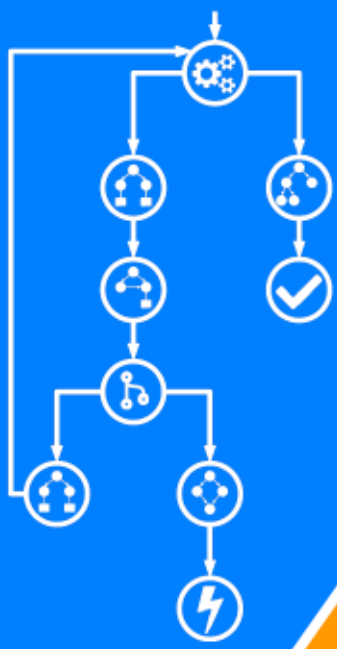
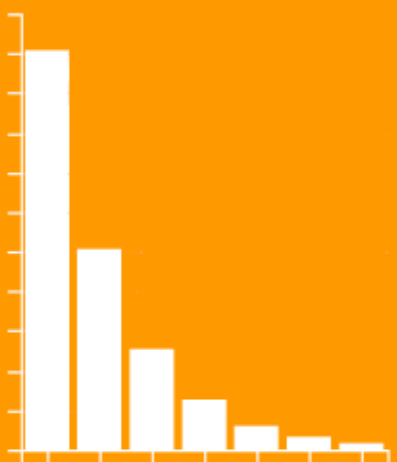
Automated Reasoning and Randomization in Separation Logic

Christoph Matheja



```
// 1/2 · [head != 0]
//     · len(hd,0)
rev := 0;
while(head != 0) {
 x := head.next;
 {
  head.next := rev;
 rev := head
} [1/2] {
 free(head)
};
head := x
}
// len(rev, 0)
```



Christoph Matheja
https://orcid.org/0000-0001-9151-0441

First Printing, January 19, 2020

AUTOMATED REASONING AND RANDOMIZATION IN SEPARATION LOGIC

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Christoph Matheja, M.Sc.

aus

Oelde

Berichter: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Dr. habil. Radu Iosif

Tag der mündlichen Prüfung: 9.1.2020

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Abstract

We study three aspects of program verification with separation logic:

1. Reasoning about *quantitative* properties, such as the probability of memory-safe termination, of *randomized heap-manipulating programs*.

2. *Automated reasoning* about the *robustness* of and *entailments* between formulas in the symbolic heap fragment of separation logic itself.

3. *Automated reasoning* about pointer programs by combining *abstractions* based on separation logic with the above techniques and *model checking*.

Regarding the first item, we extend separation logic to reason about *quantities*, which evaluate to real numbers, instead of predicates, which evaluate to Boolean values. Based on the resulting *quantitative separation logic*, we develop a weakest precondition calculus à la Dijkstra for quantitative reasoning about randomized heap-manipulating programs. We show that this calculus is a sound and conservative extension of both separation logic and McIver and Morgan's weakest preexpectations which preserves virtually all properties of classical separation logic. We demonstrate its applicability by several case studies.

Regarding the second item, we develop an *algorithmic framework* based on *heap automata* to compositionally check robustness properties, e.g., satisfiability or acyclicity, of symbolic heaps with inductive predicate definitions. We consider two approaches to discharge entailments for fragments of separation logic. In particular, this includes a pragmatic decision procedure with nondeterministic polynomial-time complexity for entailments between *graphical symbolic heaps*.

Regarding the third item, we introduce Attestor—an automated verification tool for analyzing Java programs operating on dynamic data structures. The tool involves the generation of an abstract state space employing inductive predicate definitions in separation logic. Properties of individual states are defined by heap automata. LTL model checking is then applied to this state space, supporting the verification of both *structural* and functional correctness properties. Attestor is fully automated, procedure modular, and provides informative visual feedback including counterexamples for violated properties.

Kurzfassung

Wir studieren drei Aspekte der Programmverifikation mit Separation Logic (SL):

1. Die Analyse *quantitativer* Eigenschaften, wie z.B. die Wahrscheinlichkeit der Terminierung ohne Speicherfehler, von *probabilistischen Programmen*.
2. Die *automatisierte Analyse* der *Robustheit* von und *Implikationsbeziehungen* zwischen Formeln im symbolischen Heap-Fragment von Separation Logic.
3. Die *automatisierte Analyse* von Zeigerprogrammen durch Kombination von SL-basierten *Abstraktionen* mit den obigen Techniken und *Model Checking*.

Bezüglich des ersten Punktes erweitern wir SL zu einer *quantitativen Separation Logic* (QSL) zur Analyse von *Quantitäten*, die zu reellen Zahlen ausgewertet werden, anstelle von Prädikaten, die zu Wahrheitswerten ausgewertet werden. Auf Grundlage von QSL entwickeln wir einen Kalküls der schwächsten Vorbedingungen à la Dijkstra, der praktisch alle klassichen Eigenschaften beibehält. Wir zeigen, dass dieser Kalkül eine korrekte und konservative Erweiterung sowohl von SL als auch der schwächsten Vorerwartungen von McIver and Morgan ist. Wir demonstrieren seine Anwendbarkeit anhand mehrerer Fallstudien.

Bezüglich des zweiten Punktes entwickeln wir ein *algorithmisches Grundgerüst* auf Basis von *Heap-Automaten* um Robustheitseigenschaften, z.B. Erfüllbarkeit oder Azyklizität, von symbolischen Heaps mit induktiven Definitionen nachzuweisen. Wir betrachten zwei Ansätze um Implikationen in Fragmenten von SL zu zeigen. Dies umfasst einen Algorithmus für Implikationen zwischen *grafischen symbolischen Heaps*, der in nichtdeterministischer Polynomialzeit läuft.

Bezüglich des dritten Punktes, stellen wir ATTESTOR vor - ein Werkzeug zur automatisierten Analyse von Java Programmen, die mit dynamischen Datenstrukturen arbeiten. Dies beinhaltet die Generierung eines abstrakten Zustandsraumes unter Verwendung induktiver Definitionen in SL. Eigenschaften einzelner Zustände werden durch Heap-Automaten beschrieben. Ein Model Checker ermöglicht dann die Verifikation von *struktureller* und funktionaler Korrektheit. ATTESTOR ist vollautomatisch, modular und liefert aussagekräftiges visuelles Feedback inklusive Gegenbeispielen falls die Verifikation fehlschlägt.

Acknowledgements

This thesis would not have been possible had it not been for the support of my advisors, colleagues, friends, and family. During my time both as a member of the Software Modeling and Verification Group and as a student at RWTH Aachen, I had and hopefully will continue to have the privilege to work with many wonderful people to whom I would like to express my appreciation.

First and foremost, I am grateful to my supervisor Joost-Pieter Katoen for hiring me as a Ph.D. student within the Attestor project and then giving me the freedom to also work on something completely different. All of his advice— in particular regarding research on probabilistic programs—was invaluable. His constant drive to educate both experts in the field and students about our results is highly appreciated and created many opportunities. For example, I very much enjoyed assisting him in the first two iterations of a new course on probabilistic programming. Speaking of opportunities, Joost-Pieter also enabled me to attend various workshops, conferences, and summer schools of which I still have fond memories. I would also like to thank Joost-Pieter and all colleagues for creating a marvelous working environment.

Special thanks go to Thomas Noll for suggesting to work on verifying pointer programs. I am grateful to him for always having an open door, many discussions, and the constructive remarks I received on various projects.

I am indebted to Benjamin Kaminski for getting me excited about probabilistic programs and standing up for mathematical purity. I learned a lot from our very fruitful collaboration on reasoning about probabilistic programs which hopefully will continue in the future. I also had the pleasure of working on the same topics with Federico Olmedo—who had an amazing overview of the literature—and our student-turned-colleague Kevin Batz.

Many thanks go to Jens Katelaan[1] from TU Vienna for a longstanding collaboration and his commitment to turn heap automata into a tool for reasoning about separation logic. I thank Christina Jansen for getting me excited about separation logic and our joint work on analyzing pointer programs. I also

[1] Actually, it is Jens Pagel now.

appreciate the contributions of my years-long student assistant Hannah Arndt to the Attestor project.

I thank my co-authors Alejandro Aguirre, Gilles Barthe, Justin Hsu, Maurice van Keulen, and Florian Zuleger for many enlightening discussions. Furthermore, I appreciate the detailed and constructive comments I received from my external examiner Radu Iosif before publication of this thesis.

I owe thanks to my fellow Ph.D. students Sebastian Junges, who made the final months of writing and preparing the defense much more enjoyable, Tim Quatmann, who has been a truly great office mate, and Matthias Volk, who frequently assisted me with organizational matters. My gratitude extends to current and former colleagues Philipp Berger, Harold Bruintjes, Souymodip Chakraborty, Mingshuai Chen, Friedrich Gretz, Marcel Hark, Christian Hensel, Nils Jansen, Mojgan Kamali, Shahid Khan, Lutz Klinkenberg, Tim Felix Lange, Joshua Moerman, Bahare Salmani, Falak Sher, Jip Spel, and Marcin Szymczak. All of you were helpful, open, and fun to talk to both at and after work.

Above all, I thank my parents Heidemarie and Hubert Matheja for their continuous love and support. They kept me sane while writing this thesis as well as throughout my ten years in Aachen.

Finally, I would like to thank my friends and family members who supported me throughout the years. Honorable mentions go to my fellow computer science undergraduate students Sebastian Chrobak, Roland Moers, Julian Alexander Richter, Sabrina Schulte, and Hannah Spitzer. Without you, studying would have been much less fun. I appreciate that so many of you came back to Aachen to attend and celebrate my defense. The same extends to my decades-long friend Michael Winter who traveled all-day to join.

Contents

Chapter 1

Introduction

> When we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
>
> Edsger W. Dijkstra,
> ACM Turing Award Lecture, 1972.

European regulators often require safety-critical processes, industrial plants, and consumer goods to comply with the "state of the art" in their respective fields of engineering.[1] However, it appears that *software* is rarely held accountable to a strict interpretation of the same standards. This is surprising since software is nowadays involved in virtually all of the aforementioned items. In fact, failures of prominent critical systems have frequently been attributed to software issues. Examples include, but are not limited to, medical devices [LT93], self-driving cars [Eco18], civil planes [Tra19], military planes [Reg07], airports [BBC17], space faring rockets [Lio96], university campus management systems [Aac18], and secure mail for confidential court documents [Leg19].

One explanation for this discrepancy is that software differs significantly from other engineering artifacts. It is neither subject to physics, e.g., inertia and material properties, nor inherently continuous: Changing a few bits might lead to vastly different results. Established methodologies applied by testing engineers in other fields consequently do not yield comparable levels of confidence in the correctness of software. Dijkstra [Dij72] famously summarized the issue:

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*

[1] An explicit reference to the state of the art ("Stand der Technik") is found, e.g., in §34 of the German product liability law ("Gesetz über die Bereitstellung von Produkten auf dem Markt") [Bun15]. Similar statutes are found in the product liability directive of the European Union.

His remarks continue with a less commonly cited observation:

> *"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."*

1.1 Formal Methods

Rather than considering—and evaluating—software as the product of yet another engineering discipline, *formal methods* treat the correctness of programs, i.e., their compliance with some specification, as *mathematical theorems*. The goal is then to derive a mathematical proof of these theorems under some well-understood baseline assumptions, such as the correctness of the underlying compiler, operating system, and hardware. Formal methods *can* prove the absence of certain errors—at least under the proviso that the error is covered by the specification and the baseline assumptions are realistic. As such, they yield far stronger guarantees than testing can provide.

A caveat though is that there are differences between proving theorems in mathematics and proving the correctness of software. As pointed out by Kurt Mehlhorn, mathematics has been for many centuries and still is a "complex social process".[2] That is, apart from proofs written on paper, it requires reviews, presentations, questioning, revision, and expert discussions until a novel theorem gains acceptance in the scientific community. It is hard to imagine that the implementation of an ordinary commercial software product, which is substantially larger, more detailed, and—to many people—less elegant and less interesting than a mathematical theorem, will receive the same amount of rigor.

In contrast to mathematics, effective reasoning about the correctness of programs thus additionally relies on both mechanized and automated tool support. Ideally, this means that formal methods address the following wishlist:

First, formal methods must be *sound* in order to raise the confidence level in software.[3] In particular, to avoid being subject to a complex social process, formal proofs have to provide sufficient details such that their correctness can be verified automatically. The latter task is considered, for instance, by interactive theorem provers such as Isabelle/HOL [NPW02] and Coq [BC04; Coq19].

Second, formal methods should be *compositional* to increase their scalability and support the reuse of already proven compartmentalized components.

Third, it is desirable that formal methods are amenable to *automation* as much as possible. While automation is impossible in general [Tur37], it is a goal to strive for. Automation, however, provides no relief from the first item. That

[2] The remark was made during his keynote at MFCS 2019 in Aachen.

[3] Unsound formal methods are also useful, e.g., for detecting potential flaws and debugging. However, for the moment, we focus on proving correctness.

is, rather than stating that a given program is correct, automated tools should provide machine-checkable evidence for their results.

Fourth, various software products have to interact with an environment such as noisy sensory data, (unpredictable) human users, or unreliable low-quality hardware. In these cases, formal methods need to be able to deal with *uncertainty*. This includes *quantifying* the level of confidence we should put in an implementation. For instance, a typical statement of interest in this context is "the likelihood of producing the specified result is at least 0.9."

We will give a more detailed introduction to classical formal methods for reasoning about the correctness of programs in Part I of this thesis.

1.2 Separation Logic

Many software issues can be traced back to the erroneous usage of dynamically allocated memory on the heap. This includes, for instance, dereferencing null pointers, buffer and heap overflows, and accidentally invalidating data structure invariants. Building on early ideas of Burstall [Bur72], *separation logic* [IO01; Rey02; OHe19] is both a logical language and a proof system for formal reasoning about heap manipulating programs which is quite compliant with the first three items in the above wishlist for formal methods:

- Being a formal program logic, it is sufficiently precise such that proofs in separation logic are verifiable by machines. In fact, separation logic and many of its extensions have been formalized in interactive theorem provers; for instance, as part of the Iris project [Kre+17; KTB17].

- Its characteristic connective—the *separating conjunction* $\star$—intuitively states that two specifications hold in separate regions of memory; they may thus be considered and proven in isolation. Compositionality consequently lies at the heart of formal verification with separation logic.

- As separation logic champions *local reasoning*, it admits a large degree of automation for certain correctness properties. Infer, for example, is capable of proving memory safety, i.e., the absence of illegal pointer dereferences, fully automatically on industrial codebases [Cal+11; CD11]; as such, it is deployed within large software companies such as Facebook.

Automated reasoning with separation logic about more intricate correctness properties than memory safety—for instance the restoration of the heap's initial shape upon termination—remains an active research area.

We will give a more detailed introduction to separation logic in Chapter 4 of this thesis. Moreover, automated reasoning will be discussed in Part III.

1.3 Randomization & Probabilistic Programs

Let us also have a brief look at the last item on our wishlist: Applying formal methods in uncertain environments. One possible approach for addressing this task, which has been recently considered in the context of separation logic [1; TH19; BHL19], is to consider *probabilistic programs*, i.e., programs with the ability to sample from probability distributions. Unreliable data or behavior is then modeled by sampling from an (empirically determined) probability distribution which takes noise, common user behavior, or the reliability of underlying hardware components into account. Formal methods in this context aim to *quantify* a program property rather than proving its correctness. Examples of such quantitative properties include the probability of terminating with a correct result or the expected, i.e., average, value produced by a program.

Probabilistic programs are, however, not limited to modeling uncertainty. Perhaps the oldest application of probabilistic programs lies in the implementation of *randomized algorithms* [Rab76; MR97; MU05]—a well-established branch of algorithms research which attempts to exploit sampling to improve an algorithm's performance possibly at the cost of producing wrong results with a small probability. In this case, sampling from a probabilistic distribution is not a modeling construct, but an essential programming instruction. Reasoning about probabilistic programs hence lies at the foundation of understanding the correctness of randomized algorithms.

More recently, probabilistic programs have received a lot of attention in the machine learning community as a more expressive alternative to classical probabilistic graphical models (cf. [GS14; Gor+14b; Car+17; Bin+19]). In this setting, the main appeal of probabilistic programs is that they decouple writing probabilistic models from the design of efficient inference algorithms for analyzing these models; they thus enable rapid prototyping for non-experts. Formal methods for probabilistic programs might thus also contribute to understanding, debugging, and raising the level of confidence in probabilistic models.

We will give a detailed introduction to probabilistic programs and their applications in Part II of this thesis.

1.4 Contributions and Synopsis

The goals of this thesis are threefold. First, we give a consistent overview of the developments of classical program verification techniques which—starting with Turing, Floyd, Hoare, and Dijkstra [Tur49; Flo67; Hoa69; Dij76]—led to two advanced formalisms: On the one hand, *separation logic* has been developed by, amongst others, Ishtiaq, O'Hearn, Reynolds, and Yang [IO01; ORY01; Rey02], to reason about *heap manipulating* programs. On the other hand, the *weakest*

preexpectation calculus has been developed by, amongst others, Kozen, McIver, and Morgan [Koz81; Koz83; MMS96; MM05] to reason about *probabilistic* programs.

Second, in an attempt to unify both formalisms, we develop a *probabilistic extension of separation logic* which broadens separation logic's applicability to both uncertain environments and randomized algorithms.

Third, we try to improve *automation of formal methods*. To this end, we study decision procedures for reasoning *about* separation logic and report on a tool for automated program verification *with* separation logic.

In summary, the main contributions of this thesis are:

(I) We give a textbook-style *survey of classical techniques* for reasoning about imperative programs through their operational semantics and computing weakest preconditions à la Dijkstra [Dij76]. In particular, we gradually discuss how recursive procedures, pointers, dynamic memory allocation, and sampling from probability distributions are incorporated.

(II) We present a novel *quantitative separation logic* for formal verification of probabilistic programs. Our logic conservatively extends and preserves virtually all properties of classical separation logic. Its formulas, however, evaluate to real numbers rather than Boolean values. This enables reasoning about both probabilities and expected values of probabilistic programs.

(III) We demonstrate the applicability of quantitative separation logic in several *case studies*. In particular, this includes both probabilistic versions of classical examples from the separation logic literature and textbook examples of randomized algorithms, e.g., randomized meldable priority queues.

(IV) We develop an *algorithmic framework for automated reasoning about the robustness* of inductive definitions in the popular symbolic heap fragment of separation logic. In particular, we present a refinement theorem which lies at the foundation of both asymptotically optimal decision procedures for robustness and the automated synthesis of robust inductive definitions.

(V) We investigate the *entailment problem* for the symbolic heap fragment of separation logic. In particular, we present a pragmatic decision procedure in nondeterministic polynomial time for certain "graphical" symbolic heaps.

(VI) We report on the implementation of Attestor—a tool for automated verification of intricate correctness properties of Java pointer programs.

1.4.1 How to Read this Thesis

Before we provide a more detailed synopsis, let us give a few remarks on reading this thesis. While most chapters should be sufficiently self-contained

or at least equipped with enough backward references such that they can be read individually, there are some dependencies. A rough guide respecting the order induced by these dependencies is illustrated in Figure 1.1, page 7. Here, a thick arrow between two chapters indicates frequent use of previously defined concepts; it is thus recommended to have a look at these concepts first. A thin arrow indicates a local dependency that exists only for a few individual results; it is thus safe to ignore the dependency and return to it once it is actually referenced. Furthermore, notice that Parts II and III can be read in arbitrary order. Each part is equipped with its own introduction, discussion of related work, and conclusion.

1.4.2 Part I: Foundations of Reasoning about Pointer Programs

The first part of this thesis is a gentle introduction to classical techniques for reasoning about imperative programming languages, i.e., *operational semantics*, program verification with *weakest preconditions*, and *separation logic*. Apart from providing a consistent view on these topics, there are no original contributions. Rather, Part I provides the foundations for Parts II and III.

A reader familiar with the aforementioned topics might thus want to briefly skim this part to familiarize herself with our notation.

Chapter 2: Reasoning about Imperative Programs We introduce the syntax and operational semantics of the simple imperative programming language **PL**. This language will be gradually extended by additional features throughout this thesis. Furthermore, we give a brief tour through two formal methods: Program analysis using *abstraction* and *Floyd-Hoare style program verification*.

Chapter 3: Reasoning about Recursive Procedures We study how procedures with parameters and return values are incorporated into the programming language **PL**; as a result, we obtain the *procedural* programming language $\mathbf{P^2L}$. Special attention is paid to *proof rules* for verifying recursive procedures. As long as one is willing to ignore the treatment of procedures in follow-up chapters as well, one may safely skip this chapter on a first reading.

Chapter 4: Reasoning about Pointer Programs We extend the language $\mathbf{P^2L}$ by pointers and dynamic memory allocation; this yields the procedural *pointer* programming language $\mathbf{P^3L}$. Moreover, we discuss the challenges faced when attempting to formally verify $\mathbf{P^3L}$ programs. After that, we formally define *separation logic* both as an *assertion language* and as a *verification system* using weakest preconditions à la Ishtiaq and O'Hearn [IO01]. All remaining chapters build upon these definitions in one way or another.

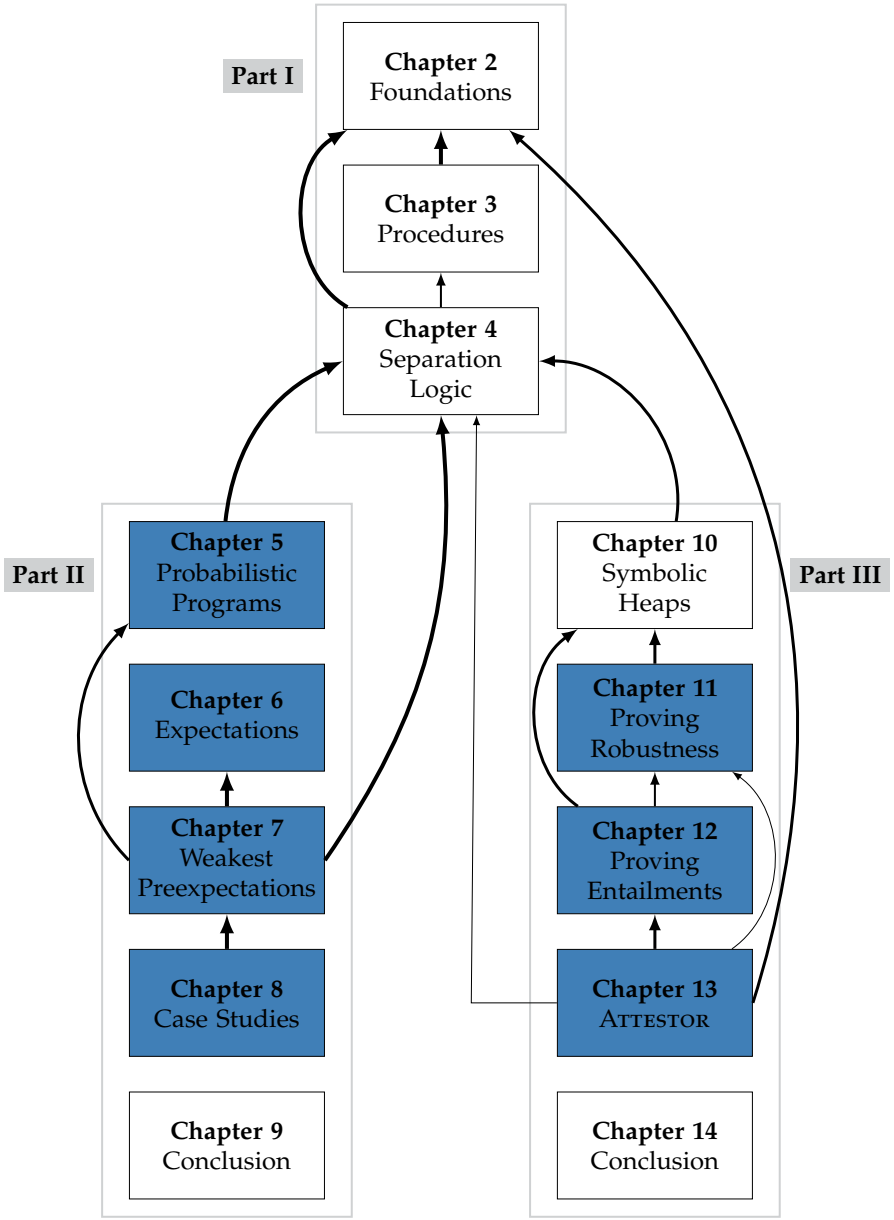


Figure 1.1: A rough guide to reading this thesis. Blue chapters contain original contributions by the author.

1.4.3 Part II: Randomization in Separation Logic

In the second part of this thesis, we develop an extension of separation logic—called *quantitative* separation logic (QSL)—for *quantitative reasoning* about *probabilistic pointer programs*. Part II covers and extends the work published in:

[1] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 34:1–34:29

Chapter 5: Reasoning about Probabilistic Pointer Programs We first give a more elaborate motivation for considering probabilistic programs and, in particular, probabilistic *pointer* programs. We also discuss the relevant literature related to the development of QSL. We introduce the *probabilistic* procedural pointer programming language $\mathbf{P^4L}$ and define its operational semantics in terms of *Markov decision processes*. Finally, we briefly discuss how the operational semantics enables reasoning about quantitative properties of $\mathbf{P^4L}$ programs.

Chapter 6: Quantitative Separation Logic: Assertion Language We introduce *expectations* that evaluate to real numbers rather than predicates that evaluate to truth values as the *assertion language* of QSL. In particular, we define a *quantitative separating conjunction* and a *quantitative separating implication*; we thus lift the characteristic connectives of separation logic to expectations. We also study various *algebraic properties* of expectations and show that virtually all properties of classical separation logic predicates are preserved by expectations.

Chapter 7: Quantitative Separation Logic: Verification System We develop a weakest precondition calculus for reasoning about both *probabilities* and *expected values* of probabilistic pointer programs in $\mathbf{P^4L}$. Our calculus conservatively extends both Ishtiaq's, O'Hearn's and Reynolds's separation logic [IO01; Rey02] for heap manipulating programs as well as Kozen's [Koz83] / McIver and Morgan's [MMS96; MM05] weakest preexpectations for probabilistic programs. Particular attention is paid to *soundness* and *local reasoning*.

Chapter 8: Case Studies We demonstrate the applicability of quantitative separation logic by means of five case studies. We verify the *correctness* of a textbook algorithm for computing random permutations. We use QSL to analyze the *expected performance* of a randomized procedure which appears in efficient implementations of priority queues, namely randomized meldable heaps [GM98]. Moreover, we consider *unreliable* variants of classical benchmarks for program verification with separation logic.

Chapter 9: Conclusion and Future Work We wrap up our work on quantitative separation logic and discuss possible directions for future work.

1.4.4 Part III: Automated Reasoning

In the third part of this thesis, we consider *automated reasoning* about heap manipulating programs using separation logic. This involves the design of algorithms for both reasoning *with* separation logic about programs as well as reasoning *about* separation logic itself.

Chapter 10: Towards Automated Reasoning We motivate the challenges for automated reasoning with separation logic. We also discuss the *decision problems* arising in this context together with related work. After that, we lay the groundwork for automated reasoning. We define a formal syntax for separation logic and, in particular, introduce separation logic's *symbolic heap fragment with inductive predicate definitions*. The latter is considered in all remaining chapters.

Chapter 11: Automated Reasoning about Robustness of Symbolic Heaps We study the question whether a given symbolic heap is *robust*, i.e., whether it satisfies a property of interest such as satisfiability, reachability, or acyclicity. We develop an *algorithmic framework* around the notion of *heap automata* and a *refinement theorem* which allows us to synthesize robust inductive definitions. We consider several examples of robustness properties and show that our algorithmic framework yields decision procedures with optimal asymptotic complexity. Moreover, we briefly report on an *implementation* of our framework. Chapter 11 is based in part on:

[2] Christina Jansen, Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "Unified Reasoning about Robustness Properties of Symbolic-Heap Separation Logic". In: *European Symposium on Programming (ESOP)*. vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 611–638.

Chapter 12: Automated Reasoning about Entailments We present two attempts for deciding the *entailment problem* for symbolic heaps, i.e., the question whether one symbolic heap implies another one. We briefly outline how heap automata can—in principle—be applied to discharge entailments. Furthermore, we study the fragment of *graphical symbolic heaps*. In particular, we present a *pragmatic decision procedure* in nondeterministic polynomial time for entailments between certain graphical symbolic heaps. Although they are put in a different context, some results in Chapter 12 are based on:

[3] Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll. "Graph-Based Shape Analysis Beyond Context-Freeness". In: *Software Engineering and Formal Methods (SEFM)*. vol. 10886. Lecture Notes in Computer Science. Springer, 2018, pp. 271–286.

Chapter 13: Attestor: Model Checking Java Pointer Programs We report on the implementation of Attestor—an automated verification tool for reasoning about Java pointer programs. We describe how Attestor's underlying *abstract domain* can be understood in terms of graphical symbolic heaps. Moreover, we give a brief tour through Attestor's features and present *experimental results*. Chapter 13 is mainly based on:

[4] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Let this Graph Be Your Witness! - An Attestor for Verifying Java Pointer Programs". In: *Computer Aided Verification (CAV), Part II*. vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 3–11.

Chapter 14: Conclusion and Future Work We conclude our work on automated reasoning with separation logic and discuss future work.

1.4.5 Part IV: Appendix

We supplement the main part of this thesis by several appendices providing, amongst others, mathematical preliminaries, omitted proofs, and detailed calculations. More precisely, Appendix A contains basic preliminaries on domain theory. Appendix B provides detailed calculations for an example presented in Part I. In Appendix C, we give selected proofs which have been omitted in Part II. Appendix D is a reference sheet with all proof rules for QSL expectations. Some of these rules can be derived from other ones; the derivations are found in Appendix E. All calculations required for our case studies on QSL are collected in Appendix F. Appendix G defines basic notation for multisets. Finally, in Appendix H, we provide selected proofs omitted in Part III.

1.5 Publications

This section contains a comprehensive list of all peer-reviewed publications which I coauthored during my time as a PhD student at RWTH Aachen University. For convenience, the list is divided into three categories: First, I list publications whose contributions are covered by this thesis. Second, I give a list of publications that are closely related to the topics of this thesis although they

are not considered in detail. A few results from these publications have been adapted to fit in the context of this thesis and are cited accordingly. The third category contains all other publications. A discussion of my own contributions to the publications covered in this thesis—which I am obliged to provide by the university's regulations for doctoral studies—is found on page 451.

1.5.1 Publications Covered in this Thesis

[1] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 34:1–34:29

[4] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Let this Graph Be Your Witness! - An Attestor for Verifying Java Pointer Programs". In: *Computer Aided Verification (CAV), Part II*. vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 3–11

[2] Christina Jansen, Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "Unified Reasoning about Robustness Properties of Symbolic-Heap Separation Logic". In: *European Symposium on Programming (ESOP)*. vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 611–638

1.5.2 Publications Relevant for this Thesis

[5] Jens Katelaan, Christoph Matheja, and Florian Zuleger. "Effective Entailment Checking for Separation Logic with Inductive Definitions". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*. vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 319–336

[6] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms". In: *Journal of the ACM* 65.5 (2018), 30:1–30:68

[3] Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll. "Graph-Based Shape Analysis Beyond Context-Freeness". In: *Software Engineering and Formal Methods (SEFM)*. vol. 10886. Lecture Notes in Computer Science. Springer, 2018, pp. 271–286

[7] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "How long, O Bayesian network, will I sample thee? - A program

analysis perspective on expected sampling times". In: *European Symposium on Programming (ESOP)*. vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213

[8] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "A Program Analysis Perspective on Expected Sampling Times". In: *Extended Abstracts of the First International Conference on Probabilistic Programming (PROBPROG)*. 2018

[9] Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "HARRSH: A Tool for Unified Reasoning about Symbolic-Heap Separation Logic". In: *International Workshop on the Implementation of Logics (IWIL)*. vol. 9. LPAR-22 Workshop and Short Paper Proceedings. 2018, pp. 23–36

[10] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs". In: *European Symposium on Programming (ESOP)*. vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389. (best paper award)

1.5.3 Further Publications

[11] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Analyzing Expected Runtimes by Program Verification". In: *Book on Probabilistic Programming resulting from the First School on Foundations of Programming and Software systems (FOPPS 2017)*. Ed. by Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. [to appear]

[12] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "On the Hardness of Analyzing Probabilistic Programs". In: *Acta Informatica* 56.3 (2019), pp. 255–285

[13] Mihaela Sighireanu et al. "SL-COMP: Competition of Solvers for Separation Logic". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part III*. vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 116–132

[14] Maurice van Keulen, Benjamin Lucien Kaminski, Christoph Matheja, and Joost-Pieter Katoen. "Rule-Based Conditioning of Probabilistic Data". In: *Scalable Uncertainty Management (SUM)*. vol. 11142. Lecture Notes in Computer Science. Springer, 2018, pp. 290–305. (best paper award)

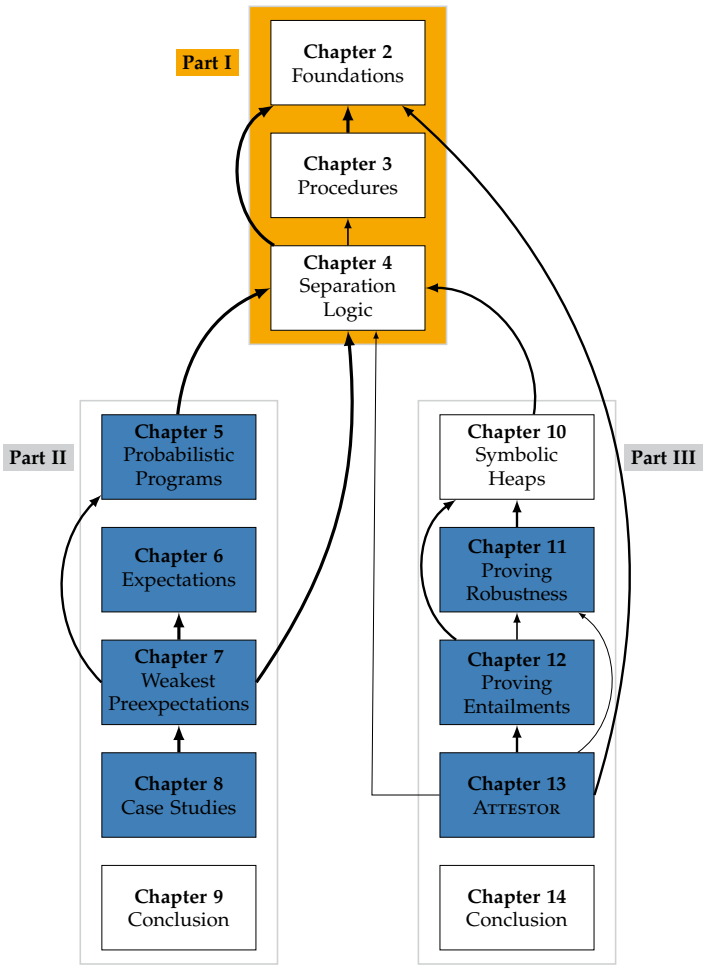
[15] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Reasoning about Recursive Probabilistic Programs". In: *Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 672–681

[16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Inferring Covariances for Probabilistic Programs". In: *Quantitative Evaluation of Systems (QEST)*. vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206

[17] Christoph Matheja, Christina Jansen, and Thomas Noll. "Tree-Like Grammars and Separation Logic". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. vol. 9458. Lecture Notes in Computer Science. Springer, 2015, pp. 90–108

Part I

Foundations of Reasoning about Pointer Programs



Chapter 2

Reasoning about Imperative Programs

The demand for reasoning about programs arose simultaneously with the early development of computers. For instance, in 1949, Alan M. Turing [Tur49] asked: "How can one check a routine in the sense of making sure that it is right?"

This chapter is a brief tour through fundamental principles for reasoning about the correctness of programs. By correctness, we mean that a program has some desirable property—for example, the program computes the intended output, it eventually terminates, or it never crashes due to a memory error. While none of the presented techniques are new, we will build upon each of them in follow-up chapters to reason about more powerful programming languages and to perform automated reasoning. Furthermore, this chapter serves us to introduce our notation and compare the main ideas of each approach in a simplified setting before we move to advanced topics.

Approaches for reasoning about program correctness come in three flavors, although the division between them is often seamless:

Executing a Program The simplest approach to reason about a program is to execute it and then check whether the result is consistent with the desired property. This is also known as testing which is an effective way to show that a program is incorrect, but inadequate to show that a program satisfies a property [Dij72]. In Section 2.1, we introduce a bare-bones programming language that is used throughout this thesis. The meaning of programs is defined in terms of an operational semantics, which mathematically describes how programs are executed step-by-step on a conceptual machine. Testing then corresponds to computing steps in our operational semantics.

Program Analysis The goal of program analysis is to discover approximations of program behaviors that occur during execution (cf. [NNH99]). Instead of running a program on a single input, we are thus interested in determining properties that hold for a large set of or even all possible inputs. These properties may serve as the basis for optimization or can be used to check correctness. Program analysis usually performs some kind of abstraction, i.e., computations

take place in an abstract domain instead of on an actual (or conceptual) machine. Discovered properties then correspond to the results of abstract computations. In Section 2.2, we discuss how abstraction—in the sense of simulations [Mil71; BK08]—are applied to perform program analysis. This is a special case of a general theory for abstraction and designing program analyses which is known as abstract interpretation [CC77; CC92; CC14].

Program Verification The aim of program verification is to prove that a given program satisfies a property of interest. To this end, the central notion is a specification which expresses the properties that we may assume to hold before and that we have to establish after program execution. Specifications are also called Floyd-Hoare triples in recognition of the seminal works of Floyd [Flo67] and Hoare [Hoa69], who developed rules to prove that a specification is correct. In Section 2.3, we consider Floyd-Hoare triples in detail. Furthermore, we discuss Dijkstra's weakest precondition calculus [Dij76] which offers a systematic way to derive correct specifications.

2.1 The Programming Language

We present techniques for reasoning about programs with respect to a small programming language which, for lack of a better name, is called *The Programming Language*, **PL** for short. **PL** is essentially a stripped-down version of Dijkstra's Guarded Command Language [Dij76] with stricter control flow statements to eliminate nondeterminism. Its syntax roughly corresponds to both Imp [Win93] and While [NN92]; two simplistic—yet Turing complete—languages, that support the characteristic features of virtually every imperative programming language. Throughout this thesis, we will gradually extend **PL**:

1. In Chapter 3, we add support for recursive procedures; this leads to the *Procedural Programming Language* ($\mathbf{P^2L}$).

2. In Chapter 4, we extend $\mathbf{P^2L}$ by instructions to manipulate dynamically allocated memory via pointers; this leads to the *Procedural Pointer Programming Language* ($\mathbf{P^3L}$).

3. In Chapter 5, we endow $\mathbf{P^3L}$ programs with instructions to sample from probability distributions; this leads to the *Probabilistic Procedural Pointer Programming Language* ($\mathbf{P^4L}$).

For each of the above languages, we discuss both the language's semantics and techniques to reason about the correctness of programs.

2.1.1 Syntax

To conveniently define the syntax of our programming language **PL**, we adhere to the following conventions:

- We assume a countably infinite set **Vars** of *program variables*. Variables in **Vars** are usually denoted by lowercase letters, such as x, y, z, or descriptive names, such as counter, pivot, etc.
- By E we denote an *arithmetic expression* over variables in **Vars** evaluating to an integer in $\mathbb{Z}$. Instead of defining a concrete syntax for arithmetic expressions, e.g., basic integer arithmetic as in [Win93, Chapter 2], we are fairly liberal: Any computable function over variables that evaluates to an integer may be used as an arithmetic expression.
- By B we denote a *Boolean expression*—also called a *guard*—over variables evaluating to a truth value in $\mathbb{T} \triangleq \{ \mathsf{true}, \mathsf{false} \}$. As for arithmetic expressions, any computable function over variables, which yields a truth value, is permitted as a Boolean expression. A simple example of a syntax for guards is found in [Win93, Chapter 2].
- Finally, by C (command), we refer to some program in our language.

With these conventions in mind, the set **PL** of programs written in *The Programming Language* is given by the context-free grammar below:

$$
\begin{array}{rcll}
C & \to & \texttt{skip} & \text{(effectless program)} \\
& | & x := E & \text{(assignment)} \\
& | & C \,;\, C & \text{(sequential composition)} \\
& | & \texttt{if}\ (\, B \,)\ \{\, C \,\}\ \texttt{else}\ \{\, C \,\} & \text{(conditional choice)} \\
& | & \texttt{while}\, (\, B \,)\, \{\, C \,\} & \text{(loop)}
\end{array}
$$

The intuitive meaning of each **PL** statement is straightforward: `skip` has no effect. An assignment $x := E$ sets the value of variable x to the value corresponding to an evaluation of expression E. The sequential composition $C_1 \,;\, C_2$ first executes program C_1 and—upon termination of C_1—continues by executing program C_2. The conditional choice $\texttt{if}\ (\, B \,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}$ executes program C_1 if guard B is evaluated to true. Otherwise, program C_2 is executed. Finally, the loop $\texttt{while}\, (\, B \,)\, \{\, C \,\}$ immediately terminates without effect if its loop guard B evaluates to false. Otherwise, we first execute the loop body C and then execute the whole loop again.

Example 2.1 Consider the **PL** program C_{odd} below, where the labels C_{loop} and C_{if} are not part of the program and serve solely to refer to the loop and the conditional statement, respectively.

```
          y := 0;
C_loop:   while ( x > 0 ) {
C_if:         if ( x mod 2 = 1 ) {
                  y := y + x
              } else {
                  skip
              };
              x := x − 1
          }
```

Program C_{odd} computes the sum of all odd integers in the interval $[0, x]$. The result is then stored in variable y. To this end, y is initialized with zero. After that, the loop C_{loop} is executed: As long as the value of variable x is greater than zero, program C_{if} checks whether the current value of x is odd, i.e., whether the remainder of dividing x by two yields one. If the answer is yes, the value of variable x is added to the value of variable y; otherwise, we execute a `skip` statement. In both cases, the value of variable x is decremented before evaluating the loop guard $x > 0$ again.

To reason about the correctness of programs, e.g., that program C_{odd} in Example 2.1 sums up all odd integers between zero and x, we have to be precise on what it means to execute a **PL** program. Our next step is thus to develop a formal semantics for **PL**.

2.1.2 Semantics

The semantics of a programming language is a mathematical model that specifies how programs behave. To precisely capture the effect of running a **PL** program on a given input, we develop a *small-step operational semantics* [Plo81; NN92; Win93] that describes how programs are executed step-by-step on an abstract machine. The mathematical model underlying our semantics is a transition system. The following presentation of transition systems and related concepts is based on [BK08, Chapter 2].

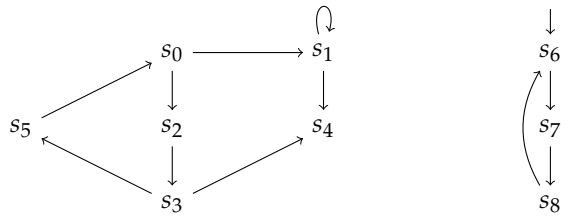


Figure 2.1: Illustration of a transition system as a directed graph.

Definition 2.2 (Transition System [BK08]) A *transition system* TS is a tuple

$$TS \triangleq \langle S, \rightarrow, S_0 \rangle,$$

where S is a set of *states*, $S_0 \subseteq S$ is a set of *initial states*, and

$$\rightarrow \subseteq S \times S$$

is an *execution relation*. We usually write $s \rightarrow s'$ instead of $(s, s') \in \rightarrow$ to denote a *step* of the execution relation.

Example 2.3 Figure 2.1 depicts a transition system $TS = \langle S, \rightarrow, S_0 \rangle$ as a directed graph. Every node in this graph represents a state, i.e., we have $S = \{ s_i \mid 0 \leq i \leq 8 \}$. Moreover, a directed edge between two states, say s and s', corresponds to a step $s \rightarrow s'$ in the execution relation. Moreover, initial states are highlighted by having an incoming edge without a source node, i.e., the set of initial states is $S_0 = \{ s_6 \}$.

