropriate receiver objects.

In addition to reference types (`Ref`), Viper supports booleans (`Bool`), mathematical integers (`Int`, corresponding to $\mathbb{Z}$) and polymorphic sets, sequences and multisets (`Set[`T`]`, `Seq[`T`]`, `Multiset[`T`]`). Permission amounts (`Perm`, corresponding to $\mathbb{R}_{\geq 0}$) are another built-in type, which allow the use of classical fractional permissions [25], but also enable the encoding of approaches that constrain abstract read permissions [58, 28] (in combination with `constraining` blocks; more details are provided in Section 2.7). Moreover, each domain declaration induces a corresponding type, which can be used in all places where built-in types can be used.

For each built-in type, Viper's expression language includes the operations one would commonly expect for that type, including: arithmetic, relational and boolean operators, set contains and cardinality, sequence lookup and concatenation, and multiset cardinality. The built-in collection types sets, sequences and multisets (and the corresponding operations) could equally well be defined via Viper's domains (which are expressive enough), but including them in the language's core enables specialised treatment in back ends (such as static analyses).

As (sequential) control flow statements, the Viper language supports conditional statements, loops (with explicitly provided invariants) and method calls (which are reasoned about in terms of the callee's specifications). `goto` statements are supported as well, and can be used to encode other control flow operations such as exiting loops early and raising exceptions. Viper only allows `goto` statements that yield reducible control flow graphs, and each loop in such a graph must be specified by an invariant. The support of structured control flow statements (in addition to `goto` statements) facilitates the development of efficient and precise back ends, and makes the language well-suited for manually encoding examples (as does the built-in support for often-used specification means such as sets and sequences).

Encoding a program (and its specifications) into a permission-based verification language commonly requires indicating points in the program at which permissions are (conceptually) *transferred* between units of the modular verification: typically, one unit gives up the permissions which another unit gains. In order to encode permission transfers, Viper provides `inhale` and `exhale` statements that are permission-aware analogues of `assume` and `assert`, respectively, statements (as, for example, provided by Boogie): inhaling an assertion can be understood as gaining all permissions included in the assertion, and assuming all pure sub-assertions; exhaling is the corresponding dual operation, and entails giving away permissions and checking pure assertions.

These two powerful statements enable, for example, the encoding of a variety of concurrency features (for race-free concurrency) which can be understood in terms of transferring permissions, such as fork-join parallelism and locks. For example, forking a thread can be modelled by exhaling permissions (to locations potentially modified by the new thread), and joining a thread by inhaling. Similarly, acquiring a lock (specified by a lock invariant) that protects a resource, for example, a memory location or a file handle, can be modelled by inhaling permissions to that resource, and releasing the lock by exhaling. Examples of such encodings are shown in Section 2.2.3.

The semantics of the Viper language do in general not ensure soundness, and users can make arbitrary assumptions, including unsound ones such as inhaling too many permissions. This is a deliberate choice, made to enable encodings whose soundness depends on external arguments that are not part of the encoding itself, or that are otherwise regarded as out-of-scope (for example, invoking system functions or other native code). This freedom is essential for facilitating the development of different front ends, and commonly used in verification; systems such as Boogie, VeriFast and Dafny [82] include similar, potentially unsound, features. Other Viper features that increase the language's expressiveness, but are potentially unsound, are inhale-exhale assertions and abstract methods and functions (which have no bodies, but may have unchecked postconditions). For example, inhale-exhale assertions, written `[`a_1`,` a_2`]`, are a generalisation of Boogie's free pre- and postconditions: property a_1 is inhaled

```
1  struct cell {
2    int val;
3  };
4
5  void inc(struct cell* c) {
6    c->val = c->val + 1;
7  };
8
9  void client() {
10   struct cell* c1 = malloc (sizeof(struct cell));
11   struct cell* c2 = malloc (sizeof(struct cell));
12
13   c1->val = 0;
14   c2->val = 1;
15
16   inc(c1);
17
18   assert(c1->val == c2->val);
19 };
```

Listing 2.1: A first example illustrating modular verification: statically proving the client's assertion requires reasoning about the effects of the invocation of `inc` on `c1` and `c2`, which in general could be aliased.

(for example, at the beginning of a method body), but a_2 is exhaled (for example, at method call-site).

2.2 Permission-Based Reasoning in Viper

Consider the simple C program shown in Listing 2.1, which we will use to motivate the concepts of permissions and permission transfers, which are crucial for achieving modular verification. The program declares a record (a struct) `cell` with a single `int`-typed field `val`, alongside a method (a function, in C vocabulary) `inc` that increments the value of a given cell by one. The program also includes a client that instantiates two cells, initialises their values to `0` and `1`, respectively, invokes `inc` to increment the first cell's value, and finally asserts (with a runtime check) that the two cells hold the same value.

In order to statically reason about the example in a modular way, the client needs to conclude that the invocation of `inc(c1)` does indeed increase `c1.val` by `1`, and in addition, that the invocation does not modify `c2.val`. The latter is known as *framing*: the client must be able to *frame* the knowledge that `c2.val` has the value `1` across the invocation of `inc(c1)`. This entails reasoning about aliasing between `c1` and `c2` (and in general, everything reachable from them). In the given program, `c1` and `c2` cannot be aliases because they each point to a distinct, newly allocated object.

Permissions are a well-established technique for enabling framing: accessing a heap location, that is, reading from or writing to it, requires the corresponding permission. Per location, the permission to access it is exclusive (unique) and a method that needs to access a location needs to obtain the required permission, for example, from its caller. Methods can require permissions via appropriate assertions included in their preconditions, which in turn enables callers to deduce which heap locations a method call may affect. Upon method invocation, the caller (conceptually) *transfers* all necessary permissions to the callee, and upon termination, the callee transfers permissions back to the caller (via its postcondition). At first, we do not distinguish between reading from and writing to a location: in both cases, the same exclusive permission is required. However, we will soon (in Section 2.2.2) introduce *fractional permissions* [25] that enable differentiating between read and write permissions.

```
1  field val: Int
2
3  method inc(c: Ref)
4    requires acc(c.val)
5    ensures  acc(c.val)
6    ensures  c.val == old(c.val) + 1
7  {
8    c.val := c.val + 1
9  }
```

Listing 2.2: A Viper encoding of method `inc` from Listing 2.1 that demonstrates specifications with permissions and functional properties.

```
1  method client() {
2    var c1: Ref
3    var c2: Ref
4
5    c1 := new(val)
6    c2 := new(val)
7
8    c1.val := 0
9    c2.val := 1
10
11   inc(c1)
12
13   assert c1.val == c2.val
14 }
```

Listing 2.3: A Viper encoding of the `client` method from Listing 2.1. Due to the specifications of `inc` shown in Listing 2.2, the final assertion succeeds.

Method `inc` from Listing 2.1 can be encoded in Viper as shown in Listing 2.2: the globally declared field `val` of type `Int` corresponds to the single field of the `cell` record, and method `inc` corresponds to the C method of the same name. `inc` takes a single `Ref`-typed argument which points to the `cell` object whose value is to be incremented. The precondition of method `inc` states that the method requires permission to write to location `c.val`, which is denoted by the *accessibility predicate* `acc(c.val)`. The first postcondition states that the permission will be transferred back to the caller, allowing it to read the location's new value. The second postcondition (all postcondition clauses are conjuncted, and likewise for preconditions) states that `inc` indeed increments the value of `c.val` by one: an *old* expression `old(`e`)` denotes the value of e in the prestate, that is, in the state in which the method was invoked. An `old` expression only affects heap dereferences (such as `c.val`), but not local variables (such as `c` itself).

Listing 2.3 shows an encoding of the client from Listing 2.1 in Viper. The first two lines declare new local variables `c1` and `c2` of reference type, and the next two lines encode the allocation of the two `cell` records. The statement `c1 := new(val)` assigns a fresh reference to `c1`, which is assumed to be different from each already existing reference, and it adds permission to access `c1.val` to the current method context. Note that `new` is a statement and not an expression, because it is not referentially transparent. The general form of the statement is x `:= new(`$f_0, \ldots, f_{n-1}$`)`, which adds permission to access each field f_i of the (potentially empty) list of fields.

After assigning to the newly allocated heap locations `c1.val` and `c2.val` — which is possible because the required permissions have been obtained from the preceding `new` statements — `inc(c1)` is invoked. From a verification perspective, the invocation proceeds by *exhaling* `inc`'s precondition followed by *inhaling* the postcondition. Exhaling and inhaling can be understood as the permission-aware analogues of asserting

```
1 method copyAndInc(c1: Ref, c2: Ref)
2   requires acc(c1.val) && acc(c2.val)
3   ensures  acc(c1.val) && acc(c2.val)
4   ensures  c1.val == old(c1.val)
5   ensures  c2.val == old(c1.val) + 1
```

Listing 2.4: The specifications of a method that sets `c2.val` to `c1.val + 1`; the combination of permissions and the separating conjunction && prevents aliasing between `c1` and `c2` (fractional permissions are introduced later on).

and assuming assertions. Exhaling an assertion consists of removing all permissions that (conceptually) need to be transferred to the callee (in this case, `acc(c1.val)`), and of checking that all pure assertions hold. Accordingly, inhaling an assertion consists of adding permissions (here, to `acc(c1.val)`) and assumptions (here, that `c1.val == 1`) to the verification state. Losing permission to a location means that assumptions about the location's value can no longer be framed, that is, safely be retained. The location's value is therefore *havoced* by setting it to an arbitrary (unknown) value.

Holding on to the permission to `c2.val` when invoking `inc(c1)` enables the client to frame the assumption that `c2.val == 1` across the call. As a result, the final assertion succeeds: combining the assumption obtained from `inc`'s postcondition (about `c1.val`) with the assumption framed across the call (about `c2.val`) enables concluding that `c1.val == c2.val`.

As mentioned earlier, framing is in general complicated by the possibility of aliasing, as illustrated in Listing 2.4. Method `copyAndInc`, whose straight-forward implementation has been omitted, is intended to set `c2.val` to `c1.val + 1`, while leaving `c1.val` unchanged. The postcondition can only be satisfied if `c1` and `c2` are not aliased, which is guaranteed by the precondition: Viper's conjunction && denotes the *separating conjunction* from separation logic, the precondition therefore states that the method requires the exclusive permission to `acc(c1.val)` in addition to the exclusive permission to `acc(c2.val)`. This implies that `c1` and `c2` cannot be aliased because an exclusive permission cannot be held twice (which is equivalent to false).

The C program shown in Listing 2.1 allocated new memory via `malloc`, which was encoded in Listing 2.3 via Viper's `new` statement. C requires manual memory management — memory should be freed (deallocated), not *leaked* as done by the simple client shown in Listing 2.1 — but Viper does not provide a dedicated statement that corresponds to C's `free` operation. Memory that has been freed should no longer be accessed, which can be prevented in a permission-based verification by exhaling (discarding) the corresponding permissions, an operation that is supported by Viper via a dedicated statement that is presented in Section 2.2.3. When verifying programs that manually manage their memory, it is not only important to prove that freed memory is not accessed, but also that memory is actually freed, not leaked. In a permission-based encoding, the latter can be achieved by proving the absence of permission leaks. Viper does not prevent permissions from being leaked — a method may terminate without returning all held permissions to its caller — but its specification language is rich enough to encode such leak checks, as demonstrated in Section 2.2.3.

2.2.1 Self-Framing Assertions

Assertions that are used as specifications in order to achieve modular verification — pre- and postconditions, and loop invariants, but also predicate bodies (Section 2.3.1) and the specifications of heap-dependent functions (Section 2.4) — are typically exhaled (checked) in a context that is different from the context(s) in which they are

```
1  field data: Seq[Int]
2
3  define sorted(s)
4    forall i: Int, j: Int ::
5          0 <= i && i < j && j < |s|
6      ==> s[i] <= s[j]
7
8  method insert(this: Ref, elem: Int) returns (idx: Int)
9    requires acc(this.data) && sorted(this.data)
10   ensures  acc(this.data) && sorted(this.data)
11   ensures  0 <= idx && idx <= old(|this.data|)
12   ensures  this.data ==    old(this.data)[0..idx]
13                         ++ Seq(elem)
14                         ++ old(this.data)[idx..]
15 {
16   idx := 0
17
18   while(idx < |this.data| && this.data[idx] < elem)
19     invariant acc(this.data, 1/2)
20     invariant 0 <= idx && idx <= |this.data|
21     invariant forall i: Int ::
22                       0 <= i && i < idx
23                   ==> this.data[i] < elem
24   {
25     idx := idx + 1
26   }
27
28   this.data :=    this.data[0..idx]
29                ++ Seq(elem)
30                ++ this.data[idx..]
31 }
```

Listing 2.5: A sorted list of integers, implemented via immutable sequences. The example will gradually be modified as we motivate and introduce additional Viper features.

inhaled (assumed): a method postcondition, for example, is exhaled at the end of the corresponding method, and inhaled at different call sites.

In order to (modularly) ensure that such assertions are always well-defined, in particular, that the context holds sufficient permission in order to evaluate all heap dereferences, Viper requires such assertions to be *self-framing* [70, 102]. Intuitively, an assertion is self-framing if it requires permissions to at least those locations that it reads. For example, `acc(x.f) && x.f == 0` is self-framing, as is `acc(x.f)`, but `x.f == 0` is not. Note that separation logic assertions, in contrast to assertions using implicit dynamic frames, are self-framing by construction since separation logic's points-to predicate couples dereferencing the heap with requiring the necessary permissions.

Viper, similar to other automated verifiers based on permission logics, checks self-framingness of an assertion from left to right, which simplifies the implementation but restricts the order in which conjuncts can occur in assertions: `acc(x.f) && x.f == 0` is accepted as self-framing, whereas `x.f == 0 && acc(x.f)` is rejected. This restriction is not a problem in practice, however.

2.2.2 Running Example

Listing 2.5 shows the specification and implementation of a sorted list of integers. The sorted list will serve as the running example for the rest of the chapter, and its

implementation and specification will be modified gradually as new Viper features are motivated and introduced.

In the initial version, the elements of a list are represented as a mathematical sequence (a value type), which is stored in a list's `data` field of type `Seq[Int]`. Method `insert` inserts a new integer value (input parameter `elem`) into a list (input parameter `this`), and returns the index at which the given element has been inserted (output parameter `idx`).

Viper methods can have an arbitrary number of input and output parameters; the `returns` keyword and the output parameter list can be omitted from the method signature if the method does not return any results (corresponding to `void` in C or Java). Input parameters are immutable (corresponding to `final` parameters in Java), that is, they cannot be assigned to in the method body. Output parameters are in scope throughout the whole method body and can be assigned to an arbitrary number of times (as illustrated by Listing 2.5). Viper does not have a dedicated `return` keyword.

The precondition of `insert` states that the method requires permission to `this.data` and that the data has to be sorted, denoted by `sorted(this.data)`, and the first postcondition ensures that the permission is returned to the caller and that the data remains sorted. `sorted` is a parametric macro and every occurrence of the macro will be expanded to the macro's definition from line 4: a pairwise comparison of all elements in the sequence. The second postcondition ensures that the return value `idx` is a valid index into the (old) sequence of elements, and the third postcondition ensures that the inserted element has indeed been inserted at position `idx` (and that the sequence of elements has not changed otherwise). The sequence expression `old(this.data)[0..idx]` selects the subsequence from and including position `0` to but excluding position `idx`, and similarly, `old(this.data)[idx..]`, selects the subsequence from position `idx` to the end of the sequence, that is, to `|old(this.data)|`.

The method implements the insert operation by iterating over the sequence of elements to determine the appropriate insert position, followed by constructing a new sequence that corresponds to inserting the new element into the sequence of elements at that position. The second and third loop invariant state the obvious functional properties, but the first postcondition needs explaining: the accessibility predicate `acc(this.data, 1/2)` uses fractional permissions [25] to express that the loop will only *read* `this.data`. At the point where the loop is reached, the method execution holds write permission to `this.data`, one half of which is (temporarily) lost when the loop invariant is exhaled. Since the method execution holds on to the other half, however, it can (modularly) conclude that the loop does not modify `this.data`, which is crucial for establishing the method postcondition.

In Viper, a fractional permission to a heap location is a rational value from the closed interval $[0 \ldots 1] \subset \mathbb{Q}$, where a value of 1 allows writing to the heap location, any value $0 < p < 1$ allows reading from the location, and a value of 0 allows neither reading nor writing, that is, it denotes the absence of permissions. The permission constant `write` denotes write permissions (as do the literals `1/1`, `2/2`, etc.), the constant `none` denotes no permissions (as do `0/1`, `0/2`, etc.). The previously used accessibility predicate `acc(this.data)` is syntactic sugar for `acc(this.data, write)`. Additional details about Viper's permission model are given in Section 2.7. In particular, about *abstract read permissions* that avoid the need for choosing arbitrary concrete fractions such as `1/2` in cases where heap locations are only read.

In subsequent sections, we will improve the encoding of the sorted list in the following ways: (1) we will encode two more-realistic implementations, one based on linked-list nodes, and one based on arrays; (2) we will introduce abstraction mechanisms that enable the specifications to abstract over details of the implementation. Before presenting these improvements, however, we will introduce a client of the sorted list that illustrates how high-level programming concepts that are not directly supported by Viper can be encoded.

```
1  class client {
2    @GuardedBy(this) List list;
3
4    @MonitorInvariant(this,
5      forall int i, j ::
6            0 <= i && i < j && j < |list.data|
7        ==> list.data[i] <= list.data[j])
8
9    @MonitorInvariant(this,
10      old(|list.data|) <= |list.data|)
11
12   synchronized void client(int e1, int e2) {
13     list.insert(e1);
14     list.insert(e2);
15     assert(list.data[0] <= list.data[1]);
16     changed = true;
17   }
18 }
```

Listing 2.6: A Java client of the sorted list, including annotations that restrict modifications of the list: threads may only access the list after acquiring the monitor corresponding to `this`, which may only be released if the list is sorted, and not shorter compared to when the monitor was acquired.

2.2.3 Encoding High-Level Concepts

Listing 2.6 shows a client of the previously discussed list, implemented in a Java-like language. The client stores a reference to the list in its `list` field. The client is thread-safe and uses coarse-grained locking to protect its list instance: the Java annotation `@GuardedBy(this)`[1] indicates that the guarded field `list` should only be accessed after object `this` has been locked. Phrased in terms of permissions, `@GuardedBy(this) List list` can intuitively be understood as obtaining permission to access the list from locking `this`.

The (hypothetical) annotation `@MonitorInvariant(this, ...)` expresses a monitor invariant of `this`. The first occurrence is a one-state invariant stating that the list is sorted. The second occurrence is a two-state invariant, stating that the list may not be shorter by the time the guarding monitor is released. In the latter invariant, an `old` expression is used to refer to the state in which the monitor was acquired.

Viper does not natively support the concept of locking or of monitor invariants. In terms of verification, acquiring a monitor can be encoded by inhaling the monitor invariant, and releasing a monitor can be encoded by exhaling it. In analogy to `assume` and `assert` statements from guarded-command languages such as Boogie, Viper therefore provides `inhale` and `exhale` statements that can be used to encode a wide range of high-level concepts, including monitor invariants and fork-join concurrency.

Listing 2.7 shows an encoding of the Java client in Viper: the program reflects acquiring and releasing the monitor, and the client's modification of the list. The initial encoding will later on be extended with a check that asserts deadlock-freedom, and a leak check that ensures that the client releases the monitor before terminating.

The `inhale` statement at the beginning of the method (line 5) encodes acquiring the monitor. The first line of the inhale encodes the `@GuardedBy` annotation: it adds permissions for accessing the guarded field and the list content. The second line of the inhale corresponds to the first monitor invariant; the second monitor invariant has been omitted because it is a two-state invariant (since it is reflexive, it could be inhaled as a stuttering invariant, as done by VCC [37]). The `exhale` statement at the

[1] The first two chapters of [45] provide a nice overview of potential semantics of the `@GuardedBy` annotation.

```
1  field list: Ref
2
3  method client(this: Ref, e1: Int, e2: Int) {
4    /* Acquire the monitor */
5    inhale acc(this.list) && acc(this.list.data)
6        && sorted(this.list.data)
7
8    label acq
9
10   var tmp: Int
11   tmp := insert(this.list, e1)
12   tmp := insert(this.list, e2)
13
14   assert this.list.data[0] <= this.list.data[1]
15
16     /* Release the monitor */
17   exhale acc(this.list) && acc(this.list.data)
18       && sorted(this.list.data)
19       && old[acq](|this.list.data|) <= |this.list.data|
20 }
```

Listing 2.7: A Viper encoding of the list client from Listing 2.6. Acquiring the monitor is modelled via an `inhale` statement, releasing it via an `exhale`. Two-state properties can be expressed via labelled `old` expressions.

end of the method (line 17) encodes releasing the monitor again. The third line of the statement checks the two-state invariant, it uses a *labelled* `old` expression to refer to the state right after acquiring the monitor. Labels are set by the `label` statement, they also serve as `goto` targets. Labels and labelled `old` expressions are generally useful to encode multi-state properties, for example, termination measures.

In addition to encoding high-level concepts, `inhale` and `exhale` statements could also be used to encode features that are natively supported by Viper such as method calls or loop invariants. Similarly, `if` statements could be replaced by combining `inhale` and `goto` statements (which Viper supports, for example, to encode exception handling). This would simplify the core language, but we believe that static analysers will benefit greatly from the preservation of (some) structural information that was present in the source program.

Encoding Leak Checks

We will now extend the encoding of the list client such that it asserts that the monitor can only be released if it has previously been acquired, and that the monitor actually has been released by the time the client terminates. The latter is an instance of a leak check; a similar encoding can be used to check for memory leaks, for example, in the context of the previously discussed C example from the beginning of Section 2.2.

Listing 2.8 shows the extended encoding. A new field, `held` (of arbitrary type since the field's value never matters), is introduced to model that a method execution has acquired a monitor: holding permission to *o*`.held` corresponds to having acquired the monitor of object *o*. Consequently, permission to `this.held` is inhaled at the point where the monitor is acquired by the client, and the permission is exhaled again when the monitor is released. If the client tried to release the monitor without acquiring it first, the client would not have obtained permission to `this.held`, and trying to exhale the latter would therefore fail.

The newly added postcondition of the client encodes the desired leak check: the pure assertion `forperm[held] r :: false` corresponds to asserting that all references `r` for which the current state holds non-zero permissions to `r.held` satisfy `false`. The assertion rightfully passes because permission to `this.held` has been exhaled

```
1  ...
2  field held: Int
3
4  method client(this: Ref, e1: Int, e2: Int)
5    ensures [true, forperm[held] r :: false]
6  {
7    /* Acquire the monitor */
8    inhale ...
9        && acc(this.held)
10
11   ... // method body
12
13   /* Release the monitor */
14   exhale ...
15       && acc(this.held)
16 }
```

Listing 2.8: An extension of the list client: permissions to the `held` field model the ability to release a previously acquired monitor; whereas the postcondition models the client's obligation to release the monitor before terminating.

when the monitor has been released. The leak check would also pass if permission to `this.held` were transferred to the client's caller, which intuitively would make it an obligation of the caller to release the monitor. Such a leak check demonstrates that it is useful to assert or assume properties about the permissions currently held, without adding or removing permission. To support this kind of permission introspection, Viper provides the already used `forperm` expression, and the related `perm` expression, where `perm(`$o.f$`)` denotes the permission amount currently held for location $o.f$.

Similar to `forperm`, `perm` expressions can be used to inspect the currently held permission amounts. Consider, for example, some kind of resource system where different operations require and consume different resources. A given operation can be enabled by different resources, which in turn makes certain resources more valuable than others because they enable more operations. If these resources are encoded as permissions (similar to the use of `held` in Listing 2.8), `perm` can be used to ensure that less valuable resources are used up to perform an operation that can be enabled by different resources. Such an encoding has been used by Müller et al. [21, 89] to modularly prove finite blocking for non-terminating programs.

Note that the fragment of our logic that is obtained by removing `forperm` and `perm` is monotonic; using the latter features renders the logic non-monotonic, as discussed in the next paragraph.

Monitor leak checks must be performed *after* any remaining monitors have been transferred to the caller via the method's postcondition (and similar for other leak checks). The check can thus not be placed at the end of the method body, where it would be performed *before* exhaling the postcondition; it is placed as (the last) method postcondition instead. Inhaling the assertion at call-site would in general be unsound, however: it would yield the (potentially incorrect) assumption that the caller holds no monitors either. To prevent contradicting assumptions, the leak check is nested in an *inhale-exhale assertion*: an assertion of the form `[`a_1`,`a_2`]` is interpreted as a_1 when the assertion is inhaled, and as a_2 when the assertion is exhaled. In our example, a_1 is `true` (and a_2 is the leak check), which avoids the potential contradiction.

It is common for encodings of high-level verification techniques to contain asymmetries between the properties that are assumed and those that are checked. The monitor leak check is an example of a property that is checked, but not assumed. It is also common to assume properties that are justified by a different (possibly weaker or even vacuous) check together with an external argument provided by a type system, soundness proof or other meta-reasoning. For example, the following assertion

```
1  ...
2  field mu: Rational
3
4  method client(..., residue: Rational)
5    requires acc(this.mu, 1/2)
6    requires residue < this.mu
7    ensures  acc(this.mu, 1/2)
8    ...
9  {
10   /* Acquire the monitor */
11   exhale forperm[held] r :: r.mu < this.mu
12       && residue < this.mu
13   inhale ...
14
15   ... // method body
16
17   /* Release the monitor */
18   ...
19 }
```

Listing 2.9: Another extension of the list client: deadlock-freedom is guaranteed by enforcing a partial order according to which monitors can be acquired. Argument `residue` models the "largest" lock held by the current thread (an alternative modelling based on abstraction functions, see Section 2.4, is possible).

allows the verifier to use a quantified property in its direct form when assuming the property, and to use the premises of the corresponding inductive argument when proving the property:

```
[forall x: Int :: 0 <= x ==> P(x),
 forall x: Int ::
       (forall y: Int :: 0 <= y && y < x ==> P(y)) && 0 <= x
   ==> P(x)
]
```

Checking for Deadlock-Freedom

Acquiring monitors can result in a deadlock if one thread tries to acquire a monitor that has already been acquired by a second thread, which in turns tries to acquire a monitor that the first thread has already acquired. A well-known approach to avoiding such cycles and to preventing deadlocks is to establish a strict partial order between the monitors that need to be acquired, and to ensure that monitors can only be acquired in increasing order. Our encoding of this approach follows the encoding that Leino et al. used in the context of Chalice [84] to modularly verify the absence of deadlocks (for improved information hiding, see recent work by Jacobs et al. [65]). In this encoding, the position of each monitor object in the partial order is called a *waitlevel*, which is recorded in a (for our purposes immutable) field `mu` of rational type[2], and acquiring a monitor is only possible if the monitor's waitlevel is strictly greater than the waitlevel of each monitor that the current thread has already acquired.

Listing 2.9 shows how to extend the previous encoding to ensure deadlock-freedom. The first newly added precondition enables the client to read the monitor's waitlevel, and the `forperm` checks that the monitor's waitlevel is greater than the waitlevel of each monitor that has already been acquired — by the current method execution. The latter is important: since the verification is modular, the `forperm` is not guaranteed to range over all monitors that have been acquired by any method execution higher

[2] At the time of writing, `Rational` is just a type alias for `Perm`. For the discussed encoding, any type admitting a dense partial order suffices.

```
1  field data: Int
2  field next: Ref
3
4  predicate lseg(this: Ref, end: Ref) {
5    this != end ==>
6         acc(this.data) && acc(this.next)
7      && acc(lseg(this.next, end))
8  }
9
10 field head: Ref
11
12 predicate list(this: Ref) {
13   acc(this.head) && acc(lseg(this.head, null))
14 }
```

Listing 2.10: A list segment predicate `lseg` specifies the segment of a linked list starting at `this` and ending at (but excluding) `end`. A full list then is a list segment ending at `null`.

```
1  method concat(this: Ref, ptr: Ref, end: Ref)
2    requires acc(lseg(this, ptr)) && acc(lseg(ptr, end))
3    ensures  acc(lseg(this, end))
4  {
5    if(this != ptr) {
6      unfold acc(lseg(this, ptr))
7      concat(this.next, ptr, end)
8      fold acc(lseg(this, end))
9    }
10 }
```

Listing 2.11: A method representing the lemma that a single list segment can be obtained by concatenating two adjacent list segments.

up in the call stack, such as the client's caller. To account for these statically unknown monitors, the client has been extended by a `residue` argument that models the highest waitlevel of all monitors already held by the current thread (an alternative modelling based on abstraction functions, see Section 2.4, is possible).

2.3 Unbounded Heap Structures

Viper supports several idioms for specifying and reasoning about unbounded heap structures. There are no specific definitions built in; instead, Viper includes three features which enable providing the relevant definitions as part of the input program: (1) recursive predicates [101], which are the traditional means of specifying unbounded heap structures in tools based on permission logics, (2) magic wands [117], which are particularly useful for specifying data structures with "holes", and (3) quantified permissions [95], which enable pointwise rather than recursive specifications. Each feature will be introduced by extending the sorted-list example from Listing 2.5. We will focus on the specification of permissions first, and only afterwards reintroduce the sortedness constraint.

2.3.1 Recursive Predicates

Recursive predicates [101] are the classical means in separation logic of specifying inductive data structures such as lists and trees. Like permissions, predicates (or rather, predicate instances) may be held by method executions and loop iterations,

and may be transferred between them. Listing 2.10 shows two such predicates: the `lseg` predicate, which provides permissions to a linked-list segment from node `this` to (but excluding) node `end`, and the `list` predicate, which provides permissions to an entire list starting at `this.head` (and ending with a `null` reference).

A predicate definition consists of a name, a list of formal parameters, and a body, which contains the assertion defining the predicate. The body is optional; omitting it results in an *abstract* predicate, which is useful to hide implementation details from clients. Semantically, a predicate instance is equivalent to its (finite) expansion (similar to the theory of equirecursive types). Automated tools, in contrast, commonly treat a predicate instance as an abstract resource held in the current verification state, which needs to be exchanged for its definition in order to reason about the predicate's semantics. Exchanging the instance for its body is known as *unfolding* or *unrolling*, the inverse operation is called *folding* or *rolling*. Determining when to perform an exchange is in general undecidable, which leaves tool implementers with two general strategies: exchanging based on heuristics, and exchanging based on user-provided annotations. The heuristics-based approach has been successfully used in automated verifiers that carefully restrict their predicate definitions to a decidable fragment, for example, Smallfoot [11] and GRASShopper [103]. To our knowledge, the only automated verifier that used the heuristics-based approach in combination with unrestricted predicate definitions was VeriCool [121]: predicates were axiomatised in the underlying SMT solver, which used its regular heuristics for quantifier instantiations to decide when to (un)fold a predicate's definition. Without additional restrictions, such an encoding can easily lead to non-termination of the SMT solver due to infinite (un)folding: the axioms are causing a *matching loop*. We believe that the choice of the heuristics-based strategy contributed to the performance problems observed by the author (see [118], p. 75 and p. 116).

In Viper, it therefore is the user's responsibility to decide when to (un)fold predicates: the statement `unfold acc(`$P(\bar{e})$`)`, where $\bar{e}$ is a sequence of argument expressions, instructs the verifier to exchange predicate instance $P(\bar{e})$ for its body, statement `fold` achieves the opposite. Predicates can be combined with fractional permissions, which makes it possible to hold a fraction of a predicate instance; consequently, it is also possible to fractionally (un)fold predicate instances. Unfolding a predicate can be understood as exhaling the instance and inhaling the definition, and analogous for folding. In order to integrate predicates and heap-dependent expressions, Viper also supports an `unfolding` expression: `unfolding acc(`$P(\bar{e})$`) in` e temporarily unfolds instance $P(\bar{e})$ for the duration of the evaluation of expression e. A corresponding `folding` expression is currently not available, because it seems less useful in practice, but it could certainly be supported.

List segment predicates can be used to specify iterative traversals of linked lists, as shown in Listing 2.12. In the loop invariant in lines 31 – 33, the first instance of `lseg` describes the list segment traversed so far, and the second instance describes the remainder of the list that still needs to be traversed. The former explains the need for a list *segment* predicate (compared to a proper list predicate): bookkeeping permissions for the partial list already traversed is needed in order to finally reassemble the whole list (line 55).

Manipulating recursive predicates can be tedious. While it is easy to prepend an element to a data structure (by folding another instance of the predicate), extending a data structure at the other end requires additional work to unfold the recursive instances until the end and then refold them including the new element. In Listing 2.12, this operation is performed by the `concat` method, which itself is shown in Listing 2.11. The method plays the role of proving the lemma that from `lseg(`x`, `y`) && lseg(`y`, `z`)` a concatenated `lseg(`x`, `z`)` can be obtained. `concat` is a specification-only method, but Viper does not distinguish between regular and ghost code. In the next subsection, we will explain an approach that reduces the overhead of writing and proving such methods in many cases.

```
1  method insert(this: Ref, elem: Int) returns (index: Int)
2    requires acc(list(this))
3    ensures  acc(list(this))
4  {
5    var tmp: Ref
6    index := 0
7
8    unfold acc(list(this))
9    if(this.head != null) { unfold acc(lseg(this.head, null)) }
10
11   if(this.head == null || elem <= this.head.data) {
12     /* Create a new head node storing the inserted element */
13     tmp := new(*)
14     tmp.data := elem
15     tmp.next := this.head
16
17     fold acc(lseg(this.head, null))
18     fold acc(lseg(tmp, null))
19
20     this.head := tmp
21   } else {
22     var hd: Ref := this.head
23     var ptr: Ref := hd
24     fold acc(lseg(hd, ptr))
25     index := index + 1
26
27     /* Find the insert position */
28     while(   ptr.next != null
29           && unfolding acc(lseg(ptr.next, null)) in
30                ptr.next.data < elem)
31       invariant acc(lseg(hd, ptr))
32       invariant acc(ptr.next) && acc(ptr.data)
33       invariant acc(lseg(ptr.next, null))
34     {
35       var ptrn: Ref := ptr.next
36
37       unfold acc(lseg(ptr.next, null))
38       fold acc(lseg(ptrn, ptrn))
39       fold acc(lseg(ptr, ptrn))
40
41       concat(hd, ptr, ptrn) /* Extend traversed list segment */
42
43       ptr := ptrn
44       index := index + 1
45     }
46
47     tmp := new(*)
48     tmp.data := elem
49     tmp.next := ptr.next
50     ptr.next := tmp
51
52     fold acc(lseg(ptr.next, null))
53     fold acc(lseg(ptr, null))
54
55     concat(hd, ptr, null) /* Concat remaining segments */
56   }
57
58   fold acc(list(this))
59 }
```

Listing 2.12: Inserting an element into a (potentially sorted) linked list; the loop traverses the list and determines the appropriate insert position. Permissions to the list nodes are specified using two list segment predicates: one from the list head to the current position, and one from there to the end of the list.

```
1  define A acc(lseg(ptr, null))
2  define B acc(lseg(hd, null))
3
4  /* Find the insert position */
5  while(   ptr.next != null
6        && unfolding acc(lseg(ptr.next, null)) in
7             ptr.next.data < elem)
8    invariant acc(ptr.next) && acc(ptr.data)
9    invariant acc(lseg(ptr.next, null))
10   invariant A --* B
11 {
12   var prev: Ref := ptr
13
14   unfold acc(lseg(ptr.next, null))
15
16   ptr := ptr.next
17   index := index + 1
18 }
```

Listing 2.13: An alternative specification of the loop from method `insert` that uses a magic wand to implicitly describe the already traversed list prefix. The lemma method `concat` is no longer needed.

2.3.2 Magic Wands

The *magic wand* is a binary connective, written $a_1 \mathrel{-\!\!*} a_2$, which can be understood as the promise that, if the wand is combined with state satisfying the assertion a_1, the combination can be exchanged for the assertion a_2 [99, 108]. Various works have demonstrated the usefulness of magic wands, for example, for proving interesting properties of partial data structures such as partially-traversed lists and trees [130, 87], for specifying protocols that ensure orderly modifications of data structures [76, 52, 68], and for reasoning about synchronisation barriers [44]. The semantics of magic wands involve a quantification over possible state extensions, which makes it challenging to support this versatile connective in automated verifiers: to our knowledge, only Viper and VerCors [17] support magic wands. In this subsection, we will use the magic wand to give an alternative specification of the loop from Listing 2.12. A detailed discussion of the magic wand and how it is supported in Viper is given in Chapter 5.

Listing 2.13 shows an alternative specification of the loop (lines 28 – 45) of method `insert` from Listing 2.12. In this alternative version, the loop invariant in line 10 implicitly achieves bookkeeping the permissions to the already traversed prefix of the list: the magic wand `acc(lseg(ptr, null)) --* acc(lseg(hd, null))` describes the promise that the still-to-be-traversed tail of the list (starting at `ptr`) in combination with the wand itself can be exchanged for the full list (starting at `hd`). The magic wand thus indirectly describes the already traversed list prefix: the permissions implicitly described by a magic wand instance are essentially the same as those explicitly described by the `acc(lseg(hd, ptr))` assertion in Listing 2.12. This way, magic wands eliminate the need for predicates that describe partial versions of data structures. In our case, the `lseg` predicate could be removed; indeed, all used `lseg` instances end in `null`, and thus describe complete lists. We decided to keep the `lseg` predicate, however, to simplify comparing the two alternatives, but we no longer need the auxiliary `concat` method to manage `lseg` predicates.

Conceptually, the magic wand instance occurring in the invariant needs to be created (established) before the loop, it needs to be extended after each iteration such that the exchange promise includes the list node visited in the current iteration, and it needs to be used after the loop to obtain the full list. In general, this requires user-provided annotations which are similar to the `fold` and `unfold` statements for predicates.

```
1  field val: Int     // Value of an array slot
2  field elems: Array // See domain definition in Section 2.5.1
3  field size: Int    // Number of array slots in use
4
5  // Permissions to slots [from..to) of array rcvr.elems
6  define elemsacc(from, to, rcvr, perms)
7    forall i: Int ::     from <= i && i < to
8                      ==> acc(loc(rcvr.elems, i).val, perms)
9
10 predicate AList(this: Ref) {
11       acc(this.elems) && acc(this.size)
12   && 0 <= this.size && this.size <= len(this.elems)
13   && 0 < len(this.elems)
14   && elemsacc(0, len(this.elems), this, write)
15 }
```

Listing 2.14: A Viper encoding of an array list. Arrays are modelled by a custom `Array` type (shown in Section 2.5.1), `loc(rcvr.elems, i).val` denotes the `i`-th array slot, and quantified permissions are used to specify permissions to all slots of the array.

Viper's support for magic wands, however, uses heuristics that automate these steps in many cases, including our list example. Further details are presented in Chapter 5.

2.3.3 Quantified Permissions

In addition to recursive predicates, Viper supports *quantified permissions* as a means of specifying unbounded heap structures. Quantified permissions is Viper's implementation of separation logic's *iterated separating conjunction* [108]: as such, it allows the *pointwise* specification of permissions. The flat structure of a pointwise specification is convenient for specifying data structures that are not limited to traversals in a single, hierarchical fashion, such as cyclic data structures [14, 136], random access data structures such as arrays [108], and general graphs [136]. Similar to magic wands, iterated separating conjunctions have been used in various by-hand proofs; yet, no existing program verifier supports general iterated separating conjunctions directly. In this subsection, we will use quantified permissions to specify an array-backed implementation of a list, a detailed discussion of quantified permissions and how it is supported in Viper are given in Chapter 4.

Quantified permissions are denoted by a universal quantifier around the usual accessibility predicates. For example, `forall x: Ref :: x in S ==> acc(x.f)` denotes permission to the `f` field of every reference in the set `S`. The quantified variable can be of any type, and we permit arbitrary boolean expressions to constrain its range. In the context of array specifications, for example, the quantified variable usually ranges over integers that are valid indices into the array.

Arrays are not supported natively in Viper but can be encoded, as is shown in Section 2.5.1. A new type `Array` is introduced, alongside two functions

```
loc(a: Array, i: Int): Ref
len(a: Array): Int
```

which model the *i*-th slot of an array *a* as the heap location `loc(`*a*`, `*i*`).val` (`loc` is injective), respectively, the length of an array. Permission to the array slots can then be denoted via quantified permissions ranging over the array indices.

In this subsection, we apply this approach to encode an array list. Listing 2.14 shows the necessary field declaration and the declaration of the predicate that specifies the array list. The field `elems` stores the array, while `size` counts the number of array slots that are currently in use. Given our array encoding, accessing the first slot of `this.elems` is denoted by `loc(this.elems, 0).val` (corresponding

to `this.elems[0]` in Java). The quantifier in the body of macro `elemsacc` (line 6) denotes permission to all slots of `rcvr.elems` in the right-open range `[from..to)`. Predicate `AList` specifies the array list, in particular, permissions to the relevant list fields and to all array slots.

Listing 2.15 shows the `insert` method for the array list: it finds the appropriate insert position (lines 8 – 16), resizes the array if all slots are in use (lines 18 – 31), shifts the later elements to the right to free a slot for the element to insert (lines 36 – 43), and finally assigns the element to the array (line 45).

The invariants of the two loops (starting at line 8 and line 36, respectively) are essentially copies of the body of predicate `AList` (with additional constraints on the loop variable), and we could use another macro to further reduce specification duplication. As before, fractional permissions are used to specify that the loops do not modify certain locations.

The inhales in lines 21 – 23 encode the instantiation of a new array that is twice as large as the current array (`new int[this.elems.length * 2]` in Java). The inhale in line 26 simulates the effect of assigning the contents of the old array to the newly instantiated one, which could also be encoded and specified. The annotation `{loc(a, i).val}` specifies a syntactic *trigger* (or *pattern*) for the quantifier, which restricts instantiations of the quantifier during proof search to situations which are concerned with `{loc(a, i).val}` (for some concrete `i`). Carefully controlling quantifier instantiations is important when working with SMT solvers [83, 91, 59]: enabling too few instantiations may cause proofs to unexpectedly fail, while too many may lead to unreliable performance or even non-termination of the solver. Viper (but also SMT solvers themselves) can apply heuristics to infer triggers, but the heuristics come without any guarantees. More details about quantifiers and triggers are given in Section 2.6.

2.4 Functional Behaviour

The specifications shown in Section 2.3 focus on the management of permissions, but do not constrain the values stored in data structures (for example, to require sortedness of the list) or computed by operations (for example, to express the functional behaviour of method `insert`). The early examples from Section 2.2 specify such properties, but in a way which exposes implementation details. In this section, we explain several ways of expressing functional behaviour in Viper while hiding implementation details from clients.

In separation logic, the default way of specifying the values stored in data structures is including additional constraints on the values in the body of a predicate, alongside the permissions necessary to access these values. For example, we could extend the body of the `lseg` predicate defined in Listing 2.10 by conjoining another assertion:

```
unfolding acc(lseg(this.next, end)) in
  this.next != end ==> this.data <= this.next.data
```

This assertion specifies sortedness of the list pairwise by constraining adjacent list nodes. Maintaining the augmented `lseg` predicate entails corresponding additions to the loop invariant of the `insert` method and to the specification of the `concat` method.

Constraining values via predicates allows one to encode representation invariants, but is not well-suited to express client-visible invariants or the functional behaviour of operations. To support such specifications, Viper supports *heap-dependent functions* that may be used in program statements and assertions. Functions (as opposed to methods) have expressions rather than statements as a body; a function's precondition must require sufficient permissions to evaluate the function's body. Since functions are side-effect free, the permissions need not actually be consumed when a function is

```
1  method insert(this: Ref, elem: Int) returns (idx: Int)
2    requires acc(AList(this))
3    ensures  acc(AList(this))
4  {
5    idx := 0
6    unfold acc(AList(this))
7
8    while (   idx < this.size
9           && loc(this.elems, idx).val < elem)
10     invariant acc(this.elems, 1/2) && acc(this.size, 1/2)
11     invariant this.size <= len(this.elems)
12     invariant elemsacc(0, len(this.elems), this, 1/2)
13     invariant 0 <= idx && idx <= this.size
14   {
15     idx := idx + 1
16   }
17
18   if(this.size == len(this.elems)) {
19     // Out of space - allocate double array size
20     var a: Array
21     inhale len(a) == len(this.elems) * 2
22     inhale forall i: Int ::     0 <= i && i < len(a)
23                             ==> acc(loc(a, i).val)
24
25     // Simulate memcpy from the old to the new array
26     inhale forall i: Int :: {loc(a, i).val}
27                   0 <= i && i < len(this.elems)
28               ==> loc(a, i).val == loc(this.elems, i).val
29
30     this.elems := a
31   }
32
33   var j: Int := this.size
34
35   // Shift the later elements to the right
36   while (j > idx)
37     invariant acc(this.elems, 1/2) && acc(this.size, 1/2)
38     invariant elemsacc(idx, this.size + 1, this, write)
39     invariant idx <= j && j <= this.size
40   {
41     loc(this.elems, j).val := loc(this.elems, j - 1).val
42     j := j - 1
43   }
44
45   loc(this.elems, idx).val := elem
46   this.size := this.size + 1
47
48   fold acc(AList(this))
49 }
```

Listing 2.15: Inserting an element into an array list: the first loop determines the appropriate insert position, the next block doubles the size of the underlying array if necessary, and the second loop shifts the elements so that the new element can be inserted.

invoked, and it is therefore not necessary to explicitly return them via the function's postcondition.

Functions are a flexible feature which can play several different roles in a Viper program. The first major role is to encode side-effect free observer methods (*pure* methods in JML [79] and Spec# [7]), which are a part of the interface of many data structures. For example, list-style collections typically provide observer methods such as `length` and `itemAt` to retrieve data. As an example, we extend our `lseg`-based specification from Section 2.3.1 with the following function definition:

```
function lsegLength(this: Ref, end: Ref): Int
  requires acc(lseg(this, end))
{
  unfolding acc(lseg(this, end)) in
    this == end ? 0 : 1 + lsegLength(this.next, end)
}
```

This definition enables us, whenever we hold an `lseg` predicate instance, to express its length via an application of `lsegLength`. The Viper verifiers axiomatise a function definition to the underlying SMT solver, in a way that carefully controls the unrolling of recursive function definitions by essentially mimicking the traversal of the corresponding `lseg` data structure [59]: by default, the definition can be unrolled only once; in addition, each (un)folding of an `lseg` predicate instance allows the solver to perform a corresponding unrolling of the function definition.

A second major role of functions is to define *abstraction functions* [60], which serve as abstractions of the underlying data representation, in order to express specifications without revealing implementation details. For example, the following function abstracts the values of a list segment to a mathematical sequence (due to a technical limitation, Silicon needs an additional postcondition to prove the quantifier; the full code is shown in Listing A.1 in Appendix A):

```
function lsegContent(this: Ref, end: Ref): Seq[Int]
  requires acc(lseg(this, end))
  ensures  forall i: Int, j: Int ::
                0 <= i && i < j && j < |result|
             ==> result[i] <= result[j]
{
  this == end
    ? Seq[Ref]()
    : unfolding acc(lseg(this, end)) in
        (Seq(this.data) ++ lsegContent(this.next, end))
}
```

Viper's verifiers reason about a function application in terms of the function's body. Nevertheless, it is sometimes useful to provide a function postcondition. In the above example, the postcondition expresses that the sequence of all values stored in the list is sorted, which is implied by the pairwise sortedness we have added to the `lseg` predicate. Note that the inductive argument required to justify this postcondition is implicit in the checking of `lsegContent`'s recursive definition.

A similar `content` function for the overall data structure (described by the `list` predicate) allows us to specify the functional behaviour of `insert` by adding another postcondition clause to the method:

```
ensures    content(this) == old(content(this))[0..index]
        ++ Seq(elem)
        ++ old(content(this))[index..]
```

Function bodies are optional in Viper, which allows hiding details when verifying client code (similar to abstract predicates). Omitting the body is also useful for axiomatising a function rather than defining it, in which case the existence of the

function needs to be justified by additional meta-reasoning (similar to the case of inhale-exhale assertions introduced in Section 2.2.3).

In the array-list example from Section 2.3.3, defining `length` and `itemAt` functions is straightforward. However, an analogous `content` function would be awkward to define recursively since our specifications for this random-access example avoid recursive definitions. Instead, we can axiomatise the function, that is, specify its meaning via a quantified postcondition:

```
function content(this: Ref): Seq[Int]
  requires acc(AList(this))
  ensures  |result| == length(this)
  ensures  forall i: Int ::
                0 <= i && i < length(this)
            ==> result[i] == itemAt(this, i)
```

The third major role of heap-dependent functions is to express refinements of existing predicate definitions. For example, instead of expressing sortedness as part of a predicate definition, we can write a boolean function `sorted` (shown next) that can be used in combination with the unchanged `AList` predicate from Section 2.3.3: the assertion `AList(this) && sorted(this)` then describes a sorted array list, while `AList(this)` alone specifies an array list that may or may not be sorted. In this way, functions can be used to augment data-structure instances with additional invariants, without requiring many versions of a predicate definition or resorting to higher-order logic.

```
function sorted(this: Ref): Bool
  requires acc(AList(this))
{
  unfolding acc(AList(this)) in
    forall i: Int, j: Int ::
          0 <= i && i < j && j < this.size
      ==> result[i] <= result[j]
}
```

In summary, the combination of predicates, functions, and quantifiers that Viper supports provides the means for writing rich functional specifications in a variety of styles, which is essential for a practical intermediate verification language.

2.5 First-Order Theories

Many specification and verification techniques provide their own mathematical vocabulary, for example, to encode algebraic data types. To support such techniques, Viper supports the declaration of custom first-order theories via *domains*: each domain introduces a (potentially polymorphic) type and may declare uninterpreted function symbols and axioms. Ultimately, the collection types that are natively supported by Viper (sets, sequences, and multisets) could be axiomatised as additional domains instead (with regular function symbols replacing the built-in operators). Organising mathematical theories into domains allows back ends to provide dedicated support for certain theories. For example, while both Viper verifiers let the underlying SMT solver reason about such custom theories, an abstract-interpretation-based inference might provide specialised abstract domains for certain Viper domains.

2.5.1 Encoding Arrays

Listing 2.16 shows a domain that can be used to model arrays, which are not natively supported in Viper. The domain, in particular, the induced `Array` type and the `loc` function, has already been used in Section 2.3.3 in the encoding of the array list.

```
domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int

  function loc_a(r: Ref): Array
  function loc_i(r: Ref): Int

  axiom loc_injective {
    forall a: Array, i: Int :: {loc(a, i)}
          0 <= i && i < len(a)
      ==>    loc_a(loc(a, i))) == a
          && loc_i(loc(a, i))) == i
  }

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

Listing 2.16: A domain definition for arrays, as used in Section 2.3.3. The injective function `loc` maps an array and an index to a reference; in combination with a field (such as `val` in Listing 2.14), an array slot `a[i]` can be encoded as `loc(a, i).val`.

The domain enables the modelling of arrays as follows: the i-th slot of an array a is denoted by `loc(`a`,`i`).val`, where `val` is a suitable field, that is, a field of type T if we want to model an array of elements of type T. Since each array slot corresponds to a dedicated memory location, `loc` must be injective; this property is expressed by the axiom `loc_injective`, which axiomatises `loc_a` and `loc_i` as the inverse functions of `loc`.

Axiomatising injectivity via inverse functions improves the performance of the SMT solver[3]by reducing the number of potential instantiations of the axiom: the straightforward injectivity axiom with two quantified variables could be instantiated for every pair of arguments, whereas the provided inverse function axiom results in a linear number of instantiations. The explicitly provided quantifier trigger `{loc(a, i)}` ensures that the SMT solver can instantiate the axiom whenever it (potentially) needs to learn injectivity of `loc(a, i)`. A more detailed discussion of this choice of trigger is given in Section 2.6.

2.5.2 Encoding Algebraic Data Types

Another use case for domains is the encoding of algebraic data types. Consider the following Haskell declaration of a list of integers:

```
data List = Nil | Cons Int List
```

Such an algebraic data type can be modelled by one *constructor* function and n *deconstructor* (or projection) functions per data type constructor (of arity n), and an appropriate set of axioms that express relevant properties such as constructor injectivity.

Listing 2.17 shows a first set of functions and axioms of a `List` domain that is used to model the algebraic list data-type. The `type` function and its three axioms express that the two data-type constructors create different data types, and that each data type element is either a `Nil` or a `Cons`. The `unique` keyword marks a constant (a nullary function) as disjoint from all other `unique` constants (of the same type) in the current program. The axioms' quantifiers have not been annotated with triggers because

[3]The "injectivity via inverse functions" trick is described in the Z3 Guide at `http://rise4fun.com/z3/tutorialcontent/guide`.

```
1   /* Constructors */
2
3   function Nil(): List
4   function Cons(head: Int, tail: List): List
5
6   /* Constructor types */
7
8   function type(xs: List): Int
9   unique function type_Nil(): Int
10  unique function type_Cons(): Int
11
12  /* Type axioms */
13
14  axiom type_of_Nil {
15    type(Nil()) == type_Nil()
16  }
17
18  axiom type_of_Cons {
19    forall head: Int, tail: List ::
20      type(Cons(head, tail)) == type_Cons()
21  }
22
23  axiom type_exhaustiveness {
24    forall xs: List ::
25      type(xs) == type_Nil() || type(xs) == type_Cons()
26  }
```

Listing 2.17: Encoding an algebraic list data-type, part I/II: different constructors create different types of values, and each list value is of one such type.

Viper infers the "obvious" choices: `{type(Cons(head, tail))}` for the axiom in line 16, and `{type(xs)}` for the axiom in line 21 (Section 2.6 provides more details). For convenience, we also introduce the following macros:

```
define is_Nil(xs)  type(xs) == type_Nil()
define is_Cons(xs) type(xs) == type_Cons()
```

The deconstructors (for `Cons`, `Nil` takes no arguments) and their axioms are shown in Listing 2.18. For these axioms, explicit triggers are required because Viper would infer triggers that are too strict and prevent certain proofs from succeeding, as described in Section 2.6. Note that the deconstructors for `Cons` are underspecified total functions, that is, applying them to `Nil` does not automatically result in a verification failure, it just provides no additional information. If a verification failure is desired, an abstraction function (that is, not a domain function) with an appropriate precondition can be used in addition to the shown domain function.

This encoding allows proving various properties of the list data-type, for example, the three properties asserted in Listing 2.19. It also enables pattern match *exhaustiveness checks* that are more precise than the checks that compilers for languages such as Haskell or Scala typically perform. Consider the following Haskell function definition:

```
sumup :: List -> Int
sumup Nil = 0
sumup (Cons x ys) = x + (sumup ys)
```

The function is defined for all inputs, that is, total, it will therefore never cause a run-time exception that would be raised if the function were applied to an argument that matched none of the function's defining patterns. Ensuring totality of a function requires checking that the set of defining patterns is exhaustive, which can be encoded in Viper as follows (for an arbitrary `xs`):

```
assert is_Nil(xs) || is_Cons(xs)
```

```
1   /* Deconstructors */
2
3   function head_Cons(xs: List): Int
4   function tail_Cons(xs: List): List
5
6   /* Deconstructor axioms */
7
8   axiom destruct_over_construct_Cons {
9     forall head: Int, tail: List :: {Cons(head, tail)}
10         head_Cons(Cons(head, tail)) == head
11      && tail_Cons(Cons(head, tail)) == tail
12  }
13
14  axiom construct_over_destruct_Cons {
15    forall xs: List :: {head_Cons(xs)} {tail_Cons(xs)}
16          is_Cons(xs)
17      ==> xs == Cons(head_Cons(xs), tail_Cons(xs))
18  }
```

Listing 2.18: Encoding an algebraic list data-type, part II/II: data-type constructors and deconstructors are inverse functions.

```
1     // The elements of a deconstructed Cons are
2     // equivalent to the corresponding arguments of Cons
3     assert forall head: Int, tail: List, xs: List ::
4       is_Cons(xs) ==>
5         (     head == head_Cons(xs) && tail == tail_Cons(xs)
6          <==> Cons(head, tail) == xs)
7
8     // Two Cons are equal iff their constructors'
9     // arguments are equal
10    assert forall head1: Int, head2: Int,
11                  tail1: List, tail2: List ::
12          Cons(head1, tail1) == Cons(head2, tail2)
13      <==> head1 == head2 && tail1 == tail2
```

Listing 2.19: Proving properties of the previously encoded algebraic list data-type.

Each disjunct of the assertion encodes one of the defining patterns of function `sumup`: both patterns unconditionally match against the structure of the argument value, which corresponds to a type test in our encoding of abstract data types. The assertion holds, proving that the encoded `sumup` function is total. If the second defining pattern is changed such that the function only sums positive list elements

```
sumup (Cons x ys) | 0 <= x = x + (sumup ys)
```

that is, if the function is deliberately made partial, then the corresponding Viper assertion

```
assert    is_Nil(xs)
       || (is_Cons(xs) && let x == (head_Cons(xs)) in 0 <= x)
```

fails (as expected), showing that `sumup` is indeed no longer a total function.

2.6 Quantifiers and Triggers

The idea of guiding quantifier instantiation heuristics by syntactic *matching triggers* originates from the Simplify theorem prover [39], and has since been adopted by state-of-the-art SMT solvers such as Z3 [92] and CVC4 [10], and by the latest version of the SMT-LIB standard (v2.5, under the term *pattern*) [9]. A single trigger $(t_1, \ldots, t_n)$ is a sequence of terms t_i (of the SMT-LIB language), suggesting to the solver to only

instantiate the annotated quantifier with terms that are subterms of ground terms that match all provided triggers. Adding terms to a trigger therefore in general reduces the instantiation possibilities of a quantifier, potentially improving the solver's performance. Multiple triggers $(t_{1,1}, \ldots, t_{1,n_1}), \ldots, (t_{m,1}, \ldots, t_{m,n_m})$ can be provided and are interpreted as alternatives: if the solver matches one of the triggers, the quantifier may be instantiated. Hence, adding triggers in general increases the instantiation possibilities of a quantifier, potentially making the proof search more complete (but less efficient).

Viper's syntax for triggers has been borrowed from Boogie: triggers precede the quantifier body, a single trigger is a comma-separated sequence of expressions inside curly braces, and multiple triggers are separated by whitespace. Triggers have already been used in previous sections, for example, in Listing 2.18, where the first axiom has been annotated with the single trigger `{Cons(head, tail)}` and the second axiom with the two (alternative) triggers `{head_Cons(xs)}` `{tail_Cons(xs)}`.

The SMT-LIB standard allows arbitrary binder-free terms in triggers, but SMT solvers often impose additional restrictions on the shape of allowed terms. Common restrictions are that triggers must be composed of *uninterpreted* function applications applied to free or bound variables, that is, they may not contain boolean connectives and arithmetic expressions. Furthermore, each trigger must cover (include) all quantified variables. Viper ultimately enforces these restrictions as well: a trigger is valid if it only contains heap-dependent functions, as well as custom and built-in domain functions (that is, of sequences, sets and multisets). In addition, Viper allows field dereferences in triggers, which is particularly useful for specifications that use quantified permissions, as illustrated in Listing 2.15 Quantifiers denoting quantified permissions do not need to (and cannot) be annotated with triggers, since they are not regular (first-order) quantifiers that are ultimately handled by an SMT solver.

Triggers can be omitted, as was done in most of the previously discussed examples. In this case, Viper does a best-effort attempt to infer valid triggers by inspecting the quantifier body, but it cannot guarantee to find optimal triggers. Viper's general strategy is to pick as many (non-redundant) alternative triggers as possible, where each individual trigger is as strict as possible. Note that Viper's trigger inference is still in an early development stage and likely to change in the future.

Consider, for example, the following quantifier, originating from Listing 2.17

```
forall head: Int, tail: List ::
  type(Cons(head, tail)) == type_Cons()
```

for which Viper infers the trigger `{type(Cons(head, tail))}`. This is a valid trigger (no disallowed subexpressions, all quantified variables covered), but it is stricter than the possible alternative `{Cons(head, tail)}`. However, the stricter trigger still allows to prove the desired properties shown in Listing 2.19. This is not the case if the trigger for the following quantifier from Listing 2.18 is omitted:

```
forall head: Int, tail: List :: {Cons(head, tail)}
     head_Cons(Cons(head, tail)) == head
  && tail_Cons(Cons(head, tail)) == tail
```

Viper will infer the two alternative triggers

```
{head_Cons(Cons(head, tail))}
{tail_Cons(Cons(head, tail))}
```

each of which is stricter than the manually provided one. These stricter triggers prevent the second property shown in Listing 2.19 from being proven because the body of the quantifier only contains `Cons`, but no applications of `head_Cons` or `tail_Cons` to `Cons`.

Viper may also try to rewrite quantifiers internally to increase the chances of finding valid triggers. The following quantifier, for example, where `idx` computes an index into the sequence `xs`

```
forall i: Int ::
  0 <= i && i < |xs| ==> xs[idx(i, j + 1)] > 0
```

does not contain any valid triggers: the only function application that covers the quantified variable is `idx(i, j + 1)`, but it also contains a forbidden arithmetic expression. Viper rewrites the quantifier by abstracting over the forbidden arithmetic expression with an additional quantified variable `a1`, which is then used in the trigger:

```
forall i, a1 :: {xs[idx(i, a1)]}
  0 <= i && i < |xs| ==> xs[idx(i, j + 1)] > 0
```

A different rewrite strategy is necessary for the next quantifier, where abstracting over the forbidden arithmetic expression `i + j` would not allow finding a valid trigger because the resulting expression `xs[a1]` would no longer cover all quantified variables:

```
forall i: Int ::
  0 <= i && i < |xs| - j ==> xs[i + j] > 0
```

Viper therefore rewrites the quantifier by replacing the forbidden expression `i + j` with a new variable `i1` that will be quantified over in place of `i`, and by rewriting the body accordingly:

```
forall i1 :: {xs[i1]}
  0 <= i1 - j && i1 - j < |xs| - j ==> xs[i1] > 0
```

It is crucial to be aware of the impact that triggers can have on the performance of SMT solver (see also [83, 91, 59]), and we recommend to carefully consider potential triggers when writing quantifiers. As a last, admonitory example, consider the following quantifier, where `pw[i]` is a sequence of integers:

```
forall i: Int :: {pw[i]}
      left <= i && i <= right
  ==> loc(a, i).val == old(loc(a, pw[i]).val)
```

If the explicit trigger were omitted, Viper would be free to pick `{loc(a, i).val}` as a trigger, which has a high chance of causing non-terminating runs of the SMT solver due to the following matching loop: every instantiation of the axiom — by matching some ground expression `loc(a, j0).val)` — yields a new ground expression `loc(a, pw[j0]).val)` that can in turn be used to instantiate the axiom again.

2.7 Permission Model

Several features of Viper's rich permission model have already been shown: fractional permissions, permission introspection via `perm` and `forperm`, magic wands and quantified permissions. This subsection presents two additional features which can be used to overcome modularity issues potentially arising from committing to concrete fractions in situations where an arbitrary read permission would suffice.

In Section 2.2.2 and Section 2.2.3, we used a concrete fractional permission such as `1/2` in specifications to express that a method or a loop needs to read a heap location. Committing to a specific value in cases were any fraction will suffice (in order to read the location) does not only burden developers with having to make an arbitrary choice: it will in general pose a modularity problem because it can result in specifications that prevent one method that only reads a given heap location from calling another read-only method because the former holds insufficient (read) permissions.

```
1  interface Expr {
2    int eval(State s)
3      @Requires(acc(s.map), π)
4      @Ensures(acc(s.map), π)
5  }
6
7  class Add implements Expr {
8    Expr left;
9    Expr right;
10
11   int eval(State s)
12     @Requires(acc(s.map), π)
13     @Ensures(acc(s.map), π)
14   {
15     forkHandleLeft  = fork left.eval(s);
16     forkHandleRight = fork right.eval(s);
17
18     return   (join forkHandleLeft)
19            + (join forkHandleRight);
20   }
21 }
```

Listing 2.20: Simplified specifications for `Add` that focus on permissions to the shared `map`; permissions to the left/right subtree have been omitted. The challenge is choosing a permission value for π.

Heule et al. [58] suggest *abstract read permissions* to overcome this problem, a technique that can be summarised as combining permission-parametric specifications with an inference scheme that automatically instantiates the permission parameter. An accessibility predicate of the shape `acc(`$x.f$`, rd)` is used to denote read permissions to $x.f$; on exhale, each occurrence of `rd` is automatically instantiated with a sufficiently small permission value. The use of `rd` in permission expressions is subject to certain syntactic restrictions which ensure by construction that such small-enough values always exist. This allows the approach to simply constrain permission amounts accordingly without having to prove (during the verification) that the set of constraints is satisfiable.

The authors implemented their approach as an extension of the Chalice verification language, but it is also possible to encode this expressive permission model in Viper. Listing 2.21 shows a simplified version of the resulting encoding that focuses on permissions to the shared state, which (for simplicity) is modelled as a single integer. Method `Add_eval` encodes `Add.eval`: the additional permission-typed argument `p` is used to parameterise `Add_eval`'s specification; its value must be non-`none` in order to permit reading the shared state.

At the beginning of `Add_eval`'s body, a new permission-typed variable is declared and constrained to permit read access. The `constraining(q)` block around the exhales (which model the two forks) allows each exhale to constrain `q` such that some permissions always remain with the calling method, which makes the `assert` in line 16 succeed. In our example, the `constraining` block is equivalent to the following code that makes the implicitly generated constraints explicit:

```
// first fork
inhale q < perm(state.map)
exhale acc(state.map, q)

// second fork
inhale q < perm(state.map)
exhale acc(state.map, q)
```

In general, however, such a desugaring would be significantly more involved due to aliasing, and because the `constraining` block admits arbitrary statements in its body, including inhales, assignments and loops.