Intuitively, the states of a transition system, which models the execution of **PL** programs, consist of the program to execute and a *stack*, i.e., a snapshot of the current assignment of a value, say an integer in $\mathbb{Z}$, to every program variable in **Vars**. We prefer the term "stack" over other terms, such as "program state" or "store" [NN92; Win93], because "stack" commonly refers to the area of memory on most computers which keeps track of local variables (cf. [SGG10]). This description suits our formalization well; in particular, once we add support for procedures and dynamic memory allocation.

Assumption 2.4 A naïve definition of stacks takes all functions from the set **Vars** of variables into the set of integers $\mathbb{Z}$ into account (cf. [NN92, p. 12]), i.e., one might define the set of stacks as

$$\textbf{Stacks} \triangleq \{\mathfrak{s} \colon \textbf{Vars} \to \mathbb{Z}\}.$$

Unfortunately, this definition leads to technical subtleties: In Chapter 5, we need a *countable* set of stacks, whereas the set of all total functions from **Vars** into $\mathbb{Z}$ is uncountable. Furthermore, to represent a stack on an actual machine, it is reasonable to consider only evaluations of the finitely many variables that are actually accessed by a program.

To address these issues, we assume that *all variables are initialized with zero*. Since any program can only change finitely many variables, it then suffices to consider the countable set of stacks in which only finitely many variables are mapped to integers different from zero.

Alternatively, one might think of stacks as *partial* functions mapping from the set of variables accessed by a program into the set of integers. While this approach also leads to a countable set of stacks, it is inconvenient: Whenever the program under consideration changes, we might have to change the domain of stacks.

With these considerations in mind, we formally define stacks as follows:

Definition 2.5 (Stack) The countable *set of stacks* is defined as

$$\begin{aligned}\textbf{Stacks} \triangleq \{\mathfrak{s} \colon \textbf{Vars} \to \mathbb{Z} \mid \exists V \subseteq \textbf{Vars} \colon |V| < \infty \\ \text{and } \forall x \in \textbf{Vars} \setminus V \colon \mathfrak{s}(x) = 0\}.\end{aligned}$$

Let us also give a precise definition of expressions. An *arithmetic expression* E is a (computable) function $E \colon \textbf{Stacks} \to \mathbb{Z}$ that assigns an integer to every stack. Analogously, a *Boolean expression* B is a (computable) function $B \colon \textbf{Stacks} \to \mathbb{T}$ that assigns a truth value, i.e., true or false, to every stack.

A step of the execution relation corresponding to a **PL** program and a stack is then understood as a transformation of both the stack and the remaining program. To formalize these transformations, we introduce notation for updating the values assigned to variables.

Definition 2.6 (Variable Substitution [Win93, p. 19]) The *substitution* of a variable $x \in \mathbf{Vars}$ by an integer $v \in \mathbb{Z}$ in a stack $\mathfrak{s} \in \mathbf{Stacks}$ is defined as

$$\mathfrak{s}[x/v] \triangleq \lambda y. \begin{cases} v, & \text{if } y = x \\ \mathfrak{s}(y), & \text{if } y \neq x. \end{cases}$$

Here, we use λ-expressions to denote functions. That is, function $\lambda y. f$ applied to an argument u evaluates to f in which every occurrence of y is replaced by u.

We describe execution relations by inference rules à la Plotkin [Plo81]:

$$\frac{\textit{premise}}{\textit{conclusion}} \text{ name}$$

Here, both *premise* and *conclusion* are propositions that are either true or false. Every rule should be read as an implication: If the *premise* is true, then we can infer that the *conclusion* is also true. In this case, we also say that we applied rule "name". If a premise is always true, it is omitted and the rule is called an *axiom*. Whenever we state that a set is "determined by inference rules", we refer to the smallest set satisfying all conclusions that can be inferred by exhaustive application of inference rules.

We are now in a position to define a transition system that assigns formal semantics to **PL** programs. We first define a single transition system for all **PL** programs and all possible inputs. After that, we consider running a single **PL** program on a given set of inputs (see Definition 2.12 on page 26).

Definition 2.7 (Operational Semantics of PL Programs [NN92, p. 33]) Let term be a special symbol indicating successful termination and $\langle \mathsf{sink} \rangle$ be a dedicated sink state. The *operational semantics of* **PL** *programs* is given by the transition system $\mathrm{oPL} \triangleq \langle\, \mathbf{States}, \rightsquigarrow, \mathbf{States} \,\rangle$, where both the set of states and the set of initial states are defined as

$$\mathbf{States} \triangleq ((\mathbf{PL} \cup \{\, \mathsf{term} \,\}) \times \mathbf{Stacks}) \cup \{\, \langle \mathsf{sink} \rangle \,\}.$$

Moreover, the execution relation

$$\rightsquigarrow \subseteq \mathbf{States} \times \mathbf{States}$$

is determined by the rules in Figure 2.2 on page 24.

Let us briefly go over the rules of our operational semantics in Figure 2.2:

$$\frac{}{\langle \texttt{skip}, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s} \rangle}\ \text{skip} \qquad \frac{E(\mathfrak{s}) = v}{\langle x := E, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[x/v] \rangle}\ \text{assign}$$

$$\frac{\langle C_1, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}' \rangle}{\langle C_1 ; C_2, \mathfrak{s} \rangle \rightsquigarrow \langle C_2, \mathfrak{s}' \rangle}\ \text{seq1} \qquad \frac{\langle C_1, \mathfrak{s} \rangle \rightsquigarrow \langle C_1', \mathfrak{s}' \rangle}{\langle C_1 ; C_2, \mathfrak{s} \rangle \rightsquigarrow \langle C_1' ; C_2, \mathfrak{s}' \rangle}\ \text{seq2}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}, \mathfrak{s} \rangle \rightsquigarrow \langle C_1, \mathfrak{s} \rangle}\ \text{if-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}, \mathfrak{s} \rangle \rightsquigarrow \langle C_2, \mathfrak{s} \rangle}\ \text{if-false}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{while}\,(B)\,\{C\}, \mathfrak{s} \rangle \rightsquigarrow \langle C ; \texttt{while}\,(B)\,\{C\}, \mathfrak{s} \rangle}\ \text{while-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{while}\,(B)\,\{C\}, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s} \rangle}\ \text{while-false}$$

$$\frac{}{\langle \mathsf{term}, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{sink} \rangle}\ \text{term} \qquad \frac{}{\langle \mathsf{sink} \rangle \rightsquigarrow \langle \mathsf{sink} \rangle}\ \text{sink}$$

Figure 2.2: The rules determining the execution relation of the operational semantics of **PL** programs.

The `skip` statement terminates without changing the stack. An assignment $x := E$ also terminates in a single step, but additionally updates the value of x in stack $\mathfrak{s}$ to the value v which is obtained by evaluating expression E in $\mathfrak{s}$.

There are two cases for the sequential composition $C_1 ; C_2$: If program C_1 terminates in a single step, we continue by executing program C_2. Otherwise, we execute one step of program C_1 and continue with the sequential composition of the resulting program C_1' and C_2.

For the conditional choice `if` (B) $\{ C_1 \}$ `else` $\{ C_2 \}$, execution depends on the evaluation of guard B in the current stack. If the guard evaluates to `true`, we continue by executing the left branch C_1. Otherwise, we continue by executing the right branch C_2. Since we assume that evaluating expressions causes no side effects, the stack remains unchanged in both cases.

Similarly to conditionals, execution of loops `while` (B) $\{ C \}$ depends on the evaluation of loop guard B. If the loop guard is evaluated to `true`, we first execute the loop body C and then continue by executing the loop again. Otherwise, the loop terminates without effect.

Finally, Figure 2.2 contains two additional rules which ensure that, upon termination, we move into a dedicated sink state and forever stay in it. The rules term and sink are not necessary to define the semantics of **PL** programs. However, it is convenient to have a dedicated state indicating termination. Moreover, we occasionally rely on the presence of a sink state with a self-loop. We thus added both rules for consistency.

The transition system oPL captures the semantics of *all* **PL** programs for *all* initial stacks. To run a particular program, say C, on a particular stack, say $\mathfrak{s}$, it then suffices to compute the steps of oPL starting in state $\langle C, \mathfrak{s} \rangle$. Sequences of such steps are called execution fragments.

Definition 2.8 (Execution [BK08, Section 2.1.1]) Let $TS = \langle S, \to, S_0 \rangle$ be a transition system. A *finite execution fragment* is a sequence of states in S

$$\chi = s_0 s_1 \dots s_n,$$

where $n \in \mathbb{N}$ is a natural number indicating the length of execution fragment χ, and, for all $0 \leq k < n$, we have $s_k \to s_{k+1}$.

An *infinite execution fragment* is an infinite sequence of states in S

$$\chi = s_0 s_1 \dots,$$

where, for all natural numbers $k \in \mathbb{N}$, we have $s_k \to s_{k+1}$.

An *initial* execution fragment is an execution fragment that starts in an initial state, i.e., $s_0 \in S_0$. Moreover, a *maximal* execution fragment is an

execution fragment which is either infinite or a finite execution fragment that ends in a state s_n such that there exists no state s' with $s_n \to s'$. Finally, an *execution* is an execution fragment which is both initial and maximal.

Example 2.9 The sequence $s_0 s_2 s_3 s_5 s_0 s_1 s_4$ is a maximal execution fragment of the transition system TS depicted in Figure 2.1, but it is not an initial one. In fact, the infinite sequence $s_6 s_7 s_8 s_6 \ldots$ is the only execution of TS.

In our operational semantics, running a program C on a stack $\mathfrak{s}$ amounts to computing all executions of the transition system oPL in which we change the set of initial states to $\{ \langle C, \mathfrak{s} \rangle \}$. Moreover, we remark that all executions of oPL are infinite: Either a program never terminates or it terminates and the execution loops forever in a dedicated sink state. Usually, only a part of oPL is relevant to compute these executions. This part is called the reachable fragment.

Definition 2.10 (Reachable Fragment [BK08, p. 26]) Let $TS = \langle S, \to, S_0 \rangle$ be a transition system. A state $s' \in S$ is *reachable* in TS from a state $s \in S$, written $s \to^* s'$, if and only if there exists a finite execution fragment

$$s = s_0 s_1 \ldots s_n = s'.$$

Let R be the set of all states reachable in TS from some initial state $s \in S_0$. Then the *reachable fragment of* TS is the transition system

$$\mathbf{Reach}\,(TS) \triangleq \langle R, \to \cap (R \times R), S_0 \rangle.$$

Example 2.11 The reachable fragment of the transition system in Figure 2.1 contains only the states s_6, s_7, and s_8.

Since we usually reason about the transition system oPL, which determines the semantics of **PL** programs, we introduce some convenient notation for dealing with reachable fragments and executions of oPL.

Definition 2.12 (Notation for Operational Semantics) Let $\mathbf{I} \subseteq \mathbf{States}$ be a set of states of the transition system $\text{oPL} = \langle \mathbf{States}, \rightsquigarrow, \mathbf{States} \rangle$. Then the *reachable fragment of the operational semantics with respect to* $\mathbf{I}$ is defined as

$$\text{oPL}\,(\mathbf{I}) \triangleq \mathbf{Reach}\,(\langle \mathbf{States}, \rightsquigarrow, \mathbf{I} \rangle).$$

Moreover, the *set of executions* of a **PL** program C and a stack $\mathfrak{s}$ is defined as

$$\mathbf{Exec}[C](\mathfrak{s}) \triangleq \{ \rho \mid \rho \text{ is an execution of oPL}\,(\{ \langle C, \mathfrak{s} \rangle \}) \}.$$

It is noteworthy that the set of executions $\mathbf{Exec}[C](\mathfrak{s})$ is always a singleton because **PL** programs are deterministic [NN92, p. 39].

Example 2.13 Recall from Example 2.1, page 20, the program C_{odd} which computes the sum of all odd integers in the interval $[0, x]$. and stores the result in y. Let us test whether C_{odd} indeed computes the desired result by applying our operational semantics for an initial stack $\mathfrak{s}$ with $\mathfrak{s}(x) = 3$. That is, we are interested in the set of executions $\mathbf{Exec}[C_{\mathsf{odd}}](\mathfrak{s})$. The only such execution of the transition system oPL $(\{\langle C_{\mathsf{odd}}, \mathfrak{s}\rangle\})$ is given by the following applications of inference rules (whenever possible we tacitly apply the rules seq1 and seq2 before any other rule):

$$
\begin{aligned}
& \langle C_{\mathsf{odd}}, \mathfrak{s}\rangle \\
\rightsquigarrow\ & \langle C_{\mathsf{loop}}, \mathfrak{s}\,[y/0]\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle C_{\mathsf{if}}\,;\, x := x-1\,; C_{\mathsf{loop}}, \mathfrak{s}\,[y/0]\,\rangle && \text{(by while-true)} \\
\rightsquigarrow\ & \langle y := y + x\,;\, x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/0]\,\rangle && \text{(by if-true)} \\
\rightsquigarrow\ & \langle x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/2]\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle C_{\mathsf{if}}\,;\, x := x-1\,; C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/2]\,\rangle && \text{(by while-true)} \\
\rightsquigarrow\ & \langle \mathtt{skip}\,;\, x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/2]\,\rangle && \text{(by if-false)} \\
\rightsquigarrow\ & \langle x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/2]\,\rangle && \text{(by skip)} \\
\rightsquigarrow\ & \langle C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/1]\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle C_{\mathsf{if}}\,;\, x := x-1\,; C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/1]\,\rangle && \text{(by while-true)} \\
\rightsquigarrow\ & \langle y := y + x\,;\, x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/3]\,[x/1]\,\rangle && \text{(by if-true)} \\
\rightsquigarrow\ & \langle x := x-1\,;\, C_{\mathsf{loop}}, \mathfrak{s}\,[y/4]\,[x/1]\,\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle C_{\mathsf{loop}}, \mathfrak{s}\,[y/4]\,[x/0]\rangle && \text{(by assign)} \\
\rightsquigarrow\ & \langle \mathsf{term}, \mathfrak{s}\,[y/4]\,[x/0]\,\rangle && \text{(by while-false)} \\
\rightsquigarrow\ & \langle \mathsf{sink}\rangle \rightsquigarrow \langle \mathsf{sink}\rangle \rightsquigarrow \ldots && \text{(by term)}
\end{aligned}
$$

Upon termination, i.e., when reaching a state $\langle \mathsf{term}, \ldots\rangle$, we have

$$\mathfrak{s}\,[y/4]\,[x/0]\,(y) \;=\; 4 \;=\; 1+3.$$

Hence, if variable x is initially 3, program C_{odd} indeed computes the sum of all odd integers between 0 and x and stores the result in y.

Since the operational semantics of **PL** programs is close to running programs step-by-step on an actual machine, its rules (cf. Figure 2.2) match our intuitive understanding of program statements. It is thus well-suited for *understanding* the semantics of **PL** programs per se.

Our operational semantics is, however, inadequate for *reasoning* about programs, i.e., proving that a program has a property of interest: One could, in principle, compute all executions of a **PL** program and some initial stacks as we did in Example 2.13. For each of these executions, one then checks whether a property holds. This approach amounts to testing. Unfortunately, there are usually infinitely many possible initial stacks, i.e., we have to compute infinitely many executions to prove a property. Furthermore, testing fails if some executions are infinite themselves, i.e., if the program does not terminate. Finally, we are forced to consider all pedantic details of program executions—even if they are irrelevant for our property.

In the next sections, we thus consider other techniques for reasoning about **PL** programs. While these techniques attempt to avoid the aforementioned drawbacks, they do not reflect the intuitive behavior of programs as well as the operational semantics. Therefore, the operational semantics serves as the reference model for proving these techniques sound.

2.2 Program Analysis with Abstraction

The operational semantics of **PL** is mainly inadequate for reasoning about programs because the transition system oPL describes all the gory details of program executions. This suggests that abstracting from these details improves our ability to analyze programs. In fact, many successful techniques for automated reasoning about programs rely on some kind of abstraction [CC14]. Moreover, abstract interpretation [CC77; CC92] provides compelling theoretical foundations to develop useful program analyses.

In this section, we briefly describe how abstraction supports reasoning about transition systems. Since the operational semantics oPL is a transition system, the same techniques apply to **PL** programs. Our treatment of abstraction is based on [BK08, Section 7.4], where abstraction is considered in terms of simulation relations [Mil71]. For a thorough treatment of abstract interpretation and applications to program analysis, we refer to [CC92; NNH99].

The main idea of abstraction is to represent a concrete transition system by an abstract transition system such that, for some property P of interest,

1. the abstract transition system is easier to analyze for property P, and
2. if property P holds for the abstract transition system, it also holds for the concrete transition system.

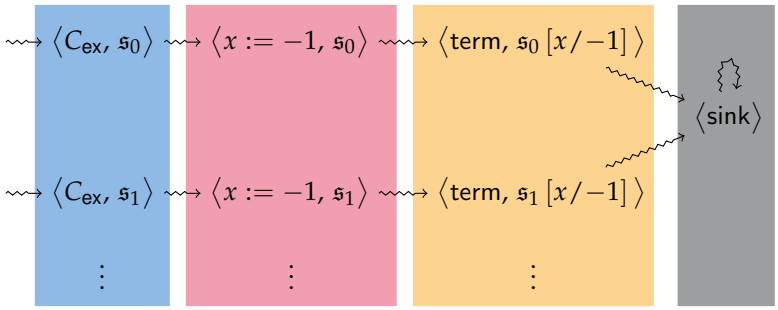


Figure 2.3: Illustration of infinite transition system.

Intuitively, this means that every state of the abstract transition system represents many states of the concrete transition system which share some common property. Moreover, the execution relation of the abstract transition system must cover all possible steps of the concrete transition system.

Example 2.14 As a running example, consider the program C below.

$$C_{\mathsf{ex}}: \qquad \texttt{while}\,(\,x \leq 0\,)\,\{\,\texttt{skip}\,\}\,;\,x := -1$$

Assume we want to know at every point of execution of C_{ex} whether variable x is positive or not. Moreover, for simplicity, let x be initially positive. Reasoning about program C_{ex} by explicitly applying the operational semantics, i.e., computing the concrete transition system

$$\mathrm{oPL}\left(\left\{\,\langle C_{\mathsf{ex}},\,\mathfrak{s}\rangle \;\middle|\; \mathfrak{s}(x) > 0\,\right\}\right),$$

is infeasible, since it has an infinite number of states. A fragment of this transition system is illustrated in Figure 2.3, page 29. Although the transition system is infinite, the program behaves very similarly for all considered initial states. In fact, states executing the same program, i.e., states in the same colored box, differ only in the values assigned to variables. The concrete value of variables does, however, not matter for our property which is only concerned with the sign of variable x. This suggests representing all states in the same colored box by a single abstract state that stores whether x is positive or not instead of a concrete stack. We formalize such an abstraction and argue why the resulting abstract transition system is sufficient to prove our property in the remainder of this section.

To construct abstract transition systems, we first need a notion of "properties of interest". Thus, we assign labels to every state.

Definition 2.15 (State Labeling [BK08, p. 20]) Let $TS = \langle S, \to, S_0 \rangle$ be a transition system. Moreover, let $\mathbf{AP}$ be a (possibly infinite) set of *atomic propositions*. A *state labeling* of TS is a function

$$\mathsf{Lab} \colon S \to 2^{\mathbf{AP}},$$

where $2^{\mathbf{AP}} \triangleq \{ S \mid S \subseteq \mathbf{AP} \}$ denotes the powerset of $\mathbf{AP}$.

Example 2.16 A property of interest in Example 2.14 is whether variable x is positive. Moreover, we want to keep track of the control flow of our program. Hence, we choose the set of atomic propositions

$$\mathbf{AP} \triangleq \{ x > 0, x \leq 0 \} \cup \mathbf{PL} \cup \{ \mathsf{term} \}.$$

We then define the state labeling of our concrete transition system as

$$\mathsf{Lab}(s) \triangleq \begin{cases} \{ C, x > 0 \}, & \text{if } s = \langle C, \mathfrak{s} \rangle \text{ and } \mathfrak{s}(x) > 0 \\ \{ C, x \leq 0 \}, & \text{if } s = \langle C, \mathfrak{s} \rangle \text{ and } \mathfrak{s}(x) \leq 0 \\ \emptyset, & \text{otherwise.} \end{cases}$$

Consequently, in our illustration of the concrete transition system, see Figure 2.3, states with the same label lie in the same colored box.

An abstraction determines the concrete states that are represented by a single abstract state. Since our goal is to reason about properties expressed in terms of atomic propositions, abstractions must preserve the state labeling.

Definition 2.17 (Abstraction Function [BK08, p. 499]) Let $TS = \langle S, \to, S_0 \rangle$ be a (concrete) transition system equipped with a state labeling $\mathsf{Lab} \colon S \to 2^{\mathbf{AP}}$. Moreover, let $S^\#$ be a set of *abstract states*. Then the function $f \colon S \to S^\#$ is an *abstraction function* if

$$\forall s, s' \in S \colon \quad f(s) = f(s') \text{ implies } \mathsf{Lab}(s) = \mathsf{Lab}(s').$$

Example 2.18 To reason about the program C_{ex} in Example 2.14, we collapse all states with the same label into a single abstract state. Hence, the set of abstract states is defined as

$$S^{\#} \triangleq \{ \langle C_{\mathsf{ex}},\, x > 0 \rangle, \langle x := -1,\, x > 0 \rangle, \langle \mathsf{term},\, x \leq 0 \rangle, \langle \mathsf{sink} \rangle \},$$

where, intuitively, every abstract state collects all states in the same colored box in Figure 2.3, page 29. Formally, the corresponding abstraction function with respect to the state labeling considered in Example 2.16 is given by

$$f(s) \triangleq \begin{cases} \langle C_{\mathsf{ex}},\, x > 0 \rangle, & \text{if } s = \langle C_{\mathsf{ex}},\, \mathfrak{s} \rangle \text{ and } \mathfrak{s}(x) > 0 \\ \langle x := -1,\, x > 0 \rangle, & \text{if } s = \langle x := -1,\, \mathfrak{s} \rangle \text{ and } \mathfrak{s}(x) > 0 \\ \langle \mathsf{term},\, x \leq 0 \rangle, & \text{if } s = \langle \mathsf{term},\, \mathfrak{s} \rangle \text{ and } \mathfrak{s}(x) \leq 0 \\ \langle \mathsf{sink} \rangle, & \text{if } s = \langle \mathsf{sink} \rangle. \end{cases}$$

Once we have chosen an abstraction function, it remains to construct an abstract transition system which is hopefully easier to analyze. The set of states of the abstract transition system is already determined by the abstraction function. Moreover, since all concrete states, which are mapped to the same abstract state, are labeled with the same atomic propositions, the labeling of abstract states is straightforward. The execution relation should cover all concrete execution steps. Hence, for every concrete step, say $s \to s'$, there must be an abstract step between the abstract states corresponding to s and s', respectively. When dealing with infinite transition systems, covering exactly all concrete executions may be infeasible—in particular if the only precise way to compute the abstract steps is to revert to concrete steps. In contrast to [BK08], we thus allow the abstract execution relation to overapproximate the concrete execution relation.

Definition 2.19 (Abstract Transition System [BK08, p. 500]) For a transition system $TS = \langle S, \to, S_0 \rangle$ equipped with a state labeling $\mathsf{Lab} \colon S \to 2^{\mathbf{AP}}$, let $f \colon S \to S^{\#}$ be an abstraction function for some set of abstract states $S^{\#}$. Then $TS_f = \langle S^{\#}, \to^{\#}, S_0^{\#} \rangle$ is an *abstract transition system* equipped with a state labeling $\mathsf{Lab}^{\#} \colon S^{\#} \to 2^{\mathbf{AP}}$ whenever

- $\to^{\#} \supseteq \{ (f(s), f(s')) \mid s \to s' \}$,
- $S_0^{\#} = \{ f(s) \mid s \in S_0 \}$, and
- for all states $s \in S$, $\mathsf{Lab}^{\#}(f(s)) = \mathsf{Lab}(s)$.

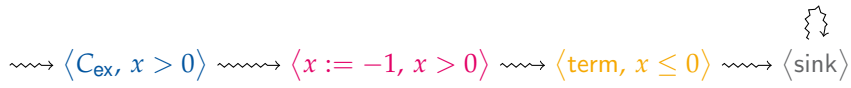


Figure 2.4: Abstract transition system.

Example 2.20 The abstract transition system with the smallest execution relation corresponding to the concrete transition system in Figure 2.3, i.e.,

$$\mathrm{oPL}\left(\left\{\, \langle C_{\mathsf{ex}}, \mathfrak{s}\rangle \;\middle|\; \mathfrak{s}(x) > 0 \,\right\}\right),$$

and the abstraction function described in Example 2.18 is depicted in Figure 2.4. In particular, the concrete states in every colored box in Figure 2.3 are collapsed into a single abstract state of the same color.

Remember that our goal in Example 2.14 was to discover whether variable x is positive for every execution step of the program

$$C_{\mathsf{ex}}\colon \qquad \texttt{while}\,(\,x \leq 0\,)\,\{\,\texttt{skip}\,\};\, x := -1$$

provided that the program starts on a stack, where x is positive. The abstract transition system in Figure 2.4 obviously stores the relevant information in each of its states (except for $\langle\mathsf{sink}\rangle$). For example, one could use this information to prove that variable x is always negative upon termination. However, what does this tell us about the concrete transition system, i.e., the reachable fragment of the operational semantics?

Intuitively, every execution, say $s_0 s_1 s_2 \ldots$, of the concrete transition system has a corresponding execution $f(s_0)f(s_1)f(s_2)\ldots$ in the abstract transition system. In other words, the abstract transition system *simulates* the concrete transition system [Mil71; BK08]. This means that proving a property for all executions of the abstract transition system implies that the same property holds for all executions of the concrete transition system.

Hence, as long as we are careful about the considered properties, reasoning by abstraction is sound. A precise characterization is found in [BK08, Chapter 7.5]: Reasoning about a property P in an abstract transition system instead of a concrete transition system is sound whenever property P can be specified in the universal fragment of the temporal logic CTL^*.[1] This includes all properties that can be specified in linear temporal logic [Pnu77], such as "x is negative, whenever we reach a state indicating termination".

[1] To be precise, soundness requires that transition systems have no finite maximal execution fragments. This is ensured in our operational semantics because we added a sink state.

Abstraction is a compelling approach for reasoning about programs. While there is a large amount of literature on abstraction techniques and related theory (cf. [Gru05; CC14] for an incomplete overview), the underlying principles sketched in this section are fairly simple.

In particular, abstraction offers a clear path towards automation: It suffices to choose an abstraction function such that (1) the abstract execution relation is effectively computable and (2) the reachable fragment of the abstract transition system has only finitely many states for a given set of initial states. With these properties at hand, we obtain a generic program analysis algorithm:

1. Choose a set of abstract initial states that covers all inputs of interest.
2. Compute the reachable fragment of the abstract transition system with respect to these initial states.

For automated program verification, the computed (finite) abstract transition system is then checked for desired properties. To this end, established techniques, such as model checking [EC80; BK08; Cla+18], are available. If the property holds for the abstract transition system, it also holds for the concrete transition system, i.e., the program we want to verify. Otherwise, we do not know whether the property is violated or our abstraction is not precise enough. In this case, we might apply counterexample guided abstraction refinement [Cla+00], i.e., exploit the witnesses for property violations, which are provided by the model checker, to derive a better abstraction. In Chapter 13, we use abstraction and model checking for automated reasoning about heap manipulating programs.

There are, however, some caveats. The main issue is that designing good abstractions is hard and requires a lot of creativity. In some sense, coming up with suitable abstractions for program analysis is even harder than verifying that a property holds for a given program [CGR18]. Furthermore, finding an abstraction function is not enough. For example, if the abstract execution relation cannot be determined without reverting to concrete execution steps, computing the abstract transition system becomes infeasible.

In the next section, we study an alternative approach, which attempts to derive formal proofs of properties directly on the program structure without constructing a transition system at all.

2.3 Program Verification

We now turn to the task of program verification, i.e., proving that a program satisfies a property. Turing [Tur49] already proposed that a

> *"programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."*

This approach is different from the techniques presented in previous sections: So far, reasoning about programs involved their execution—either on a concrete machine or in an abstract domain—and then analyzing the observed executions. In contrast, Turing suggests assembling a proof of correctness compositionally from already proven properties.

Almost twenty years later, Floyd [Flo67] developed a formal proof technique to reason about the correctness of flow charts. Hoare [Hoa69] later extended his approach to programs.[2] The central goal of the resulting technique—called Floyd-Hoare logic—is to prove that a program meets a specification. By specification, we mean a *Hoare triple* of the form

$$\langle\, \mathit{precondition} \,\rangle\ C\ \langle\, \mathit{postcondition} \,\rangle,$$

where C is a program, *precondition* is the set of stacks considered before execution of C, and *postcondition* is the set of stacks acceptable after execution of C.

For example, the informal specification "program C_{ex} from Example 2.14 terminates with variable x being less than or equal to zero whenever it is initially greater than zero" is captured by the Hoare triple

$$\langle\ \{\, \mathfrak{s} \in \mathbf{Stacks} \mid \mathfrak{s}(x) > 0 \,\}\ \rangle\ C_{\mathsf{ex}}\ \langle\ \{\, \mathfrak{s} \in \mathbf{Stacks} \mid \mathfrak{s}(x) \leq 0 \,\}\ \rangle.$$

Program verification then amounts to providing a formal proof that a Hoare triple is valid, i.e., the specification is correct. Intuitively, a Hoare triple is valid if every execution of C, which starts on a stack in the precondition, leads to a stack in the postcondition. The exact notion of validity is, however, more subtle and depends on the specific flavor of Hoare logic. The meaning of Hoare logic nowadays depends on three aspects:

1. An *assertion language* which determines how to represent sets of stacks as pre- and postconditions.

2. A precise notion of *validity* of Hoare triples.

3. A *proof system* to determine whether a Hoare triple is valid.

We briefly explore each of these aspects for reasoning about **PL** programs.

[2] There is, however, no historical evidence that either Floyd or Hoare was influenced by Turing's publication. In fact, Turing's paper was hardly readable due to various typesetting errors until it was reconstructed from the original notes [MJ84].

2.3.1 Assertion Language

Reasoning about Hoare triples requires a finite representation of (infinite) sets of stacks. Otherwise, just writing down a precondition, for instance the set of all stacks in which the value of variable x is greater than the value of variable y, becomes infeasible. We thus represent every set of stacks by a *predicate*, i.e., a function mapping every stack to a truth value in $\mathbb{T} = \{\,\mathsf{true}, \mathsf{false}\,\}$. A predicate then describes the set of all for which it evaluates to true.

Definition 2.21 (Predicate [NN92, p. 177]) The *set of predicates* is defined as

$$\mathbf{Pred} \triangleq \{\, P \colon \mathbf{Stacks} \to \mathbb{T} \,\}.$$

Moreover, the *set of stacks captured by predicate* P is defined as

$$\mathbf{Stacks}\,(P) \triangleq \{\, \mathfrak{s} \in \mathbf{Stacks} \mid P(\mathfrak{s}) = \mathsf{true} \,\}.$$

We write $\mathfrak{s} \models P$ (read: $\mathfrak{s}$ satisfies P) as a shortcut for $P\,(\mathfrak{s}) = \mathsf{true}$. Analogously, we write $\mathfrak{s} \not\models P$ (read: $\mathfrak{s}$ violates P) as a shortcut for $P\,(\mathfrak{s}) = \mathsf{false}$.

For example, the set of all stacks in which the value of variable x is greater than the value of variable y is captured by the predicate $x > y$.

An *assertion language* determines the exact set of predicates that are allowed to occur in Hoare triples. Assertion languages broadly come in two flavors (cf. [NN92, Chapter 6.2]): The *intensional approach* provides an explicit syntax for allowed predicates; this approach is taken in [Win93] and necessary for automated reasoning. The *extensional approach* allows us to use any predicate in **Pred**; this approach is taken in [NN92; NN07]. Since **PL** is liberal about the syntax of both arithmetic and Boolean expressions (cf. Section 2.1.1), we are also liberal about our assertion language. Hence, we take the second approach.

Let us collect a few examples of common predicates and their captured sets of stacks. The predicate $\mathsf{false} \triangleq \lambda\mathfrak{s}.\,\mathsf{false}$ captures the empty set, i.e., $\mathbf{Stacks}\,(\mathsf{false}) = \emptyset$. Analogously, the set **Stacks** of all stacks is captured by the predicate $\mathsf{true} \triangleq \lambda\mathfrak{s}.\,\mathsf{true}$. Moreover, we may use any Boolean expression B as a predicate. For example, for the predicate $x > y$, we have

$$\mathbf{Stacks}\,(x > y) \;=\; \{\mathfrak{s} \in \mathbf{Stacks} \mid \mathfrak{s}\,(x) > \mathfrak{s}\,(y)\}.$$

Furthermore, the predicates $P \vee Q$ and $P \wedge Q$ capture the union and intersection of the sets $\mathbf{Stacks}\,(P)$ and $\mathbf{Stacks}\,(Q)$, respectively. The predicate $\neg P$ captures the set $\mathbf{Stacks} \setminus \mathbf{Stacks}\,(P)$. Finally, the predicate $P \Rightarrow Q$ evaluates to true if and only if the set $\mathbf{Stacks}\,(P)$ is a subset of the set $\mathbf{Stacks}\,(Q)$.

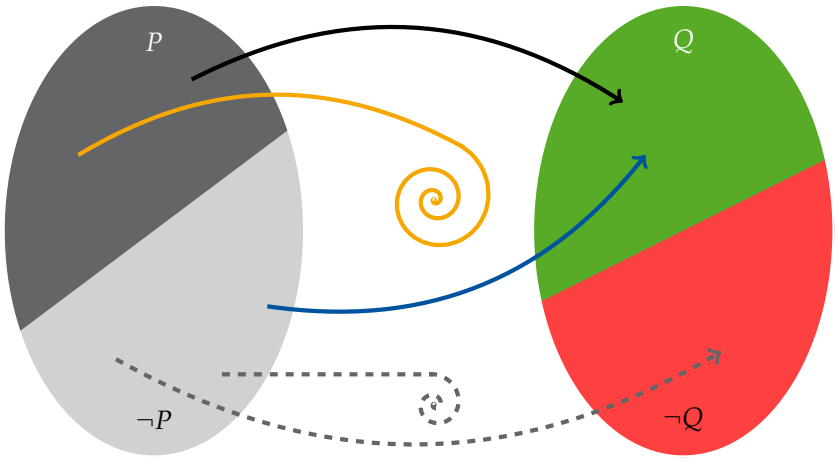


Figure 2.5: Illustration of possible program executions.

2.3.2 Validity of Hoare Triples

With an assertion language at hand, we now give a precise definition of Hoare triples: A *Hoare triple*, or triple for short, is an expression of the form[3]

$$\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle,$$

where $P \in \mathbf{Pred}$ is a predicate called the *precondition*, C is a **PL** program, and $Q \in \mathbf{Pred}$ is a predicate called the *postcondition*.

Whether a Hoare triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is valid, i.e., whether program C meets its specification given by precondition P and postcondition Q, depends on the class of properties we are interested in. Intuitively, the chosen class of properties determines which program executions are considered acceptable. Figure 2.5, page 36, illustrates possible executions of a **PL** program, say C, on various initial stacks. On the left-hand side, the set **Stacks** of all stacks is partitioned by a precondition P, i.e., we distinguish the sets $\mathbf{Stacks}\,(P)$ and $\mathbf{Stacks}\,(\neg P)$. Analogously, on the right-hand side, the set **Stacks** is partitioned by some postcondition Q. Every line illustrates an execution of program C which starts either in a stack $\mathfrak{s} \models P$ or a stack $\mathfrak{s} \not\models P$. For every execution, there are three possibilities: It terminates in a stack $\mathfrak{s}' \models Q$, i.e., the line ends in the green area indicating that the postcondition is satisfied, it terminates in a stack $\mathfrak{s}' \not\models Q$, i.e.,

[3] see, for example, [Hoa69], [NN92, p. 176], or [Win93, p. 78].

the line ends in the red area indicating that the postcondition is violated, or the execution does not terminate at all, i.e., the line ends up in a spiral.

The class of properties under consideration then determines which executions may and may not exist. For example, the topmost (black) line indicates an execution which initially satisfies the precondition and terminates in a stack that satisfies the postcondition. These executions are desirable and are included in any conceivable class of properties. Conversely, an execution that initially satisfies the precondition and terminates in a stack that violates the postcondition is never desirable. It will thus be forbidden for any conceivable class of properties (we did not even draw it in Figure 2.5). Furthermore, we do not care about the dashed lines, i.e., executions that initially violate the precondition and either do not terminate or violate the postcondition upon termination; these executions may exist, but are not considered. The remaining possible executions are design choices. For example, one might allow or forbid nonterminating executions that initially satisfy the precondition (illustrated by the orange line).

If we allow executions corresponding to the orange line in Figure 2.5, then we consider *partial correctness*: The triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is valid with respect to partial correctness if every *terminating* execution of C on a stack satisfying precondition P ends up with a stack satisfying postcondition Q. In particular, the triple

$$\langle\, P \,\rangle\ \texttt{while}\,(\,\texttt{true}\,)\,\{\,\texttt{skip}\,\}\ \langle\, Q \,\rangle$$

is valid with respect to partial correctness for all predicates $P, Q \in \mathbf{Pred}$.

If we enforce termination, i.e., we forbid executions corresponding to the orange line in Figure 2.5, then we consider *total correctness*. That is, the triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is valid with respect to total correctness if every execution of C on a stack satisfying precondition P terminates in a stack that satisfies postcondition Q. Consequently, regardless of postcondition Q, there is exactly one precondition P such that the triple

$$\langle\, P \,\rangle\ \texttt{while}\,(\,\texttt{true}\,)\,\{\,\texttt{skip}\,\}\ \langle\, Q \,\rangle$$

is valid with respect to total correctness: $P = \mathsf{false}$.

Before we turn to a system for proving Hoare triples valid, let us give precise definitions of validity in terms of the operational semantics oPL introduced in Section 2.1.2. To this end, recall that $\rightsquigarrow^*$ denotes finitely many steps of the execution relation of oPL (cf. Definition 2.10). Moreover, a program execution *terminates* in our operational semantics if it eventually reaches the sink state $\langle \mathsf{sink} \rangle$ and forever stays in that state. Hence, the set of all infinite sequences of states that indicate termination is defined as $\mathbf{Terminated} \triangleq \mathbf{States}^* \langle \mathsf{sink} \rangle^\omega$.[4]

[4] $\mathbf{States}^*$ denotes the set of all finite sequences over $\mathbf{States}$. Furthermore, the infinite sequence $\langle \mathsf{sink} \rangle \langle \mathsf{sink} \rangle \ldots$ is denoted by $\langle \mathsf{sink} \rangle^\omega$ The set in which every sequence in $\mathbf{States}^*$ is concatenated with $\langle \mathsf{sink} \rangle^\omega$ is then defined as $\mathbf{States}^* \langle \mathsf{sink} \rangle^\omega \triangleq \left\{ \rho\, \langle \mathsf{sink} \rangle^\omega \;\middle|\; \rho \in \mathbf{States}^* \right\}$.

Definition 2.22 (Validity of Hoare Triples [Hoa69], [NN92, pp. 176, 192])
The Hoare triple $\langle P \rangle\ C\ \langle Q \rangle$ is *valid for partial correctness* if and only if

$$\forall \mathfrak{s} \in \mathbf{Stacks}(P)\ \forall \mathfrak{s}' \in \mathbf{Stacks}: \qquad \langle C, \mathfrak{s} \rangle \rightsquigarrow^* \langle \mathsf{term}, \mathfrak{s}' \rangle \text{ implies } \mathfrak{s}' \models Q.$$

The Hoare triple $\langle P \rangle\ C\ \langle Q \rangle$ is *valid for total correctness* if and only if

- $\langle P \rangle\ C\ \langle Q \rangle$ is valid for partial correctness, and
- $\forall \mathfrak{s} \in \mathbf{Stacks}(P): \qquad \mathbf{Exec}[C](\mathfrak{s}) \subseteq \mathbf{Terminated}$, where

$\mathbf{Exec}[C](\mathfrak{s})$ is the set of executions of C on $\mathfrak{s}$ (cf. Definition 2.12, page 26).

All **PL** programs are deterministic, i.e., the set $\mathbf{Exec}[C](\mathfrak{s})$ is a singleton. Exploiting this property leads to a simpler notion of valid Hoare triples: A Hoare triple $\langle P \rangle\ C\ \langle Q \rangle$ is valid for total correctness if and only if for every stack $\mathfrak{s} \in \mathbf{Stacks}(P)$ there exists a stack $\mathfrak{s}' \in \mathbf{Stacks}(Q)$ such that $\langle C, \mathfrak{s} \rangle \rightsquigarrow^* \langle \mathsf{term}, \mathfrak{s}' \rangle$. This definition fails, however, if a program admits multiple executions. Since this is the case in follow-up chapters, e.g., when considering dynamic memory allocation in Chapter 4, we opted for a more robust definition of validity.

2.3.3 Proof Systems

The last aspect of Hoare logic we examine are proof systems to systematically derive valid triples. Without such a proof system, we would have to revert to the operational semantics for proving validity—a clearly undesirable scenario. Floyd [Flo67] and Hoare [Hoa69] originally developed inference rules—similar to the rules determining our operational semantics in Definition 2.7—to prove validity. For example, their rule for sequential composition (for both partial and total correctness) is defined as follows:

$$\frac{\langle P \rangle\ C_1\ \langle R \rangle \qquad \langle R \rangle\ C_2\ \langle Q \rangle}{\langle P \rangle\ C_1 ; C_2\ \langle Q \rangle}\ \text{seq}$$

Read from top to bottom, the rule assumes that we have shown the validity of triples for the composed programs C_1 and C_2, i.e., we know that the triples

$$\langle P \rangle\ C_1\ \langle R \rangle \quad \text{and} \quad \langle R \rangle\ C_2\ \langle Q \rangle$$

are valid. Since the postcondition of the left triple coincides with the precondition of the right triple, we can then conclude the validity of a triple for the composed program $C_1 ; C_2$. The rule for sequential composition is both intuitive and flexible. For example, we are allowed to derive the validity of the two triples

in the premise in an arbitrary order. Furthermore, there are no restrictions regarding the choice of the intermediate predicate R.

Unfortunately, the same flexibility also exacerbates the systematic derivation of valid triples: For proving a Hoare triple valid, we read inference rules as proof obligations, i.e., from bottom to top. Hence, to prove that the triple

$$\langle\, P \,\rangle\ C_1 ; C_2 \ \langle\, Q \,\rangle$$

is valid, we have to *choose* a predicate R such that the triples

$$\langle\, P \,\rangle\ C_1 \ \langle\, R \,\rangle \quad \text{and} \quad \langle\, R \,\rangle\ C_2 \ \langle\, Q \,\rangle$$

are valid. The rule for sequential composition thus requires a creative choice: Find a suitable predicate from infinitely many possible candidates.

A more systematic approach, which avoids the need for creative choices, has been proposed by Dijkstra [Dij75]: We fix a program C and a postcondition Q and determine the *weakest precondition* WP such that the triple $\langle\, WP \,\rangle\ C \ \langle\, Q \,\rangle$ is valid. Here, the term "weakest" means that WP is logically implied by all other preconditions leading to a valid triple. Moreover, by the logical equivalence

$$P \Rightarrow Q \quad \text{iff} \quad \mathbf{Stacks}\,(P) \subseteq \mathbf{Stacks}\,(Q)\,,$$

the weakest precondition captures the *largest* set of stacks on which program C can be executed such that postcondition Q is satisfied afterward. Intuitively, this means that executions corresponding to the blue line in Figure 2.5, page 36, do not exist if P is the weakest precondition.

When computing weakest preconditions, there is in principle no need for choosing predicates. For example, the intermediate predicate R in the rule for sequential composition is uniquely determined by the weakest precondition of program C_2 and postcondition Q.

Apart from being more systematic, weakest preconditions have other advantages: They are used to prove (relative) completeness of proof systems, i.e., every correct statement that can be expressed in the assertion language can also be shown with the proof system [NN92; Win93]. Moreover, since they avoid ambiguities, weakest preconditions are more natural to formalize. Some formalizations of (extensions of) Hoare logic, such as [Kre+17], are thus internally based on weakest preconditions. We take the same approach and study weakest preconditions for both total and partial correctness in the next two sections.

Before we consider weakest preconditions in detail, however, let us discuss why they exist at all. That is, is there a (unique) weakest precondition WP for every program C and every postcondition Q? To answer this question, we first make an observation about the structure of the set **Pred** of predicates.[5]

[5] We assume the reader is familiar with basic concepts from domain theory, e.g., complete lattices and fixed points. A summary of these notions is found in Appendix A.

Lemma 2.23 $(\,\mathbf{Pred}, \Rightarrow\,)$ is a complete lattice.

Proof. By Lemma A.5, $\left(2^{\mathbf{Stacks}}, \subseteq\right)$ is a complete lattice. Furthermore, by considering the function $\lambda P.\ \mathbf{Stacks}\,(P)$, it is straightforward to verify that $(\,\mathbf{Pred}, \Rightarrow\,)$ is isomorphic to $\left(2^{\mathbf{Stacks}}, \subseteq\right)$. □

Now, consider the set of predicates

$$V \triangleq \left\{\, P \;\middle|\; \left\langle\, P \,\right\rangle\, C \,\left\langle\, Q \,\right\rangle \text{ is valid} \,\right\}.$$

By our previous explanation of Dijkstra's approach, the weakest precondition WP of program C and postcondition Q has to satisfy two properties:[6]

$$WP \in V \quad \text{and} \quad \forall P \in V\colon\ P \Rightarrow WP.$$

In other words, WP is the least upper bound of the set V with respect to the ordering $\Rightarrow$. By Lemma 2.23, this least upper bound exists, i.e.,

$$WP \;=\; \sup \left\{\, P \;\middle|\; \left\langle\, P \,\right\rangle\, C \,\left\langle\, Q \,\right\rangle \text{ is valid} \,\right\}$$

is a predicate in **Pred**. In terms of sets of stacks, the supremum corresponds to a union over all sets of stacks captured by preconditions. The above equation serves us a semantic definition of weakest preconditions which is independent of any precise notion of validity for Hoare triples.

Definition 2.24 (Semantic Weakest Preconditions) The (semantic) *weakest precondition* WP of **PL** program C with respect to postcondition $Q \in \mathbf{Pred}$ is defined as the predicate

$$WP \triangleq \sup \left\{\, P \;\middle|\; \left\langle\, P \,\right\rangle\, C \,\left\langle\, Q \,\right\rangle \text{ is valid} \,\right\}.$$

In the next sections, we discuss how semantic weakest preconditions are computed systematically by induction on the program structure.

2.3.4 The Weakest Precondition Calculus

Let us first consider how weakest preconditions are computed for total correctness. Hence, a triple is valid if every execution of a program on a stack satisfying the precondition terminates in a stack that satisfies the postcondition.

To compute weakest preconditions compositionally, i.e., by structural induction on the syntax of **PL** programs, we define a *predicate transformer*

$$\mathsf{wp}\,[C] : \mathbf{Pred} \to \mathbf{Pred}$$

[6] see also [NN92, p. 187] and [Win93, p. 101].

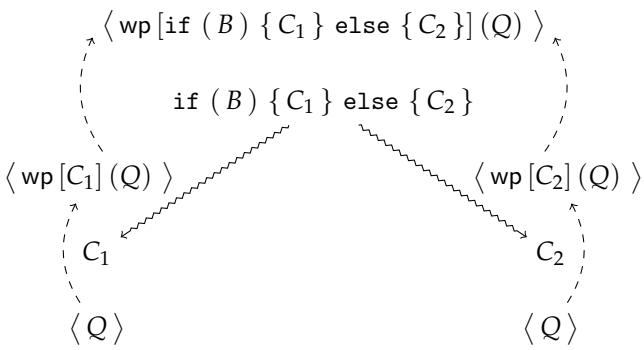


Figure 2.6: Illustration of the weakest precondition calculus wp applied to the program $C = \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ and postcondition Q. The solid lines indicate steps in the operational semantics (omitting stacks). The dashed lines indicate in which order weakest preconditions are calculated.

such that for every **PL** program C and every postcondition Q, the predicate $\mathsf{wp}\,[C]\,(Q)$ coincides with the semantic weakest precondition for total correctness (cf. Definition 2.24). We refer to the transformer wp as the *weakest precondition calculus* in order to distinguish it from the semantic notion of weakest preconditions considered in Definition 2.24.

Since we are initially given a program and a postcondition, wp is a *backward transformer*. That is, as illustrated in Figure 2.6 for a conditional statement, we move backward through the syntax tree of a given program, say C: Starting at the last program statement of every branch of C, we compute the weakest precondition with respect to the given postcondition, say Q. The resulting preconditions then serve as postconditions for the next statement in each branch until we have computed a precondition for the whole program C.

Definition 2.25 (Weakest Precondition Calculus [Dij75; Dij76]) The *weakest precondition calculus* wp is defined by structural induction on **PL** programs according to the rules in Figure 2.7.

Let us go over the individual rules for wp stated in Figure 2.7.

Effectless statements Since `skip` immediately terminates and does not have any effect on the stack, a stack s satisfies postcondition Q after execution of

C	$\mathsf{wp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(Q))$
`if` $(\,B\,)$ $\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$(B \wedge \mathsf{wp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wp}\,[C_2]\,(Q))$
`while` $(\,B\,)$ $\{\,C'\,\}$	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda I.\ (B \wedge \mathsf{wp}\,[C']\,(I)) \vee (\neg B \wedge Q)$

Figure 2.7: Inductive definition of weakest preconditions. Here, the function $\mathfrak{W}$ is called the characteristic function of loop `while` $(\,B\,)$ $\{\,C'\,\}$ with respect to postcondition Q.

`skip`, i.e., stack $\mathfrak{s} \models Q$, if and only if it also satisfies Q before execution of `skip`. Hence, the weakest precondition transformer for `skip` is

$$\mathsf{wp}\,[\texttt{skip}]\,(Q) \;\triangleq\; Q.$$

Assignments To formalize the weakest precondition transformer for assignments, we first introduce notation for updating values assigned to variables. This corresponds to lifting variable substitution for stacks to predicates. Variable substitution has been introduced in Definition 2.6, page 23, for the same purpose when considering an operational semantics.

Definition 2.26 (Substitution for Predicates [NN92, p. 178]) The *substitution* of variable $x \in$ **Vars** by arithmetic expression E in predicate $P \in$ **Pred** is defined as

$$P\,[x/E] \;\triangleq\; \lambda\mathfrak{s}.\, P(\mathfrak{s}\,[x/E\,(\mathfrak{s})]).$$

The weakest precondition transformer for assignments is then defined as

$$\mathsf{wp}\,[x := E]\,(Q) \;\triangleq\; Q\,[x/E]\,.$$

To understand its correctness, consider the execution of an assignment $x := E$ on a stack $\mathfrak{s}$ according to our operational semantics (cf. Definition 2.7):

$$\langle x := E,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\,[x/E\,(\mathfrak{s})]\,\rangle$$

Now, if the stack obtained after execution of the assignment satisfies our post-condition Q, we obtain from the definition of substitution for predicates:

$$\begin{aligned} & \mathfrak{s}\,[x/E\,(\mathfrak{s})] \models Q \\ \text{iff} \quad & Q\,(\mathfrak{s}\,[x/E\,(\mathfrak{s})]) = \mathsf{true} \\ \text{iff} \quad & Q\,[x/E](\mathfrak{s}) = \mathsf{true} \\ \text{iff} \quad & \mathfrak{s} \models Q\,[x/E]. \end{aligned}$$

Hence, our initial stack $\mathfrak{s}$ has to satisfy the proposed weakest precondition.

Sequential composition For the sequential composition $C_1\,;\,C_2$, recall that the weakest precondition calculus performs backward reasoning. Hence, let us first consider the weakest precondition of program C_2. The predicate $\mathsf{wp}\,[C_2]\,(Q)$ captures the set all stacks $\mathfrak{s}$ such that executing C_2 on $\mathfrak{s}$ terminates and leads to a stack $\mathfrak{s}'$ such that $\mathfrak{s}' \models Q$. Consequently, if running program $C_1\,;\,C_2$ on a stack $\mathfrak{s}$ terminates with a stack $\mathfrak{s}' \models Q$, then running program C_1 on $\mathfrak{s}$ has to terminate with a stack $\mathfrak{s}''$ that satisfies $\mathsf{wp}\,[C_2]\,(Q)$. Hence, we determine the weakest precondition of program C_1 with respect to postcondition $\mathsf{wp}\,[C_2]\,(Q)$. The weakest precondition for sequential composition is thus defined as

$$\mathsf{wp}\,[C_1\,;\,C_2]\,(Q) \;\triangleq\; \mathsf{wp}\,[C_1]\,(\,\mathsf{wp}\,[C_2]\,(Q)\,)\,.$$

Conditionals For the conditional statement $\texttt{if}\;(\,B\,)\;\{\,C_1\,\}\;\texttt{else}\;\{\,C_2\,\}$ the weakest precondition transformer reflects the case distinction applied in our operational semantics (Definition 2.7): If the guard B is satisfied in the initial stack, then program C_1 is executed and we compute the weakest precondition $\mathsf{wp}\,[C_1]\,(Q)$. Otherwise, the program C_2 is executed and we compute the weakest precondition $\mathsf{wp}\,[C_2]\,(Q)$. The weakest precondition of the conditional then considers both cases, i.e., the union of all captured stacks. Hence, it is defined as

$$\begin{aligned} & \mathsf{wp}\,[\texttt{if}\;(\,B\,)\;\{\,C_1\,\}\;\texttt{else}\;\{\,C_2\,\}]\,(Q) \\ \triangleq\; & (B \wedge \mathsf{wp}\,[C_1]\,(Q)) \;\vee\; (\neg B \wedge \mathsf{wp}\,[C_2]\,(Q))\,. \end{aligned}$$

Loops The weakest precondition transformer for loops is more involved. We first observe that the loop statement $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ exhibits exactly the same behavior as the program

$$\texttt{if}\;(\,B\,)\;\{\,C\,;\,\texttt{while}\,(\,B\,)\,\{\,C\,\}\,\}\;\texttt{else}\;\{\,\texttt{skip}\,\}\,.$$

We may thus use both programs interchangeably. A formal proof of this property is found in [NN92, Lemma 2.5]. Since both programs are (semantically) identical, we can derive the following equation:

$$
\begin{aligned}
& \mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q) \\
= \;& [\![\text{ by the above identity }]\!] \\
& \mathsf{wp}\,[\texttt{if}\,(\,B\,)\,\{\,C\,;\,\texttt{while}\,(\,B\,)\,\{\,C\,\}\,\}\,\texttt{else}\,\{\,\texttt{skip}\,\}]\,(Q) \\
= \;& [\![\text{ by the weakest precondition of conditionals }]\!] \\
& (B \wedge \mathsf{wp}\,[C\,;\,\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q)) \vee (\neg B \wedge \mathsf{wp}\,[\texttt{skip}]\,(Q)) \\
= \;& [\![\text{ by the weakest precondition of sequential composition and } \texttt{skip}]\!] \\
& (B \wedge \mathsf{wp}\,[C]\,(\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q))) \vee (\neg B \wedge Q)\,.
\end{aligned}
$$

Substituting the predicate $\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q)$ by some placeholder, say I, then yields the equation

$$I \;=\; \underbrace{(B \wedge \mathsf{wp}\,[C]\,(I)) \vee (\neg B \wedge Q)}_{\triangleq\, \mathfrak{W}(I)},$$

where we call the right-hand side, i.e., $\mathfrak{W}\colon$ **Pred** $\to$ **Pred**,the *characteristic function* of the loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to postcondition Q. Since $\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q)$ is a solution of the above equation, it is a *fixed point* of $\mathfrak{W}$. Fixed points of $\mathfrak{W}$ exist due to the following observation.[7]

Lemma 2.27 (Monotonicity) For every **PL** program C, $\mathsf{wp}\,[C]$ is monotone.

Proof. By induction on the structure of **PL** programs. □

Monotonicity of the characteristic function $\mathfrak{W}$ then follows from monotonicity of $\mathsf{wp}\,[C]$, conjunction $\wedge$, and disjunction $\vee$. Consequently, Tarski and Knaster's theorem (Theorem A.11) guarantees that $\mathfrak{W}$ has a fixed point.

The weakest precondition of loops for total correctness is not an arbitrary fixed point of the characteristic function, but the *least* fixed point. Hence, the weakest precondition of $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to postcondition Q is

$$
\begin{aligned}
\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q) \;&\triangleq\; \mathsf{lfp}\,(\mathfrak{W})\,, \text{ where} \\
\mathfrak{W} \;&\triangleq\; \lambda I.\; (B \wedge \mathsf{wp}\,[C]\,(I)) \vee (\neg B \wedge Q)\,.
\end{aligned}
$$

Why do we choose the least fixed point? Consider the never-terminating program $\texttt{while}\,(\,\mathsf{true}\,)\,\{\,\texttt{skip}\,\}$. The corresponding characteristic function is

$$\mathfrak{W}(I) \;=\; (\mathsf{true} \wedge \mathsf{wp}\,[\texttt{skip}]\,(I)) \vee (\neg\mathsf{true} \wedge Q) \;=\; I.$$

Hence, every predicate in **Pred** is a fixed point of $\mathfrak{W}$. However, since we consider total correctness, the only reasonable precondition is false—the least predicate in the complete lattice $(\,\textbf{Pred}, \Rightarrow\,)$.

[7] see, for example, [Dij76, Property 2, p. 18].

This concludes the inductive definition of weakest preconditions for **PL** programs. We note that the weakest precondition calculus summarized in Figure 2.7 is sound in the sense that it coincides with our earlier semantic definition of weakest preconditions introduced in Definition 2.24.

Theorem 2.28 (Soundness of the Weakest Precondition Calculus) Let $\mathsf{wp}\,[C]\,(Q)$ be the predicate computed for program C with respect to postcondition $Q \in \mathbf{Pred}$ by the transformer defined in Figure 2.7. Then

$$\mathsf{wp}\,[C]\,(Q) \;=\; \sup\,\{\, P \mid \langle\, P \,\rangle\; C \;\langle\, Q \,\rangle \text{ is valid for total correctness} \,\}\,.$$

Proof. By induction on the structure of **PL** programs. Details are found, for example, in [NN92, p. 194]. □

Now that we have rules to compute weakest preconditions, let us apply them to verify a simple **PL** program.

Example 2.29 Consider the **PL** program C_{absInc} given by

$$\texttt{if}\ (\,x < 0\,)\ \{\, x := -x \,\}\ \texttt{else}\ \{\, \texttt{skip} \,\}\,;\, y := x + 1.$$

We would like to show that C_{absInc} increments the absolute value of variable x and stores the result in variable y. We thus compute the weakest precondition with respect to postcondition $y = |x| + 1$. This involves the computation steps presented in Figure 2.8. The individual steps are depicted as source code annotations which are meant to be read from bottom to top. The lowermost annotation is our chosen postcondition and the uppermost annotation is our derived precondition. Two predicates with a program in between represent a valid Hoare triple, i.e., in Figure 2.8, we verify that

$$\langle\, \mathsf{true} \,\rangle\; C_{\mathsf{absInc}} \;\langle\, y = |x| + 1 \,\rangle$$

is valid. For two predicates with an atomic statement in between, e.g., the last three lines in Figure 2.8, the upper predicate is the weakest precondition of that statement with respect to the lower predicate. For two predicates with a comment in between, e.g., the first three lines in Figure 2.8, we prove that the upper predicate implies the lower one; a justification of that implication is provided in the comment. To improve readability, we highlight how predicates have changed in each step and black-75 out the remainder.

Since the computations in Figure 2.8 yield that true implies the weakest precondition, we conclude that, for every initial stack, the program C_{absInc} terminates and stores the absolute value of x plus one in variable y.

```
// true
// ⟹ ⟦ elementary predicate logic ⟧
// (x < 0 ∧ −x + 1 = |−x| + 1) ∨ (x ≥ 0 ∧ x + 1 = |x| + 1)
if ( x < 0 ) {
    // −x + 1 = |−x| + 1
    x := −x
    // x + 1 = |x| + 1
} else {
    // x + 1 = |x| + 1
    skip
    // x + 1 = |x| + 1
};
// x + 1 = |x| + 1
y := x + 1
// y = |x| + 1
```

Figure 2.8: Computation of the weakest precondition of the program C_{absInc}.

2.3.5 Proof Rules for Weakest Preconditions

While the weakest precondition calculus enables us to systematically perform program verification compositionally on the program structure, proving programs correct is intricate. In this section, we discuss additional proof rules to support program verification with weakest preconditions.

Proof Rules for Loops The most involved part of computing weakest preconditions is arguably reasoning about loops because it requires the computation of least fixed points. To deal with loops, recall the characteristic function $\mathfrak{W}$ of loop `while ( B ) { C }` with respect to some postcondition Q:

$$\mathfrak{W} \triangleq \lambda I.\ (B \wedge \mathsf{wp}\,[C]\,(I)) \vee (\neg B \wedge Q)\,.$$

We are then concerned with reasoning about the least fixed point of $\mathfrak{W}$. A simple condition on the least fixed point is obtained from a step in Tarski's proof of the fixed point theorem which is also known as Park's Lemma (Corollary A.12):

$$\forall I \in \mathbf{Pred}\colon \quad \mathfrak{W}(I) \Rightarrow I \text{ implies } \mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q) \Rightarrow I.$$

Unfortunately, this rule only provides a *necessary* condition on the least fixed point, but not a sufficient one. This means that there is no guarantee that the predicate I is indeed a precondition such that the Hoare triple

$$\langle\, I \,\rangle \; \texttt{while}\,(\, B \,)\,\{\, C \,\} \; \langle\, Q \,\rangle$$

is valid for total correctness. To obtain a sufficient rule, we make use of a stronger fixed point theorem due to Kleene [Kle+52] which relies on the fact that the weakest precondition calculus is continuous.

Lemma 2.30 ([NN92, p. 129]) The predicate transformer $\mathsf{wp}\,[C]$ is continuous. Moreover, the characteristic function $\mathfrak{W}$ of loop $\texttt{while}\,(\, B \,)\,\{\, C \,\}$ with respect to postcondition $Q \in \mathbf{Pred}$ is continuous.

Proof. Continuity of $\mathsf{wp}\,[C]$ is shown by induction on the structure of **PL** programs. The proof for continuity of the characteristic function then follows from continuity of $\mathsf{wp}\,[C]$, conjunction, and disjunction. □

With continuity at hand, Kleene's theorem (Theorem A.16) yields the following characterization of the least fixed point:

$$\mathsf{wp}\,[\texttt{while}\,(\, B \,)\,\{\, C \,\}]\,(Q) \;=\; \mathsf{lfp}\,(\mathfrak{W}) \;=\; \lim_{n\to\infty} \mathfrak{W}^n\,(\mathsf{false})\,.$$

We may thus attempt to compute the least fixed point by means of a fixed point iteration. That is, we compute the sequence

$$\begin{aligned} \mathfrak{W}^0(\mathsf{false}) &= \mathsf{false} \\ \mathfrak{W}^1(\mathsf{false}) &= \mathfrak{W}(\mathsf{false}) \\ \mathfrak{W}^2(\mathsf{false}) &= \mathfrak{W}(\mathfrak{W}^1(\mathsf{false})) \\ &\vdots \end{aligned}$$

until, for some natural number $n \in \mathbb{N}$, we have $\mathfrak{W}^n(\mathsf{false}) = \mathfrak{W}^{n+1}(\mathsf{false})$. In general, the existence of such a natural number is, however, not guaranteed as the fixed point iteration may only converge in the limit. Kleene's theorem still gives us a proof rule: If we find a sequence $\{\, I_n \,\}_{n\in\mathbb{N}}$ such that

$$I_0 \;=\; \mathsf{false} \quad \text{and} \quad \forall n \in \mathbb{N}\colon \quad \mathfrak{W}(I_n) \Rightarrow I_{n+1} \quad \text{and} \quad I \;=\; \lim_{n\to\infty} I_n,$$

then we can conclude that

$$I \;\Rightarrow\; \mathsf{lfp}\,(\mathfrak{W}) \;=\; \mathsf{wp}\,[\texttt{while}\,(\, B \,)\,\{\, C \,\}]\,(Q)\,.$$

Example 2.31 Consider the program C_{fac} below.

$$
\begin{array}{ll}
 & x := 10\,; \\
 & y := 1\,; \\
C_{\mathsf{loop}}: & \texttt{while}\,(\,x > 0\,)\,\{ \\
 & \quad y := y \cdot x\,; \\
 & \quad x := x - 1 \\
 & \}
\end{array}
$$

Let us verify that C_{fac} always terminates with $y = 10!$, i.e., variable y is set to the factorial of ten. We first compute the characteristic function $\mathfrak{W}(R)$ of the loop C_{loop} with respect to postcondition $y = 10!$:

$$\mathfrak{W}(R) \;=\; (x \leq 0 \wedge y = 10!) \vee \big(x > 0 \wedge R\,[x/x-1]\,[y/y \cdot x]\big)$$

Our next step is to find a predicate I which implies the least fixed point of characteristic function $\mathfrak{W}$. To this end, we propose the sequence $\{\, I_n \,\}_{n \in \mathbb{N}}$ which is given by:

$$
I_n \;\triangleq\; \begin{cases} \mathsf{false}, & \text{if } n = 0 \\ x \leq 0 \wedge y = 10!, & \text{if } n = 1 \\ \\ (x \leq 0 \wedge y = 10!) \vee & \text{if } n > 1. \\ \bigwedge_{k=1}^{n-1} \left(x = k \wedge y \cdot \prod_{\ell=0}^{k-1} (x - \ell) = 10! \right), & \end{cases}
$$

It is straightforward to verify for all $n \in \mathbb{N}$ that $\mathfrak{W}(I_n) = I_{n+1}$. Then, by our proof rule based on Kleene's theorem, the limit of sequence $\{\, I_n \,\}_{n \in \mathbb{N}}$ implies the weakest precondition of the loop C_{loop}. Hence,

$$
\begin{aligned}
I \;&=\; \lim_{n \to \infty} I_n \;=\; (x \leq 0 \wedge y = 10!) \vee \bigwedge_{k \in \mathbb{N}} \left(x = k \wedge y \cdot \prod_{\ell=0}^{k-1} (x - \ell) = 10! \right) \\
&\Rightarrow \mathsf{wp}\,[C_{\mathsf{loop}}]\,(y = 10!)\,.
\end{aligned}
$$

Verifying the whole program C_{fac} then amounts to applying the rules for weakest preconditions:

$$
\begin{aligned}
&\mathsf{wp}\,[x := 10\,;\, y := 1\,;\, C_{\mathsf{loop}}]\,(y = 10!) \\
=\;&\mathsf{wp}\,[x := 10\,;\, y := 1]\,(\mathsf{wp}\,[C_{\mathsf{loop}}]\,(y = 10!))
\end{aligned}
$$

$$
\begin{aligned}
&\Leftarrow \mathsf{wp}\,[x := 10\,;\, y := 1]\,(I) && \text{(by above rule)}\\
&= I\,[y/1]\,[x/10]\\
&= (10 \leq 0 \wedge 1 = 10!) \vee \bigwedge_{k \in \mathbb{N}} \left(10 = k \wedge 1 \cdot \prod_{\ell=0}^{k-1} (10 - \ell) = 10! \right)\\
&= \mathsf{false} \vee \prod_{\ell=0}^{9} (10 - l) = 10! \;=\; \mathsf{true}.
\end{aligned}
$$

While we did not compute the exact weakest precondition, we verified that predicate true implies $\mathsf{wp}\,[C_{\mathsf{fac}}]\,(y = 10!)$. Hence, the triple

$$\langle\, \mathsf{true} \,\rangle\; C_{\mathsf{fac}}\; \langle\, y = 10! \,\rangle$$

is valid. We thus conclude that, regardless of the initial stack, program C_{fac} always terminates and stores the factorial of 10 in variable y.

Proof Rules for Compositional Reasoning about Predicates While the weakest precondition calculus enables to reason compositionally on the *program structure*, the predicates computed along the way quickly become complicated. It is thus also desirable to reason compositionally about predicates. To this end, the following rules allow to separate concerns by reasoning individually about smaller—and hopefully simpler—predicates.

Theorem 2.32 (Compositionality of wp for PL programs [Dij75; Dij76]) For all **PL** programs C and all predicates $Q, R \in$ **Pred**, we have:

(a) Strictness: $\mathsf{wp}\,[C]\,(\mathsf{false}) \;=\; \mathsf{false}.$

(b) Conjunction rule: $\mathsf{wp}\,[C]\,(Q \wedge R) \;=\; \mathsf{wp}\,[C]\,(Q) \wedge \mathsf{wp}\,[C]\,(R).$

(c) Disjunction rule: $\mathsf{wp}\,[C]\,(Q \vee R) \;=\; \mathsf{wp}\,[C]\,(Q) \vee \mathsf{wp}\,[C]\,(R).$

Proof. By induction on the structure of **PL** programs. □

2.3.6 The Weakest Liberal Precondition Calculus

Next, we turn to computing weakest preconditions for partial correctness. Hence, a triple is valid if every execution on a stack satisfying the precondition either does not terminate or terminates and satisfies the postcondition.

As for total correctness, we define a predicate transformer by induction on the structure of **PL** programs to compute weakest preconditions compositionally.

C	$\mathsf{wlp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wlp}\,[C_1]\,(\mathsf{wlp}\,[C_2]\,(Q))$
`if` $(\,B\,)\,\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$(B \wedge \mathsf{wlp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wlp}\,[C_2]\,(Q))$
`while` $(\,B\,)\,\{\,C'\,\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where
	$\mathfrak{LW} \triangleq \lambda I.\ (B \wedge \mathsf{wlp}\,[C']\,(I)) \vee (\neg B \wedge Q)$

Figure 2.9: Inductive definition of weakest liberal preconditions. $\mathfrak{LW}$ is called the liberal characteristic function of loop `while` $(\,B\,)\,\{\,C'\,\}$ with respect to Q.

This transformer maps, for every **PL** program C, postconditions to preconditions, i.e., it has the form

$$\mathsf{wlp}\,[C] : \mathbf{Pred} \to \mathbf{Pred}.$$

Since nontermination is now allowed, $\mathsf{wlp}\,[C]$ is called the weakest *liberal* precondition transformer. A summary of the inductive definition of the *weakest liberal precondition calculus* wlp is found in Figure 2.9. Let us compare wlp to the weakest precondition transformer wp for total correctness.

Since both `skip` and the assignment $x := E$ immediately terminate, their weakest liberal precondition coincides with their weakest precondition. The definitions for sequential composition $C_1\,;\,C_2$ and conditionals are also identical except that we refer to the weakest liberal precondition to deal with the subprograms C_1 and C_2.

For loops, by a similar argument as for weakest preconditions, the weakest liberal precondition of the loop `while` $(\,B\,)\,\{\,C\,\}$ with respect to postcondition Q is a fixed point of the equation

$$I \;=\; \underbrace{(B \wedge \mathsf{wlp}\,[C]\,(I)) \vee (\neg B \wedge Q)}_{\triangleq\ \mathfrak{LW}(I)},$$

where we call the right-hand side, i.e.,

$$\mathfrak{LW}\colon \mathbf{Pred} \to \mathbf{Pred},$$

the *liberal characteristic function* of the loop `while` $(\,B\,)\,\{\,C\,\}$ with respect to postcondition Q. In contrast to the weakest precondition transformer wp, however,

we take the *greatest* fixed point instead of the least fixed point. To understand why, consider—again— the never terminating program `while ( true ) { skip }`. Its characteristic function is

$$\mathfrak{LW}(I) \;=\; (\mathsf{true} \wedge \mathsf{wlp}\,[\texttt{skip}]\,(I)) \vee (\neg\mathsf{true} \wedge Q) \;=\; I.$$

Hence, every predicate in **Pred** is a fixed point of $\mathfrak{LW}$. Since we consider partial correctness, the only reasonable precondition is true—the largest predicate in the complete lattice $(\,\mathbf{Pred}, \Rightarrow\,)$.

As for wp, the existence of fixed points is guaranteed by the Knaster-Tarski fixed point theorem (Theorem A.11) and the fact, as stated below, that $\mathsf{wlp}\,[C]$—and thus also the liberal characteristic function $\mathfrak{LW}$—is monotone.

Lemma 2.33 (Monotonicity) For every **PL** program C, $\mathsf{wlp}\,[C]$ is monotone.

Proof. Analogous to the proof of Lemma 2.27. □

Hence, the weakest liberal precondition of loop `while ( B ) { C }` with respect to postcondition Q is defined as

$$\begin{aligned}\mathsf{wlp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q) &\triangleq \mathsf{gfp}\,(\mathfrak{LW})\,, \text{ where}\\ \mathfrak{LW} &\triangleq \lambda I.\ (B \wedge \mathsf{wlp}\,[C]\,(I)) \vee (\neg B \wedge Q)\,.\end{aligned}$$

For the record, let us note that the weakest liberal precondition calculus summarized in Figure 2.9 is sound in the sense that it coincides with our semantic definition of weakest preconditions introduced in Definition 2.24.

Theorem 2.34 (Soundness of the Weakest Liberal Precondition Calculus) Let $\mathsf{wlp}\,[C]\,(Q)$ be the predicate computed for program C with respect to postcondition $Q \in \mathbf{Pred}$ as defined in Figure 2.9. Then

$$\mathsf{wlp}\,[C]\,(Q) \;=\; \sup\,\{\,P \mid \langle\,P\,\rangle\;C\;\langle\,Q\,\rangle \text{ is valid for partial correctness}\,\}\,.$$

Proof. By induction on the structure of **PL** programs. A formal proof is found, for example, in [NN92, pp. 184–187]. □

As we will see in the next section, reasoning about partial correctness is often easier because we have additional proof rules at our disposal.

2.3.7 Proof Rules for Weakest Liberal Preconditions

We now turn to proof rules for reasoning about partial correctness of loops. Moreover, we collect additional rules to support compositional reasoning.

Proof Rules for Loops The most intricate part of proving partial correctness is reasoning about loops because we have to compute fixed points. Let us first recall the liberal characteristic function $\mathfrak{LW}$ of loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to some postcondition Q:

$$\mathfrak{LW} \;\triangleq\; \lambda I.\; (B \wedge \mathsf{wlp}\,[C]\,(I)) \vee (\neg B \wedge Q)\,.$$

In contrast to reasoning about total correctness, Park's Lemma (cf. Corollary A.12) yields a proof rule that is useful for verification:

Theorem 2.35 (Invariant Rule for wlp)

$$\forall I \in \mathbf{Pred}\colon \quad I \Rightarrow \mathfrak{LW}(I) \text{ implies } I \Rightarrow \mathsf{wlp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q)\,.$$

While this rule does not yield the exact weakest liberal precondition of a loop, it is sufficient to prove validity of the Hoare triple

$$\langle\, I \,\rangle\; \texttt{while}\,(\,B\,)\,\{\,C\,\}\; \langle\, Q \,\rangle.$$

Example 2.36 Recall from Example 2.31, page 48, the program C_{fac} which computes the factorial of 10 and stores the result in variable y. Let us verify the same property for partial correctness. The liberal characteristic function $\mathfrak{LW}(R)$ of the loop C_{loop} with respect to postcondition $y = 10!$ is given by

$$\mathfrak{LW}(R) \;=\; (x \leq 0 \wedge y = 10!) \vee \big(x > 0 \wedge R\,[x/x-1]\,[y/y \cdot x]\big)\,.$$

In contrast to the total correctness proof in Example 2.31, it suffices to propose a single suitable invariant, say

$$I \;\triangleq\; (x \leq 0 \wedge y = 10!) \vee \bigwedge_{k \in \mathbb{N}} \left(x = k \wedge y \cdot \prod_{\ell=0}^{k-1} (x - \ell) = 10! \right).$$

Verifying that I is indeed an invariant, i.e., proving that

$$I \;\Rightarrow\; \mathfrak{LW}(I),$$

is straightforward. By Theorem 2.35, we then conclude that

$$I \;\Rightarrow\; \mathsf{wlp}\,[C_{\mathsf{loop}}]\,(y = 10!)\,.$$

The remaining calculations to verify the whole program C_{fac} are analogous to the steps in Example 2.31. Hence, using a different proof rule, we have verified that $\langle\, \mathsf{true} \,\rangle\; C_{\mathsf{fac}}\; \langle\, y = 10! \,\rangle$ is valid for partial correctness.

Furthermore, we notice that greatest fixed points can also be computed iteratively because the weakest liberal precondition calculus is continuous.

Lemma 2.37 The predicate transformer $\mathsf{wlp}\,[C]$ is continuous.
Moreover, the liberal characteristic function $\mathfrak{LW}$ of loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to postcondition $Q \in \mathbf{Pred}$ is continuous.

Proof. Analogously to the proof of Lemma 2.30. □

Hence, similar to wp, Kleene's theorem (Theorem A.16) yields the following characterization of greatest fixed points:

$$\mathsf{wlp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(Q) \;=\; \mathsf{gfp}\,(\mathfrak{LW}) \;=\; \lim_{n\to\infty} \mathfrak{LW}^n\,(\mathsf{true})\,.$$

As for total correctness, computing the greatest fixed point by means of the fixed point iteration may only converge in the limit.

Proof Rules for Compositional Reasoning about Predicates The weakest liberal precondition calculus enjoys similar compositionality properties as the weakest precondition calculus.

Theorem 2.38 (Compositionality of wlp for PL programs [Dij75; Dij76]) For all **PL** programs C and all predicates $Q, R \in \mathbf{Pred}$, we have:

(a) Strictness: $\mathsf{wlp}\,[C]\,(\mathsf{true}) \;=\; \mathsf{true}$.

(b) Conjunction rule: $\mathsf{wlp}\,[C]\,(Q \wedge R) \;=\; \mathsf{wlp}\,[C]\,(Q) \wedge \mathsf{wlp}\,[C]\,(R)$.

(c) Disjunction rule: $\mathsf{wlp}\,[C]\,(Q \vee R) \;=\; \mathsf{wlp}\,[C]\,(Q) \vee \mathsf{wlp}\,[C]\,(R)$.

Proof. By induction on the structure of **PL** programs. □

An additional proof rule stems from the observation that some programs C have no effect on a predicate Q, because C manipulates only variables that are irrelevant for Q. To make this observation precise, let us first define the set of variables relevant for the evaluation of an expression. Moreover, for programs, we consider the set of variables modified by a program, i.e., variables that occur on the left-hand side of an assignment.

Definition 2.39 (Relevant Variables [7, Def. 4]) The set $\mathbf{Vars}\,(E) \subseteq \mathbf{Vars}$ of variables occurring in an arithmetic expression E is given by

$$x \in \mathbf{Vars}\,(E) \quad \text{iff} \quad \exists \mathfrak{s} \in \mathbf{Stacks}\; \exists u, v \in \mathbb{Z}\colon\; E\,[x/u]\,(\mathfrak{s}) \neq E\,[x/v]\,(\mathfrak{s})\,.$$

C	$\mathbf{Mod}(C)$
`skip`	$\emptyset$
$x := E$	$\{ x \}$
$C_1 ; C_2$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`if` (B) $\{ C_1 \}$ `else` $\{ C_2 \}$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`while` (B) $\{ C' \}$	$\mathbf{Mod}(C')$

Figure 2.10: Inductive definition of the variables modified by a **PL** program C.

The set $\mathbf{Vars}(P) \subseteq \mathbf{Vars}$ of variables occurring in a predicate P is given by

$$x \in \mathbf{Vars}(P) \quad \text{iff} \quad \exists \mathfrak{s} \in \mathbf{Stacks}\ \exists u, v \in \mathbb{Z}\colon\ P[x/u](\mathfrak{s}) \neq P[x/v](\mathfrak{s}).$$

Finally, the set $\mathbf{Mod}(C)$ of variables modified by **PL** program C is determined by the inductive definition in Figure 2.10, page 54.

The above definition mixes a semantic notion of sets of variables (for arithmetic and Boolean expressions) and a syntactic notion (for **PL** programs). The reason for this is that we never introduced an explicit syntax for expressions. We thus have to fall back to a semantic notion. Once a syntax for expressions is fixed, however, any sensible practitioner will insist that variables occurring in expressions must be determined by a syntactic notion that can be easily checked.

Now, if no variable modified by program C is relevant for predicate Q, then computing the weakest precondition of C has no effect on Q (as long as C terminates). This is also known as the axiom of invariance.

Theorem 2.40 (Invariance [Apt81]) Let C be a **PL** program and Q be a predicate such that $\mathbf{Mod}(C) \cap \mathbf{Vars}(Q) = \emptyset$. Then $Q \Rightarrow \mathsf{wlp}[C](Q)$.

Proof. By induction on the structure of **PL** programs. □

Unfortunately, the invariance rule does not hold for total correctness. Consider, for instance, the counterexample

$$\mathsf{wp}[\texttt{while}(\mathsf{true})\{\texttt{skip}\}](x > 0) \;=\; \mathsf{false} \;\neq\; x > 0.$$

For *terminating* programs, however, we have $\mathsf{wp}[C](Q) = \mathsf{wlp}[C](Q) = Q$.

Example 2.41 In the annotated program C below, we verify that whenever C terminates, variable y is equal to one. Hence, our goal is to reason about the weakest liberal precondition of C for postcondition $y = 1$. Due the rule of invariance, this task is straightforward: We have $\mathbf{Vars}\,(y = 1) = \{\, y \,\}$ and $\mathbf{Mod}\,(C_{\mathsf{loop}}) = \{\, x \,\}$. Hence, $y = 1$ implies $\mathsf{wlp}\,[C_{\mathsf{loop}}]\,(y = 1)$. The remaining proof then amounts to reasoning about an assignment.

$$
\begin{array}{ll}
 & \text{// } \mathsf{true} \\
 & y := 1\,; \\
 & \text{// } y = 1 \\
C_{\mathsf{loop}}\colon & \texttt{while}\,(\,x \neq 0\,)\,\{ \\
 & \quad \texttt{if}\ (\,x > 0\,)\ \{\, x := -x + 13 \,\}\ \texttt{else}\ \{\, x := -x - 1 \,\} \\
 & \} \\
 & \text{// } y = 1
\end{array}
$$

2.3.8 Hoare Logic

Program verification by means of checking Hoare triples enables proving correctness compositionally on the program structure. We studied two weakest precondition calculi—one for total and one for partial correctness—to systematically derive correct triples. The main advantage of weakest preconditions is that they provide solid foundations for program verification: They are based upon established results from domain theory. This is particularly compelling in light of the fact that various unsound proof systems for (extensions of) Hoare logic have been proposed in the past (cf. [Nip02] for an overview). Moreover, weakest preconditions avoid ambiguous choices during the verification process, e.g., the need to find suitable intermediate predicates. We consider extensions of weakest precondition calculi to incorporate procedures in Chapter 3, dynamic memory allocation in Chapter 4, and probabilistic behavior in Chapter 6.

For practical verification, there are some caveats when using weakest preconditions. In particular, due to backward reasoning, we cannot take an already known precondition into account. Hence, we are often forced to prove more general properties than necessary. For example, to reason about the program below with weakest preconditions, we first have to analyze the loop without any knowledge about the initial value of variable x.

$$x := 10\,;\ \texttt{while}\,(\,x > 0\,)\,\{\, x := x - 1 \,\},$$

This justifies to built more flexible rules in the original style of Floyd [Flo67] and Hoare [Hoa69] on top of weakest preconditions.

Let us briefly consider how to derive inference rules for partial correctness. By Definition 2.24, we have

$$\begin{aligned} &\left\langle\, \mathsf{wlp}\,[C]\,(Q)\, \right\rangle\, C\, \left\langle\, Q \,\right\rangle \text{ is valid} \\ \text{and} \quad &P \Rightarrow \mathsf{wlp}\,[C]\,(Q) \text{ implies } \left\langle\, P \,\right\rangle\, C\, \left\langle\, Q \,\right\rangle \text{ is valid.} \end{aligned}$$

Moreover, by monotonicity of weakest liberal preconditions (Lemma 2.33),

$$Q' \Rightarrow Q \text{ implies } \mathsf{wlp}\,[C]\,(Q') \Rightarrow \mathsf{wlp}\,[C]\,(Q)\,.$$

Hence, if $P \Rightarrow \mathsf{wlp}\,[C]\,(Q')$ and $Q' \Rightarrow Q$, then the triple $\left\langle\, P \,\right\rangle\, C\, \left\langle\, Q \,\right\rangle$ is valid. This yields the rule of consequence in Hoare logic, i.e.,

$$\frac{P \Rightarrow P' \quad \left\langle\, P' \,\right\rangle\, C\, \left\langle\, Q' \,\right\rangle \quad Q' \Rightarrow Q}{\left\langle\, P \,\right\rangle\, C\, \left\langle\, Q \,\right\rangle}\ \text{cons}$$

All other inference rules of Hoare logic for partial correctness, which are found in Figure 2.11, page 57, are then straightforward to derive from the inductive definition of weakest liberal preconditions, the rule of consequence and derived proof rules. Further details on Hoare logic are found, for example, in [Hoa69; NN92; Win93]. In particular, we refer the reader to [Apt81] for a good overview.

This concludes our tour through fundamental principles for reasoning about the correctness of programs. In the remaining chapters, we study various extensions of these approaches with respect to more powerful programming language constructs and automation.

$$\frac{}{\langle\, P \,\rangle\ \texttt{skip}\ \langle\, P \,\rangle}\ \text{skip} \qquad \frac{}{\langle\, P\,[x/E] \,\rangle\ x := E\ \langle\, P \,\rangle}\ \text{assign}$$

$$\frac{\langle\, P \,\rangle\ C_1\ \langle\, R \,\rangle \quad \langle\, R \,\rangle\ C_2\ \langle\, Q \,\rangle}{\langle\, P \,\rangle\ C_1 ; C_2\ \langle\, Q \,\rangle}\ \text{seq}$$

$$\frac{\langle\, B \wedge P \,\rangle\ C_1\ \langle\, Q \,\rangle \qquad \langle\, \neg B \wedge P \,\rangle\ C_2\ \langle\, Q \,\rangle}{\langle\, P \,\rangle\ \texttt{if}\ (\, B \,)\ \{\, C_1 \,\}\ \texttt{else}\ \{\, C_2 \,\}\ \langle\, Q \,\rangle}\ \text{if}$$

$$\frac{\langle\, B \wedge P \,\rangle\ C\ \langle\, P \,\rangle}{\langle\, P \,\rangle\ \texttt{while}\,(\, B \,)\,\{\, C \,\}\ \langle\, \neg B \wedge P \,\rangle}\ \text{while}$$

$$\frac{P \Rightarrow P' \quad \langle\, P' \,\rangle\ C\ \langle\, Q' \,\rangle \quad Q' \Rightarrow Q}{\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle}\ \text{cons}$$

$$\frac{\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle \quad \langle\, R \,\rangle\ C\ \langle\, S \,\rangle}{\langle\, P \wedge R \,\rangle\ C\ \langle\, Q \wedge S \,\rangle}\ \text{conjunction}$$

$$\frac{\mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(P)\ =\ \emptyset}{\langle\, P \,\rangle\ C\ \langle\, P \,\rangle}\ \text{invariance}$$

Figure 2.11: The inference rules that make up the proof system of Hoare logic for **PL** programs with respect to partial correctness.

Many algorithms are recursive in nature. This holds in particular for algorithms operating on dynamic data structures which are often themselves based on recursive definitions (cf. [Knu97, Chapter 2]).

Before we consider reasoning about programs that manipulate dynamic data structures in Chapter 4, we thus extend **PL** by support for recursive procedures with call-by-value parameters and return values. This leads to the *Procedural Programming Language*, $\mathbf{P^2L}$ for short.

Outline The syntax of $\mathbf{P^2L}$, its scoping model, and its operational semantics are introduced in Section 3.1. The abstraction techniques presented in Section 2.2 to design program analyses for **PL** programs work analogously for $\mathbf{P^2L}$ programs. For program verification, however, reasoning about recursive procedure calls requires a careful extension of the weakest precondition calculi introduced in Section 2.3. Proof rules for $\mathbf{P^2L}$ programs are presented in Section 3.2.