```
field map: Int

method Add_eval(this: Ref, state: Ref, p: Perm)
  requires none < p
  requires acc(state.map, p)
{
  var q: Perm
  inhale none < q

  constraining(q) {
    // forks
    exhale acc(state.map, q)
    exhale acc(state.map, q)
  }

  assert 2*q < p

  // joins
  inhale acc(state.map, q)
  inhale acc(state.map, q)
}
```

Listing 2.21: A Viper encoding of Listing 2.20, in which recursive calls are parameterised with automatically constrained, sufficiently small abstract read permissions

Note that `constraining` blocks do not guarantee that the generated constraints are always satisfiable, as illustrated by the next snippet:

```
inhale acc(state.map, p)

constraining(p) {
  exhale acc(state.map, p) // assumes p < p
}
```

Recall (Section 2.1) that Viper does in general not guarantee the soundness of encodings; an important property of an intermediate verification language that increases the language's expressiveness, and enables encodings whose soundness depends on external arguments. Analogous to the use of `inhale` statements or inhale-exhale assertions (Section 2.2.3), it is the front ends's responsibility to ensure that the constraints arising from `constraining` blocks are satisfiable. To prove that a certain encoding based on `constraining` blocks by-construction generates only satisfiable constraints, the *layered constraints* approach presented by Boyland et al. [28] can be used.

2.8 Evaluation

In order to assess Viper's suitability as an intermediate verification language, we draw on the VerCors tool for observations about the use of Viper as the back end of a front-end verifier, in particular, which language features are used in practice. To evaluate the performance of Viper, we compare the tool chain VerCors → Viper → Boogie to VerCors → Chalice → Boogie, the tool chain previously used by VerCors.

Language Design

The most comprehensive front ends for Viper are the Java and OpenCL front ends developed in the VerCors project [16], and our own Chalice/Viper front end. Various

language features of Viper have proven essential for these different front ends. VerCors' front end for verifying concurrent Java programs makes heavy use of custom domains to axiomatise ADT-like data types, and of sequences, recursive functions and predicates. For VerCors' OpenCL front end, on the other hand, custom domains similar to the array encoding shown in Section 2.5.1, along with quantified permissions and pure quantifiers for specifying permission, respectively, functional properties, have become an important feature combination. Lastly, Chalice/Viper makes extensive use of `inhale` and `exhale` statements to encode high-level features, similarly to the example presented in Section 2.2.3. In summary, nearly all of Viper's key language features have been heavily used in at least one existing front end.

There are Chalice front ends for both Boogie and Viper, which support very similar (but not identical) versions of the Chalice language. For the Chalice programs from the previous subsection, the Boogie files were between 3.3 and 32.1 times the size of the corresponding Viper files, and on average 11.2 times larger. This significant difference illustrates the higher level of abstraction provided by the Viper language, compared with existing intermediate verification languages.

Performance

The VerCors project switched from using Chalice as an intermediate verification language to Viper, partly motivated by the available language features; for example, the VerCors OpenCL front end relies heavily on quantified permissions, which are not available in Chalice. Another reason was the performance of the different tool chains. In the following, we compare the performance of the two tool chains on inputs generated by the VerCors tools.

Running tests through the entire alternative tool chains proved difficult due to legacy syntactic and implementation differences; however, we identified 17 examples from VerCors' test suite which VerCors could encode in both Chalice and Viper. For each of these examples, we generated two (essentially equivalent) Boogie programs, one using the standard Chalice verifier, the other using Viper's verification-condition-generation-based verifier.

	Average size (LOC)	Mean time (s)	Max. time (s)
Boogie file via Chalice	945.0	0.83	3.22
Boogie file via Viper	631.1	0.53	0.73
Ratio	66.8%	64.3%	22.5%

Figure 2.3: Comparison of alternative back-end infrastructures for the VerCors tools. Using Viper's verification-condition-generation-based verifier significantly reduces the size and verification time of the generated Boogie programs compared to the standard Chalice infrastructure. Timings do not include JVM start-up time: a JVM has been persisted across test runs using the Nailgun tool [88]. Boogie start-up times (via Mono) are included, however. All timings were gathered on a Lenovo Thinkpad T450s running Ubuntu 15.04 64 bit, with 12GB RAM.

Figure 2.3 shows the results of our performance comparison. In all cases, the Boogie files generated via the Viper route were smaller and verified faster. The same example was slowest via both routes, and more than four times faster in the Viper-generated version. Although our sample size is small, the results suggest Viper enables a more direct encoding and offers a more streamlined verification condition generator. In practice, however, the VerCors team typically use Viper's symbolic-execution-based verifier (Silicon, presented in Chapter 3), which is substantially faster still (as demonstrated in Section 3.7).

2.9 Related Work

An overview of the related work has already been given in Section 1.2, with a particular focus on the requirements derived from the thesis' goal of developing an infrastructure for permission-based verification (Section 1.1). In this section, we discuss existing verification infrastructures under consideration of concrete features and design choices presented in the previous sections, and the use of other verification languages as intermediate verification languages. In addition, we briefly discuss existing permission models and their implementations, and compare them with Viper.

Existing Verification Infrastructures

Viper's design has been strongly influenced by Boogie [6], which has been successfully used as the backbone of many different front ends. Boogie and Viper are procedural and do not have a built-in notion of classes; their declarations (methods, functions, etc.) are global and do not take implicit receivers; they support similar sets of control flow statements, including conditionals, method calls, loops and `goto`; and they do not distinguish between real and ghost code. Both languages are sequential, and concurrency features of source languages need to be modelled in encodings, for example, via `assume`/`assert` in Boogie, respectively, `inhale`/`exhale` in Viper.

Both languages have built-in support for certain first-order theories — maps in the case of Boogie; sets, sequences and multisets in the case of Viper — and support the declaration of further first-order theories via uninterpreted types and functions, and appropriate axioms. Boogie programs, like Viper programs, are not guaranteed to be sound (which facilitates the development of front ends): arbitrary assumptions can be encoded via `assume` statements, aforementioned axioms, and via free pre-/postconditions and invariants (of which Viper's inhale-exhale assertions are a generalisation).

The most important difference between Viper and Boogie is that the latter does not have a built-in notion of a heap, and consequently, of permissions: both concepts can be encoded, as demonstrated by Chalice [84] and Viper's own Boogie-backed verifier (Figure 2.1), but this amounts to substantial work, including (1) the choice of a suitable heap encoding (as discussed by Böhme et al. [19]), (2) a similar choice for encoding permissions, which is particularly involved if the permission model is richer than classical fractional permissions (such as Viper's, Section 2.7), and (3) appropriate encodings of permission logic features such as separating conjunction, abstract predicates and magic wands. The required work significantly complicates the development of new permission-based front ends, and we interpret the fact that Chalice is the only permission-based Boogie front end as a consequence thereof. The loss of structural information, resulting from the need to encode core concepts such as a program heap, also impedes the development of efficient and precise back ends.

Boogie supports global variables and modifies-clauses for procedures and loops, which (in combination with maps) enable different encodings of heaps and permissions. In such encodings, fields (which are not a built-in concept) can be modelled as elements of an uninterpreted type, which results in an additional loss of structural information.

The resulting encoding, however, makes it possible to use program heaps analogous to other values: heaps (that is, heap-modelling maps) can be stored in local variables and in other maps (and thus on encoded heaps), and they can be quantified over; none of which can be done with Viper's built-in heap. This increases Boogie's expressiveness, and enables, for example, the encoding of Viper's labelled `old` expressions (Section 2.2.3) in Boogie. Programs that use (encoded) heaps in such intricate ways are very difficult to analyse statically, however, and since one of Viper's goals

is to facilitate the integration of verification and static analysis, we decided against supporting the use of Viper's built-in heap in such ways.

Boogie's type system is richer than Vipers: it supports quantification over types, both explicitly (via universal/existential quantifiers), and implicitly (via polymorphic procedures and functions), and each type (whether built-in or custom) is automatically equipped with a partial order. The latter is convenient for encoding, for example, source-level class hierarchies; in Viper, corresponding partial orders can be defined via appropriate domains.

The intermediate language of the Why3 [18] verification infrastructure, called WhyML, is a sequential, first-order dialect of ML. WhyML supports all of Viper's control flow statements, with the exception of `goto` statements, but it also supports raising and catching exceptions (which, for example, can be used to encode breaking out of a loop, which could be encoded in Viper via `goto` statements). Similar to Boogie, WhyML supports `assert` and `assume` statements that can be used to encode different source-level semantics and (concurrency) features, as well as adding custom first-order theories via uninterpreted types, functions and axioms. Unlike Boogie and Viper, WhyML distinguishes between real and ghost code [47].

WhyML supports the declaration of (potentially recursive) records with named components (comparable to structs with fields), and it allows mutable references. However, the language imposes static alias control for such references, which, for example, rejects the definition of recursive records with mutable fields (as necessary, for example, for mutable list or tree structures). Encoding program heaps in WhyML that permit unrestricted aliasing therefore requires similar encoding efforts as required in Boogie, with all aforementioned disadvantages; and analogous for encoding permissions and permission-based specifications. To our knowledge, no permission-logic-based front end for Why3 exists.

Using Other Verification Languages as Intermediate Languages

Various automated verifiers exist (see also Section 1.2), with input languages that have not explicitly been designed as intermediate verification languages, but which could still be considered as potential target languages of other front ends.

Such verification languages typically offer bespoke support for a small set of high-level programming (and specification) concepts, but do not provide constructs with a lower abstraction level that facilitate the modelling of additional high-level concepts not directly supported, such as Viper's `inhale/exhale` statements. In situations where the concepts supported by the source and the target language differ, front ends may no longer be able to use the high-level concepts of the intermediate language in their encodings, and may thus be left with the remaining language features (for example, basic control flow statements), which can severely complicate the encoding task, or even render it impossible. Poignantly formulated, the encoding is done struggling against the language, rather than with its support.

Chalice, for example, has built-in support for several concurrency features, including *non-reentrant* monitors that can be acquired and released. To ensure deadlock-freedom (recall Section 2.2.3), each monitor is associated with a waitlevel (a position in a partial order), and Chalice prevents threads from acquiring monitors whose waitlevels are not strictly greater than those of all monitors already held by the thread. The waitlevel checks are built into the language and cannot be bypassed, which would be necessary in an attempt to encode *reentrant* monitors by reusing Chalice's support for *non-reentrant* monitors. Due to the lack of `inhale/exhale` statements, the language does not offer much support in this situation. Technically, it might be possible to enocode `inhale/exhale` statements via invocations of methods with appropriate post-/preconditions, but without support for methods that are parametric with respect to their specifications (which is typically not given), such an encoding might require one method per encoded statement.

Permission Systems

This subsection briefly introduces relevant properties of permission systems[4] and gives an overview of existing systems, and discusses Viper's permission system in this context. The subsection is based on the introduction of the paper *Constraint Semantics for Abstract Read Permissions* by Boyland, Müller, Schwerhoff and Summers [28], published at FTfJP 2014.

Fractional permissions [25] (with permission amounts in $\mathbb{R}_{\geq 0}$ or $\mathbb{Q}_{\geq 0}$) are probably the permission system most widely supported by automated verifiers for permission logics, but alternatives exist [20, 43, 26]. To be useful for automatic program verification, a permission system must have the following three key properties: the system must be sufficiently *expressive*, it should require *low annotation overhead*, and it should be *amenable to automatic provers*, especially SMT solvers.

Bornat et al. [20] identify two criteria that characterise expressive permission systems: support for (1) *unbounded divisibility* (or "infinite splitting"), necessary, for example, to specify programs which recursively fork threads and where all sub-threads need read permission to a shared location (see also Section 2.7), and (2) *unbounded counting*, required, for example, when one thread forks off an unbounded number of threads, each with an identical permission, and then waits for them to finish (in arbitrary order). A third requirement arises from scaling (partially (un)folding) abstract predicates, which requires a permission system that supports *multiplication*.

To our knowledge, no existing implementation of a permission system satisfies all of these requirements. Fractional permissions support unbounded divisibility and multiplication, achieve low annotation overhead, and enjoy good support from SMT solvers. However, unbounded counting seems impossible since if one starts with a write permission (fraction 1), then however small a positive fraction $q > 0$ one chooses to give to each sub-thread, there always is a point n after which $1 - nq$ is no longer positive.

Counting permissions [20] support unbounded counting by splitting a permission into an unbounded number of units and the remainder. The system then tracks how many units a thread holds (or how many it lacks for a full permission). However, counting permissions support neither unbounded divisibility (because units cannot be divided further), nor multiplication.

It is possible to compound fractional and counting systems [20], for example, by representing permissions as a fraction plus a positive or negative number of units [84], as implemented in Chalice. Dockins et al. [43] achieve this combination with a tree system for permissions, but multiplication is not supported, and the encoding of counting imposes additional structure (each counting permission is represented differently), which potentially complicates implementation. The implemented decision procedures by Le et al. based on this system [78] do not support counting. In our experience (from working on Chalice), an implementation of a compound system can lead to many disjunctions in proof obligations (since access to a location is permitted if the fraction *or* the unit count is positive), which potentially slows down SMT solvers.

Boyland [26] proposes a permission system based on $\mathbb{Z}[\epsilon]^+$ (positive polynomials over an infinitesimal) which satisfies all three criteria, but we are unaware of any implementation using this complex and subtle system.

In recent work, Boyland et al. [28] introduced the concept of *layered constraints*, and showed that fractional permissions, in combination with constraints over *symbolic* permission amounts, suffice to support counting permissions (in addition to unbounded divisibility and unrestricted multiplication). The formal model is quite complex, but does not need to be exposed to users: it can be treated as if it were the real (or rational)

[4] A verification language (for example, Viper) implements a permission *system*, which is based on a mathematical permission *model*, but includes additional aspects such as permission-typed expressions, and a strategy for tracking permissions in the verification state. For brevity, we ignore this difference in the context of this brief discussion.

numbers with their usual arithmetic laws, which makes it intuitive to understand (and straightforward to implement).

Viper's permission system, presented in Section 2.7, implements fractional permissions, and thus permits unbounded divisibility and multiplication, with low annotation overhead and good SMT solver support. It also supports the idea of *abstract read permissions* [58, 28] (via `constraining` blocks), which can substantially reduce the annotation overhead in situations such as aforementioned recursive thread forking (and improve modularity, as discussed in Section 2.7). Abstract read permissions relieve users from having to pick explicit fractions in situations where any non-zero amount suffices, for example, to read common data shared between forked threads. The layered constraints introduced by Boyland et al. can be used to give a semantics to abstract read permissions, and initial experiments showed that Viper's support for abstract read permissions is expressive enough to enable counting in certain situations. We consider it future work to investigate how expressive the current support is, and if necessary, how it could be improved to enable permission counting in general.

VeriFast [67] also supports (classical) fractional permissions, and thus allows unbounded splitting (and multiplication), but not unbounded counting. However, VeriFast's standard library contains an encoding of counting permissions as *tickets*, which involves ghost methods manipulating auxiliary separation logic predicates. The idea is the following: counting starts by calling a ghost method `start_counting(`r`, `f`)` that exhales f permissions to a resource (a field or predicate) r. Intuitively, the initially exhaled permission amount f is used as a "pool" from which an unbounded amount of tickets can be taken by calling a ghost method `create_ticket(`r`)`, which requires a predicate initially provided by `start_counting`. Unlike counting permissions, however, the tickets cannot be "summed up" easily, because they are encoded as predicates, and thus cannot be counted (without further tickets). To sum up ticket predicates, one needs additional recursive predicates that essentially encode a list of tickets, or an additional ghost method that merges two tickets into a single ticket of count two.

VeriFast supports permission-typed logical variables in specifications, and offers limited support to instantiate these automatically, for example, at call site of a method. However, only the first use of the variable is considered (so fractions cannot be correlated) and that first use will use up all permissions that the caller has (so framing is not supported).

Chapter 3

Silicon: Symbolic Execution of Viper

Symbolic execution was introduced four decades ago as a technique for testing, debugging and verifying programs [74, 24, 54], but its practical breakthrough only came in the early 2000s after significant advances in the field of constraint satisfiability solving had been made that led to the development of high-performance SMT solvers [39, 92]. A major application area for symbolic execution is testing [33], where symbolic execution is applied to increase the number of execution paths through a program that are covered by tests. Another major application area is program verification, in particular building automated verifiers for permission logics [12, 42, 119, 67, 116]. Across these areas, many different flavours of symbolic execution exist, but they all share the same general idea: the execution proceeds using a *symbolic state* that holds symbolic rather than concrete values, and each executed statement updates the symbolic state according to its (abstract) semantics.

In the context of specification-based modular verification, such as Viper, a method implementation can be verified by setting up an initial symbolic state representing all concrete states that satisfy the precondition, by symbolically executing the method body, starting from this initial state, and by checking that the postcondition holds in the final state.

It is common for automated verifiers based on symbolic execution to invoke satisfiability solvers such as SMT solvers to discharge proof obligations that arise during the verification: for example, if the current thread holds the permission to write to a heap location, or if the precondition of a method call is satisfied. It is furthermore common for symbolic-execution-based verifiers to branch over conditionals such as `if-then-else` statements if the value of the condition is not statically known. In this situation, both remaining paths through the program are symbolically executed: the `if` branch under the assumption that the conditional is true, and the `else` branch under the opposite assumption.

Another popular choice for building automated verifiers is *verification condition generation*. Given, for example, a method with pre- and postcondition, verifiers based on this technique typically compute a single, potentially large formula from the postcondition and the method body, called the *weakest precondition* [41] (of the method body with respect to the postcondition). This weakest precondition formula has the important property that, if the method is executed in a state that satisfies the formula, then the postcondition will be established. Having computed this formula, automated verifiers then invoke, for example, an SMT solver to check if the user-specified method precondition implies the weakest precondition.

Both techniques — symbolic execution and verification condition generation— thus ultimately invoke SMT solvers to discharge proof obligations. Verifiers based on symbolic execution, however, typically query the solver very often, at every step of the symbolic execution, but each query is comparatively short and the information the solver has is limited, for example, to the current execution path through the method.

Verifiers based on verification condition generation, on the other hand, rarely query the solver, for example, only once per method, but each time with substantial formulas that encode the whole method body.

Compared to verification condition generation, the stepwise nature of symbolic execution allows a more fine-grained control over the verification process, which facilitates the development of efficient verifiers: in particular, because it enables the use of dedicated data structures and algorithms, for example, to represent and manipulate parts of the symbolic state, and it enables selecting when to make which information from the symbolic state available to the underlying solver.

Building on this observation, the developers of Smallfoot [11], the first automated verifier for separation logic, developed a style of symbolic execution that is tailored to verifying programs with permission-based specifications, and that facilitates the use of off-the-shelf SMT solvers by verifiers that ultimately handle a custom extension of first-order logic (such as separation logic). The defining characteristic of Smallfoot-style symbolic execution is the separation of the symbolic state into *path conditions*, corresponding to program assertions expressed in first-order logic, for example, equalities between symbolic values (*pure* assertions, Chapter 2), and a *symbolic heap*, corresponding to assertions not expressed in first-order logic, for example, permissions to fields, and predicate and magic wand instances (*spatial* assertions, Chapter 2). Specialised algorithms and data structures are then employed to handle the latter, and only the first-order-logic assertions are passed to the underlying SMT solver.

The state separation typically improves performance, but also entails incompletenesses that are common among verifiers based on Smallfoot-style symbolic execution. Such incompletenesses arise if facts that are available in the path conditions have not yet been propagated to the symbolic heap, and vice versa. Consider, for example, the Viper assertion `acc(x.f, p) && acc(y.f, q) && R`: the permissions to `x.f` and `y.f` are part of the symbolic heap, whereas `R` is (assumed to be) pure and thus part of the path conditions. If `R` implies `1 < p + q`, then the semantics of the separating conjunction and of permissions in turn imply that `x` and `y` cannot be aliases. Similarly, if `R` implies that `x` and `y` are aliases, then this in turn implies `acc(x.f, p + q)`. However, both examples require reasoning across the separated state components, which requires appropriate support from the verifiers. To improve performance, verifiers often maintain a representation of the symbolic heap that under-approximates permissions (which does not allow the reasoning required by the second example).

Since its introduction through Smallfoot, this style of symbolic execution has been successfully used and extended by several other verifiers, such as jStar [42], VeriFast [67] and VeriCool [121] (recall also Section 1.2), and it has become the predominant implementation technique for permission-based verifiers. Experience gained in the context of Chalice has shown [72] that verifiers based on Smallfoot-style symbolic execution usually perform better than comparable verifiers based on verification condition generation; an experience that has also been reported elsewhere [118, 119].

Chapter Overview

This chapter presents symbolic execution rules for Viper, which constitute the core of Silicon, Viper's symbolic-execution-based verifier. The work was influenced strongly by Smans' work on symbolic execution for implicit dynamic frames [119], in particular with respect to how the symbolic state is represented and how heap snapshots are used to frame heap-dependent expressions (Section 3.1.2), how the symbolic execution is built-up from four core execution primitives (Section 3.1.3), and the general way in which the rules are presented.

The main contributions of this chapter are the following:

- A *classification* of aliasing-related incompletenesses (Section 3.4.2) that can arise from the separation of the symbolic state (as previously illustrated), and that

are commonly exhibited by verifiers based on Smallfoot-style symbolic execution (Section 3.7.3 discusses to which degree related verifiers exhibit such incompletenesses).

- A *lazy, failure-driven* technique that reduces the number of situations in which such incompletenesses arise in practice, without noticeably degrading performance (Section 3.4.2).
- Support for quantifiers over heap-dependent expressions, which are a cornerstone of full-functional verification, but which have not previously been supported by verifiers based on symbolic execution (Section 3.4.3).
- A technique for joining symbolic execution paths that is a prerequisite for supporting quantifiers over heap-dependent expressions (Section 3.4.3).
- Support for Viper's permission introspection features (`perm` and `forperm`) which, for soundness, requires a precise representation of the permissions available in a symbolic state, whereas verifiers typically use an under-approximation-based heap representation.
- A technique for axiomatising heap-dependent functions to an SMT solver that facilitates reuse of techniques that have been proposed in the context of verification condition generation, and that significantly improves completeness with respect to reasoning about heap-dependent functions.

The remainder of this chapter is structured as follows: initially, the necessary background definitions (Section 3.1) and the symbolic execution primitives that are the basic building blocks of the verifier (Section 3.1.3) are introduced. Next, the symbolic rules for executing statements (Section 3.2), for producing (inhaling) and consuming (exhaling) assertions (Section 3.3), and for evaluating expressions (Section 3.4) are presented. Afterwards, it is shown how to establish the validity of specifications such as method contracts (Section 3.5), and how to axiomatise heap-dependent functions (Section 3.6). The chapter continues with a thorough evaluation of Silicon (Section 3.7), which includes a comparison with related work (Section 3.7.3), before it concludes with a discussion of known limitations (Section 3.8).

3.1 Technical Prelude

3.1.1 Language

Figure 3.1 shows the subset of Viper for which symbolic execution rules are given in this chapter. The subset covers all of Viper (as defined in Figure 2.2), except

- quantified permissions and magic wands, which are discussed in Chapter 4 and Chapter 5, respectively
- field declarations, which are effectively ignored by the symbolic execution: only the used field identifiers and the corresponding types matter
- custom domain declarations because domain axioms (and domain function declarations) are heap-independent, and directly translated to the underlying SMT solver
- `goto` statements: Viper supports reducible control flow graphs in which back edges are annotated with invariants, such that they can be treated analogous to `while` loops during verification; handling forward edges in a symbolic execution is straightforward. To simplify the presentation of the symbolic execution rules, goto statements are therefore omitted.
- `constraining` blocks, which are not yet fully supported by Silicon (see also Section 3.8), and the partial support is not essential for the discussion of the symbolic execution rules

- `let` expressions, whose support is conceptually straightforward, but slightly complicates the presentation (Silicon fully supports `let` expressions)

Relative to Silicon's actual implementation, the following changes to the syntax of expressions have been made:

- In the implementation, permissions to fields are denoted by `acc(`e`.`f`, `p`)` (where p denotes a permission-typed expression) and to predicate instances by `acc(`$pred$`(`$\overline{e}$`), `p`)`. For uniformity, the notation `acc(`id`(`$\overline{e}$`), `p`)` is used in the formalisation to denote permissions to either a predicate instance or a field. In the latter case, $\overline{e}$ is a singleton list: the field receiver. That is, p permissions to a field $e.f$ are denoted by `acc(`f`(`e`), `p`)`.
- Standard `old` expressions are subsumed by labelled `old` expressions: instead of the actual Viper syntax `old(`e`)`, the formalisation uses the syntax `old[pre](`e`)` and assumes that `pre` is a predefined label that denotes the prestate of the method invocation to which the `old` expression conceptually belongs.
- For simplicity, universal and existential quantifiers may only have a single trigger with a single expression inside.
- The formalisation supports a shape of `forperm` expressions that is more general than the shape that Viper currently allows: Viper's `forperm` expressions are limited to matching against fields (syntax `forperm[`f`] `x`: `T` :: `e), whereas the formalisation supports matching against fields and predicate instances (syntax `forperm` $\overline{x : T}$ `:: {`id`(`$\overline{e}$`)} `e). More details are given in Section 3.4.4.

program	::=	$\overline{decl}$
decl	::=	*preddecl* | *funcdecl* | *methdecl*
preddecl	::=	`predicate` *pred*`(`$\overline{x\texttt{: }T}$`) {`a`}`
id	::=	f | *pred*
funcdecl	::=	`function` *func*`(`$\overline{x\texttt{: }T}$`):` T `  requires` a `  ensures` e `{`e`}`

methdecl	::=	`method` *meth*`(`$\overline{x\texttt{: }T}$`) returns (`$\overline{y\texttt{: }T}$`)`
		`  requires` a
		`  ensures` a
		`{`*stmt*`}`

stmt	::=	`var` x`:` T | x `:=` e | x `:= new(`$\overline{f}$`)` | $\overline{x}$ `:=` *meth*`(`$\overline{e}$`)` | x`.`f `:=` e | *stmt*`;` *stmt* | `label` *lb* | `inhale` a | `exhale` a | `assert` a | `fold acc(`*pred*`(`$\overline{e}$`),` e`)` | `unfold acc(`*pred*`(`$\overline{e}$`),` e`)` | `if (`e`) {`*stmt*`} else {`*stmt*`}` | `while (`e`) invariant` a `{`*stmt*`}`
a	::=	e | `acc(`id`(`$\overline{e}$`),` e`)` | `[`a`,` a`]` | a `&&` a | e `?` a `:` a
e	::=	op`(`$\overline{e}$`)` | e`.`f | *func*`(`$\overline{e}$`)` | e `?` e `:` e | `old[`lb`](`e`)` | `unfolding acc(`*pred*`(`$\overline{e}$`),` e`) in` e | `perm(`id`(`$\overline{e}$`))` | `forall` $\overline{x\texttt{: }T}$ `::` $\{e\}$ e | `exists` $\overline{x\texttt{: }T}$ `::` $\{e\}$ e | `forperm` $\overline{x\texttt{: }T}$ `::` $\{id(\overline{e})\}$ e

Figure 3.1: The subset of Viper covered in this chapter. Overlining denotes repetition. Metavariable f ranges over fields, T over types, x, y over variables, *pred* over predicates, *func* over heap-dependent functions, *meth* over methods, and *lb* over labels. `acc(`$id(\overline{e})$`)` denotes permissions to a field (of a single receiver) or to a predicate instance (with an arbitrary number of arguments). op denotes any heap-independent function of arbitrary arity, including (dis)equality; arithmetic, relational and boolean functions (and literals); set, sequence and multiset functions; and custom domain functions. Custom domain declarations are omitted; for simplicity, they are treated in this thesis as theories natively supported by the underlying SMT solver.

In the rest of this chapter, all programs are assumed to be syntactically well-formed in the usual sense: all used symbols (types, fields, variables, labels, methods etc.) are properly declared, currently in scope, and of the expected type and arity, and all expressions are well-typed. `old` expressions may only occur in postconditions of functions and methods, and in method bodies.

3.1.2 Initial Definitions and Background Theory

In order to define the symbolic execution rules for the subset of Viper shown in Figure 3.1, a few initial definitions are necessary.

Definition 1. Let foldl be the fold-left function as available in functional programming languages such as Haskell. We omit its (standard) implementation, but we briefly recapitulate its signature

$$\mathsf{foldl}\colon \overline{X} \to Y \to (X \to Y \to Y) \to Y$$

which can be understood as follows: the first parameter $\overline{X}$ denotes a collection (for example, a sequence, a set or a map) of elements of some type X; the second parameter of some type Y denotes the initial accumulator value; the third parameter of type $X \to Y \to Y$ denotes a combinator function that is applied pointwise to each collection element and the current accumulator in order to obtain the next accumulator. The result of foldl is the final accumulator.

We take the liberty of using n-ary functions in positions where a function of an n-tuple is expected, and vice versa. This avoids cluttering the formalisation with trivial operations that translate between tuples and multiple arguments, for example, when applying foldl. ◁

Definition 2. The empty set symbol $\emptyset$ is used to denote empty sets, multisets, and maps; set union $\cup$ and set subtraction $\setminus$ are also overloaded and used with sets, multisets, and maps. Curly braces denote set literals, for example, $\{a, b, c\}$. Updating a map, denoted by $m[k \mapsto v]$, yields a map m' that is equivalent to m, except that m' maps k to v, that is, $m'(k) = v$.

Square brackets denote list literals, for example, $[a, b, c]$, and the empty sequence is denoted by $[\,]$. Prepending an element a to a sequence l is denoted by $a :: l$, and the concatenation of two lists l_1 and l_2 is denoted by $l_1 ::: l_2$.

Iterated operators are used with sets and sequences, for example, $\bigwedge vs$ denotes the conjunction of all elements of the set or sequence vs. Whenever convenient, syntactic repetition such as $\overline{e}$ is used analogously with iterated operators, as in $\bigwedge \overline{e}$. ◁

Definition 3. Let S be the type of statements, with typical element *stmt*; let A be the type of assertions, with typical element a; let E be the type of expressions, with typical element e in case of arbitrary expressions and p in case of permission-typed expressions; and let *Var*, a subtype of E, be the type of local variables.

Textual substitution of t_3 (an expression, an assertion or a statement) for every occurrence of t_2 in t_1 is denoted as $t_1[t_2 \mapsto t_3]$. ◁

Definition 4. Let R be the type of *verification results*: either success or failure. Verification results can be composed using a short-circuiting $\wedge$ operator. If Q is an operation returning a verification result, then the verification result of success $\wedge\, Q$ is Q's result, whereas failure $\wedge\, Q$ short-circuits to failure without executing Q. ◁

Symbolic Values and Expressions

Definition 5. Let V be the type of *symbolic values*, with typical element v, and let *Perm*, a subtype of V, be the type of symbolic permission values. ◁

Many symbolic execution rules evaluate a program expression such as e to a *symbolic expression* that denotes the symbolic value of e. In order to emphasise the relation between program expressions and their respective symbolic values, we may use primed versions for the symbolic expression that denotes the program expression's symbolic value: for example, a program expression e evaluates to a symbolic value denoted by e'. In program snippets, we alternatively may use different fonts to distinguish between program expressions and their symbolic counterparts: for example, a program variable `x` and its symbolic value x.

Symbolic expressions are terms and formulas of a many-sorted first-order logic (in practice, the syntactical subset of formulas as defined by the SMT-LIB standard). For each Viper type (built-in such as `Int`, and custom such as the `Array` type from Section 2.5.1), we employ a matching sort in the signature of our logic. The signature also includes the usual arithmetic operators, relational symbols and boolean connectives, as well as an operator $ite(v_1, v_2, v_3)$ that denotes the function that returns v_2 if v_1 is true, and v_3 otherwise. The sort that corresponds to Viper's permissions type `Perm` can either be the rational or the real numbers, with multiplication, addition and subtraction. Write permissions (`write` in Viper) are interpreted as the value 1, no permissions (`none` in Viper) as 0.

Furthermore, let the signature contain sorts for Viper's collection types, that is, for `Set[`T`]`, `Seq[`T`]` and `Multiset[`T`]`, and functions representing the necessary collection operations, for example, set contains and sequence concatenation. Silicon does not reason about these collections itself, it merely translates collection-typed expressions to a corresponding symbolic expression, and delegates the entire reasoning to the underlying SMT solver. Since SMT solvers (such as Z3 [92]) typically do not natively support such collections, Silicon uses an axiomatisation of sets, sequences and multisets that is emitted to the solver as part of Silicon's background theory. The axiomatisation is, modulo minor changes, the same as used by Dafny [82]. The axiomatisation is omitted from this thesis since it is irrelevant for the discussion of the symbolic execution.

In order to simplify the presentation of the symbolic execution rules, we take the liberty of mixing program and symbolic expressions, for example, by substituting a sub-expression (of a larger program expression) for the symbolic expression it has been evaluated to. It will be ensured (by construction) that the substituted symbolic expression's sort corresponds to the type of the replaced program expression.

Fresh Identifiers

Definition 6. An identifier or a symbol is *fresh* with respect to a point during an ongoing symbolic execution if the identifier syntactically differs from all identifiers that have been used in the execution performed so far.

We use fresh in positions where any kind of identifier is expected. For example, $\iota :=$ fresh introduces a fresh identifier referred to by ι. fresh is always used such that the context uniquely defines which kind of fresh identifier is needed, for example, a fresh symbolic value or a fresh label identifier. In the case of symbolic values, the context also uniquely determines the sort of the fresh symbolic value, and we therefore omit any kind of sort annotation. ◁

Symbolic States

Definition 7. Let Σ denote *symbolic states*, with typical element σ. Symbolic states are immutable records with at least the following entries (further entries are added in later chapters):

- A *store* γ of type Γ that maps local variables to their symbolic values (that is, Γ is the map type $Var \to V$).

- A *path condition stack* π of type Π that records all assumptions that have been made (phrased differently: all constraints that have been collected) on the current verification path. Path conditions are symbolic expressions of boolean sort, including quantified formulas.
- A *symbolic heap* h of type H that records which locations, that is, which fields, respectively, predicate instances, are currently accessible and which symbolic values they have.
- A *labelled heaps* map *lbh* of type $Label \to H$ that maps label identifiers to symbolic heaps[1].

◁

Definition 8. We use curly braces to denote the construction of symbolic states: for example, $\{\gamma := \emptyset, \pi := \emptyset, h := \emptyset, lbh := \emptyset\}$ denotes the empty state, that is, a state with each state entry initialised with its respective empty value. We also use curly braces to denote state updates: for example, $\sigma\{\gamma := \sigma.\gamma[\mathsf{b} \mapsto true]\}$ denotes a state σ' that is an exact copy of σ, except that the store of σ' maps the local variable b to the symbolic value *true*. ◁

The havoc function shown in Figure 3.2 is used to update a store by assigning a fresh symbolic value to each variable in a given collection of variables.

```
1 havoc: Γ → Var̄ → Γ
2 havoc(γ, x̄) =
3   foldl(x̄, γ, (λ x_i, γ_i · γ_i[x_i ↦ fresh]))
```

Figure 3.2: Havocing the values of variables in a symbolic store.

Definition 9. Path condition stacks π of type Π are sequences of *path condition scopes*, where each scope is a triple $(Id, V, Set[V])$ consisting of a unique *scope identifier*, a *branch condition*, and a set of *path conditions*. ◁

Scope identifiers are used to determine the branch and path conditions newly obtained between two points during a symbolic execution, branch conditions represent the condition of branches taken during an execution (they typically come from conditionals in the program or in the specifications), and path conditions represent all other constraints that have been collected during the execution. In comparison with related work (for example, [119]), our representation of path conditions as a stack of triples may look surprisingly complicated, but it is necessary in order to formalise *joining* verification paths, as is discussed in Section 3.4.3.

The union of all path conditions and branch conditions contained in a given path condition stack yields the set of constraints that have been collected on the corresponding verification path. For convenience, we may use the term *path conditions* to refer to this union in situations where the exact structure of a path condition stack does not matter.

Figure 3.3 shows functions used by the symbolic execution to manipulate path condition stacks. pc-add adds a single path condition to the top-most path condition scope; lifting pc-add such that it adds a collection of path conditions is straightforward, and we use pc-add as if it were lifted accordingly.

pc-push pushes a new path condition scope onto a given stack of path conditions, whereas pc-after returns only those scopes of a given path condition stack that have been pushed after a specific scope (identified via its scope identifier), including that scope itself.

[1] The current symbolic heap h is essentially a special case of a labelled heap and could be replaced by looking up a dedicated label in the map of labelled heaps.

```
1  pc-add: Π → V → Π
2  pc-add(π, v) =
3    Let (id, bc, pcs) :: suffix match π
4    (id, bc, pcs ∪ {v}) :: suffix
5
6  pc-push: Π → Id → V → Π
7  pc-push(π, id, bc) =
8    (id, bc, ∅) :: π
9
10 pc-after: Π → Id → Π
11 pc-after(π, id) =
12   Let prefix :::[(id, bc, pcs)] ::: suffix match π
13   prefix :::[(id, bc, pcs)]
14
15 pc-all: Π → Set[V]
16 pc-all(π) =
17   foldl(π, ∅, (λ (id_i, bc_i, pcs_i), all_i · all_i ∪ {bc_i} ∪ pcs_i))
```

Figure 3.3: Functions operating on path condition stacks: adding path conditions, pushing new path condition scopes, retrieving path conditions recorded after a certain point, and retrieving all recorded path conditions.

pc-all flattens a path condition stack by conjoining branch and path conditions across all scopes of the given stack. Intuitively, the set of path conditions returned by pc-all is the set of all constraints collected on the verification path represented by the given stack.

Definition 10. Symbolic heaps h of type H are multisets of *heap chunks* of the shape $id(\overline{v};\overline{w})$, where id denotes a field or a predicate identifier. The semicolon separates two sequences of arguments: the arguments before the semicolon are effectively in-arguments, the remaining are out-arguments. More details follow shortly. ◁

Definition 11. If id denotes a field, such a chunk is called a *field chunk* and its shape is $id(r;v,p)$, where r, v and p are symbolic expressions that denote the receiver of the location identified by $r.id$ (in the program), the location's symbolic value and the permission amount provided by the chunk, respectively. Intuitively, a field chunk can be understood as a points-to predicate from separation logic.

If id denotes a predicate, then the chunk is called a *predicate chunk* and its shape is $id(\overline{arg};s,p)$, where $\overline{arg}$ are the arguments of the corresponding predicate instance, s is the *snapshot* of the predicate and p (as before) denotes the permissions provided by that chunk. ◁

For uniformity, we may use the term *snapshot* to refer to predicate chunk snapshots and field chunk values.

The snapshot of a predicate represents the values of the heap locations abstracted over by the predicate, and is used to frame heap-dependent expressions that depend on these heap locations: for example, a heap-dependent function that requires a predicate instance in its precondition, which it then unfolds in the body (via an `unfolding`). More details about snapshots are given in Section 3.2.

Definition 12. Let Id be the type of *chunk identifiers*, with elements of the shape $id(\overline{v})$, obtained by omitting the second argument sequence (the out-arguments) of a heap chunk. ◁

Chunk identifiers are used for finding chunks in a symbolic heap that match a field access (and analogous for predicate instances): for example, in order to evaluate a field read $e.f$, which requires determining if the location is accessible and what its symbolic value is. This is implemented by finding a chunk $f(e';v,p)$ that matches

the field chunk identifier $f(e')$ (where e' denotes the symbolic value corresponding to e), and if the chunk provides sufficient permissions ($0 < p$), then v denotes the required symbolic value.

The previous example illustrates why the first sequence of arguments of a heap chunk denotes in- and the second denotes out-arguments: finding a heap chunk (in a symbolic heap) is essentially a function of the in-arguments that returns the out-arguments.

The following functions are used to manipulate symbolic heaps:

$$\mathsf{heap\text{-}add}\colon H \to \Pi \to \mathit{Id} \to \mathit{Snap} \to \mathit{Perm} \to (H, \Pi)$$
$$\mathsf{heap\text{-}rem}\colon H \to \Pi \to \mathit{Id} \to \mathit{Perm} \to (H \to \mathit{Snap} \to R) \to R$$

heap-add adds permissions to a heap, whereas heap-rem removes permissions. Intuitively, adding permissions adds a new heap chunk that provides the added permissions (an operation that cannot fail), and removing permissions subtracts the required permissions from chunks in the heap. The latter can fail if the heap holds fewer permissions than need to be removed. An initial implementation of these functions is given in Section 3.3, and later on refined in Section 3.4.2 to account for joining execution paths.

Checking and Asserting Path Conditions

Figure 3.4 shows two functions that check and assert, respectively, if a constraint, that is, a boolean-sorted symbolic expression, is true in a given state. Intuitively, the difference between *checking* and *asserting* a constraint is that the former may or may not succeed in order for the verification to continue, whereas the latter must succeed. For example, checking constraints is used to implement optimisations such a pruning infeasible verification paths, whereas constraints are asserted during the verification of a method postcondition.

check queries the underlying SMT solver to see if a given constraint is valid in a given state. We assume that the solver returns either true or false, indicating whether or not the constraint is valid. Since proof obligations arising from Viper programs are in general undecidable, the solver may also return "unknown", indicating that its proof search did not yield a definite result. "unknown" is interpreted as a negative result: that is, as false.

assert lifts the boolean result of check to a verification result (of type R). In practice, assert takes a verification failure message as an additional argument which is wrapped by the returned failure, but for simplicity, failure messages are generally omitted from the presentation of the symbolic execution.

```
1  check: Π → V → Bool
2  check(π, v) = pc-all(π) ⊢_SMT v
3
4  assert: Π → V → R
5  assert(π, v) =
6    if check(π, v) then success()
7    else failure()
```

Figure 3.4: Checking path conditions.

Snapshots and Heap-Dependent Functions

Definition 13. Let *Snap*, a subtype of V, be the sort of *snapshots*, and let the signature of the logic contain the following snapshot-related functions:

$$\begin{aligned}&\mathit{unit}\colon \mathit{Snap}\\ &\mathit{pair}\colon \mathit{Snap} \to \mathit{Snap} \to \mathit{Snap}\\ &\mathit{first}\colon \mathit{Snap} \to \mathit{Snap}\\ &\mathit{second}\colon \mathit{Snap} \to \mathit{Snap}\end{aligned}$$

The constant *unit* is the empty snapshot, *pair* is used to construct pairs of snapshots, and *first* and *second* are used to deconstruct pairs of snapshots into their constituents. Constructing and deconstructing snapshots is axiomatised as follows:

$$\forall s_1, s_2\colon \mathit{Snap} \cdot \mathit{first}(\mathit{pair}(s_1, s_2)) = s_1 \wedge \mathit{second}(\mathit{pair}(s_1, s_2)) = s_2$$

In addition, let the signature of the logic contain the following family of functions (parameterised by the sort S):

$$\begin{aligned}&\mathit{box}_S\colon S \to \mathit{Snap}\\ &\mathit{unbox}_S\colon \mathit{Snap} \to S\end{aligned}$$

box_S is used to embed values of some sort S in the snapshot sort, and unbox_S is the corresponding inverse function that inverts the embedding. For a fixed sort S, the two functions are axiomatised as follows:

$$\begin{aligned}&\forall v\colon S \cdot \mathit{unbox}_S(\mathit{box}_S(v)) = v\\ &\forall s\colon \mathit{Snap} \cdot \mathit{box}_S(\mathit{unbox}_S(s)) = s\end{aligned}$$

◁

For brevity, applications of these functions are omitted from the presentation of the symbolic execution rules. For example, given a symbolic value 0 of sort *Int*, is is implicitly understood that $\mathit{first}(\mathit{pair}(0, \_))$ is of sort *Int* (and yields symbolic value 0) because it abbreviates $\mathit{unbox}_{\mathit{Int}}(\mathit{first}(\mathit{pair}(\mathit{box}_{\mathit{Int}}(0), \_)))$.

Definition 14. Let the signature of the logic contain, for each heap-dependent function occurring in the program under verification

$$\texttt{function}\ \mathit{func}(\overline{x : T})\colon T_r$$

of arity n, a corresponding function symbol of arity $n + 1$

$$\mathit{func}\colon \overline{x : S} \to \mathit{Snap} \to S_r$$

where each sort S_i is the sort corresponding to the type of the ith argument T_i, and where sort S_r corresponds to the return type T_r. ◁

The additional function argument is the snapshot of the function, which represents the symbolic values of the heap locations a function depends on, and is used to frame function applications across heap modifications. An application $\mathit{func}(\overline{e})$ of a heap-dependent function (in a program) is encoded as the symbolic function application $\mathit{func}(\overline{e'}, s)$, where $\overline{e'}$ are the symbolic arguments, and where s is the function snapshot. More details about the treatment of heap-dependent functions are given in Section 3.4.3 and Section 3.6.

3.1.3 Symbolic Execution Primitives

The symbolic execution engine implemented in Silicon is based on the four execution primitives shown below, which execute statements, produce (inhale) and consume (exhale) assertions, and evaluate expressions, respectively. These primitives, which are elaborated in the remainder of this chapter, are presented in *continuation-passing style* [49]: the last argument of each primitive is a *continuation*, that is, a function that represents the remaining symbolic execution that still needs to be performed (in the presentation of symbolic execution rules, continuations are typically denoted by the letter Q, as in Figure 3.5).

$\mathsf{exec}\colon \Sigma \to S \to (\Sigma \to R) \to R$
$\mathsf{produce}\colon \Sigma \to A \to \mathit{Snap} \to (\Sigma \to R) \to R$
$\mathsf{consume}\colon \Sigma \to A \to (\Sigma \to \mathit{Snap} \to R) \to R$
$\mathsf{eval}\colon \Sigma \to E \to (\Sigma \to V \to R) \to R$

exec (elaborated in Section 3.2) executes a statement in a given symbolic state, and (potentially) invokes its continuation (third parameter) with an updated state. produce (Section 3.3) implements inhaling an assertion, and its continuation is therefore invoked with an updated state as well. The snapshot parameter of produce determines the symbolic values of heap locations to which permissions are inhaled (if any). consume (Section 3.3) is the dual operation: it implements exhaling an assertion, and the additional snapshot argument passed to its continuation represents the symbolic values of heap locations to which permissions have been exhaled. eval (Section 3.4) symbolically evaluates an expression in a given heap, which yields a symbolic expression, and a potentially updated state in cases where new path conditions were obtained during the evaluation. eval is also used in a lifted version that evaluates a list of expressions to a list of corresponding symbolic expressions.

3.2 Executing Statements

In addition to the functions introduced in Section 3.1.2 that operate on states, and to the four execution primitives introduced in Section 3.1.3, a number of auxiliary functions are used in Silicon's symbolic execution rules. These auxiliary functions are gradually introduced in the remainder of this chapter, starting with Figure 3.5.

For a given assertion and a permission value, scale multiplies the permission value of each accessibility predicate occurring in the assertion by the permission value passed to scale. The implementation of scale is omitted, it can be implemented as a straightforward recursive transformer over the structure of assertions.

$\mathsf{scale}\colon A \to E \to A$
$\mathsf{scale}\colon A \to \mathit{Perm} \to A$
$\mathsf{scale}(a,\ p) = \text{replace every } \mathtt{acc}(id(\overline{e}),\ q) \text{ in } a \text{ with } \mathtt{acc}(id(\overline{e}),\ p \cdot q)$

$\mathsf{branch}\colon \Sigma \to V \to (\Sigma \to R) \to (\Sigma \to R) \to R$
$\mathsf{branch}(\sigma,\ v,\ Q_v,\ Q_{\neg v}) =$
  $(\mathsf{if}\ \neg\mathsf{check}(\sigma.\pi,\ \neg v)\ \mathsf{then}\ Q_v(\sigma\{\pi := \mathsf{pc\text{-}push}(\sigma.\pi,\ \mathsf{fresh},\ v)\})$
  $\mathsf{else}\ \mathsf{success}())$
    $\wedge$
  $(\mathsf{if}\ \neg\mathsf{check}(\sigma.\pi,\ v)\ \mathsf{then}\ Q_{\neg v}(\sigma\{\pi := \mathsf{pc\text{-}push}(\sigma.\pi,\ \mathsf{fresh},\ \neg v)\})$
  $\mathsf{else}\ \mathsf{success}())$

Figure 3.5: Auxiliary functions for scaling assertions by permissions and for branching the flow of the symbolic execution. The feasibility check is an optimisation, performed to avoid exploring known-infeasible branches.

The signature of scale is overloaded in order to allow scaling accessibility predicates by a permission-typed Viper expression, respectively, by a permission-sorted symbolic value. The latter results in a Viper assertion with an (already evaluated) symbolic expression spliced in (recall that we generally allow mixing program and symbolic expressions in this way).

branch enables splitting the symbolic execution into two paths: one path (Q_v) is taken under the assumption that v is true, the second path ($Q_{\neg v}$) is taken under the assumption that v is false. As an optimisation, a feasibility check is performed to avoid exploring known-infeasible branches (on which the additional assumption about v would contradict the current path conditions).

$\mathsf{exec}(\sigma_1, s_1; s_2, Q) =$
  $\mathsf{exec}(\sigma_1, s_1, (\lambda\ \sigma_2 \cdot$
    $\mathsf{exec}(\sigma_2, s_2, Q)))$

$\mathsf{exec}(\sigma, \mathtt{var}\, x\colon T, Q) =$
  $Q(\sigma\{\gamma := \mathsf{havoc}(\sigma.\gamma, x)\})$

$\mathsf{exec}(\sigma_1, x := e, Q) =$
  $\mathsf{eval}(\sigma_1, e, (\lambda\ \sigma_2, e' \cdot$
    $Q(\sigma_2\{\gamma := \sigma_2.\gamma[x \mapsto e']\})))$

$\mathsf{exec}(\sigma_1, x.f := e, Q) =$
  $\mathsf{eval}(\sigma_1, e, (\lambda\ \sigma_2, e' \cdot$
    $\mathsf{consume}(\sigma_2, \mathtt{acc}(x.f, 1), (\lambda\ \sigma_3, \_ \cdot$
      $\mathsf{produce}(\sigma_3, \mathtt{acc}(x.f, 1)\ \&\&\ x.f == e', Q)))$

$\mathsf{exec}(\sigma_1, x := \mathtt{new}(\overline{f}), Q) =$
  $\sigma_2 := \sigma_1\{\gamma := \mathsf{havoc}(\sigma_1.\gamma, x)\}$
  $\mathsf{produce}(\sigma_2, \overline{\mathtt{acc}(x.f, 1)}, \mathsf{fresh}, Q)$

$\mathsf{exec}(\sigma_1, \overline{z} := meth(\overline{e}), Q) =$
  $\mathsf{eval}(\sigma_1, \overline{e}, (\lambda\ \sigma_2, \overline{e'} \cdot$
    $\mathsf{consume}(\sigma_2, meth_{pre}[\overline{x \mapsto e'}], (\lambda\ \sigma_3, \_ \cdot$
      $\sigma_4 := \sigma_3\{\gamma := \mathsf{havoc}(\sigma_3.\gamma, \overline{z}), lbh := \sigma_3.lbh[pre \mapsto \sigma_1.h]\}$
      $\mathsf{produce}(\sigma_4, meth_{post}[\overline{x \mapsto e'}][\overline{y \mapsto z}], \mathsf{fresh}, (\lambda\ \sigma_5 \cdot$
        $Q(\sigma_5\{lbh := \sigma_5.lbh[pre \mapsto \sigma_1.lbh(pre)]\})))))))))$

where $meth_{pre}$ / $meth_{post}$ denotes the pre-/postcondition of method $meth$, where $\overline{x}$ are the method's formal input parameters, and where $\overline{y}$ are its formal output parameters; recall that state entry *lbh* represents the labelled-old heap and that *pre* is a predefined label

$\mathsf{exec}(\sigma_1, \mathtt{inhale}\, a, Q) =$
  $\mathsf{produce}(\sigma_1, a, \mathsf{fresh}, Q)$

$\mathsf{exec}(\sigma_1, \mathtt{exhale}\, a, Q) =$
  $\mathsf{consume}(\sigma_1, a, (\lambda\ \sigma_2, \_ \cdot$
    $Q(\sigma_2)))$

$\mathsf{exec}(\sigma_1, \mathtt{assert}\, a, Q) =$
  $\mathsf{consume}(\sigma_1, a, (\lambda\ \_, \_ \cdot$
    $Q(\sigma_1)))$

```
39 exec(σ1, fold acc(pred(ē), p), Q) =
40   eval(σ1, p :: ē, (λ σ2, p′ :: ē′ ·
41     assert(σ2.π, 0 ≤ p′) ∧ (
42       bdy := scale(pred_body[x̄ ↦ ē′], p′)
43       consume(σ2, bdy, (λ σ3, s·
44         produce(σ3, acc(pred(ē′), p′), s, Q))))))
```

where $pred_{body}$ denotes the body of predicate *pred*, and where $\overline{x}$ are the predicate's formal parameters

```
45
46 exec(σ1, unfold acc(pred(ē), p), Q) =
47   eval(σ1, p :: ē, (λ σ2, p′ :: ē′ ·
48     assert(σ2.π, 0 ≤ p′) ∧ (
49       bdy := scale(pred_body[x̄ ↦ ē′], p′)
50       consume(σ2, acc(pred(ē′), p′), (λ σ3, s ·
51         produce(σ3, bdy, s, Q))))))
52
53 exec(σ, label lb, Q) =
54   Q(σ{lbh := σ.lbh[lb ↦ σ.h]})
55
56 exec(σ1, if (e) {stmt1} else {stmt2}, Q) =
57   eval(σ1, e, (λ σ2, e′ ·
58     branch(σ2, e′,
59       (λ σ3 · exec(σ3, stmt1, Q)),
60       (λ σ3 · exec(σ3, stmt2, Q)))))
61
62 exec(σ1, while (e) invariant a {stmt}, Q) =
63   γ2 := havoc(σ1.γ, x̄)
64   produce(σ1{h := ∅, γ := γ2}, a && e, fresh, (λ σ3 ·
65     exec(σ3, stmt, (λ σ4 ·
66       consume(σ4, a, (λ _, _ · success()))))))
67     ∧
68   consume(σ1, a, (λ σ2, _ ·
69     produce(σ2{γ := γ2}, a && !e, fresh, Q)))
```

where $\overline{x}$ are the loop targets, that is, local variables that are declared outside of the loop but assigned to inside the loop

Figure 3.6: Rules for the symbolic execution of statements.

Figure 3.6 presents the symbolic execution rules for executing statements. The rules are presented in continuation-passing style, and pattern matching is used to define the individual rules, for example, on line 1 and line 5. Pattern matching is also used to indicate that arguments (or return values) have a certain structure, as on line 40. An underscore denotes a pattern that matches everything without binding it, and thus indicates that an argument (or return value) is irrelevant; for example, on line 14. Symbols such as variables are declared either by binding them via pattern matching, or on first use, for example, on line 18. Sequencing is expressed via indentation: lines on the same indentation level, for example, line 18 and line 19, are sequentially composed. Sequentially composed statements form a statement block, and the return value of such a block is the return value of its last statement (as in Scala): for example, the result of executing a new statement (line 17) is the result of the production in line 19. To improve readability, statement blocks may be surrounded by parentheses and indented: for example, the block on the lines 41 – 44, which returns a verification result (recall that verification results can be composed via their short-circuiting $\wedge$ operator).

Most of the symbolic execution rules shown in Figure 3.6 are self-explanatory, but a few need additional explanations.

An assignment $x.f$ `:=` e is executed by first obtaining the symbolic value of the right-hand-side expression, followed by consuming write permission to $x.f$. The consumption achieves two things: it ensures that sufficient permission is available, and that the location's current value is discarded (that is, havoced). How the latter is technically achieved will become clear once the rule for symbolically evaluating a field read has been presented (Figure 3.10). Afterwards, the previously exhaled write permission is restored, and the location's havoced (that is, fresh) value is constrained to equal the assigned value.

A `new` statement havocs the target variable and adds write permissions per field identifier f_i included in the `new` statement. Note that the presented rule has been simplified such that it does not encode the fact that the reference to a newly instantiated object (here: the target variable) is different from each already existing reference In many cases, reference disequalities are implied by the held permission amounts (see also Section 3.4.2), but this does not always suffice. In its implementation, Silicon additionally assumes that the reference to a newly instantiated object is different from all references directly reachable in the current heap, which includes sets, sequences and multisets of references, but not reference-typed locations folded in predicates. Although incomplete, this approach works quite well in practice; more details are given in the comparison to related work in Section 3.7.3.

Executing a method call $\overline{z}$ `:=` $meth(\overline{e})$ proceeds by consuming the method's precondition, with the actual (symbolic) arguments replacing the formal input parameters, followed by havocing the call targets, and by producing the method's postcondition, again with appropriate substitutions for the formal input and output arguments. Before the callee's postcondition is produced, the heap in which the call is executed (that is, the current heap) is installed as the callee's `old` heap by updating the corresponding binding in the labelled-heaps maps. The latter is undone before the execution continues after the call.

An `assert` statement checks the given assertion by exhaling it, but it afterwards continues in the pre-exhale state. This gives `assert` the semantics of checking, but not removing, permissions.

Folding a predicate instance, and unfolding it afterwards without temporarily "giving it away" in between (exhaling it, and then inhaling an instance with the same arguments but potentially different heap values), should not incur a loss of information: that is, it should be possible to frame the values of heap-dependent expressions across such `fold`-`unfold` pairs. Consider, for example, the snippet

```
predicate pair(x: Ref) { acc(x.f) && acc(x.g) }

inhale acc(x.f) && acc(x.g) && x.f == x.g
fold acc(pair(x))
// ... arbitrary code that does not exhale acc(pair(x))
unfold acc(pair(x))
assert x.f == x.g
```

To enable such framing, Smans [119] introduced the concept of predicate chunk snapshots, which represent the values of the heap locations the predicate abstracts over. In the previous example, `pair` abstracts over `x.f` and `x.g`, and the snapshot of the folded `pair` instance thus is the pair of values (v_1, v_2), which in turn are the symbolic values of `x.f` and `x.g`. That is, they are the snapshots of the field chunks obtained from inhaling permissions to these locations, and removed as part of the `unfold`. When the predicate instance is unfolded, the reverse takes place: the `pair` instance chunk is removed in exchange for two field chunks, whose respective snapshots are v_1 and v_2; the final assertion in the previous example therefore succeeds.

The symbolic execution rules for `fold` and `unfold` realise this approach to framing as follows: a `fold` statement `fold acc(`$pred(\overline{e})$`,` p`)` is executed by consuming the

appropriately scaled body of the predicate, followed by producing the folded predicate instance. Consuming (Figure 3.9) the body yields a snapshot that captures the values of those heap locations to which (some) permissions have been consumed; (v_1, v_2) in the previous example. The obtained snapshot is passed to the subsequent produce (Figure 3.8) and thus used as the snapshot of the chunk corresponding to the newly produced predicate instance.

An `unfold` statement is analogously executed, but this time, the snapshot is taken from the chunk corresponding to the unfolded predicate instance and used to determine the values of the (newly) produced heap locations. In the previous example, the predicate chunk's snapshot of the predicate (still) is (v_1, v_2), and producing the predicate body with this snapshot adds two field chunks to the heap, corresponding to `x.f` and `x.g` and with snapshots v_1 and v_2, respectively.

Loops are symbolically executed in two steps: the first step verifies the loop body and re-establishes the invariant, the second step ensures that the invariant can be established before the loop, and it sets up the state in which the execution proceeds after the loop. In the first step, the invariant and the negated loop condition are produced into a state with an empty heap and havoced loop targets, the body is executed in the resulting state, and the postcondition is consumed afterwards. The second step consumes the invariant from the initial state, which havocs locations to which the loop required all available permissions. Subsequently producing the invariant (and the negated loop guard) results in a state corresponding to the program location after the loop, in which the execution is continued.

3.3 Producing and Consuming Assertions

The rules for producing and consuming assertions are shown in Figure 3.8 and Figure 3.9, respectively. Before presenting the rules, a few additional auxiliary functions — shown in Figure 3.7 — need to be introduced that are used by the production and consumption rules.

heap-add adds permissions to a state by adding a new heap chunk that provides the corresponding permission amount. The dual operation is heap-rem, which removes permissions from a state by trying to find a single chunk that provides at least the permission amount that needs to be removed, and (if successful) by replacing the found chunk with an updated chunk that provides the remaining permissions. If no single chunk provides sufficient permission, a verification failure is raised.

There is an obvious discrepancy between heap-add and heap-rem: heap-add adds permissions by creating new chunks, and due to fractional permissions, a heap might contain multiple chunks for the same chunk identifier (for example, field location). heap-rem, on the other hand, only looks at each chunk in isolation. This discrepancy would result in several heap-related incompletenesses which are discussed in Section 3.4.2, and as a consequence, a refined version of heap-add is presented there.

```
1  heap-add(h, π, id(v̄), s, p) =
2    (h ∪ {id(v̄; s, p)}, π)
3
4  heap-rem(h1, π1, id(v̄), p, Q) =
5    if (∃ id(w̄; s, q) ∈ h1 · check(π1, ⋀ v̄ = w̄ ∧ p ≤ q)) then
6      h2 := (h1 \ {id(w̄; s, q)}) ∪ {id(w̄; s, q − p)}
7      Q(h2, s)
8    else
9      failure()
```

Figure 3.7: The first implementation of adding and removing permissions. The existential denotes a pattern match operation, and is not a logical existential (and not passed to the underlying solver): it denotes an iteration (in arbitrary order) over heap chunks in order to find one that matches the existential's body. The existential also acts as a binder (here: of $\overline{w}$, s and q) whose scope extends to the end of the rule.

```
1  produce(σ, v: V, s, Q) =
2    Q(π{π := pc-add(σ.π, {v, s = unit})})
3
4  produce(σ1, e, s, Q) =
5    eval(σ1, e, (λ σ2, e' ·
6      produce(σ2, e', s, Q)))
7
8  produce(σ1, acc(id(ē, p)), s, Q) =
9    eval(σ1, p :: ē, (λ σ2, p' :: ē' ·
10     Let v be e' ≠ null if id denotes a field, and true otherwise
11     (h3, π3) := heap-add(σ2.h, σ2.π, id(ē'), s, p')
12     π4 := pc-add(π3, {v, 0 ≤ p})
13     Q(σ2{h := h3, π := π4}))))
14
15 produce(σ1, a1 && a2, s, Q) =
16   produce(σ1, a1, first(s), (λ σ2 ·
17     produce(σ2, a2, second(s), Q)))
18
19 produce(σ1, e ? a1 : a2, s, Q) =
20   eval(σ1, e, (λ σ2, e' ·
21     branch(σ2, e',
22       (λ σ3 · produce(σ3, a1, s, Q)),
23       (λ σ3 · produce(σ3, a2, s, Q)))))
24
25 produce(σ, [a1, _], s, Q) =
26   produce(σ, a1, s, Q)
```

Figure 3.8: Producing (inhaling) assertions.

Figure 3.8 shows the symbolic execution rules for producing assertions. As before, only the interesting cases are discussed. The first rule, with the explicit type ascription $v\colon V$, matches symbolic expressions (such as those previously spliced into an assertion), and adds them to the set of path conditions. The rule also adds the path condition $s = \mathit{unit}$, making it explicit that the production snapshot s does not denote any heap values.

Producing a conjunction a_1 && a_2 produces a_1 followed by a_2, and indirectly adds the path condition that the snapshot of the conjunction is a pair of snapshots, whose components are the snapshot of a_1 and a_2. Producing an accessibility predicate yields the additional assumptions that the produced permission amount is non-negative and that field receivers are non-null; both properties are checked by the corresponding consume rule (presented next).

The rules for consuming assertions are shown in Figure 3.9. Function consume itself creates a copy of the current heap, the *consume heap*, from which consume' then removes the permissions required by the assertion to consume. If consume' succeeds, the execution continues in the left-over heap.

The initial heap is duplicated in order to ensure that exhaling an assertion such as `acc(x.f) && x.f == 0` does not fail because of the left-to-right order in which conjunctions are handled: consuming the accessibility predicate removes permissions to `x.f` from the heap, and evaluating the field read in the resulting heap would otherwise then fail. Expressions such as `x.f == 0` are therefore evaluated in the original heap, while permissions are taken from the consume heap.

Analogous to the first rule for producing assertions, the first consume' rule matches symbolic expressions, which are asserted to be true. Consuming a symbolic expression yields the empty heap snapshot *unit*, thereby matching the corresponding produce rule. All other rules are analogous to their produce counterparts.

```
1  consume(σ1, a, Q) =
2    consume'(σ1, σ1.h, a, (λ σ2, h2, s ·
3      Q(σ2{h := h2}, s)))
4
5  consume' : Σ → H → A → (Σ → H → Snap → R) → R
6
7  consume'(σ, h, v: V, Q) =
8    assert(σ.π, v) ∧
9      Q(σ, h, unit)
10
11 consume'(σ1, h, e, Q) =
12   eval(σ1, e, (λ σ2, e' ·
13     consume'(σ2, h, e', Q)))
14
15 consume'(σ1, h1, acc(id(ē), p), Q) =
16   eval(σ1, p :: ē, (λ σ2, p' :: ē' ·
17     Let v be e' ≠ null if id denotes a field, and true otherwise
18     assert(σ2.π, 0 ≤ p' ∧ v) ∧ (
19       heap-rem(h1, σ2.π, id(ē'), p', (λ h2, s ·
20         Q(σ2, h2, s)))))
21
22 consume'(σ1, h1, a1 && a2, Q) =
23   consume'(σ1, h1, a1, (λ σ2, h2, s1 ·
24     consume'(σ2, h2, a2, (λ σ3, h3, s2 ·
25       Q(σ3, h3, pair(s1, s2))))))
26
27 consume'(σ1, h, e ? a1 : a2, Q) =
28   eval(σ1, e, (λ σ2, e' ·
29     branch(σ2, e',
30       (λ σ3 · consume'(σ3, h, a1, Q)),
31       (λ σ3 · consume'(σ3, h, a2, Q)))))
32
33 consume'(σ, h, [_, a2], Q) =
34   consume'(σ, h, a2, Q)
```

Figure 3.9: Consuming (exhaling) assertions.

Representing Partial Heaps as Snapshots

Snapshots are used to represent the values of the heap locations to which an assertion includes permissions, that is, to the partial heap the assertion describes. Following Smans' work [119], Silicon represents snapshots as binary trees (nested pairs), whose structure is determined by the structure of the source assertion, in particular by the separating conjunctions that occur in the assertion. Leaves in the tree correspond to either accessibility predicates or pure sub-assertions (of the overall assertion): for accessibility predicates, the leaf is the value of the corresponding heap location (which for predicate instances is itself a snapshot representing the partial heap described by the instance), and for pure sub-assertions, the leaf is the *unit* value, expressing that the sub-assertion does indeed not describe the shape of the heap.

Since snapshots represent heap values, they can be used to preserve values across modifications of the symbolic heap, including `fold`-`unfold` pairs (as discussed in Section 3.2): on fold, the partial heap described by the predicate instance's body is removed (by removing permissions) in exchange for the folded instance, and vice versa on unfold. Folding the body yields a snapshot that describes the values in the removed partial heap, which is stored alongside the folded predicate instance (as the

snapshot of the corresponding predicate chunk), and then used on unfold in order to "restore" the values of the partial heap to which permissions are (re)gained.

Snapshots can be obtained in two ways: by constructing a snapshot tree during the consumption of an assertion (Figure 3.9), and by exploring (or deconstructing) a given snapshot during the production of an assertion (Figure 3.8). For example, folding a predicate proceeds by consuming the body (and storing the obtained snapshot in the new predicate chunk), and unfolding proceeds by using that snapshot (taken from the chunk) in the production of the predicate body in order to determine the values (snapshots) of the newly produced heap chunks.

Preserving heap values this way requires (1) that the rules for producing and consuming assertions are symmetrical with respect to how they (de)construct snapshots (which they are), and (2) that the structure of the snapshot always matches the structure of the assertions it is used with. In particular, it is only sound to use a snapshot (obtained from consuming an assertion) in the production of the very same source assertion, which includes the structure of the assertion.

In Silicon, snapshots that are used two or more times (such as the snapshots involved with (un)folding predicates) are by-construction always used with the same source assertion. However, due to inhale-exhale assertions, the structure of an assertion when consumed might nevertheless differ from the structure it has upon production: recall that an inhale-exhale assertion $[a_1, a_2]$ is produced as a_1 but consumed as a_2.

In the context of this thesis, we therefore disallow occurrences of inhale-exhale assertions in places where they would result in the (in general unsound) use of snapshots with assertions of varying structure. This disallows inhale-exhale assertions in predicate bodies (as discussed, the same snapshot is consumed and produced) and function preconditions (similar), but for example, not in method specifications: in the latter case, the snapshot obtained from consuming the precondition at call site is discarded (see the rule for executing method invocations in Figure 3.6) and not used to produce the precondition when verifying the method body (discussed in Section 3.5). Note that this restriction only applies to *spatial* inhale-exhale assertions (which include accessibility predicates), not to *pure* inhale-exhale expressions, since the latter always yield the *unit* snapshot, independent of its further structure. More details about this limitation of the use of inhale-exhale assertions are provided in Section 3.8.

3.4 Evaluating Expressions

The rules for symbolically evaluating expressions are in general more involved than the symbolic execution rules presented so far, which is a consequence of the diversity of Viper's expression language, and cover language features such as `unfolding` expressions, heap-dependent function applications, quantifiers and permission introspection. The presentation of the evaluation rules is therefore divided into three subsections, with an additional subsection dedicated to making information that is implicitly available in the heap explicitly available to the underlying solver as path conditions.

3.4.1 Basic Evaluation Rules

The first group of evaluation rules is shown in Figure 3.10. The rule for $op(\bar{e})$ matches any heap-independent function of arbitrary arity, including built-in boolean, arithmetic and relation operators, domain function applications, and set, sequence and multiset expressions; all of which can be represented by a corresponding symbolic

function application $op'(\overline{e'})$. For partial functions such as division or sequence indexing, appropriate checks need to be performed that ensure that the function is defined for its arguments.

Evaluating a heap read $e.f$ succeeds if a heap chunk for $e.f$ can be found that provides non-zero permissions, in which case that chunk's snapshot is the symbolic value resulting from the evaluation. If no such chunk exists, the verification fails. Recall (Figure 3.7) that the existential denotes a pattern match operation that iterates over all heap chunks, it is not a logical existential (and not passed to the underlying solver). Evaluating a labelled `old` expression proceeds by temporarily making the heap stored under the given label the current heap, in which the nested expression is then evaluated as usual.

$$
\begin{array}{l}
1\ \ \mathsf{eval}(\sigma_1,\, v\colon V,\, Q) = Q(\sigma_1,\, v) \\
2 \\
3\ \ \mathsf{eval}(\sigma_1,\, x,\, Q) = Q(\sigma_1.\gamma(x)) \\
4 \\
5\ \ \mathsf{eval}(\sigma_1,\, op(\overline{e}),\, Q) = \\
6\ \ \quad \mathsf{eval}(\sigma_1,\, \overline{e},\, (\lambda\ \sigma_2,\, \overline{e'}\ \cdot \\
7\ \ \quad\quad Q(\sigma_2,\, op'(\overline{e'})))) \\
8 \\
9\ \ \mathsf{eval}(\sigma_1,\, e.f,\, Q) = \\
10\ \ \quad \mathsf{eval}(\sigma_1,\, e,\, (\lambda\ \sigma_2,\, e'\ \cdot \\
11\ \ \quad\quad \mathsf{if}\ (\exists\ f(v; s,\, p) \in \sigma_2.h\ \cdot \mathsf{check}(\sigma_2,\, v = e' \wedge 0 < p))\ \mathsf{then} \\
12\ \ \quad\quad\quad Q(\sigma_2,\, s) \\
13\ \ \quad\quad \mathsf{else} \\
14\ \ \quad\quad\quad \mathsf{failure}()) \\
15 \\
16\ \ \mathsf{eval}(\sigma_1,\, \texttt{old[}lb\texttt{](}e\texttt{)},\, Q) = \\
17\ \ \quad \mathsf{eval}(\sigma_1\{h \coloneqq lbh(lb)\},\, e,\, (\lambda\ \sigma_2,\, e'\ \cdot \\
18\ \ \quad\quad Q(\sigma_2\{h \coloneqq \sigma_1.h\},\, e)))
\end{array}
$$

Figure 3.10: A first set of symbolic evaluation rules: for local variables, heap-independent functions, field reads and labelled `old` expressions.

3.4.2 Overcoming Heap Incompletenesses

Verifiers based on Smallfoot-style symbolic execution, which maintain the symbolic state as two separate components (a set of path conditions and a set of heap chunks), commonly exhibit heap-related incompletenesses[2]that arise from (1) the heap representation and management, which typically approximates the semantics of the spatial aspects of a permission logic such as permissions and the separating conjunction, and from (2) the state separation, which complicates proofs that require the combination of spatial and pure reasoning, that is, of facts that are implied by the symbolic heap with facts implied by path conditions.

Despite being commonly exhibited by verifiers based on Smallfoot-style symbolic execution, however, there has not yet been an attempt (that we are aware of) to systematically categorise such incompletenesses, to identify different kinds of incompleteness, or to describe usage patterns that often cause incompletenesses.

To address this issue, we have identified three kinds of heap-related incompleteness (which, without taking further measures, would be exhibited by the symbolic execution rules presented so far), alongside different aliasing patterns that potentially complicate overcoming these incompletenesses. Afterwards, we describe a novel

[2]Orthogonal to this discussion are incompletenesses that arise from other sources such as a solver's incomplete support for certain theories, or generally undecidable theories.

approach to reducing the number of situations in which these incompletenesses can arise; the approach combines algorithms that "exchange" information between the separate state components with a strategy for lazily applying these algorithms. Finally (in Section 3.7), we compare Silicon's implementation of this approach to other verifiers based on Smallfoot-style symbolic execution, and demonstrate that the technique indeed reduces the likelihood that incompletenesses can be observed.

Kinds of Incompleteness

A first kind of incompleteness can potentially be observed when exhaling permissions. Recall the asymmetry between the implementations of heap-add and heap-rem from Figure 3.7: heap-add potentially adds multiple heap chunks for the same heap location, and semantically, the total permission amount (to a single location) held by such a heap is the permission sum across all heap chunks for that location. heap-rem, in contrast, looks at each chunk in isolation and greedily tries to find a *single* matching chunk that provides sufficient permission. As a consequence of this asymmetry, the following example, in which code comments are used to show the relevant parts of the symbolic state, would incorrectly fail to verify:

```
inhale acc(x.f, 1/2) && acc(x.f, 1/2)                    (~Ex. 1-1~)
  // h: f(x; _, 1/2), f(x; _, 1/2)
assert acc(x.f) // Might fail
```

This example can be made more challenging (for a symbolic execution engine) by introducing *definite* aliasing between x and y:

```
inhale acc(x.f, 1/2) && acc(y.f, 1/2)                    (~Ex. 1-2~)
  // h: f(x; _, 1/2), f(y; _, 1/2)
inhale x == y
assert acc(x.f) // Might fail
```

The complexity can be increased further by introducing *disjunctive* aliasing, which describes situations in which a reference is an alias of at least one out of several other references, as illustrated by the next example:

```
inhale acc(x.f) && acc(y.f)                              (~Ex. 1-3~)
inhale z == x || z == y
assert acc(z.f) // Might fail
```

A second, strongly related kind of incompleteness can potentially be observed when reading a field. Consider the following example:

```
inhale acc(x.f, 1/2) && acc(x.f, 1/2) && x.f == 0        (~Ex. 2-1~)
  // h: f(x; s1, 1/2), f(x; s2, 1/2)
  // π: s1 = 0
exhale acc(x.f, 1/2)
  // h: f(x; s2, 1/2)
  // π: s1 = 0
assert x.f == 0 // Might fail (s2 = 0 unknown)
```

Recall the evaluation rule for field reads from Figure 3.10: the rule selects the first matching heap chunk (that provides non-zero permissions) and returns its value as the result of the field read. As a consequence, the information that a particular heap location has a specific value could be tied to a single heap chunk, as in the previous example. If this chunk could no longer frame its value, that is, if all permissions were removed from the chunk, then the relevant information would be lost.

Analogous to the example illustrating the first kind of incompleteness, this example can also be complicated further by introducing definite or disjunctive aliasing. The aliasing information can also be introduced *retrospectively*, that is, only after already having exhaled permissions to the (then) aliased location:

```
inhale acc(x.f, 1/2) && acc(y.f, 1/2) && x.f == 0      (~Ex. 2-2~)
exhale acc(x.f, 1/2)
inhale x == y   // Retrospectively learn aliasing
assert y.f == 0 // Might fail
```

A third kind of incompleteness can arise when facts implied by the semantics of permissions are not available to the solver:

```
inhale acc(x.f) && acc(y.f) && z == x                  (~Ex. 3-1~)
assert x != y // Might fail
inhale acc(z.f, 1/10)
assert false  // Might fail
```

As before, the example can be further complicated by introducing aliasing, either immediately or retrospectively.

Overcoming Incompletenesses

Completely overcoming all of the incompletenesses would require the symbolic execution to take a *global* view on the heap and its evolution: it would require changing the rules for adding and removing permissions, respectively, and for reading fields, such that all possible aliasing relations are taking into account; and in addition, it would require some notion of the previous evolution of the heap in order to account for retrospective aliasing.

Intuitively, this (likely to be expensive) strategy would result in a symbolic execution engine that mimics the total heaps encoding of permission logics that verifiers based on verification condition generation such as Chalice (and Viper's second verifier, Carbon) achieve by encoding heaps and permissions as mathematical maps. Such verifiers typically do not exhibit any of the aforementioned incompletenesses.

Due to the potential complexity of the previously sketched strategy, Silicon implements an incomplete (but sound) approach where the greedy matching approach described for removing permissions and evaluating field reads is complemented by *state consolidations* which are performed in order to reduce the number of cases in which incompletenesses arise from this greediness. (Note that we also explored alternative, non-greedy approaches: to a limited extent in the context of permission introspection, discussed in Section 3.4.4, and extensively in the context of quantified permissions, discussed in Chapter 4.)

State consolidations potentially affect both components of a symbolic state: they can make facts that are implied by the symbolic heap available to the solver (by adding them as new path conditions), for example, non-aliasing relations that follow from permission amounts, and they can rewrite the symbolic heap according to information available in the path conditions, for example, merge chunks whose receivers are aliased.

The extent to which state consolidations improve completeness is controlled by two orthogonal parameters: to which degree (or which patterns of) aliasing is taking into account, and how frequently they are applied. We concentrate on aliasing first, and then discuss frequency of application.

State Consolidation Algorithms

A major task performed by Silicon's state consolidation is merging field chunks that definitely correspond to the same heap location, that is, chunks (for the same field) whose receivers are *definitely* aliased; merging chunks means adding up their permission amounts and equating their snapshots. Predicate chunks are merged analogously: if all arguments are definitely equal. In addition, state consolidations also infer reference disequalities from the chunks' permission values. This approach

usually works well in practice and overcomes the incompletenesses illustrated by the previously shown examples Ex. 1-1, 1-2, 2-1 and 3-1. However:

(1) It does not prevent incompletenesses which are due to disjunctive aliasing. (Ex. 1-3). To our knowledge, no verifier based on symbolic execution is complete (with respect to the discussed heap-related incompletenesses) in the presence of disjunctive aliasing; see also the comparison in Section 3.7.3.

(2) It does not prevent incompletenesses which are due to retrospective aliasing (Ex. 2-2). We are not aware of a comparable verifier that is complete in the presence of retrospective aliasing; see also Section 3.7.3.

(3) It still exhibits certain aliasing-related incompletenesses of the second kind of incompleteness. The situation in which such incompletenesses can arise is characterised in the remainder of this subsection. Comparable verifiers exhibit similar incompletenesses, but to a greater extent, as shown in Section 3.7.3.

These remaining incompletenesses can often be overcome by adding (ghost) code. For example, incompletenesses arising from disjunctive aliasing can be overcome by forcing the symbolic execution engine to branch such that the disjunctive aliasing is resolved and reduced to definite aliasing.

```
1  consolidate: H → Π → (H, Π)
2  consolidate(h, π) =
3    h_i := h; π_i := π
4    do {
5      (h_i, π_i) := merge-into(∅, π_i, h_i)
6      π_i := infer-disequalities(h_i, π_i)
7      π_i := assume-valid-permissions(h_i, π_i)
8    } while (h_i or π_i changed)
9    (h_i, π_i)
10
11 merge-into: H → Π → H → (H, Π)
12 merge-into(h_dst, π, h_src) =
13   foldl(h_src, (h_dst, π), (λ id_i(v̄_i, s_i, p_i), (h_dst_i, π_i) ·
14     if (∃ id_i(w̄_i; t_i, q_i) ∈ h_dst_i · check(π_i, ⋀ v̄_i = w̄_i)) then
15       s := fresh
16       s_def := (0 < p_i ⇒ s = s_i) ∧ (0 < q_i ⇒ s = t_i)
17       h_dst_{i+1} := h_dst_i \ {id_i(w̄_i; t_i, q_i)} ∪ {id_i(w̄_i; s, p_i + q_i)}
18       (h_dst_{i+1}, pc-add(π_i, s_def))
19     else
20       (h_dst_i ∪ {id_i(v̄_i; s_i, p_i)}, π_i)))
21
22 infer-disequalities: H → Π → Π
23 infer-disequalities(h, π) =
24   Let ps be the set of all unordered pairs {f_i(v_i, _, p_i), f_i(w_i, _, q_i)}
25         of different field chunks in h that have the same field identifier
26   foldl(ps, π, (λ {f_i(v_i, _, p_i), f_i(w_i, _, q_i)}, π_i ·
27     if check(π_i, p_i + q_i > 1) then pc-add(π_i, v_i ≠ w_i)
28     else π))
29
30 assume-valid-permissions: H → Π → Π
31 assume-valid-permissions(h, π) =
32   Let h_f ⊆ h contain all heap chunks for identifier f
33   foldl(h_f, π, (λ f_i(v_i, _, p_i), π_i ·
34     pc-add(π_i, p_i ≤ 1)))
```

Figure 3.11: Silicon's state consolidation algorithms. merge-into merges a heap h_{src} into h_{dst} by merging definitely-aliased chunks. infer-disequalities makes reference disequalities implied by permission amounts explicit by adding corresponding path conditions. assume-valid-permissions adds the assumption that a well-formed state cannot hold more than write permission to a field.

A simplified version of Silicon's state consolidation algorithm is shown in Figure 3.11. The actually implemented version contains several optimisations that gain performance in practice but do not affect the worst-case complexity, which is cubic in the number of heap chunks.

merge-into merges a heap h_{src} into h_{dst} by iterating over all chunks ch_i in h_{src}, and if it finds a matching chunk ch_j in h_{dst}, it sums up their permission values and equates their snapshots. It is this step that makes aforementioned greedy approaches to finding matching heap chunks acceptable in practice because it consolidates the heap such that a single chunk provides all permissions *definitely* held by the state to a specific location, and because it equates the snapshots of merged chunks and thereby "shares" information that was potentially tied to a single chunk (that is, snapshot). This overcomes the incompletenesses illustrated by the previously shown examples Ex. 1-1, 1-2 and 2-1.

In addition, infer-disequalities adds a reference disequality for each pair of field chunks whose combined permission amount would (definitely) exceed write permissions, and assume-valid-permissions constrains each field chunk to provide at most write permissions, from which the infeasibility of certain verification paths might be concluded. This overcomes the incompletenesses illustrated by Ex. 3-1.

Finally, consolidate consolidates a symbolic state by iteratively applying the previous three functions until no further changes can be made. The fix-point iteration performed by consolidate (which is responsible for the operation's cubic worst-case complexity) is necessary because each iteration potentially adds path conditions that enable a subsequent iteration to consolidate the state further. For example, merging chunks for `x1.f` and `x2.f` adds the equality `x1.f == x2.f`, which in the presence of a path condition such as `x1.f == x2.f ==> x2 == x3` would in turn allow merging chunks for `x2.f` and `x3.f`. In our experience, however, such situations rarely arise in practice and consolidate typically terminates after two iterations.

Termination of consolidate is guaranteed as follows:

(1) A heap contains only finitely many heap chunks, and the repeated invocations of merge-into either successively reduce the number of chunks, or do not change the heap.

(2) The invocations of infer-disequalities and assume-valid-permissions do not affect the heap directly, but the path conditions they add can potentially allow a subsequent merge-into to merge additional chunks.

(3) infer-disequalities and assume-valid-permissions add path conditions on each invocation, but two subsequent invocations add the same path conditions if no chunks were merged in between, which does not change the overall set of path conditions.

State Consolidation Frequency

Next, we discuss the frequency with which state consolidations are performed. An ideal approach with respect to completeness would be to consolidate the state after every state update, in particular, after adding permissions or path conditions. Performing state consolidations so frequently is expensive, since the worst-case complexity of a consolidation is cubic in the number of heap chunks. Moreover, most consolidations would probably be fruitless because in practice, the vast majority of path conditions obtained during a symbolic execution do not affect the heap shape, and is instead concerned with functional properties.

To achieve a practical trade-off between efficiency and completeness, Silicon implements a hybrid approach that combines statically and dynamically chosen consolidation points: by performing a partial consolidation whenever permissions are added to a state (using merge-into to add the new chunk), and by triggering a full state

consolidation on failure, that is, whenever an assertion fails (after which the assertion is checked again).

Merging new chunks into the current heap prevents the incompleteness illustrated by example 1-1, but not the one illustrated by example 1-2 (but the more expensive on-failure state consolidation does). Situations as illustrated by the first example, however, have arisen in practical examples such as the encoding of an AVL-tree that is part of Viper's test suite, where permissions to a location are in general shared between different data structure nodes (such as a parent and its children), but are temporarily combined to perform certain operations (such as inserting a node).

Compared to (only) consolidating on-failure, merging new chunks into the current heap has the advantage that it is relatively cheap (linear instead of cubic complexity). Another potential advantage is a reduced number of assertions that are checked twice, once before (which fails) and once after the on-failure consolidation.

It is of course possible to perform state consolidations at other statically chosen points during the execution, for example, after an assertion such as a precondition has been produced. We experimentally tried a corresponding strategy, but it increased the average verification time by 50%. The runtimes of this experiment are the following (the Viper test suite contained 540 test files when the experiment was conducted; see Section 3.7 for more details on how we evaluate Silicon's performance):

(1) Silicon base line (state consolidations as described here): 205s for all test files

(2) Consolidate after each invocation of produce, but don't merge each new chunk in separately: 310s in total; 5 fewer unexpected errors (prevented incompletenesses); 1 additional incompleteness (due to not merging new chunks in)

(3) Consolidate after each invocation of produce and merge each new chunk in separately: 315s in total; 5 fewer unexpected errors (prevented incompletenesses)

Since it is difficult to statically identify points at which state consolidations improve completeness in a practically relevant way without significantly degrading performance, Silicon instead implements a strategy that enables dynamically choosing the points at which a state consolidation is to be performed. The strategy, implemented by the try operation shown in Figure 3.12, effectively ensures that a state consolidation is performed before a consume operation begins that would otherwise fail, in a way that does not noticeably affect performance in practice: without try, the test suite (which contained 552 tests when the experiment was conducted) completed in 234s and 13 tests failed (due to incompletenesses), whereas with try all tests passed in 237s.

```
1  try: Σ → (Σ → (Σ → R) → R) → (Σ → R) → R
2  try(σ, Q_action, Q) =
3    res_local := failure()
4    res_global :=
5      Q_action(σ,
6            (λ σ_1 ·
7              res_local := success()
8              Q(σ_1)))
9    if res_local is a failure then Q_action(σ{h, π := consolidate(σ.h, σ.π)}, Q)
10   else res_global
11
12 consume(σ_1, a, Q) =
13   try(σ_1, (λ σ_2, Q_succ · consume'(σ_2, σ_2.h, a, Q_succ)), Q)
14
15 heap-add(h, π, id(v̄), s, p) =
16   merge-into(h, π, {id(v̄; s, p)})
```

Figure 3.12: Silicon's try function, implementing a scoped, failure-driven try-react-retry execution flow.

try takes two continuations as its arguments: the *action continuation* (Q_{action}) corresponds to the action that is to trigger a state consolidation if it fails, and that is to be retried after the consolidation. The second continuation (Q), as usual, denotes the remainder of the verification.

In addition, Figure 3.12 shows updated versions of two previously shown rules. The first is an updated version of consume from Figure 3.9, where the actual consume operation is passed as the action continuation to try in order to ensure that the consumption triggers a state consolidation, should it fail, after which it is retried. The second updated rule is an improved version of heap-add from Figure 3.7, which ensures that each newly added heap chunk is immediately merged with a potentially already existing heap chunk with the same identifier.

The invocation of the *success continuation*, which is passed by try to the action continuation (line 6), marks the point at which the action successfully executed (and shifts control back to try). This is necessary in order to limit the scope during which a failure triggers a state consolidation and a subsequent re-execution of the action. Intuitively, the success continuation acts like a backtracking barrier: once passed, a later failure will not make the execution backtrack above the barrier in order to retry an action that was performed above the barrier. In spirit, the success continuation is therefore related to the *cut* in the Prolog language.

To illustrate the effect of the success continuation, consider the following snippet:

```
exhale true
...
exhale false // Expected to fail
```

The consume operation corresponding to the first exhale trivially succeeds, and the verification can be continued. Eventually, the final exhale is reached, which fails, triggers a state consolidation, and fails again. The execution therefore backtracks and propagates the failure upwards, which, *without* the backtracking barrier, would result in a state consolidation before the first exhale, followed by another execution of the whole program. In substantial programs with many verification branches and nested try operations, unrestricted backtracking would most likely have a severe negative effect on performance in cases where the verification is expected to fail.

3.4.3 Branching and Joining Evaluations, and Quantifiers

Recall that Silicon branches over conditionals (as do similar verifiers): for example, when executing if-then-else statements (Figure 3.6), and producing and consuming conditional assertions (Figure 3.8 and Figure 3.9, respectively); performing the dual operation of *joining* execution paths, in order to prevent an exponential blow-up of paths, could thus be considered natural.

Intuitively, execution paths can be joined by taking the different symbolic states at the join point, that is, at the end of the individual paths, and making them conditional on the branch conditions under which they each were obtained. This intuitive idea can be implemented for sets of path conditions by making the individual constraints in the sets conditional on the corresponding branch conditions, and similarly for symbolic heaps: by making the symbolic permission expressions of the individual chunks conditionals of the branch conditions.

Joining path conditions and symbolic heaps in this way is precise with respect to the (ideal) semantics of assertions: an assertion should hold after the join point if and only if it holds on each individual path that reaches it (if the execution were not joined). However, due to Silicon's incomplete heap management described in Section 3.4.2, joining heaps could cause aliasing-related incompletenesses in situations where the conditionals introduced in the permission expressions (by joining chunks) prevent the greedy chunk lookup algorithms (Figure 3.7 and Figure 3.10)

from finding a chunk that *definitely* provides sufficient permissions. Indeed, recall from Section 3.4.2 that aliasing-related incompletenesses can often be overcome by forcing the symbolic execution to branch over conditionals that (per branch) reduce uncertain aliasing relations to definite aliasing, which can be handled by the combination of greedy heap algorithms and state consolidations: heap joins effectively introduce such aliasing-related uncertainty, and thus might prevent the greedy heap algorithms from succeeding. An example that illustrates such incompletenesses (by encoding the potential effects of heap joins) can be found in Listing B.1 in Appendix B.

Joining path conditions only does not entail similar problems, but limits the applicability of joins to execution paths that result in semantically equivalent symbolic heaps. An important subset of the executions for which this is guaranteed is the evaluation of expressions (which branch over conditionals, as is discussed shortly) because expressions are *pure* and their evaluation does thus not modify the heap. This statement may appear to be invalidated by the following two observations: (1) evaluating certain expressions such as `unfolding` a predicate instance at least temporarily modify the heap, and (2) evaluating an expression may trigger a state consolidation and thus change the representation of the heap. However, the first kind of heap modification is indeed only temporary and limited to the scope of the evaluated expressions (and thus "reverted" afterwards), and the second kind can be ignored (that is, the consolidated heap can be replaced with the previous, unconsolidated version) without changing the heap's semantics. Hence, all states that are reached *after* the evaluation of an expression (on potentially different evaluation paths through that expression) can soundly be treated as having the same symbolic heap, that is, the one in which the evaluation started.

Joining evaluation paths yields two results: (1) a symbolic state whose set of path conditions includes the (appropriately conditionalised) path conditions obtained on the individual paths, and (2) a single symbolic expression that jointly represents the symbolic expressions obtained from the individual evaluations.

Joining branching evaluation paths potentially reduces the total number of paths explored by the symbolic execution, but at the cost of more-complex path conditions with additional conditionals, which potentially affects the performance of the solver, for example, by causing case splits in proofs. Experience gained in previous work [72], however, suggests that trading execution paths for disjunctions is beneficial and can improve the overall verification time. Moreover, joining such evaluation paths is even *necessary* in order to support quantifiers, as is discussed next.

Quantifiers over Branching Expressions

Consider this rule *candidate* for evaluating conditional expressions (that we cannot actually use, as is explained shortly), which evaluates conditional expressions analogously to how exec handles conditional statements: by branching over the condition.

$$
\begin{array}{l}
\mathsf{eval}(\sigma_1,\, e_1 \texttt{ ? } e_2 \texttt{ : } e_3,\, Q) = \\
\quad \mathsf{eval}(\sigma_1,\, e_1,\, (\lambda\ \sigma_2,\, e_1' \cdot \\
\qquad \mathsf{branch}(\sigma_2,\, e_1', \\
\qquad\quad (\lambda\ \sigma_3\ \cdot \mathsf{eval}(\sigma_3,\, e_2,\, Q)), \\
\qquad\quad (\lambda\ \sigma_3\ \cdot \mathsf{eval}(\sigma_3,\, e_3,\, Q)))))
\end{array}
$$

In addition, consider the following general strategy for symbolically evaluating quantifiers such as `forall x: T :: e`: (1) add a fresh local variable `y` (with symbolic value y) to the current symbolic store, (2) evaluate the quantifier body with `y` substituted for `x` (here, `e[x` $\mapsto$ `y]`), and (3) obtain the final result by nesting the resulting symbolic expression (here, $e[x \mapsto y]$) with y replaced by x under a (symbolic) quantifier that binds x (here, $\forall x\colon T \cdot e$).

This strategy for evaluating quantifiers, however, does not compose with the previously shown rule candidate for evaluating conditionals, as illustrated by the following quantifier (where `f1` and `f2`, for simplicity, are heap-independent functions):

```
forall x: Int :: x > 0 ? f1(x) : f2(x)
```

Following the general strategy for evaluating quantifiers, the quantifier is instantiated with a fresh variable `y` and the body is evaluated, which branches over the condition `y > 0`: on one path, the branch condition is $y > 0$ and the resulting quantifier is $\forall x\colon Int \cdot f1(x)$; on the other path, the branch condition is $\neg(y > 0)$ and the resulting quantifier is $\forall x\colon Int \cdot f2(x)$. This requantification of partial evaluations of the body (partial with respect to possible paths through the body) is unsound, since the resulting quantifiers fail to reflect the branch condition under which they were obtained (and because the branch conditions included in the path conditions no longer relate to the bound quantified variable).

Incorporating the branch conditions in each quantifier, as in $\forall x\colon Int \cdot x > 0 \Rightarrow f1(x)$ (and analogous for the other path), would be sound but incomplete: for example, it would not be possible to assert `f2(0)` on the path with $\forall x\colon Int \cdot x > 0 \Rightarrow f1(x)$.

To remedy the situation, it is necessary to *join* evaluation paths that branch under a quantifier, and to construct a single symbolic expression that combines the partial evaluations of the body in a way that reflects the branch condition under which they were obtained.

One option is to join paths after evaluating the quantifier body; another option is to join after evaluating a conditional. The options are equivalent in the context of the previous example, the resulting quantifier would be $\forall x\colon Int \cdot ite(x > 0, f1(x), f2(x))$ in both cases (where *ite* is a symbolic conditional).

Joining after evaluating a conditional, however, has a more general impact on the control flow of the symbolic execution and potentially reduces the total number of execution paths, as discussed earlier. Silicon therefore generally joins after evaluating conditionals, not only in the context of quantifiers.

Evaluating Conditional Expressions

Joining evaluation paths can be implemented by a backtracking strategy: each path is explored until the join point is reached, the current state is recorded and the path is terminated, which makes the evaluation backtrack to the branch point and explore the next path. Once all paths have been explored (up to the join point), the information recorded on each path must be joined in a suitable way (explained below), and the symbolic execution proceeds.

```
1  eval(σ1, e1 ? e2 : e3, Q) =
2    eval(σ1, e1, (λ σ2, e1' ·
3      join(σ2,
4        (λ σ3, Q_join ·
5          branch(σ3, e1',
6            (λ σ4 · eval(σ4, e2, Q_join)),
7            (λ σ4 · eval(σ4, e3, Q_join))))),
8        (λ σ3, {({e1'}, e2'), ({¬e1'}, e3')} ·
9          cond := ite(e1', e2', e3')
10         Q(σ3, cond))))
```

Figure 3.13: Evaluating conditional expressions, by branching and joining, to a symbolic conditional.

Figure 3.13 shows a rule for symbolically evaluating conditionals that uses a backtracking-based join function to join the evaluation of the two branches of the conditional. The implementation of join is shown in Figure 3.14, but to build up some intuition, we first illustrate how it is used.

join takes two continuations: a *branching continuation* that corresponds to the execution that potentially branches and whose paths are to be joined, and, as usual, a continuation that denotes the remainder of the execution that is to be executed after the join point. In the rule for evaluating conditionals (Figure 3.13), the branching continuation is defined on line 4, and the remainder continuation is defined on line 8.

The branching continuation is itself invoked with the *join continuation*, denoted by Q_{join} on line 4, as one of its arguments. An invocation of the join continuation marks the join point, that is, the end of the *current* evaluation path, and shifts control back to join in order to either explore the next evaluation path, or create the symbolic conditional that represents the joined paths (line 9). The role of the join continuation is hence similar to that of the success continuation used in the context of the try function from Figure 3.12.

When evaluating a conditional e_1 `?` e_2 `:` e_3, the join point is reached after e_2 and e_3 have been evaluated, and thus right before the remaining evaluation is started by invoking the respective remainder continuation. Consequently, the join continuation is passed as the remainder continuation to the evaluation of e_2 (line 6) and e_3 (line 7). At the end of the evaluation of e_2 (respectively, e_3), the join continuation is then invoked with the current state (which includes the branch conditions of the current path) and the symbolic expression resulting from the evaluation as arguments. Control is shifted back to join, which records the arguments in order to eventually create the final result: the symbolic conditional (line 9).

After the evaluation paths of e_2 and e_3 have been joined, join's remainder continuation (whose definition starts at line 8) is invoked, with the previously recorded symbolic expressions and the branch conditions under which these expressions have been obtained as arguments. In the case of evaluating conditionals, the remainder continuation expects that two paths were joined: one taken under the branch condition e'_1 and with the result e'_2, the other under the branch condition $\neg e'_1$ and with the result e'_3. The final symbolic conditional is created (line 9), and the execution continues.

Joining General Evaluation Paths

Conditional *expressions* (which are pure) are not the only reason why the evaluation of an expression might branch: in addition, any expression whose evaluation potentially involves producing (or consuming) conditional *assertions* (which are spatial) might result in branching evaluations because the produce operation branches over conditionals (and analogous for consume). In Silicon, the only such expressions are `unfolding` expressions and applications of heap-dependent functions: during the evaluating of the former the body of the unfolded predicate is produced before the nested expression is evaluated, whereas evaluating a function application entails consuming the function's precondition (the corresponding evaluation rules are presented in Figure 3.15 and Figure 3.16, respectively, and discussed shortly).

As an illustration, consider the following predicate containing nested spatial conditionals:

```
predicate P(x: Ref, k: Int) {
  x != null
    ? (acc(x.f) && k > 0 && (x.f != null
                               ? (acc(x.f.f) && k == 2)
                               : k == 1))
    : k = 0
}
```

Unfolding the predicate, as in the following assignment to a local variable

```
m := unfolding acc(P(x, k)) in k
```

entails producing the predicate body (before evaluating the body of the `unfolding`), which branches over the conditional assertion (recall Figure 3.8).

Joining the paths through the evaluation of the `unfolding` expression *could* be done by (1) joining right after each conditional *assertion*, similar to the previously presented joining of conditional *expressions*, or (2) by joining after the whole `unfolding` expression (including the nested expression, here `k`) has been evaluated. The first option has the advantage of avoiding nested branches (in contrast to the second option); however, it would require joining different symbolic heaps, which (as already discussed at the beginning of Section 3.4.3) in combination with Silicon's incomplete heap management could cause aliasing-related incompletenesses.

Silicon therefore implements the second option, in a way that is similar to the joining of conditional expressions but that accounts for nested branches:

(Step 1-1) Each path through the evaluation (of which the previous example admits three) is explored all the way to the end of the evaluation (which, in the previous example, includes the evaluation of `k`).

(Step 1-2) Per evaluation path, the branch conditions (such as $x \neq \mathit{null}$), all other path conditions (such as $k > 0$) and the *path result* (which is k on all three paths through the previous `unfolding`) are recorded.

(Step 2-1) Afterwards, the path conditions are joined as a *set of implications*, each expressing that the path conditions recorded on a specific path hold if the corresponding branch conditions are met.

(Step 2-2) The joined path result, on the other hand, must be a *single* symbolic value; in the previously discussed case of joining the evaluation of a conditional *expression* the single value was obtained by constructing an appropriate symbolic *ite* conditional. Since conditional *assertions* admit nested branches which now need to be joined, we replace the symbolic conditional by a (per join) fresh, casewise defined *join function*, whose value is the path result of a specific evaluation path if the corresponding branch conditions hold.

The corresponding implementation is shown in Figure 3.14: a join operation that can be used to join an arbitrary (finite) number of evaluation paths. The operation collects the conditions and the symbolic expressions obtained on each path (corresponding to Step 1-1 and Step 1-2), and joins the path conditions (Step 2-1) but leaves it up to its clients (the remainder continuation) to join the path results (Step 2-2). Building on join, the alternative version join' also joins the latter (that is, it implements Step 2-2) in the form of a casewise defined function with one case per joined path. The signature of join is generic with respect to the types of path results it can handle, abstracted over by the type variable Ω. In all of our use cases Ω is instantiated with V, the type of symbolic values.

join proceeds in two major steps, which realise the previously described steps Step 1-1 and 1-2, respectively, Step 2-1 (Step 2-2 is implemented by join'). In the first step, it invokes the branching continuation (line 5) to collect the necessary data, and if the branching continuation succeeds, join joins the collected data (line 12), before it passes the result to the remainder continuation. The join continuation passed to Q_{branch} (line 7), which is to be invoked at the join point of each path, records the path condition delta (pc-after, line 8) between the states in which the branching evaluation started and ended. The join continuation also records the path result (ω, line 8), and then terminates the current path.

```
1  join: Σ → (Σ → (Σ → Ω → R) → R) → (Σ → Set[(Set[V], Ω)] → R) → R
2  join(σ1, Q_branch, Q) =
3    id := fresh
4    data: Set[(Π, Ω)] := ∅
5    Q_branch(
6      σ1{π := pc-push(σ1.π, id, true)},
7      (λ σ2, ω ·
8        data := data ∪ {(pc-after(σ2.π, id), ω)}
9        success()))
10   ) ∧ (
11     (π2, ωs) :=
12       foldl(data, (σ1.π, ∅), (λ (πi, ωi), (π_joined, ωs_i) ·
13         (cnds, bcs_all) := pc-segs(πi)
14         (pc-add(π_joined, cnds), ωs_i ∪ {(bcs_all, ωi)})))
15     Q(σ1{π := π2}, ωs))
16
17 join′: Σ → (Σ → (Σ → V → R) → R) → (Σ → V → R) → R
18 join′(σ1, Q_branch, Q) =
19   Let jnfn be a fresh function symbol such that
20     1. its arity is |σ2.qvs|
21     2. it can be applied to the argument vector σ2.qvs
22     3. its return sort matches v's sort
23   join(σ1,
24     Q_branch,
25     (λ σ2, (bcs, v)) ·
26       jnfn_def := ⋀ bcs ⇒ jnfn(σ2.qvs) = v
27       Q(pc-add(σ2, jnfn_def), jnfn(σ2.qvs))))
     note that, for the set of currently quantified variables qvs, it holds that σ2.qvs = σ1.qvs
28
29 pc-segs: Π → (Set[V], Set[V])
30 pc-segs(π) =
31   foldl(
32     reverse(π),
33     (∅, true),
34     (λ (_, bci, pcsi), (cndsi, bcsi) ·
35       (cndsi ∪ {⋀(bcsi ∪ {bci}) ⇒ ⋀ pcsi}, bcsi ∪ {bci})))
```

Figure 3.14: Joining evaluation paths. join invokes a potentially branching evaluation, records the obtained path conditions and evaluation results, and afterwards joins the path conditions but not the evaluation results (this is up to clients). join' also joins the latter, in the form of a casewise defined join function. join employs pc-segs to extract, from a given path condition stack, the branch-dependent path conditions and the overall set of branch conditions. The former is used to join path conditions, the latter to join evaluation results.

In join's second step (beginning at line 11), the recorded path conditions (stacks π_i) of each previously explored path are joined, which implements Step 2-1. The recorded path results (ω_i) are not joined by join itself (and instead passed on to its caller, that is, its remainder continuation); it is thus left to the client to implement Step 2-2. One such client is the already presented rule for evaluating conditional expressions (Figure 3.13), which joins the results of *two* paths by constructing an appropriate *ite* expression. Another client is join' (Figure 3.14), which joins arbitrarily many paths by introducing a casewise defined join function. The latter will be discussed shortly.

To extract the path and branch conditions from the recorded stacks π_i, join employs the auxiliary function pc-segs that takes a path condition stack and returns two sets: (1) The path conditions recorded by the stack as *branch-dependent* path conditions,

that is, conditional on the branch conditions under which they were obtained, such as $x \neq null \Rightarrow k > 0$ and $x \neq null \wedge x.f \neq null \Rightarrow k = 2$ for the first path through the previous example of unfolding `P(x, k)` (for simplicity, $x.f$ is used to denote the symbolic value of `x.f`), and (2) the set of all branch conditions recorded by the stack, such as $\{x \neq null, x.f \neq null\}$ for the first path through the previous example.

To illustrate how join and join' (Figure 3.14) implement the four steps Step 1-1 to 2-2, consider once more the previously shown example of unfolding `P(x, k) in k` and assume that join' is used to join the three paths through the unfolded predicate body (each of which ends with the evaluation of the path result `k`). From the definition of join' we see that most of the actual work is performed by join (Step 1-1 to 2-1), join' itself only joins the path results (Step 2-2). We therefore first discuss how the execution of join proceeds for this particular example.

In the first step of the execution of join, the join continuation (line 7) is invoked three times (once per path through the unfolded predicate body), each time with the same path result k (instantiating ω) but with different path conditions; this implements Step 1-1 and Step 1-2. In the second step of join, pc-segs is applied to each of the three recorded path condition stacks (π_i, on line 13) to extract the branch-dependent path conditions (*cnds*) and the overall branch conditions for the whole path (bcs_{all}). The three respective pairs of values for $(cnds, bcs_{all})$ are:

$$\begin{array}{ll}
(\{(x \neq null \Rightarrow k > 0), (x \neq null \wedge x.f \neq null \Rightarrow k = 2)\}, & \{x \neq null, x.f \neq null\}) \\
(\{(x \neq null \Rightarrow k > 0), (x = null \wedge x.f = null \Rightarrow k = 1)\}, & \{x \neq null, x.f = null\}) \\
(\{(x = null \Rightarrow k = 0)\}, & \{x = null\})
\end{array}$$

The branch-dependent path conditions *cnds* are added to the symbolic state (by invoking pc-add on line 14) such that they are available after the join point; this concludes Step 2-1.

join also pairs each path result ω_i (three times k in our example) with the overall branch conditions bcs_{all} (line 14) of the path that yielded the particular path result. These pairs can later on be used to join the path results: in our example, this will be done by join' once the execution of join is finished (and its remainder continuation invoked). In the context of our example, the corresponding pairs of branch conditions and path results, denoted by (bcs_{all}, ω_i) on line 14, are

$$\{(\{x \neq null, x.f \neq null\}, k),\ (\{x \neq null, x.f = null\}, k),\ (\{x = null\}, k)\}$$

which are passed back to join' when join finishes (line 15).

join' (starting on line 25) now performs the remaining Step 2-2 of joining the path results: it introduces a fresh join function (denoted by *jnfn*) and defines it casewise (set $jnfn_{def}$ constructed on line 26) using the branch conditions and path results that join had passed on to join' (the pairs (bcs_{all}, ω_i) that join passed to join' instantiate $\overline{(bcs, v)}$ on line 25). The newly introduced join function jointly represents the path results of the joined paths, and is thus passed to the remainder continuation of join' (line 27) as the (single) path result.

The introduction of the join function is technically slightly more involved, however: when a branching evaluation is joined that is nested (somewhere) under a quantifier, the resulting join function must (for soundness) be a function of the quantified variables. These are recorded in the state entry σ.qvs (which is maintained by the rule for evaluating quantifiers, shown in Figure 3.17), and passed as arguments to the join function constructed by join' (on line 26 and line 27).

Evaluating `unfolding` Expressions

Figure 3.15 shows a rule for evaluating `unfolding` expressions that uses join' to join the potentially branching production of the unfolded predicate body. Regarding the

previous example of unfolding a predicate, the state resulting from evaluating the `unfolding` according to the presented rules is the following:

```
m := unfolding acc(P(x, k)) in k
  // γ: ..., m ↦ jnfn_1
  // π: ..., (x ≠ null ⇒ k > 0), (x ≠ null ∧ x.f ≠ null ⇒ k = 2),
        (x = null ∧ x.f = null ⇒ k = 1), (x = null ⇒ k = 0),
        (x ≠ null ∧ x.f ≠ null ⇒ jnfn_1 = k), (x ≠ null ∧ x.f = null ⇒ jnfn_1 = k),
        (x = null ⇒ jnfn_1 = k)
assert m >= 0 // Succeeds
```

Evaluating `unfolding` expressions can potentially recurse indefinitely (see also the comparison to VeriCool in Section 3.7.3) if the predicate is recursive and contains an `unfolding` of another instance of the same predicate in its body (potentially transitively through other predicate bodies or function preconditions).

In order to prevent such infinite recursive evaluations, one could record the identifiers of visited predicates (for example, in an appropriate state entry), and replace recursing `unfolding` expressions (at a configurable depth) with unknown symbolic values, which would be used instead of the "real" symbolic values to which the expression would evaluate. This approach is sound but potentially incomplete, and raises the question how the depth at which the recursion is cut off should be chosen.

```
1  predicate LL(x: Ref, i: Int) {
2       acc(x.data, 1/2)
3    && acc(x.next)
4    && acc(x.next.data, 1/2)
5    && acc(LL(x.next, i + 1))
6    && (unfolding acc(LL(x.next, i + 1)) in x.next.data == i + 1)
7  }
8
9  method test01(z: Ref) {
10   inhale acc(LL(z, 0))
11
12   assert unfolding acc(LL(z, 0)) in
13             z.next.data == 1 /* Might fail */
14 }
```

Listing 3.1: Illustrating a potential incompleteness arising from naïvely breaking recursive unfolding cycles.

Recall that Viper's design with respect to predicates is to require explicit annotations for (un)folding predicates, and that the verifier does otherwise not attempt to deduce facts from predicate instances. In this light, consider the example shown in Listing 3.1 (which was extracted from a larger example encountered in practice): the `unfolding` expression provided on line 12 explicitly instructs the verifier to use the corresponding predicate body in order to prove the nested assertion `z.next.data == i`. The predicate body contains another explicitly provided `unfolding` on line 6, and making its nested assertion `x.next.data == i + 1` available for the proof of the original goal (`z.next.data == i`) is therefore in line with Viper's treatment of predicates.

In Silicon, expressions nested under `unfoldings` can in general only be evaluated after the predicate has been unfolded, the evaluation might otherwise fail due to missing heap chunks. A suitable rule for evaluating `unfolding` expressions should thus permit the evaluation of the `unfolding` on line 6, but it may skip the next `unfolding` that is recursively reached, which (arguably) is no longer explicitly given in the program (and since `unfolding` expressions cannot nest accessibility predicates, skipping the entire expression cannot result in missing permissions in the context of the ongoing surrounding unfolding).

The approach Silicon implements can thus be characterised as follows: it makes facts entailed by `unfolding` expressions that *explicitly* occur in the program available, but

bounds the evaluation of *implicit* `unfolding` expressions, that is, those transitively reached during the evaluation of explicitly provided ones.

The corresponding evaluation rule is shown in Figure 3.15. The production of the predicate body potentially branches, and the overall evaluation is thus joined in order to support `unfolding` expressions under quantifiers (and to improve performance in general). Infinite unfolding chains are prevented by "evaluating" implicit `unfolding` expressions to fresh symbolic values. Differentiating between explicit and implicit `unfolding` expressions requires some additional bookkeeping, which is omitted for brevity.

Cutting off the recursion at a particular depth is theoretically incomplete, but the approach we chose makes it difficult to actually observe such incompleteness (and we did not yet observe any in practice): in order to observe that an implicit `unfolding` has been cut off and replaced by an arbitrary value, it is necessary to explicitly unfold equally deeply — in which case the previously cut-off `unfolding` will be evaluated (and the next deeper implicit `unfolding` will be cut off).

```
1  eval(σ1, unfolding acc(pred(ē), p) in b, Q) =
2    if the unfolding is explicit then
3      eval(σ1, p :: ē, (λ σ2, p' :: ē' ·
4        bdy := scale(pred_body[x ↦ e'], p')
5        consume(σ2, acc(pred(ē'), p'), (λ σ3, s ·
6          join'(σ3,
7            (λ σ4, Q_join ·
8              produce(σ4, bdy, s, (λ σ5 ·
9                eval(σ5, b, (λ σ6, b' ·
10                 Q_join(σ6{h := σ2.h}, b'))))))),
11           Q))))))
12   else
13     Let recunf be a fresh function symbol such that
14       1. its arity is |σ1.qvs|
15       2. it can be applied to the argument vector σ1.qvs
16       3. its return sort matches b's sort
17     Q(recunf(σ1.qvs))
```

Figure 3.15: Symbolically evaluating `unfolding` expressions in a way that makes all facts entailed by explicitly provided `unfolding` expressions available, but still prevents infinite recursive unfolding chains.

Evaluating Heap-Dependent Function Applications

Figure 3.16 shows a rule for evaluating applications of heap-dependent functions, where join' is used to join the potentially branching consumption of the function's precondition. Consuming the precondition not only checks whether the precondition holds, it also computes the function's snapshot that is used to frame the function application: the result of evaluating a heap-dependent function is a symbolic function application parameterised by the computed snapshot. The discussion of how to give meaning to symbolic function applications is postponed until Section 3.6.

```
1  eval(σ1, func(ē), Q) =
2    eval(σ1, ē, (λ σ2, ē' ·
3      join'(σ2,
4        (λ σ3, Q_join ·
5          consume(σ3, func_pre[x̄ ↦ ē'], (λ σ4, s ·
6            Q_join(σ4{h := σ3.h}, func(ē', s))))),
7        Q)))
```

where $func_{pre}$ does not transitively contain another application of *func*

Figure 3.16: Symbolically evaluating applications of heap-dependent functions.

The rule for evaluating function applications potentially exhibits a recursion problem similar to that of evaluating `unfolding` expressions: the evaluation will recurse indefinitely if consuming the function's precondition entails evaluating another application of the same function. Precisely checking such preconditions at call site, that is, without cutting off potentially infinite recursions, would require a fix-point computation. Since we (but also others [114]) have not yet encountered functions with recursive preconditions in practice, Silicon rejects functions that recurse in their preconditions.

Evaluating Quantifiers

Finally, Figure 3.17 shows the rule for evaluating quantifiers. After having evaluated the trigger and the quantifier body (with the quantified variables $\overline{x}$ instantiated with fresh variables $\overline{y}$), two quantifiers are constructed. The first one, denoted by *quant*, is the actual result of the evaluation, and passed to the remaining execution. The second quantifier, denoted by $quant_{pcs}$, aggregates the additional path conditions obtained from evaluating the quantifier body, for example, the definition of join functions. The path conditions aggregated by the second quantifier constrain symbolic values (potentially) used by the first quantifier.

Both quantifiers use the same triggers to ensure that whenever the solver is in the position to instantiate the first quantifier, the necessary information from the second quantifier is available as well. Note that it would be incomplete to merge the two quantifiers: depending on the context of the ongoing evaluation, the first quantifier may be assumed or asserted, whereas the second quantifier is always to be assumed.

```
1  eval(σ1, forall x̄: T̄ :: {e1(x̄)} e2(x̄), Q) =
2    id := fresh; ȳ' := fresh
3    σ2 := σ1{γ := σ1.γ[ȳ ↦ ȳ'], qvs := σ1.qvs ∪ {ȳ'}}
4    σ3 := σ2{π := pc-push(σ2.π, id, true)}
5    eval(σ3, e1(ȳ), (λ σ4, e1(ȳ)' ·
6      eval(σ4, e2(ȳ), (λ σ5, e2(ȳ)' ·
7        (pcs, _) := pc-segs(pc-after(σ5.π, id))
8        quant := ∀x̄: T̄' · {e1(x̄)'} e2(x̄)'
9        quant_pcs := ∀x̄: T̄' · {e1(x̄)'} ⋀ pcs
10       σ6 := σ5{γ := σ1.γ, π := pc-add(σ5.π, quant_pcs), qvs := qvs \ {ȳ'}}
11       Q(σ6, quant)))))
```

Variables $\overline{y}$ are fresh program variables, that is, they are not already bound in the current store, and $\overline{y'}$ are their fresh symbolic values

Figure 3.17: Symbolic evaluation rule for universal quantifiers (the rule for existential quantifiers is analogous and omitted for brevity). We use the notation $e(\overline{x})$ to emphasise that variables $\overline{x}$ may occur in e (in addition to other variables), and we subsequently use $e(\overline{y})$ to denote the substitution of $\overline{x}$ for $\overline{y}$ in e.

The rule for evaluating quantifiers does not perform any path joining itself, which corresponds to the implicit assumption that potentially branching evaluations are joined in the context of their respective rules. In Silicon's case, these are the previously discussed rules for evaluating conditionals, `unfolding` expressions and heap-dependent function applications.

3.4.4 Permission Introspection

In Section 3.4.2, we argued why it is in general difficult for a symbolic execution engine to avoid heap-related incompletenesses without degrading performance. As a consequence, Silicon implements symbolic execution rules that potentially under-approximate the permissions provided by a symbolic heap; a common design choice among symbolic-execution-based verifiers. In particular, when exhaling permissions (heap-rem from Figure 3.7), which is implemented by trying to find a single chunk that definitely provides the required permission amount.

Using an under-approximation when exhaling permissions is sound because the under-approximation is used as a *lower* bound: exhaling p permissions requires proving that the current heap provides at least p permissions. If the proof succeeds with the under-approximation then it will also succeed with the — potentially higher — real value.

Using an under-approximation to establish an *upper* bound, however, would not be sound (an over-approximation would), and Viper's `perm` feature can be used to express lower *and* upper permission bounds. With an under-approximation, exhaling a lower bound such as `perm(x.f) < q` would be unsound and exhaling an upper bound would be sound (but potentially incomplete), whereas inhaling a lower bound would be sound (but potentially incomplete) and inhaling an upper bound would be unsound.

Precisely representing the permissions a heap provides, however, is cheaper in the context of permission introspection (Viper's `perm` and `forperm` features) than it is in the context of exhaling permissions: the former requires computing permission sums that reflect all potential aliasing relations between field receivers (and analogous for predicate arguments); the latter additionally requires removing permissions by updating heap chunks under consideration of all potential aliasing relations. Moreover, our experience suggests that permission introspection is used only infrequently, whereas exhaling permissions is a central operation of verifiers for permission logics.

.

1 $\mathsf{eval}(\sigma_1, \texttt{perm(}id(\overline{e})\texttt{)}, Q) =$
2 $\quad \mathsf{eval}(\sigma_1, \overline{e}, (\lambda\ \sigma_2, \overline{e'} \cdot$
3 $\quad\quad$ Let $h_{id} \subseteq \sigma_2.h$ contain all heap chunks for identifier id
4 $\quad\quad sum := \mathsf{fold}(h_{id}, 0, (\lambda\ id(\overline{v}; \_, p), q \cdot q + ite(\bigwedge \overline{e' = v}, p, 0)))$
5 $\quad\quad Q(\sigma_2, sum)$
6
7 $\mathsf{eval}(\sigma_1, \texttt{forperm}\ \overline{x\colon T}\ \texttt{::}\ \{id(\overline{e(\overline{x})})\}\ b(\overline{x}), Q) =$
8 $\quad$ Let $h_{id} \subseteq \sigma.h$ contain all heap chunks with identifier id
9 $\quad$ Let $\overline{z\colon T}$ be fresh program variables s.t. $|\overline{z}| = |\overline{x}|$,
10 $\quad\quad$ and let $\overline{z'}$ be corresponding fresh symbolic values
11 $\quad cnj(\overline{z}) := \mathsf{fold}(h_{id}, \texttt{true}, (\lambda\ id(\overline{v}; \_, p), c \cdot$
12 $\quad\quad\quad\quad\quad\quad\quad\quad c\ \texttt{\&\&}\ (p\ \texttt{>}\ \texttt{none}\ \texttt{\&\&}\ \overline{e(\overline{z})\ \texttt{==}\ v}\ \texttt{==>}\ b(\overline{z}))))$
13 $\quad \mathsf{eval}(\sigma_1\{\gamma := \sigma_1.\gamma[\overline{z \mapsto z'}]\}, cnj(\overline{z}), (\lambda\ \sigma_2, cnj(\overline{z})' \cdot$
14 $\quad\quad Q(\sigma_2\{\gamma := \sigma_1.\gamma\}, \forall\ \overline{x\colon T} \cdot cnj(\overline{x})'))$

Figure 3.18: Symbolical evaluation rules for Viper's permission introspection features: `perm` and `forperm`. As before, we use the notation $e(\overline{x})$ to emphasise that variables $\overline{x}$ may occur in expression e. For brevity, we omit triggers from the final quantifier (line 14): suitable triggers should be computed from the user-provided expressions $e(\overline{x})$ and $b(\overline{x})$; see also the discussion of triggers in Section 2.6

As usual in Viper, the semantics of `perm` and `forperm` treat predicate instances as opaque, that is, permissions folded into predicate instances are not accounted for by `perm` and `forperm`.

Figure 3.18 shows Silicon's rule for evaluating `perm` expressions, which sums up permissions across all heap chunks that potentially provide permissions to a location $x.f$ (or analogously, to a predicate instance). The resulting sum includes one summand per field chunk $f(y_i; \_, y_i)$, each of which encodes that the corresponding chunk provides p_i permissions to $x.f$ if the chunk's receiver y_i is an alias of x:

$$ite(y_1 = x, p_1, 0) + \ldots + ite(y_n = x, p_n, 0)$$

Strongly related to `perm` are `forperm` expressions. Intuitively, a `forperm` expression

```
forperm x: Ref :: {x.f} b
```

binds the reference-typed variable `x` to every receiver r for which the heap provides non-zero permissions `r.f`. For each such binding, the body `b` is evaluated. A semantically correct translation of this `forperm` expression would thus be

```
forall x: Ref :: perm(x.f) > none ==> b
```

Evaluating this quantifier, however, will most likely fail due to Silicon's greedy way of evaluating field reads, as discussed in Section 3.4.2. In order to illustrate the problem, consider the example

```
inhale acc(y1.f) && acc(y2.f) && y1.f == 1 && y1.f < y2.f
forperm x: Ref :: {x.f} x.f > 0
```

and assume that the `forperm` expression were desugared into

```
forall x: Ref :: perm(x.f) > none ==> x.f > 0
```

According to the rule for evaluating quantifiers (Figure 3.17), the quantifier body would be evaluated for some arbitrary `z`, but the evaluation of `z.f > 1` would fail because Silicon does not handle disjunctive aliasing: the rule for evaluating a field read (Figure 3.10) tries to find a chunk which *definitely* provides non-zero permissions to `z.f`; but all that is known from evaluating `perm(z.f)` is that the permission sum over all receivers that are *potential* aliases of `z` is non-zero. In the example above, the left-hand side of the implication evaluates to $ite(y_1 = z, 1, 0) + ite(y_2 = z, 1, 0) > 0$, but even under this assumption, none of the chunks $f(y_1, \_, 1)$ and $f(y_2, \_, 1)$ definitely provide non-zero permission to `z.f`.

Although the desugaring of `forperm` into a quantifier is not practically usable, it is nevertheless instructive to discuss how the desugaring could look in the general case (covering fields and predicates) in order to explain the solution that Silicon actually implements. The general shape of `forperm` expressions is

```
forperm x̄ : T̄ :: {id(ē)} b
```

where the expressions $\overline{e}$ occurring in the matching pattern $\{id(\overline{e})\}$ typically (but not necessarily) mention the quantified variables $\overline{x}$. For example, `{P(0, 1)}`, `{P(x, 1)}`, `{P(x, x + 1)}` and `{P(x, y)}` are all possible matching patterns that each constrain (a subset of) the bound variables. A general desugaring of `forperm` would thus be

```
forall x̄ : T̄ :: perm(id(ē)) > none && constrain(x̄, ē) ==> b
```

where $constrain(\overline{x}, \overline{e})$ denotes the constraints coming from the matching pattern.

Silicon effectively expands this quantifier into finitely many conjuncts by instantiating the quantifier body manually (for arbitrary $\overline{z}$), once for each heap chunk $id(\overline{v}; \_, p)$, such that (1) the permission check `perm(`$id(\overline{e})$`) > none` is replaced by p `> none`, and (2) the variables $\overline{z}$ are appropriately constrained with respect to the symbolic values $\overline{v}$ obtained from the current heap chunk; this implements $constrain(\overline{z}, \overline{v})$. In comparison to the initial, hypothetical attempt of translating `forperm` into a regular quantifier, the

expansion into conjuncts no longer suffers from the problem that the evaluation of the body might fail due to disjunctive aliasing introduced by the quantification: the expansion reduces disjunctive aliasing to definite aliasing by introducing branching executions, a work-around already mentioned in Section 3.4.2.

The corresponding rule is shown in Figure 3.18. To illustrate how the rule evaluates `forperm` expressions, consider the following snippet

```
predicate P(x: Ref, k: Int) { acc(x.f) && x.f == k }

forperm x: Ref :: {P(x, 0)} unfolding acc(P(x, 0)) in x.f == 0
```

and assume that the `forperm` is evaluated in a heap with the following predicate chunks:

$$P(y_1, 0; s_1, p_1),\ P(y_2, 1; s_2, p_2),\ P(y_3, 0; s_3, p_3),\ R(y_4, 0; s_4, p_4)$$

The partially evaluated expression *cnj* constructed by the evaluation rule for `forperm` expressions will comprise three conjuncts

```
    (p1 > none && z == y1 && 0 == 0
     ==> unfolding acc(P(z, 0)) in z.f == 0)
&& (    p2 > none && z == y2 && 0 == 1
     ==> unfolding acc(P(z, 1)) in z.f == 0)
&& (    p3 > none && z == y3 && 0 == 0
     ==> unfolding acc(P(z, 0)) in z.f == 0)
```

but only for two of these (the first and the third), the nested `unfolding` expression will be evaluated.

Note that the evaluation rule for `forperm` iterates over heap chunks, not over distinct locations, which means that the body is potentially instantiated multiple times for the same arguments, for example, if the heap contains multiple chunks for the same field location. Evaluating the body once or multiple times is semantically equivalent since `forperm` is an expression, but that would no longer hold if `forperm` *were* generalised to a spatial assertion. For example, given the hypothetical snippet (with local variables `x`, `y` and `a`)

```
inhale acc(x.f, 1/2) && acc(y.f, 1/2) && x == y
inhale forperm z: Ref :: acc(a.f)
```

it would make a difference whether or not a state consolidation is triggered before the `forperm` is inhaled: if not, the second `inhale` would be equivalent to inhaling false. Such a generalisation is currently not supported in Viper, but could be supported by building on our work in the context of quantified permissions (Chapter 4).

3.5 Well-definedness and Validity of Specifications

In order to be meaningful, specifications such as method pre- and postconditions must be *well-defined*, and they must be *valid*, for example, it must be verified that a method's postcondition follows from the method's precondition and its body. Conceptually, well-definedness and validity are orthogonal concerns that can be addressed separately; however, since their handling is intertwined in Silicon (for technical reasons), we discuss both concerns in the same section.

Well-Definedness Properties

The properties that need to be checked in order to ensure that program specifications are well-defined can be separated into two groups: properties that are essentially

independent of the logic in which the specifications are expressed, and properties that are specific to the used logic.

The first group contains properties such as termination of abstraction functions, and that partial functions such as sequence indexing and numerical division are defined for their arguments. Silicon does not yet verify function termination, but we believe that it is straightforward to adapt function termination checks performed by other verifiers such as VeriFast, Dafny and Chalice. The applicability of partial functions, on the other hand, is checked as part of the corresponding evaluation rules (Section 3.4.1).

For specifications based on permission logics, it is additionally necessary to check that assertions that may be used in different contexts are *self- framing*. Recall from Section 2.2.1 that an assertion is self-framing if it requires permissions to at least those locations it reads, which ensures that the well-definedness of the assertion does not depend on the permissions held by the context in which the assertion is evaluated. In the context of Viper, the following kinds of assertions must be checked for self-framingness: predicate bodies, function preconditions (function postconditions must be framed by the precondition), method pre- and postconditions, and loop invariants.

Checking Self-Framingness and General Well-Definedness

In Smans' work on symbolic execution [119], assertions are checked for self-framingness by producing them into an empty state[3]: if an assertion accesses a heap location to which it does not require permissions, the heap access would fail because the corresponding symbolic execution rule would not be able to find a matching heap chunk. Moreover, producing an assertion into an empty state ensures that the assertion is generally well-defined (which includes being self-framing) because all remaining well-definedness properties (such as the applicability of partial functions) are checked as part of the production: when the individual sub-expressions are evaluated.

The same is in principle true for Silicon, but in order to account for inhale-exhale assertions, it is in general necessary to produce two variants of each assertion: the *inhale variant*, obtained by replacing all inhale-exhale assertions by their respective inhale component, and the analogously obtained *exhale variant*. Recall (for example, from the discussion Section 2.2.3) that an inhale-exhale assertion $[a_1, a_2]$ is inhaled as a_1, but exhaled as a_2 (the corresponding rules are shown in Figure 3.8 and Figure 3.9).

Implementation: Checking Well-Definedness and Validity

As an optimisation (which allows re-using intermediate results), Silicon combines the checking of specification well-definedness with the checking of specification validity in a single `verify` operation that *verifies* members such as methods: Figure 3.19 shows the corresponding symbolic execution rules for methods, functions and predicates; the verification of loop invariants is discussed shortly.

Due to the possible use of inhale-exhale assertions in method specifications, checking well-definedness and validity of method specifications requires (in the general case) three steps: checking well-definedness of the specification variant that is used at call site, checking well-definedness of the variant used when verifying the method, and checking that the postcondition is valid, that is, established by the method body. If the specifications do not include inhale-exhale assertions, the first step subsumes the second, which can then be skipped (this potential optimisation is not shown in Figure 3.19).

[3]The state is empty except for the background theory and potentially other relevant information; for example, when producing a postcondition the state may also contain information established by the precondition that is needed for evaluating `old` expressions.

The first step (starting on line 4) is implemented by producing the exhale-variant of the precondition into an empty heap (which, as described earlier, ensures self-framingness and general well-definedness), followed by producing the inhale-variant of the postcondition (also into an empty heap). The second step (starting on line 8) is implemented analogously, but with the opposite variant combination. After well-definedness has been established, the third step (starting on line 12) is performed: verifying the method body followed by checking the postcondition.

The second rule shown on Figure 3.19 handles function specifications analogously: initially, the exhale-variant of the precondition is produced, followed by evaluating the postcondition; this ensures well-definedness of the function with respect to call sites. Recall that the function postcondition is an expression, not an assertion: it can therefore contain neither inhale-exhale assertions (thus, only a single variant of it exists) nor accessibility predicates, and is therefore evaluated, not produced. Furthermore, function postconditions do not need to be self-framing, but they must be framed by the corresponding preconditions. In a second step, the inhale-variant of the precondition is produced, the function body is evaluated and the postcondition is checked.