3.1 The Procedural Programming Language

The procedural programming language $\mathbf{P^2L}$ is designed to support common features of imperative programming languages, such as recursive procedures with parameters. Since our ultimate goal is formal reasoning about programs, we attempt to keep the formal semantics as simple as possible. To this end, $\mathbf{P^2L}$ contains a few internal statements to simplify the treatment of procedure parameters and return values. Moreover, the *scope* of every procedure in $\mathbf{P^2L}$ is limited to its parameters and local variables, i.e., a procedure cannot access any variable from the outside. Consequently, a value can only be passed from a program to a procedure via one of the procedure's parameters. Analogously, a value can only be passed from a procedure to its calling program via the returned value. The values of all variables which are local to a procedure are lost upon the procedure's termination.

3.1.1 Syntax

Let us first extend the syntax of **PL** by procedure calls and auxiliary statements to deal with the passing of parameters and return values. In addition to the conventions introduced in Section 2.1.1, we adhere to the following conventions:

- Let **Procs** be a finite set of *procedure names*, such as F, G, or H. With every procedure name $F \in \mathbf{Procs}$, we associate a natural number $n \in \mathbb{N}$ of *parameters* that have to be passed to procedure F.
- We occasionally denote by $\vec{E}$ a sequence $E_1, \ldots, E_n$ of length $|\vec{E}| = n \in \mathbb{N}$.
- We assume that the set of variables **Vars** contains a special variable out that models the return value of procedures.

With these conventions in mind, the syntax of the procedural programming language is defined as follows.

Definition 3.1 (Syntax of the Procedural Programming Language) The set of programs written in the *Procedural Programming Language with Auxiliaries*, denoted $\mathbf{P^2LA}$, is given by the context-free grammar below (**PL** statements are displayed in black-75):

$$
\begin{array}{rll}
C \;\to & x := F(\vec{E}) & \text{(procedure call)} \\
\mid & \texttt{enter} & \text{(enter scope)} \\
\mid & \texttt{invoke}\ F & \text{(invoke procedure)} \\
\mid & \texttt{leave} & \text{(leave scope)} \\
 & & \\
\mid & \texttt{skip} & \text{(effectless program)} \\
\mid & x := E & \text{(assignment)} \\
\mid & C\,;\,C & \text{(sequential composition)} \\
\mid & \texttt{if}\,(\,B\,)\,\{\,C\,\}\,\texttt{else}\,\{\,C\,\} & \text{(conditional choice)} \\
\mid & \texttt{while}\,(\,B\,)\,\{\,C\,\}, & \text{(loop)}
\end{array}
$$

where $x \in \mathbf{Vars}$ is a variable, $F \in \mathbf{Procs}$ is a procedure name, $\vec{E}$ is a sequence of arithmetic expressions whose length $|\vec{E}|$ matches the number of parameters associated with procedure F, E is an arithmetic expression, and B is a Boolean expression. Moreover, the set of programs in the *Procedural Programming Language*, denoted $\mathbf{P^2L}$, consists of all $\mathbf{P^2LA}$ programs, which contain neither `enter` nor `invoke` F nor `leave` statements.

Intuitively, a procedure call $x := F(\vec{E})$ executes the program corresponding to procedure F. The values of its parameters are determined by evaluating the arithmetic expressions in $\vec{E}$ on invoking F. The return value assigned to variable x upon termination is determined by the last value assigned to variable out. To be precise about what "the program corresponding to procedure F" means, every $\mathbf{P^2L}$ program must be accompanied by procedure declarations.

Definition 3.2 (Procedure Declaration) A *procedure declaration* $F(\vec{x})\{C\}$ consists of a procedure name $F \in \mathbf{Procs}$, a sequence $\vec{x}$ of pairwise distinct variables in **Vars**, which are called the *parameters* of F, and a $\mathbf{P^2L}$ program C, which is called the *procedure body*. Moreover, to ensure that the return value of every procedure is well-defined, the last statement of every execution of procedure body C must be an assignment to variable out.

For every $\mathbf{P^2L}$ program, we assume the presence of two functions

$$\mathsf{body}\colon \mathbf{Procs} \to \mathbf{P^2L} \qquad \text{and} \qquad \mathsf{param}\colon \mathbf{Procs} \to \mathbf{Vars}^*$$

that, for every procedure name $F \in \mathbf{Procs}$, yield the procedure body and the sequence of parameters of F's declaration, respectively.

The set **Procs** of procedure names under consideration is usually defined implicitly by the set of all procedure names that are declared alongside a given $\mathbf{P^2L}$ program. The same holds for the functions body and param.

Example 3.3 Consider the following declaration of a procedure *mult*:

C_{body}:

```
mult(x, y) {
    if (y > 0) {
        z := mult(x, y − 1);
        out := x + z
    } else {
        out := 0
    }
}
```

The procedure body of *mult* is $\mathsf{body}(mult) = C_{\mathsf{body}}$. Its arguments are $\mathsf{param}(mult) = (x, y)$. Intuitively, *mult* takes two parameters x and y and—if y is non-negative—returns the product of x and y. This is achieved by adding up x recursively y times. Otherwise, the procedure returns zero.

Assumption 3.4 We use a special variable void to indicate that we are not interested in the value returned by a procedure. By convention, we require that a program never reads the value of void. That is, apart from the left-hand side of assignments, void occurs nowhere in programs. For obvious reasons, void must not occur in any property we would like to reason about.

3.1.2 Scoping

In order to assign semantics to procedure calls, we first discuss the scope of variables, which determines the variables that can be accessed within a procedure's body. To this end, let ⧈ be a special *scope symbol*. Intuitively, variable names prefixed with a scope symbol are *out of scope*. These variables correspond to local variables of other procedures. They thus cannot be accessed by the currently executed procedure. Conversely, variables without a scope symbol are *within scope*. These variables correspond to the local variables of the current procedure. They thus can be accessed. The number of scope symbols in front of a variable name, say ⧈⧈x, reflects the nesting of depth of procedure calls, i.e., how many nested procedure calls have to terminate until the variable is within scope again.

Assumption 3.5 To prevent programmers with ill intent from circumventing the scoping system, we assume a syntactic check that forbids usage of the scope symbol, i.e., programs may not contain statements of the form ⧈$x := E$ or ⧈$y > 0$. The same condition does, of course, *not* apply internally, i.e., we will use such statements in our semantics.

Moreover, we assume that every variable can be prefixed by a scope symbol arbitrarily often. That is, if x is a variable name in **Vars**, then ⧈x is a variable name in **Vars** as well.

How does entering a new scope and leaving the current scope affect the stack?[1] Whenever we enter a new scope, e.g., due to a procedure call, we add a scope symbol ⧈ to the front of every variable name. Hence, after entering a scope, the value of variable x in the original stack is stored in variable ⧈x. What is then the value of variable x in the stack after entering a new scope? Variable x is a local variable that does not store any value of the original stack. We thus initialize it with zero. In other words, entering a new scope intuitively corresponds to simultaneously performing infinitely many assignments of the form

$$x := 0 \quad ⧈\, x := x \quad ⧈\, ⧈ x := ⧈ x \quad \ldots$$

[1] We remind the reader that we use the term "stack" to refer to an evaluation of variables and not in the sense of a "call stack". The latter is encoded in the names of variables.

	$\mathfrak{s}\,[-\boxdot]$	$\mathfrak{s}$	$\mathfrak{s}\,[+\boxdot]$
x	7	3	0
$\boxdot x$	11	7	3
$\boxdot\,\boxdot\,x$	13	11	7
$\boxdot\,\boxdot\,\boxdot x$	⋮	13	11
$\boxdot\,\boxdot\,\boxdot\,\boxdot\,x$		⋮	13

Figure 3.1: The effect of scope increment $\mathfrak{s}\,[+\boxdot]$ and scope decrement $\mathfrak{s}\,[-\boxdot]$ on the evaluation of variable x (at different scoping levels) in a stack $\mathfrak{s}$.

Conversely, whenever we leave the current scope, e.g., because a procedure terminates, we remove one scope symbol from every variable name. Hence, after leaving the scope, the value of variable $\boxdot x$ is now stored in variable x. The original value of variable x, which was local to the scope we just left, is lost. In other words, leaving a scope intuitively corresponds to simultaneously performing infinitely many assignments of the form

$$x := \boxdot x \quad \boxdot x := \boxdot\,\boxdot x \quad \boxdot\,\boxdot x := \boxdot\,\boxdot\,\boxdot x \quad \ldots$$

We formally capture the effect of entering and leaving scopes on a stack by two operations that are called scope increment and scope decrement.

Definition 3.6 (Scope Increment and Decrement) The *scope increment* of a stack $\mathfrak{s} \in$ **Stacks** is defined as the stack

$$\mathfrak{s}\,[+\boxdot] \;\triangleq\; \lambda x.\, \begin{cases} \mathfrak{s}(y), & \text{if } x = \boxdot y \\ 0, & \text{otherwise.} \end{cases}$$

The *scope decrement* of a stack $\mathfrak{s} \in$ **Stacks** is defined as the stack

$$\mathfrak{s}\,[-\boxdot] \;\triangleq\; \lambda x.\, \mathfrak{s}(\boxdot x).$$

Moreover, the scope increment $E\,[+\boxdot]$ and the scope decrement $E\,[-\boxdot]$ of an arithmetic expression E are defined as

$$E\,[+\boxdot] \;\triangleq\; \lambda\mathfrak{s}.\, E\,(\mathfrak{s}\,[+\boxdot]) \quad \text{and} \quad E\,[-\boxdot] \;\triangleq\; \lambda\mathfrak{s}.\, E\,(\mathfrak{s}\,[-\boxdot])\,.$$

Example 3.7 Figure 3.1 illustrates the effect of applying scope increment and scope decrement to a stack $\mathfrak{s}$ and a variable x at different scoping levels. In particular, we observe that the value $\mathfrak{s}(x) = 3$ is lost when applying the scope decrement operation. Moreover, for the same stack $\mathfrak{s}$ in Figure 3.1, an evaluation of the expression $(x + 17)\,[-\boxdot]$ in $\mathfrak{s}$ yields

$$\begin{aligned} & (x + 17)\,[-\boxdot]\,(\mathfrak{s}) \\ = {} & (x + 17)\,(\mathfrak{s}\,[-\boxdot]) \\ = {} & \mathfrak{s}\,[-\boxdot]\,(x) + 17 \\ = {} & \mathfrak{s}\,(\boxdot x) + 17 \\ = {} & 11 + 17 \;=\; 28. \end{aligned}$$

Before we consider the operational semantics of $\mathbf{P^2L}$ programs, let us collect a few useful properties of scope increments and scope decrements.

Lemma 3.8 (Properties of Scope Increment and Decrement) The scope increment $\mathfrak{s}\,[+\boxdot]$ and the scope decrement $\mathfrak{s}\,[-\boxdot]$ of a stack $\mathfrak{s} \in \mathbf{Stacks}$ satisfy the following properties:

1. Incrementing and then decrementing the scope does not affect the stack, i.e., $\mathfrak{s}\,[+\boxdot]\,[-\boxdot] = \mathfrak{s}$. The converse does, however, not hold.
2. Both scope increment and scope decrement are monotone, i.e., for all stacks $\mathfrak{s}, \mathfrak{s}' \in \mathbf{Stacks}$, we have:
   - $\forall x\colon \mathfrak{s}\,(x) \leq \mathfrak{s}'\,(x) \quad$ implies $\quad \forall x\colon \mathfrak{s}\,[+\boxdot]\,(x) \leq \mathfrak{s}'\,[+\boxdot]\,(x)$, and
   - $\forall x\colon \mathfrak{s}\,(x) \leq \mathfrak{s}'\,(x) \quad$ implies $\quad \forall x\colon \mathfrak{s}\,[-\boxdot]\,(x) \leq \mathfrak{s}'\,[-\boxdot]\,(x)$.

Proof. To prove the first property, consider the following:

$$\mathfrak{s}\,[+\boxdot]\,[-\boxdot]\,(x) \;=\; \mathfrak{s}\,[+\boxdot]\,(\boxdot x) \;=\; \mathfrak{s}(x).$$

Moreover, monotonicity follows directly from applying the definition of scope increment and scope decrement to a fixed variable $x \in \mathbf{Vars}$. □

The above properties ensure that we never accidentally lose the value of a variable due to scope changes as long as every scope decrement is preceded by a scope increment. This property is guaranteed for all $\mathbf{P^2L}$ programs.

3.1.3 Semantics

We now extend the operational semantics of **PL** programs to $\mathbf{P^2LA}$ programs. To this end, recall from Definition 2.7 the transition system oPL in which states

are of the form $\langle C, \mathfrak{s} \rangle$, where $\mathfrak{s}$ is a stack in **Stacks** and C is either a program or a special symbol term indicating termination. Before we assign semantics to procedure calls, we consider the statements `enter`, `leave`, and `invoke` F.

The statement `enter` causes the execution to enter a new scope, i.e., stack $\mathfrak{s}$ is transformed into the stack $\mathfrak{s}\,[+\boxdot]$. This corresponds to the following rule:

$$\frac{}{\langle \texttt{enter}, \mathfrak{s} \rangle \;\rightsquigarrow\; \langle \mathsf{term}, \mathfrak{s}\,[+\boxdot] \rangle} \text{ enter}$$

Similarly, the statement `leave` is used to leave the current scope, i.e., stack $\mathfrak{s}$ is transformed into the stack $\mathfrak{s}\,[-\boxdot]$. This corresponds to the following rule:

$$\frac{}{\langle \texttt{leave}, \mathfrak{s} \rangle \;\rightsquigarrow\; \langle \mathsf{term}, \mathfrak{s}\,[-\boxdot] \rangle} \text{ leave}$$

Furthermore, the statement `invoke` F executes the body of procedure F:

$$\frac{C = \mathsf{body}\,(F)}{\langle \texttt{invoke}\ F, \mathfrak{s} \rangle \;\rightsquigarrow\; \langle C, \mathfrak{s} \rangle} \text{ invoke}$$

Towards a complete definition of the operational semantics of **P²LA** programs, it remains to assign semantics to procedure calls. Intuitively, the execution of a call statement $x := F(\vec{E})$ consists of five steps:

1. To ensure that (initially) no variable is accessible by the called procedure, we enter a new scope. This is modeled by the auxiliary statement `enter`.
2. The expressions $\vec{E}$ passed to the procedure are evaluated in the original stack. The resulting values are then stored in local variables, which correspond to the procedure's parameters $\mathsf{param}\,(F)$. This is modeled using one assignment for each parameter, where the assigned expressions are evaluated in the previous, i.e., decremented, scope.
3. The body of procedure F, which is determined by its declaration, is executed. This is modeled by the auxiliary statement `invoke` F.
4. After termination of the procedure body, the return value is, by Definition 3.2, stored in variable out. It is then stored in variable x of the original stack. This is modeled by the assignment $\boxdot x := \mathsf{out}$.
5. Finally, we leave the current scope. The values of all local variables are lost in the process. This is modeled by the auxiliary statement `leave`.

Towards a formal definition of the semantics of procedure calls, we implement the above five steps as a **P²LA** program.[2]

[2]Throughout this thesis, we only consider sequential programs. In a concurrent setting, one would additionally have to ensure that the first three steps are performed atomically.

Definition 3.9 (Implementation of Procedure Calls) The *implementation* of a procedure call $x := F(E_1, \ldots, E_n)$, where the parameters of procedure F are $\mathsf{param}(F) = (x_1, \ldots, x_n)$, and the procedure body is $\mathsf{body}(F)$, is given by the **P²LA** program $\mathsf{impl}(x := F(E_1, \ldots, E_n))$ provided below:

`enter;`	(1. enter new scope)
$x_1 := E_1\,[-\boxdot]\;;\ldots;\,x_n := E_n\,[-\boxdot]\;;$	(2. assign parameters)
`invoke` F`;`	(3. invoke procedure)
$\boxdot\, x :=$ `out;`	(4. store return value)
`leave.`	(5. leave scope)

The semantics of a procedure call then amounts to executing its implementation. The operational semantics of **P²L** programs is then formalized as a transition system, which covers all possible program executions.

Definition 3.10 (Operational Semantics of P²L Programs) Let term be a special symbol indicating successful termination and $\langle \mathsf{sink} \rangle$ be a dedicated sink state. The *operational semantics of* **P²L** *programs* is the transition system $\mathrm{oP^2L} \triangleq \langle\, \mathbf{States}, \rightsquigarrow, \mathbf{States} \,\rangle$, where both the set of states and the set of initial states are defined as

$$\mathbf{States} \triangleq \big(\big(\mathbf{P^2LA} \cup \{\, \mathsf{term} \,\} \big) \times \mathbf{Stacks} \big) \cup \big\{\, \langle \mathsf{sink} \rangle \,\big\}\,.$$

Moreover, the execution relation

$$\rightsquigarrow \subseteq \mathbf{States} \times \mathbf{States}$$

is the smallest relation induced by the rules in Figure 3.2 on page 67.

As for **PL** programs, we define the *reachable fragment* of $\mathrm{oP^2L}$ with respect to the set of initial states $\mathbf{I}$ as

$$\mathrm{oP^2L}\,(\mathbf{I}) \triangleq \mathbf{Reach}\,(\langle\, \mathbf{States}, \rightsquigarrow, \mathbf{I} \,\rangle)\,.$$

Finally, the set of executions of program C and stack $\mathfrak{s}$ is defined as

$$\mathbf{Exec}[C](\mathfrak{s}) \triangleq \left\{\, \rho \;\middle|\; \rho \text{ is an execution of } \mathrm{oP^2L}\,(\{\, \langle C, \mathfrak{s} \rangle \,\}) \,\right\}.$$

$$\frac{C = \mathsf{impl}\,(x := \mathrm{Proc}(E_1, \ldots, E_n))}{\langle x := \mathrm{Proc}(E_1, \ldots, E_n),\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C,\, \mathfrak{s}\rangle}\ \text{call}$$

$$\frac{}{\langle \texttt{enter},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\,[+\boxdot]\,\rangle}\ \text{enter} \qquad \frac{}{\langle \texttt{leave},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\,[-\boxdot]\,\rangle}\ \text{leave}$$

$$\frac{C = \mathsf{body}\,(F)}{\langle \texttt{invoke}\ F,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C,\, \mathfrak{s}\rangle}\ \text{invoke}$$

$$\frac{}{\langle \texttt{skip},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\rangle}\ \text{skip} \qquad \frac{E\,(\mathfrak{s}) = v}{\langle x := E,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\,[x/v]\,\rangle}\ \text{assign}$$

$$\frac{\langle C_1,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}'\rangle}{\langle C_1\,;\, C_2,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C_2,\, \mathfrak{s}'\rangle}\ \text{seq1} \qquad \frac{\langle C_1,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C_1',\, \mathfrak{s}'\rangle}{\langle C_1\,;\, C_2,\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C_1'\,;\, C_2,\, \mathfrak{s}'\rangle}\ \text{seq2}$$

$$\frac{B\,(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C_1,\, \mathfrak{s}\rangle}\ \text{if-true}$$

$$\frac{B\,(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C_2,\, \mathfrak{s}\rangle}\ \text{if-false}$$

$$\frac{B\,(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{while}\,(\,B\,)\,\{\,C\,\},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle C\,;\, \texttt{while}\,(\,B\,)\,\{\,C\,\},\, \mathfrak{s}\rangle}\ \text{while-true}$$

$$\frac{B\,(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{while}\,(\,B\,)\,\{\,C\,\},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}\rangle}\ \text{while-false}$$

$$\frac{}{\langle \mathsf{term},\, \mathfrak{s}\rangle \;\rightsquigarrow\; \langle \mathsf{sink}\rangle}\ \text{term} \qquad \frac{}{\langle \mathsf{sink}\rangle \;\rightsquigarrow\; \langle \mathsf{sink}\rangle}\ \text{sink}$$

Figure 3.2: The rules determining the operational semantics of $\mathbf{P^2L}$ programs. Rules depicted in black-75 are identical to the rules for **PL** programs (see Definition 2.7). The program representing the implementation of procedure calls, i.e., $\mathsf{impl}\,(x := F(E_1, \ldots, E_n))$, is found in Definition 3.9.

Example 3.11 Let us consider the execution of a simple non-recursive procedure F, which is given by the declaration

$$F(x, y)\,\{\,\mathsf{out} := x \cdot y\,\}.$$

Moreover, let $\mathfrak{s}$ be a stack with $\mathfrak{s}(y) = 2$ and $\mathfrak{s}(z) = 4$. To run the program

$$x := F(y+1, z)$$

on stack $\mathfrak{s}$, we compute the only execution of the transition system

$$\mathrm{oP}^2\mathrm{L}\left(\{\,\langle x := F(y+1,z),\, \mathfrak{s}\rangle\,\}\right).$$

This involves the following execution steps:

$$\begin{aligned}
&\langle x := F(y+1,z),\, \mathfrak{s}\rangle \\
\rightsquigarrow\ &\langle \mathtt{enter}\,;\, x := (y+1)\,[-\boxdot]\,;\, y := z\,[-\boxdot]\,; && \text{(by call)}\\
&\quad \mathtt{invoke}\ F\,;\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, \mathfrak{s}\rangle \\
\rightsquigarrow\ &\langle x := (y+1)\,[-\boxdot]\,;\, y := z\,[-\boxdot]\,; && \text{(by enter)}\\
&\quad \mathtt{invoke}\ F\,;\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, \mathfrak{s}\,[+\boxdot]\rangle \\
\rightsquigarrow\ &\langle y := z\,[-\boxdot]\,; && \text{(by assign)}\\
&\quad \mathtt{invoke}\ F\,;\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, \mathfrak{s}\,[+\boxdot]\,[x/3]\rangle \\
\rightsquigarrow\ &\langle \mathtt{invoke}\ F\,;\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, && \text{(by assign)}\\
&\quad \mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\rangle \\
\rightsquigarrow\ &\langle \mathsf{out} := x \cdot y\,;\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, && \text{(by invoke)}\\
&\quad \mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\,\rangle \\
\rightsquigarrow\ &\langle\, \boxdot x := \mathsf{out}\,;\, \mathtt{leave}, && \text{(by assign)}\\
&\quad \mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\,[\mathsf{out}/12]\rangle \\
\rightsquigarrow\ &\langle \mathtt{leave}, && \text{(by assign)}\\
&\quad \mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\,[\mathsf{out}/12]\,[\boxdot x/12]\rangle \\
\rightsquigarrow\ &\langle \mathsf{term}, && \text{(by leave scope)}\\
&\quad \mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\,[\mathsf{out}/12]\,[\boxdot x/12]\,[-\boxdot]\rangle \\
\rightsquigarrow\ &\langle \mathsf{sink}\rangle \\
\rightsquigarrow\ &\ldots
\end{aligned}$$

C	$\mathbf{Mod}\,(C)$
`skip`	$\emptyset$
$x := E$	$\{\, x \,\}$
$C_1\,;\, C_2$	$\mathbf{Mod}\,(C_1) \cup \mathbf{Mod}\,(C_2)$
`if` $(\, B \,)$ $\{\, C_1 \,\}$ `else` $\{\, C_2 \,\}$	$\mathbf{Mod}\,(C_1) \cup \mathbf{Mod}\,(C_2)$
`while` $(\, B \,)$ $\{\, C' \,\}$	$\mathbf{Mod}\,(C')$
$x := F(E_1, \ldots, E_n)$	$\{\, x \,\}$

Figure 3.3: Inductive definition of the set $\mathbf{Mod}\,(C)$ of variables modified by a $\mathbf{P^2L}$ program C. The definitions for **PL** programs (in black-75) remain unchanged.

A closer look at the final stack, i.e., the stack reached upon termination indicated by the special symbol term, then reveals that

$$\mathfrak{s}\,[+\boxdot]\,[x/3]\,[y/4]\,[\mathsf{out}/12]\,[\boxdot x/12]\,[-\boxdot] \;=\; \mathfrak{s}\,[x/12]\,.$$

Hence, the procedure call sets x to the product of the supplied parameters.

Although the step-by-step calculations in Example 3.11 are very tedious, they reveal the effects of entering and leaving scopes. In particular, every variable is technically both accessed and modified by a change of scope. After execution of the whole procedure call, however, we obtain the same stack as before in which only the variable storing the procedure's return value is changed. Formally,

Lemma 3.12 For every procedure declaration, the execution

$$\langle x := F(\vec{E}),\, \mathfrak{s} \rangle \rightsquigarrow^* \langle \mathsf{term},\, \mathfrak{s}' \rangle$$

implies that for all variables $y \neq x$, we have $\mathfrak{s}(y) = \mathfrak{s}'(y)$.

Proof. By induction on the number of rules applied until a state indicating termination is reached. □

By Lemma 3.12, x is the only variable that is modified by the procedure call $x := F(\vec{E})$. Hence, we extend the set $\mathbf{Mod}\,(C)$ of variables modified by program C (see Figure 2.10) to cover $\mathbf{P^2L}$ programs as summarized in Figure 3.3.

3.2 Program Verification

Running a **P²L** program with procedures requires a lot of tedious computation steps that distract a programmer from understanding a program's behavior. For instance, the execution of the trivial program in Example 3.11 already consists of nine individual steps. It is thus desirable to reason about the correctness of procedures by means of formal verification on the program structure as we did for **PL** programs in Section 2.3.

Proof rules for recursive procedures are among the first extensions that have been proposed after the publication of the seminal papers on verification by Floyd [Flo67] and Hoare [Hoa69]. For instance, Hoare [Hoa71] introduced a set of inference rules for reasoning about partial correctness of recursive procedures with both call-by-reference and call-by-value parameters as well as local variables. Moreover, proof rules for total correctness based on weakest preconditions have been studied by Hesselink [Hes93].

In our operational semantics, we used auxiliary statements to split the semantics of procedure calls into smaller digestible parts, i.e., entering a new scope, parameter assignment, procedure invocation, assignment of the return value, and leaving the scope again. We could use the same auxiliary statements for program verification. However, this would require us to rewrite **P²L** programs into **P²LA** programs before verification—a step that goes against the spirit of reasoning compositionally on the program structure. Hence, all of the above steps are covered by the weakest precondition transformer for procedure calls.

Since this transformer is rather involved, we derive it step-by-step: We first consider wp transformers for auxiliary statements in **P²LA**. After that, we combine these transformers to define the weakest precondition transformer for procedure calls. In particular, we use the same scoping system as in our operational semantics. This is different from Hoare's approach, which relies on additional proof rules, such as the rule of adaption, for renaming variables if necessary. Alternative sets of rules to deal with scoping exist, e.g., the rules of conjunction, invariance, and specialized substitution rules; see [Apt81] for a survey. To get a feeling for the complexity of the involved rules, we remark that it took almost ten years to understand which sets of auxiliary rules lead to complete inference systems [Old83]. Furthermore, several proposals turned out to be either incomplete or incorrect along the way [Nip02]. To avoid these issues, we prefer an explicit scoping scheme for procedure calls that does not rely on additional rules for on-the-fly variable renaming.

The remainder of this section covers program verification for **P²L** programs. We first lift the relevant notions from Section 2.3, such as Hoare triples and weakest (liberal) preconditions, from **PL** programs to **P²L** programs. After that,

we discuss how valid triples are derived for procedure calls. In particular, we study proof rules for reasoning about partial correctness of recursive procedures.

3.2.1 Hoare triples for P²L

The foundations of program verification by means of deriving valid Hoare triples for **PL** programs have been presented in detail in Section 2.3. Let us briefly discuss how these foundations are adapted to reason about **P²L** programs.

Our *assertion language* remains unchanged, i.e., we allow all predicates in **Pred** to serve as pre- and postconditions. Moreover, Hoare triples for **P²L** programs are defined mutatis mutandis as for **PL** programs (cf. Section 2.3.2). That is, a *Hoare triple* is an expression of the form

$$\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle,$$

where C is a **P²L** program and both P and Q are predicates in **Pred**. To formalize the validity of Hoare triples (cf. Definition 2.22, page 38), it suffices to replace the operational semantics of **PL** programs by the semantics of **P²L** programs. That is, the triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is *valid for partial correctness* if and only if

$$\forall \mathfrak{s} \in \mathbf{Stacks}\,(P)\ \forall \mathfrak{s}' \in \mathbf{Stacks}\colon\ \langle C, \mathfrak{s}\rangle \rightsquigarrow^* \langle \mathsf{term}, \mathfrak{s}'\rangle \ \text{ implies } \ \mathfrak{s}' \models Q.$$

Furthermore, the triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is *valid for total correctness* if and only if it is valid for partial correctness and

$$\forall \mathfrak{s} \in \mathbf{Stacks}\,(P)\,: \qquad \mathbf{Exec}[C](\mathfrak{s}) \subseteq \mathbf{Terminated}.$$

The semantic definition of weakest preconditions—as introduced in Definition 2.24, page 40—remains unchanged: The *weakest precondition WP* of a **P²L** program C with respect to postcondition $Q \in \mathbf{Pred}$ is the predicate

$$WP \ \triangleq\ \sup\left\{\, P \;\middle|\; \langle\, P \,\rangle\ C\ \langle\, Q \,\rangle \text{ is valid for total correctness} \,\right\}.$$

Analogously, the *weakest liberal precondition WLP* of a **P²L** program C with respect to postcondition $Q \in \mathbf{Pred}$ is defined as the predicate

$$WLP \ \triangleq\ \sup\left\{\, P \;\middle|\; \langle\, P \,\rangle\ C\ \langle\, Q \,\rangle \text{ is valid for partial correctness} \,\right\}.$$

While the semantic foundations of triples and weakest preconditions for **P²L** programs are virtually the same as for **PL** programs, developing *proof systems* for deriving valid triples requires more work. To avoid getting trapped in technical details, we consider only a restricted set of recursive procedures.

Assumption 3.13 Throughout this thesis, we restrict ourselves to *direct recursion*. That is, if procedure F calls procedure G, then either G coincides with F or neither G nor any procedure called by G calls F again.

It is possible, however, to incorporate mutually recursive procedures into all definitions presented in the following.

3.2.2 The Weakest Precondition Calculus for P²L

We now extend the weakest precondition calculus by rules for recursive procedures. To this end, recall from Section 2.3.4 the predicate transformer

$$\mathsf{wp}\,[C] : \mathbf{Pred} \to \mathbf{Pred},$$

which determines the weakest precondition of **PL** program C with respect to postcondition Q. To extend this transformer to **P²L** programs, we have to develop a rule to compute the weakest precondition of a procedure call

$$x := F(\vec{E}).$$

As in our operational semantics, the behavior of procedure call $x := F(\vec{E})$ is determined by its *implementation*, i.e., a program that enters a new scope, assigns the values of arguments $\vec{E}$ to local variables, invokes the body of procedure F, assigns the return value to variable x, and leaves the scope again. Hence, the weakest precondition transformer for procedure calls satisfies the equation

$$\mathsf{wp}\left[x := F(\vec{E})\right](Q) \;=\; \mathsf{wp}\left[\mathsf{impl}\left(x := F(\vec{E})\right)\right](Q)\,,$$

where the implementation of procedure calls is found in Definition 3.9, page 66. Notice that the equation's right-hand side is not entirely well-defined, because implementations use the auxiliary statements `enter`, `leave`, and `invoke` F. Hence, the implementation is a **P²LA** program but not a **P²L** program.

In order to develop the weakest precondition transformer for procedure calls, let us—for the moment—allow the **P²LA** auxiliary statements `enter`, `leave`, and `invoke` F in wp: We first consider weakest precondition transformers for these auxiliary statements. Since a procedure implementation is a sequential composition of **P²LA** programs, this enables us to compute the weakest precondition of procedure implementations step-by-step. Finally, we collapse all of these steps into a single rule that determines the weakest precondition transformer for procedure calls without referring to **P²LA** programs.

To capture the weakest precondition of `enter` and `leave`, we first lift the scope increment and decrement operations (cf. Definition 3.6) to predicates.

Definition 3.14 (Scoping for Predicates) The *scope increment* $P\,[+\boxdot]$ and *scope decrement* $P\,[-\boxdot]$ of a predicate $P \in \mathbf{Pred}$ are defined as

$$P\,[+\boxdot] \triangleq \lambda\mathfrak{s}.\, P(\mathfrak{s}\,[+\boxdot]) \qquad \text{and} \qquad P\,[-\boxdot] \triangleq \lambda\mathfrak{s}.\, P(\mathfrak{s}\,[-\boxdot]).$$

Intuitively, changing the scope of a stack corresponds to a simultaneous assignment to every variable in **Vars**. Hence, the rationale underlying the weakest precondition transformer for entering a new scope is similar to the rationale underlying the rule for assignments. That is, to execute the statement `enter` on a stack $\mathfrak{s}$, we apply the following rule of our operational semantics (cf. Figure 3.2):

$$\frac{}{\langle \texttt{enter}, \mathfrak{s} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}\,[+\boxdot] \rangle}\ \text{enter}.$$

Now, if $\mathfrak{s}\,[+\boxdot]$ satisfies postcondition Q, we have

$$\begin{aligned} &\mathfrak{s}\,[+\boxdot] \models Q \\ \text{iff } &Q(\mathfrak{s}\,[+\boxdot]) = \mathsf{true} \\ \text{iff } &Q\,[+\boxdot]\,(\mathfrak{s}) = \mathsf{true} \\ \text{iff } &\mathfrak{s} \models Q\,[+\boxdot]. \end{aligned}$$

Hence, the weakest precondition transformer of `enter` is defined as

$$\mathsf{wp}\,[\texttt{enter}]\,(Q) \triangleq Q\,[+\boxdot].$$

Furthermore, by an analogous argument, the weakest precondition transformer of `leave`, which leaves the current scope, is defined as

$$\mathsf{wp}\,[\texttt{leave}]\,(Q) \triangleq Q\,[-\boxdot].$$

The last $\mathbf{P^2LA}$ statement we have to consider is the procedure invocation `invoke` F. The procedure invocation executes—operationally speaking—the body of F after entering a new scope and passing parameter values. Hence, the weakest precondition of `invoke` F has to coincide with the weakest precondition of $\mathsf{body}\,(F)$. Thus, for every postcondition Q, we have

$$\mathsf{wp}\,[\texttt{invoke}\ F]\,(Q) = \mathsf{wp}\,[\mathsf{body}\,(F)]\,(Q).$$

In fact, defining the weakest precondition transformer of procedure invocations by the above equation is well-defined as long as the body of procedure F does not contain recursive procedure calls (and thus also no invocations of F). However, if the body of F contains recursive calls, then the situation is more involved because the weakest precondition transformer of $\mathsf{body}\,(F)$ depends on

$\mathsf{wp}\,[\texttt{invoke}\ F]$ again. Hence, similarly to our treatment of loops in Section 2.3.4, the weakest precondition of procedure invocations is determined by a fixed point. This time, however, the fixed point is not a predicate in **Pred**, but a *predicate transformer*

$$\mathsf{wp}\,[\texttt{invoke}\ F] : \mathbf{Pred} \to \mathbf{Pred}.$$

In other words, we have to consider *higher-order* fixed points. To formally reason about the desired fixed point, we define an auxiliary transformer

$$\mathsf{wp}_\theta^F\,[C] : \mathbf{Pred} \to \mathbf{Pred},$$

that is defined mutatis mutandis as $\mathsf{wp}\,[C]$ except for the rule for invocations of procedure F. The behavior of $\texttt{invoke}\ F$ is captured by the attached predicate transformer $\theta\colon \mathbf{Pred} \to \mathbf{Pred}$, i.e.,

$$\mathsf{wp}_\theta^F\,[\texttt{invoke}\ F] \ \triangleq\ \theta.$$

The transformer $\mathsf{wp}\,[\texttt{invoke}\ F]$ is then a fixed point of the following equation:

$$\theta \ \triangleq\ \lambda\theta.\,\mathsf{wp}_\theta^F\,[\mathsf{body}\,(F)]\,.$$

Why do suitable fixed points exist? To answer this question, we first lift our ordering on predicates, i.e., implication, to predicate transformers by pointwise application. That is, for two predicate transformers θ, θ', we have

$$\theta \Rightarrow \theta' \quad \text{iff} \quad \forall P \in \mathbf{Pred}\colon \theta(P) \Rightarrow \theta'(P).$$

Next, we observe that the set of all (monotone) predicate transformers together with ordering $\Rightarrow$ forms a complete lattice. Moreover, the higher-order transformer wp_θ^F is monotone for every monotone predicate transformer θ. This is a direct consequence of monotonicity of wp for **PL** programs and monotonicity of wp for `enter` and `leave` (see Lemma 3.8). In particular, the function

$$\lambda\theta.\,\mathsf{wp}_\theta^F\,[\mathsf{body}\,(F)]$$

is monotone. Existence of fixed points is then guaranteed by the Knaster-Tarski fixed point theorem (cf. Theorem A.11). In fact, as we will in detail discuss for procedure calls, we take the *least* fixed point.

Now that we have a weakest precondition transformer for **P²LA** auxiliary statements, we can—in principle—compute the weakest precondition of procedure calls due to the identity

$$\mathsf{wp}\left[x := F(\vec{E})\right](Q) \ =\ \mathsf{wp}\left[\mathsf{impl}\left(x := F(\vec{E})\right)\right](Q)\,.$$

That is, we first replace every procedure call by its implementation and then compute its weakest precondition.

Example 3.15 Recall from Example 3.11 the non-recursive procedure F, which is given by the declaration

$$F(x, y)\,\{\,\mathsf{out} := x \cdot y\,\}.$$

Let us prove that calling F with parameters $y + 3$ and z computes their product and assigns the result to variable x. Hence, our goal is to compute

$$\mathsf{wp}\,[x := F(y + 3, z)]\,(x = (y + 3) \cdot z)\,.$$

This amounts to computing the weakest precondition of the call's implementation. To this end, consider the following computation (read from bottom to top as in Example 2.29):

$$\begin{array}{ll}
\text{// } & \mathsf{true} \\
\text{// } & \Longleftarrow \quad [\![\ \text{elementary predicate logic}\ ]\!] \\
\text{// } & (y + 3) \cdot z = (y + 3) \cdot z \\
\texttt{enter;} & \\
\text{// } & (\boxdot y + 3) \cdot \boxdot z = (\boxdot y + 3) \cdot \boxdot z \\
x := \boxdot y + 3\texttt{;} & \\
\text{// } & x \cdot \boxdot z = (\boxdot y + 3) \cdot \boxdot z \\
y := \boxdot z\texttt{;} & \\
\text{// } & x \cdot y = (\boxdot y + 3) \cdot \boxdot z \\
\texttt{invoke}\ F\texttt{;} & \qquad (\mathsf{wp}\,[\texttt{invoke}\ F] \text{ is discussed below}) \\
\text{// } & \mathsf{out} = (\boxdot y + 3) \cdot \boxdot z \\
\boxdot\, x := \mathsf{out}\texttt{;} & \\
\text{// } & \boxdot x = (\boxdot y + 3) \cdot \boxdot z \\
\texttt{leave} & \\
\text{// } & x = (y + 3) \cdot z
\end{array}$$

The weakest precondition of `invoke` F is the least fixed point of the equation

$$\theta \;=\; \mathsf{wp}^F_\theta\,[\mathsf{body}\,(F)] \;=\; \mathsf{wp}^F_\theta\,[\mathsf{out} := x \cdot y]\,.$$

Since the body of procedure F contains no recursive calls, the least fixed point is given by the weakest precondition of the procedure's body, i.e.,

$$\theta(Q) \;=\; \mathsf{wp}\,[\mathsf{out} := x \cdot y]\,(Q) \;=\; Q\,[\mathsf{out}/x \cdot y]\,.$$

This justifies the tagged step in the above proof, i.e.,

$$\mathsf{wp}\,[\texttt{invoke}\ F]\,(\mathsf{out} = (\boxdot y + 3) \cdot \boxdot z) \;=\; (x \cdot y = (\boxdot y + 3) \cdot \boxdot z)\,.$$

We now develop a dedicated rule to compute the weakest precondition of procedure calls without referring to **P²LA** auxiliary statements. Consider the call $x := F(E_1, \ldots, E_n)$ of a procedure F with parameters are $x_1, \ldots, x_n$. Since procedure calls are semantically equivalent to their implementation, we have

$$\mathsf{wp}\,[x := F(E_1, \ldots, E_n)]\,(Q) \;=\; \mathsf{wp}\,[\mathsf{impl}\,(x := F(E_1, \ldots, E_n))]\,(Q)\,.$$

Let us thus compute the weakest precondition of the procedure's implementation with respect to postcondition Q (read from bottom to top as in Example 2.29):

$$\begin{array}{l}
\mathit{//}\quad \mathsf{wp}\,[\mathsf{body}\,(F)]\,(Q\,[-\boxdot]\,[\boxdot x/\mathsf{out}])\,[x_1/E_1\,[-\boxdot]]\ldots[x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
\texttt{enter} \\
\mathit{//}\quad \mathsf{wp}\,[\mathsf{body}\,(F)]\,(Q\,[-\boxdot]\,[\boxdot x/\mathsf{out}])\,[x_1/E_1\,[-\boxdot]]\ldots[x_n/E_n\,[-\boxdot]] \\
x_1 := E_1\,[-\boxdot]\ ;\ \ldots\ ;\ x_n := E_n\,[-\boxdot]\ ; \\
\mathit{//}\quad \mathsf{wp}\,[\mathsf{body}\,(F)]\,(Q\,[-\boxdot]\,[\boxdot x/\mathsf{out}]\,) \\
\mathsf{body}\,(F)\ ; \\
\mathit{//}\quad Q\,[-\boxdot]\,[\boxdot x/\mathsf{out}] \\
\boxdot\, x := \mathsf{out}; \\
\mathit{//}\quad Q[-\boxdot] \\
\texttt{leave} \\
\mathit{//}\quad Q
\end{array}$$

Hence, the weakest precondition of procedure calls must satisfy the equation

$$\begin{array}{l}
\quad \mathsf{wp}\,[x := F(E_1, \ldots, E_n)]\,(Q) \\
= \mathsf{wp}\,[\mathsf{body}\,(F)]\,(Q\,[-\boxdot]\,[\boxdot x/\mathsf{out}])\,[x_1/E_1\,[-\boxdot]]\ldots[x_n/E_n\,[-\boxdot]]\,[+\boxdot]\,.
\end{array}$$

If the body of procedure F contains no recursive calls, this rule is sufficient. Otherwise, we have to determine a fixed point of the above equation. This fixed point is not a predicate in **Pred**, but a *predicate transformer*

$$\mathsf{wp}\,[x := F(E_1, \ldots, E_n)] : \mathbf{Pred} \to \mathbf{Pred}.$$

Similar to our treatment of the auxiliary statement `invoke` F, we characterize this fixed point by means of a higher-order predicate transformer

$$\mathsf{wp}^F_\theta\,[C] : \mathbf{Pred} \to \mathbf{Pred},$$

C	$\mathsf{wp}_\theta^F\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wp}_\theta^F\,[C_1]\,(\mathsf{wp}_\theta^F\,[C_2]\,(Q))$
`if` $(\,B\,)$ $\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$(B \wedge \mathsf{wp}_\theta^F\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wp}_\theta^F\,[C_2]\,(Q))$
`while` $(\,B\,)$ $\{\,C'\,\}$	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda I.\ (B \wedge \mathsf{wp}_\theta^F\,[C']\,(I)) \vee (\neg B \wedge Q)$
$x := F(E_1,\ldots,E_n)$	$\theta\,(x)(E_1,\ldots,E_n)(Q)$
$x := G(E_1,\ldots,E_m)$	$\mathsf{lfp}\,(\mathfrak{P}_G)\,(x)(E_1,\ldots,E_m)(Q)$

Figure 3.4: Inductive definition of the auxiliary weakest precondition transformer $\mathsf{wp}_\theta^F\,[C]$ that depends on the transformer θ to resolve calls of procedure F. Here, G is a procedure name in $\mathbf{Procs} \setminus \{F\}$. Moreover, the parameters of F and G are given by $\mathsf{param}\,(F) = (x_1,\ldots,x_n)$ and $\mathsf{param}\,(G) = (x_1,\ldots,x_m)$, respectively. Notice that—by Assumption 3.13—procedure G does not call procedure F. Hence, the characteristic function $\mathfrak{P}_G$ (see Figure 3.5) is independent of F.

where the attached transformer θ captures the behavior of calls of procedure F. In contrast to `invoke` F, however, procedure calls—and thus also the attached transformer θ—additionally depend on the variable x storing the return value and the arguments $E_1,\ldots,E_n$ that are passed to the procedure. Thus, if $\mathbf{AE}$ denotes the set of arithmetic expressions, the transformer θ is of the form

$$\theta\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbf{Pred} \to \mathbf{Pred}).$$

Our higher-order predicate transformer then resolves calls of procedure F by referring to the attached transformer θ. That is, we define

$$\mathsf{wp}_\theta^F\,[x := F(E_1,\ldots,E_n)]\,(Q) \;\triangleq\; \theta(x)(E_1,\ldots,E_n)(Q).$$

A complete definition of $\mathsf{wp}_\theta^F\,[C]$ is found in Figure 3.4. The weakest precondition transformer for the procedure call $x := F(E_1,\ldots,E_n)$ is then a fixed point of the call's *characteristic function* $\mathfrak{P}_F$, which is given by

$$\begin{aligned}\mathfrak{P}_F \;\triangleq\; &\lambda\theta\lambda x'\lambda(E_1',\ldots,E_n')\lambda Q'.\\ &\mathsf{wp}_\theta^F\,[\mathsf{body}\,(F)]\,\big(Q'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,[x_1/E_1'\,[-\boxdot]]\ldots[x_n/E_n'\,[-\boxdot]]\,[+\boxdot]\,,\end{aligned}$$

where $n = |\mathsf{param}(F)|$ is the number of parameters of procedure F. Analogously to `invoke` F, the set of all monotone higher-order transformers θ forms a complete lattice, where the ordering is (pointwise) implication $\Rightarrow$.

Lemma 3.16 (Monotonicity) For every $\mathbf{P^2L}$ program C and monotone transformer θ, the predicate transformers $\mathsf{wp}_\theta^F[C]$ and $\mathsf{wp}[C]$ are monotone.

Proof. By induction on the structure of $\mathbf{P^2L}$ programs. □

As a consequence of Lemmas 3.8 and 3.16, the characteristic function $\mathfrak{P}_F$ is itself monotone. The existence of fixed points is then guaranteed by the Knaster-Tarski fixed point theorem (Theorem A.11). Since there might exist multiple fixed points, we take the *least* fixed point. This is justified because we consider *total* correctness: The procedure F without parameters and with procedure body

$$\mathsf{body}(F) \triangleq \mathsf{out} := F()$$

obviously never terminates. Moreover, its characteristic function is given by

$$\mathfrak{P}_F = \lambda\theta\lambda x'\lambda Q'.\,\theta(x')(Q'[-\boxdot][\boxdot x'/\mathsf{out}])[+\boxdot].$$

This characteristic function has infinitely many fixed points θ. However, the only reasonable choice for θ is $\lambda x'\lambda Q'.\,\mathsf{false}$—the least element of the complete lattice of monotone predicate transformers.

This concludes the development of a rule for procedure calls. We summarize the weakest precondition calculus for $\mathbf{P^2L}$ below.

Definition 3.17 (Weakest Precondition Calculus) The *weakest precondition* $\mathsf{wp}[C](Q)$ of $\mathbf{P^2L}$ program C with respect to postcondition $Q \in \mathbf{Pred}$ is defined by structural induction as shown in Figure 3.5, page 79.

All properties of the weakest preconditions calculus for **PL** are preserved when considering $\mathbf{P^2L}$ programs instead. The proofs are similar to the ones for **PL** and thus omitted. We already mentioned that both the transformer wp and the characteristic function $\mathfrak{P}_F$ are monotone. They are even continuous—thus Lemma 2.30 can be lifted to $\mathbf{P^2L}$—if transformers for resolving procedure calls are restricted to continuous functions.

Furthermore, the weakest precondition calculus for $\mathbf{P^2L}$ is sound with respect to the semantic notion of weakest preconditions.

C	$\mathsf{wp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(Q))$
`if` $(\,B\,)\,\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$(B \wedge \mathsf{wp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wp}\,[C_2]\,(Q))$
`while` $(\,B\,)\,\{\,C'\,\}$	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda I.\ (B \wedge \mathsf{wp}\,[C']\,(I)) \vee (\neg B \wedge Q)$
$x := F(E_1, \ldots, E_n)$	$\mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \ldots, E_n)(Q)$, where
$\mathfrak{P}_F \triangleq \lambda\theta\lambda x'\lambda(E'_1, \ldots, E'_n)\lambda Q'.$	
$\mathsf{wp}^F_\theta\,[\mathsf{body}\,(F)]\,(Q'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}])\,[x_1/E'_1\,[-\boxdot]]\ldots[x_n/E'_n\,[-\boxdot]]\,[+\boxdot]$	

Figure 3.5: Inductive definition of the weakest precondition calculus for **P²L** programs. Here, $\mathfrak{P}_F$ is the characteristic function of procedure F with parameters $\mathsf{param}\,(F) = (x_1, \ldots, x_n)$. The auxiliary transformer wp^F_θ is found in Figure 3.4.

Theorem 3.18 (Soundness of the Weakest Precondition Calculus)
Let $\mathsf{wp}\,[C]\,(Q)$ be the predicate computed for **P²L** program C with respect to postcondition $Q \in$ **Pred** by the transformer defined in Figure 3.5. Then

$$\mathsf{wp}\,[C]\,(Q) \;=\; \sup\left\{\, P \;\middle|\; \langle\, P \,\rangle\; C\; \langle\, Q \,\rangle \text{ is valid for total correctness} \right\}.$$

Finally, the proof rules presented for **PL** programs in Theorem 2.32 hold for **P²L** programs as well.

Theorem 3.19 (Compositionality of wp for P²L programs) For every program C in **P²L** and all predicates Q, R, we have:

(a) Strictness: $\mathsf{wp}\,[C]\,(\mathsf{false}) \;=\; \mathsf{false}$.

(b) Conjunction rule: $\mathsf{wp}\,[C]\,(Q \wedge R) \;=\; \mathsf{wp}\,[C]\,(Q) \wedge \mathsf{wp}\,[C]\,(R)$.

(c) Disjunction rule: $\mathsf{wp}\,[C]\,(Q \vee R) \;=\; \mathsf{wp}\,[C]\,(Q) \vee \mathsf{wp}\,[C]\,(R)$.

Proof. By induction on the program structure. □

3.2.3 The Weakest Liberal Precondition Calculus for P²L

Let us briefly consider how **P²L** programs are verified with respect to partial correctness. We thus incorporate procedure calls into the weakest liberal precondition calculus, i.e., the predicate transformer

$$\mathsf{wlp}\,[C] : \mathbf{Pred} \to \mathbf{Pred}$$

A summary of the weakest liberal precondition calculus for **P²L** programs is found in Figure 3.6, page 84. The rules for **PL** programs, which we discussed in detail in Section 2.3.6, remain unchanged. The rationale underlying procedure calls is the same as in the weakest precondition calculus for total correctness that we developed in the previous section. That is, the weakest precondition of a procedure call must satisfy the equation

$$\mathsf{wlp}\,[x := F(E_1, \ldots, E_n)] \;=\; \mathsf{wlp}\,[\mathsf{impl}\,(x := F(E_1, \ldots, E_n))]\,.$$

The solution of this equation is characterized as a fixed point using a higher-order predicate transformer wlp^F_θ, where the attached transformer θ captures the behavior of procedure calls. This transformer is defined analogously to auxiliary calculus for total correctness (cf. Figure 3.4) except for using wlp instead of wp. A formal definition of the rules of wlp^F_θ is found in Figure 3.7, page 85. The weakest liberal precondition transformer of a procedure call $x := F(E_1, \ldots, E_n)$ is then a fixed point of the *liberal characteristic function* $\mathfrak{LP}_F$ that is defined as

$$\begin{aligned}\mathfrak{LP}_F \;\triangleq\; & \lambda\theta\lambda x'\lambda(E'_1, \ldots, E'_n)\lambda Q'.\\ & \mathsf{wlp}^F_\theta\,[\mathsf{body}\,(F)]\,\big(Q'\,[-\boxdot]\,\big[\boxdot x'/\mathsf{out}\big]\big)\,\big[x_1/E'_1\,[-\boxdot]\big]\ldots\big[x_n/E'_n\,[-\boxdot]\big]\,[+\boxdot]\,,\end{aligned}$$

where $n = |\mathsf{param}\,(F)\,|$ is the number of parameters of procedure F. Analogously to weakest preconditions for total correctness, the existence of fixed points is guaranteed (for monotone transformers θ) by Theorem A.11 because the characteristic function $\mathfrak{LP}_F$ is monotone (cf. Lemma 3.16 and Section 3.2.2).

In contrast to total correctness, however, we take the *greatest* fixed point of the liberal characteristic function $\mathfrak{LP}_F$. This is justified because we consider *partial* correctness: The procedure F without parameters and procedure body

$$\mathsf{body}\,(F) \;\triangleq\; \mathsf{out} := F()$$

obviously never terminates. Every pair of pre-and postconditions thus leads to a valid Hoare triple. Hence, the only reasonable choice for our higher-order fixed point is the transformer $\lambda x'\lambda Q'.\,\mathsf{true}$—the greatest element of the complete lattice of monotone predicate transformers.

As for **PL** programs, the weakest liberal precondition calculus coincides with the semantic definition of weakest preconditions for partial correctness introduced in Section 3.2.1.

Theorem 3.20 (Soundness of the Weakest Liberal Precondition Calculus) Let $\mathsf{wlp}\,[C]\,(Q)$ be the predicate computed for $\mathbf{P^2L}$ program C with respect to postcondition Q as defined in Figure 3.6. Then

$$\mathsf{wlp}\,[C]\,(Q) \;=\; \sup\left\{\, P \;\middle|\; \langle\, P \,\rangle\; C\; \langle\, Q \,\rangle \text{ is valid for partial correctness} \,\right\}.$$

Furthermore, the proof rules for weakest liberal preconditions studied in Section 2.3.7 also apply to the weakest liberal precondition calculus for $\mathbf{P^2L}$.

Theorem 3.21 (Proof Rules for $\mathbf{P^2L}$ programs) For every $\mathbf{P^2L}$ program C and all predicates $Q, R \in \mathbf{Pred}$, we have:

(a) Strictness: $\mathsf{wlp}\,[C]\,(\mathsf{true}) \;=\; \mathsf{true}$.

(b) Conjunction rule: $\mathsf{wlp}\,[C]\,(Q \wedge R) \;=\; \mathsf{wlp}\,[C]\,(Q) \wedge \mathsf{wlp}\,[C]\,(R)$.

(c) Disjunction rule: $\mathsf{wlp}\,[C]\,(Q \vee R) \;=\; \mathsf{wlp}\,[C]\,(Q) \vee \mathsf{wlp}\,[C]\,(R)$.

(d) Invariance: If $\mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(Q) = \emptyset$ then $Q \Rightarrow \mathsf{wlp}\,[C]\,(Q)$.

Finally, reasoning about partial correctness admits a simple, invariant-based proof rule for reasoning about recursive procedures.

Theorem 3.22 (Proof rule for recursive procedures) Let $\mathfrak{LP}_F$ be the liberal characteristic function of a procedure F. Then, for all transformers θ:

$$\theta \;\Rightarrow\; \mathfrak{LP}_F(\theta) \quad \text{implies} \quad \theta(x)(\vec{E}) \;\Rightarrow\; \mathsf{wlp}\left[x := F(\vec{E})\right].$$

Proof. Assume $\theta \Rightarrow \mathfrak{LP}_F(\theta)$. By Park's lemma (Corollary A.12), we have

$$\theta \;\Rightarrow\; \mathsf{gfp}\,(\mathfrak{LP}_F).$$

By Figure 3.6, this means that

$$\theta(x)(\vec{E}) \;\Rightarrow\; \mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(\vec{E}) \;=\; \mathsf{wlp}\left[x := F(\vec{E})\right]. \qquad \square$$

Theorem 3.22 enables reasoning about recursive procedures by providing a *higher-order invariant* ρ . In general, finding a suitable ρ is very difficult because we have to capture the behavior of a procedure with respect to all possible postconditions. To avoid this daring task, it is convenient to first fix a set of postconditions, say $\mathbf{Pred}' \subseteq \mathbf{Pred}$, and then construct a transformer that

captures only the behavior for the chosen postconditions. That is, we construct a transformer ρ' such that

$$\forall x\ \forall \vec{E}\ \forall Q \in \mathbf{Pred}' : \qquad \rho'(x)(\vec{E})(Q) \Rightarrow \left(\mathfrak{LP}_F(\rho')\right)(x)(\vec{E})(Q).$$

The predicate transformer ρ' is then extended to an invariant ρ in the sense of Theorem 3.22 by choosing the exact fixed point for all other postconditions:

$$\rho(x)(\vec{E})(Q) \triangleq \begin{cases} \rho'(x)(\vec{E})(Q), & \text{if } Q \in \mathbf{Pred}' \\ \mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(\vec{E})(Q), & \text{otherwise.} \end{cases}$$

Of course, this approach is only worthwhile if the chosen set of postconditions $\mathbf{Pred}'$ covers all postconditions encountered in procedure calls when checking whether ρ' implies $\mathfrak{LP}_F(\rho')$.

To conclude this chapter, let us apply the above proof rule to reason about a recursive $\mathbf{P^2L}$ program.

Example 3.23 Recall from Example 3.3, the following declaration of a recursive procedure *mult* with two parameters:

```
mult(x, y) {
    if (y > 0) {
        z := mult(x, y − 1);
        out := x + z
    } else {
        out := 0
    }
}
```

Let us verify that this procedure returns the product of its parameters if parameter y is initially greater than or equal to zero. Hence, our goal is to determine (an approximation of) the weakest liberal precondition

$$\mathsf{wlp}\,[z := \mathit{mult}(x, y)]\,(y \geq 0 \wedge z = x \cdot y)\,.$$

To this end, we propose the higher-order invariant

$$\theta \triangleq \lambda x' \lambda (E, E') \lambda Q.\ \left(E' > 0 \wedge Q\left[x'/E \cdot E'\right]\right) \vee \left(E' \leq 0 \wedge Q\left[x'/0\right]\right).$$

Assuming θ is a suitable invariant in the sense of Theorem 3.22, we obtain

$$\begin{aligned}
&\mathsf{wlp}\,[z := \mathit{mult}(x,y)]\,(y \geq 0 \wedge z = x \cdot y)\\
\Leftarrow\ &\theta(z)(x,y)(y \geq 0 \wedge z = x \cdot y)\\
=\ &(y > 0 \wedge y \geq 0 \wedge x = x \cdot y) \vee (y \leq 0 \wedge y \geq 0 \wedge 0 = x \cdot y)\\
=\ &y \geq 0 \wedge x = x \cdot y.
\end{aligned}$$

In words, whenever y is initially greater or equal than zero, procedure *mult* will compute the product of x and y. It then remains to verify that θ is indeed a correct invariant in the sense of Theorem 3.22. To this end, consider the proof further below (cf. Example 2.29 for an explanation of program annotations). We performed a few simplifications along the way to improve readability. Furthermore, according to the definition of liberal characteristic functions, we have to apply a few transformations before and after computing wlp of the procedure body to account for scoping, return value, and parameters. These transformations are performed when leaving and entering the procedure body, respectively.

$// \quad (E' > 0 \wedge Q\,[x'/E \cdot E']) \vee (E' < 0 \wedge Q\,[x'/0])$
mult$(x,y)\,\{$
  $// \quad (y > 0 \wedge Q\,[-\boxdot]\,[\boxdot x'/x \cdot y]\,) \vee (y < 0 \wedge Q\,[-\boxdot]\,[\boxdot x'/0])$
  `if` $(\,y > 0\,)\,\{$
    $// \quad (y - 1 > 0 \wedge Q\,[-\boxdot]\,[\boxdot x'/x \cdot y])$
    $// \quad\quad \vee (y - 1 \leq 0 \wedge Q\,[-\boxdot]\,[\boxdot x'/x])$
    $z := \mathit{mult}(x, y-1)\,;$
    $// \quad Q\,[-\boxdot]\,[\boxdot x'/x + z]$
    $\mathsf{out} := x + z$
    $// \quad Q\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]$
  $\}$ `else` $\{$
    $// \quad Q\,[-\boxdot]\,[\boxdot x'/0]$
    $\mathsf{out} := 0$
    $// \quad Q\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]$
  $\}$ $// \quad Q[-\boxdot]\,[\boxdot x'/\mathsf{out}]$
$\}$ $// \quad Q$

C	$\mathsf{wlp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wlp}\,[C_1]\,(\mathsf{wlp}\,[C_2]\,(Q))$
`if` $(\,B\,)$ $\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$(B \wedge \mathsf{wlp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wlp}\,[C_2]\,(Q))$
`while` $(\,B\,)$ $\{\,C'\,\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where
	$\mathfrak{LW} \triangleq \lambda I.\ (B \wedge \mathsf{wlp}\,[C']\,(I)) \vee (\neg B \wedge Q)$
$x := F(E_1, \ldots, E_n)$	$\mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(E_1, \ldots, E_n)(Q)$, where
$\mathfrak{LP}_F \triangleq \lambda\theta\lambda x'\lambda(E'_1, \ldots, E'_n)\lambda Q'.$	
$\mathsf{wlp}^F_\theta\,[\mathsf{body}\,(F)]\,\big(Q'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,[x_1/E'_1\,[-\boxdot]]\ldots[x_n/E'_n\,[-\boxdot]]\,[+\boxdot]$	

Figure 3.6: Inductive definition of the weakest liberal precondition calculus for $\mathbf{P^2L}$ programs. Here, $\mathfrak{LP}_F$ is the liberal characteristic function of procedure F.

C	$\mathsf{wlp}_\theta^F\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$C_1\,;\,C_2$	$\mathsf{wlp}_\theta^F\,[C_1]\left(\mathsf{wlp}_\theta^F\,[C_2]\,(Q)\right)$
`if` $(\,B\,)\,\{\,C_1\,\}$ `else` $\{\,C_2\,\}$	$\left(B \wedge \mathsf{wlp}_\theta^F\,[C_1]\,(Q)\right) \vee \left(\neg B \wedge \mathsf{wlp}_\theta^F\,[C_2]\,(Q)\right)$
`while` $(\,B\,)\,\{\,C'\,\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where $\mathfrak{LW} \triangleq \lambda I.\ \left(B \wedge \mathsf{wlp}_\theta^F\,[C']\,(I)\right) \vee (\neg B \wedge Q)$
$x := F(E_1, \ldots, E_n)$	$\theta\ (x)(E_1, \ldots, E_n)(Q)$
$x := G(E_1, \ldots, E_m)$	$\mathsf{gfp}\,(\mathfrak{LP}_G)\,(x)(E_1, \ldots, E_m)(Q)$

Figure 3.7: Inductive definition of the auxiliary weakest liberal precondition transformer $\mathsf{wlp}_\theta^F\,[C]$. Here, G is a procedure name in $\mathbf{Procs} \setminus \{F\}$. Moreover, we assume that the parameters of procedures F and G are $\mathsf{param}\,(F) = (x_1, \ldots, x_n)$ and $\mathsf{param}\,(G) = (x_1, \ldots, x_m)$, respectively. Notice that—by Assumption 3.13—procedure G does not call procedure F.

Pointers constitute an essential concept in computer programming that is "key to the representation of complex structures" [Knu97, p. 214]. Examples of such structures include linked lists, trees, graphs, etc. As most efficient algorithms rely on efficient data structures, this means that pointers are at the foundation of both programming and algorithm design.

Theoretical models of computation that support pointers can be traced back to Kolmogorov [Kol53]. He proposed a model which is similar to Turing machines but operates on graphs instead of an infinite tape. However, "contrary to Turing's tape whose topology is fixed, Kolmogorov's "tape" is reconfigurable" [BG03, p. 10]. Kolmogorov's "pointer machine" reveals another important feature of data structures based on pointers: Their structure can be mutated at runtime; in particular, they can grow or shrink in size. Data structures with this feature are thus called *dynamic data structures*.

The first implementation of pointers in a programming language is due to Lawson [Law67]. His design principles are still present in modern languages, such as C, C++, and (to a lesser extent) Java: The available memory is split into a *stack* and a *heap*.[1] The *stack* takes care of the assignment of values to local variables—just like the notion of stacks used in previous chapters; see Definition 2.5. It is managed by the program environment. For example, Figure 4.1, page 88, depicts a stack which assigns the values 12, 9, and 4 to the variables x, y, and z, respectively. In contrast, the *heap* is a large array of memory that is shared amongst programs. It is under direct control of the programmer. Every element of the heap is a single value that is referenced by its *address*, i.e., its position in the array. Since multiple programs may wish to operate on the heap, each program first has to *allocate* addresses such that it is safe to access and manipulate them. Analogously, addresses that are not needed anymore must be *deallocated* before they are available to other programs again. Consequently, the heap accessible by a given program does not necessarily consist of a consecutive block of addresses. For instance, the heap in Figure 4.1

[1] Lawson originally called the heap a table, but the term heap has become prevalent (cf. [SGG10]).

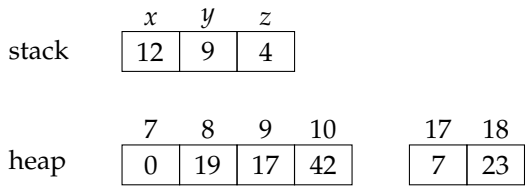
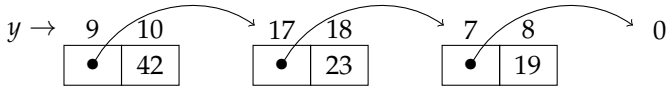


Figure 4.1: A stack and a heap of a program.

contains 6 addresses—7, 8, 9, 10, 17, and 18—that are allocated. The value at each address is provided in the memory cell below the address. For example, the value at address 18 is 23. A *pointer variable*, say x, is a variable that stores an address. The actual value which the variable "points to" is then obtained by *dereferencing* x, i.e., determining the value at the stored address; we denote this value by $<x>$. If we interpret variable y in Figure 4.1 as a pointer variable, for instance, then the value given by $<y>$ is the value at address 9, i.e., 17. Since 17 is an allocated address itself, we may also dereference it, which leads to the value 7. This way, the heap in Figure 4.1 encodes a singly-linked list: Variable y points to the head of the list. The list is terminated by a *null reference*, i.e., a pointer to 0, which is by definition not a valid address. Moreover, every list element stores a value at the address that is its immediate successor. An illustration of the encoded list is found below:



While pointers are important to implement dynamic data structures, the above example also hints at a potential risk: If we attempt to traverse the list by iteratively dereferencing addresses, we might eventually dereference 0 (or another address that is not allocated). In this case, we encounter a *memory fault*; its consequences are implementation-specific, but the program will most likely crash. This behavior is different from **PL** and **P²L** programs considered in previous chapters: While a **P²L** program may produce the wrong result or not terminate at all, it is never the case that a **P²L** program suddenly aborts execution. Even worse, memory faults due to incorrect pointer usage are a common source of errors. Considering just properly handling the null reference, Hoare [Hoa09] stated that "this has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and

damage in the last forty years". It is thus desirable to *reason* about *pointer programs*, i.e., programs that allow to access and manipulate the heap via pointers. At least, one would like to formally guarantee the absence of memory faults. Although there have been early attempts to prove pointer programs correct, e.g., by Burstall [Bur72], developing a reasonable Floyd-Hoare style verification system for pointer programs was long regarded as an unsolved challenge for program verification. The solution to this challenge was developed in a series of papers by O'Hearn, Reynolds, Yang, and others [IO01; ORY01; Rey02; YO02; OHe19]: *Separation logic* is an extension of Hoare logic that allows to write elegant proofs about pointer programs. In particular, separation logic champions *compositional reasoning*. As a consequence, it is suitable for *automated* verification, which sparked the development of various program analysis and verification tools based on separation logic, e.g., [BCO05a; Yan+08; DP08; VLC10; BCI11; DPV11; Cal+11; CD11; Chi+12].

In this chapter, we give an introduction to reasoning about pointer programs with separation logic. We first endow $\mathbf{P^2L}$ with statements to access and mutate the heap. This results in the *procedural pointer programming language*, $\mathbf{P^3L}$ for short, which is presented in Section 4.1. In Section 4.2, we briefly discuss the challenges for program verification when dealing with pointer programs. The assertion language of separation logic is studied in Section 4.3. Finally, in Section 4.4, we use separation logic to perform Floyd-Hoare style verification of $\mathbf{P^3L}$ programs by means of computing weakest preconditions.

We build upon the foundations of separation logic presented in this chapter in the remainder of this thesis. In particular, we consider an extension of separation logic for reasoning about *probabilistic* pointer programs in Part II. Moreover, we turn to *automated* reasoning about pointer programs in Part III.

4.1 The Procedural Pointer Programming Language

In order to support dynamic data structures in our programming language, we extend the procedural programming language $\mathbf{P^2L}$ by statements to mutate the heap via pointers. This results in the procedural *pointer* programming language, $\mathbf{P^3L}$ for short. After introducing the syntax of $\mathbf{P^3L}$, we formalize the notion of heaps. Finally, we present the operational semantics of $\mathbf{P^3L}$ programs.

Support for pointers in $\mathbf{P^3L}$ is inspired by the C programming language [KR88]. Let us briefly elaborate on this design choice. The underlying memory model is close to the original proposal by Lawson [Law67]: It is rather low-level, it supports accessing single addresses, and it allows for pointer arithmetic. Our choice of this memory model is mainly motivated by its (conceptual) simplicity: To understand the exact semantics of pointer manipulations, we believe it is best to first consider a model that is close to an

actual implementation. Furthermore, other memory models, e.g., the Java object model [AGH05], can be implemented in terms of the low-level **P³L** statements.

In Part III, we consider a more high-level memory model in the context of automated program verification.

4.1.1 Syntax

We extend the syntax of **P²L** by statements for *allocation* of new addresses, *deallocation* of an address, and *pointer dereferences*. By $<E>$ we denote the value at the address given by an evaluation of expression E. We require that both arithmetic and Boolean expressions are evaluated over variables, i.e., *expressions depend on the stack only*. Furthermore, we use pointer dereferences in two idiomatic forms: First, a *lookup* is an assignment that dereferences a given address and stores the result in a variable. Second, a *mutation* sets the value at a given address to the value obtained from evaluating an expression. Other pointer manipulations can be implemented using multiple lookup and mutation statements.

Definition 4.1 (Syntax of the Procedural Pointer Programming Language) The set of programs written in the *Procedural Pointer Programming Language with Auxiliaries*, denoted **P³LA**, is given by the context-free grammar below (**P²LA** statements are displayed in black-75):

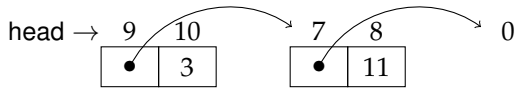
$$
\begin{array}{rll}
C \;\rightarrow & x := \texttt{alloc}(\vec{E}) & \text{(allocation)} \\
\mid & \texttt{free}(E) & \text{(deallocation)} \\
\mid & x := <E> & \text{(lookup)} \\
\mid & <E> := E' & \text{(mutation)} \\
\\
\mid & \texttt{skip} & \text{(effectless program)} \\
\mid & x := E & \text{(assignment)} \\
\mid & C\,;\,C & \text{(sequential composition)} \\
\mid & \texttt{if}\ (\,B\,)\ \{\,C\,\}\ \texttt{else}\ \{\,C\,\} & \text{(conditional choice)} \\
\mid & \texttt{while}\,(\,B\,)\,\{\,C\,\} & \text{(loop)} \\
\mid & x := F(\vec{E}) & \text{(procedure call)} \\
\\
\mid & \texttt{enter} & \text{(enter scope)} \\
\mid & \texttt{invoke}\ F & \text{(invoke procedure)} \\
\mid & \texttt{leave}, & \text{(leave scope)}
\end{array}
$$

where $x \in$ **Vars** is a variable and $F \in$ **Procs** is a procedure name. Moreover, $\vec{E}$ is a sequence of arithmetic expressions, E and E' are arithmetic expressions, and B is a Boolean expression. All expressions are over variables, i.e., arithmetic expressions are of the form $E\colon$ **Stacks** $\to \mathbb{Z}$ and Boolean expressions are of the form $B\colon$ **Stacks** $\to \mathbb{T}$, respectively.

Moreover, the set of programs in the *Procedural Pointer Programming Language*, denoted $\mathbf{P^3L}$, consists of all $\mathbf{P^3LA}$ programs which contain neither `enter` nor `invoke` F nor `leave` statements.

Intuitively, the statement $x := \texttt{alloc}(\vec{E})$ allocates $|\vec{E}|$ fresh addresses whose initial values are given by the arithmetic expressions $\vec{E}$. Conversely, the statement $\texttt{free}(E)$ deallocates the address given by E. The lookup statement $x := <E>$ stores the value at address E in variable x. Moreover, the mutation statement $<E> := E'$ sets the value at address E to the value given by expression E'.

Example 4.2 Assume that a variable head initially points to the head of a singly-linked list, where every list element consists of two addresses: The first address stores the next list element's address and the second address stores a value. The last list element is indicated by a *null pointer*, i.e., the last list element stores 0 as the address of the next element. An example of such a list is depicted below:



Now, consider the following $\mathbf{P^3L}$ program C_{sum} operating on lists:

```
x := 0;
while ( head ≠ 0 ) {
    y := <head + 1>;
    x := x + y;
    z := <head>;
    free(head);
    free(head + 1);
    head := z
}
```

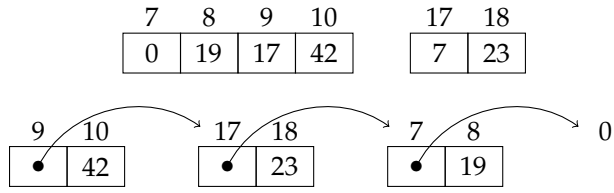


Figure 4.2: Two graphical representations of the same heap.

Program C_{sum} traverses the list until the end of the list is reached, i.e., until $\mathsf{head} = 0$ holds, and computes the sum of all values stored in the list. The traversal is performed by setting variable head to its successor in the list, i.e., $<\mathsf{head}>$. Moreover, after determining the value of a list element, which is given by $<\mathsf{head}+1>$, the list element is deleted. This means that both the address head and the address $\mathsf{head}+1$ are deallocated. Hence, for the heap from above, program C_{sum} terminates with an empty heap, where head is equal to zero and x is equal to 14.

4.1.2 Heaps

The heaps considered so far were represented by pictures. For example, Figure 4.2 depicts two graphical representations of the same heap. The topmost picture represents the intuitive notion of heaps: A heap consists of a bunch of "memory cells", where each memory cell has an address and contains a value. The picture below stresses how linked data structures are constructed. We will continue to use such pictures to illustrate heaps. To assign formal semantics to **P^{3}L** programs, however, we need a precise definition of heaps.

Mathematically speaking, a heap is a function from a finite set of *allocated addresses*, which we represent by natural numbers, to *values*, which we represent by integers. Although the number of allocated addresses must be finite, we impose no further restrictions on the available amount of memory. Hence, it is always possible to allocate additional addresses. Furthermore, the value 0 is not admitted as an address, because it serves as a null reference.

Definition 4.3 (Heap [Rey02; 1]) The set of *heaps* is defined as

$$\mathbf{Heaps} \triangleq \{\, \mathfrak{h} \mid \mathfrak{h} \colon N \to \mathbb{Z},\ N \subseteq \mathbb{N}_{>0},\ |N| < \infty \,\}.$$

Moreover, for a given heap $\mathfrak{h} \colon N \to \mathbb{Z}$, we denote by $\mathsf{dom}(\mathfrak{h})$ its *domain* N.

Example 4.4 The heap illustrated in Figure 4.2 is given by the function

$$\mathfrak{h}\colon \underbrace{\{\,7,8,9,10,17,18\,\}}_{=\,\mathsf{dom}(\mathfrak{h})} \to \mathbb{Z}, \text{ where}$$
$$\mathfrak{h}(7) = 0 \quad \mathfrak{h}(8) = 19 \quad \mathfrak{h}(9) = 17$$
$$\mathfrak{h}(10) = 42 \quad \mathfrak{h}(17) = 7 \quad \mathfrak{h}(18) = 23.$$

Let us consider a few common heaps. By $\mathfrak{h}_\emptyset$, we denote the *empty heap* with $\mathsf{dom}(\mathfrak{h}_\emptyset) = \emptyset$, i.e., no address is allocated in $\mathfrak{h}_\emptyset$. We write $\{\, u :: v_1, \ldots, v_n \,\}$ as a shorthand for a *consecutive block of addresses* with values $v_1, \ldots, v_n$ starting at address u. Formally, $\{\, u :: v_1, \ldots, v_n \,\}$ is defined as the heap $\mathfrak{h}$ given by

$$\begin{aligned} \mathsf{dom}(\mathfrak{h}) &= \{\, u, u+1, \ldots, u+n-1 \,\}, \text{ and} \\ \mathfrak{h}(u+k) &= v_{k+1} \qquad \forall k \in \{\, 0, 1, \ldots, n-1 \,\}. \end{aligned}$$

It is convenient to construct larger heaps from existing ones. To this end, we define a union operation for heaps that is well-defined as long as the domains of the combined heaps do not overlap.

Definition 4.5 (Disjoint Union [IO01]) The heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are *disjoint*, denoted $\mathfrak{h}_1 \# \mathfrak{h}_2$, if and only if $\mathsf{dom}(\mathfrak{h}_1) \cap \mathsf{dom}(\mathfrak{h}_2) = \emptyset$.

For disjoint heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ with domains $\mathsf{dom}(\mathfrak{h}_1) = N_1$ and $\mathsf{dom}(\mathfrak{h}_2) = N_2$, the *disjoint union* of $\mathfrak{h}_1$ and $\mathfrak{h}_2$ is defined as

$$\mathfrak{h}_1 \uplus \mathfrak{h}_2 \colon N_1 \cup N_2 \to \mathbb{Z}, \quad (\mathfrak{h}_1 \uplus \mathfrak{h}_2)(\ell) \triangleq \begin{cases} \mathfrak{h}_1(\ell), & \text{if } \ell \in N_1 \\ \mathfrak{h}_2(\ell), & \text{if } \ell \in N_2. \end{cases}$$

Otherwise, i.e., if $\mathfrak{h}_1$ and $\mathfrak{h}_2$ are not disjoint, $\mathfrak{h}_1 \uplus \mathfrak{h}_2$ is undefined.

For example, the heap $\mathfrak{h}$ depicted in Figure 4.2 can be concisely defined as

$$\mathfrak{h} \triangleq \{\, 7 :: 0, 19, 17, 42 \,\} \uplus \{\, 17 :: 7, 23 \,\}.$$

The disjoint union above *partitions* the *addresses*, i.e., the memory cells depicted in Figure 4.2, of heap $\mathfrak{h}$ into two parts. An overlap between the domains of the resulting two heaps is thus not permitted—even if the overlapping addresses are mapped to the same value. The *values* in one part of $\mathfrak{h}$ may, however, coincide with addresses of the other part. In the alternative graphical representation in Figure 4.2 (lowermost), this corresponds to a partitioning of the *edges*. The disjoint union of heaps is quite well-behaved as the set **Heaps** of all heaps with the disjoint union operator $\uplus$ and the empty heap $\mathfrak{h}_\emptyset$ as a neutral element forms a (partial) commutative monoid [COY07, Sec. 2]. In particular, this means that heaps can be composed in any order and without brackets.

The disjoint union also induces a useful notion of *heap inclusion*:

$$\mathfrak{h}_1 \subseteq \mathfrak{h}_2 \quad \text{iff} \quad \exists \mathfrak{h}' \colon \mathfrak{h}_1 \uplus \mathfrak{h}' = \mathfrak{h}_2.$$

In other words, a heap $\mathfrak{h}_1$ is included in another heap $\mathfrak{h}_2$ if all of its addresses are mapped to the same value by the heap $\mathfrak{h}_2$. To conclude this section, we introduce notation for updating the value at a given address in a heap. This is similar to the substitution mechanism for stacks introduced in Definition 2.6.

Definition 4.6 (Heap Update [Rey02, p. 3]) The *update* of the value at address u by an integer value v in a heap $\mathfrak{h} \in \mathbf{Heaps}$ is defined as

$$\mathfrak{h}\,[u/v] \triangleq \lambda \ell. \begin{cases} v, & \text{if } \ell \in \mathsf{dom}(\mathfrak{h}) \text{ and } \ell = u \\ \mathfrak{h}(\ell), & \text{if } \ell \in \mathsf{dom}(\mathfrak{h}) \text{ and } \ell \neq u \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

4.1.3 Semantics

Similar to **PL** and **P²L** programs, the operational semantics of **P³L** (and **P³LA**) programs is defined as a transition system. However, there are subtle differences between the transition system for **P²L** and the transition system for **P³L**. A technical difference is that every state has to be equipped with a heap in addition to a stack. We thus consider stack-heap pairs (cf. [IO01, Section 3.1]):

Definition 4.7 (Stack-Heap Pair) The set of *stack-heap pairs* is defined as

$$\mathbf{SHPairs} \triangleq \{\, (\mathfrak{s}, \mathfrak{h}) \mid \mathfrak{s} \in \mathbf{Stacks},\ \mathfrak{h} \in \mathbf{Heaps} \,\}.$$

Program executions may *fail*, e.g., due to dereferencing a null pointer. We use a dedicated sink state $\langle \mathsf{fault} \rangle$ which indicates that program execution has been aborted due to a failure; $\langle \mathsf{fault} \rangle$ thus represents *unsuccessful* termination. The set of states of the operational semantics of **P³L** programs is then defined as

$$\mathbf{States} \triangleq \big((\mathbf{P^3LA} \cup \{\, \mathsf{term} \,\}) \times \mathbf{SHPairs} \big) \cup \big\{\, \langle \mathsf{sink} \rangle, \langle \mathsf{fault} \rangle \,\big\}.$$

To define the execution relation of the transition system for **P³L** programs, i.e.,

$$\rightsquigarrow \subseteq \mathbf{States} \times \mathbf{States},$$

we rely on the fact that all pointer dereferences are explicitly captured by the lookup statement statement $x := \langle E \rangle$ and the mutation statement $\langle E \rangle := E'$. This is guaranteed because expressions depend on the stack only.

Let us first consider the rules for **P³L** statements that actually access the heap. These rules are—with minor modifications to match our notation—identical to the operational semantics proposed by Yang and O'Hearn [YO02].

The statement $x := \texttt{alloc}(E_1, \ldots, E_n)$ allocates a consecutive block of n addresses and stores the first allocated address, say u, in variable x. The allocated addresses are initialized with the values given by the arithmetic expressions $E_1, \ldots, E_n$. Hence, if we are initially given a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, the stack is updated to $\mathfrak{s}[x/u]$. Moreover, the heap is extended to

$$\mathfrak{h} \uplus \{ u :: E_1(\mathfrak{s}), \ldots, E_n(\mathfrak{s}) \}.$$

How is the newly allocated address u determined? We first notice that allocation never fails because we assume an infinite amount of available memory (every natural number except for 0 is a potential address). The exact choice of address u depends on the implementation of the memory allocator which may vary for different machines. To abstract from these implementation details, we select address u *nondeterministically*. In contrast to our previous semantics, a state might admit multiple executions. In fact, since we may choose any natural number (except for finitely many already allocated ones) for address u, there are *countably infinite* many executions for every allocation statement. Formally, the rule for allocations is defined as follows:

$$\frac{u, u+1, \ldots, u+n-1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(\mathfrak{h}) \qquad E_1(\mathfrak{s}) = v_1, \ldots, E_n(\mathfrak{s}) = v_n}{\langle x := \texttt{alloc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[x/u], \mathfrak{h} \uplus \{ u :: v_1, \ldots, v_n \} \rangle} \text{ alloc}$$

To improve readability, we omit brackets around stack-heap pairs in all rules.

The statement $\texttt{free}(E)$ disposes the memory cell at the address given by expression E if it is present. This is reflected by the following rule:

$$\frac{E(\mathfrak{s}) = u}{\langle \texttt{free}(E), \mathfrak{s}, \mathfrak{h} \uplus \{ u :: v \} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle} \text{ free}$$

If expression E does not evaluate to an allocated address, however, we encounter a memory error and the execution fails:

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(h)}{\langle \texttt{free}(E), \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{fault} \rangle} \text{ free-fail}$$

The lookup statement $x := <E>$ determines the value at address E and stores it in variable x. Hence, only the stack is modified. Again, execution fails if E does not evaluate to an allocated address. The semantics of lookups is thus determined by the following two rules:

$$\frac{E(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad \mathfrak{h}(u) = v}{\langle x := <E>, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \rangle} \text{ lookup}$$

$$\frac{E\,(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\langle x := <E>,\, \mathfrak{s}, \mathfrak{h}\rangle \;\rightsquigarrow\; \langle \mathsf{fault}\rangle} \text{ lookup-fail}$$

Finally, the mutation statement $<E> := E'$ updates the value at address E to the value of expression E' if E is evaluated to an allocated address; otherwise, it fails. Hence, only the heap is modified. The semantics of mutations is thus formalized by the following two rules:

$$\frac{E\,(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad E'\,(\mathfrak{s}) = v}{\langle <E> := E',\, \mathfrak{s}, \mathfrak{h}\rangle \;\rightsquigarrow\; \langle \mathsf{term},\, \mathfrak{s}, \mathfrak{h}\,[u/v]\,\rangle} \text{ mutation}$$

$$\frac{E\,(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\langle <E> := E',\, \mathfrak{s}, \mathfrak{h}\rangle \;\rightsquigarrow\; \langle \mathsf{fault}\rangle} \text{ mutation-fail}$$

Towards a complete definition of the operational semantics of **P³LA** programs, we also have to provide rules for statements that do not access the heap, i.e., all **P²LA** statements. The remaining rules are found in Figure 4.3, page 98. Except for the additional heap component, these rules are identical to the rules that determine the semantics of **P²LA** programs (cf. Definition 3.10). However, we added a third rule for sequential composition to account for previous memory failures. In this case, program execution is aborted:

$$\frac{\langle C_1,\, \mathfrak{s}, \mathfrak{h}\rangle \;\rightsquigarrow\; \langle \mathsf{fault}\rangle}{\langle C_1\,;\, C_2,\, \mathfrak{s}, \mathfrak{h}\rangle \;\rightsquigarrow\; \langle \mathsf{fault}\rangle} \text{ seq3}$$

Furthermore, to ensure that every terminating execution eventually ends up in the sink state, we added a rule to move from failure states to $\langle \mathsf{sink}\rangle$.

Let us summarize the formal definition of the transition system that specifies the operational semantics of **P³L** programs.

Definition 4.8 (Operational Semantics of P³L Programs) Let term be a special symbol indicating successful termination. Moreover, let $\langle \mathsf{sink}\rangle$ be a dedicated sink state and $\langle \mathsf{fault}\rangle$ be a state indicating unsuccessful termination. The *operational semantics of* **P³LA** *programs* is given by the transition system $\mathrm{oP^3L} \triangleq \langle\, \mathbf{States}, \rightsquigarrow, \mathbf{States}\,\rangle$, where both the set of states and the set of initial states is defined as

$$\mathbf{States} \triangleq \big(\big(\mathbf{P^3LA} \cup \{\,\mathsf{term}\,\}\big) \times \mathbf{SHPairs}\big) \cup \big\{\,\langle \mathsf{sink}\rangle, \langle \mathsf{fault}\rangle\,\big\}\,.$$

Moreover, the execution relation

$$\rightsquigarrow \subseteq \textbf{States} \times \textbf{States}$$

is the smallest relation induced by the rules in Figure 4.3, page 98, and Figure 4.4, page 99. As for **PL** programs, we define the *reachable fragment* of oP^{3}L with respect to the set of initial states $\mathbf{I}$ as

$$\text{oP}^3\text{L}\,(\mathbf{I}) \triangleq \textbf{Reach}\,(\langle\, \textbf{States}, \rightsquigarrow, \mathbf{I} \,\rangle)\,.$$

The *set of executions* of a **P^{3}L** program C and a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ is

$$\textbf{Exec}[C](\mathfrak{s}, \mathfrak{h}) \triangleq \left\{ \rho \;\middle|\; \rho \text{ is an execution of oP}^3\text{L}\,(\{\, \langle C, \mathfrak{s}, \mathfrak{h}\rangle \,\}) \right\}.$$

We conclude this section with an example. Since the allocation statement allows nondeterministic behavior, let us implement a **P^{3}L** program that "guesses" two positive natural numbers and then adds them to the value stored at address 17. This program is, of course, very silly. On an actual machine, the memory allocator would probably produce the same numbers most of the time. However, it illustrates potential pitfalls when programming with pointers. In particular, we may encounter failures or accidentally delete data.