```
1  verify(π₀, method meth(x̄: T̄) returns (ȳ: T̄)) =
2    x̄′ := fresh; ȳ′ := fresh
3    σ₁ := {γ := ∅[x ↦ x′][y ↦ y′], h := ∅, π := π₀}
4    produce(σ₁, exh-variant(meth_pre), fresh, (λ σ₂ ·
5      σ₃ := σ₂{h := ∅, lbh := σ₂.lbh[pre ↦ σ₂.h]}
6      produce(σ₃, inh-variant(meth_post), fresh, (λ _ ·
7        success()))))) ∧
8    produce(σ₁, inh-variant(meth_pre), fresh, (λ σ₂ ·
9      σ₃ := σ₂{lbh := σ₂.lbh[pre ↦ σ₂.h]}
10     produce(σ₃{h := ∅}, exh-variant(meth_post), fresh, (λ _ ·
11       success())) ∧
12     exec(σ₃, meth_body, (λ σ₄ ·
13       consume(σ₄, meth_post, (λ _, _ ·
14         success()))))
15
16 verify(π₀, function fun(x̄: T̄): T_r) =
17   x̄′ := fresh
18   σ₁ := {γ := ∅[x ↦ x′], h := ∅, π := π₀}
19   produce(σ₁, exh-variant(func_pre), fresh, (λ σ₂ ·
20     eval(σ₂{γ := σ₂.γ[result ↦ fresh]}, func_post, (λ _, _ ·
21       success()))))) ∧
22   produce(σ₁, inh-variant(func_pre), fresh, (λ σ₂ ·
23     eval(σ₂, func_body, (λ σ₃, body ·
24       consume(σ₃{γ := σ₃.γ[result ↦ fresh]}, func_post, (λ _, _ ·
25         success()))))))
26
27 verify(π₀, predicate pred(x̄: T̄)) =
28   x̄′ := fresh
29   σ₁ := {γ := ∅[x ↦ x′], h := ∅, π := π₀}
30   produce(σ₁, inh-variant(pred_body), fresh, (λ _ · success())) ∧
31     produce(σ₁, exh-variant(pred_body), fresh, (λ _ · success()))
32
33 inh-variant: A → A
34 inh-variant(a) = Replace each occurrence of [a₁, a₂] in a by a₁
35
36 exh-variant: A → A
37 exh-variant(a) = Replace each occurrence of [a₁, a₂] in a by a₂
```

Figure 3.19: Checking well-definedness and validity of member specifications. The rules can be simplified if the specifications do not contain inhale-exhale expressions.

Verifying a predicate body (the third rule from Figure 3.19) is straightforward: it only requires checking well-definedness since checking whether or not a predicate body (instance) is satisfied is part of the execution of `fold` statements.

In order to verify loop invariants, only a small change to the rule for executing loops from Figure 3.6 is necessary. The rule already ensures well-definedness of the inhale-variant of the loop invariant because the invariant is produced into an empty heap before the loop body is executed, and producing an assertion is equivalent to producing the assertion's inhale-variant. Hence, the necessary change is to also produce the exhale-variant into an empty state, the result of which can be discarded. For brevity, we omit the additional step from the formalisation.

3.6 Axiomatising Heap-Dependent Functions

Recall that an application $func(e)$ of a heap-dependent function is evaluated (Figure 3.16) by consuming the function's precondition, which yields the function's snapshot s, and by constructing a symbolic function application $func(e', s)$ that is the result of the evaluation.

In this section, we show how Silicon gives meaning to such symbolic function applications: by axiomatising the corresponding heap-dependent functions. This differs from the approach used in Smans' work [119], where heap-dependent functions are evaluated *on the fly*: in addition to constructing a symbolic function application $func(e, s)'$, Smans' evaluation rule evaluates the function body $b[\overline{x \mapsto e}]$ (where $\overline{x}$ are the formal arguments) to a corresponding symbolic expression $b'[\overline{x \mapsto e'}]$ and subsequently adds the equality $func(e', s) = b'[\overline{x \mapsto e'}]$ as a new path condition. To prevent recursive functions from causing infinite evaluation chains, recursive applications are evaluated to symbolic applications, but these remain "uninterpreted" because the corresponding body instance is not evaluated (similar to how Silicon prevents infinite predicate unfolding, as discussed in Section 3.4.3).

Instead of evaluating the bodies of heap-dependent functions each time such a function is applied, Silicon axiomatises heap-dependent functions once and for all, by the following two-step approach: when checking well-definedness of a function and verifying its postcondition (Figure 3.19), a mapping from accessibility predicates (in the precondition) to the resulting heap chunks, and another mapping from heap accesses (in the body) to the "accessed" heap chunks are recorded. Afterwards, and based on the recorded mappings, two axioms are created per function definition and emitted to the underlying solver: one axiom equates a symbolic function application and the respective function body, the second encodes a function's postcondition.

The major advantage of this approach over Smans' approach is that axiomatising functions simplifies the re-use of techniques that use sophisticated quantifier triggering strategies (recall Section 2.6) to improve verifier completeness with respect to reasoning about user-provided recursive functions. Several such techniques have been proposed in the context of verifiers based on verification condition generation, where it is common to axiomatise functions such that the SMT solver can only unroll the function definition a fixed number of times (typical once or twice), which avoids matching loops (infinite unrolling chains) at the expense of completeness.

An example of such a technique is work by Heule et al. [59] that uses triggers to allow solvers to unroll recursive function definitions which depend on recursive predicates, for example, a `length` function for lists, as deeply as the corresponding predicate was explored by the program (via `fold/unfold/unfolding`). More details about this technique are given in Section 3.7.3. Another example is work by Amin et al. [3], where triggers are used to allow solvers to unroll function definitions arbitrarily[4]deeply if

[4] In practice, the verifier still enforces an upper bound on the number of unrollings to achieve a good trade-off between completeness and performance, see [3] for details.

the value of the recursion variant (the ranking function) is statically bounded, for example, if it is a literal.

For a verifier based on symbolic execution that maintains a symbolic heap that is not (directly) accessible by the underlying solver, the main challenge posed by axiomatising heap-dependent functions is to make the relevant parts of the heap available to the solver without encoding the whole heap to it. This challenge can be broken down into two sub-problems: (1) identifying the partial heap that a function depends on and encoding it to the solver, and (2) relating heap-dependent expressions used in the function's definition, such as `x.f`, to the corresponding heap locations in the encoded partial heap.

However, not only have we already solved both sub-problems, but we can also piggy-back on the existing solutions in order to create the desired function axioms. The first problem, identifying and encoding the relevant partial heap, has been solved in the form of function snapshots; indeed, it already needed to be solved to achieve function framing (recall Figure 3.16 on page 82). With this, the second problem becomes relating heap-dependent expressions to the corresponding snapshot components: that is, to the corresponding heap locations as encoded by the snapshot.

To obtain a mapping from heap-dependent expressions to snapshot components, which can be used to axiomatise a function definition to the solver, we can utilise the production rules from Figure 3.8 on page 64 in combination with the evaluation rules from Figure 3.10 on page 67 and Figure 3.16 on page 82: producing an assertion, such as a function's precondition, yields heap chunks whose values are components of the function snapshot, that is, it establishes a mapping between snapshot components and heap locations; and evaluating a heap-dependent expression, for example, a field read, implicitly establishes a mapping between the expression and the accessed heap location. By composing these two mappings, the desired mapping from heap-dependent expressions to snapshot components can be obtained.

As an example, consider the following function definition:

```
function fun(x: Ref, y: Ref): Int
  requires acc(x.f) && y == x
{ y.f + 1 }
```

Producing the function precondition with an initial production snapshot s results in a symbolic heap with the chunk $f(x, \mathit{first}(s), 1)$ (and a, in this case irrelevant, path condition $\mathit{second}(s) = \mathit{unit}$). Evaluating `y.f` in this state means finding a matching chunk and returning its snapshot, which is $\mathit{first}(s)$. This gives us the required mapping from the heap access `y.f` to the function snapshot component $\mathit{first}(s)$.

Based on the previously discussed observations, Silicon implements the following algorithm for axiomatising heap-dependent functions:

(1) While establishing the validity of a function (Figure 3.19), Silicon records, in dedicated state entries, *branch-dependent* mappings from field dereferences and heap-dependent function applications to the corresponding snapshots, and from `perm` and `forperm` expressions to the corresponding symbolic expressions. When paths are joined, these mappings need to be joined as well: since they are branch-dependent, the union of the mappings can be taken. Recording and joining mappings requires straightforward changes to the corresponding rules.

(2) Afterwards, Silicon *translates* function definitions into a definitional axiom and a postcondition axiom. The translation structurally recurses over the function body, respectively, postcondition, and translates expressions as follows:

- Field accesses and heap-dependent function applications are translated to the previously-recorded snapshots; `perm` and `forperm` are translated to the recorded symbolic expressions. Since the recorded snapshots/values are branch-dependent, a single expression is potentially translated to

multiple snapshots/values which are conditional on the recorded branch conditions.

- `unfolding` expressions are translated by translating their body only.
- Accessibility predicates are omitted from the translation (they are translated to *true*).
- Occurrences of formal arguments are translated to the corresponding, previously-recorded symbolic values.
- `result` is translated to the appropriate application of the currently translated function.
- All other expressions are translated to the corresponding symbolic expressions understood by the underlying solver.

Next, we illustrate Silicon's function axiomatisation algorithm with two examples. For the sake of readability, the presented mappings and function axioms have been slightly simplified. In particular, by simplifying symbolic conditionals where the if- and the else-branch yield the same value, and by omitting snapshot components for pure conjuncts: the latter would only increase the size of snapshots by adding (here irrelevant) *unit* snapshots.

To illustrate the results of the axiomatisation algorithm, consider the following definition of a heap-dependent function:

```
function double(b: Bool, x: Ref, y: Ref, z: Ref): Int
  requires acc(x.val) && acc(y.val)
  requires b ? z == x : z == y
{
  b ? x.val + z.val : y.val + z.val
}
```

While establishing the function's validity, the following mapping from field dereferences to snapshots is recorded:

$$\texttt{x.val} \mapsto \{(\{b\}, \mathit{first}(s))\}$$
$$\texttt{y.val} \mapsto \{(\{\neg b\}, \mathit{second}(s))\}$$
$$\texttt{z.val} \mapsto \{(\{b\}, \mathit{first}(s)), (\{\neg b\}, \mathit{second}(s))\}$$

Based on this mapping, the following definitional axiom is created:

$$\forall\, b, x, y, z, s \cdot \mathit{ite}(b, z = x, z = y) \Rightarrow \\ \quad \mathit{double}(b, x, y, z, s) = \\ \quad\quad \mathit{ite}(b, \mathit{first}(s) + \mathit{ite}(b, \mathit{first}(s), \mathit{second}(s)), \mathit{second}(s) + \mathit{ite}(b, \mathit{first}(s), \mathit{second}(s)))$$

Note that the resulting symbolic expressions can be simplified by applying a few syntactic rewriting rules, if desired.

As a second illustrating example, consider the following function definition, which has a conditional assertion in its precondition (recall that conditional assertions, unlike conditional expressions, are not joined per se):

```
function zero(b: Bool, x: Ref, y: Ref, z: Ref): Int
  requires b ? acc(x.val) && z == x
             : acc(y.val) && z == y
{
  z.val - (b ? x.val : y.val)
}
```

Producing the precondition branches over the spatial conditional, and on each path, a single accessibility predicate is produced (`acc(x.val)`, respectively, `acc(y.val)`),

after which the body is evaluated. As a result, the following mapping is recorded:

$$\texttt{z.val} \mapsto \{(\{b\}, s), (\{\neg b\}, s)\}$$

and afterwards, the following definitional axiom is created:

$$\begin{array}{l}\forall\, b, x, y, z, s \cdot \\ \quad \mathit{ite}(b, z = x, z = y) \Rightarrow \\ \quad \mathit{zero}(b, x, y, z, s) = s - \mathit{ite}(b, s, s)\end{array}$$

It might be unsettling to see that all accesses to `z.val` are translated to the same snapshot value s, since `z.val` corresponds to `x.val` on one path and to `y.val` on the other. This, however, is sound because the rules for producing and consuming spatial conditionals are symmetric: consuming the function precondition at call site also branches over the spatial conditionals, and on each path, only one of the accessibility predicates is consumed, whose value is the function application snapshot on that path.

To illustrate this, consider the following client of `zero`, where the relevant parts of the symbolic state have been added as code comments:

```
method client1(b: Bool, x: Ref, y: Ref, z: Ref)
  requires acc(x.val) && acc(y.val)
  requires b ? z == x : z == y
{
    // h: val(x; v, 1), val(y; w, 1)
    // π: ite(b, z = x, z = y), ...
  var t: Int := zero(b, x, y, z)
    // γ: ..., t ↦ zero_joined(b, x, y, z)
    // π: ..., b ⇒ zero_joined(b, x, y, z) = zero(b, x, y, z, v),
    //      ¬b ⇒ zero_joined(b, x, y, z) = zero(b, x, y, z, w)
}
```

Consuming the precondition of `zero` branches over `b`, and the resulting paths are joined according to the rule for evaluating function applications from Figure 3.16.

A different, but semantically equivalent behaviour can be observed if the client is changed such that producing its precondition already branches over `b`:

```
method client2(b: Bool, x: Ref, y: Ref, z: Ref)
  requires b ? acc(x.val) && z == x : acc(y.val) && z == y
{
    // Branch one: h: val(x; v, 1),   π: b,  z = x, ...
    // Branch two: h: val(y; w, 1),   π: ¬b,  z = y, ...
  var t: Int := zero(b, x, y, z)
    // Branch one: γ: ..., t ↦ zero(b, x, y, z, v)
    // Branch two: γ: ..., t ↦ zero(b, x, y, z, w)
}
```

In contrast to `client1`, the symbolic execution now takes two paths through `client2`, because producing the client's precondition already branches over `b`. The application of `zero` is evaluated per path, but consuming the function's precondition effectively does not branch again because one potential branch is always infeasible, and therefore, the join operation performed as part of the function evaluation is essentially vacuous because there is only one path to join.

For simplicity, we omitted two aspects from the discussion of axiomatising functions that are crucial in practice but orthogonal to the work presented in this thesis: inter-function dependencies and function axiom triggering. With respect to inter-function dependencies, Silicon implements the approach described by Rudich et al. [114]: it computes a function dependency graph, that is, a call graph between functions that

includes calls made from specifications, and it processes the graph bottom up by axiomatising (and checking the validity of) a function fun_1 before a function fun_2 if the latter depends on the former, which permits using properties of fun_1 when checking the validity of fun_2.

Regarding function axiom triggering, Silicon implements the already mentioned approach by Heule et al. [59], which is in turn partially based on [83]: per axiomatised function, a *limited* and an *unlimited* function symbol are declared, and recursive invocations of a function are translated to the limited symbol. The function definitional axiom relates the unlimited function to the function body, and it is also triggered on the unlimited function. Since the body of the definitional axiom mentions only the limited version, the solver cannot unroll the axiom arbitrarily deeply, which prevents matching loops. The work of Heule et al. bases another triggering strategy on top of limited functions that, as mentioned before, ensures that the solver can unroll function definitions as deeply as the corresponding recursive data structure (if the function is defined over such a structure), such as a linked list, is explored by the program.

3.7 Evaluation

In order to assess the quality of Silicon we performed several (quantitative and qualitative) evaluations:

(1) We evaluated Silicon's *performance* (Section 3.7.1) by running Silicon on Viper's test suite (554 files in total) and comparing its verification times to those of Viper's second, verification-condition-generation-based verifier. The results demonstrate that Silicon performs well in general (with a mean of 0.29 seconds per file) and that it is in average more than seven times faster than Viper's second verifier.

(2) We evaluated the *stability* of Silicon's performance (Section 3.7.1) by manually seeding verification failures in the 20 longest (in terms of lines of code) and the 20 longest-running tests (27 files in total), and compared Silicon's performance on the original and the seeded versions; the results show that Silicon performs equally well in both cases.

(3) We evaluated the *completeness* of Silicon in two ways (Section 3.7.2): (1) by manually inspecting the 27 longest (as previously defined) files and counting the number of assertions that exist solely to overcome an incompleteness of the verifier: in average, one assertion per 1282 lines; and (2) by creating a test suite dedicated to uncovering heap-related incompletenesses and encoding it to the two closest-related existing verifiers (VeriFast [67] and VeriCool [120]): the results show that Silicon exhibits fewer such incompletenesses.

(4) We performed a detailed *comparison with existing verifiers* (VeriFast and VeriCool; Section 3.7.3), including a discussion of alternative solutions to common problems such as reasoning about mathematical data types (such as sequences) and handling recursive definitions.

Further evaluations are presented in the remaining chapters: an evaluation of Silicon's support for quantified permissions and magic wands in Section 4.5 and Section 5.6, respectively, and a concluding evaluation is shown in Chapter 6, where we evaluate Silicon with respect to the problem statement from Section 1.1.

3.7.1 Performance

To evaluate the performance of Silicon, we compare Silicon against Viper's second verifier (called Carbon), which is based on verification condition generation (see

Input programs	Number of programs	Average size (LOC)	Mean time (s) Si	Mean time (s) C	Max. time (s) Si	Max. time (s) C
Chalice	246	121.94	0.29	2.53	16.00	19.45
Viper tests	261	34.09	0.27	1.94	18.72	24.36
VerCors	47	152.09	0.98	4.79	15.42	23.49

Figure 3.20: Performance evaluation of Silicon (Si) in comparison to Carbon (C). Lines of code (LOC) measurements do not include whitespace lines and comments. All input programs were run 10 times and average times recorded. The mean and maximum times were calculated based on these averages (standard deviations were always negligible). Timings do not include JVM start-up time: we persist a JVM across test runs using the Nailgun tool [88] (for Carbon, timings include start-up of Boogie [6]). All timings were gathered on a Lenovo Thinkpad W540 running Windows 10 x64, with 16GB RAM.

also Figure 2.1). The performance is measured on the following collections of input programs: our own Viper test suite, Viper programs generated by VerCors [16], and programs generated from Chalice examples via Viper's Chalice front end. The set of VerCors examples was provided to us by the VerCors developers as representative of their Viper usage. The input programs do not use quantified permissions or magic wands, the corresponding chapters (Chapter 4 and Chapter 5, respectively) include separate evaluation sections.

The results are shown in Figure 3.20. Both verifiers perform consistently well in the average case, but Silicon is significantly faster. As the average times suggest, the maximum times are true outliers — these were typically examples designed to be complex, in order to test what the tools could handle. The Viper tests (which are mostly regression tests) tend to be shorter and less challenging than the VerCors-generated programs, which are representative of real usage of Viper as a back-end infrastructure.

Since SMT encodings sometimes exhibit worse performance for *failed* verification attempts, we also collected experimental data in order to determine the effect of failures on the verification time. For this purpose, we took the 20 longest (in terms of lines of code) and the 20 longest-running (in terms of verification time) examples from the programs used in the previous evaluation (27 programs with a total of 11,675 lines of code) and ran Silicon on five versions of each of these programs in which errors were seeded (135 programs in total). The results, reported in Figure 3.21, show that the performance of Silicon is stable: in nearly all cases, the difference in verification time between the original and the seeded version is marginal (and the few outliers nevertheless verify in less than two seconds). Column "Mean original" shows the times for the original versions (in seconds), column "Max. seeded" shows the maximum across all five seeded versions. The differences in verification times are shown in column "Differences", in seconds and as the percentage difference. Note that half of the original programs (13 out of 27) already contained verification failures (typically a "faulty" client that, for example, intentionally violates a callee's precondition); these were preserved in the seeded versions. Programs that already contained verification failures are marked with an asterisk in column "Program". All original programs are included in our test suite, their file names can be retrieved from Figure B.1 in Appendix B.

3.7.2 Completeness

When verifying programs, the verification sometimes fails although all the necessary information is available, which constitutes an incompleteness of the verifier (but workarounds are usually possible, such as adding additional assertions or lemma methods). In the context of Viper and Silicon, incompletenesses can be classified into one of the following categories:

Program	Mean original (s)	Max. seeded (s)	Difference (s)	Difference (%)
1	22.52	23.49	+0.96	+04.27
2	19.56	19.93	+0.37	+01.90
3*	17.73	18.23	+0.50	+02.84
4*	17.53	17.91	+0.38	+02.17
5	16.77	16.89	+0.12	+00.71
6	4.47	4.46	−0.01	−00.26
7*	2.83	2.84	+0.01	+00.33
8	2.27	2.18	−0.09	−03.89
9	2.23	2.27	+0.04	+01.75
10*	1.86	1.92	+0.06	+03.23
11	1.69	1.93	+0.25	+14.59
12	1.37	1.39	+0.02	+01.48
13	1.31	1.20	−0.11	−08.56
14*	1.20	1.18	−0.02	−01.47
15*	1.16	1.22	+0.06	+05.11
16	1.13	1.18	+0.05	+04.53
17*	1.12	1.19	+0.06	+05.58
18	1.02	0.83	−0.19	−19.06
19	1.00	1.30	+0.30	+30.08
20	0.81	0.97	+0.15	+18.81
21*	0.79	0.85	+0.06	+07.32
22*	0.79	0.80	+0.00	+00.57
23*	0.77	0.77	+0.00	+00.20
24*	0.77	0.81	+0.04	+05.57
25	0.71	0.69	−0.02	−03.14
26*	0.47	0.48	+0.02	+03.46
27*	0.38	0.38	+0.00	+00.97

Figure 3.21: Evaluating the stability of Silicon's performance with respect to seeded verification failures; the data was collected as in the previous experiment (Figure 3.20; on the same hardware, averaged over ten runs).

Background theories In our experience, most exhibited incompletenesses are due to incomplete axiomatisations of background theories such as sets, sequences and multisets (see Listing B.2 in Appendix B for an example).

Proof principles Verification might fail because a mathematical proof principle is not supported. In particular, Silicon does not have built-in support for inductive reasoning, and since SMT solvers typically do not perform induction either, proofs that require an inductive argument usually fail (see Listing B.3 in Appendix B for an example).

Implementation Incompletenesses can also arise from the implementation of Silicon, such as the heap-related incompletenesses discussed in Section 3.4.2.

SMT solver In addition, some incompletenesses exhibited by Silicon originate from incompletenesses in the underlying SMT solvers, for example, when reasoning about undecidable, built-in theories such as non-linear integer arithmetic.

Language design Finally, there are situations in which the verification fails due to decisions made when designing the Viper language and defining its semantics: for example, in order to prevent non-terminating SMT solver runs, the unrolling depth of recursive functions is limited and triggers are used to control quantifier instantiations (see Listing B.4 in Appendix B for an example). It is debatable whether or not such verification failures should be considered as incompletenesses.

In order to evaluate the completeness of Silicon, we performed two experiments: (1) we determined the number of intermediate assertions required by a set of examples, and (2) we created a number of unit-test style test cases that challenge Silicon's heap management algorithms, encoded them for two other verifiers and compared the exhibited incompletenesses.

For the first experiment, we manually inspected the 27 files used in the previous evaluation (Figure 3.21) and removed all intermediate assertions that were not required for the verification to succeed (such assertions were typically left-overs from previous attempts to debug a now-working proof). We consider an assertion — typically an `assert`, sometimes a pure `exhale` — as *intermediate* if it is not the final proof obligation on a program path: any `assert` that is followed by another statement, or a postcondition or loop invariant check, is considered as intermediate. Note that this is a coarse but conservative over-approximation because it also considers assertions as intermediate that encode proof obligations of the front end that are potentially unrelated to the final proof obligation, for example, particular well-definedness requirements. We also looked for other incompleteness-related workarounds, for example, manually introduced case splits or postconditions that encode an inductive argument, but found only one such workaround. In total, four out of 27 files required at least one additional annotation to overcome an incompleteness:

- Program 1 required three additional assertions (across two different methods): each assertion temporarily unfolds a predicate instance (via `unfolding`) to make facts from the predicate body available that are needed for proving subsequent assertions (the assertions are needed by both Viper verifiers). Note that it is debatable whether or not these assertions should be considered as *intermediate* since Viper verifiers are not meant to reason about predicate definitions without explicit direction (via (un)fold). However, the three `assert` statements match our (over-approximating) definition of intermediate assertions and we therefore counted them as such.
- Program 2 is a variant of program 1 in which (among other changes) a postcondition has been strengthened. This version requires four additional assertions: the three from above plus an additional `unfolding` (for the same reason).
- Program 3 required (for both Viper verifiers) one additional assertion to prove a sequence-related property.
- Program 8 required (for both Viper verifiers) one additional function postcondition that encodes an inductive argument.

The results show that Silicon performs well with respect to incompletenesses: only four out of the 27 programs require additional annotations; for the inspected programs, this amounts to an average of one additional annotations per 1282 lines of code.

For the second experiment, we used our experience gained from developing Silicon's state consolidation algorithms, and from working with Silicon in general, to craft a number of challenging test cases with the goal of uncovering heap-related incompletenesses. We then encoded these test cases for the two symbolic-execution-based verifiers that are closest related to Silicon (VeriFast and VeriCool) and compared the observed incompletenesses: the results are favourable for Silicon, which exhibits fewer heap-related incompletenesses. The comparison is described in detail in the next section, as part of a thorough, more general comparison between Silicon and the two other verifiers.

3.7.3 Comparison with Related Verifiers

As discussed in Section 1.2, various automated verification tools for permission logics exist, including symbolic-execution-based verifiers such as Smallfoot [11], jStar [42], SpecCheck [119] and VeriFast [67], but also verifiers that are not based on symbolic

Feature	VeriCool	VeriFast	Silicon
Heap-dependent expressions	✓	✗	✓
Mathematical collections (sets, ...)	✓	✗	✓
Fractional permissions	✓	✗	✓
Definite aliasing	✓*	(✓)	(✓)
Disjunctive aliasing	✗	✗	✗
Retrospective aliasing	–	✗	✗
Path conditions from heap constraints	(✓)	(✓)	(✓)
Quantifiers	✗	✓	✓
Permission introspection	✗	✗*	✓
Object allocation	✗	✗	(✓)
Function unrolling depth	✗	–	✓
Predicate unfolding depth	✗*	–	✓

Table 3.1: Comparing VeriCool, VeriFast and Silicon on a selected set of features (in a rather broad sense). Asterisked entries indicate that the comparison is subject to restrictions, parentheses indicate that support for a feature is provided but incomplete, and a dash indicates that a comparison cannot be made for conceptual reasons; in each case, further details are given in the text. The second feature block (starting with "definite aliasing") refers to the sources of incompletenesses discussed in the context of state consolidations (Section 3.4.2). "Object allocation" compares to which extent verifiers encode the fact that a newly allocated object is indeed different from all existing objects (Viper's `new` statement Section 3.2). "Function unfolding depth" refers to the need for preventing infinite recursive function unrollings, and how the maximum recursion depth is chosen (Section 3.6). Similarly, "Predicate unfolding depth" refers to the need for preventing infinite recursive `unfolding` evaluations.

execution, such as HIP/SLEEK [35] and GRASShopper [103]. Of these verifiers, SpecCheck and VeriFast are closest related to Silicon (see Section 1.2 for an overview and discussion of the other verifiers): they require fully-specified programs, they support custom predicate definitions, they require explicit annotations to exchange predicate instances for their bodies (and vice versa), and both strictly separate the symbolic state into a verified-managed heap and solver-managed path conditions. As far as we know, VeriFast is the only one of these four verifiers that is still under active development.

SpecCheck was the first verifier for implicit dynamic frames, and its symbolic execution rules and their presentation had a strong influence on the work presented in this thesis. To our knowledge, SpecCheck has been superseded by VeriCool [120], a verifier for implicit dynamic frames that optionally uses either verification condition generation or symbolic execution. Publications [122, 118, 120] only describe VeriCool's verification condition generation back-end, but as far as we know, its symbolic execution back-end is essentially an improved version of SpecCheck. We will therefore compare Silicon to VeriCool's symbolic execution back-end (version 3.4), and not to SpecCheck (both SpecCheck and VeriCool are no longer under active development). The specification features supported by VeriCool are essentially all supported by Viper as well, which simplifies the comparison of the tools.

VeriFast is a mature verifier for separation logic, which has been successfully used in various research experiments and in industrial case studies. The specification language supported by VeriFast differs more strongly from Viper's: for example, VeriFast supports higher-order methods and predicates, but does not support heap-dependent expressions such as Viper's abstraction functions. The comparison therefore focuses on the core language features, and on design choices that are characteristic for the symbolic execution engines. An in-depth comparison of the specification languages and an analysis of how the differences in language design affect how examples are encoded and verified would be interesting and insightful, but is outside the scope of this thesis.

Table 3.1 lists features of Silicon, and indicates to which extent these are supported by VeriCool and VeriFast. Regarding the comparison of potential incompletenesses arising from how the verifiers handle the heap in the presence of different kinds of aliasing (the second block in the table), a word of caution is in order. We are not aware of any publications that describe the heap management of VeriCool or VeriFast, and we therefore did a best-effort testing of the verifiers: we used our experience gained from working with Silicon, and our knowledge about Silicon's state consolidation algorithms, to craft challenging tests cases which we encoded for all three verifiers (if possible). The tests can be found in Appendix B.1.

Heap-dependent expressions Heap-dependent expressions, and in particular heap-dependent abstraction functions, are a versatile specification feature (as argued in Chapter 2) and commonly used in verification: traditionally in approaches not based on a permission logic, for example, ownership or dynamic frames (see also Section 1.2), but increasingly also in permission-logic-based approaches, for example, verifiers based on implicit dynamic frames and by GRASShopper [103]. In addition to being an important feature of an intermediate verification language, heap-dependent expressions also pose interesting challenges for symbolic execution, and are therefore relevant for this comparison.

Mathematical collections VeriFast and Silicon both support mathematical collections such as sequences and sets, which are necessary to specify full-functional properties. The individual support for such structures differs significantly between Silicon and VeriFast, however: Silicon provides built-in support for sequences, sets and multisets, and operations to manipulate them, by including axiomatisations (taken from Dafny) of the mathematical structures in the verifier's background theory. In practice, the axiomatisation achieves a decent trade-off between completeness and performance, but it is (inherently) incomplete, and certain usage patterns are known to cause performance degradation. Support for additional mathematical structures can be added to Viper by declaring corresponding domains, as discussed in Section 2.5.

VeriFast does not include a similar axiomatisation in its background theory; instead, it supports defining custom inductive data types such as Haskell-style nil/cons lists, and (heap-independent and side-effect free) recursive functions to manipulate them, for example, list indexing and list append. Relevant properties, for example, of indexing over append, are expressed as *lemma* functions: abstract functions whose postcondition is the desired property. Lemma functions need to be invoked explicitly (via ghost code), which reduces the degree of automation VeriFast achieves, but in general improves performance since it reduces the workload of the underlying solver. To increase automation, lemmas can be marked as *auto*: heap-independent auto lemmas are encoded as axioms to the solver (triggers can be provided in order to control axiom instantiations), whereas heap-dependent lemmas are automatically instantiated by VeriFast (according to heuristics). VeriFast's standard library includes a definition of lists, list-manipulating functions and lemma functions for various properties thereof, as well as functions (and lemmas) for modelling sets on top of lists.

Fractional permissions VeriFast and Silicon support fractional permissions, which are essential for encoding shared read access, whereas VeriCool does not. The absence of fractional permissions significantly simplifies the management of the symbolic heap because the need for merging chunks does not arise, and without fractional permissions, it is impossible to express retrospective aliasing, which eliminates a whole class of incompletenesses.

Definite aliasing We were not able to observe incompletenesses due to the verifiers' heap management if only definite aliasing is present and if no fractional permissions are used. This is not unexpected, since it is straightforward to manage the heap in a complete way (assuming a perfect solver) in such a restricted setting.

However, both VeriFast and Silicon are incomplete in the presence of fractional permissions, as illustrated by the test cases shown in Appendix B.1.1. The tests also give evidence that Silicon's handling of the heap is slightly more complete than VeriFast's, which is due to Silicon's try operation discussed in Section 3.4.2. Although small, the difference is potentially important in practice because heap-dependent expressions increase the potential number of program points at which incompletenesses can be observed.

Disjunctive aliasing All three verifiers are equally incomplete in the presence of disjunctive aliasing, but in each of the verifiers, disjunctive aliasing can be reduced to definite aliasing by forcing the symbolic execution engine to branch appropriately. See Appendix B.1.2 for the corresponding test cases.

Retrospective aliasing Retrospective aliasing cannot be expressed in VeriCool due to the missing support for fractional permissions. VeriFast and Silicon are equally incomplete in the presence of retrospective aliasing, but work-arounds are possible. See Appendix B.1.3 for the corresponding test cases.

In Silicon, at least two work-arounds are possible (illustrated in Appendix B.1.3). The first (`test02a` in Listing B.12) uses labelled `old` expressions to trigger a retrospective state consolidation, whereas the second (`test02b`) uses an `if` statement to force the verifier to eagerly branch over the future aliasing constraint. The latter work-around is also possible in VeriFast.

Both work-arounds include inserting additional statements in a way that requires foreseeing the potential incompleteness to a degree that is probably hard to achieve during an automated encoding, which is likely to render the work-arounds impractical for an intermediate verification language such as Viper. For VeriFast, this is potentially less of a problem because it is not intended to serve as an intermediate verification language.

So far, we did not encounter retrospective aliasing in practice, but it nevertheless is an interesting kind of aliasing since it is challenging for verifiers based on symbolic execution but does not pose any problems for verifiers based on verification condition generation.

Path conditions from heap constraints Due to the separation of the state into a symbolic heap and a set of path conditions, various constraints that are implicitly present in the heap need to be added as path conditions in order to make them available to the solver (as discussed in Section 3.4.2). All three verifiers are incomplete in that respect, as illustrated by the test cases shown in Appendix B.1.4. As before, some tests cannot be encoded in VeriCool due to its lack of support for fractional permissions.

The tests also demonstrate that Silicon exhibits fewer incompletenesses; in particular, that it is more stable with respect to changes in the specifications. VeriCool, for example, infers receiver disjointness from permissions obtained in the precondition, but not if the permissions are obtained from a called method's postcondition (tests `test01` vs. `test01a` from Listing B.13 in Appendix B.1.4). VeriFast does not exhibit this incompleteness, but it fails to infer receiver disjointness if the relevant permissions are distributed over multiple chunks (tests `test01` vs. `test02` from Listing B.14 in Appendix B.1.4).

Quantifiers First-order-logic quantifiers (recall Section 3.4.3) are encoded by Silicon as a quantified formula to the underlying solver. VeriFast supports such quantifiers as well (and encodes them analogously), but in practice, it seems much more common to use variants of recursively defined, higher-order `forall` (respectively, `exists`) functions that test if a given boolean function holds for each (respectively, at least one) element of a collection, such as a list. We conjecture that this is a consequence of VeriFast's design decision to support mathematical structures via inductive data types, recursive functions and explicitly invoked

lemma functions, which integrates more naturally with recursive `forall` functions than with quantifiers.

Analogous to the handling of mathematical collections, Viper favours automation at the price of potential performance degradation (if the quantifier triggers are too permissive), whereas VeriFast favours performance at the price of additional ghost code. The choice of trade-offs are a consequence of the design philosophy behind the languages and tools, and reflect their intended use: requiring ghost code (in ways that is challenging to anticipate automatically) can significantly reduce the practical use of an intermediate verification language, but is not a key concern for a verifier that is typically used to verify manually encoded examples.

Permission introspection Viper supports permission introspection via the `perm` and `forperm` expressions discussed in Section 3.4.4, which can be used to make assertions conditional on the availability of permissions or to encode leak checks, as discussed in Section 2.2.3.

VeriFast does not support a construct similar to `forperm`, but its C front end has the special case of permission leak-checks built in because C requires manual memory management; explicitly choosing to leak permissions is possible via ghost statements. The Java front end does not have leak checks built in. Instead of a `perm` expression, VeriFast supports asserting the existence of a points-to predicate with an existentially quantified permission amount, which introduces a new variable whose value is the permission amount of the asserted points-to predicate. The newly introduced variable can then be used to, for example, branch over the availability of permissions. However, since asserting points-to predicates is subject to the previously discussed heap-related incompletenesses, the obtained permission amount is potentially an under-approximation of the actually available permission amount, which can result in incompletenesses and unsoundnesses, as illustrated by the test cases shown in Appendix B.1.5.

Object allocation As illustrated by the test cases in Appendix B.1.6, VeriCool does not appear to infer any reference disequalities from allocating new objects. VeriFast appears to not infer any reference disequalities from allocating new objects either, but it infers disequalities from the available permission amounts (as already illustrated by the examples in Appendix B.1.4). Silicon is more complete: it infers disequalities for all references directly reachable from the current symbolic state, i.e. local variables and fields, but neither for references hidden in predicates nor for references returned by functions (that only read previously allocated state).

Function unrolling depth VeriCool and Silicon both support heap-dependent recursive functions, which are routinely used to inspect recursively defined data structures such as linked lists. An example is given in Appendix B.1.7, in which a recursively defined function computes the length of a linked list.

As discussed in Section 3.6, Silicon axiomatises such function definitions to the solver, whereas VeriCool (symbolically) evaluates the function body each time a function is applied. Both approaches require taking measures to prevent recursive functions from causing non-terminating verification runs: on each application, VeriCool evaluates recursive functions up to a fixed depth (one), and Silicon analogously allows the solver to instantiate the function definition axiom once per application. Silicon complements this static approach by the more dynamic approach from Heule et al. [59] that permits the solver to unroll function definition axioms as deeply as the corresponding data structures were inspected in the program (via `fold/unfold(ing)`). The example shown in Appendix B.1.7 illustrates that Silicon's approach to handling heap-dependent functions is more complete than VeriCool's, which, for example, fails to assert that a list obtained from concatenating two nodes is of length two.

For comparison, Appendix B.1.7 also includes an encoding of a linked list in VeriFast. Instead of heap-dependent recursive functions, an additional argument of the list predicate is used to represent the length of the list, and the body of the length function is essentially inlined in the predicate, with existentially bound variables replacing recursive function invocations. This makes the predicate definition more involved: refining the linked list further, for example, by requiring that all stored elements satisfy a certain property, requires adding additional arguments to the predicate and assertions to its body, as discussed in Section 2.4.

Due to this encoding, indefinite function unrolling is not an issue for VeriFast: as in Viper, predicates need to be (un)folded explicitly (via `open`/`close` statements) and can therefore not cause non-termination. However, the exclusive use of predicates can incur the need for "no-op" `open`/`close` pairs that are necessary in order to learn a fact from a predicate body, which might not be necessary in an encoding that uses a corresponding getter function (because functions are unrolled when applied). For an illustration, see `test03` in Appendix B.1.7.

Predicate unfolding depth Viper's `unfolding` expressions are analogous to VeriCool's `opening` expressions. As described in Section 3.4.3, explicit measures need to be taken to prevent the symbolic execution from indefinitely recursing while `unfolding` a predicate whose body (transitively) contains an `unfolding` of another instance of the same predicate. Silicon prevents infinite recursion by replacing recursive `unfolding` expressions with unknown values, as discussed in Section 3.4.3. VeriCool, however, does not appear to take similar measures, and crashes with a stack overflow. Corresponding test cases can be found in Appendix B.1.8.

3.8 Limitations

In addition to the heap-related incompletenesses that are typical for verifiers based on Smallfoot-style symbolic execution (as discussed in Section 3.7), Silicon has a few other known limitations where its behaviour differs from the expected behaviour (defining a formal semantics for Viper is active work in progress).

Constraining Blocks

Recall from Section 2.7 that Viper's `constraining` blocks, which implement work by Heule et al. [58] on abstract read permissions, can be used to avoid the need of having to explicitly choose permission fractions in situations where any arbitrarily small (but positive) fraction will suffice because the corresponding locations only need to be read.

For example, the `constraining` block in the snippet

```
inhale acc(x.f)

constraining(p) {
  exhale acc(x.f, p) && acc(x.f, p)
}
```

is conceptually translated by assuming, per accessibility predicate, that the abstract permission amount to exhale (here `p`) is smaller than the still held permission amount:

```
inhale p < write
exhale acc(x.f, p)
inhale p < write - p
exhale acc(x.f, p)
```

A naïve implementation of this idea, however, is prone to incompletenesses

```
constraining(p) {
  exhale acc(x.f, p) && acc(x.f, 1/2)
}
// --- Incomplete if naively translated as: ---
inhale p < write     // Permits 1/2 < p
exhale acc(x.f, p)
exhale acc(x.f, 1/2) // Could fail
```

and unsoundnesses

```
constraining(p) {
  exhale acc(x.f, write - p) && acc(x.f, p)
}
// --- Unsound if naively translated as: ---
inhale p < write - p
exhale acc(x.f, write - p)
inhale p < write - (write - p) // Contradiction
exhale acc(x.f, p)
```

Heule et al. use a *three-phase exhaling* to prevent such incompletenesses and unsoundnesses, but Silicon does not yet implement the approach and therefore exhibits both issues. Implementing the three-phase exhaling is expected to be an engineering task only.

Inhale-Exhale Assertions and Snapshots

Recall from Section 3.3 that inhale-exhale assertions $[a_1, a_2]$ affect the structure of assertions because of the asymmetry between how inhale-exhale assertions are inhaled (as a_1) and exhaled (as a_2). Furthermore, recall that snapshots are used to represent the values of the partial heap an assertion describes, and that this representation requires the structure of the snapshot to match the structure of the assertion. For example, folding a predicate instance yields a snapshot (by consuming the instance's body) describing the values of the partial heap folded into the instance, and upon a subsequent unfolding of the instance, the snapshot is used to "restore" the values of this partial heap (when producing the body).

The structural representation of heap values via snapshots complicates the use of inhale-exhale assertions (spatial assertions, not pure expressions) in positions in which the same snapshot is potentially used with the consumption and production of assertions: in Silicon, such positions are predicate bodies and function preconditions. Silicon currently disallows inhale-exhale assertions in such positions; allowing them without taking further measures would be unsound, as illustrated by the following example:

```
predicate pair(x: Ref) {
  [acc(x.f) && acc(x.g),
   acc(x.g) && acc(x.f)]
}

  // h: f(x;22,1), g(x;3,1)
assert x.f == 22 && x.g == 3
fold acc(pair(x))
  // h: pair(x;(3,22),1)
unfold acc(pair(x))
  // h: f(x;3,1), g(x;22,1)
assert x.f == 3 && x.g == 22 // Would incorrectly succeed
```

Lifting this restriction is not straightforward, and probably requires a more semantic (and less structural) representation of snapshots, for example, as maps from locations to values, which is likely to be more expensive (in terms of solver involvement) than the current structural representation. However, a similar problem already needed to be solved in in the context of quantified permissions; the solution performs well and can probably be adapted in order to lift the restriction on the use of inhale-exhale assertions.

Inhale-Exhale Assertions and Well-Definedness Checks

Recall from the discussion in Section 3.5 that inhale-exhale assertions also complicate the well-definedness checks of, for example, method specifications: since the specifications used at call site differ from those used when verifying the method body, two different specification variants (combinations of pre- and postconditions) need to be checked for well-definedness.

Silicon currently only checks the callee-variant of specifications that use inhale-exhale assertions for well-definedness (using a_1 for every such assertion), and thus does not report malformed specifications such as the following:

```
method bad(x: Ref)
  requires [true, x.f == 0] // Not reported as non-self-framing
{}
```

Implementing the necessary checks (discussed in Section 3.5) is expected to be straightforward.

General Shape of `forperm`

Viper does not yet support the general shape of `forperm` expressions that is discussed in Section 3.4.4, and consequently, neither does Silicon. `forperm` expressions are currently limited to fields, the supported syntax is `forperm[`f`] `x` :: `e, where f is a field and x is fixed to type `Ref`.

Chapter 4

Quantified Permissions

Specifying general unbounded heap structures using recursive predicates — which are typically the only means for specifying such structures that existing automated verifiers for permission logics provide — can become cumbersome if the pattern according to which a structure is accessed does not follow the corresponding predicate structure (for example, traversing a doubly-linked list from the end) or in general follows no specific order, such as traversing an array or a graph (as was already argued in Section 2.3). In such cases, programmers are typically required to provide substantial manual proof steps (for example, as ghost code) to bridge the mismatch between the program's access pattern and the imposed predicate structure.

Iterated separating conjunction [108] is an alternative way to denote permissions to a set of heap locations, which has, for example, been used in by-hand proofs to specify arrays [108], cyclic data structures [14, 136], the objects stored in linked lists [42], and graph algorithms [136]. Unlike recursive predicates, an iterated separating conjunction does not prescribe any particular traversal order.

Despite its usefulness and inclusion in early presentations of separation logic, no (already) existing program verifier supports general iterated separating conjunction directly. Among the tools based on symbolic execution (recall Section 1.2 and Section 3.7.3), Smallfoot does not support iterated separating conjunction, neither does VeriCool's symbolic execution back-end, and VeriFast and jStar allow programmers to encode only some forms of iterated separating conjunction via abstract predicates that can be manipulated by auxiliary operations and lemmas (in VeriFast) or tailored rewrite rules (in jStar). For arrays, this encoding is partially supported by libraries. However, in the general case, programmers need to provide the extra machinery, which significantly increases the necessary manual effort.

Among the verifiers based on verification condition generation, Chalice [84] supports only a restricted form of iterated separating conjunction (ranging over all objects stored in a sequence), and VeriCool uses an encoding that leads to unreliable behaviour of the SMT solver [120, p. 46]. The GRASShopper tool [103] does not provide built-in or general support for iterated separating conjunction, but some ingredients of the technique presented here (particularly, the technical usage of inverse functions) have been employed there in specifications involving arrays. The Dafny verifier [82] can be used to write similar set- and quantifier-based specifications, but it neither supports permission-based reasoning nor concurrency (nor expressive language constructs such as `in-/exhale` that facilitate the encoding thereof).

This chapter presents the first symbolic execution technique that directly supports general forms of iterated separating conjunction. The technique is compatible with other features of permission logics: it supports fractional permissions, such that a heap location may be ranged over by several iterated separating conjunctions, and it allows iterated separating conjunction to occur in predicate bodies and in the preconditions of abstraction functions. This combination of features allows one to specify and verify challenging examples such as graph-marking algorithms (see Appendix C) that so far were beyond the scope of automated verifiers based on permission logics.

Chapter Overview

Viper supports iterated separating conjunction in the form of quantified permission assertions, and we use the term *quantified permissions*, as in previous chapters, to refer to Viper's language feature that corresponds to the separation logic connective *iterated separating conjunction*.

The work described in this chapter has in parts been published at CAV 2016, in the paper *Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution* by Müller, Schwerhoff and Summers [95]. At the verification competition VerifyThis@ETAPS'16 [62], quantified permissions was awarded the *Distinguished User-Assistance Tool Feature* price for the feature that proved particularly useful during the competition.

The main contributions of this chapter are:

(1) A language feature, called *quantified permissions*, that enables the direct use of general iterated separating conjunctions in a permission-based verification language.

(2) A novel representation of the partial heaps denoted by an iterated separating conjunction, along with algorithms to manipulate this representation.

(3) A technique to preserve across heap changes the values of expressions that depend on unbounded sets of heap locations as denoted by iterated separating conjunctions.

(4) An integration of iterated separating conjunction with all other important Viper features (in addition to heap-dependent functions), in particular, with predicates, magic wands and permission introspection.

(5) An SMT encoding of the necessary axioms and proof obligations that carefully controls quantifier instantiations.

(6) An implementation of (most of) the presented technique in Silicon.

The remainder of this chapter is structured as follows: the main technical challenges addressed by our work are explained in Section 4.1, and illustrated with a simple motivating example. A design for a symbolic heap that can represent permissions described by iterated separating conjunctions is presented in Section 4.2, followed by an explanation of the symbolic evaluation of expressions and framing with respect to this heap representation in Section 4.3. Afterwards, we discuss the controlling of quantifier instantiations in Section 4.4. An evaluation of the presented technique, as implemented in Silicon, is presented in Section 4.5, and Section 4.6 gives additional details of the implementation.

4.1 Motivation and Technical Challenges

Listing 4.1 introduces the running example of this chapter, which illustrates the use of quantified permissions (in a pseudo-code that resembles Viper, but natively supports arrays and parallel composition): method `replace` replaces all occurrences of integer `from` with integer `to` in the segment of array `a` between `left` and `right`. The recursive calls to smaller array segments are performed concurrently using parallel composition $\|$. The second precondition requires access permissions to all elements in the array segment, and the first postcondition returns these permissions to the caller; both are expressed using quantified permissions. Arrays are encoded in Viper as discussed in Section 2.5.1: by declaring a custom `Array` domain, an injective `Ref`-typed function `loc`, and a field `val` (of appropriate type), such that `loc(a, i).val` models the array slot `a[i]`.

```
1  method replace(a: Int[], left: Int, right: Int,
2                 from: Int, to: Int)
3    requires 0 <= left < right <= a.length
4    requires forall i: Int :: left <= i < right ==> acc(a[i])
5    ensures  forall i: Int :: left <= i < right ==> acc(a[i])
6    ensures  forall i: Int :: left <= i < right ==>
7                 (old(a[i]) == from
8                    ? a[i] == to
9                    : a[i] == old(a[i]))
10 {
11   if (right - left <= 1) {
12     if(a[left] == from) { a[left] := to }
13   } else {
14     var mid := left + (right - left) / 2
15
16     replace(a, left, mid, from, to)
17          ||
18     replace(a, mid, right, from, to)
19   }
20 }
```

Listing 4.1: A parallel replace operation on array segments. The second precondition and the first postcondition employ quantified permissions to specify permissions to the elements of the array. Arrays are encoded in Viper as discussed in Section 2.5.1.

The running example illustrates several technical challenges that must be addressed in order to support quantified permissions in an automated verifier. These challenges are discussed next, and for each challenge, we briefly recapitulate how the corresponding challenge has been solved for the simpler case of *non-quantified* permissions (as discussed in Chapter 3) and we describe how *quantified* permissions complicate the challenge.

(1) Permissions and heap values must be *represented* in the verification state, for example, to represent the array locations to which permissions are obtained from the precondition on line 4 of Listing 4.1. In the non-quantified setting, permissions and heap values are represented as heap chunks that each map a single field location to a permission and a location value (and similar for predicate instances), and constraints on these values are recorded in the path conditions. In order to support quantified permissions, heap chunks must be generalised to denote permissions to an unbounded number of locations, and encode a symbolic value per location.

(2) It must be possible to *remove* permissions, for example, on line 17 for the parallel calls or when exhaling the postcondition on line 5. Without quantified permissions, Silicon in general[1] under-approximates the permissions a symbolic heap provides by only taking permissions into account that are definitely available (for a specific heap location). Such an approximation is in general incomplete (but sufficiently complete in practice, as shown in Section 3.7), but very efficient. With the addition of quantified permissions, such an under-approximation is no longer an option, however: when exhaling the postcondition on line 5, permissions must be taken from both available generalised heap chunks (one chunk per recursive call), but for an arbitrary `a[i]`, no individual heap chunk definitely provides the required permission. Moreover, removing quantified permissions from a generalised chunk may affect only some of the locations to which it provides permissions: for example, when exhaling the precondition of the first recursive call, the permissions required for the second call must be retained in the symbolic state.

[1]The exception are Viper's permission introspection features (Section 3.4.4) which, for soundness, require a precise permission representation.

(3) Similarly, it must be possible to *check* for permissions in a state, for example, when accessing `a[left]` on line 12, and supporting quantified permissions complicates this challenge analogously to how quantified permissions affect removing permissions: permissions may be spread across multiple heap chunks, and in the presence of fractional permissions, these chunks may even partially overlap (that is, partially provide fractions to the same locations).

(4) Previously, it also sufficed to consider each heap chunk in isolation when *evaluating* heap-dependent expressions. With the addition of quantified permissions, however, heap-dependent expressions under quantifiers may rely on symbolic values from multiple heap chunks: for example, proving the postcondition on line 6 requires information from both recursive calls.

(5) Silicon's technique for *framing* the values of heap-dependent expressions across heap changes — the snapshots introduced in Chapter 3 — needs to be generalised as well, since heap-dependent expressions, such as pure quantifiers over heap locations and functions whose preconditions use quantified permissions, may be framed by an unbounded number of symbolic values (see Appendix C, Listing C.2 for an example of such a function).

The technique presented in this chapter is the first to provide automated solutions to these challenging problems. Section 4.2 tackles the first three problems; Section 4.3 tackles the remaining two.

4.2 Treatment of Permissions

The canonical shape of quantified permission assertions as currently supported by Viper is

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

in which $c(x)$ is a boolean expression, $e(x).f$ denotes a field location, and $p(x)$ is an expression denoting a permission amount. We write $c(x)$ (and analogously, $e(x)$ and $p(x)$) to emphasise that the variable x may occur in the expression c (potentially in addition to other bound or free variables).

More-complex assertions can be desugared into this canonical shape, for instance, iterating over the conjunction of two accessibility predicates can be encoded by repeating the quantification over each conjunct. Nested quantified permission assertions are not yet supported, but an extension is possible (see also Section 4.6). The canonical shape is sufficient to directly support quantifying over all receivers in a set (useful for graph examples), and over integer indices into an array, as used by the running example shown in Listing 4.1.

The permission expression $p(x)$ may be a complex expression including conditionals, and need not evaluate to the same value for each instantiation of x. This enables encoding complex access patterns such as requiring non-zero permission to every nth slot of an array, which is, for example, important for the verification of GPU programs [16]. As before, pure quantifiers over potentially heap-dependent expressions can be used to specify functional properties.

In the remainder of this section, we present the first key ingredient of our symbolic execution technique: a representation of quantified permissions as part of the verifier's symbolic state along with algorithms to manipulate this representation.

4.2.1 Symbolic Heap Representation

To simplify the presentation, the core idea behind our heap representation is first introduced in the context of field locations, and then generalised to predicate instances; afterwards, the corresponding symbolic execution rules are presented in full detail.

Quantified Field Chunks

The field chunks introduced in Chapter 3 are of the shape $f(o; v, p)$, which can be understood as mapping a field receiver o to a location value v and permission amount p. A naïve generalisation of this representation would be to make o, v and p functions of the variable x bound by a quantified permission assertion (recall the canonical shape of quantified permission assertions introduced above). However, such a representation would have severe drawbacks. Checking whether a heap chunk provides permission to a location $y.f$ (challenge 2 above) amounts to the existential query $\exists x \cdot o(x) = y$, and SMT solvers typically provide poor support for such existential queries. In the presence of fractional permissions, determining *how much* permission such a heap chunk provides is worse still, requiring to calculate the sum of *all* $p(x_i)$ such that x_i satisfies the existential query.

Our design avoids these difficulties with a simple restriction: it requires the receiver expressions $e(x)$ in a quantified permission assertion to be *injective* in x, for all values of x to which the assertion provides permission. Under this restriction, it can soundly be assumed that the mapping between the bound variable x and receiver expression $e(x)$ is *invertible* for such values, by some function e^{-1}. A quantified permission assertion over receivers $r = e(x)$ can then be represented by directly quantifying over references r (instead of x): by replacing each occurrence of x with $e^{-1}(r)$ throughout the assertion.

The resulting design is to use *quantified field chunks* of the shape $f(r; sm(r), p(r))$, in which r, which is implicitly bound in such a chunk, plays the role of a quantified (reference-typed) receiver. A quantified field chunk represents $p(r)$ permission to all locations $r.f$; $p(r)$ may be any expression denoting a permission amount. The *domain* of a quantified field chunk is the set of field locations $r.f$ for which $p(r) > 0$. The values of these locations are modelled by the *snapshot map sm*: such maps from references to field values generalise the snapshots of non-quantified field chunks and record the snapshot (that is, value) per location to which a quantified field chunk provides non-zero permission. Details about snapshot maps are given in Section 4.3.

The aforementioned injectivity of field receiver expressions $e(x)$ is asserted when a quantified permission assertion is exhaled, and assumed when a quantified permission assertion is inhaled[2] (the corresponding symbolic execution rules are shown in Figure 4.1); injectivity is assumed by axiomatising a fresh inverse function. That is, inhaling a quantified permission assertion

`forall` x: T `::` $c(x)$ `==>` `acc(`$e(x).f$`,`$p(x)$`)`

entails declaring a fresh inverse function[3]

e^{-1}: $Ref \rightarrow T$

and adding a quantified field chunk

$f(r; sm(r), ite(c(e^{-1}(r)), p(e^{-1}(r)), 0))$

[2] In future versions of Viper (and Silicon) we might check injectivity on inhale as well, for example, in order to reduce the likelihood of not detecting inconsistent preconditions.

[3] For simplicity, we use the Viper type T in the presented encoding, instead of an appropriate sort S that (in Silicon's background theory, recall Section 3.1.2) corresponds to T; all following encodings in this chapter are simplified analogously.

to the symbolic heap that provides permissions to all field locations specified by the source-level quantified permission assertion (and zero permissions to all other locations). The inverse function *could* then be axiomatised by adding the following constraints to the path conditions (the final constraints are shown in the next subsection):

$$\forall r\colon Ref \cdot c(e^{-1}(r)) \wedge 0 < p(e^{-1}(r)) \Rightarrow e(e^{-1}(r)) = r$$
$$\forall x\colon T \cdot c(x) \wedge 0 < p(x) \Rightarrow e^{-1}(e(x)) = x$$

The injectivity restriction does not limit the data structures that can be handled by our technique, provided specifications are expressed appropriately. The restriction applies to memory *locations*, not to the *values* stored in the locations. Many examples such as quantified permissions ranging over array indices or elements of a set naturally satisfy the restriction. Ranges that may contain duplicates (for instance, the fields of all objects stored in an array) can be encoded by mapping them to a set (thereby ignoring multiplicities) or by using complex permission expressions p that reflect multiplicities appropriately.

Quantifications over Finite Domains

As it was shown above, the definition of inverse functions would implicitly assume that the set T over which the quantified permission assertion ranges is of the same cardinality as the set of references *Ref* — an assumption that would be unsound in general, for example, when quantifying over booleans. Consider the snippet

```
inhale forall b: Bool :: true ==> acc(e(b).f)
```

where the expression `e(b)` maps each `b` to one of two different references. If the generated first inverse axiom were

$$\forall r\colon Ref \cdot true \Rightarrow e(e^{-1}(r)) = r$$

then e^{-1} would constitute a bijection between *Ref* and *Bool*, and thus unsoundly assume that the two sets were of the same cardinality.

We prevent this unsoundness by effectively limiting the domain of the inverse function to the image of e, which we achieve by introducing an uninterpreted function img_e that identifies the references in the image of e. Concretely, we declare a fresh function

$$img_e\colon Ref \rightarrow Bool$$

that evaluates to *true* for each reference r in the image of e. The latter is axiomatised as follows:

$$\forall x\colon T \cdot c(x) \wedge 0 < p(x) \Rightarrow img_e(e(x))$$

Given such an image function img_e, we can now axiomatise the inverse function e^{-1} as follows:

$$\forall r\colon Ref \cdot img_e(r) \wedge c(e^{-1}(r)) \wedge 0 < p(e^{-1}(r)) \Rightarrow e(e^{-1}(r)) = r$$
$$\forall x\colon T \cdot c(x) \wedge 0 < p(x) \Rightarrow e^{-1}(e(x)) = x$$

General Quantified Chunks

The presented idea enables support for quantified permission assertions over field locations (and is already implemented in Silicon), but not yet for quantified permission assertions over predicate instances. A corresponding generalisation is possible, however (but not yet implemented in Silicon, see also Section 4.6), as is discussed next.

As the first step, the canonical shape of quantified permission assertions is generalised to

`forall` $x\colon T$ `::` $c(x)$ `==>` `acc(`$id(\overline{e(x)})$`,` $p(x)$`)`

where $id(\overline{e(x)})$, as in previous chapters, denotes either a predicate instance (with multiple arguments $\overline{e(x)}$) or a field location (in which case $id(\overline{e(x)})$ denotes a field location $e(x).id$). Since predicate instances may have multiple arguments, it is necessary to also generalise the inverse function that belongs to a quantified permission assertion such that it maps a vector of arguments $\overline{e(x)}$ to the quantified variable x. Such generalised inverse functions are axiomatised as

$$\forall \overline{r\colon E} \cdot img_e(\overline{r}) \wedge c(e^{-1}(\overline{r})) \wedge 0 < p(e^{-1}(\overline{r})) \Rightarrow \bigwedge \overline{e_i(e^{-1}(\overline{r})) = r_i}$$
$$\forall x\colon T \cdot c(x) \wedge 0 < p(x) \Rightarrow e^{-1}(\overline{e(x)}) = x$$

where the quantified variables $\overline{r}$ correspond to the arguments $\overline{e(x)}$: one variable $r_i\colon E_i$ per expression $e_i(x)\colon E_i$. The corresponding generalised image function img_e is axiomatised as follows:

$$\forall x\colon T \cdot c(x) \wedge 0 < p(x) \Rightarrow img_e(\overline{e(x)})$$

Finally, we generalise the previously introduced quantified field chunks to (general) *quantified chunks*, which are of the shape

$$id(\overline{r}; sm(\overline{r}), ite(c(e^{-1}(\overline{r})), p(e^{-1}(\overline{r})), 0))$$

where the snapshot map sm is of type $\overline{E} \rightarrow Snap$ (details about snapshot maps are given in Section 4.3), and where $e^{-1}\colon \overline{E} \rightarrow T$ is a generalised inverse function.

Integrating Quantified and Non-Quantified Permissions

Mixing quantified permission assertions with regular accessibility predicates (that is, non-quantified permission assertions) concerned with the same identifier (field or predicate) in specifications is desirable, and supported by Silicon. For example,:

```
inhale forall x: Ref :: x in xs ==> acc(x.f)
inhale y in xs
exhale acc(y.f)
```

For ease of presentation, throughout this chapter we treat regular accessibility predicates `acc(`$id(\overline{e})$`)` as syntactic sugar for the quantified permission assertion

`forall` $\overline{x\colon E}$ `::` $\overline{x == e}$ `==>` `acc(`$id(\overline{x})$`)`

For example, `acc(y.f)` is interpreted as

```
inhale forall x: Ref :: x == y ==> acc(x.f)
```

Correspondingly, consuming/producing a regular accessibility predicate is interpreted as consuming/producing the corresponding quantified permission assertion. Handling regular accessibility predicates this way makes it unnecessary to mix quantified and non-quantified chunks (for the same identifier) in a symbolic heap; in the rest of this chapter we therefore consider only symbolic heaps that are exclusively composed of quantified chunks. Silicon's implementation, however, special-cases regular accessibility predicates in order to optimise performance, as is discussed in Section 4.6.

4.2.2 Inhaling and Exhaling Quantified Permissions

The symbolic execution rules for producing and consuming quantified permission assertions, which are based on the symbolic heap design explained above, are shown in Figure 4.1. The evaluation performed in the first line will be discussed shortly; its relevant results are the symbolic expressions $c(x)'$, $\overline{e(x)'}$ and $p(x)'$.

---

1 $\mathsf{produce}(\sigma_1, \texttt{forall}\ x\colon T\ \texttt{::}\ c(x)\ \texttt{==>}\ \texttt{acc}(id(\overline{e(x)}), p(x)), sm, Q) =$
2 $\quad \mathsf{eval}(\sigma_1, \texttt{forall}\ x\colon T\ \texttt{::}\ \{\overline{e(x)}\}\ c(x)\ \texttt{==>}\ \mathcal{D}(\overline{e(x)}, p(x)),$
3 $\qquad (\lambda\ \sigma_2, (\forall x\colon T\ \cdot \{\overline{e(x)'}\}\ c(x)' \Rightarrow \mathcal{D}'(\overline{e(x)'}, p(x)')) \cdot$
4 $\qquad\quad$ Let img_e be a fresh function of type $\overline{E} \rightarrow Bool$
5 $\qquad\quad img_{def} := \forall x\colon T\ \cdot c(x)' \wedge 0 < p(x)' \Rightarrow img_e(\overline{e(x)'})$
6 $\qquad\quad$ Let e^{-1} be a fresh function of type $\overline{E} \rightarrow T$
7 $\qquad\quad inv_1 := \forall \overline{r\colon E}\ \cdot img_e(\overline{r}) \wedge \overline{c(e^{-1}(\overline{r}))' \wedge 0 < p(e^{-1}(\overline{r}))'}$
8 $\qquad\qquad\qquad\qquad \Rightarrow \bigwedge \overline{e_i(e^{-1}(\overline{r}))' = r_i}$
9 $\qquad\quad inv_2 := \forall x\colon T\ \cdot c(x)' \wedge 0 < p(x)' \Rightarrow e^{-1}(\overline{e(x)'}) = x$
10 $\qquad\quad ch := id(\overline{r}; sm(\overline{r}), ite(c(e^{-1}(\overline{r})), p(e^{-1}(\overline{r})), 0))$
11 $\qquad\quad h_3 := \sigma_2.h \cup \{ch\}$
12 $\qquad\quad \pi_3 := \mathsf{pc\text{-}add}(\sigma_2.\pi, \{inv_1, inv_2, img_{def}\})$
13 $\qquad\quad Q(\sigma_2\{h := h_3, \pi := \pi_3\})))$
14
15 $\mathsf{consume}'(\sigma_1, h, \texttt{forall}\ x\colon T\ \texttt{::}\ c(x)\ \texttt{==>}\ \texttt{acc}(id(\overline{e(x)}), p(x)), Q) =$
16 $\quad$ Proceed as above to obtain $c(x)', \overline{e(x)'}$ and $p(x)'$, let σ_2 be the post-state
17 $\quad$ Let y_1, y_2 be fresh symbolic constants of type T
18 $\quad \mathsf{assert}(\sigma_2.\pi, (c(y_1)' \wedge c(y_2)' \wedge 0 < p(y_1) \wedge 0 < p(y_2) \wedge \overline{e(y_1)' = e(y_2)'})$
19 $\qquad\qquad\qquad \Rightarrow y_1 = y_2)$
20 $\quad$ Introduce fresh functions e^{-1} and img_e,
21 $\qquad\qquad$ and axioms inv_1, inv_2 and img_{def} as above
22 $\quad sm, sm_{def}, \_ := \mathsf{qp\text{-}summarise}(h, id)$
23 $\quad h_3 := \mathsf{qp\text{-}remove}(\sigma_2.\pi, h, id, (\lambda\ \overline{r}\ \cdot ite(c(e^{-1}(\overline{r})), p(e^{-1}(\overline{r})), 0)))$
24 $\quad \pi_3 := \mathsf{pc\text{-}add}(\sigma_2.\pi, \{inv_1, inv_2, img_{def}, sm_{def}\})$
25 $\quad Q(\sigma_2\{\pi := \pi_3\}, h_3, sm)$

Figure 4.1: Produce and consume rules for quantified permission assertions. The consumption rule employs two auxiliary operations: qp-summarise yields a snapshot map *sm* that summarises the snapshots (heap values) of all chunks for *id*; qp-remove removes the permissions to be consumed from the symbolic heap and returns an updated heap. The assumptions (produce), respectively, assertions (consume) about field receivers not being null and permission amounts being non-negative, which are present in the regular produce/consume rules for permissions (Figure 3.8/Figure 3.9), have been omitted for brevity.

Following the encoding described in the previous subsection, the production rule introduces a fresh inverse function e^{-1} (starting on line 6), which is constrained as the partial inverse of the (evaluated) arguments $\overline{e(x)}$ by adding the axioms inv_1 and inv_2 to the path conditions. Similarly, it introduces a fresh image function img_e (starting on line 4), defined by the axiom img_{def}. The triggers chosen for the corresponding quantified axioms are discussed separately in Section 4.4. The rule also adds the newly-produced quantified chunk (line 10) to the current heap.

The snapshot map *sm* models the values of the heap locations in the domain of the new quantified chunk. As before (recall the discussion of using snapshots for framing from Section 3.3), the produce rule is parameterised with the snapshot (map) *sm*, which enables "producing" quantified chunks that assign particular values to the heap locations they provide permissions to. As before (recall Section 3.1.2), we omit explicitly boxing snapshots (of sort *Snap*) to/from the appropriate sorts (here $\overline{E} \rightarrow Snap$).

In order to construct symbolic expressions such as the definitional axioms of the newly-introduced inverse function, it is necessary to evaluate the pure subexpressions $c(x)$, $\overline{e(x)}$ and $p(x)$ of the quantified permission assertion — all of which potentially mention the quantified variable x — to the corresponding symbolic expressions $c(x)'$, $\overline{e(x)'}$ and $p(x)'$. Evaluating quantified expressions is technically involved, however: recall (from Figure 3.17 on page 82) that evaluating a quantified expression is not straightforward and requires the construction of two quantifiers, one corresponding to the evaluated expression and one aggregating additional path conditions obtained during the evaluation of the quantifier's body. In order to avoid duplicating significant parts of the corresponding evaluation rule, a dummy quantified expression containing the relevant pure subexpressions is constructed (on line 2), evaluated as usual (that is, by the rule for evaluating quantified expressions) and afterwards decomposed (via pattern matching on line 3) to obtain the desired symbolic expressions. To simplify constructing a type-correct dummy expression, the pure subexpressions are "wrapped" in an application of a dummy function $\mathcal{D}$ (with symbolic counterpart $\mathcal{D}'$) that itself has no meaning (that is, is not axiomatised in any way). A simple, syntactic pre-analysis — inspecting all quantified permission assertions syntactically occurring in the program under verification — suffices to determine which dummy functions $\mathcal{D}/\mathcal{D}'$ are required (and to then declare them): one function per vector of types $\overline{E}$.

Given the production rule from Figure 4.1, inhaling the second precondition of the running example from Listing 4.1 (at the start of checking the method body) entails introducing an inverse function a^{-1} mapping array locations back to corresponding indices, axiomatised as

$$\forall i : Int \cdot left \leq i < right \Rightarrow a^{-1}(a_i) = i$$
$$\forall r : Ref \cdot left \leq a^{-1}(r) < right \Rightarrow a_{a^{-1}(r)} = r$$

where a_i denotes the ith array location (not the value at that location), and then adding the quantified chunk

$$\texttt{val}(r; sm(r), (ite(left \leq a^{-1}(r) < right, 1, 0)))$$

Correspondingly, inhaling the first postcondition of the recursive calls yields a new inverse function each; and the symbolic heap at the program point after the calls contains two quantified chunks, one for each array segment.

The consumption rule for quantified permission assertions is initially similar to the production rule, one difference being that the injectivity of the argument expressions is checked before defining the inverse function. Removing permissions from the heap is more complex than adding permissions because it may involve updates to many existing quantified chunks in the symbolic state. This operation is delegated to the auxiliary operation qp-remove, shown in Figure 4.2.

The injectivity check performed when consuming a quantified permission assertion guarantees that the introduced inverse functions exist and satisfy the constraints added to the path conditions, which is required for soundness. If there is a corresponding exhale (consume) point for each inhaling (produce) of quantified permissions, then the check performed on exhale also covers the inverse functions introduced during a corresponding inhale.

In addition to an updated symbolic state, consuming a quantified permission assertion also returns (as before in Chapter 3) a snapshot that represents the values of those heap locations to which permissions have been removed. Computing this snapshot is delegated to the auxiliary operation qp-summarise, discussed in Section 4.3.1.

Removing Permissions

The previously shown rule for consuming a quantified permission assertion (Figure 4.1) delegates the task of removing sufficient permission from a given symbolic heap to qp-remove, which is shown in Figure 4.2 and discussed next.

```
1  qp-remove(π, h, id, q) =
2    Let h_id ⊆ h be all chunks for identifier id
3    h'_id := ∅
4    q_needed := q
5    foreach id(r̄; sm_i(r̄), q_i(r̄)) ∈ h_id do
6      q_current := λr̄ · min(q_i(r̄), q_needed(r̄))
7      q_needed := λr̄ · q_needed(r̄) − q_current(r̄)
8      h'_id := h'_id ∪ {id(r̄; sm_i(r̄), q_i(r̄) − q_current(r̄))}
9    assert(π, ∀r̄ · q_needed(r̄) = 0)
10   (h \ h_id) ∪ h'_id
```

Figure 4.2: The qp-remove operation. The symbolic expression q maps the arguments $\overline{r}$ of an identifier id (such as a single reference r if id denotes a field) to a permission amount. qp-remove checks that the symbolic heap contains at least $q(\overline{r})$ permission to each heap location $id(\overline{r})$, and removes it.

qp-remove takes as inputs a stack of path conditions π, a symbolic heap h, an identifier id (denoting a field or a predicate), and a function-typed symbolic expression q that yields, for each argument vector $\overline{r}$, the permission amount for location $id(\overline{r})$ to be removed. qp-remove fails with a verification error if the initial heap does not contain the permissions denoted by q, and otherwise returns an updated symbolic heap that provides q permissions less than the initial heap. This is achieved by iterating over all available chunks for id, greedily taking as much of the still-required permissions (q_{needed}) as possible from the current chunk ($q_{current}$). Updating the chunks is expressed by the pointwise construction of function-typed symbolic expressions describing the corresponding permission amounts; they involve permission arithmetic, but no existential quantifiers, and can be handled efficiently by the underlying SMT solver. After the iteration, qp-remove asserts that all requested permissions have been removed.

In the running example (Listing 4.1), the second precondition is exhaled (consumed) before each recursive call; this requires finding the appropriate permissions from the (single) quantified chunk in the state at this point, and removing them. Dually, when exhaling the postcondition at the end of the method body, all permissions from both of the two quantified chunks yielded by the recursive calls must be removed: the iteration in the qp-remove algorithm achieves this.

Note that qp-remove's permission accounting is precise, which is important for soundness and completeness: it maintains the invariant that (for all $\overline{r}$), the difference between the permissions held in the initial state and those requested via parameter q is equal

to the difference between those held in the updated state and those still needed. If the operation succeeds, the last check implies that those permissions still needed are exactly 0, from it follows that precisely the correct amounts were subtracted.

Relation to Disjunctive Aliasing

Recall from the classification of aliasing-related incompletenesses typically exhibited by verifiers based on symbolic execution (Section 3.4.2 and Section 3.7.3) that *disjunctive aliasing* describes situations in which a reference x is an alias of at least one out of several other references $\overline{y}$, but it is not statically known which y_i is aliased by x. Disjunctive aliasing naturally arises in the context of quantified permissions (and a verifier supporting quantified permissions should therefore not exhibit incompletenesses arising from disjunctive aliasing), for example, when exhaling permissions to a single slot $a[i]$, for an arbitrary but valid index i, of an array a to which quantified permissions have been inhaled: in this case $a[i]$ aliases one of the array slots $a[0], \ldots, a[n-1]$, but it is not statically known which one. Resolving disjunctive aliasing by forcing the verifier to branch over possible aliasing relations — a potential work-around suggested in Section 3.4.2 — is not possible in this situation because the number of aliasing relations is statically unknown (if the array length is).

Consuming (exhaling) permissions via qp-remove is complete in the presence of disjunctive aliasing because it accounts for *all* possible aliasing relations across *all* chunks (unlike the previously shown heap management algorithms from Chapter 3, which only consider a single, definitely aliased chunk): this is achieved by iteratively updating the still-needed permission expression q_{needed} such that, for a given reference r, the permission amount denoted by the expression is decreased by the permission amount that the current heap chunk could provide for r if the reference happened to be in the domain of the chunk (which may follow from the current path conditions). Indeed, the disjunctive aliasing tests discussed in Section 3.7.3 succeed if the quantified permission algorithms presented in this chapter are used, as illustrated by Listing C.6 in Appendix C.

4.3 Treatment of Symbolic Values

We have so far addressed the first three technical challenges described in Section 4.1 that need to be solved in order to support iterated separating conjunctions: we introduced quantified permissions, a language feature of Viper that enables encoding iterated separating conjunctions, and we presented a novel heap representation for quantified permissions together with algorithms that let the verifier efficiently add, as well as check for and remove permissions. In this section we present our solution to the remaining two challenges, concerned with the evaluation and framing of heap-dependent expressions: the handling of fields and predicates are discussed first, after which the treatment of heap-dependent functions is discussed. Operation qp-summarise, which has already been used in the consumption rule for quantified permission assertions (Figure 4.1), is introduced in the context of evaluating field reads. The operation is more general, however, and used in other contexts as well (for example, in Figure 4.4).

4.3.1 Handling Fields and Predicates

Quantified field chunks $f(r; sm(r), q(r))$ represent value information via the snapshot map sm. The existence of such a chunk in a symbolic heap allows, for any receiver in the domain of the heap chunk, the evaluation of a read of field f to an application of the snapshot map. Intuitively, sm represents a partial function from this domain to symbolic values (of the type of the field f). Since SMT solvers typically do not

natively support partial functions, snapshot maps are modelled as under-specified total functions from the receiver reference (the field f is fixed) to the type of f. These functions are *applied* only to references whose f field location is in the chunk's domain. This is why the consume rule for quantified permission assertions (Figure 4.1) does not need to explicitly remove information about the values stored in the locations whose permissions are removed; the underlying total function still represents appropriate values for the new (smaller) domain. Moreover, permissions are never added to already-existing chunks: instead, the rule for producing quantified permission assertions (Figure 4.1) adds a new chunk with a fresh snapshot map (whose domain is implicitly defined by the chunk's permissions).

Summarising Values

Inhaling permissions (Figure 4.1) adds a fresh heap chunk with a fresh snapshot map, and a symbolic heap may thus contain multiple chunks for the same field, each with its own snapshot map. In the presence of fractional permissions, the domains of these chunks may overlap such that the value of one location $x.f$ may be represented by multiple snapshot maps. Similarly, the value of $x.f$ may be represented by multiple maps when the receiver x is quantified over and the permissions to different instantiations of the quantifier are recorded in different chunks. Therefore, all of these value maps need to be considered when evaluating such a field access.

In order to incorporate information from all relevant chunks, and to provide a simple translation for field lookups, the snapshot maps of all chunks for a field f are *lazily summarised* before an expression $e.f$ is evaluated. This summarisation is defined by the qp-summarise operation shown in Figure 4.3. For each chunk with the appropriate identifier *id*, such as a field f, it equates a newly-introduced snapshot map with the snapshot map in the chunk at all locations in the chunk's domain. Analogously, it builds up a symbolic permission expression summarising the permissions held per field location, across all heap chunks for *id*; the resulting expression is later used to assert that appropriate permissions are held somewhere in the summarised state.

Note that the definition of qp-summarise does not depend on path conditions, only on the symbolic heap; it can be computed without querying the SMT solver. This is because all relevant information is maintained as symbolic expressions recorded by the chunks (which are updated by qp-remove), which can be combined by qp-summarise in order to achieve the desired summarisation (see Section 4.6 for a comment on the size of the expressions resulting from executions of qp-remove and qp-summarise).

---

1 qp-summarise$(h, id) =$
2 Let $h_{id} \subseteq h$ be all chunks for identifier *id*
3 Let *sm* be a fresh snapshot map of type $\overline{E} \to Snap$
4 $sm_{def} := \emptyset$
5 $perm := \lambda \overline{r} \cdot 0$
6 foreach $id(\overline{r}; sm_i(\overline{r}), q_i(\overline{r})) \in h_{id}$ do
7 $sm_{def} := sm_{def} \cup \{\forall \overline{r}: \overline{E} \cdot 0 < q_i(\overline{r}) \Rightarrow sm(\overline{r}) = sm_i(\overline{r})\}$
8 $perm := \lambda \overline{r} \cdot perm(\overline{r}) + q_i(\overline{r})$
9 $(sm, sm_{def}, perm)$

where $\overline{E}$ are the sorts corresponding to the arguments of *id*, for example, *Ref* if *id* denotes a field

---

Figure 4.3: The qp-summarise operation iterates over all chunks for *id* in a given heap and computes (1) a snapshot map *sm* that summarises the heap values potentially represented by multiple chunks, and (2) a permission-typed symbolic expression *perm* that summarises the permission amounts provided by individual chunks.

A New Evaluation Rule for Field Reads

Building on qp-summarise, the rule for symbolically evaluating field reads can be defined as shown in Figure 4.4. The evaluation proceeds by asserting that at least some permission to the field location is held in the current symbolic heap, followed by invoking the remainder continuation with the symbolic value of the field lookup: the summarising snapshot map applied to the field receiver. Via the path conditions generated by qp-summarise, any properties known about the snapshot maps of the corresponding quantified chunks will also be known about the resulting symbolic value. Silicon memoizes qp-summarise, avoiding the duplication of the function declarations and path conditions defining the snapshot maps and permission values; more details about this memoization are provided in Section 4.6.

```
1  eval(σ1, e.f, Q) =
2    eval(σ1, e, (λ σ2, e' ·
3      sm, sm_def, perm := qp-summarise(σ2.h, f)
4      assert(σ2.π, 0 < perm(e'))
5      Q(σ2{π := pc-add(σ2.π, sm_def)}, sm(e'))))
```

Figure 4.4: Symbolically evaluating a field read. The symbolic field value is obtained by applying the summarising snapshot map sm to the symbolic receiver e'.

Unchanged Field- and Predicate-Related Rules

The rules for executing assignments $e_1.f$ `:=` e_2 and `new` statements x `:= new(`$\overline{f}$`)`, respectively, do not need to change with respect to their definition from Figure 3.6, because they essentially desugar into already redefined operations: exhaling and inhaling permissions, and inhaling value constraints. A field write e_1`.`f `:=` e_2 is effectively desugared into the sequence of statements (where e_2' denotes the symbolic value of e_2)

```
exhale acc(e1.f)
inhale acc(e1.f)
inhale e1.f == e2'
```

The exhale checks that the heap has the required permission and removes it; the inhales create a new chunk with the previously-removed permission and constrain the associated snapshot map such that it maps receiver e_1 to the symbolic value of e_2.

For example, the field write `a[left] := to` in the running example (Listing 4.1 on page 105) is executed in a symbolic heap with a single quantified chunk that provides full permission to each array location. After the field write has been executed, the heap contains two quantified chunks: the initial one, still providing full permission to each array location *except for* `a[left]` (and with an unchanged snapshot map), and a second one that provides full permission to `a[left]` only, with a fresh snapshot map representing the updated value.

The rules for executing `fold` and `unfold` statements (Figure 3.6), as well as the rule for evaluating `unfolding` expressions (Figure 3.15), can also remain as previously defined because they also build on inhaling and exhaling permissions.

4.3.2 Heap-Dependent Functions

Framing Heap-Dependent Function Applications

Recall from Chapter 3 (in particular, from Figure 3.16) that Silicon's encoding of heap-dependent functions, and of applications thereof, uses function snapshots (introduced

by Smans et al. [119]) to represent the symbolic values of the heap locations a function depends on. Consequently, two function applications (on the SMT level) yield the same result if they take the same arguments and have equal function snapshots.

Quantified permissions complicate this approach because a function whose precondition contains a quantified permission assertion may depend on an unbounded set of heap locations, and the values of these locations cannot be represented by a fixed number of snapshots. It is also not possible to represent them directly as a snapshot map since these are modelled at the SMT level as under-specified total functions, causing two problems. First, requiring equality of total functions would include locations the heap-dependent function does not actually depend on; since the values for these locations are under-specified, the equality check would fail even when the function value could be soundly framed. Second, a function cannot be used as a function argument in the first-order logic supported by SMT solvers.

We address both problems by introducing a second kind of snapshot maps, called *partial snapshot maps*, which are used instead of the previously introduced snapshot maps whenever such a map is used as the snapshot of a heap-dependent function application (for performance reasons we continue to encode snapshot maps that are not used as function snapshots, such as those recorded in quantified chunks, as total functions on the SMT level). Partial snapshot maps are encoded on the SMT level by applying *defunctionalisation* [107] (addressing problem two) and by explicitly modelling their partial domains (addressing problem one). The encoding requires the following steps:

- Add an uninterpreted sort *PSM* to the background theory, alongside two functions per field or predicate identifier *id* (used in the program under verification): $domain_{id}\colon PSM \rightarrow Set[\overline{E}]$ and $lookup_{id}\colon PSM \rightarrow \overline{E} \rightarrow Snap$, where $\overline{E}$ are the sorts of the parameters of *id*. Function $domain_{id}$ represents the domain of the partial snapshot map, and $lookup_{id}$ is used for applying a partial snapshot map (to a receiver reference or to predicate arguments). If *id* denotes a field, $\overline{E}$ is the single sort *Ref*, and the result of applying the lookup function needs to be boxed to the expected field type (omitted as usual).
- In all rules and operations presented so far (and in the remainder of this chapter), snapshot map lookup $sm(\overline{r})$ must be treated as syntactic sugar for $lookup_{id}(sm, \overline{r})$ if *sm* denotes a *partial* snapshot map (this information and the identifier *id* can be preserved by additional, straightforward bookkeeping).
- In order to prove equality when two partial snapshot maps are equal as partial functions, add the following extensionality axiom per field or predicate identifier *id*:

$$
\begin{aligned}
&\forall\, psm_1, psm_2\colon PSM \cdot \\
&\quad domain_{id}(psm_1) = domain_{id}(psm_2) \wedge \\
&\quad \forall \overline{r\colon E} \cdot \overline{r} \in domain_{id}(psm_1) \Rightarrow lookup_{id}(psm_1, \overline{r}) = lookup_{id}(psm_2, \overline{r}) \\
&\Rightarrow \\
&\quad psm_1 = psm_2
\end{aligned}
$$

- Change the rule for symbolically evaluating function applications (Figure 3.16) such that it sets a partiality flag (that is, a new state entry such as $\sigma.psm$) to true before the function precondition is consumed, and back to false afterwards. The flag is used to indicate that snapshot maps obtained from consuming quantified permission assertions that occur in the function's precondition are encoded as *partial* snapshot maps.
- Change qp-summarise (Figure 4.3) such that it creates and axiomatises partial snapshot maps if the partiality flag is set (line 3), which includes adding the following path condition that defines the domain of the newly introduced partial

snapshot map sm (c, e^{-1} and p need to be passed as arguments to qp-summarise):

$$\forall \overline{r : E} \cdot \bar{r} \in domain_{id}(sm) \Leftrightarrow c(e^{-1}(\bar{r})) \wedge 0 < p(e^{-1}(\bar{r}))$$

That is, the domain of a partial snapshot map consists of all heap locations for which the corresponding quantified permission assertion specified non-zero permissions.

In order to prevent the solver from reasoning about snapshot map equality outside of a partial snapshot map's domain, the axiom (line 7 of Figure 4.3) that pointwise relates the summarising partial snapshot map to the snapshot map provided by a chunk (encoded as a total function) needs to be changed as follows:

$$sm_{def} \coloneqq sm_{def} \cup \{\forall \overline{r : E} \cdot 0 < q_i(\bar{r}) \wedge \bar{r} \in domain_{id}(sm) \Rightarrow sm(\bar{r}) = sm_i(\bar{r})\}$$

That is, by adding (if the partiality flag is set) the second conjunct on the left of the implication, which restricts applications of the partial snapshot map to elements from its domain. This mirrors the already present restriction $0 < q_i(\bar{r})$, which restricts the equality to elements from the domain of the chunk's snapshot map: as defined in Section 4.2.1, the domain of a chunk's snapshot map is implicitly defined by the permissions q_i the chunk provides.

Axiomatising Heap-Dependent Functions

Recall from Section 3.6 that Silicon axiomatises heap-dependent functions by first recording mappings from heap-dependent expressions (for example, field reads) to the corresponding symbolic values (that is, to the snapshots of the accessed heap chunks) while the function is checked for well-definedness, followed by translating the function definition into a definitional axiom and a postcondition axiom. In the latter step, the previously recorded mappings are used to translate heap-dependent expressions.

In order to integrate quantified permissions into this approach to axiomatising heap-dependent functions, it is additionally necessary to record all definitions that arise from using quantified permissions in functions, such as the inverse functions introduced by using quantified permission assertions in a function's precondition or the definitions of snapshot maps introduced by qp-summarise (Figure 4.3) when evaluating a heap access in a function's body: it is these definitions that give meaning to the symbolic values that are (already being) recorded in the mapping from heap-dependent expressions to symbolic values. During the translation of a heap-dependent function into corresponding axioms, the recorded definitions are then included as part of the definitional and the postcondition axiom.

```
1 function eq(xs: Seq[Ref], ys: Seq[Ref]): Bool
2   requires |xs| <= |ys|
3   requires forall i: Int :: 0 <= i && i < |xs| ==> acc(xs[i].f)
4   requires forall i: Int :: 0 <= i && i < |xs| ==> acc(ys[i].f)
5   ensures  result <==> forall i: Int ::
6                              0 <= i && i < |xs|
7                         ==> xs[i].f == ys[i].f
```

Listing 4.2: An abstract function that defines equality between two sequences of heap-allocated cells by pairwise comparison of the cells' values.

Without quantified permissions, the map from heap-dependent expressions to symbolic values records, for a field read $x.f$, a mapping from $x.f$ to some chunk's snapshot s; with quantified permissions, the recorded mapping is from $x.f$ to the application $sm(x)$ of a snapshot map sm constructed by qp-summarise. To ensure that all constraints that determine the value represented by sm are preserved when the function is

axiomatised, it is necessary to record also the following information: (1) the definition of *sm* itself, as emitted by qp-summarise (Figure 4.3), (2) the definition of *sm*'s domain if qp-summarise declares a *partial* snapshot map, (3) the definition of inverse functions e^{-1} introduced when producing/consuming quantified permission assertions that directly (in a function's precondition) or indirectly (for example, due to an unfolding in the function's body) occur in the function definition, and (4) the definition of the image functions img_e that are introduced alongside inverse functions (recall Section 4.2.1).

As an example, consider the heap-dependent function `eq` from Listing 4.2 that expresses equality of two sequences of simple cells (objects with a single integer field `f`) pairwise in terms of equality of the cells' values. The function is abstract and its axiomatisation therefore consists of the following postcondition axiom only:

$$
\begin{aligned}
&\forall\, xs, ys, s \cdot \\
&\quad |xs| \leq |ys| \Rightarrow \\
&\qquad eq(xs, ys, s) \Leftrightarrow \forall\, 0 \leq i < |xs| \cdot sm_1(xs[i], xs, ys, s) = sm_2(ys[i], xs, ys, s) \\
&\quad \text{/* Def. of image function } img_{xs} \text{ introduced in line 3 */} \\
&\quad \wedge\, \forall i \cdot 0 \leq i < |xs| \Rightarrow img_{xs}(xs[i], xs, ys, s) \\
&\quad \text{/* Def. of image function } img_{ys} \text{ introduced in line 4 */} \\
&\quad \wedge\, \forall i \cdot 0 \leq i < |xs| \Rightarrow img_{ys}(ys[i], xs, ys, s) \\
&\quad \text{/* Def. of inverse function } xs^{-1} \text{ introduced in line 3 */} \\
&\quad \wedge\, \forall i \cdot 0 \leq i < |xs| \Rightarrow xs^{-1}(xs[i], xs, ys, s) = i \\
&\quad \wedge\, \forall r \cdot img_{xs}(r, xs, ys, s) \wedge 0 \leq xs^{-1}(r, xs, ys, s) < |xs| \Rightarrow xs[xs^{-1}(r, xs, ys, s)] = r \\
&\quad \text{/* Def. of inverse function } ys^{-1} \text{ introduced in line 4 */} \\
&\quad \wedge\, \forall i \cdot 0 \leq i < |xs| \Rightarrow ys^{-1}(ys[i], xs, ys, s) = i \\
&\quad \wedge\, \forall r \cdot img_{ys}(r, xs, ys, s) \wedge 0 \leq ys^{-1}(r, xs, ys, s) < |xs| \Rightarrow ys[ys^{-1}(r, xs, ys, s)] = r \\
&\quad \text{/* Def. of snapshot map } sm_1 \text{ for field read } \texttt{xs[i].f} \text{ in line 7 */} \\
&\quad \wedge\, \forall r \cdot 0 \leq xs^{-1}(r, xs, ys, s) < |xs| \Rightarrow sm_1(r, xs, ys, s) = first(s)(r) \\
&\quad \wedge\, \forall r \cdot 0 \leq ys^{-1}(r, xs, ys, s) < |xs| \Rightarrow sm_1(r, xs, ys, s) = second(s)(r) \\
&\quad \text{/* Def. of snapshot map } sm_2 \text{ for field read } \texttt{ys[i].f} \text{ in line 7 */} \\
&\quad \wedge\, \forall r \cdot 0 \leq xs^{-1}(r, xs, ys, s) < |xs| \Rightarrow sm_2(r, xs, ys, s) = first(s)(r) \\
&\quad \wedge\, \forall r \cdot 0 \leq ys^{-1}(r, xs, ys, s) < |xs| \Rightarrow sm_2(r, xs, ys, s) = second(s)(r)
\end{aligned}
$$

The first main conjunct of the axiom (the first and second line of the body of the outermost quantifier) encodes the function's postcondition; all subsequent conjuncts correspond to quantified-permission-related definitions. The first two pairs of subsequent conjuncts are the definitions of the image function (recall Section 4.2.1) introduced when inhaling the preconditions on line 3, respectively, line 4 of Listing 4.2; the next two pairs are the corresponding inverse functions; the last two pairs are the definitions of the summarising snapshot maps[4] introduced during the evaluation of the field reads `xs[i].f`, respectively, `ys[i].f` on line 7. For soundness, the recorded functions must be functions of the arguments of the axiomatised heap-dependent function (here *eq*), as illustrated by the application $sm_1(xs[i], xs, ys, s)$.

Function `eq` has two preconditions that require permissions, and the function snapshot s is therefore a pair of snapshots, that is, $s = (first(s), second(s))$. The first component, $first(s)$, is the snapshot map of the quantified chunk that corresponds to the quantified permission assertion on line 3 of Listing 4.2 (as usual, snapshots are implicitly unboxed to the appropriate type), and analogously for the second component and the assertion on line 4.

[4] Recall that the implementation memoizes snapshot maps: the definitions of sm_1 and sm_2 are equivalent, and Silicon's implementation only introduces a single summarising snapshot map *sm*.

Illustrating Function Framing

In order to illustrate how the previously discussed postcondition axiom is used to frame applications of `eq` across unrelated heap changes we will step through the symbolic execution of a simple client of `eq` next. The client starts by inhaling permissions to two arrays `xs` and `ys` (for ease of presentation, we omit details such as `eq`'s first postcondition `|xs| < |ys|` and simply assume that they are satisfied):

```
inhale forall i: Int :: 0 <= i && i < |xs| ==> acc(xs[i].f)
inhale forall i: Int :: 0 <= i && i < |ys| ==> acc(ys[i].f)
```

According to the rule for producing quantified permission assertions from Figure 4.1, this adds two heap chunks $f(r; sm_1(r), q_1(r))$ and $f(r; sm_2(r), q_2(r))$ to the symbolic heap, where the symbolic permission expression q_1 is $\lambda r \cdot ite(c_1(xs_1^{-1}(r)), 1, 0)$ and where the condition c_1 is $\lambda i \cdot 0 \leq i < |xs|$. Moreover, appropriate definitions (according to Figure 4.1) of the inverse function xs_1^{-1} and the image function img_{xs_1} are added to the path conditions. The constituents of the second heap chunk, q_2, c_2, ys_1^{-1} and img_{ys_1} are defined analogously. The client continues by assuming that `xs` and `ys` are equal according to `eq`:

```
inhale eq(xs, ys)
```

Following Section 4.3.2, this adds the path condition $eq(xs, ys, (psm_1, psm_2))$, where the partial snapshot map psm_1 (and analogous for psm_2) is defined according to Section 4.3.2, that is, by the following path conditions:

$$\forall r\colon Ref \cdot r \in domain_f(psm_1) \Leftrightarrow c_1(xs_1^{-1}(r)) \wedge 0 < q_1(r)$$
$$\forall r\colon Ref \cdot 0 < q_1(r) \wedge domain_f(psm_1) \Rightarrow psm_1(r) = sm_1(r)$$
$$\forall r\colon Ref \cdot 0 < q_2(r) \wedge domain_f(psm_1) \Rightarrow psm_1(r) = sm_2(r)$$

Next, the client updates some heap location `z.f` which is known to not be aliased from `xs` or `ys` (the client previously inhaled write permission to `z.f`):

```
z.f := 0
```

Recall that a field assignment can be treated as exhaling and re-inhaling permission (Section 4.3.1), and that exhaling permission to a single location can be handled by exhaling an appropriate quantified permission assertion that denotes permission to a single location only (Section 4.2.1). The chunks for field `f` (of which there are three in the heap: two for the arrays `xs` and `ys`, and one for `z.f`) are thus updated according to qp-remove from Figure 4.2: by updating the symbolic permission expressions q_i such that they no longer provide permission to `z.f`. In case of the chunks for `xs` and `ys`, however, this update does not change the denoted permission values because the solver can deduce (from the inverse functions and the permission-induced information that `z.f` is not aliased) that the location `z.f` was not in the domain of the chunks corresponding to the arrays `xs` and `ys`. For simplicity, we therefore continue the execution as if qp-remove had updated the chunk for `z.f` only, but not those for the arrays `xs` and `ys`. Finally, the client asserts that `xs` and `ys` are still equal:

```
assert eq(xs, ys)
```

This introduces two new partial snapshot maps psm_3 and psm_4, whose definitions are equivalent to those for the previously introduced maps psm_1 and psm_2. Using these definitions and the extensionality axiom for partial snapshot maps (introduced at the beginning of Section 4.3.2), the solver can conclude that psm_3 and psm_4 are equal to psm_1 and psm_2, respectively, and thus that $eq(xs, ys, (psm_3, psm_4))$ is equal to the previously added path condition $eq(xs, ys, (psm_1, psm_2))$, which proves the client's assertion.

Performance

Silicon's support for quantified permissions performs well in general, as demonstrated by the evaluation in Section 4.5, both for succeeding and failing verification runs. However, we observed performance degradation in examples with lots of (repeated) function applications: for example, given the function definition

```
function foo(xs: Set[Ref]): Bool
  requires forall x: Ref :: x in xs ==> acc(x.f)
```

the following snippet verifies instantaneously

```
inhale forall x: Ref :: x in xs ==> acc(x.f)
inhale foo(xs)
exhale foo(xs)
```

If the `exhale` is repeated 20 times (without any heap modifications in between), Silicon needs about five seconds to verify the example, and with 40 repetitions the verification time increases to more than 100 seconds. In contrast, Viper's second, verification-condition-generation-based verifier exhibits essentially the same verification time for all three versions of the example. We have not yet investigated the problem thoroughly, but we see two potential (not necessarily orthogonal) causes for the slowdown: (1) repeated applications of `qp-remove`, performed when exhaling the function's precondition, which update heap chunks in order to subtract the permissions required by the precondition; and (2) repeated introductions of fresh but semantically equivalent inverse functions (for the quantified permission assertion in the precondition) and snapshot maps (passed to the function as its heap snapshot).

As future work, we plan to address the performance issue in two ways, the first of which is to implement additional memoization and caching mechanisms that reduce the number of introduced inverse functions and snapshot maps, similar to the already implemented memoization of `qp-summarise` that reduces the number of snapshot maps introduced by the latter (further details are provided in Section 4.5 and Section 4.6). In addition, we plan to reconsider the semantics of function preconditions with respect to permissions: currently, function preconditions combine permissions via separating conjunctions (as do all other assertions), but since functions can only read the heap, regular (non-separating) conjunctions typically result in function definitions that, with respect to non-permission-related properties, are equivalent to the definitions with enforced separation. We believe that such a change can improve the handling of heap-dependent functions because it will no longer be necessary to (temporarily) exhale permissions when checking a function application's precondition, which in turn avoids modifying the symbolic heap, and in the context of quantified permissions, introducing new definitions (for example, of inverse functions).

4.4 Controlling Quantifier Instantiations

We so far omitted the triggers from quantifiers that are generated by our technique for supporting quantified permissions, but as previously discussed (Section 2.6), it is important to carefully control quantifier instantiations when working with SMT solvers. In this section, we recapitulate the quantifiers generated by our approach, and discuss the choice of their triggers.

Instantiations of the axioms that define inverse and image functions (recall Section 4.2.1) are controlled as follows (the axioms are unchanged, and only repeated

here to provide context for the triggers):

$$\forall \overline{r : E} \cdot \{e^{-1}(\bar{r})\}\ img_e(\bar{r}) \wedge c(e^{-1}(\bar{r})) \wedge 0 < p(e^{-1}(\bar{r})) \Rightarrow \bigwedge \overline{e_i(e^{-1}(\bar{r})) = r_i}$$
$$\forall x : T \cdot \{\overline{e(x)}\}\ c(x) \wedge 0 < p(x) \Rightarrow e^{-1}(\overline{e(x)}) = x$$
$$\forall x : T \cdot \{\overline{e(x)}\}\ c(x) \wedge 0 < p(x) \Rightarrow img_e(\overline{e(x)})$$

The first axiom's trigger, $e^{-1}(\bar{r})$, is essential for relating occurrences of the inverse function to the original expressions $\bar{e}$; which is particularly important since quantified chunks mention only the inverse function, not the $\bar{e}$ themselves. The case of the second axiom's trigger is almost symmetrical, but comes with extra technicalities. Since the $\bar{e}$ come from the source program, they may not (all) be expressions allowed in triggers. Recall from the discussion in Section 2.6 that triggers must typically include at least one function application (if an $e(x)$ were simply x, it could not be used), and no built-in operators such as addition. In the former case, the trigger $sm(x)$ is used, where sm is the snapshot map of the relevant chunk; the quantifier will then be instantiated whenever a value from the chunk is looked up, which is when the definition of the inverse function is needed. In the latter case, Viper's trigger inference attempts to find triggers (Section 2.6); if it fails we resort to leaving the SMT solver to infer triggers. The trigger for the third axiom (defining img_e) is also $\overline{e(x)}$ (if possible; otherwise, the same alternative triggers as for the second axiom are used): this ensures that the solver can use the definition of img_e whenever it instantiates the first axiom, which uses img_e.

Instantiating either of the two inverse axioms gives rise to potentially new function application terms suitable for triggering the other axiom. For example, when instantiating the second axiom due to a (single) term of the shape $e(x)$, the equality $e^{-1}(e(x)) = x$ is obtained, in which the function application $e^{-1}(e(x))$ matches the trigger of the first inverse axiom. Instantiating this axiom, in turn, will provide the equalities $e(e^{-1}(e(x))) = e(x)$. Note however, that this will not cause a matching loop, because SMT solvers consider quantifier instantiations *modulo* known equalities. Thus, the function application $e(e^{-1}(e(x))) = e(x)$ does not give rise to a *new* instantiation of the second axiom, since the term $e^{-1}(e(x))$ to be matched against the quantified variable is already known to be equal to x, which was used for the prior instantiation.

Instantiations of the axiom that relates summarising snapshot maps to the snapshot maps of heap chunks (see qp-summarise from Figure 4.3) are controlled by two alternative triggers:

$$\forall \overline{r : E} \cdot \{sm(\bar{r})\}\{sm_i(\bar{r})\}\ 0 < q_i(\bar{r}) \Rightarrow sm(\bar{r}) = sm_i(\bar{r})$$

The alternatives allow instantiating the axiom if *either* of the two snapshot maps has been applied to the terms instantiating $\bar{r}$. This flexibility is important because of how qp-summarise relates the newly-generated snapshot map to each quantified chunk individually, which indirectly allows deriving relationships between two evaluated expressions: the evaluations introduce summarising snapshot maps $sm_a(\bar{r})$ and $sm_b(\bar{r})$, respectively, each of which is related to all $sm_i(\bar{r})$ in the heap; hence, relationships between $sm_a(\bar{r})$ and $sm_b(\bar{r})$ can be derived via the $sm_i(\bar{r})$.

Instantiations of the domain definition axiom emitted by the updated qp-summarise operation described in Section 4.3.2 — extended such that it can generate partial snapshot maps — are controlled as follows:

$$\forall \bar{r} \cdot \{\bar{r} \in domain_{id}(sm)\}\ \bar{r} \in domain_{id}(sm) \Leftrightarrow c(e^{-1}(\bar{r})) \wedge 0 < p(e^{-1}(\bar{r}))$$

The trigger allows instantiating the axiom whenever the solver needs to learn which locations are in the domain of the partial function modelled by a partial snapshot map. In particular, the extensionality axiom for partial snapshot maps (recapitulated below) involves proving that the domains of two such maps are equivalent, which in general requires instantiating the corresponding domain definition axioms.

The last axiom to consider is the extensionality axiom, introduced in Section 4.3.2 in order to prove equality of partial snapshot maps. Recall that partial snapshot maps are used only as function snapshots, to frame applications of heap-dependent functions, and that (dis)equality of two such maps is relevant only when reasoning about the corresponding function applications. To control the number of potential instantiations, we exploit a technical detail of our encoding:

$$\begin{aligned}&\forall\, psm_1, psm_2 : PSM \cdot \{box_{PSM}(psm_1), box_{PSM}(psm_2)\} \\ &\quad domain_{id}(psm_1) = domain_{id}(psm_2) \wedge \\ &\quad \forall \overline{r : E} \cdot \bar{r} \in domain_{id}(psm_1) \Rightarrow lookup_{id}(psm_1, \bar{r}) = lookup_{id}(psm_2, \bar{r}) \\ &\quad \Rightarrow psm_1 = psm_2\end{aligned}$$

The chosen trigger allows the solver to instantiate the axiom with partial snapshot maps that are used as function snapshots, which will be embedded into the snapshot sort *Snap* by "wrapping" them in an application of the appropriate *box* function (introduced in Section 3.1.2).

4.5 Evaluation

To evaluate the performance of our techniques as implemented in Silicon, we ran experiments with three kinds of input programs: (1) nine hand-coded verification problems involving arrays and graphs (listed below), including the running example, (2) 65 examples generated by VerCors [2], which use our implementation to encode GPU verification problems, and (3) 82 additional regression tests. All examples are included in Viper's test suite, which is part of Viper's sources. Further information can be found on the Viper project page [94].

The hand-coded verification problems are the following:

- `arraylist` is an encoding of a list implemented on top of an array, with operations to append an element to the list, and to insert an element into the list such that the list, if it was sorted before, remains sorted afterwards.
- `array-quickselect` is an encoding of a (recursive) quickselect implementation over an array, with strong specifications such as "the array has been permuted", and "the nth smallest element has been selected".
- `binary-search-array` is an encoding of an (iterative) binary search performed over a sorted array.
- `graph-copy` is the encoding of an algorithm that copies a graph. Its specifications make use of a custom axiomatisation of maps to record relations between original and copied nodes.
- `graph-marking` is the encoding of a graph marking algorithm, in the spirit of mark-and-sweep garbage collectors, with strong specifications such as "nodes reachable from marked nodes are marked themselves".
- `longest-common-prefix` is a challenge from the VerifyThis Verification Competition 2012: finding the longest common prefix of two arrays.
- `max-array-elimination` is a challenge from the COST Verification Competition 2011: finding the maximum in an array by elimination.
- `max-array-standard` is an encoding of the straightforward way of finding the maximum in an array; it uses the same interface specifications and the same client as the previous example.
- `parallel-array-replace` is the running example from this paper: replace each occurrence of an element in an array segment by recursing over the two half-segments in parallel.

Program	Size (LOC)	Time (s)	w/o memoiz.	w/o triggers	w/o ordering
`arraylist`	114	1.93	−7.29%	−16.53%	+2.82%
`quickselect`	132	2.51	+24.44%	−4.23%	+1.92%
`binary-search`	47	0.31	+14.15%	−8.94%	+20.22%
`graph-copy`	120	1.81	+14.93%	+21.21%	+82.49%
`graph-marking`	53	1.71	+41.29%	−30.95%	+2.72%
`longest-common-prefix`	34	0.19	+6.51%	−10.73%	+24.59%
`max-elimination`	59	0.50	+45.41%	−0.07%	+11.66%
`max-standard`	53	0.24	+16.40%	+2.43%	+24.82%
`parallel-replace`	56	0.27	+3.71%	−6.12%	+20.56%

Figure 4.5: Performance evaluation of Silicon on verification challenges. Lines of code (LOC) neither includes blank lines nor comments. Column "Time (s)" gives runtimes of the base version of Silicon; the remaining columns show the percentage difference in time relative to the base version.

	No.	Size	Time		w/o memoiz.		w/o triggers		w/o ordering	
	Files	Mean	Mean	Max	Mean	Max	Mean	Max	Mean	Max
Program Set	(#)	(LOC)	(s)	(s)	(±)	(s)	(±)	(s)	(±)	(s)
VerCors	65	104	0.72	11.81	+0.92%	15.71	-4.40%	8.83	+64.80%	58.04
Regressions	82	34	0.22	3.41	+0.58%	3.81	-2.24%	3.38	+62.88%	4.86

Figure 4.6: Performance evaluation of Silicon on two sets of programs: the "VerCors" set contains (non-trivial) programs generated by the VerCors tool, "Regressions" contains (usually simple) regression tests; column "Files" displays the number of files per program set.

Figure 4.5 shows the results for (1), and Figure 4.6 those for (2) and (3). The experiments were performed on an Intel Core i7-4770 3.40GHz with 16GB RAM machine running Windows 7 x64 with an SSD. The reported times are averaged over 10 runs of each verification (with negligible standard deviations). Timings do not include JVM start-up; as before, the Nailgun tool [88] was used to persist a JVM across test runs.

The experiments show that Silicon is consistently fast: all examples verify in a few seconds. Since SMT encodings sometimes exhibit worse performance for *failed* verification attempts, we also tested four variants of each example from Figure 4.5 in which errors were seeded: in all cases the errors were detected with lower runtimes (which can be explained by the fact that Silicon stops verifying the current verification unit, such as a method, as soon as the first error is detected).

To evaluate the impact of the chosen quantifier triggers on the performance, (see Section 4.4), we also compare with a variant of the implementation in which triggers are omitted, resorting to Viper (and Z3) to infer triggers. The relative times are shown in the "w/o triggers" columns. It can be observed that this variant typically *improves* verification time. However, the triggers chosen automatically by Viper and Z3 are too strict: 7% of the programs (11 out of the 156 original programs) fail spuriously in this version. This, as well as a general reduction in quantifier instantiations, explains the effect on the runtime: the longest-running example in the base implementation (averaging 11.82s) takes only 3s without predefined triggers, but incorrectly fails to verify. The longest-running example in the variant without triggers takes 8.83s but also has a high standard deviation of 4.71s, suggesting that performance becomes unpredictable when triggers are selected automatically. The triggers that we choose thus avoid spurious errors and provide predictable, fast performance.

To measure the effect of memoizing invocations of `qp-summarise`, we disabled this feature and measured the difference in runtimes over the same inputs. As shown in the "w/o memoiz." columns, disabling this optimisation typically increases the runtime, but not enormously; a likely explanation for the relatively small difference

is that qp-summarise performs the iteration over quantified chunks without querying the SMT solver, and that the chosen quantifier triggers are sufficiently restrictive. The number of chunks in a given symbolic state is also typically kept small: Silicon performs modular verification per method/loop body, and eagerly removes quantified chunks that no longer provide permissions (after an exhale, see Section 4.6).

The columns labelled "w/o ordering" show the effect of another optimisation: a heuristic that aims at ordering the chunks that qp-remove iterates over such that the likelihood of finding the required permissions among the first few chunks is increased, which allows qp-remove to terminate prematurely. More details about this optimisation are given in Section 4.6.

4.6 Implementation

Observations and Optimisations

When discussing qp-summarise (Figure 4.3 on page 114), it was mentioned that Silicon memoizes snapshot maps in order to reduce the number of introduced maps, which in general reduces the verification time (as shown in Section 4.5). The current memoization strategy is simple: Silicon maintains a cache that maps a set of heap chunks (all for the same location identifier such as a field name) to a snapshot map previously introduced when summarising the heap values recorded by this set of chunks; if a cache entry exists the previously computed snapshot map is reused (instead of recomputing the summary). Cache entries are kept for the current execution path only, that is, removed when Silicon backtracks, which prevents potentially unsound reuse of snapshot maps on paths with equivalent symbolic heaps but different path conditions. As future work, we plan to investigate the possibility of memoizing inverse functions as well.

Another optimisation is concerned with the handling of non-quantified permission assertions: in Section 4.2.1, it was stated that a (non-quantified) assertion $\texttt{acc}(id(\overline{e}))$ could be understood as syntactic sugar for the quantified permission assertion $\texttt{forall}\ \overline{x : E}\ \texttt{::}\ \overline{x == e}\ \texttt{==>}\ \texttt{acc}(id(\overline{x}))$. To optimise performance, Silicon does not implement this desugaring, and instead special-cases non-quantified permission assertions. On inhale, such an assertion still yields a quantified chunk (not mixing quantified and non-quantified permissions in a symbolic heap simplifies the execution, and a simple, syntactic pre-analysis suffices to determine which fields are used in combination with quantified permissions), but no inverse function is introduced (which would be axiomatised as the identity function). On exhale, the axioms introduced by qp-summarise (defining the values, and potentially the domain, of a snapshot map) are not quantified, but instead pre-instantiated with the single possible argument vector (for example, the domain axiom would be $domain_{id}(sm) = \{\overline{e}\}$).

It was mentioned in Section 4.5 that Silicon removes heap chunks that no longer provide any permissions: this is done in the loop in qp-remove (Figure 4.2). This check amounts to an SMT solver query, whose runtime potentially outbalances the gain possibly obtained from reducing the heap size. A coarse experimental evaluation performed during the development indicated that a good performance trade-off is achieved by running this query with a short time-out (with a current default value of 250ms).

Silicon also implements a simple, syntax-driven heuristic that determines the order in which qp-remove iterates over the heap chunks to find the required permissions, aiming to increase the likelihood that the iteration can be ended prematurely. The heuristic implements the observation that permissions are often inhaled and subsequently exhaled with syntactically similar receiver expressions (respectively, predicate argument expressions). This is typically the case for loop invariants, but also common in other specifications, for example, in a method contract that requires and

ensures permissions to all `xs[i]`. The heuristic therefore orders the heap chunks according to how closely the exhale-receiver and the inhale-receiver (who need to be recorded in heap chunks on inhale) match. To benefit from the ordering, qp-remove checks after every iteration if sufficient permission have already been removed, in which case the iteration is ended prematurely. The corresponding solver query is again run with a short time-out (defaulting to 250ms) in order to prevent that repeated (unsuccessful) queries render the optimisation counter-productive. Ending the iteration early also helps with reducing the size of the generated symbolic permission expressions: qp-remove modifies the permission expression of each chunk it iterates over by subtracting the permission amount the chunk can provide to certain locations. If the chunk cannot provide any permissions to these locations then the solver will deduce that the subtracted amount equals zero, but this incurs additional work. Experimental evaluations (Section 4.5) have shown that the simple chunk ordering heuristic, in combination with repeatedly checking if sufficient permission have already been removed, is quite effective in practice and reduces the average verification time by more than one third (Figure 4.6).

Another potential target for optimisations could be the extensionality axiom that defines the equality of two partial snapshot maps (discussed in Section 4.3.2): the axiom admits a quadratic number of instantiations because it can be instantiated for each (ordered) pair of partial snapshot maps. This may be a factor contributing to the performance issues we observed when verifying programs with many function applications (Section 4.3.2), and as future work we plan to investigate the potential for breaking the axiom's symmetry with respect to instantiations by ordering the snapshot maps according to the point at which they were introduced by the symbolic execution.

Lastly, we noticed that repeated executions of qp-remove (Figure 4.2 on page 112) can result in relatively large symbolic permission expressions: in practice, we observed dozens of lines of SMT-LIB code if pretty-printed in a human-readable way, and even a few hundred in the case of appropriately tailored stress tests. We did not observe this to be a problem for the underlying solver, however, but it impedes debugging the generated encoding and Silicon therefore implements a few simple, syntactic rewriting rules that aim at reducing the size of the generated symbolic expressions (and work reasonably well in practice).

Unsupported Features

Silicon's implementation does not yet support predicates or magic wands under quantified permissions, but because of Silicon's uniform representation of the corresponding instances in the heap, extending the existing implementation is expected to be straightforward. The symbolic execution rules presented in this chapter already account for predicates under quantified permissions, and how an integration of magic wands and quantified permissions can be achieved is briefly discussed next (more details are provided in Chapter 5).

Magic wand instances are represented as *magic wand chunks*, which are of the same shape as predicate chunks: a magic wand chunk $wand_{uid}(\overline{v}; s, p)$ represents an instance of a particular magic wand (the latter is uniquely identified by $wand_{uid}$) with "arguments" $\overline{v}$ and snapshot s, and it provides p permissions to the wand instance. Consequently, magic wand instances are inhaled and exhaled similarly to predicate instances, that is, by corresponding produce and consume rules, and magic wand chunks can be lifted to quantified chunks in the same way that predicate chunks are. Implementing this lifting and supporting magic wands under quantified permissions is left as future work.

Generalising the symbolic execution rules for Viper's permission introspection features `perm` and `forperm` also remains future work. After having decided on appropriate syntax, `perm` and `forperm` over fields can be supported as follows (and analogously for predicates):

- Recall from Figure 4.2 that `perm(`x`.`f`)` is evaluated to a symbolic permission sum that represents the total permission amount provided by a symbolic heap: each (non-quantified) heap chunk $f(y_i; \_, p_i)$ contributes a summand of the shape $ite(x = y_i, p_i, 0)$ to the total sum. Correspondingly, each quantified chunk $f(\overline{r}; \_, p_i(\overline{r}))$ needs to contribute a summand of the shape $p_i(x)$ — the equality test between receivers is already included in the symbolic permission expression p_i recorded by quantified chunks.
- Similarly, recall from Figure 4.2 that `forperm[`f`] x ::` $b(x)$ is evaluated by instantiating the body $b(x)$ with all possible receivers to which a given symbolic heap provides permissions: the resulting symbolic value consists of conjuncts of the shape $p_i > 0 \Rightarrow b(y_i)$, one for each (non-quantified) heap chunk $f(y_i; \_, p_i)$. Correspondingly, each quantified chunk $f(\overline{r}; \_, p_i(\overline{r}))$ needs to contribute a conjunct $\forall x \cdot p_i(x) > 0 \Rightarrow b(x)$.

Regarding the state consolidations discussed in Section 3.4.2, two aspects are worth mentioning:

- Recall that state consolidations are used to avoid certain incompletenesses that typically arise from separating the symbolic state into heap chunks and path conditions; in particular, that all (non-quantified) chunks that provide permissions to the same heap location are merged during a state consolidation. This is necessary because the algorithms concerned with non-quantified chunks greedily operate on the first matching chunk only. Quantified chunks do *not* need to be merged since the quantified permission operations presented in this chapter (qp-summarise and qp-remove) always take all quantified chunks into account.
- However, state consolidations also make field receiver disequalities explicit (as path conditions) that are implied by the permission amounts provided by (non-quantified) chunks: for each unordered pair of chunks $f(x; \_, p)$ and $f(y; \_, q)$, the path condition $x = y \Rightarrow p + q \leq 1$ is added. Correspondingly, for each unordered pair of quantified chunks $f(r; \_, p(r))$ and $f(r; \_, q(r))$, the path condition $\forall r_1, r_2 \cdot r_1 = r_2 \Rightarrow p(r_1) + q(r_2) \leq 1$ needs to be added. Silicon does not yet implement the latter.

Recall (Section 4.2.1) the canonical shape of quantified permission assertions used in this thesis, which suffices to specify and verify interesting examples (as demonstrated in the evaluation in Section 4.5):

`forall` x`:` T `::` $c(x)$ `==> acc(`$id(\overline{e(x)})$`,` $p(x)$`)`

As future work, we plan to generalise this canonical shape in two directions: first, by generalising the implication's right-hand side such that it may contain multiple accessibility predicates (for example, `acc(x.f) && acc(x.g)`) and also pure constraints (such as `x.f == x.g`). This generalisation is purely for convenience and can be implemented as a source-to-source translation on the Viper level, thus avoiding the need for adding new symbolic execution rules to Silicon. The second generalisation would be to add support for multiple bound variables (that is, nesting quantified permission assertions on the right-hand side; nesting on the left-hand side is not allowed since the left-hand side must be an expression), which would increase the expressiveness of Viper. This generalisation would entail a corresponding generalisation of the inverse functions introduced by our approach: currently, these map a vector of values (such as predicate arguments) to a single value (corresponding to the single quantified variable bound by a quantified permission assertion); they would have to be generalised to functions from vectors to vectors.

Chapter 5

Magic Wands

Separation logic's *separating conjunction*, denoted by $*$, is the defining connective of separation logic and used to combine assertions such that $a_1 * a_2$ can be understood as the following constraint: the current state σ can be partitioned into two states, $\sigma = \sigma_1 \uplus \sigma_2$, such that σ_1 satisfies a_1 and σ_2 satisfies a_2. Here, $\uplus$ denotes the combination of two compatible partial program states; in particular that the partial states do not disagree on the values of local variables and heap locations.

The separating implication, or *magic wand* $a_1 \mathrel{-\!\!*} a_2$, a connective originally introduced along with the separating conjunction in the first papers on separation logic [64, 99, 108], can instead be understood as a constraint on a *hypothetical extension* of the current state: "if any partial heap satisfying a_1 is added, the resulting state will satisfy a_2". The semantics of the magic wand connective is defined as follows:

$$\sigma \models a_1 \mathrel{-\!\!*} a_2 \;\Leftrightarrow\; \forall \sigma' \perp \sigma \cdot \; (\sigma' \models a_1 \;\Rightarrow\; \sigma \uplus \sigma' \models a_2)$$

Here, $\sigma' \perp \sigma$ expresses that the hypothetical extension σ' is compatible with the current state: per memory location, the combined permission amounts do not exceed write permissions, and the states agree on the values of local variables and heap locations.

The ability to express guarantees about hypothetical (future) additions to the state makes the magic wand well-suited for concisely specifying *partial* versions of data structures, for example, for describing ongoing traversals of data structures [130, 87]. Such situations commonly require specifications (for example, of loops) expressing that the overall data structure (such as a tree) is re-obtained by "plugging back" the traversed and potentially manipulated part (a sub-tree) into the untouched rest of the structure (the surrounding tree). A related use case for magic wands is enabling clients to reason about "recombining" a view on the whole data structure while hiding the internal definitions, which has been used for specifying protocols that enforce orderly modifications of data structures [76, 52, 68]. Yang employs the magic wand for a by-hand proof of the Schorr-Waite graph marking algorithm [136], while Dodds et al. employ it for specifying synchronisation barriers for deterministic parallelism [44].

Despite its history and this variety of applications, the magic wand connective is generally not supported in automated verifiers built upon separation logic (and related theories) [11, 42, 67, 84]; the only exceptions are our verification infrastructure Viper, and VerCors [17] (see also Section 5.8).

The quantification over states in the wand's semantics makes the connective challenging to support in automated tools. Recent developments in *propositional* separation logics [81, 61] show its proof theory to be intricate. In the presence of local variables and fields (selector functions), reasoning without any user guidance is known to be undecidable [29]. We address the problem of magic wand support in the context of Viper: in particular, we support magic wands in combination with arbitrary user-defined predicates and functions; due to a novel approach for automatically choosing suitable *footprints* of magic wands, the required user annotations remain lightweight.

Chapter Overview

The work described in this chapter has in parts been published at ECOOP 2015, in the paper *Lightweight Support for Magic Wands in an Automatic Verifier* by Schwerhoff and Summers [117]. The chapter shows how to support the magic wand connective in an automated verifier, including the following specific contributions:

- A design for the representation of wands in a symbolic verification state, and the provision of suitable *ghost operations* for directing their use (Section 5.2.1).
- An automatic strategy and a corresponding algorithm for choosing suitable *footprints* for magic wand instances, without additional user direction (Section 5.2.2 and Section 5.2.3).
- A mechanism for integrating existing ghost operations (such as folding predicates) with our automatic footprint computation, and a soundness argument for the presented algorithms (Section 5.3).
- A set of additional heuristics, which aim to infer the magic-wand-related annotations required by our approach (Section 5.5).
- An implementation of (most of) the presented techniques in Silicon, along with examples demonstrating the conciseness and versatility of the approach (Section 5.6).
- An integration of magic wands with all other important Viper features (in addition to predicates), in particular with quantified permissions, the framing of heap-dependent expressions such as functions, and permission introspection (Section 5.7).

The remainder of this chapter is structured as follows: a motivating example is shown in Section 5.1 and used to illustrate the use of magic wands in specifications. The representation of magic wands in our technique, related annotations and our automatic footprint computation algorithm are presented in Section 5.2. Next, the integration of existing ghost operations such as `unfolding` is presented in Section 5.3, alongside a soundness argument for the involved algorithms; followed by Section 5.4, which explains how our technique enables framing of heap-dependent expressions. Heuristics that attempt to infer magic-wand-related annotations are discussed in Section 5.5; afterwards, Section 5.6 and Section 5.7 provide an evaluation and a general discussion of the implementation, respectively. The chapter concludes in Section 5.8 with a discussion of the related work.

5.1 Background and Motivation

Listing 5.1 shows a simple Viper program used as the running example of this chapter: a straightforward iterative implementation to calculate the sum of the nodes in a linked list. In this subsection, we give a high-level overview of the concepts involved in the specification and attempted verification of this example.

The precondition of method `sum_it` requires an instance of the usual `list` predicate, and the method's postcondition promises that such an instance will be returned to the caller, along with the guarantee that the returned value is the sum of the values stored in the list.

The verification of the `while` loop (line 23) relies on the provided loop invariant, which (among other properties) specifies that the loop context holds permission to the suffix of the input list starting at `xs`. The loop is straightforward: in each iteration, it unfolds the current `list` predicate instance `list(xs)` — which makes permissions to `xs.val` and `xs.next`, and potentially to `list(xs.next)`, available — before it updates the `sum` and advances the current node pointer `xs`.

```
1  field val: Int
2  field next: Ref
3
4  predicate list(ys: Ref) {
5       acc(ys.val) && acc(ys.next)
6    && (ys.next != null ==> acc(list(ys.next)))
7  }
8
9  function sum_rec(ys: Ref): Int
10   requires acc(list(ys))
11 {
12   unfolding acc(list(ys)) in
13     ys.val + (ys.next == null ? 0 : sum_rec(ys.next)) }
14
15 method sum_it(ys: Ref) returns (sum: Int)
16   requires ys != null && acc(list(ys))
17   ensures  acc(list(ys))
18   ensures  sum == old(sum_rec(ys))
19 {
20   var xs: Ref := ys
21   sum := 0
22
23   while (xs != null)
24     invariant xs != null ==> acc(list(xs))
25     invariant sum ==   old(sum_rec(ys))
26                      - (xs == null ? 0 : sum_rec(xs))
27   {
28     unfold acc(list(xs))
29     sum := sum + xs.val
30     xs := xs.next
31   }
32
33   /* Postcondition error: permissions required by list(ys)
34    *                      are not available */
35 }
```

Listing 5.1: Running example (with insufficient loop invariant): iteratively computing the sum of a linked list.

The loop terminates once the end of the list is reached, at which point `xs` is `null`. The loop invariant does not provide any permission in this situation and the first postcondition (line 17) therefore fails to verify because the predicate instance `list(ys)` is not available: it has been unfolded completely during traversal of the list, but the permissions obtained from each `unfold` were not retained in the loop invariant. These "unfolded" permissions can be retained using a list segment predicate which describes the already traversed list prefix (as illustrated by the examples shown in Section 2.3.1), but the resulting encoding is rather cumbersome since it requires the declaration of an appropriate list segment predicate and the provision and use of (ghost) code that manipulates list segment instances (such as the `concat` lemma method shown in Section 2.3.1 which appends to the end of a list segment). In the next subsection we will instead describe how magic wands can be used to specify the loop without having to introduce additional specification-only predicates and methods.

Overview of our Magic Wand Support

Listing 5.2 shows the body of the `sum_it` method, specified using Viper's magic wand support (the full example is shown in Listing D.1 in Appendix D). The loop invariant has been strengthened (line 11) to include an additional *magic wand instance* (`xs != null ==> acc(list(xs)) --* acc(list(ys))` (as usual, the use of macros is optional but makes the code more readable).

```
1  var xs: Ref := ys
2  sum := 0
3
4  define A xs != null ==> acc(list(xs))
5  define B acc(list(ys))
6
7  package A --* B
8
9  while (xs != null)
10   invariant xs != null ==> acc(list(xs))
11   invariant A --* B
12   invariant sum ==   old(sum_rec(ys))
13                    - (xs == null ? 0 : sum_rec(xs))
14 {
15   wand w := A --* B /* Give magic wand instance the name w */
16
17   var zs: Ref := xs /* Value of xs at start of iteration */
18   unfold acc(list(xs))
19   sum := sum + xs.val
20   xs := xs.next
21
22   package A --* folding acc(list(zs)) in applying w in B
23 }
24
25 apply A --* B
```

Listing 5.2: The verified version of the body of `sum_it`, from Listing 5.1.

Informally, this magic wand instance represents the following promise: "if you give up permission to the remainder of the list (starting at `xs`), you will in exchange be given permission to the entire list structure (starting at `ys`)". This assertion plays the role of representing the permissions to the partial list inspected by the loop so far; we say these permissions make up the *footprint* of the magic wand.

The footprint of a magic wand must include enough permissions to make this informal promise justified. In order to ensure that the promise remains justified until it is used, the footprint is removed from the current symbolic heap when a new magic wand instance is created, which effectively renders the corresponding heap locations immutable until the wand instance is used. More details follow in Section 5.2.1.

The verifier can be directed to create a new magic wand instance (and to choose a suitable footprint) using a `package` statement, such as that used on line 7 of Listing 5.2, which creates the wand instance necessary for showing that the loop invariant holds on entry. There, an empty footprint suffices since `xs` and `ys` are equal at this point (due to line 1).

Figure 5.1 conceptually illustrates the permissions that the magic wand instance in our loop invariant represents, by stepping through the important stages of verifying the loop body (for simplicity, the cases of `xs`/`xs.next` being `null` are ignored in the illustration). At the beginning of the loop body, the magic wand's footprint includes the permissions (to fields `val` and `next`) from the head of the linked list `ys` all the way down to — but excluding — the current node `xs`. The remaining permissions, that is, those to the current node and the tail of the list, are contained in the predicate instance `list(xs)`. The latter is then unfolded, providing permissions to the fields of `xs` (that is, to `acc(xs.val)` and `acc(xs.next)`) and `xs` is afterwards advanced such that it points to the next node (that is, to `zs.next`). In order to re-establish the loop invariant, in particular, to re-establish that the wand instance includes the permissions to the already-visited prefix of the list, it is necessary to add the permissions to `zs.val` and `zs.next` to the wand instance. This is achieved by the final `package` statement: the ghost operations on the right-hand side of the wand force the wand's footprint to

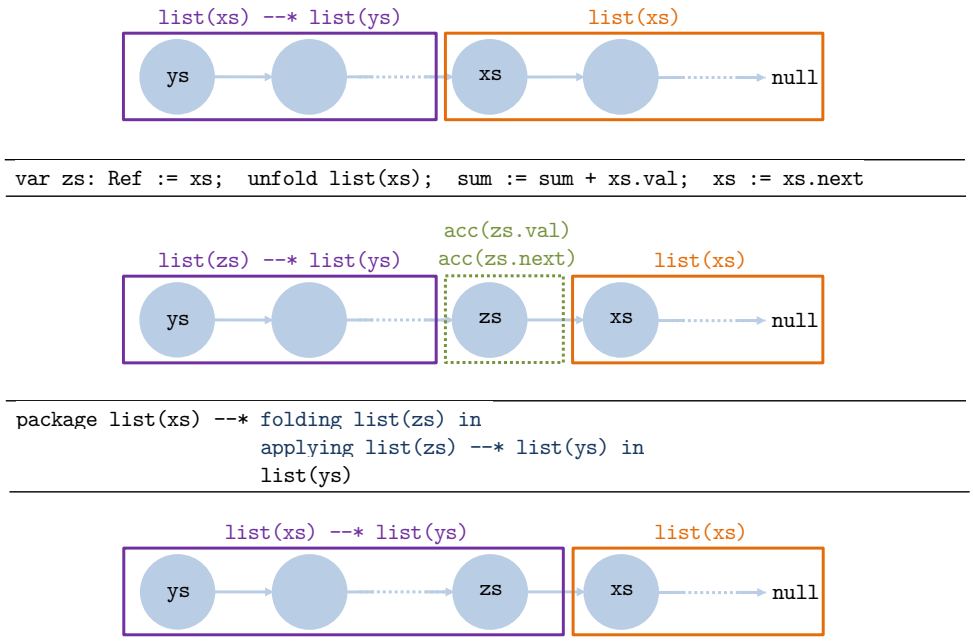

Figure 5.1: Illustration of the bookkeeping of permissions in the loop invariant from Listing 5.2, via magic wand and predicate instances. The magic wand instances cover permissions to the prefix of the list (starting at `ys`) that has already been traversed by the loop. For simplicity, the cases of `xs`/`xs.next` being `null` are ignored.

include the footprint of the wand held so far, plus the permissions to `zs.val` and `zs.next` (necessary for the `folding` ghost operation on line 22 of Listing 5.2).

The role of these ghost operations is explaining to the verifier *how*, given the left-hand-side assertion, the right-hand-side assertion can be obtained[1]; their role is thus comparable to that of other ghost operations such as `unfolding`. Further details about such ghost operations are provided in Section 5.3. Given these operations, the computation of the wand instance's footprint, that is, of the extra permissions which must be associated with the new instance, is performed automatically; the automated footprint computation is an important contribution of this chapter.

A wand instance, together with its left-hand-side assertion, can be exchanged for the right-hand-side assertion; this is called *applying* the magic wand instance. For example, after the loop body in Listing 5.2, the magic wand instance from the loop invariant is applied (line 25): the instance and its left-hand side must be given up, and its right-hand side `list(ys)` is added to the verification state, providing the permission required by the method's postcondition.

The support for magic wands presented in this chapter allows a natural specification of such "left-over" parts of data structures, in a way which requires few annotations, and applies equally well to other data structures and predicates. This is an important use case of magic wands, but (as previously discussed) not the only one (e.g. [76, 52, 68, 136, 44]); the possibility of practical tool support via the contributions of this chapter will likely also lead to further applications being explored.

5.2 Magic Wand Support with Automatic Footprints

In this chapter, our technique for supporting the magic wand connective in an automated verifier is presented in the context of symbolic execution, and in particular

[1] Variable `zs` records the node that `xs` pointed to at the beginning of the current loop iteration, while `w` gives a name to the magic wand instance belonging to the loop invariant at the start of the iteration. Both are not strictly necessary, but make the annotations on line 22 succinct.

in the context of the symbolic execution rules presented in previous chapters. The technique is more general, however: in [117] we present our technique independent of any particular implementation strategy for the underlying verification tool, and we have already implemented the technique in both Viper verifiers (symbolic-execution-based Silicon and verification-condition-generation-based Carbon; recall Figure 2.1 and Section 3.7.1). Moreover, our approach is presented in the context of implicit dynamic frames but does not depend on any special feature of this logic (such as heap-dependent expressions): it is therefore straightforward to adapt it to a separation-logic-based tool or to other permission logics.

5.2.1 Representing Wand Instances as Opaque Resources

Recall (from the beginning of this chapter) that reasoning about magic wands is in general undecidable if no user guidance is provided. In order to devise an approach that is based on user guidance but still reasonably lightweight and that can be efficiently implemented, we took inspiration from the handling of (recursive) predicates: just as for predicates, our approach requires instructing the verifier to create new magic wand instances (`package`) and to apply their meaning while verifying code (`apply`). In between packaging and applying a wand instance, the instance is treated — analogous to a predicate instance — as an *opaque* resource recorded in the verification state: when one is available, the verifier need not attempt to deduce anything that follows from the wand's semantics, without instruction to do so.

The choice to use such an opaque magic wand instance must be directed by a ghost statement `apply` a_1 `--*` a_2 (see, for example, line 25 of Listing 5.2). Recall the formal semantics of a magic wand assertion from the beginning of this chapter (note: change of variable names):

$$\sigma_{foot} \models a_1 \mathrel{-\!\!*} a_2 \;\Leftrightarrow\; \forall \sigma_{lhs} \perp \sigma_{foot} \cdot (\sigma_{lhs} \models a_1 \Rightarrow \sigma_{lhs} \uplus \sigma_{foot} \models a_2)$$

This semantics intuitively says that $a_1 \mathrel{-\!\!*} a_2$ is true in a state σ_{foot} if it is guaranteed that any state created by combining this state with some additional state σ_{lhs} satisfying a_1, satisfies a_2. One can see this as a definition in terms of what can be *deduced* from a magic wand, according to the following *Modus-Ponens*-like inference rule from separation logic: $a_1 * (a_1 \mathrel{-\!\!*} a_2) \models a_2$. The operation of applying a magic wand instance is defined analogously (shown in Figure 5.5); intuitively represented by the following sequence of Viper statements:

`apply` a_1 `--*` a_2 can be understood as **`exhale`** a_1 `&&` `(`a_1 `--*` a_2`)` **`inhale`** a_2

Just as for predicate instances, the opaque treatment of magic wand instances requires for soundness that the state σ_{foot} in the semantics above must notionally *belong* to the magic wand instance, in the sense that the program is not allowed to modify that part of the state until the wand instance is applied. We call such a state the *footprint* of the magic wand instance. Whenever a new magic wand instance is to be added to the verification state, it is therefore necessary to compute some suitable part σ_{foot} of the current state σ, that will suffice to guarantee the wand's semantics, and then remove σ_{foot} from the current state, and in its place add the new magic wand instance. We call this operation (of choosing a suitable footprint for a magic wand instance and exchanging the footprint for the wand instance) *packaging* the magic wand instance, and use a ghost command `package` a_1 `--*` a_2 to direct the verifier to perform a corresponding package operation.

Given a suitable choice of footprint (discussed next in Section 5.2.2), the operation of packaging a magic wand can intuitively be understood as the following sequence of Viper statements, where a_{foot} is an assertion describing a suitable footprint of the

magic wand instance to package:

package a_1 `--*` a_2 can be understood as **exhale** a_{foot} **inhale** a_1 `--*` a_2

5.2.2 Strategy for Choosing Footprints

The representation of magic wands as opaque resources in the verification state, and their manipulation via lightweight user annotations, requires automatically choosing a suitable footprint for a magic wand instance, which is challenging. As outlined in the previous subsection, a package operation `package` a_1 `--*` a_2 must attempt to choose a footprint σ_{foot}, which can be any portion of the current state as long as it satisfies the wand's semantics. In checking this criterion, it would be unsound to use any facts from the current state which are *not* framed by permissions that were chosen for the footprint, since these facts might no longer be true by the time the wand instance is applied[2]. For example, when packaging `acc(x.f) --* acc(x.f) && x.f == 3` in a state where `x.f == 3`, this fact may be used only if permission to `x.f` is included in the wand instance's footprint: otherwise, the value of `x.f` could be changed by the time the wand instance is applied.

Choosing a suitable footprint for a magic wand instance is complicated by the quantification over *hypothetical* states σ_{lhs} in the wand's semantics: (1) the wand's left-hand side a_1 is to be evaluated in such a σ_{lhs} state (that is, not in the current state), and moreover (2) depending on that state, different choices of the footprint state σ_{foot} may be possible such that the wand's right-hand side a_2 is satisfied by the combination of σ_{foot} and σ_{lhs}. To illustrate this difficulty, consider the magic wand `(acc(x.b) && x.b ==> acc(x.f)) --* acc(x.f)`: its left-hand side would evaluate to true in a hypothetical state with permission to `x.b` and `x.f` if `x.b` were true, in which case the empty footprint would suffice to satisfy the wand's semantics; as would the combination of a hypothetical state with permission to `x.b` only and a footprint which provides the necessary permission to `x.f`, if `x.b` were false.

In deciding on a strategy for choosing footprints, the choice of a wand instance's footprint could soundly be restricted to *any* portion of the current state if a subsequent check ensures that the choice suffices to guarantee the wand's semantics. Certain strategies for choosing a footprint are, however, more useful than others.

For example, one *could* always choose the empty state as a footprint σ_{foot} and therefore not use up any permissions at a `package` statement; the check of the wand's semantics would then fail in all cases but those where a_1 entails a_2. Alternatively, one *could* always choose σ_{foot} to be the *entire* current state. This would allow many wands to be proven, but the subsequent verification will almost certainly fail due to missing permissions. Although either of these approaches would be sound, they would not be useful in practice.

Intuitively, it makes sense to choose a footprint which is as small as possible, while still guaranteeing enough information for the wand's semantics. However, the notion of "as small as possible" is not straightforward to define precisely. For example, the semantics of a wand $a_1 \mathrel{-\!\!*} a_2$ can be satisfied by choosing a footprint state which includes enough permissions such that a state satisfying a_1 can never again be obtained: this would yield a true wand instance — the inability to find a σ_{lhs} state that is compatible with this choice of footprint makes the semantics of the wand vacuously true — but one which could never be applied, which is not useful as a verification construct.

Recall the previous example, in which `acc(x.f) --* acc(x.f) && x.f == 3` is to be packaged in a state in which `x.f == 3`. As discussed, this fact may only be

[2]We only allow magic wand assertions a_1 `--*` a_2 in which both assertions a_1 and a_2 are *self-framing* (recall Section 2.2.1). This is not a strong restriction in practice; indeed, in standard separation logics, all assertions are self-framing [102].

soundly used when proving the right-hand side of the wand if permission to `x.f` goes into the wand's footprint (and is thus removed from the current state). Although this extra logical fact is useful in proving the right-hand side, such a decision would again yield a wand instance which cannot be applied: since the left-hand side of the wand requires this permission to be provided when applying the wand instance. Essentially, any permissions which are taken from the current state although they are already provided by the left-hand side of the wand, are *leaked* at the point of packaging an instance of that wand, which is typically not useful for verifying the rest of the program.

Motivated by these observations, our strategy for choosing wand footprints is: *include all permissions required by the wand's right-hand side which cannot be proven to be provided by the wand's left-hand side*. Note that restricting the choice of footprint to *only* these permissions is not really a restriction in practice: if the tool user *intends* to include extra permissions from the current state in a wand's footprint, they can achieve it by writing a right-hand side which requires more permissions than the left-hand side provides.

For example, our strategy does not allow packaging the (already shown) wand

```
acc(x.f) --* acc(x.f) && x.f == 3
```

in a state such that `x.f == 3`: permission to `x.f` are provided by the left-hand side and thus not taken from the current state; consequently, asserting the right-hand side in the combination of an empty footprint with the left-hand side will fail. The user can remedy this situation in two ways, however: by "forcing" the footprint to include some permission to `x.f`, or by strengthening the left-hand side. Changing the wand to

```
acc(x.f, 1/2) --* acc(x.f) && x.f == 3
```

means that $1/2$ permission to `x.f` are included in the wand, which effectively renders the location immutable and thus allows using the fact `x.f == 3` when proving the right-hand side. Alternatively, the wand's left-hand side can be strengthened:

```
acc(x.f) && x.f == 3 --* acc(x.f) && x.f == 3
```

Permission to `x.f` remain in the current state (and the location mutable), but the wand can only be applied in states where `x.f == 3`.

5.2.3 Automated Footprint Computation

The idea of the previously described strategy for choosing a wand instance's footprint is simple, but devising an algorithm that automates the footprint computation is still challenging: there is a technical circularity to the problem. The footprint for a wand is determined in terms of the permissions required by its right-hand side. Exactly which permissions are required by the right-hand side can (due to conditionals) depend on properties of heap values. Properties known about heap values in the *current* state may be soundly used if and only if permissions to those heap locations are included in the wand's footprint — whose computation is ongoing.

To break this circularity, we devised an algorithm that *simultaneously* checks the right-hand side of the wand to be packaged, which includes determining the required permissions and constructs a new state σ_{used} which contains these permissions. The required permissions are taken from the current state if they cannot be proved to be provided by the wand's left-hand side; thus, the algorithm implicitly carves out a suitable footprint for the wand from the current state.