Example 4.9 Consider the **P^{3}L** program C_{num} below:

$$\begin{aligned}
&x := \texttt{alloc}(0)\,;\\
&y := \texttt{alloc}(0)\,;\\
&z := <17>\,;\\
&<17> := x + y + z\,;\\
&\texttt{free}(x)\,;\, \texttt{free}(y)
\end{aligned}$$

We compute two executions of C_{num} on the stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, where $\mathfrak{s}$ initially evaluates every variable to zero and $\mathfrak{h} = \mathfrak{h}_\emptyset$ is the empty heap.

For the first execution, assume that the memory allocator chooses two distinct addresses $u \neq v$ in the two allocation statements such that neither u nor v equals 17. This yields the following execution:

$$\begin{aligned}
&\langle C_{\text{num}}, \mathfrak{s}, \mathfrak{h}_\emptyset \rangle\\
\rightsquigarrow\ &\langle y := \texttt{alloc}(0)\,;\, z := <17>\,;\, \ldots,\, \mathfrak{s}\,[x/u], \{\, u :: 0 \,\} \rangle\\
\rightsquigarrow\ &\langle z := <17>\,;\, \ldots,\, \mathfrak{s}\,[x/u]\,[y/v], \{\, u :: 0 \,\} \uplus \{\, v :: 0 \,\} \rangle
\end{aligned}$$

$$\frac{}{\langle \texttt{skip}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{skip} \qquad \frac{E(\mathfrak{s}) = v}{\langle x := E, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \rangle}\ \text{assign}$$

$$\frac{\langle C_1, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}', \mathfrak{h}' \rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C_2, \mathfrak{s}', \mathfrak{h}' \rangle}\ \text{seq1} \qquad \frac{\langle C_1, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C_1', \mathfrak{s}', \mathfrak{h}' \rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C_1' ; C_2, \mathfrak{s}', \mathfrak{h}' \rangle}\ \text{seq2}$$

$$\frac{\langle C_1, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{fault} \rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{fault} \rangle}\ \text{seq3} \qquad \frac{}{\langle \mathsf{fault} \rangle \rightsquigarrow \langle \mathsf{fault} \rangle}\ \text{fault}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C_1, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{if-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C_2, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{if-false}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{while}\,(B)\,\{C\}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C ; \texttt{while}\,(B)\,\{C\}, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{while-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{while}\,(B)\,\{C\}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{while-false}$$

$$\frac{}{\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{sink} \rangle}\ \text{term} \qquad \frac{}{\langle \mathsf{sink} \rangle \rightsquigarrow \langle \mathsf{sink} \rangle}\ \text{sink}$$

$$\frac{C = \mathsf{impl}\,(x := \mathrm{Proc}(E_1, \ldots, E_n))}{\langle x := \mathrm{Proc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{call}$$

$$\frac{}{\langle \texttt{enter}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[+\boxdot], \mathfrak{h} \rangle}\ \text{enter-scope}$$

$$\frac{}{\langle \texttt{leave}, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle \mathsf{term}, \mathfrak{s}[-\boxdot], \mathfrak{h} \rangle}\ \text{leave}$$

$$\frac{C = \mathsf{body}\,(F)}{\langle \texttt{invoke}\ F, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow \langle C, \mathfrak{s}, \mathfrak{h} \rangle}\ \text{invoke}$$

Figure 4.3: Adaption of the rules for **P²LA** programs presented in Definition 3.10 to account for heaps and potential memory faults.

$$\frac{u, u+1, \ldots, u+n-1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(\mathfrak{h}) \qquad E_1(\mathfrak{s}) = v_1, \ldots, E_n(\mathfrak{s}) = v_n}{\left\langle x := \texttt{alloc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}[x/u], \mathfrak{h} \uplus \{ u :: v_1, \ldots, v_n \} \right\rangle} \text{ alloc}$$

$$\frac{E(\mathfrak{s}) = u}{\left\langle \texttt{free}(E), \mathfrak{s}, \mathfrak{h} \uplus \{ u :: v \} \right\rangle \rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \right\rangle} \text{ free}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(h)}{\left\langle \texttt{free}(E), \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{fault} \right\rangle} \text{ free-fail}$$

$$\frac{E(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad \mathfrak{h}(u) = v}{\left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \right\rangle} \text{ lookup}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{fault} \right\rangle} \text{ lookup-fail}$$

$$\frac{E(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad E'(\mathfrak{s}) = v}{\left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h}[u/v] \right\rangle} \text{ mutation}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow \left\langle \mathsf{fault} \right\rangle} \text{ mutation-fail}$$

Figure 4.4: The rules that determine the execution relation of the operational semantics of **P³LA** program for statements that access the heap.

$$\rightsquigarrow \langle \mathsf{fault} \rangle \rightsquigarrow \langle \mathsf{fault} \rangle \rightsquigarrow \ldots \qquad (u \neq 17 \neq v)$$

Hence, as expected, the execution is prematurely aborted because we attempt to dereference address 17 although it is not allocated. For the second execution, we assume that the memory allocator chooses address 17 for the first allocation statement. In this case, the execution of program C_{num} terminates successfully, but the result is lost:

$$\begin{aligned}
& \langle C_{\mathsf{num}}, \mathfrak{s}, \mathfrak{h}_\emptyset \rangle \\
\rightsquigarrow^* & \langle z := <17>\,;\, <17> := x + y + z\,;\, \mathtt{free}(x)\,;\, \mathtt{free}(y), && \text{(as above)} \\
& \qquad \mathfrak{s}[x/17]\,[y/v], \{\, 17 :: 0 \,\} \uplus \{\, v :: 0 \,\} \rangle \\
\rightsquigarrow & \langle <17> := x + y + z\,;\, \mathtt{free}(x)\,;\, \mathtt{free}(y), \\
& \qquad \mathfrak{s}\,[x/17]\,[y/v]\,[z/0], \{\, 17 :: 0 \,\} \uplus \{\, v :: 0 \,\} \rangle \\
\rightsquigarrow & \langle \mathtt{free}(x)\,;\, \mathtt{free}(y), \mathfrak{s}\,[x/17]\,[y/v]\,[z/0]\,, \{\, 17 :: 17 + v \,\} \uplus \{\, v :: 0 \,\} \rangle \\
\rightsquigarrow & \langle \mathtt{free}(y), \mathfrak{s}\,[x/17]\,[y/v]\,[z/0]\,, \{\, v :: 0 \,\} \rangle \\
\rightsquigarrow & \langle \mathsf{term}, \mathfrak{s}\,[x/17]\,[y/v]\,[z/0]\,, \mathfrak{h}_\emptyset \rangle \rightsquigarrow \langle \mathsf{sink} \rangle \rightsquigarrow \ldots
\end{aligned}$$

4.1.4 Data Structures in P³L

Programming with pointers requires caution to avoid unexpected behavior. Let us briefly discuss a few idioms for implementing dynamic data structures in **P³L**. Furthermore, we introduce auxiliary notation for conveniently manipulating these data structures as in popular programming languages such as C or Java.

Arrays An *array* is a collection of values such that every element can be accessed via an index, i.e., a natural number ranging from 0 to the number of elements in the array (minus one). Arrays in **P³L** are implemented as in the C programming language [KR88]. That is, an array is a consecutive block of n allocated addresses. For example, the **P³L** statement $x := \mathtt{alloc}(0, 0, 0)$ creates a new array consisting of three memory cells that are initialized with zero. Moreover, the first address of the array is assigned to variable x. Consequently, each element of the array can be accessed trough x: $<x + 0>$ denotes the first value stored in the array, $<x + 1>$ its second value, and $<x + 2>$ its third, respectively. The more common notation $x\,[\,i\,]$, which refers to the i-th element of the array given by variable x, is thus syntactic sugar for $<x + i>$. We remark that—as in the C programming language—no range checks are performed when attempting to access an array element. Hence, statements like $y := x\,[\,i\,]$ may fail or access memory cells outside of the array. Furthermore, the length of an array cannot be reconstructed from the array itself. It must thus be stored in a separate

```
sumAndDelete(array, length) {
    x := 0; y := 0;
    while ( x < length ) {
        y := y + array [ x ] ;
        free(array + x);
        x := x + 1
    }
    out := y
}
```

Figure 4.5: A procedure declaration in $\mathbf{P^3L}$ that operates on arrays.

variable. As an example, consider the $\mathbf{P^3L}$ procedure sumAndDelete(array, length) in Figure 4.5. This procedure takes an array and its length as an input and returns the sum of all (length) values stored in the array. In addition to that, the array is deleted.

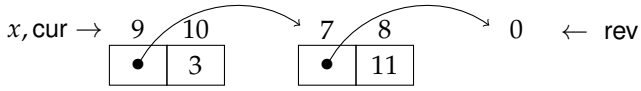
Linked data structures Linked data structures, such as lists and trees, consist of multiple elements that reference each other via pointers. Elements of a data structure are usually represented by objects (Java), structs (C), or records (Pascal) that encapsulate all members of an element which can then be accessed through a single variable. For instance, every element of a singly-linked list, say x, consists of two members: A pointer to the next element and some payload data. The members of an element are then accessible via *selectors*, e.g., x.next for the next pointer and x.val for the payload. To model objects in $\mathbf{P^3L}$, let $\mathbf{Sel} \triangleq (sel_0, \ldots, sel_{n-1})$ be a fixed sequence of $n \in \mathbb{N}$ selectors. Then, similarly to structs in C, an object in $\mathbf{P^3L}$ is a consecutive block of n allocated memory cells.[2] Analogously to array variables, an *object variable*, say x, stores the first address in such a block. The statement $x := \texttt{alloc}(v_1, \ldots, v_n)$ thus creates a new object in which the i-th member is initialized with the value v_i. Consequently, the value of the i-th selector of the object referenced by x is given by $<x + i>$. For convenience, we write $x.sel_i$ instead of $<x + i>$. Coming back to our example of

[2]For simplicity, we assume that all objects use the same selectors. Otherwise, we would also have to store additional information such as the object's type or its number of selectors. Moreover, we remark that the usage of "objects" in $\mathbf{P^3L}$ relies on *conventions*. Nothing prevents us from manipulating or deleting the memory cells that make up an object.

C_{reverse}: `while ( cur` $\neq 0$ `) {`
C_{body}: $y :=$ cur.next;
cur.next := rev;
rev := cur;
cur := y
}

Figure 4.6: A $\mathbf{P^3L}$ program manipulating a singly-linked list with head x.

singly-linked lists, we choose the sequence of selectors $\mathbf{Sel} = (\mathsf{next}, \mathsf{val})$. Hence, the first address of every list object stores a pointer to the next list object and the second address stores the payload. In the stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ depicted below, x.val then refers to the value 3 at address 10. Analogously, x.next.val refers to the value 11 at address 8:

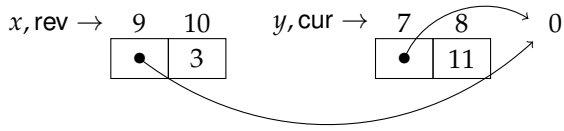


We conclude this section with an example program that manipulates lists.

Example 4.10 Let us execute the $\mathbf{P^3L}$ program C_{reverse} in Figure 4.6 on the above stack-heap pair $(\mathfrak{s}, \mathfrak{h})$. Since variable cur initially stores address 9, i.e., $\mathfrak{s}(\mathsf{cur}) = 9$, the first steps of an execution of C_{reverse} on $(\mathfrak{s}, \mathfrak{h})$ are as follows:

$$\begin{aligned} & \langle C_{\text{reverse}}, \mathfrak{s}, \mathfrak{h} \rangle \\ \rightsquigarrow \; & \langle C_{\text{body}} ; C_{\text{reverse}}, \mathfrak{s}, \mathfrak{h} \rangle \\ \rightsquigarrow^* \; & \langle C_{\text{reverse}}, \mathfrak{s}[y/7]\,[\mathsf{rev}/9]\,[\mathsf{cur}/7], \mathfrak{h}[9/0] \rangle, \end{aligned}$$

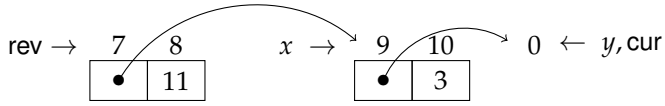
where the last stack-heap pair is illustrated below:



In this stack-heap pair, variable cur is still not equal to 0. Hence, we enter the loop body once more and the execution proceeds as follows:

$$
\begin{aligned}
& \langle C_{\mathsf{reverse}}, \mathfrak{s}\,[y/7]\,[\mathsf{rev}/9]\,[\mathsf{cur}/7], \mathfrak{h}\,[9/0]\,\rangle \\
\rightsquigarrow\ & \langle C_{\mathsf{body}}\,;\, C_{\mathsf{reverse}}, \mathfrak{s}\,[y/7]\,[\mathsf{rev}/9]\,[\mathsf{cur}/7], \mathfrak{h}\,[9/0]\,\rangle \\
\rightsquigarrow^*\ & \langle C_{\mathsf{reverse}}, \mathfrak{s}[y/0]\,[\mathsf{rev}/7]\,[\mathsf{cur}/0], \mathfrak{h}\,[9/0]\,[7/9]\rangle \\
\rightsquigarrow\ & \langle \mathsf{term}, \mathfrak{s}\,[y/0]\,[\mathsf{rev}/7]\,[\mathsf{cur}/0], \mathfrak{h}\,[9/0]\,[7/9]\,\rangle \rightsquigarrow \langle \mathsf{sink}\rangle \rightsquigarrow \ldots
\end{aligned}
$$

To understand the effect of program C_{reverse} on the initial stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, consider the stack-heap pair reached upon termination:



The value of variable x remains unchanged. The list itself, however, has been reversed. Moreover, variable rev points to the head of the reversed list.

4.2 Hurdles for Program Verification

Pointer programs are at the foundation of many efficient algorithms and enable the implementation of dynamic data structures. This makes pointers—at least at some level of abstraction—an indispensable part of programming. Unfortunately, programming with pointers is also highly error-prone: More than a decade after the release of the first language with support for pointers [Law67], pointer manipulation was (and still is) regarded as "one of the trickiest operations in programming and [...] a major source of programming errors" [Suz80].

All of the above aspects of pointer programs justify why their formal verification is desirable. However, like programming with pointers, reasoning about pointer operations is challenging: While extensions of Floyd-Hoare logic [Flo67; Hoa69] for reasoning about both pointer programs [Bur72] and recursive procedures [Hoa71] have been proposed at roughly the same time, there is a stark difference: The rule proposed in [Hoa71] is an elegant solution for proving partial correctness of procedures (cf. Chapter 3). In contrast, various Hoare logics for pointer programs have been proposed, e.g., [HW73; LS79; Bij89], and dismissed. Almost thirty years later, Bornat [Bor00] adequately summarized the situation up until year 2000: "It is possible, but difficult, to reason in Hoare logic about programs which address and modify data structures defined by pointers. [...] Hoare logic isn't widely used to verify pointer programs".

In this section, we briefly examine the challenges faced when reasoning about pointer programs with Hoare logic. After that, we study *separation logic*, which is an extension of Hoare logic that addresses these challenges.

No immediate technical difficulties arise when incorporating pointers and dynamic memory allocation into Hoare logic. In fact, in comparison to previous chapters, only a few changes of the foundations of Hoare logic—predicates, triples, validity, and semantic weakest preconditions—are needed. First, to express properties of the heap, we extend the set **Pred** of predicates.

Definition 4.11 (Predicate over Stack-Heap Pairs) The *set of predicates over stack-heap pairs* is defined as

$$\mathbf{Pred} \triangleq \{ P \colon \mathbf{SHPairs} \to \mathbb{T} \}.$$

Moreover, the *set of stack-heap pairs* captured by predicate P is defined as

$$\mathbf{SHPairs}(P) \triangleq \{ (\mathfrak{s}, \mathfrak{h}) \in \mathbf{SHPairs} \mid P(\mathfrak{s}, \mathfrak{h}) = \mathsf{true} \}.$$

We write $\mathfrak{s}, \mathfrak{h} \models P$ (read: $(\mathfrak{s}, \mathfrak{h})$ satisfies P) as a shortcut for $P(\mathfrak{s}, \mathfrak{h}) = \mathsf{true}$. Analogously, we write $\mathfrak{s}, \mathfrak{h} \not\models P$ (read: $(\mathfrak{s}, \mathfrak{h})$ violates P) for $P(\mathfrak{s}, \mathfrak{h}) = \mathsf{false}$.

Every predicate considered in previous chapters, e.g., $x > 0$, is also a predicate over stack-heap pairs. Throughout the remainder of this thesis, we always consider predicates over stack-heap pairs. Hence, we use the terms "predicate" and "predicate over stack-heap pairs" synonymously. Furthermore, for our extended set of predicates **Pred**, (**Pred**, $\Rightarrow$) remains a complete lattice, where the logical implication $\Rightarrow$ is given by

$$P \Rightarrow Q \quad \text{iff} \quad \mathbf{SHPairs}(P) \subseteq \mathbf{SHPairs}(Q).$$

The proof is analogous to the proof of Lemma 2.23.

Hoare triples for $\mathbf{P^3L}$ are defined mutatis mutandis as for **PL** programs. That is, a Hoare triple $\langle P \rangle\ C\ \langle Q \rangle$ consists of a precondition $P \in \mathbf{Pred}$, a $\mathbf{P^3L}$ program C, and a postcondition $Q \in \mathbf{Pred}$. The question whether a Hoare triple is valid or not, however, is more subtle. Consider, for example, the $\mathbf{P^3L}$ program

$$\mathtt{free}(x)\,;\,\mathtt{free}(x).$$

Regardless of the initial stack-heap pair, this program will always lead to a memory failure: If the address stored in variable x is initially not allocated, then we immediately fail. Otherwise, the same address will not be allocated after execution of $\mathtt{free}(x)$. Hence, we fail when trying to execute $\mathtt{free}(x)$ again. Every execution of our operational semantics is thus of the form

$$\langle \mathtt{free}(x)\,;\,\mathtt{free}(x), \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow^* \langle \mathsf{fault} \rangle \rightsquigarrow \ldots$$

In particular, no execution ever reaches a state $\langle \mathsf{term}, \mathfrak{s}', \mathfrak{h}' \rangle$ indicating successful termination. This means that—if we do not change our definition of validity introduced in Definition 2.22—the Hoare triple

$$\langle\, \mathsf{true} \,\rangle\ \texttt{free}(x)\,;\,\texttt{free}(x)\ \langle\, \mathsf{true} \,\rangle$$

is valid with respect to partial correctness. It is even valid for total correctness because every execution eventually terminates, i.e., reaches the sink state. In both cases, however, every execution terminates *unsuccessfully* with a failure. This is clearly undesirable. If we are able to prove a Hoare triple valid, i.e., show that a program meets its specification, then this should rule out the possibility of encountering a failure. Ishtiaq and O'Hearn [IO01] gave an adequate summary of this situation: "Well-specified programs don't go wrong." This is formalized by considering *memory safe* Hoare triples [YO02, Section 3].

Definition 4.12 (Memory Safe Hoare Triple) The Hoare triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is *memory safe* if and only if

$$\forall (\mathfrak{s}, \mathfrak{h}) \in \mathbf{SHPairs}\,(P)\ \ \neg\exists (\mathfrak{s}', \mathfrak{h}') \in \mathbf{SHPairs}\colon \langle C, \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow^{*} \langle \mathsf{fault} \rangle.$$

From now on, we require that a Hoare triple is memory safe in order to be valid. That is, $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is *valid for partial correctness* if and only if it is memory safe and valid for partial correctness in the sense of Definition 2.22. Analogously, $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is *valid for total correctness* if and only if it is memory safe and valid for total correctness in the sense of Definition 2.22.

Since failures in $\mathbf{P^3L}$ are caused by illegal attempts to access the heap, all Hoare triples considered for **PL** and $\mathbf{P^2L}$ programs are automatically memory safe.

Apart from using memory safe Hoare triples under the hood, the definition of (semantic) weakest preconditions remains unchanged (cf. Definition 2.24, page 40). That is, for both total and partial correctness, the *semantic weakest precondition* WP of $\mathbf{P^3L}$ program C with respect to postcondition $Q \in \mathbf{Pred}$ is

$$WP \ \triangleq\ \sup \left\{\, P \;\middle|\; \langle\, P \,\rangle\ C\ \langle\, Q \,\rangle \text{ is valid} \,\right\}.$$

Why is reasoning about pointer programs more challenging? While we had to slightly change the notion of valid Hoare triples, our formalization of Hoare logic and (semantic) weakest preconditions for $\mathbf{P^3L}$ has barely changed if compared to the definitions for **PL** and $\mathbf{P^2L}$ programs. The difficulties thus do *not* lie in the underlying formalization, but in the way weakest preconditions are computed. O'Hearn, Reynolds, and Yang [ORY01] got to the heart of the issue: "The main difficulty is not one of finding an in-principle adequate axiomatization of pointer operations; rather, there is a mismatch between simple intuitions about the way

that pointer operations work and the complexity of their axiomatic treatments". In other words, the main challenge is to find inference rules for $\mathbf{P^3L}$ programs that preserve the compelling aspects of Hoare logic. So what makes Hoare logic compelling? Hoare logic is a systematic approach for reasoning about programs that facilitates *local reasoning*: For proving a program correct, it suffices to consider only the aspects relevant for the program's correctness instead of all pedantic details of program executions. This is exactly the reason we gave when introducing Hoare logic in Section 2.3. From a technical perspective, local reasoning is enabled by the rule for assignments, i.e.,

$$\mathsf{wlp}\,[x := E]\,(Q) \;\triangleq\; Q\,[x/E]\,.^3$$

An assignment to variable x thus amounts to a purely syntactic substitution of all (free) occurrences of x by the assigned expression E. Consequently, exactly those parts of postcondition Q for which variable x is relevant are changed. In particular, if x does not occur in Q, then the assignment has no effect on it. The same principle permeates Hoare logic as a whole: It shows up in the *rule of invariance*, which we derived for weakest liberal preconditions in Theorem 2.40:

$$Q \Rightarrow \mathsf{wlp}\,[C]\,(Q) \qquad \text{if } \mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(Q) = \emptyset.$$

Furthermore, the rule of invariance (or a similar notion from which it can be derived) is *necessary* for the completeness of Hoare logic in the presence of recursive procedures [Apt81; Old83]. A concise formalization of local reasoning is obtained when combining the rule of invariance with the rule of conjunction (Theorem 2.38): If $\mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(R) = \emptyset$, then

$$\begin{array}{lll} & \mathsf{wlp}\,[C]\,(Q) \wedge R & \\ \Rightarrow & \mathsf{wlp}\,[C]\,(Q) \wedge \mathsf{wlp}\,[C]\,(R) & \text{(Theorem 2.40)} \\ = & \mathsf{wlp}\,[C]\,(Q \wedge R)\,. & \text{(Theorem 2.38)} \end{array}$$

Hence, it suffices to reason only about the aspects of a predicate that are relevant to a program. Other parts, i.e., the predicate R in the above equality, do not have to be considered when computing weakest (liberal) preconditions. In Hoare logic, this also known as the *rule of constancy* [Rey81; Rey02]:

$$\frac{\langle\, P \,\rangle\; C \;\langle\, Q \,\rangle \qquad \mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(R) = \emptyset}{\langle\, P \wedge R \,\rangle\; C \;\langle\, Q \wedge R \,\rangle}\ \text{constancy}$$

[3] For the sake of the argument, we ignore the distracting possibility of nontermination and restrict ourselves to reasoning about partial correctness.

Unfortunately, incorporating pointers into Hoare logic inflicts causalities to all of the above aspects of local reasoning: For example, consider the $\mathbf{P^3L}$ program

$$<x> := 3\,;\ <y> := 17\,;\ z := <x>.$$

It is tempting to conclude that this program terminates with $z = 3$. In fact, a naïve application of the weakest liberal precondition calculus from previous chapters seems to confirm this hypothesis (read from bottom to top as in Example 2.29):

$$\begin{array}{ll}
\mathit{//}\quad 3 = 3 & \text{(assuming standard rule for assignments)} \\
<x> := 3\,; & \\
\mathit{//}\quad <x> = 3 & \text{(assuming rule of invariance)} \\
<y> := 17\,; & \\
\mathit{//}\quad <x> = 3 & \text{(assuming standard rule for assignments)} \\
z := <x> & \\
\mathit{//}\quad z = 3 &
\end{array}$$

This "proof", however, ignores the possibility that variables x and y could be *aliases*, i.e., store the same address. As the following execution of the above program on a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ with $\mathfrak{s}(x) = \mathfrak{s}(y) = 2$ demonstrates, we indeed obtain a different result—$z = 17$—for this corner case:

$$\begin{aligned}
&\big\langle <x> := 3\,;\ <y> := 17\,;\ z := <x>,\ \mathfrak{s}, \mathfrak{h} \big\rangle \\
\rightsquigarrow\ &\big\langle <y> := 17\,;\ z := <x>,\ \mathfrak{s}, \mathfrak{h}[2/3] \big\rangle \\
\rightsquigarrow\ &\big\langle z := <x>,\ \mathfrak{s}, \mathfrak{h}[2/17] \big\rangle \\
\rightsquigarrow\ &\big\langle \mathsf{term},\ \mathfrak{s}[z/17], \mathfrak{h}\,[2/17]\,\big\rangle \\
\rightsquigarrow\ &\big\langle \mathsf{sink} \big\rangle \rightsquigarrow \big\langle \mathsf{sink} \big\rangle \rightsquigarrow \ldots
\end{aligned}$$

Hence, the rule of invariance is *unsound* in the presence of pointer manipulations. Furthermore, it is unsound to use a purely syntactic substitution to deal with assignments because substituting every occurrence of $<y>$ in $<x> = 3$ by 17 would have led to the same result. Similar counterexamples exist to prove that the rule of constancy is unsound in the presence of aliasing [Rey02].

We might get away with a *semantic* substitution for assignments, i.e., substitute all addresses equal to $<y>$ by 17 in the above example. However, such rules enforce global rather than local reasoning: We would have to constantly check whether variables are aliases or not; thus blowing up predicates with case distinctions. Even worse, these checks have to performed for *all* variables—including variables that do not occur in a predicate. Other approaches suffer

from the same problem: For instance, the rule for assignments in [LS79] adds a global condition that requires the non-existence of other pointers to the same address. Similarly, [Bur72] introduced rules that rely on universal quantifiers over all possible addresses to avoid aliasing.

To sum up, the *main challenge when incorporating pointers in Hoare logic is to present sound and complete rules that recover local reasoning as much as possible*.

A satisfactory solution of this challenge was developed in a series of papers by O'Hearn, Reynolds, Yang, and others [IO01; ORY01; YO02; Rey02]. The key idea lies in two novel connectives between predicates—the *separating conjunction* $\star$ and the *separating implication* $\mathrel{-\!\!\star}$ —that keep segments of the heap separate and thus prevent unintended aliasing. Burstall [Bur72] already stressed the importance of separation for reasoning about pointer programs. In particular, he introduced "distinct non-repeating list systems" for reasoning about linked lists. While his approach is tailored to lists and trees, it significantly simplifies Floyd-Hoare style proofs. We encourage the reader to compare his proof of a list reversal algorithm with and without exploiting separation [Bur72, pages 31–33]. Separation logic builds on his ideas to achieve local reasoning.

The remainder of this chapter is an introduction to separation logic. First, we study separation logic's assertion language, i.e., its characteristic predicates and connectives. We then present rules for computing weakest preconditions within this language. Finally, we discuss proof rules and local reasoning.

4.3 Separation Logic Assertions

Although the assertion language of separation logic is usually defined following an intensional approach, i.e., by providing an explicit syntax [IO01; Rey02], we adhere to an extensional approach as in previous chapters. An explicit syntax for a fragment of the assertion language is considered in Chapter 10. Recall from Definition 4.11 that our assertion language consists of all predicates

$$P\colon \mathbf{SHPairs} \to \mathbb{T}$$

mapping stack-heap pairs to either true or false. Every predicate $P\colon \mathbf{Stacks} \to \mathbb{T}$ considered in previous chapters can be written as such a predicate, namely

$$\lambda(\mathfrak{s},\mathfrak{h}).\, P\,(\mathfrak{s})\,.$$

Moreover, we use the same connectives as in Section 2.3.1 to compose predicates. That is, $P \wedge Q$ and $P \vee Q$ denote predicates that capture the intersection and the union of the sets $\mathbf{SHPairs}\,(P)$ and $\mathbf{SHPairs}\,(Q)$, respectively. Furthermore, $\neg P$ captures the set of all stack heap pairs in **SHPairs** that are not contained in $\mathbf{SHPairs}\,(P)$. Since the heap is not concerned with variables, we lift our auxiliary

definitions for manipulating variables, i.e., variable substitution (Definition 2.26), scope increment, and scope decrement (Definition 3.14) to predicates over stack-heap pairs by applying them to the stack component only:

- $P\,[x/E] \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, P\,(\mathfrak{s}\,[x/E\,(\mathfrak{s})]\,,\mathfrak{h})$,
- $P\,[+\boxdot] \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, P\,(\mathfrak{s}\,[+\boxdot]\,,\mathfrak{h})$, and
- $P\,[-\boxdot] \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, P\,(\mathfrak{s}\,[-\boxdot]\,,\mathfrak{h})$.

Finally, analogously to Definition 2.39, the set $\mathbf{Vars}\,(P) \subseteq \mathbf{Vars}$ of variables occurring in predicate P is given by:

$$x \in \mathbf{Vars}\,(P) \quad \text{iff} \quad \exists(\mathfrak{s},\mathfrak{h}) \in \mathbf{SHPairs}\, \exists u, v \in \mathbb{Z}\colon P\,[x/u]\,(\mathfrak{s},\mathfrak{h}) \neq P\,[x/v]\,(\mathfrak{s},\mathfrak{h})\,.$$

4.3.1 The Atoms of Separation Logic

Let us take a look at predicates that actually depend on the heap. To this end, the atomic formulas of separation logic, which have been originally presented in [IO01; Rey02], serve us as a source of examples.

The *empty-heap* predicate **emp** captures all stack-heap pairs in which the heap is empty. Formally, it is defined as:

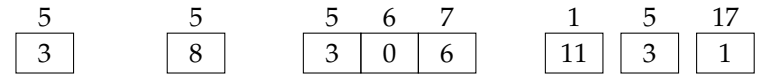
$$\mathbf{emp} \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, \begin{cases} \mathsf{true}, & \text{if } \mathsf{dom}(\mathfrak{h}) = \emptyset \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

The *points-to* predicate $E \mapsto E'$ evaluates to true if and only if the heap consists of exactly one memory cell whose address is given by expression E and whose content is given by expression E', respectively:

$$E \mapsto E' \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, \begin{cases} \mathsf{true}, & \text{if } \mathfrak{h} = \{\, E\,(\mathfrak{s}) :: E'\,(\mathfrak{s})\,\} \\ \mathsf{false}, & \text{otherwise,} \end{cases}$$

where $\{\, E\,(\mathfrak{s}) :: E'\,(\mathfrak{s})\,\}$ is our shorthand notation for heaps introduced in Section 4.1.2. Furthermore, the predicate $E \mapsto E_1, \ldots, E_n$ specifies a consecutive block of n memory cells:

$$E \mapsto E_1, \ldots, E_n \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, \begin{cases} \mathsf{true}, & \text{if } \mathfrak{h} = \{\, E\,(\mathfrak{s}) :: E_1\,(\mathfrak{s}), \ldots, E_n\,(\mathfrak{s})\,\} \\ \mathsf{false}, & \text{otherwise,} \end{cases}$$



$5 \mapsto 3$
$5 \mapsto -$
$5 \hookrightarrow 3$

$5 \mapsto -$

$5 \mapsto 3, 0, 6$
$5 \hookrightarrow 3$

$5 \hookrightarrow 3$

Figure 4.7: A collection of four heaps and four predicates. If a heap satisfies a predicate (for an arbitrary stack) it is provided below the respective heap.

Sometimes, we only care about the fact that an address is allocated rather than what value is stored at that address. In this case, we use the *allocated pointer* predicate $E \mapsto -$ which is defined as:

$$E \mapsto - \triangleq \lambda(\mathfrak{s}, \mathfrak{h}). \begin{cases} \mathsf{true}, & \text{if } \mathsf{dom}(\mathfrak{h}) = \{ E(\mathfrak{s}) \} \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

All of the above predicates are *precise* in the sense that they specify exactly the addresses the heap must consist of. A more relaxed notion is the *contains-pointer* predicate $E \hookrightarrow E'$, which evaluates to true if and only if the heap contains a memory cell with address E and value E':

$$E \hookrightarrow E' \triangleq \lambda(\mathfrak{s}, \mathfrak{h}). \begin{cases} \mathsf{true}, & \text{if } \{ E(\mathfrak{s}) :: E'(\mathfrak{s}) \} \subseteq \mathfrak{h} \\ \mathsf{false}, & \text{otherwise,} \end{cases}$$

where $\subseteq$ denotes heap inclusion as introduced in Section 4.1.2.

Example 4.13 Consider the following four predicates:

$$5 \mapsto 3 \qquad 5 \mapsto 3, 0, 6 \qquad 5 \mapsto - \qquad 5 \hookrightarrow 3.$$

Since none of these predicates contain any variables, their captured stack-heap pairs depend on the heap only. Figure 4.7 illustrates four different heaps and, for each of these heaps $\mathfrak{h}$, collects which of the above predicates are satisfied by $\mathfrak{h}$ (and an arbitrary stack).

4.3.2 Separating Conjunction and Separating Implication

We now turn to the two prominent connectives of separation logic: separating conjunction and separating implication. The key idea is to treat the heap as a

resource that is modified prior to evaluation; the separating conjunction cuts off a part of the heap whereas the separating implication extends it (cf. [IO01]).

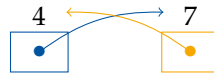
Separating Conjunction In contrast to the standard conjunction $P \wedge Q$, the *separating conjunction* $P \star Q$ of two predicates P and Q first *partitions* the heap into two pieces and evaluates each conjunct in one piece.

Definition 4.14 (Separating Conjunction [IO01; Rey02]) The *separating conjunction* $P \star Q$ of two predicates P and Q is given by

$$\mathfrak{s}, \mathfrak{h} \models P \star Q \quad \text{iff} \quad \exists \mathfrak{h}_1, \mathfrak{h}_2 \colon \ \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \text{ and } \mathfrak{s}, \mathfrak{h}_1 \models P \text{ and } \mathfrak{s}, \mathfrak{h}_2 \models Q.$$

Intuitively, the separating conjunction $P \star Q$ cuts off a heap specified by predicate P before evaluating Q. For example, the predicate $4 \mapsto 7 \star Q$ cuts off the memory cell $\{ 4 :: 7 \}$ and evaluates Q in the remaining heap. Consequently, for the heap $\mathfrak{h}$ depicted below and an arbitrary stack $\mathfrak{s}$, we have $\mathfrak{s}, \mathfrak{h} \models 4 \mapsto 7 \star 7 \mapsto 4$.

$$\mathfrak{h} = \{ 4 :: 7 \} \uplus \{ 7 :: 4 \}$$



Separating Implication The *separating implication* $P \mathrel{-\!\!\star} Q$ of two predicates P and Q first *extends* the heap as specified by predicate P (and disjoint from the original heap) and then evaluates Q in the extended heap.

Definition 4.15 (Separating Implication [IO01; Rey02]) Given two predicates P and Q, the *separating implication* $P \mathrel{-\!\!\star} Q$ is given by

$$\mathfrak{s}, \mathfrak{h} \models P \mathrel{-\!\!\star} Q \quad \text{iff} \quad \forall \mathfrak{h}' \colon \ \big(\mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P \big) \text{ implies } \mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \models Q.$$

For example, consider the contains-pointer predicate $7 \hookrightarrow 4$ as well as the heap $\mathfrak{h} = \{ 4 :: 7 \} \uplus \{ 7 :: 4 \}$ from above. Clearly, for every stack $\mathfrak{s}$, we have

$$\mathfrak{s}, \{ 4 :: 7 \} \not\models 7 \hookrightarrow 4 \qquad \text{and} \qquad \mathfrak{s}, \{ 4 :: 7 \} \uplus \{ 7 :: 4 \} \models 7 \hookrightarrow 4.$$

Moreover, there exists exactly one heap, namely $\{ 7 :: 4 \}$, such that the points-to predicate $7 \mapsto 4$ is satisfied. Consequently, it holds that

$$\begin{aligned}
& \mathfrak{s}, \{ 4 :: 7 \} \models 7 \mapsto 4 \mathrel{-\!\!\star} 7 \hookrightarrow 4 \\
\text{iff} \quad & \forall \mathfrak{h}' \colon \ \big(\{ 4 :: 7 \} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models 7 \mapsto 4 \big) \text{ implies } \mathfrak{s}, \{ 4 :: 7 \} \uplus \mathfrak{h}' \models 7 \hookrightarrow 4 \\
\text{iff} \quad & \mathfrak{s}, \{ 4 :: 7 \} \uplus \{ 7 :: 4 \} \models 7 \hookrightarrow 4
\end{aligned}$$

which is valid by definition of $7 \hookrightarrow 4$.

Example 4.16 Let us consider a few more examples to compare the separating connectives $\star$ and $\mathrel{-\!\!\star}$ with the standard connectives $\wedge$ and $\Rightarrow$. We have already discussed that, for an arbitrary stack $\mathfrak{s}$, it holds

$$\mathfrak{s}, \{\, 4 :: 7 \,\} \uplus \{\, 7 :: 4 \,\} \models 4 \mapsto 7 \star 7 \mapsto 4.$$

This is not true for a standard conjunction, i.e.,

$$\mathfrak{s}, \{\, 4 :: 7 \,\} \uplus \{\, 7 :: 4 \,\} \not\models 4 \mapsto 7 \wedge 7 \mapsto 4$$

as $\{\, 4 :: 7 \,\} \uplus \{\, 7 :: 4 \,\}$ is not a single memory cell. Conversely, it clearly holds that $\mathfrak{s}, \{\, 4 :: 7 \,\} \models 4 \mapsto 7 \wedge 4 \mapsto 7$. If we replace the conjunction by a separating one, however, the predicate is violated:

$$\mathfrak{s}, \{\, 4 :: 7 \,\} \not\models 4 \mapsto 7 \star 4 \mapsto 7.$$

What about separating implication and standard implication? The predicate $4 \mapsto 7 \mathrel{-\!\!\star} 4 \mapsto 7$ and the predicate $4 \mapsto 7 \Rightarrow 4 \mapsto 7$ are both satisfied by the stack-heap pair $(\mathfrak{s}, \mathfrak{h}_\emptyset)$. If we consider another heap, however, we have

$$\mathfrak{s}, \{\, 4 :: 7 \,\} \not\models 4 \mapsto 7 \mathrel{-\!\!\star} 4 \mapsto 7 \quad \text{and} \quad \mathfrak{s}, \{\, 4 :: 7 \,\} \models 4 \mapsto 7 \Rightarrow 4 \mapsto 7.$$

What about combinations of $\star$ and $\mathrel{-\!\!\star}$? The predicate

$$4 \mapsto 7 \star (4 \mapsto 2 \mathrel{-\!\!\star} 4 \hookrightarrow x)$$

updates the value stored at address 4 to 2 before evaluating $4 \hookrightarrow x$. To this end, we first cut off the heap $\{\, 4 :: 7 \,\}$, then insert the heap $\{\, 4 :: 2 \,\}$, and finally evaluate $4 \hookrightarrow x$ in the result. Hence, for every stack $\mathfrak{s}$ such that

$$\mathfrak{s}, \{\, 4 :: 7 \,\} \models 4 \mapsto 7 \star (4 \mapsto 2 \mathrel{-\!\!\star} 4 \hookrightarrow x)$$

holds, we have $\mathfrak{s}(x) = 2$. Furthermore, if we cut off a heap and then insert the *same* heap again, the result is evaluated in the original heap. Hence,

$$\mathfrak{s}, \mathfrak{h} \models 4 \mapsto 7 \star (4 \mapsto 7 \mathrel{-\!\!\star} 4 \hookrightarrow 7) \quad \text{iff} \quad \mathfrak{s}, \mathfrak{h} \models 4 \hookrightarrow 7.$$

An immediate benefit of the separating conjunction is that it *prevents unintended aliasing effects*: For instance, consider the predicate $x \mapsto E \star y \mapsto E$. Is it possible that x and y are aliases, i.e., they store the same address? For any $(\mathfrak{s}, \mathfrak{h})$, we have

$$\begin{aligned}
& \mathfrak{s}, \mathfrak{h} \models x \mapsto E \star y \mapsto E \\
\text{iff} \quad & \exists \mathfrak{h}_1, \mathfrak{h}_2 \colon \mathfrak{h}_1 \uplus \mathfrak{h}_2 \text{ and } \mathfrak{s}, \mathfrak{h}_1 \models x \mapsto E \text{ and } \mathfrak{s}, \mathfrak{h}_2 \models y \mapsto E \\
\text{iff} \quad & \exists \mathfrak{h}_1, \mathfrak{h}_2 \colon \mathfrak{h}_1 \uplus \mathfrak{h}_2 \text{ and } \mathfrak{h}_1 = \{\, \mathfrak{s}(x) :: E \,\} \text{ and } \mathfrak{h}_2 = \{\, \mathfrak{s}(y) :: E \,\}.
\end{aligned}$$

Now, the disjoint union $\mathfrak{h}_1 \uplus \mathfrak{h}_2$ is defined if and only if $\mathfrak{h}_1 \# \mathfrak{h}_2$ holds, i.e., the heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ have disjoint domains. Hence,

$$\begin{array}{lll} & \mathfrak{h}_1 \# \mathfrak{h}_2 & \\ \text{iff} & \mathsf{dom}(\mathfrak{h}_1) \cap \mathsf{dom}(\mathfrak{h}_2) = \emptyset & \\ \text{iff} & \{\, \mathfrak{s}\,(x)\,\} \cap \{\, \mathfrak{s}\,(y)\,\} = \emptyset & \text{(by definition of } \mathfrak{h}_1, \mathfrak{h}_2 \text{ from above)} \\ \text{iff} & \mathfrak{s}\,(x) \neq \mathfrak{s}\,(y)\,. & \end{array}$$

In other words, if $x \mapsto E \star y \mapsto E$ holds, then the variables x and y are not aliases. A direct consequence of this observation is that predicates of the form

$$E \mapsto E' \;\star\; E \mapsto E''$$

cannot be satisfied by any stack-heap pair.

Properties of Separating Conjunction and Implication Let us collect a few algebraic properties of the separating connectives $\star$ and $\mathrel{-\!\!\star}$; all of which are taken from [IO01; Rey02].

First, we may omit brackets, add and remove the predicate **emp** and reorder the conjuncts of $\star$ due to the underlying monoidal structure.

Theorem 4.17 (Monoidicity of $\star$ [IO01]) $(\mathbf{Pred}, \star, \mathbf{emp})$ is a commutative monoid, i.e., for all predicates $P, Q, R \in \mathbf{Pred}$, we have:

(a) Associativity: $P \star (Q \star R) = (P \star Q) \star R$.

(b) Neutrality of **emp** : $P \star \mathbf{emp} = \mathbf{emp} \star P = P$.

(c) Commutativity: $P \star Q = Q \star P$.

Furthermore, the separating conjunction is a monotone connective:

Theorem 4.18 (Monotonicity of $\star$ [Rey02]) For all predicates $P, Q, R \in \mathbf{Pred}$:

$$P \Rightarrow R \quad \text{implies} \quad (P \star Q) \Rightarrow (R \star Q).$$

The separating conjunction $\star$ behaves similarly to the standard conjunction $\wedge$. In particular, it satisfies the same (at least in one direction) distributivity laws for disjunction, conjunction, and quantifiers.[4]

[4]Since we do not have an explicit syntax for predicates, let us briefly give an extensional definition of the sets of stack-heap pairs captured by quantifiers: For the existential quantification, we define this set as $\mathbf{SHPairs}\,(\exists x\colon P) \triangleq \bigcup_{v \in \mathbb{Z}} \mathbf{SHPairs}\,(P\,[x/v])$. Analogously, for the universal quantification, we define $\mathbf{SHPairs}\,(\forall x\colon P) \triangleq \bigcap_{v \in \mathbb{Z}} \mathbf{SHPairs}\,(P\,[x/v])$.