```
1  exec(σ1, package  a1 --* a2, Q) =
2    produce(σ1{h := ∅}, a1, fresh, (λ σlhs ·
3      consume-ext([σlhs.h, σ1.h], σlhs{h := ∅}, a2, (λ [_, h2], _, _ ·
4        Let ch be a magic wand chunk corresponding to a1 --* a2
5        Q(σ1{h := heap-add(h2, ch)})))))
```

Figure 5.2: Simplified rule for packaging magic wand instances. Potentially branching executions (of produce and consume-ext) are not accounted for, and neither is the framing of heap locations to which all permission are lost to the wand instance's footprint. The rule is gradually refined in Section 5.4.1 and Section 5.4.2 to account for framing and branching executions, respectively.

Packaging Magic Wands

A simplified algorithm is shown in Figure 5.2 (with additional definitions such as that of consume-ext shown in Figure 5.3), formalised as a symbolic execution rule. The simplified rule focuses on the handling of permissions, and does neither account for framing heap values across `package-apply` pairs (similar to framing across `fold-unfold` statements, as discussed in Chapter 3), nor for magic wands that include conditionals (and can result in branching executions of `package/apply` statements). Both can result in incompletenesses; how to avoid them is discussed in Section 5.4.1 (framing) and Section 5.4.2 (branching). In both subsections, the rules for executing `package` and `apply` statements are gradually extended.

Given a current state and a wand instance to package, the rule proceeds as follows: it first creates an *arbitrary* hypothetical state σ_{lhs} satisfying the wand's left-hand side a_1 (by producing a_1 into an empty heap). The next step combines removing a suitable footprint from the current state *and* checking the right-hand side in the combination of the footprint and the left-hand-side state: this is done by consume-ext (defined in Figure 5.3), which takes permissions from the current heap $\sigma_1.h$ only if they are not provided by the left-hand-side heap $\sigma_{lhs}.h$. Finally, a representation of the packaged wand instance (as a *magic wand chunk*, which are similar to predicate chunks and defined in Section 5.2.5) is added to the remainder of the (initial) current heap h_2 and the verification continues: the footprint has been consumed and replaced by the packaged wand instance.

Note that on line 5, heap-add is used as if it only returned an updated heap, whereas the signature given in Section 3.1.2 allows heap-add to also return potentially updated path conditions: for example, the refined definition of heap-add from Section 3.4.2 may add equalities between snapshots (obtained from merging chunks). To simplify the presentation, we omit the potentially updated path conditions throughout this chapter (which in practice would result in incompletenesses, as discussed in Section 3.4.2, and is *not* done in Silicon's implementation).

Keeping the initial heap $\sigma_1.h$ and the hypothetical extra heap $\sigma_{lhs}.h$ separate is essential for a correct checking of the right-hand side. A naïve implementation which simply combined both heaps before checking the wand's right-hand side (and determining a suitable footprint) would be unsound: the combination might be inconsistent, which would unsoundly trivialise the check. For example, the operations

```
inhale acc(x.f)
package acc(x.f) --* false
```

and

```
inhale acc(x.f, 1/3) && x.f == 1
package acc(x.f, 1/3) && x.f == 2 --* false
```

should fail (in consistent states), but might unsoundly succeed if the current and the left-hand-side state *were* combined: in the first case because of too much permission, in the second because of contradicting value facts.

Assumptions (path conditions) obtained from producing the hypothetical left-hand side must be available when checking the right-hand side, for example, to allow packaging the vacuous wand `false --* false`, but they must be retracted once the package operation finished: asserting `false` should fail after this vacuous wand has been packaged. This is achieved in Figure 5.2 by making the path conditions obtained from producing the left-hand side available to the subsequent check of the right-hand side (done by consume-ext, which is discussed shortly). After the right-hand side has been checked, however, the verification continues (on line 5) with the (initial) current path conditions, and thus discards the path conditions obtained from the hypothetical left-hand side.

Removing Footprints and Checking Right-Hand Sides

Recall that the strategy for automatically choosing wand footprints is to take permissions required by the right-hand side from the left-hand-side heap if possible, and only otherwise from the current heap. This is implemented as an extended version of the consume operation, called consume-ext (defined in Figure 5.3), and used by the previously discussed rule for packaging wands (Figure 5.2).

This operation achieves two goals: it computes a footprint and removes it from the current heap, and it simultaneously checks that the right-hand side holds (in the combination of the footprint and the left-hand side). To achieve its goals, the operation attempts to successively construct a state (denoted by σ_{used}) that satisfies the right-hand side, by transferring permissions from the left-hand-side heap whenever possible, and only otherwise from the current heap. The footprint of a wand instance is thus only computed implicitly and the constructed state (which satisfies the right-hand side if consume-ext succeeds) is *not* the footprint itself: it is a combination of the footprint and (a part of) the left-hand side heap.

consume-ext takes the following three inputs (in addition to the usual continuation):

(1) A stack $\overline{h}$ of symbolic heaps, which (for now) is the left-hand-side heap on top of the current heap, expressing that permissions are preferentially to be taken from the left-hand-side heap. Stacks of greater height are necessary in order to package nested wands, which in turn requires nesting package operations and thus multiple left-hand-side heaps; this is explained in Section 5.3.

(2) A state σ_{used}, typically empty on the initial invocation of consume-ext (for example, when used by `package`). As described, σ_{used} is successively populated by transferring permission from the stack of heaps (so that it eventually is the combination of footprint and left-hand side), and it is used to check the wand's right-hand side.

(3) An assertion a, which is the wand's right-hand side (for example, when consume-ext is used by `package`) or a suffix thereof. This assertion determines σ_{used} — permissions required by a are transferred into σ_{used} — and is itself checked in σ_{used}.

If consume-ext succeeds, it returns (passes to its continuation) the stack of remainder heaps not transferred into σ_{used}, σ_{used} itself (which by-construction satisfies a), and a snapshot that (as usual) represents the values of the heap locations to which permissions were consumed (transferred). Note that the simplified `package` rule presented in Figure 5.2 ignores the snapshot: its discussion is postponed until Section 5.4.1.

consume-ext recurses over the assertion a (that is, the wand's right-hand side), and maintains the following invariant: at any point *during* the execution of consume-ext, the current $\sigma_{used}.h$ satisfies the prefix of the initial a that has been processed so far.

```
1  consume-ext: List[H] → Σ → A → (List[H] → Σ → Snap → R) → R
2  consume-ext(h̄, σ_used, acc(id(ē), p), Q) =
3    eval(σ_used, p :: ē, (λ σ_used_1, p′ :: ē′ ·
4      transfer(h̄, σ_used_1.h, σ_used_1.π, id(ē′), p′) matches
5        Some(h̄_1, h_used_2, s):
6          Q(h̄_1, σ_used_1{h := h_used_2}, s)
7        None:
8          failure()))
9
10 consume-ext(h̄, σ_used, e, Q) =
11   eval(σ_used, e, (λ σ_used_1, e′ ·
12     assert(σ_used_1.π, v) ∧
13       Q(h̄, σ_used_1, unit)
14
15 consume-ext(h̄, σ_used, a_1 && a_2, Q) =
16   consume-ext(h̄, σ_used, a_1, (λ h̄_1, σ_used_1, s_1 ·
17     consume-ext(h̄_1, σ_used_1, a_2, (λ h̄_2, σ_used_2, s_2 ·
18       Q(h̄_2, σ_used_2, pair(s_1, s_2))))))
```

Figure 5.3: consume-ext transfers the footprint of a from $\overline{h}$ into σ_{used}. Other cases of consume-ext (for example, for conditionals) are analogous to consume (as illustrated by the rule for separating conjunction), but such that expressions are evaluated in σ_{used}. *Some*$[T]$ and *None* are of type *Option* $[T]$, corresponding to the homonymous Scala type (and to Haskell's *Maybe*). Operation transfer returns such an *Option*; keyword matches allows pattern matching against the return value.

Recall that both sides of a wand have to be self-framing; on a wand's right-hand side the permission to access a heap location thus occurs before any expressions that depend on the location's value. It is therefore guaranteed that consume-ext can (attempt to) transfer permission into σ_{used} before it needs to check any expression (in σ_{used}) that potentially accesses a corresponding heap location.

For most assertions, operation consume-ext is defined analogously to consume (recall Section 3.3), as illustrated by the definition of consume-ext for a_1 `&&` a_2. The only interesting cases are those for permissions (case `acc(`$id(\overline{e})$`, p)`) and pure assertions (case e): consuming permissions transfers permissions from the stack of heaps into σ_{used} (or fails, if permissions cannot be found); consuming a pure assertion entails evaluating and asserting it in σ_{used}. When consuming an assertion such as `acc(x.f) && x.f > 0`, permissions to `x.f` are first transferred into σ_{used}, and the pure assertion `x.f > 0` can thus afterwards be (evaluated and) checked in σ_{used} (recall that both sides of a magic wand are required to be self-framing).

Transferring permissions is implemented by the transfer operation, which traverses the stack of heaps top-down and greedily transfers as many permissions as possible (and still needed) from each heap into h_{used}. As results, transfer returns the remainder heaps, h_{used} extended with the transferred permissions, and a snapshot that corresponds to the value of the location to which permission were transferred. transfer therefore corresponds to the core of our strategy of only taking permissions from the current state if absolutely necessary. Note that the presented implementation of transfer is slightly simplified because it does not equate snapshots in cases were permissions are taken from multiple heaps (denoted by s_1 and s_2 in the corresponding rule), which could result in incompletenesses. Silicon's implementation, however, adds appropriate equalities to the path conditions.

```
1  transfer: List[H] → H → Π → Id → Perm → Option[(List[H], H, Snap)]
2  transfer(h :: h̄, h_used, π, id(v̄), p) =
3    heap-rem-max(h, π, id(v̄), p) matches
4      Some(h_1, s_1, p_1):
5        h_used_1 := heap-add(h_used, id(v̄), s_1, p − p_1)
6        if check(π, p_1 = 0) then
7          Some(h_1 :: h̄, h_used_1, s_1)
8        else
9          transfer(h̄, h_used_1, π, id(v̄), p_1) matches
10           Some(h̄_rem, h_used_2, s_2): Some(h_1 :: h̄_rem, h_used_2, s_2)
11           None: None
12     None:
13       transfer(h̄, h_used, π, id(v̄), p) matches
14         Some(h̄_rem, h_used_1, s): Some(h :: h̄_rem, h_used_1, s)
15         None: None
16
17 transfer([], h_used, π, id(v̄), p) = None
18
19 heap-rem-max: H → Π → Id → Perm → Option[(H, Snap, Perm)]
20 heap-rem-max(h, π, id(v̄), p) =
21   if (∃ id(w̄; s, q) ∈ h) · check(π, ⋀ v̄ = w̄) ∧ 0 < q) then
22     h_1 := h \ {id(w̄; s, q)} ∪ {id(w̄; s, q − min(p, q))}
23     Some(h_1, s, p − min(p, q))
24   else
25     None
```

Figure 5.4: $\overline{h}$ denotes a (potentially empty) stack of heaps, $h :: \overline{h}$ places h on top of $\overline{h}$, [] denotes the empty stack. transfer descends a stack of heaps and tries to transfer sufficient permissions to h_{used}. Per heap, heap-rem-max removes as many permissions as possible (and still needed).

The transfer operation in turn employs heap-rem-max to remove permissions from individual heaps. The implementation of heap-rem-max, shown in Figure 5.4, corresponds to the permission removal operation heap-rem used in Chapter 3 (defined in Section 3.3), except that heap-rem-max returns the permission amount that still needs to be removed (from another heap) instead of immediately failing if the heap does not provide sufficient permission. This simple implementation of heap-rem-max gives rise to the incompletenesses discussed in detail in Chapter 3, for example, in the presence of disjunctive aliasing (recall in particular Section 3.4.2 and Section 3.7.3). However, the choice of implementation is *orthogonal* to our technique for supporting magic wands and computing wand footprints: more involved implementations such as the permission removal operation used in the context of quantified permissions (Chapter 4), which is complete with respect to disjunctive aliasing, could be plugged-in instead.

In order to illustrate how the packaging of a magic wand instance proceeds, consider the following example:

```
inhale acc(x.f) && acc(x.g) && x.f == 2 && x.g == 1
package acc(x.f) && x.f == 1
          --*
        acc(x.f) && acc(x.g) && x.f == x.g /* Succeeds */
assert x.f == 1 /* Fails (correctly) */
```

The initially constructed left-hand-side state (line 2 of Figure 5.2) contains permission to `x.f` and the information that `x.f` has the value `1` (in the left-hand-side heap). In the next step (line 3), consume-ext is invoked in order to compute a footprint for the right-hand side and remove it from the current state, and to check that the

footprint and the left-hand side satisfy the right-hand side. The left-hand-side heap ($\sigma_{lhs}.h$) corresponds to `acc(x.f)`, whereas the current heap ($\sigma_1.h$) corresponds to `acc(x.f) && acc(x.g)`. The second argument of consume-ext ($\sigma_{lhs}\{h := \emptyset\}$) is the initial σ_{used} state into which permissions are transferred: its heap is empty, but it contains the path conditions from the current and the left-hand-side state. consume-ext recurses over the wand's right-hand side and first transfers permission to `x.f` and `x.g` from the left-hand-side heap, respectively, the current heap into σ_{used}: both transfers are done by transfer, which has to take permission to `x.g` from the current heap because it cannot take them from the left-hand-side heap. Due to the permission transfer (that is, the transfer of the corresponding heap chunks), the facts `x.g == 1` (from the current heap) and `x.f == 1` (from the left-hand-side heap) are available in σ_{used}. Hence, when consume-ext finally checks the constraint `x.f == x.g` (in σ_{used}), permission to both locations are available and the equality can be proven. After consume-ext removed the footprint from the current state and checked the wand's right-hand side (line 3 of Figure 5.2), the remainder of the left-hand-side heap is discarded[3], whereas the remainder of the current heap (h_2) is used for the remaining verification, after the packaged wand instance has been added (line 5). The final `assert` in the above example thus fails as expected: `x.f` still has the value 2 (in h_2) and the hypothetical fact `x.f == 1` (from the wand's left-hand side) is no longer available.

5.2.4 Applying Magic Wands

Recall (Section 5.2.1) that applying a magic wand a_1 `--*` a_2 can be understood as exhaling a_1 `&&` (`(`a_1 `--*` a_2`)`), followed by inhaling a_2. This intuitive idea is formalised in Figure 5.5, which shows a simplified version of the rule for applying magic wands. Analogous to the simplified rule for packaging wands (Figure 5.2), the rule for applying wands does not yet account for branching executions and framing values (via snapshots). The rule is straightforward: it first consumes the left-hand side a_1 and a magic wand chunk that corresponds to the wand instance to apply (details about magic wand chunks are given in Section 5.2.5), and afterwards it produces the right-hand side a_2 (and continues the verification).

Effectively, applying a wand instance exchanges the instance and the left-hand side for the right-hand side. This exchange is justified by the previously described package operation: it ensures that *any* left-hand side combined with the instance's footprint satisfies the right-hand side and it removed the footprint, which notionally belongs to the wand instance and is regained as part of the obtained right-hand side when the instance is applied.

```
1  exec(σ1, apply  a1 --* a2, Q) =
2    Let id_wand(ē') be a magic wand chunk identifier
3        corresponding to a1 --* a2
4    consume(σ1, a1, (λ σ2, _ ·
5      consume(σ2, acc(id_wand(ē)), (λ σ3, _ ·
6        produce(σ3, a2, fresh, Q)))))
```

Figure 5.5: Simplified rule for applying magic wand instances, implementing the *Modus-Ponens*-like rule for wands. The rule does not yet account for framing and for branching executions, the respective refinements are made in Section 5.4.1 and Section 5.4.2.

So far, we discussed (simplified versions of) the operations for packaging and applying magic wands, and in particular how the package operation implements our strategy for choosing suitable footprints. In the next section we provide details about the representation of magic wand instances in the symbolic state (as magic wand chunks).

[3]It would be possible to warn users that a wand's left-hand side requires more permissions than necessary (which are effectively leaked when the wand is applied), but this is currently not done.

Afterwards, we extend the packaging operation to account for ghost operations (such as `(un)folding`) which may be required when packaging a wand in order to guide the verifier during the proof of the right-hand side.

5.2.5 Magic Wand Chunks

In the previous presentation of the rules for packaging and applying wands (for example, Figure 5.2) it was stated that there is a way of representing magic wand instances in the symbolic heap such that instances can be added to and removed from symbolic heaps: this subsection provides the previously omitted details.

Requirements

Our representation of magic wand instances is motivated by two requirements: (1) it must be possible to look up wand instances in *different* states, and (2) looking up wand instances must be implementable *efficiently*. To illustrate this difficulty, consider the magic wand `(acc(x.b) && x.b ==> acc(x.f)) --* acc(x.f)`: its left-hand side would evaluate to true in a hypothetical state with permission to `x.b` and `x.f` if `x.b` were true, in which case the empty footprint would suffice to satisfy the wand's semantics; as would the combination of a hypothetical state with permission to `x.b` only and a footprint which provides the necessary permission to `x.f`, if `x.b` were false.

To illustrate the first requirement, consider the following example:

```
n := m - 1
inhale acc(x.f) && x.f > n + 1 --* acc(x.f) && x.f > m
// Should fail
assert acc(y.f) && y.f > n + 1 --* acc(y.f) && y.f > m
y := x; n := n + 1
// Should hold
assert acc(y.f) && y.f > n     --* acc(y.f) && y.f > m
```

All wands are specific to the (at the respective points) *current* values of the occurring local variables `x`, `y`, `n` and `m`: the first `assert` should fail because `y` might not be an alias of `x`, whereas the second `assert` should succeed (due to the preceding assignments to `y` and `n`).

The illustrated property does not only hold for local variables: sub-assertions occurring in a magic wand can in general be partitioned into those whose values are completely independent of the hypothetical left-hand-side state and those whose values are (at least in part) determined by the state in which the wand will eventually be applied in. The first partition includes heap-independent expressions (such as `x`, `n` and `n + 1` from above), but also `old` expressions: these may be heap-dependent, but the dependency is on a known heap and not on the hypothetical one.

This observation constitutes the first step towards our representation of magic wand instances in the symbolic heap: it allows substituting assertions whose value is determined by the current (or any other known) state for their symbolic values, and comparing these values when looking up a specific wand instance.

Handling the remaining assertions in a similar way is complicated by their dependency on the hypothetical left-hand-side state, and in general involves (potentially undecidable) entailment checks between wand instances. To account for the second requirement from above — the representation of magic wand instances should facilitate efficient instance lookup — we instead compare the remaining assertions purely syntactically, and represent magic wand instances as parametric "structural skeletons with holes": the holes correspond to assertions whose values are independent of the hypothetical state, and the structural skeleton is given by the remaining assertions.

To illustrate this representation, consider the magic wand

```
acc(x.f) && x.f > n + 1 --* acc(x.f) && x.f > m
```

that is inhaled at the beginning of the previous example, whose representation in the symbolic state can be understood as follows:

$$\big(\texttt{acc(\_\_.f) \&\& \_\_.f > \_\_ --* acc(\_\_.f) \&\& \_\_.f > \_\_}\big)\big(x, x, n+1, x, x, m\big)$$

That is, as the application of symbolic values to the structural skeleton with holes (the arguments take the positions of the holes from left to right).

Based on this representation, we define two wand instances to *match* if their skeletons match structurally (syntactically) and if their arguments are pairwise (semantically) equivalent. Only the latter involves queries to the underlying solver, and these are usually of limited complexity (in particular when compared to proper entailment checks between wand instances). With this representation of wand instances and the corresponding definition of when instances match, the first `assert` statement from the example above indeed fails (as expected), whereas the second one (correctly) verifies.

Since instance lookup is partially based on a syntactic comparison, its efficiency comes at the cost of completeness, as illustrated by the following snippet:

```
inhale true --* acc(x.f) && acc(y.f)
assert true --* acc(y.f) && acc(x.f) // Fails (incompleteness)
```

However, due to the simplicity of the representation and the lookup approach, it is straightforward to explain (and to understand) why and when such incompletenesses can arise (and in most cases also how to overcome them). It is also possible to relate such incompletenesses to similar incompletenesses that arise in the context of predicates, instances of which are also treated as an opaque entity and matched by combining syntactic and semantic equality:

```
predicate P(x: Ref, y: Ref) { acc(x.f) && acc(y.f) }

inhale acc(P(x, y))
exhale acc(P(y, x)) // Fails (incompleteness)
```

While of course not ideal, this consistency might facilitate explaining and understanding such potentially arising incompletenesses.

Magic Wand Chunks

Based on the discussed ideas and definitions, we represent magic wand instances in symbolic heaps as *magic wand chunks* of the shape $id(\overline{v}; \_)$, where id uniquely denotes a specific wand skeleton, and where $\overline{v}$ are the instance arguments, that is, the symbolic values that replaced the evaluated sub-assertions. Since a given Viper program only contains a statically known set of magic wand skeletons (obtained by replacing appropriate sub-assertions by holes), it is always possible to represent each such skeleton by a unique id. Wand chunks also include a snapshot (here omitted) which is used to preserve the values of locations to which (all) permissions were (temporarily) "lost" to a magic wand instance. More details about wand chunk snapshots are provided in Section 5.4.1.

We do not support fractional wand instances (and wand chunks thus do not record a permission amount, which is effectively fixed to 1): the certainty that each wand instance can be applied only once simplifies the handling of snapshots and path conditions (more details are given in Section 5.4.1), and we so far did not come across the need for fractional instances. Our approach can be extended to support fractional instances, however, as is discussed in Section 5.7.

Note that this representation of magic wand instances (as magic wand chunks) is analogous to the representation of predicate instances; this enables a direct integration of wand chunks with previously presented algorithms for manipulating heaps (such as the operations consume-ext, transfer and heap-rem-max, but also the symbolic execution rules discussed in Chapter 3), both in terms of permissions and snapshots. As a consequence, no additional measures need to be taken to, for example, support magic wands inside predicate bodies or to integrate magic wands with Viper's permission introspection features (Section 3.4.4). For the same reason, it is also straightforward to support magic wands in quantified permission assertions. The opposite (quantified permission assertions in magic wands) is possible as well, see Section 5.7.

The previously discussed rules for packaging and applying wands (Figure 5.2 and Figure 5.5, respectively) need to change as follows in order to properly use magic wand chunks: packaging requires (on line 4 of Figure 5.2) extracting all *value-determined* expressions $\overline{e}$ (defined next) and evaluating these (in the initial state σ_1) to the corresponding symbolic expressions $\overline{e'}$, determining the unique id id_{wand} associated with the skeleton of the wand to package (that is, after replacing the expressions $\overline{e}$ in a_1 `--*` a_2 with "holes"), and finally constructing the magic wand chunk representing the packaged wand instance ($id_{wand}(\overline{e'}; \_)$). The rule for applying wand instances needs to change analogously (on line 2 of Figure 5.5).

The set of *value-determined* sub-expressions of a magic wand are those sub-expressions that are either `old` expressions or heap-independent expressions. An expression is heap-independent if it does not contain any of the following expressions: a field read, a predicate unfolding or an application of a heap-dependent function.

5.3 Integrating Ghost Operations

The magic wand support described in the previous section (packaging and applying instances, and representing them in the symbolic state) forms the core of our solution, but it is not yet expressive enough to integrate well with all features of Viper, such as predicates. In particular, in the proof (packaging) of a new magic wand instance, it is often necessary to be able to specify ghost operations *between* the hypothetical addition of the wand's left-hand side and the proof of the right-hand side. For example, this might be necessary because the wand's right-hand side is a predicate instance that can be folded only once the state described by the wand's left-hand side is provided.

The running example from Listing 5.2 on page 130 exhibits an instance of this situation on line 22, when re-establishing the magic wand in the loop invariant. Recall that the wand `(xs != null ==> acc(list(xs))) --* acc(list(ys))` expresses that a predicate instance describing the complete list can be obtained by giving up the "remainder list" starting at `xs`. Consider how this invariant can be re-established at the end of the loop body: in particular, the state before line 22. In this state, permissions to the fields `zs.val` and `zs.next` of the current node (obtained from the `unfold` at line 18) are available, as is the magic wand instance `w` from the loop invariant at the beginning of the iteration (line 15), which has the same right-hand side, but requires `acc(list(zs))` on its left-hand side. Not all permissions required to package the wand instance needed in the new loop invariant are directly available; conceptually, those missing are in the footprint of the wand instance `w`. However, given the left-hand side assertion `(xs != null ==> acc(list(xs)))`, the right-hand side *can* be obtained by first folding the predicate instance `list(zs)`, and then applying the wand instance `w`. These ghost operations explain *how*, given the left-hand side, the permissions available in the state can be rearranged to obtain the desired right-hand side, which requires in the process the additional permissions `acc(zs.val)` and `acc(zs.next)` and the wand instance `w` (these constitute the footprint of the new wand instance).

---

g ::= a | `folding acc(`$pred(\overline{e})$`, `e`) in `g

```
| unfolding acc(pred(ē), e) in g
| packaging a -* g in g
| applying a -* a in g
```

Figure 5.6: Extending the syntax of Viper's assertion to support nesting a wand's right-hand side in a chain of ghost operations.

```
1 nested: G → A
2 nested(g) = g matches
3   folding _ in g1: nested(g1)
4   unfolding _ in g1: nested(g1)
5   applying _ in g1: nested(g1)
6   packaging _ in g1: nested(g1)
7   a: a
```

Figure 5.7: nested returns the inner-most assertion nested in a chain of ghost operations; its definition matches Figure 5.7.

5.3.1 Extended Algorithms

In order to allow ghost operations such as `(un)folding` a predicate instance to be expressed when packaging wand instances, we generalise the `package` statement to the form `package` a `--*` g, where g is an assertion a possibly nested inside ghost operations, as defined by Figure 5.6.

This definition allows nesting an assertion inside a ghost operation for each ghost operation that Viper supports in statement position: `(un)fold`, `package` and `apply` (in other verifiers, more could be added). The difference is that the syntax here indicates that the ghost operation should be applied *during* the footprint computation for the new wand instance, rather than in the current state.

A successful `package` a `--*` g operation does *not* add a wand instance of the form a `--*` g to the state, but rather a `--*` nested(g), where nested(g) is the inner-most assertion nested in the ghost operations (as defined in Figure 5.7). The role of the ghost operations is to indicate *how* the wand's semantics can be guaranteed, but they do not affect *what* the resulting wand instance represents.

```
    nested(folding acc(list(zs)) in applying w in acc(list(xs)))
=   acc(list(xs))
```

The automatic footprint computation is extended to support these ghost operations, as shown in Figure 5.8 and Figure 5.9. The rules given there define a modified version of `package`, and how to execute the ghost operations. The latter requires finding and transferring suitable permissions from the stack of input heaps, such that the specified ghost operation can be executed. For example, in order to execute a `folding`, it must be possible to find, in the input heaps, the permissions required by the body of the corresponding predicate instance.

Note that the rules for `package`, `packaging` and `applying` provided in this section are simplified analogously to the previously presented rules for `package` and `apply`: they focus on the handling of permissions and do not account for potential incompletenesses arising from framing heap values across `package`-`apply` pairs and branching executions. These aspects are discussed in Section 5.4.

```
exec(σ1, package a --* g, Q) =
  produce(σ1{h := ∅}, a, fresh, (λ σ_lhs ·
    σ_emp := σ_lhs{h := ∅}
    exec-ext([σ_lhs.h, σ1.h], σ_emp, g, (λ [h_lhs1, h2], σ_used, _ ·
      Let ch be a magic wand chunk corresponding to a --* nested(g)
          (as discussed in Section 5.2.5)
      Q(σ1{h := heap-add(h2, ch)})))))
```

Figure 5.8: Packaging a wand with ghost operations proceeds analogous to Figure 5.2, but checking the right-hand side (which involves computing a footprint) is preceded by executing the ghost operations via exec-ext (which potentially affect the footprint computation).

```
exec-ext: List[H] → Σ → G → (List[H] → Σ → Snap → R) → R

exec-ext(h̄, σ_ops, folding acc(pred(ē), p) in g, Q) =
  eval(σ_ops, p :: ē, (λ σ_ops1, p' :: ē' ·
    σ_emp := σ_ops1{h := ∅}
    bdy := scale(pred_body[x ↦ e'], p')
    consume-ext(σ_ops1.h :: h̄, σ_emp, bdy, (λ h_ops2 :: h̄1, σ_used1, _ ·
      exec(σ_used1, fold acc(pred(ē'), p'), (λ σ_used2 ·
        exec-ext(h̄1, merge-into(σ_used2, h_ops2), g, Q)))))))

exec-ext(h̄, σ_ops, unfolding acc(pred(ē), p) in g, Q) =
  eval(σ_ops, p :: ē, (λ σ_ops1, p' :: ē' ·
    σ_emp := σ_ops1{h := ∅}
    a := acc(pred(ē'), p')
    consume-ext(σ_ops1.h :: h̄, σ_emp, a, (λ h_ops2 :: h̄1, σ_used1, _ ·
      exec(σ_used1, unfold acc(pred(ē'), p'), (λ σ_used2 ·
        exec-ext(h̄1, merge-into(σ_used2, h_ops2), g, Q)))))))

exec-ext(h̄, σ_ops, applying a1 --* a2 in g, Q) =
  Let id_wand(ē') be a magic wand chunk identifier corresponding
      to a1 --* a2 (as discussed in Section 5.2.5)
  σ_emp := σ_ops{h := ∅}
  a := (a1 && acc(id_wand(ē')))
  consume-ext(σ_ops.h :: h̄, σ_emp, a, (λ h_ops2 :: h̄1, σ_used1, _ ·
    exec(σ_used1, apply a1 --* a2, (λ σ_used2 ·
      exec-ext(h̄1, merge-into(σ_used2, h_ops2), g, Q)))))

exec-ext(h̄, σ_ops, packaging a1 --* g1 in g2, Q) =
  produce(σ_ops{h := ∅}, a1, s, (λ σ_lhs ·
    σ_emp := σ_lhs{h := ∅}
    exec-ext([σ_lhs.h, σ_ops.h] :: h̄, σ_emp, g1, (λ [_, h_ops1] :: h̄1, _, _ ·
      Let ch be a magic wand chunk corresponding to a1 --* nested(g1)
      exec-ext(h̄1, σ_ops{h := heap-add(h_ops1, ch)}, g2, Q)))))))

exec-ext(h̄, σ_ops, a, Q) =
  consume-ext(σ_ops.h :: h̄, σ_ops{h := ∅}, a, Q)
```

Figure 5.9: Executing ghost operations. The first three rules exhibit the same structure: (1) consume-ext determines the footprint of the operation and transfers it from $\sigma_{ops_1}.h :: \overline{h}$ to $\sigma_{emp}.h$, yielding $h_{ops_2} :: \overline{h_1}$ and $\sigma_{used_1}.h$, (2) the actual operation is performed, rewriting $\sigma_{used_1}.h$ into $\sigma_{used_2}.h$, and (3) the execution continues in the updated states. The packaging ghost operation proceeds analogously to the package statement. The last case handles assertions with no further ghost operations.

As an example, consider the `package` statement on line 22 of Listing 5.2, and assume that σ denotes the state before the `package` statement. Hence, line 22 corresponds to performing an operation

$$\mathsf{exec}(\sigma, \texttt{package A --* folding acc(list(zs)) in ...}, Q)$$

Following Figure 5.8, this will essentially result in

$$\mathsf{exec\text{-}ext}([\sigma_{lhs}.h, \sigma.h], \sigma_{emp}, \texttt{folding acc(list(zs)) in ...}, (\lambda \ldots))$$

which means that all permissions necessary for executing the ghost operations must come from either the current heap $\sigma.h$ or the hypothetical left-hand side heap $\sigma_{lhs}.h$.

The rules for executing the first three ghost operations shown in Figure 5.9 (that is, `(un)folding` and `applying`) exhibit the same structure: first, consume-ext is used to find the permission necessary for executing the ghost operation at hand (and to check that all necessary pure assertions are true), and to transfer the corresponding heap chunks from $\sigma_{ops_1}.h :: \overline{h}$ into σ_{emp}, which yields $h_{ops_2} :: \overline{h_1}$ (the remainders of the input heaps) and σ_{used_1}. Next, the actual ghost operation is performed on σ_{used_1}, which is thereby rewritten into σ_{used_2}. Note that this operation is guaranteed to succeed because of the preceding consume-ext. This rewriting of the state does not change which assertions are satisfied by the state in terms of the ideal semantics of assertions, but for a verifier which differentiates between predicate instances and their bodies (and between wand instances and their footprints), the ghost operation can affect what the tool can show about the resulting state. Finally, the execution continues in the updated states.

In the context of the operations on line 22 of the running example (Listing 5.1), the execution of the ghost operation `folding acc(list(zs)) in ...` proceeds by invoking

$$\mathsf{consume\text{-}ext}([\sigma_{emp}.h, \sigma_{lhs}.h, \sigma.h], \sigma_{emp}, \texttt{list(zs)}_{body}, (\lambda \ldots))$$

which transfers the footprint of the body of `list(zs)` to σ_{emp} (the second argument). The footprint comprises `acc(zs.val)` and `acc(zs.next)`, and, assuming `zs.next != null`, the predicate instance `list(zs.next)`. As before, the algorithm tries to take as many permissions as possible from $\sigma_{lhs}.h$ (in general: from the top of the stack), but since $\sigma_{lhs}.h$ only provides `list(zs.next)`, the other permissions are taken from $\sigma.h$ (the current heap). The resulting stack of heaps is $[\emptyset, h_{lhs_1}, h_1]$ (where h_{lhs_1} is also the empty heap $\emptyset$), along with the single state σ_{used_1} which is a state satisfying $\texttt{list(zs)}_{body}$.

In the next step, σ_{used_1} is rewritten into σ_{used_2} by folding `list(zs)`, which replaces the permissions required by the predicate body by an instance of the predicate. Finally, the execution of the package statement from line 22 continues by invoking

$$\mathsf{exec\text{-}ext}([h_{lhs_1}, h_1], \sigma_{used_2} \uplus \sigma_{emp}, \texttt{applying w in acc(list(ys))}, Q)$$

thus executing the `applying` ghost operation, which proceeds by transferring the predicate instance `list(zs)` (the left-hand side of `w`) and the wand instance `w` itself from $[\sigma_{used_2}.h, h_{lhs_1}, h_1]$ to another fresh state σ_{emp}. In particular, `list(zs)` is taken from $\sigma_{used_2}.h$, and `w` is taken from h_1. Afterwards, `apply` is executed to rewrite the state that now contains `list(zs)` and `w` (denoted by σ_{used_1} on line 25 of Figure 5.9) such that it contains the right-hand side of `w`, that is, `list(ys)` (denoted by σ_{used_2} on line 25). Finally, `list(ys)` itself is transferred into a fresh σ_{emp} (see the last rule of Figure 5.9). This concludes the execution of the ghost operations, the computation and removal of the wand instances footprint, and the checking of the right-hand side; the only thing left to do is adding the packaged wand instance to the state (before continuing the verification). Consequently, the execution returns to the `package` rule from Figure 5.8, in which the σ_{used} is essentially a state satisfying exactly `list(ys)` — and as such, satisfies the right-hand side of the wand instance that was to be packaged.

The remainder of the left-hand side heap, h_{lhs_1}, is discarded (in the running example, it is empty anyway), and the remainder of the current heap, h_2, is extended with an instance of `A --* acc(list(ys))` before the symbolic execution continues to verify the remainder of the program. The permissions taken from the initial (current) heap $\sigma_1.h$, that is, the difference between $\sigma_1.h$ and h_2 (`acc(zs.val)` and `acc(zs.next)`, as well as the wand instance `w`) conceptually make up the newly-added wand's footprint.

Two cases from Figure 5.8 remain to explain: the first matches the ghost operation `packaging` a_1 `--*` g_1 `in` g_2, the second matches ghost-operation-free assertion a. The first case, which represents a recursive packaging of a wand instance (necessary, for example, for packaging nested magic wands of the form a_1 `--*` `(`a_2 `--*` a_3`)`) works similarly to the `package` statement: it creates a hypothetical left-hand side state σ_{lhs} satisfying a, pushes the corresponding heap onto the stack of already existing heaps, and executes ghost operations potentially occurring in g_1 (to which σ_{lhs} is available). Finally, it adds the magic wand instance a_1 `--*` $\mathsf{nested}(g_1)$ to the verification state and continues the execution. Note that the possibility of nesting package operations makes it necessary to work with heap stacks of arbitrary height: without it, a fixed number of heaps would suffice.

The last case, executing a ghost-operation-free assertion a, only applies to the innermost assertion nested inside a chain of ghost operations (that is, a is $\mathsf{nested}(g)$, for the g that is the right-hand side of the wand instance currently being packaged). At this point, the footprint computation falls back to the consume-ext operation of Figure 5.3.

5.3.2 Soundness of the Footprint Computation

In comparison to the symbolic execution rules presented in the previous chapters, the rules for handling magic wands are significantly more involved, in particular those for executing ghost operations. We therefore deem it necessary to sketch a formal soundness argument for the core of the presented magic wand support, the soundness of which essentially depends on the correctness of the footprint computation. In particular, we argue why the state removed by a package operation satisfies the properties that it was a part of the original state and satisfies the semantics of the newly-packaged wand (thus, any future `apply` of the wand instance will be justified). In the process, similar results are formulated for each of the operations heap-rem-max, transfer (both shown in Figure 5.4), consume-ext (Figure 5.3) and exec-ext (Figure 5.9), which are ultimately instantiated to show the desired property for `package`.

Preliminaries

We prove properties of the symbolic execution with respect to the semantics of Viper assertions. Assertion semantics have not yet been formalised for all of Viper's features, but previous work by Parkinson and Summers [102] and Summers and Drossopoulou [126] provide semantics for a subset of Viper that could be extended accordingly. The first work [102] introduces a semantics for a permission logic that combines elements of implicit dynamic frames (the permission logic that Viper uses, introduced by Smans et al. [118]) and separation logic: the resulting logic is called *total heaps permission logic*, and it includes accessibility predicates (to fields), magic wands, separating conjunctions and heap-dependent expressions (field dereferences). The second work [126] extends this logic with *equi-* and *isorecursive* semantics for (potentially recursive) predicates and functions: given a (recursively defined) predicate (or function), the *equirecursive* semantics treats a predicate instance as equivalent to its body, whereas the *isorecursive* semantics does not; the latter is thus closer to how most verifiers (including Silicon) reason about recursive definitions.

The assertion semantics in their work are given with respect to a program state consisting of a *store* $\mathcal{S}$ mapping variables to values, a *total heap* $\mathcal{H}$ mapping pairs of object identifiers and field names to values, and a *permission mask* $\mathcal{P}$ mapping such

pairs to permission values. In their semantics, a field read can always be evaluated to a value (which may be "undefined" due to potential race conditions), but in a well-defined program in which all assertions are *self-framing* (recall Section 2.2.1: an assertion is self-framing if it requires permissions to all heap locations it accesses), such potentially undefined field reads can occur only as part of assertions that are, due to insufficient permissions, *false* in any event.

The definition of well-definedness used in their work is similar to the definition used in Section 3.5: a program is *well-defined* if (1) each function body is framed by its function's precondition, (2) each function application is guaranteed to terminate if its precondition holds, (3) functions are only applied when their preconditions hold, (4) all specifications, including predicate bodies, are self-framing, and (5) partial functions are only applied if their preconditions hold. In the rest of this section, we only concern ourselves with well-defined programs. Fully defining a similar assertion semantics for Viper is outside of the scope of this thesis, we only provide the core cases (taken from [102] and [126]) to illustrate how such a semantics could look.

Definition 15 (Satisfiability of Path Conditions)**.** Let τ denote the *background theory* used by Silicon: a many-sorted first-order logic with non-linear integer and real arithmetic, theories for sets and sequences, axiomatisations of heap-dependent functions, definitions for snapshots, and all other requirements from Section 3.1.2 (and subsequent chapters). Then, let $\tau \vdash \pi$ denote that the path conditions π are *satisfiable* with respect to the background theory τ. ◁

Definition 16 (Evaluating Expressions)**.** Expressions are *evaluated* in a store and a heap, denoted by $[\![e]\!]_{\mathcal{S},\mathcal{H}}$. Assuming a well-formed program, the evaluation function can be defined as follows:

$$
\begin{array}{lcl}
[\![x]\!]_{\mathcal{S},\mathcal{H}} & = & \mathcal{S}(x) \\
[\![op(\overline{e})]\!]_{\mathcal{S},\mathcal{H}} & = & op(\overline{[\![e]\!]_{\mathcal{S},\mathcal{H}}}) \\
[\![e.f]\!]_{\mathcal{S},\mathcal{H}} & = & \mathcal{H}([\![e]\!]_{\mathcal{S},\mathcal{H}}, f) \\
[\![func(\overline{e})]\!]_{\mathcal{S},\mathcal{H}} & = & [\![func_{body}[\overline{x \mapsto e}]]\!]_{\mathcal{S},\mathcal{H}} \\
[\![e_1 \,?\, e_2 : e_3]\!]_{\mathcal{S},\mathcal{H}} & = & \text{if } [\![e_1]\!]_{\mathcal{S},\mathcal{H}} \text{ then } [\![e_2]\!]_{\mathcal{S},\mathcal{H}} \text{ else } [\![e_3]\!]_{\mathcal{S},\mathcal{H}}
\end{array}
$$

As before (for example, in Figure 3.1 on page 52), $op(\overline{e})$ denotes any heap-independent function of arbitrary arity, including literals. The value of a function is obtained by evaluating the expansion of its body (with actual instead of formal arguments), which is finite due to the requirements imposed on well-defined programs. Note that expression evaluation is in general a partial function (for example, in the case of division), but this is not a problem in practice due to the aforementioned restriction to well-formed programs. ◁

Definition 17 (Operations on Symbolic States)**.** Two permission masks $\mathcal{P}_1$ and $\mathcal{P}_2$ are *compatible*, denoted by $\mathcal{P}_1 \perp \mathcal{P}_2$, if $\forall\, o, f \cdot \mathcal{P}_1(o,f) + \mathcal{P}_2(o,f) \leq 1$.

The *combination* of two permission masks, denoted by $\mathcal{P}_1 * \mathcal{P}_2$, is defined pointwise as $(\mathcal{P}_1 * \mathcal{P}_2)(o,f) = \mathcal{P}_1(o,f) + \mathcal{P}_2(o,f)$ if $\mathcal{P}_1 \perp \mathcal{P}_2$ (and is otherwise undefined).

The *readable locations* of a permission mask $\mathcal{P}$, denoted by $rds(\mathcal{P})$, are all locations to which $\mathcal{P}$ provides non-zero permissions: $rds(\mathcal{P}) = \{(o,f) \mid \mathcal{P}(o,f) > 0\}$. The complement is denoted by $\overline{rds}(\mathcal{P})$.

Two program heaps $\mathcal{H}_1$ and $\mathcal{H}_2$ *agree* on a set of locations L, denoted by $\mathcal{H}_1 \overset{L}{\equiv} \mathcal{H}_2$, if $\forall\, (o,f) \in L \cdot \mathcal{H}_1(o,f) = \mathcal{H}_2(o,f)$. ◁

Definition 18 (Disjoint State Extensions)**.** The set of *locally-havoced disjoint extensions* of a program heap $\mathcal{H}$ and a permission mask $\mathcal{P}$ are all extensions $\mathcal{H}'$ and $\mathcal{P}'$ such that $\mathcal{P}'$ is compatible with $\mathcal{P}$ and $\mathcal{H}'$ agrees with $\mathcal{H}$ on all locations except those to which $\mathcal{P}'$ newly provides permissions:

$$locDisjExt(\mathcal{H},\mathcal{P}) = \{(\mathcal{H}', \mathcal{P}' \mid \mathcal{P}' \perp \mathcal{P} \wedge \mathcal{H}' \overset{rds(\mathcal{P}) \cup \overline{rds}(\mathcal{P}')}{\equiv} \mathcal{H})\}$$ ◁

Definition 19 (Total Heaps Assertion Semantics)**.** The satisfaction relation $\models$ for assertions with respect to program states is defined as the least fixpoint of the following equations:

$$
\begin{array}{lcl}
\mathcal{S},\mathcal{H},\mathcal{P} \models e & \Leftrightarrow & [\![e]\!]_{\mathcal{S},\mathcal{H}} = \mathit{true} \\
\mathcal{S},\mathcal{H},\mathcal{P} \models \mathtt{acc}(e.f,p) & \Leftrightarrow & \mathcal{P}([\![e]\!]_{\mathcal{S},\mathcal{H}},f) \geq [\![p]\!]_{\mathcal{S},\mathcal{H}} \\
\mathcal{S},\mathcal{H},\mathcal{P} \models a_1 * a_2 & \Leftrightarrow & \exists\, \mathcal{P}_1,\mathcal{P}_2 \cdot \mathcal{P} = \mathcal{P}_1 * \mathcal{P}_2 \;\wedge\; \mathcal{S},\mathcal{H},\mathcal{P}_1 \models a_1 \;\wedge \\
 & & \qquad \mathcal{S},\mathcal{H},\mathcal{P}_2 \models a_2 \\
\mathcal{S},\mathcal{H},\mathcal{P} \models \mathtt{acc}(\mathit{pred}(\overline{e}),p) & \Leftrightarrow & \mathcal{S},\mathcal{H},\mathcal{P} \models \mathsf{scale}(\mathit{pred}_{\mathit{body}}[\overline{x \mapsto e}],p) \\
\mathcal{S},\mathcal{H},\mathcal{P} \models a_1 \mathrel{-\!\!-\!\!*} a_2 & \Leftrightarrow & \forall\,(\mathcal{H}',\mathcal{P}') \in \mathit{locDisjExt}(\mathcal{H},\mathcal{P}) \cdot \\
 & & \qquad \mathcal{S},\mathcal{H}',\mathcal{P}' \models a_1 \Rightarrow \mathcal{S},\mathcal{H},\mathcal{P} * \mathcal{P}' \models a_2
\end{array}
$$
◁

Definition 20 (Concretisation of Symbolic States)**.** The *concretisation* of a symbolic state $\sigma = (\gamma,h,\pi)$, denoted by $\mathcal{C}(\gamma,h,\pi)$, yields the set of program states that the symbolic state represents. The concretisation is defined as follows, given that n is the number of heap chunks in h:

$$
\begin{array}{l}
\mathcal{C}(\gamma,h,\pi) = \{(\mathcal{S},\mathcal{H},\mathcal{P}) \mid \exists\, \mathcal{P}_1,\ldots,\mathcal{P}_{n+1} \cdot \mathcal{P}_1 * \ldots * \mathcal{P}_{n+1} = \mathcal{P} \wedge (\tau \vdash \\
\quad (\textstyle\bigwedge \pi) \wedge \\
\quad (\forall\, x \in \mathit{dom}(\gamma) \cdot \gamma(x) = S(x)) \wedge \\
\quad (\forall\, \mathit{id}_i(\overline{v};s,p) \in h \text{ if } \mathit{id}_i(\overline{v}) \text{ matches} \\
\qquad f(v): \\
\qquad\quad (\mathcal{S},\mathcal{H},\mathcal{P}_i) \models \mathtt{acc}(v.f,\ p)) \wedge \\
\qquad\quad s = \mathit{snap}(\mathtt{acc}(v.f,\ p),\mathcal{S},\mathcal{H},\mathcal{P}_i) \\
\qquad \mathit{pred}(\overline{v}): \\
\qquad\quad (\mathcal{S},\mathcal{H},\mathcal{P}_i) \models \mathtt{acc}(\mathit{pred}(\overline{v}),\ p)) \wedge \\
\qquad\quad s = \mathit{snap}(\mathtt{acc}(\mathit{pred}(\overline{v}),\ p),\mathcal{S},\mathcal{H},\mathcal{P}_i) \\
\qquad \mathit{id}_{\mathit{wand}}(\overline{v}): \\
\qquad\quad (\mathcal{S},\mathcal{H},\mathcal{P}_i) \models \mathit{id}_{\mathit{wand}}(\overline{v})) \wedge \\
\qquad\quad s = \mathit{snap}(\mathit{id}_{\mathit{wand}}(\overline{v}),\mathcal{S},\mathcal{H},\mathcal{P}_i)))\}
\end{array}
$$
◁

For brevity, we use symbolic values in assertions (for example, in $\mathtt{acc}(v.f,\ p)$) which can be regarded as syntactic sugar for extending the appropriate program store with fresh variables constrained to equal the symbolic values occurring in the assertions (such as v and p) and substituting the symbolic values in the assertions for the corresponding fresh variables. Moreover, we take the liberty of using the internal representation of magic wand instances ($\mathit{id}_{\mathit{wand}}(\overline{v})$) as an assertion, instead of the syntactically correct form $a_1 \mathrel{-\!\!-\!\!*} a_2$. However, the latter is uniquely determined by the former and a conversion is straightforward.

The concretisation of a symbolic state can be understood as a filter on program states that starts by requiring that the symbolic path conditions must be satisfiable with respect to a candidate program state (and the background theory τ), and that then adds further constraints that must be satisfiable in order for the candidate state to be included in the concretisation. More precisely, a program state $(\mathcal{S},\mathcal{H},\mathcal{P})$ is in the concretisation of a given symbolic state (γ,h,π) if it meets the following requirements:

(1) The permission mask $\mathcal{P}$ can be decomposed into $n+1$ sub-masks: one for each of the n chunks in the symbolic heap (further constraints will follow), plus an additional sub-mask that corresponds to permissions not represented in the symbolic heap but held in the program state, for example, in parent call frames.

(2) The path conditions π are satisfiable with respect to the background theory τ used by Silicon (in subsequent items, satisfiability with respect to τ is implicit).

(3) It must be possible to assign the same values to the symbolic store γ as are provided by the program store $\mathcal{S}$.

(4) Pairing up each heap chunk id_i with the corresponding sub-mask $\mathcal{P}_i$, the program sub-state $(\mathcal{S}, \mathcal{H}, \mathcal{P}_i)$ must satisfy the accessibility predicate that corresponds to the current heap chunk id_i, and it must be possible to assign values to the chunk's symbolic snapshot such that the symbolic snapshot is equal to the "real" snapshot of the corresponding accessibility predicate, that is, the snapshot computed from the program sub-state (*snap* is defined next in Definition 21).

The decomposition of a permission mask $\mathcal{P}$ into individual sub-masks $\mathcal{P}_i$ is in general not unique, but this is not an issue: a program state is in the concretisation of a symbolic state as long as there exists at least one decomposition for which the constraints imposed by the concretisation function are satisfiable.

Definition 21 (Semantic Snapshots)**.** The *semantic snapshot* of an assertion a in a program state $(\mathcal{S}, \mathcal{H}, \mathcal{P})$, denoted by $snap(a, \mathcal{S}, \mathcal{H}, \mathcal{P})$, is defined if $\mathcal{S}, \mathcal{H}, \mathcal{P} \models a$ and as the least fixpoint of the following equations:

$$
\begin{array}{lcl}
snap(\texttt{acc(}e\texttt{.}f\texttt{, \_)}, \mathcal{S}, \mathcal{H}, \mathcal{P}) & = & \mathcal{H}(\llbracket e \rrbracket_{\mathcal{S},\mathcal{H}}, f) \\
snap(e, \mathcal{S}, \mathcal{H}, \mathcal{P}) & = & unit \\
snap(a_1 \texttt{ \&\& } a_2, \mathcal{S}, \mathcal{H}, \mathcal{P}) & = & pair(snap(a_1, \mathcal{S}, \mathcal{H}, \mathcal{P}), snap(a_2, \mathcal{S}, \mathcal{H}, \mathcal{P})) \\
snap(\texttt{acc(}pred(\overline{e})\texttt{, \_)}, \mathcal{S}, \mathcal{H}, \mathcal{P}) & = & snap(pred_{body}[\overline{x \mapsto e}], \mathcal{S}, \mathcal{H}, \mathcal{P}) \\
snap(b \texttt{ ? } a_1 \texttt{ : } a_2, \mathcal{S}, \mathcal{H}, \mathcal{P}) & = & \text{if } \llbracket b \rrbracket_{\mathcal{S},\mathcal{H}} \text{ then } snap(a_1, \mathcal{S}, \mathcal{H}, \mathcal{P}) \\
 & & \text{else } snap(a_2, \mathcal{S}, \mathcal{H}, \mathcal{P})
\end{array}
$$

$$
snap(a_1 \texttt{ --* } a_2, \mathcal{S}, \mathcal{H}, \mathcal{P}) = \begin{cases} \lambda s_{hyp} \cdot snap(a_2, \mathcal{S}, \mathcal{H}', \mathcal{P} * \mathcal{P}') & \\ \qquad \text{if } \exists\, (\mathcal{H}', \mathcal{P}') \in locDisjExt(\mathcal{H}, \mathcal{P}) & \\ \qquad \text{such that } \mathcal{S}, \mathcal{H}', \mathcal{P}' \models a_1 & \\ \qquad \text{and } s_{hyp} = snap(a_1, \mathcal{S}, \mathcal{H}', \mathcal{P}') & \\ choose() \qquad \text{otherwise} & \end{cases}
$$

◁

Note that the snapshots returned by *snap* are of sort *Snap* (recall Section 3.1.2), that is, of the same sort as the symbolic snapshots used by Silicon (and correspondingly, the embedding of heap location values into *Snap* is omitted). This design decision is possible because the sorts (such as for booleans, integers and references) used by the symbolic execution (that is, the sorts of symbolic values) are the same sorts as used by the program states, and it is a sensible choice because it enables a direct comparison of symbolic and semantic snapshots.

All cases are straightforward except the one for magic wands: this case requires arguing that the snapshot of a magic wand in a given program state is uniquely determined, which in particular requires (1) arguing that the non-deterministic choice of a possible state extension $\mathcal{H}'$ and $\mathcal{P}'$ does not affect the snapshot of the right-hand side (if the imposed constraints are met), and (2) an explanation of the use of the *angelic choice* [48] operator $choose()$ if no suitable state extension exists. Before we discuss the two issues, observe (from the definition of *snap* for magic wands) that the permission masks $\mathcal{P}$ and $\mathcal{P}'$ are the footprint, respectively, the left-hand-side mask of the magic wand instance under consideration.

We address the angelic choice first: if there exists no possible extension of the footprint that satisfies the imposed constraints, then this characterises a vacuous wand instance that cannot be used, for example, because the chosen footprint includes permission that are (also) required by the left-hand side. Consequently, the snapshots of such inapplicable wands are effectively irrelevant because they will never be used. However, satisfying the semantics of these wands is trivial exactly because there are no possible state extensions and the definition of *snap* must therefore account for such wands. In the context of the concretisation of symbolic states the definition should be such that, given a symbolic state with inapplicable wand instances (whose snapshots can in general be arbitrary), it does not result in unsatisfiable constraints that remove program states from the concretisation that (trivially) satisfy such wand instances. We therefore assume an angelic choice of a snapshot: its concrete value is irrelevant since the snapshot will never be applied, but the snapshot may occur in constraints

(see the definition of the concretisation function), and the angelic choice ensures that the satisfiability of the overall constraints (which determines the membership of a program state in the concretisation) does not depend on this choice.

Next, we address the non-deterministic choice of a possible state extension. Let $(\mathcal{H}', \mathcal{P}')$ and $(\mathcal{H}'', \mathcal{P}'')$ be two such state extensions (that satisfy the imposed constraints), and (1) observe that the combination of $\mathcal{P}$ with $\mathcal{P}'/\mathcal{P}''$ satisfies a_2 (because $\mathcal{S}, \mathcal{H}, \mathcal{P} \models a_1 \mathrel{-\!\!-\!\!*} a_2$), and (2) recall from the definition of locally-havoced disjoint extensions (Definition 18) that the values of $\mathcal{H}'/\mathcal{H}''$ may only differ from those of $\mathcal{H}$ for locations to which $\mathcal{P}$ (that is, the wand instance's footprint) does not provide any permission. Furthermore, (3) observe (from the definition of *snap*) that the snapshot of an assertion (such as a_2) is ultimately determined by the accessibility predicates for fields that (transitively) occur in the assertion: for each $\texttt{acc}(e.f, \_)$, the snapshot contains an entry $\mathcal{H}'(e, f)$ (respectively, $\mathcal{H}''(e, f)$). For an arbitrary such accessibility predicate occurring in a_2 (that is, the wand's right-hand side), the corresponding permission are either provided (at least partially) by the wand's footprint $\mathcal{P}$ or (exclusively) by $\mathcal{P}'/\mathcal{P}''$ (which satisfy a_1). In the first case, observation (2) implies that $\mathcal{H}'(e, f) = \mathcal{H}(e, f) = \mathcal{H}''(e, f)$. Otherwise, it follows from the constraint $s_{hyp} = snap(a_1, \mathcal{S}, \mathcal{H}', \mathcal{P}')$ (imposed by the definition of *snap*), the analogous constraint for $\mathcal{H}''/\mathcal{P}''$, and from observation (3) that there exists a component s_j of s_{hyp} (the snapshot that represents the heap values of the left-hand side a_1) such that $\mathcal{H}'(e, f) = s_j = \mathcal{H}''(e, f)$. Hence, $\mathcal{H}'$ and $\mathcal{H}''$ agree in both cases on all values that possibly determine the returned snapshot. Finally, (4) observe (from the definition of *snap*) that the exact permission amount provided by $\mathcal{P}'/\mathcal{P}''$ does not determine the values of the snapshot, *except* that it may (recursively) affect the potential choice of state extensions for nested wands that occur in a_1 or a_2. However, by combining the arguments from the discussion about the angelic choice of snapshots (in cases where no state extension exists) with the arguments from the current discussion, we conclude that this potential effect does not pose a problem: if there are potential state extensions then their heaps agree on all relevant locations; if there are not (for specific choices of $\mathcal{P}'/\mathcal{P}''$ that prevent the nested wands from being applied), then the angelic choice ensures that the satisfiability of the overall constraints does not depend on the snapshots chosen for these inapplicable wand instances.

Definition 22 (Symbolic States Assertion Semantics)**.** The satisfaction relation $\models$ for assertions with respect to symbolic states is defined in terms of the program states in the concretisation of the given symbolic state:

$$\gamma, h, \pi \models a \Leftrightarrow \forall (\mathcal{S}, \mathcal{H}, \mathcal{M}) \in \mathcal{C}(\gamma, h, \pi) \cdot \mathcal{S}, \mathcal{H}, \mathcal{M} \models a \qquad \triangleleft$$

Definition 23 (Operations on Symbolic States)**.** The proof sketches to come make use of the following *sub-state*, respectively, *sub-heap* relation $\sqsubseteq$:

$$\begin{array}{lcl} \sigma_1 \sqsubseteq \sigma_2 & \Leftrightarrow & \forall a \cdot \sigma_1 \models a \;\Rightarrow\; \sigma_2 \models a \\ h_1 \sqsubseteq h_2 & \Leftrightarrow & \forall a, \gamma, \pi \cdot \gamma, h_1, \pi \models a \;\Rightarrow\; \gamma, h_2, \pi \models a \end{array}$$

In addition, the proof sketches use $h_1 \cup h_2$ to denote the union of two symbolic heaps h_1 and h_2. Recall (from Section 3.1.2) that symbolic heaps are defined to be multisets of heap chunks; the result of a heap union is the straightforward union of the two multisets. $\triangleleft$

Definition 24 (Leads-to Relation)**.** The proof sketches employ the *leads-to* relation $\rightsquigarrow$, used as in $\texttt{exec}(\ldots, Q) \rightsquigarrow Q(\ldots)$, which expresses that the execution of an operation (here exec) eventually results in the execution of another operation (here the continuation Q).

The leads-to relation is used in order to specify properties of the results of operations that do not (directly) return their results but instead pass them on to another (nested) operation, typically the outer operation's continuation. We omit the formal definition of the leads-to relation since it would effectively require the provision of an operational semantics for the language in which the symbolic execution rules are formalised. $\triangleleft$

Proof of Soundness of the Footprint Computation

By convention, we use unprimed variables (such as σ and h) to denote input arguments, primed versions (such as σ' and h') to indicate corresponding outputs and hatted versions (such as $\hat{h}$) to indicate the "removed parts" of input heaps.

The first theorem is concerned with heap-rem-max (Figure 5.4), which removes as many permissions as possible from a given symbolic heap.

Theorem 1. If $\mathsf{heap\text{-}rem\text{-}max}(h, \pi, id(\overline{v}), p) = \mathit{Some}(h', \_, p')$ then there exists $\hat{h}$ such that:

(P_1) $\hat{h} \cup h' \sqsubseteq h$

(P_2) $\forall \gamma \cdot \gamma, \hat{h}, \pi \models \mathtt{acc}(id(\overline{v}),\ p - p')$ ◁

The first property of Theorem 1 states that the input heap h can be partitioned into the sub-heap $\hat{h}$, removed by heap-rem-max, and the returned remainder heap h'; these two properties imply that heap-rem-max does not add permissions. The second property states that the removed sub-heap provides the permissions that were to be removed, that is, that sufficient permission were removed from the input heap h.

Proof of Theorem 1. The proof is straightforward: instantiating the removed heap $\hat{h}$ to be $\{id(\overline{w}; s, \mathit{min}(p, q))\}$, both properties follow immediately. □

The next theorem is concerned with transfer (Figure 5.4), which transfers permissions from a stack of heaps into h_{used}.

Theorem 2. If $\mathsf{transfer}(\overline{h}, h_{used}, \pi, id(\overline{v}), p) = \mathit{Some}(\overline{h'}, h'_{used}, \_)$ then there exist $\overline{\hat{h}}$ such that:

(P_1) $\overline{\hat{h} \cup h' \sqsubseteq h}$

(P_2) $h'_{used} \sqsubseteq (\bigcup \overline{\hat{h}}) \cup h_{used}$

(P_3) $\forall a, \gamma \cdot \gamma, h_{used}, \pi \models a \ \Rightarrow\ \gamma, h'_{used}, \pi \models a\ \mathtt{\&\&}\ \mathtt{acc}(id(\overline{v}),\ p)$ ◁

The first property is a lifting of the first property proven for heap-rem-max to a stack of heaps: the output heaps $\overline{h'}$ are obtained by removing $\overline{\hat{h}}$ from the input heaps $\overline{h}$. The second property states that permissions added to h_{used} have been removed from the stack of input heaps, and the third property states that the part h'_{used} was extended with provides the required permissions. In combination, the three properties state that transfer indeed transfers sufficient permissions from the input heaps into h_{used}.

Proof of Theorem 2. The proof proceeds by induction on the stack of input heaps. All cases in which transfer would return *None* (such as when the heap stack is empty) can be ruled out because a return value of *None* would contradict the assumption made in Theorem 2. If heap-rem-max returns $\mathit{Some}(h'', \_, p'')$, it has removed a $\hat{h}''$ from h that satisfies the properties from Theorem 1. In case all required permissions were transferred ($p'' = 0$), instantiating $\overline{\hat{h}}$ with $\hat{h}'' :: \overline{\emptyset}$ suffices to satisfy all three properties: the first is obvious; the second holds because the permissions added to h'_{used} have been removed from the top of the stack; the third holds because sufficient permission have been added to h'_{used}. In case further permissions need to be transferred ($p'' \neq 0$), instantiating $\overline{\hat{h}}$ with $\hat{h}'' :: \overline{\hat{h}'''}$, where $\overline{\hat{h}'''}$ are the sub-heaps removed by the recursive invocation of transfer (to which the induction hypothesis applies), in combination with the respective properties of heap-rem-max and transfer, suffices to show the desired properties. If heap-rem-max returns *None*, the properties follow from the induction hypothesis applied to the recursive invocation if $\overline{\hat{h}}$ is instantiated with $\emptyset :: \overline{\hat{h}''}$, where $\overline{\hat{h}''}$ are the parts the recursive invocation transferred. □

The next theorem is analogous to the previous, but concerned with consume-ext (Figure 5.3), that is, with the operation that in parallel removes the footprint of a wand instance and checks the instance's right-hand side.

Theorem 3. If $\mathsf{consume\text{-}ext}(\overline{h}, \sigma_{used}, a, Q) \rightsquigarrow Q(\overline{h'}, \sigma'_{used}, \_)$, that is, if the continuation Q is eventually invoked with the specified arguments, then there exist $\overline{\hat{h}}$ such that:

(P_1) $\overline{\hat{h} \cup h' \sqsubseteq h}$

(P_2) $\sigma'_{used}.h \sqsubseteq (\bigcup \overline{\hat{h}}) \cup \sigma_{used}.h$

(P_3) $\forall a_1 \cdot \sigma_{used} \models a_1 \;\Rightarrow\; \sigma'_{used} \models a_1 \;\&\&\; a$ ◁

The first two properties of consume-ext are analogous to those of transfer, and the third property is a generalisation thereof, since consume-ext finds sub-heaps satisfying a general assertion instead of a single accessibility predicate.

The proof of Theorem 3 requires two additional lemmas: the first states that the evaluation of an expression does not semantically modify the symbolic heap (whose structure might change, however, because of state consolidations).

Lemma 1. If $\mathsf{eval}(\sigma, e, Q) \rightsquigarrow Q(\sigma', \_)$ then $\forall a \cdot \sigma.h \models a \Leftrightarrow \sigma'.h \models a$. ◁

The second lemma states a similar property: that executing a ghost operation does not semantically modify the heap (the lemma for `fold` is shown, those for `unfold` and `apply` are analogous).

Lemma 2. If $\mathsf{exec}(\sigma,$ `unfold acc(`$pred$`(_), _)`$, Q) \rightsquigarrow Q(\sigma')$ then $\forall a \cdot \sigma.h \models a \Leftrightarrow \sigma'.h \models a$. ◁

Proof of Theorem 3. The proof proceeds by induction on the structure of the assertion a. In case of `acc(`$id(\overline{e})$`, `p`)`, the desired results immediately follow from transfer (and Lemma 1). In case of a pure assertion e, the properties hold if $\overline{\hat{h}}$ are instantiated with $\overline{\emptyset}$ (nothing is consumed). The only interesting case is that of a_1 `&&` a_2: applying the induction hypothesis to each of the two recursive invocations implies the existence of removed heaps $\overline{\hat{h}''}$ and $\overline{\hat{h}'''}$, respectively. By instantiating $\overline{\hat{h}}$ with $\overline{\hat{h}'' \cup \hat{h}'''}$, each of the three desired properties can be shown by appropriately combining the corresponding properties of $\overline{\hat{h}''}$ and $\overline{\hat{h}'''}$. □

The next theorem is again similar to the previous, but about exec-ext (Figure 5.9), that is, the operation that executes ghost operations potentially occurring on the right-hand side of a magic wand instance to be packaged.

Theorem 4. If $\mathsf{exec\text{-}ext}(\overline{h}, \sigma_{ops}, g, Q) \rightsquigarrow Q(\overline{h'}, \sigma'_{ops}, \_)$ then there exist $\overline{\hat{h}}$ such that:

(P_1) $\overline{\hat{h} \cup h' \sqsubseteq h}$

(P_2) $\sigma'_{ops}.h \sqsubseteq (\bigcup \overline{\hat{h}}) \cup \sigma_{ops}.h$

(P_3) $\sigma'_{ops} \models \mathsf{nested}(g)$ ◁

The first two properties of exec-ext are analogous to those of consume-ext and transfer. The third one expresses the intuitive understanding that exec-ext (if it succeeded) removed a suitable footprint for the wand that is being packaged, whose actual right-hand side is $\mathsf{nested}(g)$. In the process, the ghost operations nesting g rewrite $\sigma_{ops}.h$, which may entail transferring further permissions into the footprint.

Proof of Theorem 4. The proof proceeds by induction on the structure of the assertion g. The cases for `(un)folding/applying ... in` g_1 are all similar to each other: let $\overline{\hat{h}''}$ be the heaps removed by the invocation of consume-ext, and let $\overline{\hat{h}'''}$ be the heaps removed by the final recursive invocation of exec-ext (to which the induction hypothesis applies), and recall (Lemma 2) that executing a ghost operation does not semantically affect the heap since it only rewrites the representation. Instantiating $\overline{\hat{h}}$ with $\overline{\hat{h}'' \cup \hat{h}'''}$, properties ($P_1$) and ($P_2$) can be shown by appropriately

combining the properties of $\overline{\hat{h}''}$ and $\overline{\hat{h}'''}$ guaranteed by the respective theorems. Showing (P_3) is straightforward and only requires observing that $\mathsf{nested}(g)$ is equivalent to $\mathsf{nested}(g_1)$. In case of `packaging ... in` g_2, properties (P_1) and (P_3) can be shown analogously, but proving (P_2) is more involved, and in particular requires the following observations: the first recursive invocation of exec-ext indeed removes a footprint for the nested packaged wand (follows from suitably combining (P_2) and (P_3) of exec-ext), and since any footprint is at least as "expressive" (in terms of what can be deduced from it) as the corresponding wand instance (this follows from the wand's semantics, see Definition 19), replacing the footprint with a wand instance does not increase the expressiveness of the resulting state. In case of a (ghost-operation-free) assertion a, the desired properties follow directly from Theorem 3. □

Theorem 5. If $\mathsf{exec}(\sigma,\ \texttt{package}\ \ a\ \texttt{--*}\ \ g, Q)\ \ \rightsquigarrow Q(\sigma' \cup \{id_{wand}(\_;\_)\})$ then there exist $\hat{h}$ such that:

(P_1) $\hat{h} \cup \sigma'.h \sqsubseteq \sigma.h$

(P_2) $\sigma.\gamma, \hat{h}, \sigma.\pi \models a$ `--*` $\mathsf{nested}(g)$

◁

Theorem 5 finally expresses the desired property of `package`: that packaging a wand indeed removes a footprint and thus, that adding a corresponding wand instance to the verification state is justified.

Proof of Theorem 5. The proof is straightforward: both properties (in particular, the second) can be shown by combining the properties obtained from instantiating Theorem 4 for the invocation to exec-ext, if $\hat{h}$ is chosen to be the heap removed by the latter. □

With Theorem 5, we conclude the discussion of supporting ghost operations such as `unfolding` in our automatic footprint computation, which is necessary in order to integrate our magic wand support with other Viper features that require user annotations, such as predicates. We also shift the focus of the discussion from permissions to heap values: the next section discusses potential value-related incompletenesses and how to overcome them.

5.4 Magic Wand Snapshots

The presentation of our technique for supporting magic wands focused so far on the handling of permissions, in particular on how to automatically compute suitable footprints. In this section we discuss two challenges that are related to the *values* of heap locations (rather than permissions to those locations), and that can result in incompletenesses if left unaddressed: (1) *framing* the values of heap locations to which permissions are temporarily "lost" to a magic wand instance's footprint across `package-apply` pairs, and (2) handling *branching* executions of `package` that (speculatively) branch over values from the hypothetical left-hand-side heap and "synchronising" such branches across `package-apply` pairs. The key to addressing both sources of incompleteness are *magic wand chunk snapshots*, which are introduced next.

5.4.1 Framing

Consider the program in Listing 5.3 and observe that the footprint of the packaged magic wand instance is the permission to `x.g`, and that the final assertion should hold because `x.g` had value 1 when permission to `x.g` was given up (as the wand's footprint) and `x.f` has value 2 when the wand is applied.

This reasoning requires framing the value of `x.g` across `package-apply` pairs, similar to how it is necessary to frame the value of locations to which permissions are folded

```
1  field f: Int
2  field g: Int
3
4  predicate pair(x: Ref) {
5    acc(x.g) && acc(x.f)
6  }
7
8  function sum(x: Ref): Int
9    requires acc(pair(x))
10 { unfolding acc(pair(x)) in x.f + x.g }
11
12 method test(x: Ref) {
13   inhale acc(x.f) && acc(x.g)
14   x.g := 1
15   package acc(x.f) --* folding acc(pair(x)) in acc(pair(x))
16   x.f := 2
17   apply acc(x.f) --* acc(pair(x))
18   assert sum(x) == 3
19 }
```

Listing 5.3: A simple Viper program illustrating framing issues that arise from our treatment of magic wands.

into a predicate instance across `fold`-`unfold` pairs. The latter was discussed in detail in Chapter 3 (in particular in Section 3.3) and is achieved via predicate *snapshots*, which represent the values of the partial heap described by a predicate instance: upon fold, the consumption of the body yields a snapshot that represents the values of the locations to which permissions were (partially) consumed; this snapshot is stored in the folded predicate chunk and used upon unfold to restore the values of the locations to which the predicate body provides permissions again.

This idea can be adapted for magic wand chunks as follows: during the packaging of a wand instance the right-hand side is consumed, and the obtained snapshot — which includes the values of the footprint locations — can be stored as the snapshot of the packaged wand instance (that is, of the corresponding magic wand chunk). The snapshot is then used upon applying the wand instance to restore the values of those locations to which permissions come from the instance's footprint.

In the (simplified) rule for packaging magic wands, shown in Figure 5.2, the consumption of the right-hand side takes place on line 3: it computes a suitable wand instance footprint and removes it from the current heap, and it returns a snapshot (omitted in Figure 5.2, but shown in the definition of consume-ext in Figure 5.3) representing the heap values of the consumed right-hand side. This snapshot in general comprises heap values from the footprint *and* the (at the point of packaging) hypothetical left-hand side, and it can thus be understood as a function of the heap values of the *actual* left-hand side that will be provided if the wand instance is applied.

To illustrate this idea, consider again the previous example from Listing 5.3: the snapshot of the predicate instance obtained from folding `pair(x)` on line 15 is the pair of snapshots $(1, s)$, where 1 is the value of `x.g` (permission to which go into the packaged wand instance's footprint) and s is the hypothetical value of `x.f` (permission to which are provided by the left-hand side). The snapshot of the consumed right-hand side (which, after having erased the ghost operations, is just `acc(pair(x))`) therefore is the pair $(1, s)$, and the snapshot of the packaged wand instance thus is $\lambda s \cdot (1, s)$, making it a function of the heap values of the to-be-provided left-hand side. When the wand instance is applied on line 17, the snapshot obtained from consuming the actually provided left-hand side is 2 (the value of `x.f`): applying it to the wand instance's snapshot yields $(1, 2)$, which in turn is used to determine the heap values of the produced right-hand side (that is, it becomes the snapshot of the predicate instance `pair(x)`). Consequently, the final assertion succeeds.

```
1  exec(σ1, package  a --* g, Q) =
2    s_lhs := fresh
3    produce(σ1{h := ∅}, a, s_lhs, (λ σ_lhs ·
4      exec-ext([σ_lhs.h, σ1.h], σ_lhs{h := ∅}, g, (λ [_, h2], _, s_rhs ·
5        Let id_wand(ē') be a magic wand chunk identifier corresponding
6            to a --* nested(g) (as discussed in Section 5.2.5)
7        ch := id_wand(ē'; (λ s_lhs · s_rhs))
8        Q(σ1{h := heap-add(h2, ch)})))))
9
10 exec(σ1, apply  a1 --* a2, Q) =
11   Let id_wand(ē') be a chunk identifier corresponding to a1 --* a2
12   consume(σ1, a1, (λ σ2, s_lhs ·
13     consume(σ2, acc(id_wand(ē)), (λ σ3, (λ s_hyp · s_wand) ·
14       produce(σ3, a2, (λ s_hyp · s_wand)(s_lhs), Q)))))
```

Figure 5.10: Rules for packaging and applying magic wand instances that account for framing the values of locations to which permissions were transferred into an instance's footprint. Differences relative to the previous rules are highlighted.

Figure 5.10 shows rules for packaging and applying wand instances that implement the discussed idea of magic wand chunk snapshots as functions of the left-hand-side heap (differences with respect to to the simplified rules shown in Figure 5.2, respectively, Figure 5.5, are highlighted).

When a magic wand instance is packaged, a fresh snapshot s_{lhs} is used to represent the values of the hypothetical left-hand-side heap (by using it as the production snapshot of the left-hand side on line 3). The snapshot s_{rhs} that represents the values of the right-hand side (obtained from consuming the right-hand side on line 4) is in general composed of values (snapshot components) from the current heap and from the left-hand-side heap: the former in case permissions (to a specific heap location) are provided by the footprint, the latter in case the permissions come from the wand's left-hand side. s_{rhs} can therefore be understood as a function of s_{lhs}, and consequently, the new chunk's snapshot is set to be $\lambda s_{lhs} \cdot s_{rhs}$.

Upon application of a wand instance, the snapshot s_{lhs} representing the values of the actually provided left-hand side is applied to the snapshot of the "applied" magic wand chunk and used to determine the values of the obtained right-hand side (line 14).

5.4.2 Branching Executions

In this subsection we discuss potential incompletenesses arising from branching `package` operations, and how magic wand chunk snapshots can be used to prevent them. As an illustration of this problem, consider the example shown in Listing 5.4, for which the symbolic execution of the `package` statement branches over the unknown hypothetical value of `x.b` (when consuming the right-hand side in order to compute and remove the packaged instance's footprint). On one out of two potential symbolic execution paths, the wand's footprint consumed during packaging would not match the footprint produced as part of the obtained right-hand side at the point where the wand were applied: for example, considering the if-branch of the right-hand side during packaging, which would remove a footprint of `acc(x.f, 1/10)`), but the else-branch during applying, which would add a footprint of `acc(x.f, 1/10) && acc(x.g, 1/10)`. The resulting state would hold too many permissions — `acc(x.f, 2/10) && acc(x.g, 3/10)` — but since the state would be consistent, the final assertion would incorrectly fail.

```
1  inhale acc(x.b) && acc(x.f, 2/10) && acc(x.g, 2/10)
2  inhale x.f == 2 && x.g == 1
3  package acc(x.b) --*    acc(x.b)
4                       && (x.b
5                             ? acc(x.f, 1/10)
6                             : acc(x.f, 1/10) && acc(x.g, 1/10))
7  x.b := false
8  apply   /* same wand */
9  assert perm(x.f) == perm(x.g)
```

Listing 5.4: A simple Viper program illustrating a potential incompleteness arising from branching executions (here: of package).

1 exec(σ_1, package a --* g, Q) =
2 s_{lhs} := fresh
3 id_{scope} := fresh
4 σ_{emp} := $\sigma_1\{h := \emptyset, \pi :=$ pc-push$(\sigma_1.\pi, id_{scope}, true)\}$
5 produce($\sigma_1\{h := \emptyset\}$, a, s_{lhs}, (λ σ_{lhs} ·
6 exec-ext($[\sigma_{lhs}.h, \sigma_1.h]$, $\sigma_{lhs}\{h := \emptyset\}$, g, (λ [_, h_2], σ_{used}, s_{rhs} ·
7 (_, bcs) := pc-segs(pc-after($\sigma_{used}.\pi$, id_{scope}))
8 Let $id_{wand}(\overline{e'})$ be a magic wand chunk identifier
9 corresponding to a --* nested(g)
10 $ch := id_{wand}(\overline{e'}; (\lambda\ s_{lhs}\ \cdot s_{rhs}))$
11 $Q(\sigma_1\{h :=$ heap-add(h_2, ch), $\pi :=$ pc-add$(\sigma_2.\pi, bcs)\}$))))))
12
13 exec(σ_1, apply a_1 --* a_2, Q) =
14 Let $id_{wand}(\overline{e'})$ be a chunk identifier corresponding to a_1 --* a_2
15 consume(σ_1, a_1, (λ σ_2, s_{lhs} ·
16 consume(σ_2, acc$(id_{wand}(\overline{e}))$, ($\lambda$ σ_3, (λ s_{hyp} $\cdot s_{wand}$) ·
17 produce($\sigma_3\{\pi :=$ pc-add$(\sigma_3.\pi, s_{hyp} := s_{lhs})\}$, a_2, s_{wand}, Q)))))

Figure 5.11: The final rules for packaging and applying magic wand instances: in addition to accounting for framing, these rules also account for branching executions. Differences relative to the previous rules are highlighted.

This potential incompleteness is caused by mismatching assumptions about the (conceptually) *same* value of `x.b`: on one path through the packaging operation, the symbolic execution speculates that the hypothetical (future) value of `x.b` might be *true*, but when the wand is applied later on (and the value is fixed), the initial assumption is not checked against the actual value, and the infeasibility of the incorrectly taken execution path is thus not detected.

Detecting such an infeasible path (and thereby preventing incompletenesses such as the previously illustrated one) requires preserving branch conditions that speculate on hypothetical left-hand-side values after the packaging operation finished (in contrast to regular path conditions coming from the hypothetical left-hand side, which must be retracted). This is achieved by the final versions of the rules for packaging and applying magic wands, shown in Figure 5.11.

When packaging a wand instance, the branch conditions obtained from producing a hypothetical left-hand side and consuming the right-hand side are (as usual) recorded in the path conditions, from which they are extracted afterwards (by computing the delta in branch conditions between the states on line 4 and line 7) and then preserved by adding them to the final state (line 11). Upon applying a wand (also shown in Figure 5.11), it is necessary to synchronise the branch taken on package (by speculating on hypothetical left-hand-side heap values) with the branch taken on apply (which is determined by values from the actually provided left-hand side).

It order to see how this synchronisation can be achieved, consider again the previous example where the execution of `package` branches over the hypothetical value of `x.b`: this value is the snapshot[4] s_{lhs} used to produce the hypothetical left-hand side (see line 2 and line 5 of Figure 5.11). The branch condition resulting from taking the (with hindsight infeasible) if-branch thus is $s_{lhs} = \mathit{true}$ (which is then preserved after the package operation succeeded). However, when the packaged wand instance is afterwards applied, the *actual* value of `x.b` turns out to be *false* and the else-branch is taken instead.

Synchronising these branches can be achieved by substituting, in the path conditions, all occurrences of the snapshot that represents the heap values of the *hypothetical* left-hand side for the snapshot that represents the heap values of the *actual* left-hand side: in the rule for applying wands (in Figure 5.11), this would correspond to replacing all occurrences of s_{hyp} (taken from the "applied" wand chunk on line 16) with s_{lhs} (obtained from consuming the left-hand side on line 15). Note that this substitution can be understood as instantiating path conditions that are functions of the left-hand-side heap with the appropriate actual heap values, similar to how the magic wand chunk snapshot ($\lambda s_{hyp} \cdot s_{wand}$) is a function of the left-hand-side heap and can thus be instantiated.

However, instead of actually performing these instantiations (of path conditions and the magic wand snapshot), the rule for applying wand instances equates the formal snapshot argument s_{hyp} with the actual snapshot argument s_{lhs} and then uses the function body s_{wand} (and analogously, the path conditions) without substituting s_{hyp} for s_{lhs} (on line 17 of Figure 5.11). Letting the formal argument s_{hyp} escape this way is sound because: (1) by construction, each s_{hyp} is a unique, that is, syntactically different, symbol (see line 2 of Figure 5.11), and (2) each wand chunk can be applied at most once, due to the absence of fractional wands. Fractional wands would allow applying a single wand chunk multiple times, in which case the taken approach of equating the actual and the formal left-hand-side snapshot would no longer be sound: it would result in the in general unsound assumption that the values of each provided left-hand side are the same. In order to support fractional wands, it would therefore be necessary to properly instantiate magic wand chunk snapshots and path conditions.

5.4.3 Ghost Operations

The previously presented rules for executing the ghost operations `packaging` and `applying` (Figure 5.9 on page 144) need to be changed analogously to the final rules for executing the corresponding statements `package` and `apply` (Figure 5.11); for brevity, these changes are omitted.

It is *not* necessary to change the rules for the remaining ghost operations (that is, for (un)folding predicates), also shown in Figure 5.9: these rules transfer snapshots between states (as part of transferring permissions) and they rewrite the state representation (which typically affects the structure of snapshots, for example, when a predicate instance is folded), but they ultimately preserve the left-hand-side snapshot between producing the hypothetical left-hand side and consuming the innermost right-hand-side assertion that finally yields the magic wand chunk's snapshot.

5.4.4 Soundness of the Snapshot Computation

The main theorem of a soundness proof concerned with the computation of magic wand snapshots would state that the wand instance's snapshot obtained from packaging a wand does not provide more information than the footprint that has

[4] In general, the heap values of the hypothetical left-hand side are components of s_{lhs}, but in this example the left-hand side contains only a single accessibility predicate and the value of the corresponding location is therefore s_{lhs} itself.

been removed during the package operation. Such a theorem could be expressed by stating that if a magic wand chunk (recording such a snapshot), obtained from a package operation in some state σ, is afterwards "applied" in another state σ_0 then the resulting state is no more expressive than the combination of σ_0 with the footprint removed during the original package operation (from which the wand chunk and its snapshot were obtained).

We believe that it is possible to prove the main theorem similarly to how the main theorem (Theorem 5) from the sketch of soundness of the footprint computation is proven: by proving similar results for the "lower-level" operations such as heap-remmax, transfer, consume and consume-ext. For example, a corresponding theorem for consume would state that if a snapshot, obtained from consuming an assertion a in a state σ, is afterwards used to produce a in another state σ_0 then the resulting state is no more expressive than the combination of σ_0 with the sub-heap removed during the original consumption. This lemma, for example, can be argued intuitively as follows, assuming that a result similar to Theorem 3 (concerned with consume-ext) is proven for consume (which we expect to be straightforward because Theorem 3 is essentially a generalised version of consume): given an initial state σ and an assertion a, consume removes a sub-heap $\hat{h}$ of σ that satisfies (the spatial sub-assertions of) a. By construction, the snapshot returned by consume is composed of heap values from the removed sub-heap, and the snapshot can therefore not provide more information than the sub-heap it was computed from.

However, a proper formalisation of the main theorem and the necessary lemmas would require a notion of "relatedness" between the states from which snapshots are computed and the states in which they are used, such as the state σ in which a wand instance is packaged and the state σ_0 in which it is subsequently applied. Being appropriately related requires, for example, that all conditionals included in an assertion evaluate to the same value in both states, and that all expressions in the "holes" of a magic wand instance (Section 5.2.5) evaluate (pairwise) to the same symbolic values. We expect that formally defining "relatedness" will require a representation of snapshots that provides additional information about the relation between the snapshot and the partial heap it corresponds to, that is, the partial heap it was obtained from. Currently, (1) a snapshot is effectively just a tree of values with no (explicit) relation to the heap locations to which these values correspond, and no information under which condition (if any) this correspondence holds; and (2) this relation is (implicitly) restored when a snapshot is used to produce an assertion, but there are no checks in place that guarantee[5] that the assertion has any relation to the originally consumed partial heap (recall also the discussion of the limitations of the current representation of snapshots from Section 3.3). We believe that the representation of snapshots can be extended accordingly (potentially for the purpose of the proofs only, not for the actual implementation), for example, as a map from locations to values, and that this will enable us to prove the previously described results. However, such a change would most likely affect the majority of the presented rules and increase their complexity by additional, snapshot-related "bookkeeping", and a proof of soundness for the snapshot computation is therefore left as future work.

5.5 Inferring Annotations

In order to reduce the annotation overhead involved in specifying magic-wand-related ghost operations, Silicon additionally implements a set of simple (and optional) heuristics that attempt to insert additional `package` and `apply` operations into an input program. As described in Section 5.3, a `package` operation may also require nested ghost operations in order to succeed; the heuristics also attempt to infer these.

[5] An external argument is necessary to provide these guarantees, as discussed in Section 3.3.

```
1  var xs: Ref := ys
2  sum := 0
3
4  while (xs != null)
5    invariant xs != null ==> acc(list(xs))
6    invariant (xs != null ==> acc(list(xs))) --* acc(list(ys))
7    invariant sum ==   old(sum_rec(ys))
8                    - (xs == null ? 0 : sum_rec(xs))
9  {
10   unfold acc(list(xs))
11   sum := sum + xs.val
12   xs := xs.next
13 }
```

Listing 5.5: Verified version of the body of `sum_it` (Listing 5.1), with heuristics enabled.

The heuristics are *failure-driven* (similar to the try-react-retry execution flow discussed in Section 3.4.2 in the context of heap incompletenesses and state consolidations): if exhaling (consuming) an assertion a, for example, a loop invariant, fails due to insufficient permissions (to a field, a predicate or a wand), then the heuristics are applied in order to search for ghost operations that avoid the failure. The heuristics search (in a depth-first manner) for a sequence of ghost operations (comprised of `(un)folding`, `applying` and `packaging`) that would rewrite the state such that the initially missing permissions can be found. The width of the search tree is bounded by the number of predicate and magic wand instances held in the current state (which is finite and typically small). The depth of the search is bounded by a configurable threshold. The candidate ghost operations are also ordered according to a number of syntactic criteria on the symbolic state and program text, as a coarse estimate of which operations are "likely" to be successful: for example by preferentially unfolding predicate instances whose bodies appear (according to a syntactic inspection) to contain a suitable permission.

Currently, Silicon implements the following heuristics: (1) apply and unfold wand and predicate instances, respectively, that potentially contain the missing permission, (2) package and fold missing wand and predicate instances, respectively, and (3) apply and unfold any other wand and predicate instances, respectively, that the current state contains. The heuristics are (currently) applied only when the verification fails due to insufficient permission because the failing assertion is a suitable criterion for deciding which heuristic to attempt (for example, which predicate to unfold first). If an assertion fails otherwise, that is, because of a functional property, it is less clear how useful the failed assertion would be in order to guide potential heuristics; investigating this is left as future work. The three heuristics are tried in the order in which they are presented: the first heuristic is to apply/unfold any wand/predicate instances that the state contains and that may provide the missing permission. In case of a missing wand/predicate instance, an alternative heuristic is to attempt packaging/folding the missing instance: this heuristic comes second because, if the state held a wand/predicate instance that could be applied/unfolded to gain the missing instance, an attempt to packaging/folding another instance is in general less likely to succeed (since it might require permission which are already transitively in the instance that should be applied/unfolded instead). Note that VeriFast [67] implements a similar heuristic (essentially (1) and (2) from above), which attempts to unfold predicate instances that appear to provide missing permission, and to fold missing instances. The heuristics implemented by Silicon are more general, however, since they also account for magic wands (which are not supported by VeriFast); VeriFast also does not (currently) implement heuristic (3).

As an example, consider the loop body from the running example (Listing 5.2), and assume that the `package` statement on line 22 were removed. The resulting program cannot (immediately) be verified, in particular because the verifier fails to prove that

the invariant is preserved by the loop body because it cannot find the wand instance `A --* acc(list(ys))` in the current state. This triggers the heuristics, which first detect that there is no predicate (or wand) instance in the current state that can be unfolded (or applied) to get a suitable wand instance. The heuristics then try to package `A --* acc(list(ys))`, which fails because the required predicate instance `list(ys)` cannot be found in either the current state or the hypothetical left-hand-side state. This failure triggers the heuristics again, and results in an attempt to apply the wand instance `w` (which mentions `list(ys)`). Applying `w` fails as well, however, because this wand's left-hand side `list(zs)` is missing. The heuristics are triggered once again, and try to fold `list(zs)`, which succeeds. The previously failing operations are then retried: that is, applying `w` and exhaling `list(ys)`, both of which succeed now. With these nested ghost operations, the initially triggered packaging of `A --* acc(list(ys))` also succeeds, which enables the verifier to find the previously missing wand instance, and therefore, to show that the loop invariant is preserved.

Silicon's heuristics enable removing all `package` and `apply` statements from the examples listed in Section 5.6. Regarding the running example (Listing 5.2), they also enable removing `w` and `zs` (which were only used to facilitate writing the `package` statement on line 22), declared on line 15 and line 17, respectively. The resulting encoding of the running example is shown in Listing 5.5: Silicon verifies it if heuristics are enabled. The latter is currently done by adding the declaration of a special field (`field __CONFIG_HEURISTICS: Bool`) to the program for which heuristics are to be enabled. In future versions of Silicon, it should be possible to enable heuristics in a more suitable manner, for example, via a command-line option.

5.6 Evaluation

To evaluate the performance of our technique for supporting magic wands, we evaluated Silicon on a number of interesting examples, listed below. In addition to these, Viper's test suite also contains numerous regression tests that make use of magic wands, but these are typically short and simple, and were thus not included in the evaluation.

The evaluation was performed on a Intel Core i7-2600K 3.40GHz machine running Windows 10 x64 from an SSD. Each example[6] is included in two versions; the version with the suffix `_heuristics.sil` is the example with heuristics activated and with all magic-wand-related annotations removed. The reported runtimes are averaged over ten runs per example (the standard deviations were always less than 0.1s), and (as in previous evaluations) the Nailgun tool was used to persist a JVM between verification runs. Per example, two runtimes are provided: the first figure is the runtime of the version of the example that includes all magic-wand-related annotations, the second figure is the runtime if the annotations are not included and instead inferred by the heuristics (that is, of the example version with the suffix `_heuristics.sil`).

- `list_sum.sil` is the running example of this chapter. It verifies in 0.3s, both with and without heuristics enabled.
- `list_insert.sil` is an encoding of an iterative algorithm for inserting a value into a sorted linked list. It verifies in 1.0s/2.0s (without/with heuristics).
- `tree_delete_min.sil` is an encoding of challenge 3 from the VerifyThis verification competition at Formal Methods 2012 [63], which was verifying an iterative implementation removing the minimal element from a binary search tree. The example verifies in 0.4s/0.5s. VerCors [17] (the only comparable tool we are aware of with magic wand support) requires substantially more annotations to specify this example, and takes substantially longer to verify the example:

[6] The examples are included in Viper's test suite, which is part of Viper's sources. Further information can be found on the Viper project page [94].

[17] reports a runtime of 13 minutes (on a comparable machine). According to personal communication with the authors, the runtime has since been decreased to 60s by replacing the previously used verification back end Chalice [84] with Viper (and Silicon). However, VerCors still uses its own encoding of magic wands (see Section 5.8), not our direct support for magic wands.

- `un_currying.sil` demonstrates how nested ghost operations can be used to prove the standard "currying" and "uncurrying" property of magic wands: $a_1 * a_2 \mathrel{-\!\!*} a_3 \Leftrightarrow a_1 \mathrel{-\!\!*} (a_2 \mathrel{-\!\!*} a_3)$. The "$\Rightarrow$" case is especially interesting since it requires nested packaging operations. The example verifies in 0.2s, both with and without heuristics enabled.
- `conditionals.sil` illustrates the use of magic wands where the footprint is affected by conditionals whose guards depend on locations that are provided by the left-hand side of the wand. The example verifies in 0.2s. No version for activated heuristics is provided because adding assertions that initially fail (and then trigger the heuristics) amounts to more work than directly writing the corresponding `package`/`apply` statements.

5.7 Implementation

Section 5.4.2 described the potential problem of incompletenesses arising from assumptions made about the hypothetical left-hand side (on apply) that do not hold for the actually provided left-hand side (on package), and explained how path conditions and wand chunk snapshots can be used to prevent such incompletenesses: by ensuring that contradicting assumptions result in inconsistent states and by terminating infeasible execution paths.

To prevent taking such infeasible execution paths in the first place (each of which potentially branches again, for example, due to subsequent conditionals or further `package` statements), the implementation performs a join of all execution paths through a `package` statement. The join is similar to the join of branching expression evaluations discussed in detail in Section 3.4.3, but more involved since it requires joining symbolic heaps (in addition to path conditions). As before, joining paths effectively shifts work from the verifier to the prover, by reducing the number of execution paths at the expense of disjunctions in the path conditions. It would be interesting, regarding Silicon's style of symbolic execution in general, to investigate if this trade-off between execution paths and disjunctive path conditions exhibits a sweet spot and if it is possible to devise heuristics that guide the verifier towards that spot.

Silicon's implementation does not yet support the following features and feature combinations; their implementation is left as future work:

- Magic wand snapshots, discussed in Section 5.4, are not supported and the examples discussed in that subsection therefore cannot currently be verified.
- Magic wands are not integrated into Viper's permission introspection features (`perm` and `forperm`, recall Section 3.4.4). However, due to the uniform representation of heap chunks, we do not expect any conceptual problems to arise from integrating magic wands with `(for)perm`.
- Using quantified permission assertions (Chapter 4) on the left- or right-hand side of a wand is not yet supported, but is not expected to pose conceptual problems: similarly to how consume-ext extended consume for non-quantified permissions (transferring permissions from a stack of heaps into a target heap, instead of just removing them), consume-ext needs a case for quantified permission assertions, which would be analogous to the regular rule for consuming such assertions (Figure 4.1), but invoking an extension of qp-remove that transfers permissions from a stack of heaps.

- The opposite integration — specifying permission to an unbounded number of magic wand instances via quantified permissions — is not yet supported either, but implementing the corresponding support is expected to be straightforward, due to Silicon's uniform representation of chunks: magic wand chunks are, from a technical point of view, essentially predicate chunks, and can therefore be lifted to quantified chunks in the same way that predicate chunks are.
- Magic wands are not yet supported in function preconditions because Viper does not provide an `applying ... in` *e* expression (the `applying` ghost operation discussed in Section 5.4.3 is only supported on the right-hand side of a wand to package) that applies a magic wand for the scope of the nested expression, similar to an `unfolding` expression. Without such an expression, there is no way to make use of magic wands in a function's body. Furthermore, in order to frame functions with magic wands in their preconditions, it would be necessary to implement support for magic wand chunk snapshots.

  Extending Silicon's implementation correspondingly is possible: the rule for `applying` would be similar to that for `unfolding` (Figure 3.15), and would require an analogous joining of local execution branches; and integrating magic wands and their chunks' snapshots into Silicon's technique for axiomatising functions would require recording a similar mapping from (syntactic) magic wand assertions to the corresponding wand chunks, as is already done for fields and predicates (Section 3.6).

  Since wand chunk snapshots are functions — as are the snapshots of quantified chunks (recall Chapter 4 and in particular Section 4.3.2) — and because SMT solvers typically do not support higher-order functions, it would again be necessary to apply defunctionalisation [107] in order to encode heap-dependent functions whose preconditions require magic wand instances.
- Fractional wand instances are currently not supported by Silicon, but an extension is possible: besides adapting Viper's grammar accordingly (for example, to support appropriate accessibility predicates of the shape `acc(`*a* `--*` *g*`,` *p*`)`) and to recording symbolic permission expressions in magic wand chunks, the main change would affect the use of magic wand chunk snapshots and of branch conditions that mention values from the hypothetical left-hand side. Recall (from Section 5.4) that wand chunk snapshots are functions of the actually provided left-hand side (Section 5.4.1) and that branch conditions that mention values from the hypothetical left-hand side can be understood analogously, and that the current rules for applying magic wand instances (Section 5.4.2) does not (need to) perform proper instantiations of snapshots and branch conditions (and instead equates the formal and the actual function arguments) exactly because each wand chunk can only be "applied" once. In order to support fractional wand instances, it would be necessary to properly instantiate wand chunk snapshots and branch conditions.

5.8 Related Work

Lee and Park have recently developed a proof system for a separation logic supporting the magic wand connective [81], and also provided a decision procedure for propositional separation logic (that is, without variables). In a richer logic such as Viper's, however, the magic wand is known to be undecidable [29]. Our technique addresses this difficulty with the combination of `apply` and `package` annotations (which can often be inferred by simple heuristics, see Section 5.5), along with novel algorithms for computing appropriate magic wand footprints automatically.

In parallel with the work presented here, Blom and Huisman have developed support for magic wands in their VerCors verifier [17]. VerCors verifies Java programs enriched with separation-logic-style specifications, and magic wands are eliminated during the

translation by a clever encoding into ghost data ("witness objects") that represents magic wand instances. This translation is automatic, but (similar to our work) also requires annotations to direct the creation and use of magic wands. In contrast to our approach, the user must also manually specify annotations defining the permissions and logical facts to be used from the current state for each wand's footprint, which are then combined to show the wand's right-hand side via arbitrary user-defined ghost code. The ability to use arbitrary code is potentially more flexible than the fixed set of ghost operations supported by Viper (for example, ghost methods could be employed), but the resulting annotation overhead is significantly higher than with the automatic footprint computation presented in this chapter (even comparing without the additional heuristics described in Section 5.5). Moreover, their translation does not support nested wands such as $a_1 \mathrel{-\!\!*} (a_2 \mathrel{-\!\!*} a_3)$ or wands inside predicate definitions (although it might be possible to extend their approach accordingly). It is also unclear how efficient the resulting encoding into ghost data is in practice, as illustrated by the verification challenge example discussed in Section 5.6, whose verification takes 60 seconds in VerCors but only 0.5s in Silicon.

In the context of a permission-based type system, Boyland [27] has defined a "sceptre" operator to represent "borrowing" of permission. This connective is more restricted than the general magic wand, but is sufficient for many loop invariants, such as the one of this chapter's running example. The PhD thesis of Retert [106] provides an abstract-interpretation-based approach supporting this connective.

The specific problem of rewriting and maintaining appropriate predicate definitions during data structure traversals has already received much attention. Without an alternative to simple `fold`/`unfold` annotations, one needs to define a new predicate type to represent "partial" versions of the data structure, and write ghost methods to "append" to this partial version, as well as to rewrite it into the original predicate once the traversal is completed. The problems of tracking suitable permissions in loop invariants are discussed in detail by Tuerk [130], who proposed alternative pre-/postcondition specifications for loops. A magic wand of the form $pre_{rest} * (post_{rest} \mathrel{-\!\!*} post_{all})$ gives an alternate expression of his idea (where "*rest*" refers to the remaining loop iteration, and "*all*" to the entire loop). Making use of magic wands is more general than Tuerk's proof rule, for example when further code after the loop is needed before restoring the overall predicate, as in the tree-min-delete challenge (Section 5.6).

A variety of existing work aims to reduce the annotation overhead associated with managing and rewriting predicate definitions with explicit `fold` and `unfold` operations. For example, Smallfoot [11] and GRASShopper [103] achieve concise specifications without user direction by building in specific support for list and tree predicates. Lee et al. [80] provide a static analysis capable of identifying when objects participate in many such data structures simultaneously. Nguyen and Chin [98] and Brotherston et al. [31] provide techniques for proving and applying user-supplied lemmas automatically. Chin et al. [35] provide support for a wider class of predicate definitions, including functional abstractions of data structures, provided that exactly one reference parameter is traversed in the predicate's definition. Their entailment checker "carves out" a suitable portion of the input heap, which (for one input heap) is similar to the operation of our footprint computation algorithm.

These techniques improve the usability of recursive predicate reasoning, and can complement our work in a practical tool. Each comes with limitations: they cannot be applied equally to fully general predicates. One consequence of available magic wand support is that iterative code (such as our running example) can be specified without the need for extra predicate types to represent "partial" versions of data structures. These extra predicates do not describe structures which the program operates on, and are cumbersome to define for structures more complex than linked lists; loop invariants employing magic wands can be defined analogously for other data structures, and also support the specification of functional properties (for example, the use of `sum_rec` in our example).

VeriFast [67] is a mature and expressive verifier for programs annotated with separation logic. We believe that it is possible to partly work around the absence of magic wands using VeriFast's support for *lemma function pointers* and predicates. One can encode a wand $a_1 \mathrel{-\!\!*} a_2$, using a predicate F (representing the wand's footprint), and a pointer to a lemma function with precondition $F * a_1$ and postcondition a_2, whose body shows how to rewrite the state (for example, by (un)folding predicates). The need to define the footprint manually, however, entails substantial additional overhead (to define a predicate for each footprint, and the appropriate lemma methods for manipulating them) for the user, compared to automatically computing a footprint.

Chapter 6

Conclusions and Future Work

We presented Viper, the first verification infrastructure for permission-based reasoning, designed with the goal of providing a reusable infrastructure that facilitates the development of automated program verifiers for different programming languages, specification styles and permission logics. In particular, we presented an expressive intermediate verification language and a powerful symbolic-execution-based verifier for this language.

6.1 Concluding Evaluation

Problem Statement

In order to evaluate the overall contributions of this thesis (evaluations of specific aspects of the presented work were shown in Section 2.8, Section 3.7, Section 4.5 and Section 5.6), we revisit the requirements for a verification infrastructure that we derived from our problem statement (see Section 1.1) and evaluate the work we presented against these requirements.

The first requirement we identified was that an intermediate verification language should provide a level of abstraction that enables the encoding of different programming languages and specification styles but does not impede the development of efficient and precise verifiers and static analysers. The key design decisions we made in order to ensure that Viper meets this requirement are: (1) to provide a built-in notion of a heap and of permissions, which facilitates the development of efficient and precise back ends, and (2) to complement higher-level features such as if-then-else statements, loops, methods, predicates and functions with lower-level features such as the novel `inhale` and `exhale` statements (but also, for example, with `goto` statements). This ensures the expressiveness and flexibility necessary for encoding a diverse range of high-level concepts, and it enables encodings that preserve the structural information typically needed by static analysers in order to compute precise results.

Viper's expressiveness has been demonstrated in different ways: (1) by the existence of several front ends for different programming languages that implement different high-level concepts (a list of Viper front ends is presented shortly), (2) by presenting possible encodings of different concurrency features, such as lock synchronisation and asynchronous method calls, (3) by specifying and verifying a wide range of properties, including the shape of data structures and their functional behaviour, the absence of resource leaks, and deadlock freedom, (4) by discussing different specification styles, for example, specifications based solely on predicates and specifications that combine predicates, magic wands and abstraction functions, and (5) by enabling specifications that have been used for years in by-hand proofs but so far were beyond the scope of automated verifiers, such as specifications based on iterated separating conjunctions or magic wands.

Moreover, with the development of Silicon we demonstrated that Viper's expressiveness does not impede the development of efficient and precise back ends: Silicon in general exhibits good and predictable performance, and a degree of automation and completeness that is higher than that of other symbolic-execution-based verifiers. Silicon has been carefully engineered to achieve this challenging combination by, among other things, (1) complementing cheap, greedy algorithms with more expensive, lazily-triggered state consolidation algorithms that overcome certain kinds of heap-related incompletenesses without additional directions from users, (2) carefully encoding proof obligations to the underlying SMT solver; in particular, by controlling the number of potential quantifier instantiations, and (3) implementing a variety of technical optimisations such as heuristically ordering the inputs of some algorithms and memoizing the results of others. In addition to developing two verifiers for Viper, our research group is also working on a permission inference system based on abstract interpretation: initial results of this line of work indicate that Viper programs are also amenable to precise static analyses.

Another requirement that was derived from the problem statement is that an intermediate verification language should enable users to declare and specify arbitrary heap-implemented data structures as well as a wide range of mathematical structures (for example, to serve as abstractions of the heap-implemented ones), and that it should be possible for users to express the necessary definitions directly on the level of the intermediate verification language: Viper meets this requirement as well.

A third requirement we identified was the suitability for manually encoding examples, for instance, when experimenting with alternative encodings and for educational purposes. We argue that Viper is suitable for such applications as well: its combination of lower- and higher-level language constructs, advanced features such as the automatic computation of magic wand footprints, and design decisions such as supporting mathematical data structures via uninterpreted functions and axioms (which increases the degree of automation) achieve a good balance between expressiveness and conciseness. So far, we have used Viper (without a front end) for giving various demos, in the verification competition VerifyThis@ETAPS'16 [62] and during lectures and assignments at the Marktoberdorf Summer School 2016 [105], and we consider the successful outcomes and the positive feedback we received as anecdotal evidence supporting our claim. We also received positive feedback for how verification failures are reported, but did not perform a systematic evaluation which would allow drawing sound conclusions with respect to how helpful Viper's error reporting is.

Another requirement that we derived from our problem statement is that potential automation limitations should be understandable on the level of the intermediate verification language and not require detailed knowledge about the verifier's implementation. Regarding Viper and Silicon, we conclude that this requirement is largely fulfilled: most automation limitations follow from deliberately made and theoretically motivated design decisions, such as the need for `(un)fold/package/apply` statements and manually encoding proofs by induction as a consequence of the logical complexity of recursive definitions, magic wands and inductive theories, respectively. The only exceptions we are currently aware of (ignoring incompletenesses that originate from the SMT solver, for example, when reasoning about undecidable, built-in theories such as non-linear integer arithmetic) arise from (1) incomplete axiomatisations of background theories such as sets, sequences and multisets; these incompletenesses are exhibited by both Viper verifiers, and (2) Silicon's incomplete management of the symbolic heap (that is, the remaining incompletenesses that are not overcome by Silicon's lazily triggered state consolidations). Both kinds of incompletenesses can be overcome by adding additional assertions, but the evaluation of Silicon (Section 3.7) indicates that this is not often necessary in practice.

The last requirements we derived are related to performance: a verifier should be sufficiently fast such that it can be used in an IDE-like manner to continuously verify programs in the background, and possible performance limitations (if any) should be understandable on the level of the intermediate verification language. We consider

the first aspect as met: the performance evaluations presented in this thesis show that Silicon performance well in general and can therefore indeed be used as the background verifier of a verification IDE (a corresponding plug-in for VSCode [90] will be released soon), and they also show that Silicon's performance is stable across succeeding and failing verifications. The evaluation of Silicon with respect to the second aspect is less clear, however: the only major performance issue we are currently aware of is due to repeated applications of heap-dependent functions that use quantified permissions in their specifications, but the resulting performance degradation can only be explained on the level of Silicon's implementation (and is not exhibited by Viper's other, verification-condition-generation-based verifier). However, we believe that it is possible to overcome this problem (as discussed in Section 4.3.2), and plan to do so in future work.

Impact

The Viper verification infrastructure already had noteworthy impact that goes beyond published papers and that demonstrates the significance of the work presented in this thesis: (1) Viper was used in the VerifyThis@ETAPS'16 verification competition and won the *Distinguished User-Assistance Tool Feature* award for quantified permissions, and during lectures and student assignments at the Marktoberdorf Summer School 2016, (2) Viper is already being used at two universities other than ETH, (3) six Viper front ends have been developed so far and the development of two more has recently been started, and (4) the work on two additional Viper tools, an alternative back end and a permission inference, has recently yielded first results.

Of the six existing Viper front ends, we consider four as comprehensive front ends (and the remaining as prototypical), two of which have been developed by our group: a front end for (an extended version of) Chalice [75, 89] that encodes various concurrency features and related properties such as finite blocking, and a front end for Python programs that is part of the SCION project [8] for developing a secure, next-generation internet architecture. The other two comprehensive front ends are part of the VerCors project [16] developed at the University of Twente: a front end for Java programs and another one for programs written in OpenCL. Since the front ends support different programming languages, properties and specification styles, they typically also generate encodings that use different Viper features to different degrees. For example, the OpenCL front end makes heavy use of quantified permissions (to encode matrices), whereas the Chalice front end relies on Viper's permission introspection features (to reason about finite blocking). The two prototypical front ends have been developed in order to experiment with encodings of particular language features: a Scala front end [30, 50] to experiment with verifying lazily executed code, and a Rust front end [53] to explore the possibility of inferring permission annotations from Rust's ownership-based type system. The development of two further front ends has recently been started: a front end for investigating potential encodings of relaxed separation logic for C11 programs [132], developed by our group, and a C# front end [100], developed at the Charles University in Prague.

The Viper verification infrastructure also enables the development of verification tools other than front ends: so far, of an experimental back end [57] that encodes Viper programs to the GRASShopper verifier [103], and an abstract-interpretation-based inference for quantified permissions [134]; both tools are work in progress, but first results are promising.

6.2 Future Work

As future work, we suggest to investigate the potential for using Viper in order to develop automated verifiers for recently published, highly-specialised separation logics such as TaDA [109], tailored towards the verification of lock-free concurrent code

that uses atomic instructions (such as compare-and-swap), and LiLi [86], developed for proving safety and liveness properties of concurrent objects. In by-hand proofs, such logics have been used to prove substantial properties of intricate programs, but the proofs quickly become unmanageable due to the number of required proof steps and side conditions. So far, no automated verifiers for these logics exist, which is problematic for two reasons: (1) applying the logics to programs longer than a dozen lines of code can be very cumbersome, which impedes understanding potential shortcomings of the logics and identifying areas of further research, and (2) establishing confidence in the correctness of manually constructed proofs is difficult, even for proofs about very short programs. As a first step towards building automated verifiers for such logics we suggest to develop automated proof outline checkers, that is, tools that take as input a substantially simplified proof outline (similar to those that are typically provided in the papers that present the logics) and that automatically fill in the omitted proof steps and verify the proof outline.

In addition to new front ends concerned with automating proofs in specialised separation logics, we also suggest to develop front ends in order to experiment with specifying and verifying code that follows certain "modern" programming paradigms that gained substantial popularity in recent years, in particular actor-based concurrency (for building distributed systems) and reactive programming (for building dataflow-centric systems and user interfaces). Both paradigms originate from functional programming languages with referentially transparent computations, a restriction that simplifies reasoning about the event-based, non-local nature of the style of programming these paradigms advocate. However, the paradigms are also increasingly being used in imperative languages, for example, JavaScript, Scala and C#, where the paradigms are typically implemented in libraries that make heavy use of closures for modelling event-based computations. Reasoning about imperative code that follows these paradigms is significantly more difficult than reasoning about corresponding functional code because the former may use mutable shared state, for example, to improve performance. We expect that the development of appropriate front ends will pose several research challenges, such as designing suitable specification languages; additionally, the development might uncover limitations of Viper's intermediate language (and thus result in language extensions that enable or facilitate desirable encodings) and its back ends (for example, performance or completeness issues of the verifiers).

Silicon also presents various opportunities for future work, with challenges ranging from mainly theoretical to mostly technical. On the theoretical side, one could increase the confidence in the correctness of Silicon's symbolic execution rules, for example, by proving soundness of the computation and use of snapshots; as discussed in Section 5.4.4, this is expected to require an alternative, less syntactical representation of snapshots. In more substantial work, one could also identify a core subset of Viper and Silicon, and attempt to mechanise soundness proofs for this subset, as was done for Featherweight VeriFast [133]. In order to increase confidence in Silicon's implementation — correctness of which would need to be proven independently of any soundness proofs about Silicon's formalisation — one could explore the possibility of testing Silicon's implementation against the semantics of Viper (which are currently being formalised). To that end, one could take the operational semantics for Viper and mechanise them in a way that (1) does not require much encoding effort and thus minimises the risk of introducing bugs during the mechanisation, and (2) yields an executable version of the semantics. A candidate for such a mechanisation might be the K framework [112]. The executable semantics could then serve as a test oracle: for a concrete input state and a piece of Viper code, the executable semantics would determine the expected successor state(s), and after setting up a symbolic state that corresponds to the concrete state and symbolically executing the code, one could check (for example, via an appropriate encoding into SMT) if the symbolic output state corresponds to the concrete output state(s).

Another possible direction of future work is to explore if Silicon would benefit from implementing Jahob's [137] approach of decomposing proof obligations into simpler sub-obligations that fall into specific logical fragments and can be handled by specialised solvers, or according to some other classification, such as whether or not a fact is concerned with permissions. For example, we occasionally observed that the verification time (of both Viper verifiers) increases if fixed fractions (such as $1/2$ and $1/4$) are replaced by symbolic fractions and appropriate inequalities (such as p_1, p_2 and $p_2 < p_1$); such inequalities are also generated during the translation of Chalice's abstract read permissions [58] into Viper. In the presence of predicates, the inequalities may also include non-linear arithmetic (such as $p_3 < p_1 p_2$) as a result of fractionally unfolding predicates. In order to check for satisfiability of arithmetic inequalities, SMT solvers typically try to find satisfying assignments, whereas it would suffice to "just" know that the inequalities are satisfiable. It might therefore improve performance if a different solver were used for permission queries. A related optimisation might be to "hide" permission-related facts from the SMT solver when checking functional program properties: the latter are often independent of permission-related facts (ignoring, for example, reference disequalities), in which case the solver would needlessly take them into consideration (by trying to find satisfying assignments).

As future work with a focus on software engineering one could devise an architecture and a corresponding API for Silicon that facilitates the integration and orchestration of "actions" which are executed in response to certain events, in particular to verification failures. Such an architecture should enable the implementation of the two already existing (and currently hard-coded) actions — performing state consolidations, and heuristically (un)folding predicates and packaging/applying wands — as separate "plug-ins". The architecture should also be flexible and facilitate declaring, for instance, when an action is triggered (for example, by any verification failure, by insufficient permissions only, or simply periodically), to which extent already performed symbolic execution steps are to be re-executed afterwards (for example, only the previously failing one or all steps after the last inhaling of permissions) and in which order alternative actions should be tried. Using this architecture it should be possible to declare, for example, a chain of actions such that a state consolidation is performed in response to insufficient permissions, after which the failing assertion is checked again and if it still fails, the heap management is switched from Silicon's cheap but incomplete default algorithms to the complete but more expensive quantified-permission-style algorithms, and the assertion is checked once more.

Chapter 7

Bibliography

[1] W. Ahrendt et al. 'The KeY platform for verification and analysis of Java programs '. In: *VSTTE*. Vol. 8471. LNCS. Springer, 2014, pp. 55–71 (cit. on p. 8).

[2] A. Amighi et al. 'Specification and verification of atomic operations in GPGPU programs '. In: *SEFM*. Vol. 9276. LNCS. Springer, 2015, pp. 69–83 (cit. on pp. 8, 122).

[3] N. Amin, K. R. M. Leino and T. Rompf. 'Computing with an SMT solver '. In: *TAP*. Vol. 8570. LNCS. Springer, 2014, pp. 20–35 (cit. on p. 88).

[4] T. Ball, V. Levin and S. K. Rajamani. 'A decade of software model checking with SLAM '. In: *Commun. ACM* 54.7 (2011), pp. 68–76 (cit. on p. 1).

[5] A. Banerjee, D. A. Naumann and S. Rosenberg. 'Regional logic for local reasoning about global invariants '. In: *ECOOP*. Vol. 5142. LNCS. Springer, 2008, pp. 387–411 (cit. on p. 8).

[6] M. Barnett et al. 'Boogie: A modular reusable verifier for object-oriented programs '. In: *FMCO*. Vol. 4111. LNCS. Springer, 2005, pp. 364–387 (cit. on pp. 2, 3, 5, 10, 13, 14, 44, 93).

[7] M. Barnett et al. 'Specification and verification: The Spec# experience '. In: *Commun. ACM* 54.6 (2011), pp. 81–91 (cit. on pp. 9, 34).

[8] D. Barrera et al. 'SCION five years later: Revisiting scalability, control, and isolation on next-generation networks '. In: *CoRR* abs/1508.01651 (2015) (cit. on pp. 13, 167).

[9] C. Barrett, P. Fontaine and C. Tinelli. *The SMT-LIB standard: Version 2.5*. Tech. rep. Available at `www.smt-lib.org`. Department of Computer Science, The University of Iowa, 2015 (cit. on p. 38).

[10] C. Barrett et al. 'CVC4 '. In: *CAV*. Vol. 6806. LNCS. Springer, 2011, pp. 171–177 (cit. on p. 38).

[11] J. Berdine, C. Calcagno and P. W. O'Hearn. 'Smallfoot: modular automatic assertion checking with separation logic '. In: *FMCO*. Vol. 4111. LNCS. Springer, 2005, pp. 115–137 (cit. on pp. 6, 11, 28, 50, 95, 127, 163).

[12] J. Berdine, C. Calcagno and P. W. O'Hearn. 'Symbolic execution with separation logic '. In: *APLAS*. Vol. 3780. LNCS. Springer, 2005, pp. 52–68 (cit. on p. 49).

[13] J. Bertrane et al. 'Static analysis and verification of aerospace software by abstract interpretation '. In: *Foundations and Trends in Programming Languages* 2.2-3 (2015), pp. 71–190 (cit. on p. 10).

[14] L. Birkedal, N. Torp-Smith and J. C. Reynolds. 'Local reasoning about a copying garbage collector '. In: *POPL*. ACM, 2004, pp. 220–231 (cit. on pp. 31, 103).

[15] R. Blanc et al. 'An overview of the Leon verification system: verification by translation to recursive functions '. In: *SCALA*. ACM, 2013, 1:1–1:10 (cit. on p. 10).

[16] S. Blom and M. Huisman. 'The VerCors tool for verification of concurrent programs '. In: *FM*. Vol. 8442. LNCS. Springer, 2014, pp. 127–131 (cit. on pp. 7, 13, 42, 93, 106, 167).

[17] S. Blom and M. Huisman. 'Witnessing the elimination of magic wands '. In: *STTT* 17.6 (2015), pp. 757–781 (cit. on pp. 7, 30, 127, 160–162).

[18] F. Bobot et al. 'Why3: Shepherd your herd of provers '. In: *Boogie*. 2011, pp. 53–64 (cit. on pp. 2, 3, 5, 45).

[19] S. Böhme and M. Moskal. 'Heaps and data structures: A challenge for automated provers '. In: *CADE*. Vol. 6803. LNCS. Springer, 2011, pp. 177–191 (cit. on p. 44).

[20] R. Bornat et al. 'Permission accounting in separation logic '. In: *POPL*. ACM, 2005, pp. 259–270 (cit. on p. 46).

[21] P. Boström and P. Müller. 'Modular verification of finite blocking in non-terminating programs '. In: *ECOOP*. Vol. 37. LIPIcs. Schloss Dagstuhl, 2015, pp. 639–663 (cit. on p. 25).

[22] M. Botincan, M. J. Parkinson and W. Schulte. 'Separation logic verification of C programs with an SMT solver '. In: *Electr. Notes Theor. Comput. Sci.* 254 (2009), pp. 5–23 (cit. on p. 6).

[23] M. Botinčan et al. 'coreStar: The core of jStar '. In: *Boogie*. 2011, pp. 65–77 (cit. on p. 6).

[24] R. S. Boyer, B. Elspas and K. N. Levitt. 'SELECT — A formal system for testing and debugging programs by symbolic execution '. In: *Proceedings of the International Conference on Reliable Software*. ACM, 1975, pp. 234–245 (cit. on p. 49).

[25] J. Boyland. 'Checking interference with fractional permissions '. In: *SAS*. Vol. 2694. LNCS. Springer, 2003, pp. 55–72 (cit. on pp. 3, 10, 17, 18, 22, 46).

[26] J. Boyland. 'Fractional permissions '. In: *Aliasing in Object-Oriented Programming*. Vol. 7850. LNCS. Springer, 2013, pp. 270–288 (cit. on p. 46).

[27] J. T. Boyland. 'Semantics of fractional permissions with nesting '. In: *TOPLAS* 32.6 (2010) (cit. on p. 163).

[28] J. T. Boyland et al. 'Constraint semantics for abstract read permissions '. In: *FTfJP*. ACM, 2014, 2:1–2:6 (cit. on pp. 10, 17, 42, 46, 47).

[29] R. Brochenin, S. Demri and É. Lozes. 'On the almighty wand '. In: LNCS 5213 (2008), pp. 323–338 (cit. on pp. 127, 162).

[30] B. Brodowsky. 'Translating Scala to SIL '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2013 (cit. on p. 167).

[31] J. Brotherston, D. Distefano and R. L. Petersen. 'Automated cyclic entailment proofs in separation logic '. In: *CADE*. Vol. 6803. LNCS. Springer, 2011, pp. 131–146 (cit. on p. 163).

[32] L. Brutschy, P. Ferrara and P. Müller. 'Static analysis for independent app developers '. In: *OOPSLA*. ACM, 2014, pp. 847–860 (cit. on p. 10).

[33] C. Cadar and K. Sen. 'Symbolic execution for software testing: Three decades later '. In: *Commun. ACM* 56.2 (2013), pp. 82–90 (cit. on p. 49).

[34] C. Calcagno et al. 'Moving fast with software verification '. In: *NFM*. Vol. 9058. LNCS. Springer, 2015, pp. 3–11 (cit. on pp. 1, 10).

[35] W. Chin et al. 'Automated verification of shape, size and bag properties via user-defined predicates in separation logic '. In: *Sci. Comput. Program.* 77.9 (2012), pp. 1006–1036 (cit. on pp. 7, 96, 163).

[36] D. Chu and J. Jaffar. 'Local reasoning with first-class heaps, and a new frame rule '. In: *CoRR* abs/1511.07267 (2015) (cit. on p. 7).

[37] E. Cohen et al. 'Local verification of global invariants in concurrent programs '. In: *CAV*. Vol. 6174. LNCS. Springer, 2010, pp. 480–494 (cit. on p. 23).

[38] E. Cohen et al. 'VCC: A practical system for verifying concurrent C '. In: *TPHOLs*. Vol. 5674. LNCS. Springer, 2009, pp. 23–42 (cit. on p. 9).

[39] D. Detlefs, G. Nelson and J. B. Saxe. 'Simplify: A theorem prover for program checking '. In: *J. ACM* 52.3 (2005), pp. 365–473 (cit. on pp. 16, 38, 49).

[40] W. Dietl and P. Müller. 'Object ownership in program verification '. In: *Aliasing in Object-Oriented Programming*. Vol. 7850. LNCS. Springer, 2013, pp. 289–318 (cit. on p. 9).

[41] E. W. Dijkstra. 'Guarded commands, nondeterminacy and formal derivation of programs '. In: *Commun. ACM* 18.8 (1975), pp. 453–457 (cit. on p. 49).

[42] D. Distefano and M. J. Parkinson. 'jStar: Towards practical verification for Java '. In: *OOPSLA*. ACM, 2008, pp. 213–226 (cit. on pp. 6, 49, 50, 95, 103, 127).

[43] R. Dockins, A. Hobor and A. W. Appel. 'A fresh look at separation algebras and share accounting '. In: *APLAS*. Vol. 5904. LNCS. Springer, 2009, pp. 161–177 (cit. on p. 46).

[44] M. Dodds, S. Jagannathan and M. J. Parkinson. 'Modular reasoning for deterministic parallelism '. In: *POPL*. ACM, 2011, pp. 259–270 (cit. on pp. 30, 127, 131).

[45] M. D. Ernst et al. 'Semantics for locking specifications '. In: *CoRR* abs/1501.05338 (2015) (cit. on p. 23).

[46] P. Ferrara, F. Logozzo and M. Fähndrich. 'Safer unsafe code for .NET '. In: *OOPSLA*. ACM, 2008, pp. 329–346 (cit. on p. 10).

[47] J. Filliâtre, L. Gondelman and A. Paskevich. 'The spirit of ghost code '. In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 1–16 (cit. on pp. 16, 45).

[48] R. W. Floyd. 'Nondeterministic algorithms '. In: *J. ACM* 14.4 (1967), pp. 636–644 (cit. on p. 149).

[49] D. P. Friedman and M. Wand. *Essentials of programming languages (3. ed.)* MIT Press, 2008 (cit. on p. 59).

[50] S. Fritsche. 'Verifying Scala's vals and lazy vals '. Bachelor's Thesis. ETH Zurich, Zurich, Switzerland, 2014 (cit. on p. 167).

[51] S. de Gouw et al. 'OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case '. In: *CAV (1)*. Vol. 9206. LNCS. Springer, 2015, pp. 273–289 (cit. on p. 1).

[52] C. Haack and C. Hurlin. 'Resource usage protocols for iterators '. In: *JOT* 8.4 (2009), pp. 55–83 (cit. on pp. 30, 127, 131).

[53] F. Hahn. 'Rust2Viper: building a static verifier for Rust '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2016 (cit. on p. 167).

[54] S. L. Hantler and J. C. King. 'An introduction to proving the correctness of programs '. In: *ACM Comput. Surv.* 8.3 (1976), pp. 331–353 (cit. on p. 49).

[55] C. Hawblitzel et al. 'Ironclad apps: End-to-end security via automated full-system verification '. In: *OSDI*. USENIX Association, 2014, pp. 165–181 (cit. on p. 1).

[56] C. Hawblitzel et al. 'IronFleet: Proving practical distributed systems correct '. In: *SOSP*. ACM, 2015, pp. 1–17 (cit. on p. 1).

[57] A. Helfenstein. 'From Viper to GRASShopper: Translating between intermediate verification languages for software verification '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2016 (cit. on p. 167).

[58] S. Heule et al. 'Abstract read permissions: fractional permissions without the fractions '. In: *VMCAI*. Vol. 7737. LNCS. Springer, 2013, pp. 315–334 (cit. on pp. 10, 17, 41, 47, 100, 169).

[59] S. Heule et al. 'Verification condition generation for permission logics with abstract predicates and abstraction functions '. In: *ECOOP*. Vol. 7920. LNCS. Springer, 2013, pp. 451–476 (cit. on pp. 16, 32, 34, 40, 88, 92, 99).

[60] C. A. R. Hoare. 'Proof of correctness of data representations '. In: vol. 1. 1972, pp. 271–281 (cit. on pp. 4, 34).

[61] Z. Hou et al. 'Proof search for propositional abstract separation logics via labelled sequents '. In: *POPL*. ACM, 2014, pp. 465–476 (cit. on p. 127).

[62] M. Huisman, R. Monahan and P. Müller. *VerifyThis at ETAPS 2016*. 2016. URL: `http://etaps2016.verifythis.org/` (visited on 16/08/2016) (cit. on pp. 104, 166).

[63] M. Huisman, V. Klebanov and R. Monahan. *VerifyThis verification competition 2012 — Organizer's report*. 2012. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000034373` (cit. on p. 160).

[64] S. S. Ishtiaq and P. W. O'Hearn. 'BI as an assertion language for mutable data structures '. In: *POPL*. ACM, 2001, pp. 14–26 (cit. on pp. 12, 127).

[65] B. Jacobs, D. Bosnacki and R. Kuiper. *Modular termination verification: extended version*. CW Reports CW680. Department of Computer Science, KU Leuven, Jan. 2015 (cit. on p. 26).

[66] B. Jacobs et al. 'A programming model for concurrent object-oriented programs '. In: *TOPLAS* 31.1 (2008) (cit. on p. 9).

[67] B. Jacobs et al. 'VeriFast: A powerful, sound, predictable, fast verifier for C and Java '. In: *NFM*. Vol. 6617. LNCS. Springer, 2011, pp. 41–55 (cit. on pp. 6, 16, 47, 49, 50, 92, 95, 127, 159, 164).

[68] J. B. Jensen, L. Birkedal and P. Sestoft. 'Modular verification of linked lists with views via separation logic '. In: *FTfJP*. ACM, 2010, 4:1–4:7 (cit. on pp. 30, 127, 131).

[69] J. Jourdan et al. 'A formally-verified C static analyzer '. In: *POPL*. ACM, 2015, pp. 247–259 (cit. on p. 10).

[70] I. T. Kassios. 'A theory of object oriented refinement '. PhD thesis. University of Toronto, 2006 (cit. on p. 21).

[71] I. T. Kassios. 'Dynamic frames: Support for framing, dependencies and sharing without restrictions '. In: *FM*. Vol. 4085. LNCS. Springer, 2006, pp. 268–283 (cit. on pp. 2, 8).

[72] I. T. Kassios, P. Müller and M. Schwerhoff. 'Comparing verification condition generation with symbolic execution: An experience report '. In: *VSTTE*. Vol. 7152. LNCS. Springer, 2012, pp. 196–208 (cit. on pp. 50, 74).

[73] M. Kaufmann, J. S. Moore and P. Manolios. *Computer-aided reasoning: an approach*. Kluwer Academic Publishers, 2000 (cit. on p. 10).

[74] J. C. King. 'Symbolic execution and program testing '. In: *Commun. ACM* 19.7 (1976), pp. 385–394 (cit. on p. 49).

[75] C. Klauser. 'Translating Chalice into SIL '. Bachelor's Thesis. ETH Zurich, Zurich, Switzerland, 2012 (cit. on pp. 13, 167).

[76] N. R. Krishnaswami. 'Reasoning about iterators with separation logic '. In: *SAVCBS*. ACM, 2006, pp. 83–86 (cit. on pp. 30, 127, 131).

[77] A. Lal and S. Qadeer. 'Powering the Static Driver Verifier using Corral '. In: *SIGSOFT FSE*. ACM, 2014, pp. 202–212 (cit. on p. 1).

[78] X. B. Le, C. Gherghina and A. Hobor. 'Decision procedures over sophisticated fractional permissions '. In: *APLAS*. Vol. 7705. LNCS. Springer, 2012, pp. 368–385 (cit. on p. 46).

[79] G. T. Leavens, A. L. Baker and C. Ruby. 'JML: A notation for detailed design '. In: *Behavioral Specifications of Businesses and Systems*. Vol. 523. The Kluwer International Series in Engineering and Computer Science. Springer, 1999, pp. 175–188 (cit. on p. 34).

[80] O. Lee, H. Yang and R. Petersen. 'Program analysis for overlaid data structures '. In: *CAV*. Vol. 6806. LNCS. Springer, 2011, pp. 592–608 (cit. on p. 163).

[81] W. Lee and S. Park. 'A proof system for separation logic with magic wand '. In: *POPL*. ACM, 2014, pp. 477–490 (cit. on pp. 127, 162).

[82] K. R. M. Leino. 'Dafny: An automatic program verifier for functional correctness '. In: *LPAR*. Vol. 6355. LNCS. Springer, 2010, pp. 348–370 (cit. on pp. 8, 14, 17, 54, 103).

[83] K. R. M. Leino and R. Monahan. 'Reasoning about comprehensions with first-order SMT solvers '. In: *SAC*. ACM, 2009, pp. 615–622 (cit. on pp. 32, 40, 92).

[84] K. R. M. Leino and P. Müller. 'A basis for verifying multi-threaded programs '. In: *ESOP*. Vol. 5502. LNCS. Springer, 2009, pp. 378–393 (cit. on pp. 5, 7, 14, 26, 44, 46, 103, 127, 161).

[85] K. R. M. Leino and P. Müller. 'Object invariants in dynamic contexts '. In: *ECOOP*. Vol. 3086. LNCS. Springer, 2004, pp. 491–516 (cit. on p. 9).

[86] H. Liang and X. Feng. 'A program logic for concurrent objects under fair scheduling '. In: *POPL*. ACM, 2016, pp. 385–399 (cit. on p. 168).

[87] T. Maeda, H. Sato and A. Yonezawa. 'Extended alias type system using separating implication '. In: *TLDI*. ACM, 2011, pp. 29–42 (cit. on pp. 30, 127).

[88] I. Martian Software. *Nailgun: insanely fast Java*. 2012. URL: `http://www.martiansoftware.com/nailgun/` (visited on 23/06/2016) (cit. on pp. 43, 93, 123).

[89] R. Meier. 'Verification of finite blocking in Chalice '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2015 (cit. on pp. 25, 167).

[90] Microsoft. *Visual Studio Code*. 2016. URL: `https://code.visualstudio.com` (visited on 29/08/2016) (cit. on p. 167).

[91] M. Moskal. 'Programming with triggers '. In: *SMT*. Vol. 375. ACM International Conference Proceeding Series. ACM, 2009, pp. 20–29 (cit. on pp. 32, 40).

[92] L. M. de Moura and N. Bjørner. 'Z3: An efficient SMT solver '. In: *TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 337–340 (cit. on pp. 13, 38, 49, 54).

[93] P. Müller. 'Modular specification and verification of object-oriented programs '. PhD thesis. FernUniversität Hagen, 2001 (cit. on pp. 2, 8).

[94] P. Müller and the Viper project team. *Viper project page*. 2016. URL: `http://viper.ethz.ch/` (visited on 16/08/2016) (cit. on pp. 122, 160).

[95] P. Müller, M. Schwerhoff and A. J. Summers. 'Automatic verification of iterated separating conjunctions using symbolic execution '. In: *CAV (1)*. Vol. 9779. LNCS. Springer, 2016, pp. 405–425 (cit. on pp. 27, 104).

[96] P. Müller, M. Schwerhoff and A. J. Summers. 'Viper: A verification infrastructure for permission-based reasoning '. In: *VMCAI*. Vol. 9583. LNCS. Springer, 2016, pp. 41–62 (cit. on pp. 7, 10, 13).

[97] G. Nelson and D. C. Oppen. 'Simplification by cooperating decision procedures '. In: *TOPLAS* 1.2 (1979), pp. 245–257 (cit. on p. 8).

[98] H. H. Nguyen and W. Chin. 'Enhancing program verification with lemmas '. In: *CAV*. Vol. 5123. LNCS. Springer, 2008, pp. 355–369 (cit. on p. 163).

[99] P. W. O'Hearn, J. C. Reynolds and H. Yang. 'Local reasoning about programs that alter data structures '. In: *CSL*. Vol. 2142. LNCS. Springer, 2001, pp. 1–19 (cit. on pp. 2, 6, 15, 30, 127).

[100] P. Parízek and P. Hudeček. *A C# front end for Viper*. 2016. URL: `http://d3s.mff.cuni.cz/people/parizek/` (visited on 29/08/2016) (cit. on p. 167).

[101] M. J. Parkinson and G. M. Bierman. 'Separation logic and abstraction '. In: *POPL*. ACM, 2005, pp. 247–258 (cit. on pp. 3, 27).

[102] M. J. Parkinson and A. J. Summers. 'The relationship between separation logic and implicit dynamic frames '. In: *LMCS* 8.3 (2012) (cit. on pp. 7, 21, 133, 146, 147).

[103] R. Piskac, T. Wies and D. Zufferey. 'GRASShopper — complete heap verification with mixed specifications '. In: *TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 124–139 (cit. on pp. 7, 28, 96, 97, 103, 163, 167).

[104] N. Polikarpova et al. 'Flexible invariants through semantic collaboration '. In: *FM*. Vol. 8442. LNCS. Springer, 2014, pp. 514–530 (cit. on p. 9).

[105] A. Pretschner and D. Peled. *Marktoberdorf summer school 2016*. 2016. URL: `https://sites.google.com/site/marktoberdorf16/` (visited on 29/08/2016) (cit. on p. 166).

[106] W. S. Retert. 'Implementing permission analysis '. PhD thesis. Milwaukee, WI, USA: University of Wisconsin at Milwaukee, 2009 (cit. on p. 163).

[107] J. C. Reynolds. 'Definitional interpreters for higher-order programming languages '. In: *Proceedings of the ACM Annual Conference*. Vol. 2. ACM, 1972, pp. 717–740 (cit. on pp. 116, 162).

[108] J. C. Reynolds. 'Separation logic: A logic for shared mutable data structures '. In: *LICS*. IEEE Computer Society, 2002, pp. 55–74 (cit. on pp. 2, 11, 15, 30, 31, 103, 127).

[109] P. da Rocha Pinto, T. Dinsdale-Young and P. Gardner. 'TaDA: A logic for time and data abstraction '. In: *ECOOP*. Vol. 8586. LNCS. Springer, 2014, pp. 207–231 (cit. on p. 167).

[110] S. Rosenberg, A. Banerjee and D. A. Naumann. 'Decision procedures for region logic '. In: *VMCAI*. Vol. 7148. LNCS. Springer, 2012, pp. 379–395 (cit. on p. 8).

[111] G. Rosu, C. Ellison and W. Schulte. 'Matching logic: An alternative to hoare/-floyd logic '. In: *AMAST*. Vol. 6486. LNCS. Springer, 2010, pp. 142–162 (cit. on p. 9).

[112] G. Rosu and T. Serbanuta. 'An overview of the K semantic framework '. In: *J. Log. Algebr. Program.* 79.6 (2010), pp. 397–434 (cit. on p. 168).

[113] G. Rosu and A. Stefanescu. 'Checking reachability using matching logic '. In: *OOPSLA*. ACM, 2012, pp. 555–574 (cit. on p. 9).

[114] A. Rudich, Á. Darvas and P. Müller. 'Checking well-formedness of pure-method specifications '. In: *FM*. Vol. 5014. LNCS. Springer, 2008, pp. 68–83 (cit. on pp. 82, 91).

[115] S. Sagiv, T. W. Reps and R. Wilhelm. 'Solving shape-analysis problems in languages with destructive updating '. In: *POPL*. ACM Press, 1996, pp. 16–31 (cit. on p. 10).

[116] M. Schwerhoff. 'Symbolic execution for Chalice '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2011 (cit. on p. 49).

[117] M. Schwerhoff and A. J. Summers. 'Lightweight support for magic wands in an automatic verifier '. In: *ECOOP*. Vol. 37. LIPIcs. Schloss Dagstuhl, 2015, pp. 614–638 (cit. on pp. 27, 128, 132).

[118] J. Smans. 'Specification and automatic verification of frame properties for Java-like programs '. PhD thesis. KU Leuven, May 2009 (cit. on pp. 7, 28, 50, 96, 146).

[119] J. Smans, B. Jacobs and F. Piessens. 'Heap-dependent expressions in separation logic '. In: *FMOODS/FORTE*. Vol. 6117. LNCS. Springer, 2010, pp. 170–185 (cit. on pp. 7, 11, 49, 50, 55, 62, 65, 86, 88, 95, 116).

[120] J. Smans, B. Jacobs and F. Piessens. 'Implicit dynamic frames '. In: *TOPLAS* 34.1 (2012), p. 2 (cit. on pp. 92, 96, 103).

[121] J. Smans, B. Jacobs and F. Piessens. 'Implicit dynamic frames: Combining dynamic frames and separation logic '. In: *ECOOP*. Vol. 5653. LNCS. Springer, 2009, pp. 148–172 (cit. on pp. 7, 10, 14, 16, 28, 50).

[122] J. Smans, B. Jacobs and F. Piessens. 'VeriCool: An automatic verifier for a concurrent object-oriented language '. In: *FMOODS*. Vol. 5051. LNCS. Springer, 2008, pp. 220–239 (cit. on p. 96).

[123] W. Sonnex, S. Drossopoulou and S. Eisenbach. 'Zeno: an automated prover for properties of recursive data structures '. In: *TACAS*. Vol. 7214. LNCS. Springer, 2012, pp. 407–421 (cit. on p. 10).

[124] B. Steensgaard. 'Points-to analysis in almost linear time '. In: *POPL*. ACM Press, 1996, pp. 32–41 (cit. on p. 10).

[125] A. Stefanescu. 'MatchC: A matching logic reachability verifier using the K framework '. In: *Electr. Notes Theor. Comput. Sci.* 304 (2014), pp. 183–198 (cit. on p. 9).

[126] A. J. Summers and S. Drossopoulou. 'A formal semantics for isorecursive and equirecursive state abstractions '. In: *ECOOP*. Vol. 7920. LNCS. Springer, 2013, pp. 129–153 (cit. on pp. 16, 146, 147).

[127] A. J. Summers and S. Drossopoulou. 'Considerate reasoning and the composite design pattern '. In: *VMCAI*. Vol. 5944. LNCS. Springer, 2010, pp. 328–344 (cit. on p. 9).

[128] G. Tassey. *The economic impacts of inadequate infrastructure for software testing*. 2002 (cit. on p. 1).

[129] The Washington Post. *Nissan vehicles recalled*. 2014. URL: `http://www.washingtonpost.com/cars/989701-nissan-vehicles-recalled-altima-leaf-pathfinder-sentra-and-infiniti-jx35-q50-qx60/2014/03/26/cc7aa1b4-b504-11e3-bab2-b9602293021d_story.html` (visited on 31/08/2016) (cit. on p. 1).

[130] T. Tuerk. 'Local reasoning about while-loops '. In: *VS-Theory*. 2010 (cit. on pp. 30, 127, 163).

[131] *Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia*. Report GAO/IMTEC-92-26. Washington, D.C.: Information Management and Technology Division, United States General Accounting Office, Feb. 1992, p. 16 (cit. on p. 1).

[132] V. Vafeiadis and C. Narayan. 'Relaxed separation logic: A program logic for C11 concurrency '. In: *OOPSLA*. ACM, 2013, pp. 867–884 (cit. on p. 167).

[133] F. Vogels, B. Jacobs and F. Piessens. 'Featherweight VeriFast '. In: *CoRR* abs/1507.07697 (2015) (cit. on p. 168).

[134] S. Walter. 'Automatic inference of quantified permissions by abstract interpretation '. Master's Thesis. ETH Zurich, Zurich, Switzerland, 2016 (cit. on p. 167).

[135] C. Walther and S. Schweitzer. 'About VeriFun '. In: *CADE*. Vol. 2741. LNCS. Springer, 2003, pp. 322–327 (cit. on p. 10).

[136] H. Yang. 'An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm '. In: *SPACE*. 2001 (cit. on pp. 31, 103, 127, 131).

[137] K. Zee, V. Kuncak and M. C. Rinard. 'Full functional verification of linked data structures '. In: *PLDI*. ACM, 2008, pp. 349–361 (cit. on pp. 8, 169).

Appendix A

Viper