Theorem 4.19 ((Sub)distributivity Laws [Rey02]) For all $P, Q, R \in \mathbf{Pred}$:

(a) $(P \vee Q) \star R \;=\; (P \star R) \vee (Q \star R)$,

(b) $((P \wedge Q) \star R) \;\Rightarrow\; (P \star R) \wedge (Q \star R)$,

(c) if $x \notin \mathbf{Vars}(P)$, then $P \star (\exists x\colon Q) \;=\; \exists x\colon (P \star Q)$, and

(d) if $x \notin \mathbf{Vars}(P)$, then $(P \star (\forall x\colon Q)) \;\Rightarrow\; \forall x\colon (P \star Q)$.

We previously used $\star$ to *shrink* the heap and $\mathrel{-\!\!\star}$ to *enlarge* the heap prior to further evaluating a predicate. This suggests a relationship between $\star$ and $\mathrel{-\!\!\star}$ which is similar to the relationship $a + b \leq c$ iff $b \leq c - a$ for addition and subtraction. In fact, $\star$ and $\mathrel{-\!\!\star}$ are adjoint:

Theorem 4.20 (Adjointness [IO01]) For all predicates $P, Q, R \in \mathbf{Pred}$:

$$(P \star Q) \Rightarrow R \qquad \text{iff} \qquad Q \Rightarrow (P \mathrel{-\!\!\star} R).$$

In particular, by considering $(P \mathrel{-\!\!\star} Q) \Rightarrow (P \mathrel{-\!\!\star} Q)$, adjointness clarifies the effect of first cutting off a heap and then inserting the same heap again:

Corollary 4.21 (Modus Ponens [Rey02]) For all predicates $P, Q \in \mathbf{Pred}$:

$$P \star (P \mathrel{-\!\!\star} Q) \Rightarrow Q.$$

Finally, we notice that the predicates $E \mapsto -$, $E \hookrightarrow E'$, and $E \mapsto E_1, \ldots, E_n$ are "syntactic sugar" as they can be expressed in terms of $E \mapsto E'$, $\star$, and existential quantifiers. That is, the following equalities hold:

$$\begin{aligned} E \mapsto - \quad &= \quad \exists x\colon E \mapsto x \\ E \hookrightarrow E' \quad &= \quad E \mapsto E' \star \mathsf{true} \\ E \mapsto E_1, \ldots, E_n \quad &= \quad E \mapsto E_1 \;\star\; E+1 \mapsto E_2 \;\star\; \ldots \;\star\; E+n-1 \mapsto E_n \end{aligned}$$

4.3.3 Fragments of Predicates

Before we move on to reason about programs with separation logic, we study a few fragments of predicates, where separating conjunction and separating implication enjoy additional properties.

4.3.3.1 Pure Predicates

A predicate P is *pure* if it does not depend on the heap, i.e.,

$$\forall \mathfrak{s} \in \mathbf{Stacks}\; \forall \mathfrak{h}, \mathfrak{h}' \in \mathbf{Heaps}\colon \qquad P(\mathfrak{s}, \mathfrak{h}) \;=\; P(\mathfrak{s}, \mathfrak{h}').$$

In particular, all predicates considered in previous chapters, e.g., $z = x \cdot y$, are pure. Since the heap does not matter for pure predicates, the separating connectives coincide with the standard ones for pure predicates.

Theorem 4.22 (Algebraic Laws for $\star$ and $\mathrel{-\!\!\star}$ under Purity [Rey02]) Let P be a pure predicate. Moreover, let Q and R be predicates. Then:

(a) $(P \wedge Q) \;\Rightarrow\; (P \star Q)$,

(b) $(P \wedge Q) \star R \;=\; P \wedge (Q \star R)$, and

(c) $(P \mathrel{-\!\!\star} Q) \;\Rightarrow\; (P \Rightarrow Q)$.

Moreover, if both P and Q are pure, then:

(d) $P \star Q \;=\; P \wedge Q$, and

(e) $P \mathrel{-\!\!\star} Q \;=\; P \Rightarrow Q$.

4.3.3.2 Domain-exact Predicates

A predicate P is *domain-exact* if and only if

$$\forall \mathfrak{s} \in \textbf{Stacks}\ \forall \mathfrak{h}, \mathfrak{h}' \in \textbf{Heaps}\colon \quad (\mathfrak{s}, \mathfrak{h} \models P \text{ and } \mathfrak{s}, \mathfrak{h}' \models P) \text{ implies } \mathsf{dom}(\mathfrak{h}) = \mathsf{dom}(\mathfrak{h}').$$

In other words, after evaluation of variables, there is a fixed set of addresses that must be allocated in the heap in order to satisfy a domain-exact predicate P. For example, the predicates **emp**, $x \mapsto y, z$, and $E \mapsto -$ are all domain-exact whereas the predicates true and $x \hookrightarrow E$ are not. Moreover, if the predicates P and Q are domain-exact, then the predicate $P \star Q$ is domain-exact as well. For domain-exact predicates, the separating conjunction is fully distributive:

Theorem 4.23 (Distributivity Laws for Domain-Exact Predicates [Rey02]) Let $P, Q, R \in \textbf{Pred}$ be predicates and let R be domain-exact. Then:

(a) $(P \wedge Q) \star R = (P \star R) \wedge (Q \star R)$, and

(b) if $x \notin \textbf{Vars}\,(R)$, then $(\forall x\colon P) \star R = \forall x\colon (P \star R)$.

4.3.3.3 Intuitionistic Predicates

A predicate P is *intuitionistic* if and only if

$$\forall \mathfrak{s} \in \textbf{Stacks}\ \forall \mathfrak{h}, \mathfrak{h}' \in \textbf{Heaps}\colon \quad (\mathfrak{h} \subseteq \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h} \models P) \quad \text{implies} \quad \mathfrak{s}, \mathfrak{h}' \models P.$$

In words, as long as the heap grows, an intuitionistic predicate only becomes "more true". For instance, both the predicate true and the predicate $E \hookrightarrow E'$ are intuitionistic. We could have alternatively defined the latter predicate as

$$E \hookrightarrow E' \;\triangleq\; E \mapsto E' \star \mathsf{true}.$$

In fact, there is a general approach to construct intuitionistic predicates:

Theorem 4.24 (Tightest Intuitionistic Predicates [Rey02]) Let $P \in \mathbf{Pred}$ be a predicate. Then the predicate $P \star \mathsf{true}$ is the strongest intuitionistic predicate that is weaker than P. That is,

- $P \star \mathsf{true}$ is intuitionistic,
- $P \Rightarrow (P \star \mathsf{true})$, and
- for all intuitionistic predicates Q with $P \Rightarrow Q$, we have $(P \star \mathsf{true}) \Rightarrow Q$.

Furthermore, the predicate $\mathsf{true} \mathrel{-\!\!\star} P$ is the weakest intuitionistic predicate that is stronger than P. That is,

- $\mathsf{true} \mathrel{-\!\!\star} P$ is intuitionistic,
- $(\mathsf{true} \mathrel{-\!\!\star} P) \Rightarrow P$, and
- for all intuitionistic predicates Q that satisfy $Q \Rightarrow P$, the predicate $Q \Rightarrow (\mathsf{true} \mathrel{-\!\!\star} P)$ holds.

The intuitionistic predicate $E \hookrightarrow E'$ is a compact way to express that the value at address E is given by expression E'. As the following lemma shows, we could have expressed the same without intuitionistic predicates by first cutting off the memory cell $E \mapsto E'$ and then inserting the same cell again.

Lemma 4.25 ([Rey02]) For all expressions E, E' and predicates Q, we have:

$$E \hookrightarrow E' \wedge Q \quad = \quad E \mapsto E' \star (E \mapsto E' \mathrel{-\!\!\star} Q).$$

4.3.4 Recursive Data Structures

A motivation for introducing pointers is to support dynamic data structures, such as linked lists and binary trees. How do we specify data structures systematically in separation logic? For every data structure, we define a predicate $\boldsymbol{P}(\vec{E})$, where $\vec{E}$ is a sequence of parameters passed to the predicate, e.g., the head of a list. These predicates are defined by means of a recursive equation

$$\boldsymbol{P}(\vec{v}) \;=\; \Phi(\boldsymbol{P})(\vec{v}),$$

where, for some natural number $n = |\vec{v}|$, Φ is a monotone function of the form

$$\Phi\colon (\mathbb{Z}^n \to \mathbf{Pred}) \to (\mathbb{Z}^n \to \mathbf{Pred}).$$

The predicate $\boldsymbol{P}(E_1, \ldots, E_n)$ is then given by the least fixed point of Φ, i.e.,

$$\mathfrak{s}, \mathfrak{h} \models \boldsymbol{P}(E_1, \ldots, E_n) \quad \text{iff} \quad \mathfrak{s}, \mathfrak{h} \models \mathsf{lfp}\,(\Phi)\,(E_1\,(\mathfrak{s}), \ldots, E_n\,(\mathfrak{s})).$$

Since Φ is monotone and the set of all functions $\boldsymbol{P}\colon \mathbb{Z}^n \to \mathbf{Pred}$ form a complete lattice with respect to $\Rightarrow$ (applied pointwise), the existence of the least fixed point is guaranteed by the Knaster-Tarski fixed point theorem (Theorem A.11). While it is technically not required, for practical purposes, it is desirable that Φ is not only monotone, but continuous. In this case, the predicate is already captured by the limit of its finite unrollings (cf. Theorem A.16). All recursive equations considered throughout this thesis are also continuous.

Example 4.26 Consider the following recursive equation:

$$\boldsymbol{sll}(u, v) \quad = \quad \underbrace{\left(u = v \wedge \mathbf{emp} \right) \quad \vee \quad \left(\exists u'\colon u \mapsto u' \star \boldsymbol{sll}(u', v) \right)}_{= \Phi(\boldsymbol{sll})(u,v)}$$

This equation defines a predicate $\boldsymbol{sll}(u, v)$ that captures *singly-linked list segments* from address u to address v. Each list element consists only of a pointer to the next element, i.e., the list stores no data. To sharpen our intuition, let us compute the first few steps of the fixed point iteration that determines the precise semantics of predicate $\boldsymbol{sll}(u, v)$. That is, we compute $\Phi^0(\mathsf{false})$, $\Phi^1(\mathsf{false})$, $\Phi^2(\mathsf{false})$, and $\Phi^3(\mathsf{false})$, where $\mathsf{false} \triangleq \lambda(u, v).\,\mathsf{false}$ is the least element of our underlying complete lattice.

For the first step, we have $\Phi^0(\mathsf{false}) = \mathsf{false}$. The next iteration yields

$$\Phi^1(\mathsf{false}) \quad = \quad \lambda(u, v).\, u = v \wedge \mathbf{emp}.$$

Thus, the predicate $\boldsymbol{sll}(u, v)$ is satisfied if the heap is empty and its parameters are identical, i.e., if u is the head of a list segment from u to v of length zero. After one more iteration, we have

$$\begin{aligned}
\Phi^2(\mathsf{false}) \quad &= \quad \lambda(u, v).\, \left(u = v \wedge \mathbf{emp} \right) \\
&\qquad\qquad \vee \quad \left(\exists u'\colon u \mapsto u' \star \Phi^1(\mathsf{false})(u', v) \right) \\
&= \quad \lambda(u, v).\, \left(u = v \wedge \mathbf{emp} \right) \\
&\qquad\qquad \vee \quad \left(\exists u'\colon u \mapsto u' \star (u' = v \wedge \mathbf{emp}) \right) \\
&= \quad \lambda(u, v).\, \left(u = v \wedge \mathbf{emp} \right) \quad \vee \quad u \mapsto v.
\end{aligned}$$

Hence, $\mathit{sll}(u,v)$ is also satisfied if u points to v, i.e., if u is the head of a list segment from u to v of length one. By a similar computation, we obtain

$$\begin{aligned}\Phi^3(\mathsf{false}) \quad = \quad & \lambda(u,v).\ (\ u = v \wedge \mathbf{emp}\ ) \\ & \vee \quad u \mapsto v \\ & \vee \quad \exists u' \colon u \mapsto u' \star u' \mapsto v.\end{aligned}$$

In other words, $\mathit{sll}(u,v)$ is also satisfied if u is the head of a list segment from u to v of length two. In general, the predicate $\mathit{sll}(u,v)$ captures all stack-heap pairs in which u is the head of a list segment from u to v.

As discussed in Section 4.1.4, the organization and usage of dynamic data structures relies—at least in our low-level memory model—heavily on conventions. For example, before we define a predicate that specifies binary trees, we have to agree how we refer to the left and the right child of a node, respectively. If we agree that the first address of a node points to the node's left child and the next address points to the node's right child, then a predicate $\mathit{tree}(u)$ specifying binary trees with root u is given by the equation

$$\mathit{tree}(u) \quad = \quad (\ u = 0 \wedge \mathbf{emp}\ ) \quad \vee \quad (\ \exists v, w \colon u \mapsto v, w \star \mathit{tree}(v) \star \mathit{tree}(w)\ )$$

This definition requires us to keep in mind that address u stores the left child and $u+1$ stores the right child of node u. Let us introduce a more convenient notation to describe members of a data structure element which is similar to the notation in Section 4.1.4 used for implementing data structures in **P³L**: For an a priori fixed sequence of pairwise different *selectors*

$$\mathbf{Sel} \ = \ (\ sel_0, sel_1, \ldots, sel_{n-1}\,),$$

we write $u.sel_i \mapsto v$ to denote the points-to predicate $u + i \mapsto v$. For instance, when dealing with binary trees, we might choose selectors left and right to denote the left and right child of a node, respectively. Hence, we have $\mathbf{Sel} = (\mathsf{left}, \mathsf{right})$. An equivalent definition of the predicate $\mathit{tree}(u)$ is then the following:

$$\begin{aligned}\mathit{tree}(u) \quad = \quad & (\ u = 0 \wedge \mathbf{emp}\ ) \\ & \vee \quad (\ \exists v, w \colon u.\mathsf{left} \mapsto v \star u.\mathsf{right} \mapsto w \star \mathit{tree}(v) \star \mathit{tree}(w)\ )\end{aligned}$$

A binary tree captured by the predicate $\mathit{tree}(u)$ is illustrated in Figure 4.8.

It is noteworthy that there are various notions of recursive predicate definitions in the separation logic literature, e.g., [IO01; Rey02; Bro07; BDP11; IRS13; Ant+14; Bro+14; 2]. A key difference between most of these notions is the underlying definition of heaps. For example, for automated verification approaches, it is often convenient to assume that the heap consists of objects, i.e.,

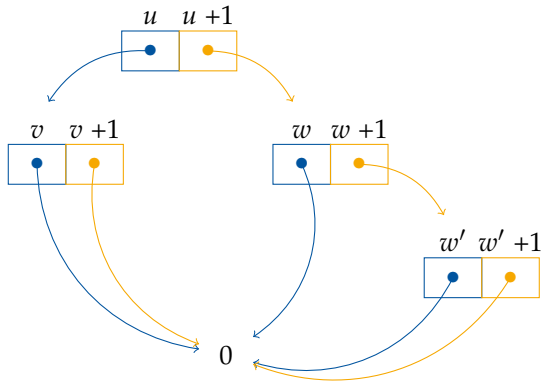


Figure 4.8: Illustration of a heap that (for some stack $\mathfrak{s}$) satisfies $\textbf{\textit{tree}}(u)$.

every address is mapped to a *tuple* of values that make up the members of an object (cf. [IRS13; Bro+14]). In particular, this approach ensures that we always specify all members of an object. Following Reynolds [Rey02], we use a low-level definition of heaps (cf. Definition 4.3) instead. This enables us to access and manipulate individual memory cells. While our memory model subsumes more high-level definitions of heaps, some discipline is required when manipulating objects, i.e., consecutive blocks of memory of a fixed length. In particular, our low-level memory model allows us to specify "partial" data structures, i.e., data structures in which some objects lack a pointer.

Example 4.27 Let us consider two definitions of *doubly-linked* list segments. To this end, we use the selectors $\mathbf{Sel} = (\mathsf{p}, \mathsf{n})$, where p is the *previous* list element and n is the *next* list element, respectively. A *partial* specification of doubly-linked list segments from u to v of length at least two is given by the following equation:

$$\begin{aligned} \textbf{\textit{pdll}}(u, v) \quad = \quad & (\, u.\mathsf{n} \mapsto v \star v.\mathsf{p} \mapsto u \,) \\ \vee \quad & (\, \exists w \colon u.\mathsf{n} \mapsto w \star w.\mathsf{p} \mapsto u \star \textbf{\textit{pdll}}(w, v) \,) \end{aligned}$$

The definition is partial because two memory cells are missing: the pointer to the previous element u' of u and the pointer to the next element v' of v. A complete specification of doubly-linked list segments is then obtained by

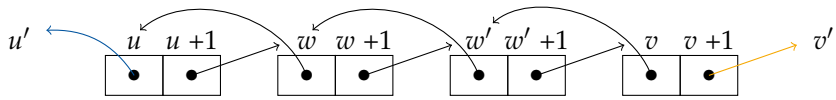


Figure 4.9: Illustration of doubly-linked list segments captured by $\mathit{dll}(u', u, v, v')$.

(1) adding the two missing pointers for lists of length greater or equal to two and (2) adding cases for lists of length zero and length one:

$$\begin{aligned} \mathit{dll}(u', u, v, v') \quad = \quad & \mathbf{emp} \\ & \vee \quad \left(u = v \wedge u.\mathsf{p} \mapsto u' \star u.\mathsf{n} \mapsto v' \right) \\ & \vee \quad \left(u.\mathsf{p} \mapsto u' \star \mathit{pdll}(u, v) \star v.\mathsf{n} \mapsto v' \right) \end{aligned}$$

Figure 4.9 illustrates a doubly-linked list segment that is captured by the above predicate. An alternative specification of doubly-linked list segments that does not rely on a partial specification—and is compatible with more high-level definitions of heaps [BDP11; IRS13]—is found below:

$$\begin{aligned} \mathit{dll}(u', u, v, v') \quad = \quad & \left(u = v' \wedge u' = v \wedge \mathbf{emp} \right) \\ & \vee \quad \left(\exists w \colon u.\mathsf{p} \mapsto u' \star u.\mathsf{n} \mapsto w \star \mathit{dll}(u, w, v, v') \right) \end{aligned}$$

Both definitions only differ for empty list segments: The first specification requires that the heap is empty, whereas the second specification additionally imposes a condition on the parameters.

Reasoning about recursive data structure specifications in separation logic is a challenging task—in particular in the context of automated program verification—that has received much attention in the literature, e.g., [BDP11; IRS13; IRV14; Bro+14; Bro+16; 2]. We consider automated reasoning about (fragments of) recursive data structure specifications in detail in Chapter 11.

4.4 Program Verification with Separation Logic

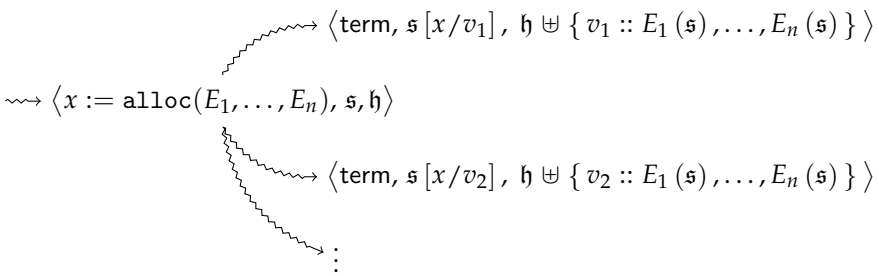
We now turn to Floyd-Hoare style verification of $\mathbf{P^3L}$ programs with separation logic by means of computing weakest preconditions. To this end, we proceed as for **PL** programs (cf. Section 2.3.4). That is, we employ a predicate transformer

$$\mathsf{wp}\,[C] : \mathbf{Pred} \to \mathbf{Pred}$$

which is defined by induction on the structure of $\mathbf{P^3L}$ programs. For all statements that do *not* access the heap, the rules of the resulting weakest precondition calculus are identical to the rules presented for $\mathbf{P^2L}$ programs in Section 3.2.2.

Let us consequently focus on the rules for computing weakest preconditions of the novel statements introduced in $\mathbf{P^3L}$, i.e., allocation $x := \texttt{alloc}(\vec{E})$, deallocation $\texttt{free}(E)$, mutation $<E> := E'$, and lookup $x := <E>$. All of the above statements immediately terminate either successfully or due to a memory fault. Hence, their weakest precondition transformers for total and partial correctness coincide. We thus consider only the case of total correctness. The rules presented in the following correspond to Ishtiaq's and O'Hearn's backward reasoning rules [IO01; Rey02]. A summary of all inference rules for computing both weakest preconditions and weakest liberal preconditions of $\mathbf{P^3L}$ programs is found at the end of this section in Figure 4.11 and Figure 4.12, respectively.

Allocation The program $x := \texttt{alloc}(E_1, \ldots, E_n)$ allocates a consecutive block of n memory cells that are initialized with $E_1, \ldots, E_n$. Moreover, the first address of the allocated block, say v, is stored in variable x. The allocated addresses—in particular the value v—are chosen nondeterministically to abstract from specific implementations of the memory allocator. In our operational semantics, this means that executing an allocation yields infinitely many successor states:



Now, let Q be a postcondition. Since a valid Hoare triple must cover all possible executions, the weakest precondition of $x := \texttt{alloc}(E_1, \ldots, E_n)$ evaluates to true (for the initial stack-heap pair $(\mathfrak{s}, \mathfrak{h})$) if and only if Q evaluates to true for *all* stack-heap pairs $(\mathfrak{s}', \mathfrak{h}')$ reached upon termination. Consequently, the weakest precondition of $x := \texttt{alloc}(\vec{E})$ with respect to Q has to account for all changes of the initial stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ by any execution of the allocation statement. So what are these changes? Once the first address of the newly allocated block of memory, say v, is known, it is stored in variable x. We account for this change as for a standard assignment $x := v$. This leads us to the predicate

$$Q[x/v].$$

In addition to updating variable x, we have to allocate a consecutive block of n memory cells with first address v. The values of these memory cells are determined by the expressions $E_1, \ldots E_n$. In other words, we extend the original heap $\mathfrak{h}$ by the heap $\{ v :: \mathfrak{s}(E_1), \ldots, \mathfrak{s}(E_n) \}$, which is captured by the predicate $v \mapsto E_1, \ldots, E_n$. Moreover, extending a heap before the evaluation of a predicate is precisely what the separating implication does. We thus compute the predicate

$$v \mapsto E_1, \ldots, E_n \mathrel{-\!\!*} Q[x/v].$$

Finally, we have to account for the fact that the first allocated address v is chosen nondeterministically, i.e., we consider all possible values v instead of a fixed one. Hence, the weakest precondition transformer for allocations is defined as

$$\mathsf{wp}[x := \mathtt{alloc}(E_1, \ldots, E_n)](Q) \triangleq \forall v\colon v \mapsto E_1, \ldots, E_n \mathrel{-\!\!*} Q[x/v].$$

Deallocation The program $\mathtt{free}(E)$ attempts to deallocate the memory cell at address E. If address E is not allocated, then we encounter a memory failure. Consequently, there are two mutually exclusive executions:

$$\rightsquigarrow \langle \mathtt{free}(E), \mathfrak{s}, \mathfrak{h} \uplus \{ E(\mathfrak{s}) :: v \} \rangle \rightsquigarrow\!\!\!\rightsquigarrow \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle$$

$$\rightsquigarrow \langle \mathtt{free}(E), \mathfrak{s}, \mathfrak{h} \rangle \rightsquigarrow\!\!\!\!\!\!\rightsquigarrow \langle \mathsf{fault} \rangle$$

The weakest precondition of $\mathtt{free}(E)$ with respect to postcondition Q has to account for the changes of the heap in both cases. So what are these changes? If address E is allocated in the original heap, then the corresponding memory cell, say $\{ E(\mathfrak{s}) :: v \}$, is removed prior to evaluation of postcondition Q. Cutting off a heap before evaluating a predicate in the remaining heap is precisely what the separating conjunction does. Since the value v at address E is irrelevant, the removed heap is captured by the predicate $E \mapsto -$. Hence, the weakest precondition transformer for allocations is defined as

$$\mathsf{wp}[\mathtt{free}(E)](Q) \triangleq E \mapsto - \mathbin{\star} Q.$$

Notice that the predicate $E \mapsto -$ evaluates to false if address E is initially not allocated. Thus, if the deallocation of address E leads to an error, the weakest precondition of $\mathtt{free}(E)$ evaluates to false.

Lookup The program $x := <E>$ attempts to assign the value at address E to variable x. As in previous cases, we encounter a memory failure if address E is not allocated. This means there are two mutually exclusive executions:

$$\rightsquigarrow \left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \uplus \{ E(\mathfrak{s}) :: v \} \right\rangle \rightsquigarrow\!\!\rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \uplus \{ E(\mathfrak{s}) :: v \} \right\rangle$$

$$\rightsquigarrow \left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow \left\langle \mathsf{fault} \right\rangle$$

The weakest precondition of $x := <E>$ with respect to postcondition Q has to account for the changes of the original stack-heap pair in both cases. So what are these changes? The heap is accessed to determine the value at address E, say v, but remains unchanged. The stack is updated such that variable x evaluates to v. This is analogous to the assignment $x := v$, i.e., we compute the predicate

$$Q[x/v].$$

Before updating the stack, however, we have to determine the value v stored at address E. This is conveniently achieved by the intuitionistic predicate $E \hookrightarrow v$. If E is not allocated, this predicate evaluates to false. Otherwise, it evaluates to true if and only if v is the value stored at address E. Consequently, the weakest precondition of a lookup is given by the predicate

$$\exists v\colon E \hookrightarrow v \wedge Q[x/v].$$

By Lemma 4.25, the same predicate can also be expressed with separating connectives only. The weakest precondition of a lookup is then defined as

$$\mathsf{wp}[x := <E>](Q) \triangleq \exists v\colon E \mapsto v \star (E \mapsto v \mathrel{-\!\!\star} Q[x/v]).$$

Mutation The program $<E> := E'$ attempts to set the value stored at address E to E'. If address E is not allocated, then we encounter a memory failure. This means there are two mutually exclusive executions:

$$\rightsquigarrow \left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \uplus \{ E(\mathfrak{s}) :: v \} \right\rangle \rightsquigarrow\!\!\rightsquigarrow \left\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \uplus \{ E(\mathfrak{s}) :: E'(\mathfrak{s}) \} \right\rangle$$

$$\rightsquigarrow \left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \right\rangle \rightsquigarrow\!\!\rightsquigarrow\!\!\rightsquigarrow \left\langle \mathsf{fault} \right\rangle$$

The weakest precondition of $<E> := E'$ with respect to postcondition Q has to account for the changes of the heap in both cases. So what are these changes? Assuming that address E is allocated, we update the value stored at that address. Intuitively, this corresponds to first deallocating the memory cell at address E and then allocating a cell at the same address that stores the updated value. Consequently, we first use a separating conjunction to deallocate the old memory cell, which is captured by the predicate $E \mapsto -$. This leads to the predicate

$$E \mapsto - \star Q.$$

This predicate evaluates to false whenever E is not allocated, i.e., if execution of $<E> := E'$ fails. After removing the original memory cell, we use a separating implication to allocate the updated one, which is captured by the predicate $E \mapsto E'$. Hence, the weakest precondition transformer for mutations is

$$\mathsf{wp}\left[<E> := E'\right](Q) \triangleq E \mapsto - \mathbin{\star} \left(E \mapsto E' \mathbin{-\!\!\star} Q\right).$$

Let us conclude our discussion of rules for computing weakest preconditions of $\mathbf{P^3L}$ programs with a summary of all involved inference rules.

Definition 4.28 (Weakest (Liberal) Precondition Calculus for $\mathbf{P^3L}$[IO01; 1]) The *weakest precondition calculus* wp is defined by structural induction on $\mathbf{P^3L}$ programs according to the rules in Figure 4.11, page 132. Moreover, the *weakest liberal precondition calculus* wlp is defined by structural induction on $\mathbf{P^3L}$ programs according to the rules in Figure 4.12, page 133.

Theorem 4.29 (Soundness of wp and wlp Calculi for $\mathbf{P^3L}$ [IO01; 1]) For every $\mathbf{P^3L}$ program C and every predicate $Q \in \mathbf{Pred}$, we have:

(a) $\mathsf{wp}\,[C]\,(Q) = \sup\left\{ P \mid \left\langle P \right\rangle C \left\langle Q \right\rangle \text{ is valid for total correctness} \right\}$, and

(b) $\mathsf{wlp}\,[C]\,(Q) = \sup\left\{ P \mid \left\langle P \right\rangle C \left\langle Q \right\rangle \text{ is valid for partial correctness} \right\}$.

4.4.1 Proof Rules

In Sections 2.3.5, 2.3.7, 3.2.2 and 3.2.3 we discussed proof rules for weakest precondition reasoning about both total and partial correctness. Which of these rules remain valid in the presence of pointers and dynamic memory allocation?

Let us first consider *monotonicity* of wp and wlp (Lemmas 2.27, 2.33 and 3.16). Monotonicity is essential such that the rules for loops and procedure calls are well-defined: It ensures that the characteristic functions of loops and procedures are themselves monotone and thus—by Theorem A.11—least and greatest fixed points exist. Fortunately, monotonicity is preserved by $\mathbf{P^3L}$ programs.

Lemma 4.30 (Monotonicity) For all $\mathbf{P^3L}$ programs C, the predicate transformers $\mathsf{wp}\,[C]$ and $\mathsf{wlp}\,[C]$ are monotone.

Proof. By induction on the structure of $\mathbf{P^3L}$ programs. □

In particular, this means that the invariant-based proof rules for partial correctness, i.e., Theorem 2.35 and Theorem 3.22, also work for reasoning about $\mathbf{P^3L}$ programs. Unfortunately, this does *not* hold for the proof rule for reasoning

about total correctness presented in Section 2.3.5. The underlying reason is that this rule depends on the continuity of wp (Lemma 2.30). However, the infinite nondeterminism introduced by the allocation statement destroys continuity.

Lemma 4.31 (Non-Continuity [AP86; 1]) There exist $\mathbf{P^3L}$ programs C such that neither $\mathsf{wp}\,[C]$ nor $\mathsf{wlp}\,[C]$ are continuous.

Proof. Consider the set of predicates $\mathbf{X} = \{\, 1 \le x \le n \mid n \in \mathbb{N} \,\}$. Clearly, $\sup \mathbf{X} = (1 \le x)$. Then, for some stack $\mathfrak{s}$ and the empty heap $\mathfrak{h}_\emptyset$, we have

$$\begin{aligned} & \mathfrak{s}, \mathfrak{h}_\emptyset \models \mathsf{wp}\,[x := \mathtt{alloc}(0)]\,(\sup \mathbf{X}) \\ \text{iff} \quad & \mathfrak{s}, \mathfrak{h}_\emptyset \models \forall v\colon v \mapsto 0 \mathrel{-\!\!*} (1 \le x)\,[x/v] \\ \text{iff} \quad & \mathfrak{s}, \mathfrak{h}_\emptyset \models \forall v\colon v \mapsto 0 \mathrel{-\!\!*} \mathsf{true} \\ \text{iff} \quad & \mathsf{true}. \end{aligned}$$

However, if we consider a single predicate $(1 \le x \le n) \in \mathbf{X}$, we have

$$\begin{aligned} & \mathfrak{s}, \mathfrak{h}_\emptyset \models \mathsf{wp}\,[x := \mathtt{alloc}(0)]\,(1 \le x \le n) \\ \text{iff} \quad & \mathfrak{s}, \mathfrak{h}_\emptyset \models \forall v\colon v \mapsto 0 \mathrel{-\!\!*} (1 \le x \le n)\,[x/v] \\ \text{iff} \quad & \mathfrak{s}, \mathfrak{h}_\emptyset \models \forall v\colon v \mapsto 0 \mathrel{-\!\!*} v \le n \\ \text{iff} \quad & \mathsf{false}. \qquad\qquad (\text{choose } v > n) \end{aligned}$$

This yields a contradiction to $\mathsf{wp}\,[x := \mathtt{alloc}(0)]$ being continuous:

$$\begin{aligned} \mathsf{wp}\,[x := \mathtt{alloc}(0)]\,(\sup \mathbf{X})\,(\mathfrak{s}, \mathfrak{h}_\emptyset) &= \mathsf{true}, \text{ but} \\ \sup \{\, \mathsf{wp}\,[x := \mathtt{alloc}(0)]\,(1 \le x \le n)\,(\mathfrak{s}, \mathfrak{h}_\emptyset) \mid n \in \mathbb{N} \,\} &= \mathsf{false}. \end{aligned}$$

By an analogous argument, $\mathsf{wlp}\,[x := \mathtt{alloc}(0)]$ is not continuous. □

For programs that contain no allocation statements, however, continuity is restored. It might thus be tempting to search for an alternative semantics of allocations. However, without severe restrictions, e.g., imposing an upper bound on the available amount of memory, there is little hope to obtain a continuous predicate transformer: Yang and O'Hearn [YO02, Section 4.1] argue that nondeterministic allocation is *essential* to enable local reasoning which was our main motivation for considering separation logic. In particular, they note that always choosing the smallest available address is incompatible with local reasoning, because "adding memory would change what this new address was, and the difference could be detected with address arithmentic". Furthermore, Apt and Plotkin [AP86] showed that it is *impossible* to define a (fully abstract)

continuous least fixed point semantics, such as wp, that exhibits countably infinite nondeterministic assignments.

We summarize other properties that have been considered in Sections 2.3.5, 2.3.7, 3.2.2 and 3.2.3 and that remain correct for $\mathbf{P^3L}$ programs below.

Theorem 4.32 (Compositionality of wp and wlp for $\mathbf{P^3L}$ programs) For every $\mathbf{P^3L}$ program C and all predicates $Q, R \in \mathbf{Pred}$, we have:

(a) Strictness: $\mathsf{wp}\,[C]\,(\mathsf{false}) = \mathsf{false}$.

(b) Conjunction rule: $\mathsf{wp}\,[C]\,(Q \wedge R) \Rightarrow \mathsf{wp}\,[C]\,(Q) \wedge \mathsf{wp}\,[C]\,(R)$.
If C contains no allocation, an equality holds. The same holds for wlp.

(c) Disjunction rule: $\mathsf{wp}\,[C]\,(Q \vee R) \Rightarrow \mathsf{wp}\,[C]\,(Q) \vee \mathsf{wp}\,[C]\,(R)$.
If C contains no allocation, an equality holds. The same holds for wlp.

Proof. By induction on the structure of $\mathbf{P^3L}$ programs. □

Note that—in contrast to $\mathbf{P^2L}$ programs—strictness does *not* hold for weakest liberal preconditions. That is, $\mathsf{wlp}\,[C]\,(\mathsf{true}) = \mathsf{true}$ does not hold for $\mathbf{P^3L}$ programs in general due to the possibility of encountering a memory fault, e.g.,

$$\mathsf{wlp}\,[\mathtt{free}(x)]\,(\mathsf{true}) \;=\; x \mapsto - \star \mathsf{true} \;=\; x \hookrightarrow - \neq \mathsf{true}.$$

4.4.2 Local Reasoning

While the rules of separation logic for computing weakest preconditions are both sound and complete (cf. Theorem 4.29), our main motivation was to recover local reasoning for $\mathbf{P^3L}$ programs. Does separation logic achieve this?

On the level of rules for $\mathbf{P^3L}$ programs, this is the case: Changes of the stack are expressed by syntactic substitutions of the form $Q\,[x/E]$ as it is the case for standard assignments in Hoare logic. Moreover, the heap is accessed and manipulated through syntactic operations: We either cut off memory cells with a separating conjunction $\star$ or we add memory cells to the heap with a separating implication $\mathrel{-\!\!\star}$. In particular, all predicates used to specify which memory cells are added or removed are domain-exact. That is, they capture precisely those memory cells that are actually accessed by a program.

What about aliasing? We have already shown that the separating conjunction prevents aliasing in predicates, i.e., $x \mapsto E \star y \mapsto E'$ implies that $x \neq y$. When computing weakest preconditions, it also rules out aliasing effects that would invalidate a property. For example, recall the $\mathbf{P^3L}$ program

$$<x> := 3\,;\, <y> := 17\,;\, z := <x>.$$

We discussed in Section 4.2 that a naïve approach to compute the weakest precondition of this program with respect to postcondition $z = 3$ fails to account for the possibility that x and y are aliases. As demonstrated below, this is not the case when using separation logic (read from bottom to top as in Example 2.29):

$$
\begin{array}{l}
\text{// } \quad x \mapsto - \mathbin{\ast} y \mapsto - \mathbin{\ast} \mathsf{true} \\
\text{// } \quad \Longleftarrow \quad [\![\text{ algebra (see Appendix B.1 for details) }]\!] \\
\text{// } \quad x \mapsto - \mathbin{\ast} (x \mapsto 3 \mathrel{-\!\!\ast} (y \mapsto - \mathbin{\ast} (y \mapsto 17 \mathrel{-\!\!\ast} x \hookrightarrow 3))) \\
<x> := 3; \\
\text{// } \quad y \mapsto - \mathbin{\ast} (y \mapsto 17 \mathrel{-\!\!\ast} x \hookrightarrow 3) \\
<y> := 17; \\
\text{// } \quad x \hookrightarrow 3 \\
\text{// } \quad \Longleftarrow \quad [\![\text{ elementary predicate logic }]\!] \\
\text{// } \quad \exists v \colon x \hookrightarrow v \wedge v = 3 \\
\text{// } \quad \Longleftarrow \quad [\![\text{ Lemma 4.25 }]\!] \\
\text{// } \quad \exists v \colon x \mapsto v \mathbin{\ast} (x \mapsto v \mathrel{-\!\!\ast} v = 3) \\
z := <x> \\
\text{// } \quad z = 3
\end{array}
$$

Our computed precondition thus states that the above program terminates with $z = 3$ if both x and y store allocated addresses which are not aliases.

There is one more issue faced when reasoning about pointer programs that we discussed in Section 4.2: The rules of invariance and constancy, which embody local reasoning, are unsound. Before we discuss both rules in the context of separation logic, we extend the inductive definition of the set $\mathbf{Mod}(C)$ of variables modified by program C to cover $\mathbf{P^3L}$ programs:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathbf{Mod}\left(x := \mathtt{alloc}(\vec{E})\right) & \triangleq & \{x\} & \mathbf{Mod}\left(\mathtt{free}(E)\right) & \triangleq & \emptyset \\
\mathbf{Mod}\left(x := <E>\right) & \triangleq & \{x\} & \mathbf{Mod}\left(<E> := E'\right) & \triangleq & \emptyset
\end{array}
$$

Notice that deallocation and mutation affect the heap, but not the stack. A full definition of $\mathbf{Mod}(C)$ for $\mathbf{P^3L}$ programs is found in Figure 4.13, page 134.

With this notion at hand, O'Hearn showed that the rule of constancy, which embodies local reasoning (cf. Section 4.2), can be recovered in separation logic as long as we use separation conjunctions rather than standard conjunctions:

Theorem 4.33 (Frame Rule [IO01; Yan01; YO02]) Let C be a $\mathbf{P^3L}$ program and R be a predicate such that $\mathbf{Mod}(C) \cap \mathbf{Vars}(R) = \emptyset$. Then:

(a) $\mathsf{wp}[C](Q) \star R \Rightarrow \mathsf{wp}[C](Q \star R)$, and

(b) $\mathsf{wlp}[C](Q) \star R \Rightarrow \mathsf{wlp}[C](Q \star R)$.

Proof. By induction on the structure of $\mathbf{P^3L}$ programs. □

A reader familiar with separation logic might know the frame rule in a different form as it is usually presented as an inference rule for Hoare logic:

$$\frac{\langle P \rangle\ C\ \langle Q \rangle \qquad \mathbf{Mod}(C) \cap \mathbf{Vars}(R) = \emptyset}{\langle P \star R \rangle\ C\ \langle Q \star R \rangle}\ \text{frame}$$

Let us thus briefly verify that both versions are equivalent for total correctness. The proof for partial correctness is analogous. To this end, let C be a $\mathbf{P^3L}$ program and R be a predicate such that $\mathbf{Mod}(C) \cap \mathbf{Vars}(R) = \emptyset$. Moreover, recall that the triple $\langle P \rangle\ C\ \langle Q \rangle$ is valid for total correctness if and only if $P \Rightarrow \mathsf{wp}[C](Q)$ holds. Then the above inference rule is equivalent to

$$(P \quad \Rightarrow \quad \mathsf{wp}[C](Q)) \qquad \Rightarrow \qquad (P \star R \quad \Rightarrow \quad \mathsf{wp}[C](Q \star R))\,.$$

For $P = \mathsf{wp}[C](Q)$, the above statement yields

$$\underbrace{(\mathsf{wp}[C](Q) \Rightarrow \mathsf{wp}[C](Q))}_{=\ \mathsf{true}} \qquad \Rightarrow \qquad (\mathsf{wp}[C](Q) \star R \quad \Rightarrow \quad \mathsf{wp}[C](Q \star R))\,,$$

which is logically equivalent to the statement in Theorem 4.33, i.e.,

$$\mathsf{wp}[C](Q) \star R \quad \Rightarrow \quad \mathsf{wp}[C](Q \star R)\,.$$

Conversely, assume that $P \Rightarrow \mathsf{wp}[C](Q)$ holds. Then:

$$\begin{array}{lll} & P \star R & \\ \Rightarrow & \mathsf{wp}[C](Q) \star R & \text{(Monotonicity of } \star\text{, see Theorem 4.18)} \\ \Rightarrow & \mathsf{wp}[C](Q \star R)\,. & \text{(Theorem 4.33)} \end{array}$$

Consequently, $P \Rightarrow \mathsf{wp}[C](Q)$ implies $P \star R \Rightarrow \mathsf{wp}[C](Q \star R)$. Both formulations of the frame rule are thus equivalent.

We conclude this chapter with an example that demonstrates the usefulness of the frame rule for program verification.