```
/* An encoding of a linked list using predicates and
 * abstraction functions, and a client of the list.
 */

/****************** Nodes ******************/

field data: Int
field next: Ref

predicate lseg(this: Ref, end: Ref) {
  this != end ==>
       acc(this.data) && acc(this.next)
    && lseg(this.next, end)
    && unfolding lseg(this.next, end) in
         this.next != end ==> this.data <= this.next.data
}


/* The additional postcondition (the first two ensures-clauses)
 * are required by Silicon because of a technical limitation:
 * the body of the recursive invocation of lsegContent is not
 * available to the verifier when proving the last
 * postcondition, which therefore fails.
 * Overcoming this limitation is possible, e.g. by following
 *   A. Rudich, A. Darvas and P. Mueller: Checking Well-
 *   Formedness of Pure-Method Specifications, FM'08
 * and planed as future work.
 */
function lsegContent(this: Ref, end: Ref): Seq[Int]
  requires lseg(this, end)
  ensures  this == end ==>
             result == Seq[Int]()
  ensures  this != end ==>
             result[0] == unfolding lseg(this, end) in
                                this.data
  ensures  forall i: Int, j: Int ::
             0 <= i && i < j && j < |result| ==>
                 result[i] <= result[j]
{
  this == end
    ? Seq[Int]()
    : unfolding lseg(this, end) in
          Seq(this.data) ++ lsegContent(this.next, end)
}

function lsegLength(this: Ref, end: Ref): Int
  requires lseg(this, end)
  ensures  result == |lsegContent(this, end)|
{
  unfolding lseg(this, end) in
  this == end ? 0 :
```

```
52                 1 + lsegLength(this.next, end)
53 }
54
55 /****************** Lists *****************/
56
57 field head: Ref
58
59 predicate List(this: Ref) {
60   acc(this.head) && lseg(this.head, null)
61 }
62
63 function content(this: Ref): Seq[Int]
64   requires List(this)
65   ensures  forall i: Int, j: Int ::
66               0 <= i && i < j && j < |result| ==>
67                 result[i] <= result[j]
68 {
69   unfolding List(this) in lsegContent(this.head, null)
70 }
71
72 function length(this: Ref): Int
73   requires List(this)
74   ensures result == |content(this)|
75 {
76   unfolding List(this) in lsegLength(this.head, null)
77 }
78
79 function peek(this: Ref): Int
80   requires List(this)
81   requires 0 < length(this)
82   ensures result == content(this)[0]
83 {
84   unfolding List(this) in unfolding lseg(this.head, null) in
85     this.head.data
86 }
87
88 method create() returns (this: Ref)
89   ensures List(this)
90   ensures content(this) == Seq[Int]()
91 {
92   this := new(*)
93   this.head := null
94   fold lseg(this.head, null)
95   fold List(this)
96 }
97
98 method concat(this: Ref, ptr: Ref, end: Ref)
99   requires lseg(this, ptr)
100  requires lseg(ptr, end)
101  requires end != null ==> acc(end.next, 1/2)
102  requires    0 < |lsegContent(this, ptr)|
103           && 0 < |lsegContent(ptr, end)|
104        ==>
105             lsegContent(this, ptr)[|lsegContent(this, ptr)|-1]
106          <= lsegContent(ptr, end)[0]
107  ensures  lseg(this, end)
108  ensures lsegContent(this, end) ==
109            old(lsegContent(this, ptr) ++ lsegContent(ptr, end))
110  ensures end != null ==> acc(end.next, 1/2)
111 {
112   if(this != ptr)
113   {
114     unfold lseg(this, ptr)
115     concat(this.next, ptr, end)
116     fold lseg(this, end)
117   }
```

```
118 }
119
120 method insert(this: Ref, elem: Int) returns (index: Int)
121   requires List(this)
122   ensures List(this)
123   ensures 0 <= index && index <= |old(content(this))|
124   ensures content(this) ==
125               old(content(this))[0..index]
126             ++ Seq(elem)
127             ++ old(content(this))[index..]
128 {
129   var tmp: Ref
130   index := 0
131
132   unfold List(this)
133
134   if(this.head != null)
135   {
136     unfold lseg(this.head, null)
137   }
138   if(this.head == null || elem <= this.head.data)
139   {
140     tmp := new(*)
141     tmp.data := elem
142     tmp.next := this.head
143     fold lseg(this.head, null)
144     fold lseg(tmp, null)
145     this.head := tmp
146   }
147   else
148   {
149     var ptr: Ref := this.head
150     fold lseg(this.head, ptr)
151     index := index + 1
152     while(   ptr.next != null
153           && unfolding lseg(ptr.next, null) in
154                 ptr.next.data < elem)
155       invariant acc(this.head)
156       invariant acc(ptr.next) && acc(ptr.data)
157       invariant ptr.data <= elem
158       invariant lseg(ptr.next, null)
159       invariant lseg(this.head, ptr)
160       invariant
161           forall i: Int ::
162                   0 <= i && i < |lsegContent(this.head, ptr)|
163               ==> lsegContent(this.head, ptr)[i] <= ptr.data
164       invariant
165           forall i: Int ::
166                   0 <= i && i < |lsegContent(ptr.next, null)|
167               ==> ptr.data <= lsegContent(ptr.next, null)[i]
168       invariant index-1 == |lsegContent(this.head, ptr)|
169       invariant
170           old(content(this)) ==
171                lsegContent(this.head, ptr)
172             ++ Seq(ptr.data)
173             ++ lsegContent(ptr.next, null)
174     {
175       unfold lseg(ptr.next, null)
176       index := index + 1
177       var ptrn: Ref := ptr.next
178       fold lseg(ptrn, ptrn)
179       fold lseg(ptr, ptrn)
180       concat(this.head, ptr, ptrn)
181       ptr := ptrn
182     }
183
```

```
184       tmp := new(*)
185       tmp.data := elem
186       tmp.next := ptr.next
187       ptr.next := tmp
188       fold lseg(ptr.next, null)
189
190       fold lseg(ptr, null)
191       concat(this.head, ptr, null)
192     }
193
194     fold List(this)
195   }
196
197   method dequeue(this: Ref) returns (res: Int)
198     requires List(this)
199     requires 0 < length(this)
200     ensures List(this)
201     ensures res == old(content(this)[0])
202     ensures content(this) == old(content(this)[1..])
203   {
204     unfold List(this)
205     unfold lseg(this.head, null)
206     res := this.head.data
207     this.head := this.head.next
208     fold List(this)
209   }
210
211   /****************** Client *****************/
212
213   // Monitor invariant:
214   //   List(this) && length(this) <= old(length(this))
215   field held: Int
216
217   method test(mon: Ref)
218     // Check that all monitors have been released
219     ensures [true, forperm[held] r :: false]
220   {
221     // Acquire the monitor (without deadlock checking)
222     inhale List(mon)
223     inhale acc(mon.held)
224
225     label acq
226
227     if(2 <= length(mon)) {
228       var r1: Int
229       r1 := dequeue(mon)
230       assert r1 <= peek(mon)
231     }
232
233     // Release the monitor
234     exhale List(mon) && length(mon) <= old[acq](length(mon))
235     exhale acc(mon.held)
236   }
```

Listing A.1: An encoding of a sorted linked-list based on list segments, as discussed in Section 2.4. Referenced on page 34.

Appendix B

Silicon

```
1  field f: Int
2
3
4  method test01(x: Ref, y: Ref)
5    requires acc(x.f)
6    ensures  acc(x.f)
7  {
8    if (x == y) {
9      exhale acc(x.f)
10     inhale acc(y.f)
11   }
12 }
13
14 method test01join(x: Ref, y: Ref)
15   requires acc(x.f)
16   ensures  acc(x.f) /* Fails now */
17 {
18   // if (x == y) {
19     exhale acc(x.f, x == y ? write : none)
20     inhale acc(y.f, x == y ? write : none)
21 }
22
23
24 method test02(x: Ref, y: Ref)
25 {
26   if (x != y) {
27     inhale acc(x.f)
28   } else {
29     inhale acc(x.f, 1/2)
30     inhale acc(y.f, 1/2)
31   }
32
33   assert acc(x.f)
34 }
35
36 method test02join(x: Ref, y: Ref)
37 {
38   // if (x != y)
39     inhale acc(x.f, x != y ? write : none)
40   // else
41     inhale acc(x.f, x == y ? 1/2 : none)
42     inhale acc(y.f, x == y ? 1/2 : none)
43
44   assert acc(x.f) /* Fails now */
45 }
```

Listing B.1: Illustrating that joining symbolic heaps after branching over conditionals, in combination with Silicon's greedy heap management, can incur incompletenesses. Referenced on page 74.

No.	Original file name
1	AVLTree.nokeys.sil
2	chaliceSuite_substantial-examples_AVLTree.nokeys.chalice.sil
3	testTreeWand.sil
4	testTreeWandE1.sil
5	0075_AVLTree.nokeys.sil
6	testTreeWandE2.sil
7	sequences.sil
8	chaliceSuite_predicates_LinkedList-various.chalice.sil
9	testListAppend.sil
10	oldC2SCases_nonnull_inference.chalice.sil
11	RingBufferRd.sil
12	testHistoryThreadsApplication.sil
13	chaliceSuite_regressions_workitem-10221.chalice.sil
14	oldC2SCases_quantifiers.chalice.sil
15	oldC2SCases_linked_list.chalice.sil
16	chaliceSuite_examples_PetersonsAlgorithm.chalice.sil
17	oldC2SCases_fields_fork_join.chalice.sil
18	chaliceSuite_general-tests_ll-lastnode.chalice.sil
19	chaliceSuite_examples_producer-consumer.chalice.sil
20	chaliceSuite_examples_iterator.chalice.sil
21	chaliceSuite_general-tests_cell-defaults.chalice.sil
22	oldC2SCases_basic.chalice.sil
23	oldC2SCases_eval_and_branching.chalice.sil
24	chaliceSuite_permission-model_basic.chalice.sil
25	chaliceSuite_examples_cell.chalice.sil
26	testThreadInheritanceE1.sil
27	oldC2SCases_controlflow.chalice.sil

Figure B.1: The original file names of the tests used in Section 3.7.1 to evaluate the stability of Silicon's performance by seeding verification failures. Referenced on page 93.

```
1 method test(ms1: Multiset[Ref], n: Int) {
2   inhale |ms1| == n
3   var ms2: Multiset[Ref] := ms1 intersection ms1
4   assert ms2 == ms1
5   assert |ms2| == n /* Fails without the preceding assertion */
6 }
```

Listing B.2: Illustrating an incompleteness in Viper's axiomatisation of multisets: an intermediate assertion is required to prove the final assertion. Referenced on page 94.

```
function fac(n: Int): Int
  requires 0 <= n
  ensures  0 < result /* Encode inductive argument */
{ n <= 1 ? 1 : n * fac(n - 1) }

method test_fac(n: Int)
  requires 0 <= n
  /* Requires function postcondition (or induction) */
  ensures  0 < fac(n)
{}



field next: Ref

predicate list(this: Ref) {
  acc(this.next) && (this.next != null ==> list(this.next))
}

function len(this: Ref): Int
  requires list(this)
  ensures  0 < result /* Encode inductive argument */
{
  unfolding list(this) in
    1 + (this.next == null ? 0 : len(this.next))
}

method test_len(this: Ref)
  requires list(this)
  ensures  list(this)
  /* Requires function postcondition (or induction) */
  ensures  0 < len(this)
{}
```

Listing B.3: Illustrating an incompleteness that is due to not having built-in support for induction: additional function postconditions are necessary in order to perform inductive reasoning and to prove the method postconditions. Referenced on page 94.

```
1  field next: Ref
2
3  predicate list(this: Ref) {
4    acc(this.next) && (this.next != null ==> list(this.next))
5  }
6
7  function len(this: Ref): Int
8    requires list(this)
9    ensures  0 < result /* Encode inductive argument */
10 {
11   unfolding list(this) in
12     1 + (this.next == null ? 0 : len(this.next))
13 }
14
15 function after(this: Ref, i: Int): Ref
16   requires list(this)
17   requires 0 <= i && i < len(this)
18 {
19   unfolding list(this) in
20     (i == 0 ? this.next : after(this.next, i - 1))
21 }
22
23 method test_recursion(this: Ref) {
24   inhale list(this) && len(this) == 3
25   assert unfolding list(this) in
26            unfolding list(this.next) in
27              unfolding list(this.next.next) in true
28   /* Fails without the preceding assertion */
29   assert after(this, 2) == null
30 }
31
32
33
34 function id(i: Int): Int { i }
35
36 method test_triggers1(xs: Seq[Int]) {
37   inhale 0 < |xs|
38   inhale forall i: Int :: {id(i)}
39            0 <= i && i < |xs| ==> xs[i] == i
40     /* The next assertions fails because the explicitly speci-
41      *  fied trigger is too strict. Specifying the trigger
42      *  {xs[i]}, or letting Viper select triggers automatical-
43      *  ly, will allow the assertion to suceed.
44      */
45   assert xs[0] == 0
46 }
47
48 method test_triggers2(xs: Seq[Int]) {
49   inhale 0 < |xs|
50   inhale forall i: Int :: 0 <= i && i < |xs| ==> id(xs[i]) == i
51     /* The next assertion fails because the triggers automati-
52      * cally chosen by Viper (since none have been provided
53      * explicitly) are too strict: Viper choses
54      * {|xs|}{id(xs[i])} as the (alternative) triggers.
55      */
56   assert xs[0] == 0
57 }
```

Listing B.4: Illustrating an incompleteness arising from decisions made when designing the Viper language: preventing non-terminating SMT solver runs by curbing the unrolling of recursive functions and using triggers to control quantifier instantiations. Referenced on page 94.

B.1 Comparing VeriCool, VeriFast and Silicon

The examples shown in this section comprise various small tests that illustrate heap-related incompletenesses discussed in the comparison of VeriCool, VeriFast and Silicon in Section 3.7.3. Each test has been encoded for all three verifiers (if possible).

B.1.1 Definite Aliasing

The following examples are referenced on page 98.

```
1  class Cell {
2    int val;
3  }
4
5  class Tests {
6    void inhale_val(Cell c)
7      requires c != null;
8      ensures acc(c.val);
9    {
10     assume false;
11   }
12
13   void exhale_val(Cell c)
14     requires c != null &*& acc(c.val);
15     ensures  true;
16   {
17     assume false;
18   }
19
20   void inhale_val_eq(Cell c, int i)
21     requires c != null &*& acc(c.val);
22     ensures acc(c.val) &*& c.val == i;
23   {
24     assume false;
25   }
26
27
28   void test01a(Cell c1, Cell c2)
29     requires c1 != null &*& c2 != null;
30     requires acc(c1.val) &*& c2 == c1;
31     requires c1.val == 1;
32   {
33     assert c2.val == 1;
34   }
35
36   void test01b(Cell c1, Cell c2)
37     requires c1 != null &*& c2 != null;
38     requires acc(c1.val) &*& c2 == c1;
39   {
40     inhale_val_eq(c1, 1);
41     assert c2.val == 1;
42   }
43
44   void test01v(Cell c1, Cell c2)
45     requires c1 != null &*& c2 != null;
46     requires acc(c1.val) &*& c1.val == 1;
47   {
48     assume c2 == c1;
49     assert c2.val == 1;
50   }
51
52
53   predicate pure bool eq(Cell c1, Cell c2) {
```

```
54      return c1 == c2;
55    }
56
57    void test01d(Cell c1, Cell c2)
58      requires c1 != null &*& acc(c1.val) &*& c1.val == 0;
59      requires eq(c1, c2);
60    {
61      assert opening eq(c1, c2) in c2.val == 0;
62    }
63  }
```

Listing B.5: Testing VeriCool's handling of the heap in the presence of definite aliasing. Referenced on page 98.

```
1   #include <stdlib.h>
2
3   struct Cell {
4     int val;
5   };
6
7   void inhale_val(struct Cell* c, real p)
8     //@ requires 0 <= p && p <= 1;
9     //@ ensures  [p]c->val |-> _;
10  {
11    //@ assume(false);
12  }
13
14  void exhale_val(struct Cell* c, real p)
15    //@ requires 0 <= p && p <= 1 &*& [p]c->val |-> _;
16    //@ ensures  true;
17  {
18    //@ assume(false);
19  }
20
21  void inhale_val_eq(struct Cell* c, int i)
22    //@ requires c->val |-> _;
23    //@ ensures  c->val |-> i;
24  {
25    //@ assume(false);
26  }
27
28
29  void test01a(struct Cell* c1, struct Cell* c2)
30    //@ requires c1->val |-> 1 &*& c2 == c1;
31    //@ ensures  true;
32  {
33    //@ assert c2->val == 1;
34
35    //@ assume(false);
36  }
37
38  void test01b(struct Cell* c1, struct Cell* c2)
39    //@ requires c1->val |-> _ &*& c2 == c1;
40    //@ ensures  true;
41  {
42    inhale_val_eq(c1, 1);
43    //@ assert c2->val == 1;
44
45    //@ assume(false);
46  }
47
48  void test01c(struct Cell* c1, struct Cell* c2)
49    //@ requires c1->val |-> 1;
50    //@ ensures  true;
51  {
```

```
52    //@ assume(c2 == c1);
53    //@ assert c2->val == 1;
54
55    //@ assume(false);
56  }
57
58
59  void test02a(struct Cell* c1, struct Cell* c2)
60    /*@ requires     [1/3]c1->val |-> _ &*& c1 == c2
61                  &*& [1/3]c2->val |-> _; @*/
62    //@ ensures  true;
63  {
64    //@ assume(c1->val == 1);
65    exhale_val(c1, 1/3);
66    //@ assert c2->val == 1;
67
68    //@ assume(false);
69  }
70
71  void test02b(struct Cell* c1, struct Cell* c2)
72    //@ requires [1/3]c1->val |-> _;
73    //@ ensures true;
74  {
75    //@ assume(c1 == c2);
76    inhale_val(c2, 1/3);
77    //@ assume(c1->val == 1);
78    exhale_val(c1, 1/3);
79    //@ assert c2->val == 1;
80
81    //@ assume(false);
82  }
83
84
85  void test03a(struct Cell* c1, struct Cell* c2)
86    /*@ requires     [1/3]c1->val |-> ?v &*& c1 == c2
87                  &*& [1/3]c2->val |-> ?w; @*/
88    //@ ensures  true;
89  {
90    //@ assert v == w;
91
92    //@ assume(false);
93  }
94
95  void test03b(struct Cell* c1, struct Cell* c2)
96    /*@ requires     [1/3]c1->val |-> ?v &*& [1/3]c2->val |-> ?w
97                  &*& c1 == c2; @*/
98    //@ ensures  true;
99  {
100   //@ merge_fractions c2->val |-> _;
101   //@ assert v == w; // FAILS w/o merge_fractions
102
103   //@ assume(false);
104 }
105
106 void test03c(struct Cell* c1, struct Cell* c2)
107   //@ requires [1/3]c1->val |-> _ &*& [1/3]c2->val |-> _;
108   //@ ensures  true;
109 {
110   //@ assume(c1 == c2);
111   //@ assume(c1->val == 1);
112   //@ assert c2->val == 1;
113   //@ merge_fractions c2->val |-> _;
114   exhale_val(c1, 1/3);
115   //@ assert c2->val == 1; // FAILS w/o merge_fractions
116
117   //@ assume(false);
```

```
}


void test04a(struct Cell* c1, struct Cell* c2)
  //@ requires [1/3]c1->val |-> _;
  //@ ensures true;
{
  inhale_val(c2, 1/3);
  //@ assume(c1 == c2);
  //@ assert [1/3]c1->val |-> _ &*& [1/3]c1->val |-> _;
  //@ merge_fractions c1->val |-> _;
  //@ assert [2/3]c1->val |-> _; // FAILS w/o merge_fractions
  //@ assume(c1->val == 1);
  exhale_val(c1, 1/3);
  //@ assert c2->val == 1; // FAILS w/o merge_fractions

  //@ assume(false);
}

void test04b(struct Cell* c1, struct Cell* c2)
  //@ requires [1/3]c1->val |-> _ &*& [1/3]c2->val |-> _;
  //@ ensures true;
{
  //@ assume(c2->val == 1);
  //@ assume(c1 == c2);
  //@ merge_fractions c1->val |-> _;
  exhale_val(c2, 1/3);
  //@ assert c1->val == 1; // FAILS w/o merge_fractions

  //@ assume(false);
}
```

Listing B.6: Testing VeriFast's handling of the heap in the presence of definite aliasing. Referenced on page 98.

```
field val: Int

method test01a(c1: Ref, c2: Ref)
  requires acc(c1.val) && c2 == c1
  requires c1.val == 1
{
  assert c2.val == 1
}

method test01b(c1: Ref, c2: Ref)
  requires acc(c1.val) && c2 == c1
{
  inhale acc(c1.val) && c1.val == 1
  assert c2.val == 1
}

method test01c(c1: Ref, c2: Ref)
  requires acc(c1.val) && c1.val == 1
{
  inhale c2 == c1
  assert c2.val == 1
}


predicate eq(c1: Ref, c2: Ref) { c1 == c2 }

method test01d(c1: Ref, c2: Ref)
  requires acc(c1.val) && c1.val == 0 && acc(eq(c1,c2))
{
  assert unfolding acc(eq(c1, c2)) in c2.val == 0
```

```
31 }
32
33
34 method test02a(c1: Ref, c2: Ref)
35   requires acc(c1.val, 1/3) && c1 == c2 && acc(c2.val, 1/3)
36 {
37   inhale c1.val == 1
38   exhale acc(c1.val, 1/3)
39   assert c2.val == 1
40 }
41
42 method test02b(c1: Ref, c2: Ref)
43   requires acc(c1.val, 1/3)
44 {
45   inhale c1 == c2
46   inhale acc(c2.val, 1/3)
47   inhale c1.val == 1
48   exhale acc(c1.val, 1/3)
49   assert c2.val == 1
50 }
51
52
53 method test03a(c1: Ref, c2: Ref, v: Int, w: Int)
54   requires acc(c1.val, 1/3) && c1.val == v
55   requires c1 == c2
56   requires acc(c2.val, 1/3) && c2.val == w
57 {
58   assert v == w
59 }
60
61 method test03b(c1: Ref, c2: Ref, v: Int, w: Int)
62   requires acc(c1.val, 1/3) && c1.val == v
63   requires acc(c2.val, 1/3) && c2.val == w
64   requires c1 == c2
65 {
66   assert v == w
67 }
68
69 method test03c(c1: Ref, c2: Ref)
70   requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
71 {
72   inhale c1 == c2
73   inhale c1.val == 1
74   assert c2.val == 1
75   exhale acc(c1.val, 1/3)
76   assert c2.val == 1 // FAILS w/o preceding assert
77 }
78
79
80 method test04a(c1: Ref, c2: Ref)
81   requires acc(c1.val, 1/3)
82 {
83   inhale acc(c2.val, 1/3)
84   inhale c1 == c2
85   assert acc(c1.val, 2/3)
86   inhale c1.val == 1
87   exhale acc(c1.val, 1/3)
88   assert c2.val == 1 // FAILS w/o preceding assert
89 }
90
91 method test04b(c1: Ref, c2: Ref)
92   requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
93 {
94   inhale c2.val == 1
95   inhale c1 == c2
96   assert true // Triggers a heap compression
```

```
97    exhale acc(c2.val, 1/3)
98    assert c1.val == 1 // // FAILS w/o preceding assert
99  }
100
101
102 function reqval(c: Ref, p: Perm): Bool
103   requires none <= p && p <= write
104   requires acc(c.val, p)
105
106 method test05a(c1: Ref)
107   requires acc(c1.val, 1/3) && acc(c1.val, 1/3)
108   requires reqval(c1, 2/3)
109 {}
110
111 method test05b(c1: Ref, c2: Ref)
112   requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
113   requires c1 == c2 && reqval(c1, 2/3)
114 {}
```

Listing B.7: Testing Silicon's handling of the heap in the presence of definite aliasing. Referenced on page 98.

B.1.2 Indefinite Aliasing

The following examples are referenced on page 98.

```
1  class Cell {
2    int val;
3  }
4
5  class Tests {
6    void test01(Cell c1, Cell c2, Cell c3)
7      requires c1 != null &*& c2 != null;
8      requires acc(c1.val) &*& acc(c2.val);
9      requires (c3 == c1 || c3 == c2);
10   {
11     assert acc(c3.val); // FAILS
12   }
13
14   void test02a(Cell c1, Cell c2, Cell c3)
15     requires c1 != null &*& c2 != null;
16     requires acc(c1.val) &*& acc(c2.val);
17     requires (c3 == c1 || c3 == c2);
18   {
19     c3.val = 0; // FAILS
20   }
21
22   void test02b(Cell c1, Cell c2, Cell c3)
23     requires c1 != null &*& c2 != null;
24     requires acc(c1.val) &*& acc(c2.val);
25     requires (c3 == c1 || c3 == c2);
26   {
27     if (c3 == c2) {
28       c3.val = 0;
29     }
30   }
31 }
```

Listing B.8: Testing VeriCool's handling of the heap in the presence of indefinite aliasing. Referenced on page 98.

```
1  #include "stdlib.h"
```

```
2
3  struct Cell {
4    int val;
5  };
6
7
8  void test01(struct Cell* c1, struct Cell* c2, struct Cell* c3)
9    /*@ requires     c1->val |-> _ &*& c2->val |-> _
10                &*& (c3 == c1 || c3 == c2); @*/
11   //@ ensures true;
12 {
13   //@ assert [_]c3->val |-> _; // FAILS
14
15   //@ assume(false);
16 }
17
18 void test02a(struct Cell* c1, struct Cell* c2, struct Cell* c3)
19   /*@ requires     c1->val |-> _ &*& c2->val |-> _
20                &*& (c3 == c1 || c3 == c2); @*/
21   //@ ensures true;
22 {
23   c3->val = 0; // FAILS
24
25   //@ assume(false);
26 }
27
28 void test02b(struct Cell* c1, struct Cell* c2, struct Cell* c3)
29   /*@ requires     c1->val |-> _ &*& c2->val |-> _
30                &*& (c3 == c1 || c3 == c2); @*/
31   //@ ensures true;
32 {
33   if (c3 == c2) {
34     c3->val = 0;
35   }
36
37   //@ assume(false);
38 }
```

Listing B.9: Testing VeriFast's handling of the heap in the presence of indefinite aliasing. Referenced on page 98.

```
1  field val: Int
2
3  method test01(c1: Ref, c2: Ref, c3: Ref)
4    requires acc(c1.val) && acc(c2.val) && (c3 == c1 || c3 == c2)
5  {
6    assert acc(c3.val) // FAILS
7  }
8
9  method test02a(c1: Ref, c2: Ref, c3: Ref)
10   requires acc(c1.val) && acc(c2.val) && (c3 == c1 || c3 == c2)
11 {
12   c3.val := 0 // FAILS
13 }
14
15 method test02b(c1: Ref, c2: Ref, c3: Ref)
16   requires acc(c1.val) && acc(c2.val) && (c3 == c1 || c3 == c2)
17 {
18   if (c3 == c2) {
19     c3.val := 0
20   }
21 }
```

Listing B.10: Testing Silicon's handling of the heap in the presence of indefinite aliasing. Referenced on page 98.

B.1.3 Retrospective Aliasing

The following examples are referenced on page 98.

```
#include <stdlib.h>

struct Cell {
  int val;
};

void exhale_val(struct Cell* c, real p)
  //@ requires 0 <= p && p <= 1 &*& [p]c->val |-> _;
  //@ ensures  true;
{
  //@ assume(false);
}


void test01(struct Cell* c1, struct Cell* c2)
  //@ requires [1/3]c1->val |-> _ &*& [1/3]c2->val |-> _;
  //@ ensures true;
{
  //@ assume(c1->val == 1);
  exhale_val(c1, 1/3);
  //@ assume(c1 == c2);
  //@ assert c2->val == 1; // FAILS

  //@ assume(false);
}

void test02(struct Cell* c1, struct Cell* c2)
  //@ requires [1/3]c1->val |-> _ &*& [1/3]c2->val |-> _;
  //@ ensures true;
{
  //@ assume(c1->val == 1);
  /*@ if (c1 == c2) {
        merge_fractions c2->val |-> _;
        assert c2->val == 1;
      } @*/
  exhale_val(c1, 1/3);
  //@ assume(c1 == c2);
  //@ assert c2->val == 1;

  //@ assume(false);
}
```

Listing B.11: Testing VeriFast's handling of the heap in the presence of retrospective aliasing. Referenced on page 98.

```
field val: Int

method test01(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
{
  inhale c1.val == 1
  exhale acc(c1.val, 1/3)
  inhale c1 == c2
```

```
  assert c2.val == 1 // FAILS
}

method test02a(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
{
  inhale c1.val == 1
label pre_exhale
  exhale acc(c1.val, 1/3)
  inhale c1 == c2
  assert old[pre_exhale](c2.val == 1)
  assert c2.val == 1
}

method test02b(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
{
  inhale c1.val == 1
  if (c1 == c2) {
    assert c2.val == 1
  }
  exhale acc(c1.val, 1/3)
  inhale c1 == c2
  assert c2.val == 1
}
```

Listing B.12: Testing Silicon's handling of the heap in the presence of retrospective aliasing. Referenced on page 98.

B.1.4 Path Conditions from Permissions

The following examples are referenced on pages 98, 99.

```
class Cell {
  int val;
}

class Tests {
  void inhale_val(Cell c)
    requires c != null;
    ensures acc(c.val);
  {
    assume false;
  }


  void test01(Cell c1, Cell c2)
    requires c1 != null &*& c2 != null;
    requires acc(c1.val) &*& acc(c2.val);
    ensures c1 != c2;
  {}

  void test01a(Cell c1, Cell c2)
    requires c1 != null &*& c2 != null;
  {
    inhale_val(c1);
    inhale_val(c2);
    assert c1 != c2; // FAILS
  }

  void test01b(Cell c1, Cell c2)
    requires c1 != null &*& c2 != null;
  {
```

```
31      inhale_val(c1);
32      inhale_val(c2);
33      assume c1 == c2;
34      assert false; // FAILS
35    }
36
37
38    predicate pure bool pred_val(Cell c) {
39      return c != null &*& acc(c.val);
40    }
41
42    void test03a(Cell c1, Cell c2)
43      requires c1 != null &*& acc(c1.val);
44      requires pred_val(c2);
45    {
46      open pred_val(c2);
47      assert c1 != c2; // FAILS
48    }
49  }
```

Listing B.13: Testing to which extent VeriCool infers path conditions from heap-related information. Referenced on pages 98, 99.

```
1   #include "stdlib.h"
2
3   struct Cell {
4     int val;
5   };
6
7   void inhale_val(struct Cell* c, real p)
8     //@ requires 0 <= p && p <= 1;
9     //@ ensures  [p]c->val |-> _;
10  {
11    //@ assume(false);
12  }
13
14
15  void test01(struct Cell* c1, struct Cell* c2)
16    //@ requires c1->val |-> _ &*& c2->val |-> _;
17    //@ ensures true;
18  {
19    //@ assert c2 != c1;
20
21    //@ assume(false);
22  }
23
24  void test01a(struct Cell* c1, struct Cell* c2)
25    //@ requires true;
26    //@ ensures true;
27  {
28    inhale_val(c1, 1/1);
29    inhale_val(c2, 1/3);
30    //@ assert c2 != c1;
31
32    //@ assume(false);
33  }
34
35  void test01b(struct Cell* c1, struct Cell* c2)
36    //@ requires true;
37    //@ ensures true;
38  {
39    inhale_val(c1, 1/1);
40    inhale_val(c2, 1/3);
41    //@ assume (c1 == c2);
42    //@ assert false;
```

```
43  }
44
45
46  void test02(struct Cell* c1, struct Cell* c2)
47    /*@ requires     [1/3]c1->val |-> _
48                 &*& [2/3]c2->val |-> _ &*& [1/3]c2->val |-> _; @*/
49    //@ ensures true;
50  {
51    //@ merge_fractions c1->val |-> _;
52    //@ merge_fractions c2->val |-> _;
53    //@ assert c2 != c1; // FAILS despite merge_fractions
54
55    if (c1 == c2) {
56      //@ merge_fractions c1->val |-> _;
57      //@ assert false; // FAILS w/o merge_fractions
58    }
59
60    //@ assume(false);
61  }
62
63  void test02a(struct Cell* c1, struct Cell* c2)
64    //@ requires true;
65    //@ ensures true;
66  {
67    inhale_val(c1, 1/2);
68    inhale_val(c1, 1/2);
69    inhale_val(c2, 1/3);
70    //@ merge_fractions c1->val |-> _;
71    //@ merge_fractions c2->val |-> _;
72    //@ assert c2 != c1; // FAILS despite merge_fractions
73
74    //@ assume(false);
75  }
76
77  void test02b(struct Cell* c1, struct Cell* c2)
78    //@ requires true;
79    //@ ensures true;
80  {
81    inhale_val(c1, 1/2);
82    inhale_val(c1, 1/2);
83    inhale_val(c2, 1/3);
84    //@ assume(c1 == c2);
85    //@ merge_fractions c1->val |-> _;
86    //@ merge_fractions c2->val |-> _;
87    //@ assert false; // FAILS w/o merge_fractions
88
89    //@ assume(false);
90  }
91
92
93  //@ predicate pred_val(struct Cell* c) = c->val |-> _;
94
95  void test03a(struct Cell* c1, struct Cell* c2)
96    //@ requires c1->val |-> _ &*& pred_val(c2);
97    //@ ensures true;
98  {
99    //@ open pred_val(c2);
100   //@ assert c1 != c2;
101
102   //@ assume(false);
103 }
104
105 void test03b(struct Cell* c1, struct Cell* c2)
106   /*@ requires [2/3]c1->val |-> _ &*& [2/3]pred_val(c2)
107                &*& c1 == c2; @*/
108   //@ ensures true;
```

```
109 {
110   //@ open [2/3]pred_val(c2);
111   //@ assert false;
112 }
113
114 void test03c(struct Cell* c1, struct Cell* c2)
115   //@ requires [2/3]c1->val |-> _ &*& [2/3]pred_val(c2);
116   //@ ensures true;
117 {
118   //@ open [2/3]pred_val(c2);
119   //@ merge_fractions c1->val |-> _;
120   //@ assume(c1 == c2);
121   //@ assert false; // FAILS despite merge_fractions
122
123   //@ assume(false);
124 }
125
126
127 void test04(struct Cell* c1, struct Cell* c2)
128   /*@ requires     [1/3]c1->val |-> ?v &*& [1/3]c2->val |-> ?w
129               &*& v != w; @*/
130   //@ ensures true;
131 {
132   //@ assert c1->val != c2->val;
133   //@ assert c1 != c2; // FAILS
134 }
```

Listing B.14: Testing to which extent VeriFast infers path conditions from heap-related information. Referenced on pages 98, 99.

```
1  field val: Int
2
3  method test01(c1: Ref, c2: Ref)
4    requires acc(c1.val) && acc(c2.val)
5  {
6    assert c2 != c1
7  }
8
9  method test01a(c1: Ref, c2: Ref) {
10   inhale acc(c1.val)
11   inhale acc(c2.val)
12   assert c2 != c1
13 }
14
15 method test01b(c1: Ref, c2: Ref) {
16   inhale acc(c1.val)
17   inhale acc(c2.val, 1/3)
18   inhale c1 == c2
19   assert false
20 }
21
22
23 method test02(c1: Ref, c2: Ref)
24   requires acc(c1.val, 1/3)
25   requires acc(c2.val, 2/3) && acc(c2.val, 1/3)
26 {
27   assert c2 != c1
28
29   if (c1 == c2) {
30     assert false
31   }
32 }
33
34 method test02a(c1: Ref, c2: Ref) {
35   inhale acc(c1.val, 1/2)
```

```
36    inhale acc(c1.val, 1/2)
37    inhale acc(c2.val, 1/3)
38    assert c2 != c1
39  }
40
41  method test02b(c1: Ref, c2: Ref) {
42    inhale acc(c1.val, 1/2)
43    inhale acc(c1.val, 1/2)
44    inhale acc(c2.val, 1/3)
45    inhale c1 == c2
46    assert false
47  }
48
49
50  predicate pred_val(c: Ref) {
51    acc(c.val)
52  }
53
54  method test03a(c1: Ref, c2: Ref)
55    requires acc(c1.val)
56    requires acc(pred_val(c2))
57  {
58    unfold acc(pred_val(c2))
59    assert c1 != c2
60  }
61
62  method test03b(c1: Ref, c2: Ref)
63    requires acc(c1.val, 2/3)
64    requires acc(pred_val(c2), 2/3)
65    requires c1 == c2
66  {
67    unfold acc(pred_val(c2), 2/3)
68    assert false
69  }
70
71  method test03c(c1: Ref, c2: Ref)
72    requires acc(c1.val, 2/3)
73    requires acc(pred_val(c2), 2/3)
74  {
75    unfold acc(pred_val(c2), 2/3)
76    inhale c1 == c2
77    assert false
78  }
79
80
81  method test04(c1: Ref, c2: Ref)
82    requires acc(c1.val, 1/3) && acc(c2.val, 1/3)
83    requires c1.val != c2.val
84  {
85    assert c2 != c1 // FAILS
86  }
```

Listing B.15: Testing to which extent Silicon infers path conditions from heap-related information. Referenced on pages 98, 99.

B.1.5 Permission Introspection

The following examples are referenced on page 99.

```
1  field val: Int
2
3  /*
4   * inhale perm(x.f) > p (potentially unsound if permissions
```

```
 *                          are under-approximated)
 */

method test01a(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2) && c1 == c2
{
  assume perm(c1.val) >= 1/1
  assert false // FAILS expectedly
}

method test01b(c1: Ref, c2: Ref, c3: Ref)
  requires acc(c1.val, 1/2)
  requires acc(c2.val, 1/2) && acc(c2.val, 1/2)
  requires c3 == c1 || c3 == c2
{
  assume perm(c3.val) >= 1/1
  assert false // FAILS expectedly
}

method test01c(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2)
{
  assume perm(c1.val) >= 1/1
  assume c1 == c2
  assert false // FAILS expectedly
}

method test01d(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2)
{
  assume perm(c1.val) >= 1/1
  exhale acc(c1.val, 1/2)
  assume c1 == c2
  assert false // FAILS expectedly
}

/*
 * exhale perm(x.f) < p (potentially unsound)
 */

method test02a(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2) && c1 == c2
{
  assert perm(c1.val) <= 1/2 // FAILS expectedly
}

method test02b(c1: Ref, c2: Ref, c3: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2)
  requires acc(c3.val, 1/2)
  requires c3 == c1 || c3 == c2
{
  assert perm(c3.val) <= 1/2 // FAILS expectedly
}


/*
 * exhale perm(x.f) > p (potentially incomplete)
 */

method test03a(c1: Ref, c2: Ref)
  requires acc(c1.val, 1/2) && acc(c2.val, 1/2) && c1 == c2
{
  assert perm(c1.val) >= 1/1
}

method test03b(c1: Ref, c2: Ref, c3: Ref)
```

```
71    requires acc(c1.val, 1/2) && acc(c2.val, 1/2)
72    requires acc(c3.val, 1/2)
73    requires c3 == c1 || c3 == c2
74  {
75    assert perm(c3.val) >= 1/1
76  }
```

Listing B.16: Tests that illustrate potential unsoundnesses and incompletenesses that could arise if `perm` under-approximated permission amounts. Referenced on page 99.

```
1   #include <stdlib.h>
2
3   struct Cell {
4     int val;
5   };
6
7   void exhale_val(struct Cell* c, real p)
8     //@ requires 0 <= p && p <= 1 &*& [p]c->val |-> _;
9     //@ ensures  true;
10  {
11    //@ assume(false);
12  }
13
14
15  /*
16   * inhale perm(x.f) > p (potentially unsound if permissions
17   *                       are under-approximated)
18   */
19
20  /* Definite aliasing */
21  void test01a(struct Cell* c1, struct Cell* c2)
22    /*@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
23                &*& c1 == c2; @*/
24    //@ ensures  true;
25  {
26    ///@ merge_fractions c1->val |-> _;
27    //@ assert [?p]c1->val |-> _;
28    //@ assume(p >= 1/1);
29    //@ assert false; // HOLDS unsoundly w/o merge_fractions
30  }
31
32  /* Indefinite aliasing */
33  void test01b(struct Cell* c1, struct Cell* c2, struct Cell* c3)
34    /*@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
35                &*& [1/2]c3->val |-> _
36                &*& (c3 == c1 || c3 == c2); @*/
37    //@ ensures  true;
38  {
39    //@ merge_fractions c1->val |-> _;
40    //@ assert [?p]c3->val |-> _;
41    //@ assume(p >= 1/1);
42    //@ assert false; // HOLDS unsoundly
43  }
44
45  /* Retrospective aliasing */
46  void test01c(struct Cell* c1, struct Cell* c2)
47    //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _;
48    //@ ensures  true;
49  {
50    //@ assert [?p]c1->val |-> _;
51    //@ assume(p >= 1/1);
52    //@ assume(c1 == c2);
53    //@ assert false; // HOLDS unsoundly
54  }
55
```

```
56 /* Retrospective aliasing */
57 void test01d(struct Cell* c1, struct Cell* c2)
58   //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _;
59   //@ ensures  true;
60 {
61   //@ assert [?p]c1->val |-> _;
62   //@ assume(p >= 1/1);
63   exhale_val(c1, 1/2);
64   //@ assume(c1 == c2);
65   //@ assert false; // HOLDS unsoundly
66 }
67
68
69 /*
70  * exhale perm(x.f) < p (potentially unsound)
71  */
72
73  void test02a(struct Cell* c1, struct Cell* c2)
74   //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
75                &*& c1 == c2; @*/
76   //@ ensures  true;
77 {
78   //@ merge_fractions c1->val |-> _;
79   //@ assert [?p]c1->val |-> _;
80   //@ assert p <= 1/2; // HOLDS unsoundly w/o merge_fractions
81
82   //@ assume(false);
83 }
84
85 void test02b(struct Cell* c1, struct Cell* c2, struct Cell* c3)
86   //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
87                &*& [1/2]c3->val |-> _
88                &*& (c3 == c1 || c3 == c2); @*/
89   //@ ensures  true;
90 {
91   //@ merge_fractions c1->val |-> _;
92   //@ assert [?p]c3->val |-> _;
93   //@ assert p <= 1/2; // HOLDS unsoundly
94
95   //@ assume(false);
96 }
97
98
99 /*
100  * exhale perm(x.f) > p (potentially incomplete)
101  */
102
103  void test03a(struct Cell* c1, struct Cell* c2)
104   //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
105                &*& c1 == c2; @*/
106   //@ ensures  true;
107 {
108   //@ merge_fractions c1->val |-> _;
109   //@ assert [?p]c1->val |-> _;
110   //@ assert p >= 1/1; // FAILS w/o merge_fractions
111
112   //@ assume(false);
113 }
114
115 void test03b(struct Cell* c1, struct Cell* c2, struct Cell* c3)
116   //@ requires     [1/2]c1->val |-> _ &*& [1/2]c2->val |-> _
117                &*& [1/2]c3->val |-> _
118                &*& (c3 == c1 || c3 == c2); @*/
119   //@ ensures  true;
120 {
121   //@ merge_fractions c1->val |-> _;
```

```
122   //@ assert [?p]c3->val |-> _;
123   //@ assert p >= 1/1; // FAILS
124 }
```

Listing B.17: Encodings of the tests from Listing B.16 that illustrate unsoundnesses and incompletenesses that arise from under-approximating permission amounts. Referenced on page 99.

B.1.6 Object Allocation

The following examples are referenced on page 99.

```
1  class Cell {
2    Cell val;
3
4    Cell()
5      ensures acc(val);
6    {}
7  }
8
9  class Tests {
10   void test01() {
11     Cell c1 = new Cell();
12     Cell c2 = new Cell();
13     assert acc(c1.val) && acc(c2.val);
14     assert c1 != c2; // FAILS
15   }
16
17   void test02(Cell c1) {
18     Cell c2 = new Cell();
19     assert c1 != c2; // FAILS
20   }
21
22   void test04(Cell c1)
23     requires c1 != null &*& acc(c1.val);
24     requires c1.val != null &*& acc(c1.val.val);
25   {
26     Cell c2 = new Cell();
27     assert c2 != c1.val || c2 != c1.val.val; // FAILS
28   }
29
30
31   predicate pure bool pred_val(Cell c) {
32     return c != null &*& acc(c.val);
33   }
34
35   void test05(Cell c1)
36     requires pred_val(c1);
37   {
38     Cell c2 = new Cell();
39     open pred_val(c1);
40     assert c2 != c1.val; // FAILS
41   }
42 }
```

Listing B.18: Testing to which extent VeriCool infers reference disequalities from object allocation. Referenced on page 99.

```
1  #include "stdlib.h"
2
3  struct Cell {
4    struct Cell* val;
```

```
};


void test01()
  //@ requires true;
  //@ ensures true;
{
  struct Cell* c1 = malloc(sizeof(struct Cell));
  struct Cell* c2 = malloc(sizeof(struct Cell));
  if (c1 == 0 || c2 == 0) abort();
  //@ assert c1 != c2;

  //@ assume(false);
}

void test02(struct Cell* c1)
  //@ requires c1 != 0;
  //@ ensures true;
{
  struct Cell* c2 = malloc(sizeof(struct Cell));
  if(c2 == 0) abort();
  //@ assert c1 != c2; // FAILS

  //@ assume(false);
}

void test03(list<struct Cell*> cs)
  //@ requires 0 < length(cs) && head(cs) != 0;
  //@ ensures true;
{
  struct Cell* c2 = malloc(sizeof(struct Cell));
  if (c2 == 0) abort();
  //@ assert c2 != head(cs); // FAILS

  //@ assume(false);
}

void test04(struct Cell* c1)
  /*@ requires     [1/3]c1->val |-> ?c1v
               &*& [1/3]c1v->val |-> ?c1vv; @*/
  //@ ensures true;
{
  struct Cell* c2 = malloc(sizeof(struct Cell));
  if (c2 == 0) abort();
  //@ assert c2 != c1v;
  //@ assert c2 != c1vv; // FAILS

  //@ assume(false);
}
```

Listing B.19: Testing to which extent VeriFast infers reference disequalities from object allocation. Referenced on page 99.

```
field val: Ref

method test01() {
  var c1: Ref; c1 := new(val)
  var c2: Ref; c2 := new(val)
  assert c1 != c2
}

method test02(c1: Ref) {
  var c2: Ref; c2 := new(val)
  assert c1 != c2
}
```

```
13
14 method test03(cs: Seq[Ref])
15   requires 0 < |cs|;
16 {
17   var c2: Ref; c2 := new(val)
18   assert c2 != cs[0]
19 }
20
21 method test04(c1: Ref)
22   requires acc(c1.val, 1/3) && acc(c1.val.val, 1/3)
23 {
24   var c2: Ref; c2 := new(val)
25   assert c2 != c1.val && c2 != c1.val.val
26 }
27
28
29 predicate pred_val(c: Ref) {
30   acc(c.val)
31 }
32
33 method test05(c1: Ref)
34   requires acc(pred_val(c1))
35 {
36   var c2: Ref; c2 := new(val)
37   unfold acc(pred_val(c1))
38   assert c2 != c1.val // FAILS
39 }
40
41
42 predicate abs(c: Ref)
43
44 function fun(c: Ref): Ref
45   requires acc(abs(c))
46
47 method test06(c1: Ref)
48   requires acc(abs(c1))
49 {
50   var c2: Ref; c2 := new(val)
51   assert c2 != fun(c1) // FAILS
52 }
```

Listing B.20: Testing to which extent Silicon infers reference disequalities from object allocation. Referenced on page 99.

B.1.7 Function Unrolling Depth

The following examples are referenced on pages 99, 100.

```
1  class Node {
2    Node next;
3
4    Node()
5      ensures acc(next) &*& next == null;
6    {}
7
8    predicate pure bool node() {
9      return     acc(next)
10            &*& ifthenelse(next != null, next.node(), true);
11   }
12
13   pure int length()
14     requires node();
15     ensures  result > 0;
```

```
16    {
17      return
18        1 + opening node() in
19                ifthenelse(next != null, next.length(), 0);
20    }
21  }
22
23  class Client {
24    void test01() {
25      Node n1 = new Node();
26      n1.next = null;
27      close n1.node();
28
29      Node n2 = new Node();
30      n2.next = n1;
31      close n2.node();
32
33      assert n2.length() == 2; // FAILS
34    }
35
36    void test02(Node n4)
37      requires n4 != null &*& n4.node() &*& n4.length() == 4;
38    {
39      open n4.node();
40      open n4.next.node();
41      open n4.next.next.node();
42      open n4.next.next.next.node();
43
44      assert n4.next.next.next.next == null; // FAILS
45    }
46
47    void test03(Node n)
48      requires n != null &*& n.node();
49      ensures  n.node();
50      ensures  n.length() > 0;
51        /* FAILS: Function postconditions appear to be ignored */
52    {
53      open n.node();  // Do not ...
54      close n.node(); // ... help
55    }
56  }
```

Listing B.21: A VeriCool example illustrating that the depth to which the solver can unroll function definitions should be determined dynamically. Referenced on pages 99, 100.

```
1   field next: Ref
2
3   predicate node(this: Ref) {
4     acc(this.next) && (this.next != null ==> acc(node(this.next)))
5   }
6
7   function length(this: Ref): Int
8     requires acc(node(this))
9     ensures  result > 0
10  {
11    1 + unfolding acc(node(this)) in
12            this.next == null ? 0 : length(this.next)
13  }
14
15  method test01() {
16    var n1: Ref; n1 := new(next)
17    n1.next := null
18    fold acc(node(n1))
19
```

```
20    var n2: Ref; n2 := new(next)
21    n2.next := n1
22    fold acc(node(n2))
23
24    assert length(n2) == 2
25  }
26
27  method test02(n4: Ref)
28    requires acc(node(n4)) && length(n4) == 4
29  {
30    unfold acc(node(n4))
31    unfold acc(node(n4.next))
32    unfold acc(node(n4.next.next))
33    unfold acc(node(n4.next.next.next))
34
35    assert n4.next.next.next.next == null
36  }
37
38  method test03(n: Ref)
39    requires acc(node(n))
40    ensures  acc(node(n)) && length(n) > 0
41  {}
```

Listing B.22: A Silicon example illustrating that the depth to which the solver can unroll function definitions should be determined dynamically. Referenced on pages 99, 100.

```
1   #include "stdlib.h"
2
3   struct Node {
4     struct Node* next;
5   };
6
7   /*@
8   predicate node(struct Node *this; int len) =
9         this != 0
10    &*& this->next |-> ?nxt
11    &*& (nxt != 0
12          ? (    node(nxt, ?nxtlen)
13             &*& nxtlen > 0 &*& len == 1 + nxtlen)
14          : len == 1);
15  @*/
16
17  void test01()
18    //@ requires true;
19    //@ ensures  true;
20  {
21    struct Node* n1 = malloc(sizeof(struct Node));
22    if(n1 == 0) abort();
23    n1->next = 0;
24    //@ close node(n1, ?len1);
25
26    struct Node* n2 = malloc(sizeof(struct Node));
27    if(n2 == 0) abort();
28    n2->next = n1;
29    //@ close node(n2, ?len2);
30
31    //@ assert len2 == 2; // SUCCEEDS
32
33    //@ assume(false);
34  }
35
36  void test02(struct Node* n4)
37    //@ requires node(n4, ?len4) &*& len4 == 4;
38    //@ ensures  true;
```

```
39 {
40   //@ open node(n4, len4);
41   //@ open node(n4->next, _);
42   //@ open node(n4->next->next, _);
43   //@ open node(n4->next->next->next, _);
44   //@ assert n4->next->next->next->next == 0;
45
46   //@ assume(false);
47 }
48
49 void test03(struct Node* n)
50   //@ requires node(n, ?len);
51   //@ ensures  node(n, len) &*& len > 0;
52           // len > 0 FAILS w/o open-close pair
53 {
54   //@ open node(n, len);
55   //@ close node(n, len);
56 }
```

Listing B.23: A VeriFast example illustrating how the linked-list example is encoded in VeriFast. Referenced on pages 99, 100.

B.1.8 Predicate Unfolding Depth

The following examples are referenced on page 100.

```
1  class Node {
2    Node next;
3    int val;
4
5    /* CRASH: The definition of node01 results in a
6     * StackOverflowError, probably during the predicate's
7     * well-definedness check.
8     */
9
10   predicate pure bool node01() {
11     return
12       acc(next) &*& acc(val) &*&
13       ifthenelse(
14         next != null,
15         next.node01()
16           &*& opening next.node01() in val < next.val,
17         true);
18   }
19
20   /* CRASH: The definitions of node02 and get02 result in a
21    * StackOverflowError, probably during the well-definedness
22    * check of the predicate or the function.
23    */
24
25   predicate pure bool node02() {
26     return
27       acc(next) &*& acc(val) &*&
28       ifthenelse(
29         next != null,
30         next.node02() &*& val < next.get02(),
31         true);
32   }
33
34   pure int get02()
35     requires node02();
36   { return opening node02() in val; }
37 }
```

Listing B.24: A VeriCool example illustrating that the symbolic evaluation of an `unfolding` expression can indefinitely recurse if no explicit countermeasures are taken. Referenced on page 100.

```
field next: Ref
field val: Int

predicate node01(this: Ref) {
  acc(this.next) && acc(this.val) &&
  (this.next != null ==>
    acc(node01(this.next)) &&
    unfolding acc(node01(this.next)) in
      this.val < this.next.val)
}

method test01(n: Ref)
  requires acc(node01(n))
{
  assert
    unfolding acc(node01(n)) in
      n.next != null ==>
        unfolding acc(node01(n.next)) in n.val < n.next.val
}

predicate node02(this: Ref) {
  acc(this.next) && acc(this.val) &&
  (this.next != null ==>
    acc(node02(this.next)) && this.val < get02(this.next))
}

function get02(this: Ref): Int
  requires acc(node02(this))
{ unfolding acc(node02(this)) in this.val }

method test02(n: Ref)
  requires acc(node02(n))
{
  assert
    unfolding acc(node02(n)) in
      n.next != null ==> n.val < get02(n.next)
}
```

Listing B.25: An example illustrating that Silicon prevents indefinitely recursing symbolic evaluations of `unfolding` expressions. Referenced on page 100.

Appendix C

Quantified Permissions

Listing C.1 shows an encoding of the running example from Chapter 4. The example is included as `parallel-replace` in the test set described in Section 4.5. The encoding uses the array domain discussed in Section 2.5, and parameterised macros (first used in Section 2.2.2), which are inlined similarly to C-style macros, to allow reusing pre- and postconditions. The parallel recursive calls are modelled by appropriate `exhale` (fork) and `inhale` (join) statements.

```
define pre1(a, l, r)
  0 <= l && l < r && r <= len(a)
define pre2(a, l, r)
  forall i: Int :: l <= i && i < r ==> acc(loc(a, i).val)
define post1(a, l, r)
  pre2(a, l, r)
define post2(a, l, r)
  forall i: Int :: l <= i && i < r ==>
    (old(loc(a, i).val == from)
      ? loc(a, i).val == to
      : loc(a, i).val == old(loc(a, i).val))

method replace(a: Array, left: Int, right: Int,
               from: Int, to: Int)

  requires pre1(a, left, right)
  requires pre2(a, left, right)
  ensures  post1(a, left, right)
  ensures  post2(a, left, right)
{
  if (right - left <= 1) {
    if(loc(a, left).val == from) {
      loc(a, left).val := to
    }
  } else {
    var mid: Int := left + (right - left) \ 2

    //fork-left
    exhale pre1(a, left, mid) && pre2(a, left, mid)

    //fork-right
    exhale pre1(a, mid, right) && pre2(a, mid, right)

    //join-left
    inhale post1(a, left, mid) && post2(a, left, mid)

    //join-right
    inhale post1(a, mid, right) && post2(a, mid, right)
  }
}
```

Listing C.1: The running example from Listing 4.1, encoded in Viper.

Listing C.2 shows a client that uses `replace`, and a heap-dependent boolean function `contains` that yields true if an array contains a given value in the array prefix `[0..before)`. `contains` is intentionally left abstract (that is, it has no body) to demonstrate that the only way of reasoning about the function is via function framing, which indeed allows proving the final assertion.

```
method client(a: Array)
  requires 1 < len(a)
  requires forall i: Int ::
              0 <= i && i < len(a) ==> acc(loc(a, i).val)
  requires contains(a, 5, 1)
{
  replace(a, 1, len(a), 5, 7)
  assert contains(a, 5, 1)  // Requires function framing
}

function contains(a: Array, v: Int, before: Int): Bool
  requires 0 <= before && before <= len(a)
  requires forall i: Int ::
              0 <= i && i < before ==> acc(loc(a, i).val)
```

Listing C.2: Client of the `replace` method from Listing C.1. Function framing allows proving the assertion in method `client`.

Listing C.3 shows an encoding of a graph-marking algorithm in Viper; included as `graph-marking` in the test set described in Section 4.5.

The macro `INV` describes a graph in terms of accessibility predicates and closure properties over a given set of nodes (of the graph): the first three `forall`s denote (quantified) permissions to the fields of each node in the set of nodes, the remaining two `forall`s express that the set of nodes is closed under following the `left` and `right` fields.

```
field left: Ref
field right: Ref
field is_marked: Bool

/* Automatically chosen triggers are not always ideal, using
 * hand-picked triggers can improve performance noticeably, as
 * witnessed by this example.
 */

define INV(nodes)
    !(null in nodes)
  && (forall n: Ref :: n in nodes ==> acc(n.left))
  && (forall n: Ref :: n in nodes ==> acc(n.right))
  && (forall n: Ref :: n in nodes ==> acc(n.is_marked))
  && (forall n: Ref :: {n.left in nodes}{n in nodes, n.left}
        n in nodes && n.left  != null ==> n.left  in nodes)
  && (forall n: Ref :: {n.right in nodes}{n in nodes, n.right}
        n in nodes && n.right != null ==> n.right in nodes)

method trav_rec(nodes: Set[Ref], node: Ref)
  requires node in nodes && INV(nodes)
  requires !node.is_marked

  ensures node in nodes && INV(nodes)

  /* We do not unmark nodes. This allows us to prove that the
   * current node will be marked.
   */
  ensures forall n: Ref :: {n in nodes, n.is_marked}
            n in nodes ==> (old(n.is_marked) ==> n.is_marked)
  ensures node.is_marked
```

```
32
33    /* The nodes is not being modified. */
34    ensures forall n: Ref :: {n in nodes, n.left}
35              n in nodes ==> (n.left == old(n.left))
36    ensures forall n: Ref :: {n in nodes, n.right}
37              n in nodes ==> (n.right == old(n.right))
38
39    /* Propagation of the marker. */
40    ensures forall n: Ref :: {n in nodes, n.is_marked}
41                             {n in nodes, n.left.is_marked}
42      n in nodes ==> (   old(!n.is_marked)
43                      && n.is_marked ==>
44                           (n.left == null || n.left.is_marked))
45    ensures forall n: Ref :: {n in nodes, n.is_marked}
46                             {n in nodes, n.right.is_marked}
47      n in nodes ==> (   old(!n.is_marked)
48                      && n.is_marked ==>
49                           (n.right == null || n.right.is_marked))
50  {
51    node.is_marked := true
52
53    if (node.left != null && !node.left.is_marked) {
54      trav_rec(nodes, node.left)
55    }
56
57    if (node.right != null && !node.right.is_marked) {
58      trav_rec(nodes, node.right)
59    }
60  }
```

Listing C.3: An encoding of a simple graph-marking algorithm in Viper.

Listing C.4 shows a client that asserts that the graph marking algorithm indeed marks all nodes of a connected graph; the same assertions fails in the second client shown in Listing C.5 because its graph contains unconnected nodes.

```
1  method client_success() {
2    var a: Ref; a := new(*); a.is_marked := false
3    var b: Ref; b := new(*); b.is_marked := false
4
5    a.left := b;    a.right := null
6    b.left := null; b.right := a
7
8    var nodes: Set[Ref] := Set(a, b)
9
10   assert forall n: Ref :: n in nodes ==> !n.is_marked
11
12   trav_rec(nodes, a)
13
14   assert forall n: Ref :: n in nodes ==> n.is_marked
15 }
```

Listing C.4: A client of the graph marking algorithm that sets up a small graph and asserts that all nodes in the graph have been marked.

```
1  method client_failure() {
2    var a: Ref; a := new(*); a.is_marked := false
3    var b: Ref; b := new(*); b.is_marked := false
4
5    a.left := a; a.right := a;
6    b.left := a; b.right := a;
7
8    var nodes: Set[Ref] := Set(a, b)
9
```

```
10    assert forall n: Ref :: n in nodes ==> !n.is_marked
11
12    trav_rec(nodes, a)
13
14    /* The assertion is expected to fail because b is in nodes,
15     * but b is not reachable from a
16     */
17    assert forall n: Ref :: n in nodes ==> n.is_marked
18  }
```

Listing C.5: A client of the graph marking algorithm that sets up a small graph, but node b is not reachable from the start node and thus not marked, which makes the final assertion fail.

Listing C.6 demonstrates that the heap management algorithms used in the context of quantified permissions can handle disjunctive aliasing (discussed in Section 3.4.2), unlike the greedy algorithms that Silicon uses per default.

```
1   field val: Int
2
3   define MARK_val_FOR_QP
4     /* Using field val in a quantified permission assertion marks
5      * it as a field for which the QP algorithms are to be used.
6      * Note that the assertion is vacuous and does not specify any
7      * permissions.
8      */
9     forall c: Ref :: false ==> acc(c.val)
10
11  method test01(c1: Ref, c2: Ref, c3: Ref)
12    requires acc(c1.val) && acc(c2.val) && (c3 == c1 || c3 == c2)
13    ensures  MARK_val_FOR_QP
14  {
15    assert acc(c3.val)
16  }
17
18  method test02a(c1: Ref, c2: Ref, c3: Ref)
19    requires acc(c1.val) && acc(c2.val) && (c3 == c1 || c3 == c2)
20    ensures  MARK_val_FOR_QP
21  {
22    c3.val := 0
23  }
```

Listing C.6: The disjunctive aliasing tests from Listing B.10 succeed if the heap management algorithms for quantified permissions are used. Referenced on page 113.

Appendix D

Magic Wands

```
1  field val: Int
2  field next: Ref
3
4  predicate list(ys: Ref) {
5       acc(ys.val) && acc(ys.next)
6    && (ys.next != null ==> acc(list(ys.next)))
7  }
8
9  function sum_rec(ys: Ref): Int
10   requires acc(list(ys))
11 {
12   unfolding acc(list(ys)) in
13     ys.val + (ys.next == null ? 0 : sum_rec(ys.next)) }
14
15 method sum_it(ys: Ref) returns (sum: Int)
16   requires ys != null && acc(list(ys))
17   ensures  acc(list(ys))
18   ensures  sum == old(sum_rec(ys))
19 {
20   var xs: Ref := ys
21   sum := 0
22
23   define A xs != null ==> acc(list(xs))
24   define B acc(list(ys))
25
26   package A --* B
27
28   while (xs != null)
29     invariant xs != null ==> acc(list(xs))
30     invariant A --* B
31     invariant sum ==   old(sum_rec(ys))
32                      - (xs == null ? 0 : sum_rec(xs))
33   {
34     wand w := A --* B /* Give magic wand instance the name w */
35
36     var zs: Ref := xs /* Value of xs at start of iteration */
37     unfold acc(list(xs))
38     sum := sum + xs.val
39     xs := xs.next
40
41     package A --* folding acc(list(zs)) in applying w in B
42   }
43
44   apply A --* B
45 }
```

Listing D.1: The running example from Chapter 5: a straightforward iterative implementation to calculate the sum of the nodes in a linked list. Referenced on page 129.