Example 4.34 Let us consider a classical example by O'Hearn [OHe12, p. 4–5]. The $\mathbf{P^3L}$ procedure delete(x), which is depicted in Figure 4.10, takes a binary tree with root x as an input. If the tree is not empty, i.e., x is not 0, then the procedure first deletes both subtrees of x and finally the root of the tree itself. Our goal is to prove that delete(x) successfully deletes the whole tree, i.e., the Hoare triple

$$\left\langle\, \textit{tree}(x) \,\right\rangle\ y := \text{delete}(x)\ \left\langle\, \mathbf{emp} \,\right\rangle$$

is valid for partial correctness, where—as discussed in Section 4.3.4—the predicate $\textit{tree}(x)$, which specifies binary trees with root x, is given by

$$\begin{aligned} \textit{tree}(u) \quad &= \quad (\ u = 0 \wedge \mathbf{emp}\ ) \\ &\quad\ \vee \quad (\ \exists v, w\colon u \mapsto v, w \star \textit{tree}(v) \star \textit{tree}(w)\ )\,. \end{aligned}$$

To deal with recursive procedure calls, we show that

$$\begin{aligned} \textit{tree}(E) \Rightarrow \quad & \mathsf{wlp}\,[\mathsf{body}\,(\text{delete}(x))]\,(\mathbf{emp}\,[-\boxdot]\,[\boxdot z/\mathsf{out}])\,[x/x\,[-\boxdot]]\,[+\boxdot] \\ = \quad & \mathsf{wlp}\,[\mathsf{body}\,(\text{delete}(x))]\,(\mathbf{emp})\,[x/x\,[-\boxdot]]\,[+\boxdot] \end{aligned}$$

holds under the assumption that, for all E and z,

$$\textit{tree}(E) \Rightarrow \mathsf{wlp}\,[z := \text{delete}(E)]\,(\mathbf{emp})\,.$$

Theorem 3.22 then yields

$$\textit{tree}(x) \Rightarrow \mathsf{wlp}\,[y := \text{delete}(x)]\,(\mathbf{emp})\,.$$

Hence, the Hoare triple $\left\langle\, \textit{tree}(x) \,\right\rangle\ y := \text{delete}(x)\ \left\langle\, \mathbf{emp} \,\right\rangle$ is valid for partial correctness. The essential steps of the proof are provided in Figure 4.10. In particular, notice that we apply the frame rule prior to the two recursive calls. This is crucial in order to obtain a postcondition that allows us to apply our assumption for procedure calls. For instance, for the recursive call $\mathsf{out} := \text{delete}(\mathsf{right})$, we have

$$\begin{aligned} & \mathsf{wlp}\,[\mathsf{out} := \text{delete}(\mathsf{right})]\,(x \mapsto - \star x+1 \mapsto - \star \mathbf{emp}) \\ \Leftarrow\ & \mathsf{wlp}\,[\mathsf{out} := \text{delete}(\mathsf{right})]\,(\mathbf{emp}) \star x \mapsto - \star x+1 \mapsto - && \text{(frame rule)} \\ \Leftarrow\ & \textit{tree}(\mathsf{right}) \star x \mapsto - \star x+1 \mapsto - && \text{(assumption)} \\ =\ & x \mapsto - \star x+1 \mapsto - \star \textit{tree}(\mathsf{right}) \star \mathbf{emp}. && \text{(Theorem 4.17)} \end{aligned}$$

As a final remark, we observe that the most difficult aspects of writing proofs in separation logic are twofold: First, as usual in Hoare logic, we have to find suitable invariants for loops and procedures. Second, reasoning about *entailments* between predicates, i.e., proving that $P \Rightarrow Q$ holds for two predicates P and Q, is more complicated (or at least less standard) if P and Q may contain separating connectives and recursive predicates that specify data structures. In fact, proving entailments has received a lot of attention in the separation logic literature, e.g., [BCO04; BDP11; Chi+12; IRS13; IRV14; Ant+14; LP14; Ta+16; Ene+17a; Le+17; Ta+18]. We will consider the entailment problem in more detail in Chapter 12.

```
// tree(x)
// ⟹ ⟦ By definition of scoping (Definition 3.6) ⟧
// tree(⧈x) [+⧈]
delete(x) {
    // tree(x)
    // ⟹ ⟦ By definition of tree(x), emp ⇒ tree(x); elementary predicate logic ⟧
    // (x ≠ 0 ∧ tree(x)) ∨ (x = 0 ∧ emp)
    if ( x ≠ 0 ) {
        // tree(x)
        // ⟹ ⟦ as in previous step; elementary logic; definition of tree(x) ⟧
        // ∃u: x ↪ u ∧ ∃v: x ↦ − ⋆ x + 1 ↦ v ⋆ tree(v) ⋆ tree(u)
        left := <x>;
        // ∃v: x ↦ − ⋆ x + 1 ↦ v ⋆ tree(v) ⋆ tree(left)
        // ⟹ ⟦ using E ↦ E′ ⇒ E ↪ E′ ∧ E ↦ − ⟧
        // ∃v: x + 1 ↪ v ∧ x ↦ − ⋆ x + 1 ↦ − ⋆ tree(v) ⋆ tree(left)
        right := <x + 1>;
        // x ↦ − ⋆ x + 1 ↦ − ⋆ tree(right) ⋆ tree(left)
        void := delete(left);
        // x ↦ − ⋆ x + 1 ↦ − ⋆ tree(right) ⋆ emp
        // ⟹ ⟦ Theorem 4.17 ⟧
        // x ↦ − ⋆ x + 1 ↦ − ⋆ tree(right)
        void := delete(right);
        // x ↦ − ⋆ x + 1 ↦ − ⋆ emp
        free(x);
        // x + 1 ↦ − ⋆ emp
        free(x + 1)
        // emp
    } else { // emp
        skip
        // emp
    }; // emp
    out := 0
    // emp
} // emp
```

Figure 4.10: Correctness proof of a tree deletion procedure.

C	$\mathsf{wp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$x :=$ `alloc`$(E_1, \ldots, E_n)$	$\forall v\colon v \mapsto E_1, \ldots, E_n \mathrel{-\!\!\star} Q\,[x/v]$
`free`(E)	$E \mapsto - \star Q$
$x :=\, <E>$	$\exists v\colon E \mapsto v \star (E \mapsto v \mathrel{-\!\!\star} Q\,[x/v])$
$<E> := E'$	$E \mapsto - \star (E \mapsto E' \mathrel{-\!\!\star} Q)$
$C_1\,;\, C_2$	$\mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(Q))$
`if` $(\,B\,)$ `{` C_1 `}` `else` `{` C_2 `}`	$(B \wedge \mathsf{wp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wp}\,[C_2]\,(Q))$
`while` $(\,B\,)$ `{` C' `}`	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda I.\ (B \wedge \mathsf{wp}\,[C']\,(I)) \vee (\neg B \wedge Q)$
$x := F(E_1, \ldots, E_n)$	$\mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \ldots, E_n)(Q)$, where
$\mathfrak{P}_F \triangleq \lambda\theta\lambda x'\lambda(E'_1, \ldots, E'_n)\lambda Q'.$	
$\mathsf{wp}^F_\theta\,[\mathsf{body}\,(F)]\,\big(Q'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,[x_1/E'_1\,[-\boxdot]]\ldots[x_n/E'_n\,[-\boxdot]]\,[+\boxdot]$	

Figure 4.11: Rules of the weakest precondition calculus for **P³L** programs. The auxiliary transformer wp^F_θ is found in Figure 3.4, page 77. It is extended to cover the novel **P³L** statements using the same rules as above.

C	$\mathsf{wlp}\,[C]\,(Q)$
`skip`	Q
$x := E$	$Q\,[x/E]$
$x := \texttt{alloc}(E_1, \ldots, E_n)$	$\forall v\colon v \mapsto E_1, \ldots, E_n \mathrel{-\!\!*} Q\,[x/v]$
$\texttt{free}(E)$	$E \mapsto - \mathbin{*} Q$
$x := <E>$	$\exists v\colon E \mapsto v \mathbin{*} (E \mapsto v \mathrel{-\!\!*} Q\,[x/v])$
$<E> := E'$	$E \mapsto - \mathbin{*} (E \mapsto E' \mathrel{-\!\!*} Q)$
$C_1\,;\,C_2$	$\mathsf{wlp}\,[C_1]\,(\mathsf{wlp}\,[C_2]\,(Q))$
$\texttt{if}\ (B)\ \{C_1\}\ \texttt{else}\ \{C_2\}$	$(B \wedge \mathsf{wlp}\,[C_1]\,(Q)) \vee (\neg B \wedge \mathsf{wlp}\,[C_2]\,(Q))$
$\texttt{while}\,(B)\,\{C'\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where
	$\mathfrak{LW} \triangleq \lambda I.\ (B \wedge \mathsf{wlp}\,[C']\,(I)) \vee (\neg B \wedge Q)$
$x := F(E_1, \ldots, E_n)$	$\mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(E_1, \ldots, E_n)(Q)$, where
$\mathfrak{LP}_F \triangleq \lambda\theta\lambda x'\lambda(E'_1, \ldots, E'_n)\lambda Q'.$	
$\mathsf{wlp}^F_\theta\,[\mathsf{body}\,(F)]\,(Q'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}])\,[x_1/E'_1\,[-\boxdot]]\ldots[x_n/E'_n\,[-\boxdot]]\,[+\boxdot]$	

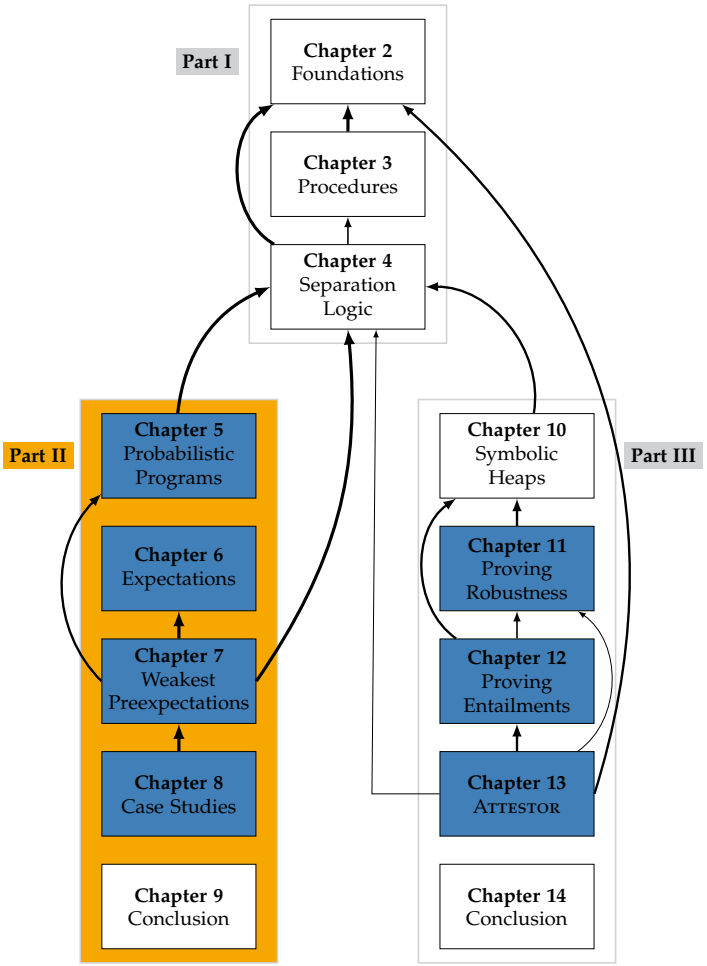
Figure 4.12: Rules of the weakest liberal precondition calculus for $\mathbf{P^3L}$ programs. The auxiliary transformer wlp^F_θ is found in Figure 3.7, page 85. It is extended to cover the novel $\mathbf{P^3L}$ statements using the same rules as above.

C	$\mathbf{Mod}(C)$
`skip`	$\emptyset$
$x := E$	$\{ x \}$
$x :=$ `alloc`$(E_1, \ldots, E_n)$	$\{ x \}$
`free`(E)	$\emptyset$
$x := <E>$	$\{ x \}$
$<E> := E'$	$\emptyset$
$C_1 \,;\, C_2$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`if` (B) $\{ C_1 \}$ `else` $\{ C_2 \}$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`while` (B) $\{ C' \}$	$\mathbf{Mod}(C')$
$x := F(E_1, \ldots, E_n)$	$\{ x \}$

Figure 4.13: Definition of the set of variables modified by a $\mathbf{P^3L}$ program.

Part II

Randomization in Separation Logic



This chapter is based on prior publications, namely [1; 18; 10; 6], which are presented, discussed, and extended in the broader context of this thesis.

Probabilistic programs are ordinary programs with the ability to sample values from a probability distribution (cf. [Gor+14b; Kat+15]). Since their execution may depend on sampled values, their behavior can be summarized as follows:

When running a probabilistic program repeatedly on a given input, we obtain a probability distribution of outputs.

As a first example of probabilistic programs, consider the procedure geometric in Figure 5.1 (next page). This program flips a fair coin that yields either heads (0) or tails (1). If the coin flip yields heads, the procedure returns zero. Otherwise, it returns one plus the result of a recursive call. When running geometric repeatedly, its return value is distributed as illustrated in Figure 5.2.

The above summary describes the behavior of probabilistic programs from the perspective of running them on an actual machine. This is why we stress the word repeatedly: Running a probabilistic program once on an input yields one result whose probability can be quantified. From a program analysis perspective, however, the following characterization is more convenient:

For every given input, a probabilistic program computes a probability distribution of executions (and thus also a probability distribution of outputs).

Intuitively, we may thus think of the distribution sketched in Figure 5.2 as the output of procedure geometric.

Probabilistic programs inject *randomization* into computations. Randomization is a fundamental concept in computer science. It is an important tool for the design and analysis of efficient algorithms [MR97; Cor+09]. Further applications include approximate computing, security, and artificial intelligence.

Before we have a closer look at applications of probabilistic programs, we observe that—since the probability of executions can be quantified—probabilistic programs offer a more fine-grained view on a program's behavior than arbitrary

```
geometric() {
    coin := flip();
    if (coin = 0) {
        out := 0
    } else {
        x := geometric();
        out := 1 + x
    }
}
```

Figure 5.1: A recursive probabilistic program.

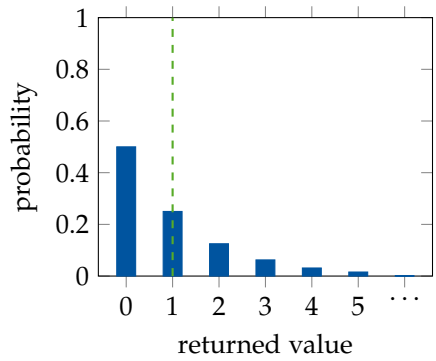


Figure 5.2: The probability distribution computed by geometric.

nondeterministic programs. A nondeterministic program chooses one out of potentially infinitely many possible executions for a given input. There is, however, no way to determine the likelihood of any execution. In fact, it is perfectly conceivable that running a nondeterministic program repeatedly on the same input always results in the same execution.

In contrast, Figure 5.2 indicates that repeatedly flipping a fair coin and always seeing tails is highly unlikely. Understanding probabilistic programs thus requires us to consider all possible executions *and* take their probabilities into account. There are two approaches:

1. We attempt to determine the precise probability distribution over all executions of a probabilistic program for a given input. Further properties of interest, such as the probability of terminating with variable x being positive or the probability that the program's runtime is linear in the size of the input, can then be derived from this distribution. Intuitively, this means we have to compute the plot in Figure 5.2 to understand the program in Figure 5.1. Broadly speaking, this approach is taken by, amongst others, Kozen [Koz81], Karp [Kar91; Kar94], Monniaux [Mon01], den Hartog and de Vink [HV02], and Tassarotti and Harper [TH18].

2. A less precise, but often more convenient, approach is to study a probabilistic program's *expected* behavior. That is, we consider the mean over all executions weighted according to the probability of each execution. For instance, the expected return value of the program in Figure 5.1 is one as indicated by the dashed green line in the plot.

A common expected value of interest is the *expected runtime*: It is defined as the weighted average $\sum_i p_i \cdot t_i$ over all executions, where p_i is the probability of an execution and t_i is its time until termination. This approach is taken, for example, by Kozen [Koz83; Koz85], Jones [Jon90], McIver & Morgan [MMS96; MM05], Chatterjee et al. [CNZ17; CF17; Cha+18], Ngo et al. [NCH18], Kaminski [Kam19] as well as in [10; 15; 16; 6; 1].

Outline of Part II The second part of this thesis is concerned with formal verification of probabilistic pointer programs by means of a *quantitative separation logic* (QSL for short). To this end, we take the second approach to understanding probabilistic programs from above. That is, we extend the techniques of McIver & Morgan [MM05] to reason about expected values of probabilistic pointer programs in a weakest precondition style calculus. We show that both the assertion language as well as the verification system of QSL is a conservative extension of classical separation logic as introduced in Chapter 4.

The remainder of this part is structured as follows. Section 5.1 motivates the need for reasoning about probabilistic *pointer* programs. Related work is discussed in Section 5.2. In Section 5.3, we present the programming language **P^{4}L**—a probabilistic extension of programming languages considered in Part I. The assertion language of QSL is introduced in Chapter 6. The corresponding verification system is developed in Chapter 7. Chapter 8 applies QSL to several case studies. Finally, Chapter 9 concludes and discusses future work.

5.1 Why Probabilistic Pointer Programs?

In this section, we briefly walk through a few applications of probabilistic programs. Furthermore, in Section 5.1.2, we argue why reasoning about probabilistic pointer programs is desirable.

5.1.1 Applications of Probabilistic Programs

Let us have a closer look at one classical application of probabilistic programs—randomized algorithms—and two more recently emerging fields: probabilistic programming, which is a discipline in statistics and machine learning, and approximate computing, which poses novel challenges for program verification.

Randomized Algorithms Injecting randomization into an algorithm often enables significant speed-ups—in expectation—over deterministic approaches. These speed-ups are usually obtained with rather little implementation effort. Both observations motivate the study of randomized algorithms—a field that dates back to 1976: In his seminal paper, Rabin [Rab76] proposed an approach

to solve the closest-pair problem in computational geometry in *expected linear time* whereas a naïve deterministic algorithm requires quadratic time. Although by a smaller margin, Rabin's randomized algorithm still outperforms the best known deterministic solution of the closest-pair problem [FH79; Cla83] which runs in $\mathcal{O}(n \cdot \log \log n)$ in the same computation model.

Randomized algorithms are implemented by probabilistic programs. They can be roughly divided into two flavors (cf. [MR97, Chapter 1.2]):

A *Las Vegas algorithm* always computes the correct solution, but its runtime may vary due to sampling from a probability distribution. A prime example is Hoare's randomized quicksort [Hoa62]. Randomized quicksort is identical to standard quicksort except that a single line is changed: The pivot element is chosen at random. Since quicksort computes the correct result for any chosen pivot element, this change does not affect the algorithm's correctness. It does, however, have significant consequences on the runtime: While quicksort with a deterministic pivot selection has a worst-case runtime of $\mathcal{O}(n^2)$, randomized quicksort has a worst-case *expected* runtime of $\mathcal{O}(n \cdot \log n)$.

In contrast, a *Monte Carlo algorithm* may produce an *incorrect* result with some probability, but its runtime does (usually) not vary. A prominent example is Freivalds' approach to solve the matrix product verification problem: Given three $n \times n$ matrices A B, and C, does $A \cdot B = C$ hold? A naïve solution of this problem would compute the product of A and B and compare the result with C. Using the best known algorithm for matrix multiplication, this approach leads to a worst-case runtime of $O(n^{2.3729})$ (cf. [Wil12]). In contrast, Freivalds' randomized algorithm requires only quadratic time, but may yield the wrong result: For every constant $k \in \mathbb{N}_{>0}$, his algorithm runs in $O(k \cdot n^2)$ but produces a wrong result with probability at most 2^{-k} [Fre79].

Probabilistic Programming Probabilistic programs are a powerful modeling formalism that is not limited to randomized algorithms. For example, there is a large number of probabilistic programming languages, such as Figaro [Pfe09], WebPPL [GS14], Tabular [Gor+14a], Anglican [WMM14], and Stan [Car+17], which are designed for applications in machine learning and artificial intelligence. Programs written in these languages are *not* intended to be executed in the classical sense; rather, they serve as probabilistic models.

There is a vast amount of literature on classical probabilistic graphical models (cf. [Bis07; KF09; Dar09]), such as Bayesian networks, Hidden Markov Models, etc. For example, since their invention by Pearl in 1985 [Pea85], Bayesian networks have been intensively studied and applied in a wide range of domains including machine learning [Hec08], speech recognition [ZR98], sports betting [CFN12], medical diagnosis [JC10], and finance [NJ10]. Why are probabilistic programs preferable to such well-established models? Given a Bayesian

network, *the* problem of interest is *probabilistic inference*, i.e., determine the probability of an event given observed evidence. This problem has been studied for over three decades. In particular, it has been shown that both exact as well as approximate inference to arbitrary precision are NP-hard [Coo90; DL93]. For a long time, it was thus considered folklore that inference algorithms must be manually tailored to individual probabilistic models.

A key feature of probabilistic programming languages is that inference algorithms can be selected and optimized automatically at compile time (cf. [MW08; Gor+14b; MW19]). Hence, the implementation of inference techniques is decoupled from writing probabilistic models. This obviates the need to manually tweak inference techniques while still achieving reasonably well performance. Consequently, probabilistic programming languages enable rapid prototyping. Furthermore, engineers need less theoretical background to apply probabilistic modeling to a concrete problem; rather, it suffices that experts focus on optimizing general purpose inference algorithms. Probabilistic programming languages are thus more accessible to practitioners.

Finally, probabilistic programming languages are not limited to Bayesian networks. In fact, there is a trend towards ever more expressiveness. For example, Edward [Tra+16; Tra+17] supports neural networks and implicit generative models. Moreover, Pyro [Bin+19] is a "universal" probabilistic programming language that can represent any computable probability distribution.

Approximate Computing A recent application of probabilistic programs lies in *approximate computing*, i.e., understanding programs where some instructions occasionally produce incorrect results. One possible reason for such behavior is that programs are running on unreliable hardware [CMR16]. Unreliable statements are naturally modeled by probability distributions that govern how often and to what degree noisy results instead of correct ones are returned.

Apart from economic considerations to use cheaper and less reliable hardware, it might seem undesirable to expose programs to unreliable hardware. There is, however, a trend to reduce power usage on mobile devices whenever possible. This might lead to occasional computation errors but is deemed acceptable for some applications, such as video decoding [Dor+19].

Approximate computing also poses a challenge for program verification: Since incorrect results are *unavoidable* when running a program on unreliable hardware, the notion of a program's correctness becomes blurred: That is, quantifying—and subsequently minimizing—the probability of encountering a failure or the expected error of a program becomes paramount. The need for quantitative reasoning is also stressed by Henzinger [Hen13] who argues (more generally for hybrid systems) that:

"the Boolean partition of software into correct and incorrect programs falls short of the practical need to asses the behavior of software in a more nuanced fashion [...]."

5.1.2 Reasoning about Probabilistic Pointer Programs

Formal verification of probabilistic programs is desirable due to their manifold—and occasionally safety-critical—applications. Furthermore, as argued at the end of the previous section, verification techniques must be quantitative in nature: Rather than proving that a specification is always satisfied, we need to reason about both the probability that a specification holds and the expected behavior of a program. Suitable program logics have been studied since 1983 (cf. [Koz83]) for probabilistic versions of simple programming languages that are similar to the language **PL** considered in Chapter 2.

Why, however, is there a need to study probabilistic *pointer* programs?

Let us briefly answer this question by revisiting the applications listed in the previous section in reverse order:

Approximate computing is typically concerned with rather low-level programs or software for embedded systems, e.g., device drivers or video decoding. Due to the prevalence of the C programming languages in systems-level code, pointers are a commonly encountered feature in such software. For example, device drivers contain many intricate list-like structures (cf. [Ber+07]).

A reader with a background in *probabilistic programming* might be particularly skeptical: If probabilistic programs serve as a concise formalism for probabilistic models, should they support complicated features such as pointers, dynamic data structures, and memory allocation? Since there is an ever-growing trend towards more expressive probabilistic programming languages—Microsoft's Infer.net [MW19] started as a convenient interface for Bayesian networks whereas Uber's Pyro [Bin+19] is a "universal probabilistic programming language"—it is conceivable that these features will eventually be supported. In fact, in 2014, Ruttenberg and Pfeffer [RP14] stated that "decisions are often made based on complex data structures, such as social networks and protein sequences, and rich processes involving those structures." They consequently studied probabilistic models described by programs with "complex data structures and control flow".

Finally, data structures are omnipresent in *randomized algorithms*: The very first chapter on applications of randomized algorithms of Motwani and Raghavan's classical textbook [MR97, Chapter 8] is called "Data Structures". It presents randomized algorithms processing arrays, trees, graphs, etc. Similarly, all examples of randomized algorithms considered previously in this chapter operate on arrays or linked data structures.

In summary, there are plenty of applications that rely on pointers, data structures *and* sampling from a probability distribution. This justifies our study of probabilistic pointer programs.

5.2 Related Work

Throughout Part II, we develop a *quantitative separation logic* [1], QSL for short, that enables formal verification of programs that employ both probabilistic sampling and dynamic data structures implemented via pointers. Let us briefly discuss work related to QSL. Further references will be given where appropriate.

Technically, QSL is a marriage between classical separation logic—as presented in the seminal papers by Ishtiaq, O'Hearn, and Reynolds [IO01; Rey02]—and weakest preexpectations à la McIver & Morgan [MMS96; MM05]. We thus first consider these two aspects in isolation.

Classical Separation Logic A detailed discussion of separation logic is found in Chapter 4. Regarding its influence on QSL, our rules for computing the weakest preexpectation of heap manipulating statements coincide with the backward reasoning rules of Ishtiaq and O'Hearn [IO01]. However, all underlying logical connectives are interpreted differently, i.e., in a quantitative setting. Furthermore, QSL's assertion language is designed to broadly adhere to the same algebraic properties, which have been collected by Reynolds [Rey02], as separation logic.

Jones [Jon90] demonstrated that reasoning about expected values of probabilistic programs in a strongest postcondition style is not possible. Hence, we stick to a weakest precondition style calculus. As pointed out by O'Hearn [OHe19], program verification with separation logic is rarely performed with weakest preconditions due to the need for using separating implications. An exception is work by Krebbers et al. [Kre+17] which uses weakest preconditions internally to formalize (concurrent) separation logics. Brotherston et al. [BBC08] incorporate the separating implication in a proof system for verifying program termination. However, they rely on an oracle to discharge implications between formulas.

Weakest Preexpectations Seminal work on formal reasoning about expected values of probabilistic programs is due to Kozen [Koz85]. McIver & Morgan [MMS96; MM05] built on his approach and domain theoretic foundations by Jones [Jon90] to reason about both probabilistic and nondeterministic statements within a single weakest precondition style calculus. They also coined the term "*expectation*" for random variables, which take the role of predicates in classical Floyd-Hoare style program verification. Hence, their verification system is known as the weakest *preexpectation* calculus.

If we disregard heap manipulating statements, QSL essentially coincides with weakest preexpectations. However, the underlying domain is different in two aspects: First, we allow for *unbounded* random variables; our domain is thus a complete lattice. Second, we incorporate a heap component into expectations.

The first soundness proof of weakest preexpectations with respect to an operational semantics is due to Gretz, Katoen, and McIver [GKM14; Gre16]. Similar to their approach, we show that QSL's verification system is sound with respect to an operational semantics that is formalized as a Markov decision process. However, their approach heavily relies on (1) continuity of weakest preexpectations and (2) that the involved Markov decision processes are finitely branching. These two properties allow reasoning about loops in terms of their finite unfoldings. Since continuity breaks for QSL (cf. Section 4.4) and our operational model requires infinite branching in the presence of dynamic memory allocation (cf. Section 4.1.3), our proof strategy is different: Similarly to Apt and Plotkin's [AP86] soundness proofs for programs with countable nondeterminism, we show that both weakest preexpectations and the operational semantics can be described as the solution of a system of Bellman equations. After that, we verify that both solutions coincide.

Verification of Randomized Algorithms Although various algorithms rely on randomized data structures, formal reasoning about probabilistic programs that mutate memory has—at least until 2018—received scarce attention. Apart from QSL, there are several recent works addressing this task:

In [15], the expected runtime of a (recursive) randomized binary search is analyzed by means of a weakest-precondition style calculus (cf. [10; 6]). However, the involved data structures are addressed in an ad-hoc fashion; they are not formalized in the underlying program logic.

Chatterjee, Fu, and Murhekar [CFM17] developed a method for solving recurrence relations that often appear in the analysis of expected runtimes of randomized algorithms, e.g., randomized quicksort or the Coupon's collector problem (cf. [10]). However, similar to most textbook proofs, they do not consider how these recurrences are formally derived from a given program.

Eberl, Haslbeck, and Nipkow [EHN18] presented a case study that verifies properties of various probabilistic algorithms, such as the expected number of comparisons in randomized quicksort, in the theorem prover Isabelle/HOL. They express randomized algorithms in an existing formalization of probability theory (cf. [EHN15]) based on the Giry monad [FP10]. Their approach is not based on a program logic, such as Hoare logic or weakest preexpectations. Rather than reasoning directly on the program structure, some of their proofs involve gradually refining an encoded program and then formalizing textbook-style correctness proofs. Memory safety is not considered.

Tassarotti and Harper [TH19] provided a Coq formalization of a separation logic—called Polaris—to reason about randomized concurrent programs. Their work is most closely related to QSL and has been developed simultaneously and independently.[1] Apart from concurrency, which is not supported by QSL, there are a few notable differences:

First, Polaris is a combination of concurrent separation logic and probabilistic *relational* Hoare logic [BGB12]. Verification is thus understood as establishing a relation between a program to be analyzed and a program that is known to be well-behaved. In contrast, QSL attempts to directly measure a quantitative program property on the source code of a single given program.

Second, Polaris requires programs to *certainly* terminate. That is, there must be an exact number of steps after which every execution on a given initial state halts. Hence, almost-surely terminating programs, i.e., programs that admit infinite executions, but only with probability zero, are outside of the scope of their approach; at least outside of their current soundness results (cf. [TH19, Theorem 3.1]). QSL requires no such restriction. In fact, QSL can be used to compute the exact probability that a program terminates. It thus allows proving almost-sure termination.

Third, Polaris is limited to *bounded* expectations. Hence, it is unclear, whether the approach of Tassarotti and Harper can be extended to reason about expected values in general. It is noteworthy that this includes reasoning about expected runtimes because probabilistic programs might terminate with probability one, but have an unbounded expected runtime (cf. [10]).

Quantitative Aspects of Separation Logic Quantitative aspects of separation logic have been considered independently of randomization. In this case, "quantitative" is understood as reasoning about properties of recursively defined data structures, e.g., the length of linked lists, the height of trees, etc. For example, Bozga, Iosif, and Perarnau [BIP10] presented a decidable fragment of separation logic that contains predicates for singly-linked lists of a specified length. Furthermore, Chin et al. [Chi+12] applied fold/unfold reasoning to automatically verify shape-numeric properties, such as the balancedness of binary trees.

Atkey [Atk11] applied separation logic to amortized resource analysis à la Hofmann and Jost [HJ03]. In particular, he added an operation for consumable resources into the logic and extends the logical connectives of separation logic to treat resources similar to the heap. This enables reasoning about quantities that appears to be similar to QSL in some cases. For instance, one might reason about the amount of time required to traverse a list. Moreover, Atkey's logic, which simultaneously deals with different resources, e.g., heap and time, suggests

[1] In fact, both papers were presented within 36 minutes in the same session at the same conference.

that QSL might be applied to achieve local reasoning about other kinds of consumable resources, such as access to I/O devices. However, Atkey considers neither probabilistic programs nor programs in which resources do not solely depend on the heap, e.g., numerically bounded loops.

Finally, various notions of *permissions* [Boy03; Bor+05] have been proposed on top of separation logic to reason about concurrent programs. Intuitively, a permission models the amount of ownership, e.g., read or write access, a thread has for a given memory cell. In this sense, permissions can be thought of as a quantitative aspect of separation logic as well. We refer to [DHA09; DLL17] for a discussion of various permission theories.

5.3 The Probabilistic Procedural Pointer Programming Language

We now endow the procedural pointer programming language $\mathbf{P^3L}$ considered in Chapter 4 with instructions to sample from probability distributions. This results in the *probabilistic* procedural pointer programming language, $\mathbf{P^4L}$ for short. First, we introduce the syntax of $\mathbf{P^4L}$ and discuss how probability distributions are incorporated. Since transition systems, which are at the foundation of the operational semantics presented in Chapters 2 to 4, do not take probabilities into account, we present the operational semantics of $\mathbf{P^4L}$ in terms of Markov Decision Processes [BK08; Put05]—an established model for probabilistic systems. Finally, we discuss operational reasoning about expected values.

5.3.1 Syntax of $\mathbf{P^4L}$

We extend the syntax of $\mathbf{P^3L}$ by two probabilistic statements:

First, the *probabilistic choice* $\{\, C_1 \,\}\ [\, p \,]\ \{\, C_2 \,\}$ flips a coin with bias p. If the coin shows heads, we execute program C_1. Otherwise, i.e., if the coin shows tails, we execute program C_2. For example, a fair choice between assigning either 0 or 1 to variable x is modeled by the program $\{\, x := 0 \,\}\ [\, 1/2 \,]\ \{\, x := 1 \,\}$.

Second, the *probabilistic assignment* $x :\approx \mu$ samples a value from a probability distribution specified by *distribution expression* μ and then assigns it to variable x. We assume distribution expressions to represent *discrete* probability distributions over the set of integers with possibly infinite support. Moreover, we denote by

$$p_1 \cdot \langle v_1 \rangle + p_2 \cdot \langle v_2 \rangle + \ldots + p_n \cdot \langle n \rangle$$

the distribution expression that assigns probability p_i to value v_i; all values not explicitly listed in the above sum have probability 0. Since many randomized algorithms employ uniform distributions over some interval, we denote by

$$\mathsf{uniform}\,(n,m) \triangleq \sum_{k=0}^{m-n} \frac{1}{m-n+1} \cdot \langle n+k \rangle$$

the distribution expression that assigns the same probability to every integer ranging from n to m. Hence, the following programs have the same effect:

- $x :\approx 1/2 \cdot \langle 0 \rangle + 1/2 \cdot \langle 1 \rangle$,
- $x :\approx \mathsf{uniform}\,(0,1)$, and
- $\{\, x := 0 \,\}\ [\,1/2\,]\ \{\, x := 1 \,\}$.

Definition 5.1 (Syntax of P^{4}L [1]) The set of programs in the *Probabilistic Procedural Pointer Programming Language with Auxiliaries*, denoted **P^{4}LA**, is given by the context-free grammar below (**P^{3}LA** statements are black-75):

$C \;\to$	$\{\, C_1 \,\}\ [\, p \,]\ \{\, C_2 \,\}$	(probabilistic choice)
$\mid$	$x :\approx \mu$	(probabilistic assignment)
$\mid$	$x := \texttt{alloc}(\vec{E})$	(allocation)
$\mid$	$\texttt{free}(E)$	(deallocation)
$\mid$	$x := <E>$	(lookup)
$\mid$	$<E> := E'$	(mutation)
$\mid$	`skip`	(effectless program)
$\mid$	$x := E$	(assignment)
$\mid$	$C\,;\,C$	(sequential composition)
$\mid$	$\texttt{if}\ (\,B\,)\ \{\, C \,\}\ \texttt{else}\ \{\, C \,\}$	(conditional choice)
$\mid$	$\texttt{while}\,(\,B\,)\,\{\, C \,\}$	(loop)
$\mid$	$x := F(\vec{E})$	(procedure call)
$\mid$	`enter`	(enter scope)
$\mid$	$\texttt{invoke}\ F$	(invoke procedure)
$\mid$	`leave`,	(leave scope)

where $p \in \mathbb{Q} \cap [0,1]$ is a rational probability, $x \in \mathbf{Vars}$ is a variable, and μ is a distribution expression. Furthermore, $F \in \mathbf{Procs}$ is a procedure name, $\vec{E}$ is a sequence of arithmetic expressions, E and E' are arithmetic expressions, and B is a Boolean expression. All involved expressions are evaluated over variables, i.e., independent of the heap.

Moreover, the set of programs in the *Probabilistic Procedural Pointer Programming Language*, denoted $\mathbf{P^4L}$, consists of all $\mathbf{P^4LA}$ programs, which contain neither `enter` nor `invoke` F nor `leave` statements.

Before we assign formal semantics to $\mathbf{P^4L}$ programs, let us consider an example of a randomized algorithm implemented in $\mathbf{P^4L}$. A popular scheme to design randomized algorithms consists of two steps (cf. [Cor+09, Chapter 5.1]): First, compute a random permutation of the input and then apply a deterministic algorithm to solve the actual problem. The intuition behind this approach is that—provided the deterministic algorithm performs badly on a few worst-case instances only—we reduce the risk of bad performance because it is unlikely that a random permutation of the input coincides with a worst-case instance. By the same argument, however, we also reduce the chance of encountering a best-case instance on which the algorithm is particularly fast. Randomized algorithms that aim to improve the (expected) worst-case performance at the cost of worsening best-case performance are also known as "Sherwood algorithms".[2]

Example 5.2 Sherwood algorithms require a reliable technique to compute random permutations of a given input, say an array consisting of n elements. Cormen et al. [Cor+09, Chapter 5.3] discuss several approaches to address this problem. The $\mathbf{P^4L}$ procedure below implements one of them:

```
randomize(array, n) {
    i := 0;
    while ( 0 ≤ i < n ) {
        j :≈ uniform (i, n − 1) ;
        void := swap(array, i, j);
        i := i + 1
    };
    out := 0
}
```

```
swap(array, i, j) {
    y := array [ i ] ;
    z := array [ j ] ;
    array [ i ] := z;
    array [ j ] := y;
    out := 0
}
```

[2] The phrase is a reference to Sherwood forests famous resident Robin Hood, who takes from the rich (best-case instances) and gives to the poor (worst-case instances) [BB96, Chapter 10.7].

Procedure randomize permutes its input—an array of length n—in-place: In the i-th iteration, the i-th array element is swapped with some element from $\mathsf{array}[i]$ to $\mathsf{array}[n-1]$. After that, the element's position in the array is fixed, i.e., it is not swapped with some other element again. We will show in Section 8.4 that procedure randomize is correct in the sense that it produces every possible permutation of array with probability $1/n!$.

5.3.2 Semantics of P⁴L

Similarly to the operational semantics of **PL**, **P²L**, and **P³L** programs considered in Chapters 2 to 4, we define the operational semantics of **P⁴L** programs in terms of a transition system. However, there are now *two* possible causes for a program to admit multiple executions: On the one hand, both the probabilistic choice and the probabilistic assignment admit multiple steps, but the probability of each step can be precisely quantified. On the other hand, we assume that the allocation statement is truly nondeterministic, i.e., we cannot assign a probability to any particular value chosen by the memory allocator.[3]

To incorporate both features in our operational semantics, we attach two labels to the underlying transition system's execution relation:

1. We endow every step with the probability of its execution.
2. Whenever a step represents a choice between values, e.g., due to sampling or memory allocation, we label it with the chosen value.

Formally, we add a set **Act** of *actions* to transition systems (cf. Definition 2.2): Hence, throughout the remainder of Part II, a *transition system TS* is a tuple

$$TS \triangleq \langle S, \to, S_0 \rangle,$$

where S is a set of *states*, $S_0 \subseteq S$ is a set of *initial states*, and

$$\to \subseteq S \times \mathbf{Act} \times S$$

is an *execution relation*. We usually write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \to$ to denote a *step* of the execution relation. Moreover, the transition system (without actions) underlying TS is defined as $TS' = \langle S, \to', S_0 \rangle$, where

$$s \to' s' \quad \text{iff} \quad \exists a \in \mathbf{Act}\colon\ s \xrightarrow{a} s'.$$

[3] While one could fix a probability distribution that governs memory allocation, such a distribution would surely depend on in-depth knowledge about a specific hardware platform. In general, we consider it more sensible to assume no prior knowledge about the internals of the memory allocator.

All previous notions considered for transition systems in Definitions 2.8 and 2.10, e.g., executions and reachable states, are defined analogously for transition systems with actions. In particular, if R is the set of all states reachable in TS from some initial state $s \in S_0$, then the *reachable fragment of TS* is defined as

$$\mathbf{Reach}\,(TS) \;\triangleq\; \langle\, R,\, \to \cap\, (R \times \mathbf{Act} \times R),\, S_0 \,\rangle.$$

Towards a transition system with actions that determines the operational semantics of $\mathbf{P^4L}$ programs, we choose the set of actions

$$\mathbf{Act} \;\triangleq\; \underbrace{\mathbb{Q} \cap [0,1]}_{\text{“probability of a step”}} \;\times\; \underbrace{\mathbb{Z}}_{\text{“value chosen during a step”}}.$$

Moreover, similarly to the operational semantics of $\mathbf{P^3L}$ programs introduced in Definition 4.8, the set of states is defined as

$$\mathbf{States} \;\triangleq\; \Big(\Big(\mathbf{P^4LA} \cup \{\,\mathsf{term}\,\}\Big) \;\times\; \mathbf{SHPairs}\Big) \;\cup\; \big\{\,\langle \mathsf{sink} \rangle, \langle \mathsf{fault} \rangle\,\big\},$$

where term indicates successful termination, **SHPairs** is the set of all stack-heap pairs, $\langle \mathsf{sink} \rangle$ is a dedicated sink state, and $\langle \mathsf{fault} \rangle$ is a state indicating unsuccessful termination due to a memory fault. The execution relation of the operational semantics is thus of the form

$$\rightsquigarrow \;\subseteq\; \mathbf{States} \;\times\; \mathbf{Act} \;\times\; \mathbf{States}.$$

We usually write $\langle C, \mathfrak{s}, \mathfrak{h} \rangle \overset{p}{\underset{v}{\rightsquigarrow}} \langle C', \mathfrak{s}', \mathfrak{h}' \rangle$ instead of the more cumbersome

$$((C, (\mathfrak{s}, \mathfrak{h})), p, v, (C', (\mathfrak{s}', \mathfrak{h}'))) \in \rightsquigarrow.$$

Most $\mathbf{P^4L}$ statements neither perform sampling nor admit multiple executions due to memory allocation. In this case, their steps coincide with the operational semantics of $\mathbf{P^3L}$ programs except that we have to provide an action. To this end, we set the probability of a deterministic step to one and fix value zero for the action's second component. For example, the rule for assignments in $\mathbf{P^4L}$ is:

$$\frac{E\,(\mathfrak{s}) = v}{\langle x := E,\, \mathfrak{s}, \mathfrak{h} \rangle \;\overset{1}{\underset{0}{\rightsquigarrow}}\; \langle \mathsf{term},\, \mathfrak{s}\,[x/v]\,, \mathfrak{h} \rangle}\;\text{assign}$$

For the memory allocation $x := \mathtt{alloc}(E_1, \ldots, E_n)$, the value chosen during a step coincides with the first address that is allocated. Since this value is chosen

truly nondeterministically, we assign a probability of one to each possible step. This leads to the following rule:

$$\frac{u, u+1, \ldots, u+n-1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(\mathfrak{h}) \qquad E_1(\mathfrak{s}) = v_1, \ldots, E_n(\mathfrak{s}) = v_n}{\langle x := \mathtt{alloc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h} \rangle \overset{1}{\underset{u}{\rightarrow}} \langle \mathsf{term}, \mathfrak{s}[x/u], \mathfrak{h} \uplus \{ u :: v_1, \ldots, v_n \} \rangle} \text{ alloc}$$

The probabilistic choice $\{ C_1 \} [p] \{ C_2 \}$ flips a coin with bias p. With probability p, the coin yields heads, which we identify with value 0, and we execute program C_1. With probability $1 - p$, the coin yields tails, which we identify with value 1, and we execute program C_2. Hence, the execution relation for probabilistic choices is determined by the following two rules:

$$\frac{}{\langle \{ C_1 \} [p] \{ C_2 \}, \mathfrak{s}, \mathfrak{h} \rangle \overset{p}{\underset{0}{\rightarrow}} \langle C_1, \mathfrak{s}, \mathfrak{h} \rangle} \text{ pchoice-h}$$

$$\frac{}{\langle \{ C_1 \} [p] \{ C_2 \}, \mathfrak{s}, \mathfrak{h} \rangle \overset{1-p}{\underset{1}{\rightsquigarrow}} \langle C_2, \mathfrak{s}, \mathfrak{h} \rangle} \text{ pchoice-t}$$

Notice that it is important to attach the result of the coin flip, i.e., the value 0 or 1, to the execution relation. For instance, the program C given by

$$\{ x := 17 \} [1/2] \{ x := 17 \}$$

flips a fair coin and assigns 17 to variable x. The probability that x is equal to 17 after execution is thus, intuitively, one. In fact, if we fix some stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, there are two possible steps whose attached probabilities add up to one:

$$\langle C, \mathfrak{s}, \mathfrak{h} \rangle \overset{1/2}{\underset{0}{\rightsquigarrow}} \langle x := 17, \mathfrak{s}, \mathfrak{h} \rangle \quad \text{and} \quad \langle C, \mathfrak{s}, \mathfrak{h} \rangle \overset{1/2}{\underset{1}{\rightsquigarrow}} \langle x := 17, \mathfrak{s}, \mathfrak{h} \rangle$$

Without the attached values 0 and 1, however, both steps are indistinguishable. If we omit them, the execution relation contains only one step, i.e., it would seem that the probability that variable x equals 17 is $1/2$ instead of 1.

For the probabilistic assignment $x :\approx \mu$, let us first be precise about the meaning of distribution expressions: A *distribution expression* μ is a function

$$\mu \colon \mathbf{Stacks} \to \mathbf{Dist}(\mathbb{Z}),$$

where $\mathbf{Dist}(\mathbb{Z})$ is the set of discrete probability distributions that assign a rational probability to every integer. That is,

$$\mathbf{Dist}(\mathbb{Z}) \triangleq \left\{ f \colon \mathbb{Z} \to \mathbb{Q} \cap [0,1] \;\middle|\; \sum_{v \in \mathbb{Z}} f(v) = 1 \right\}.$$

Hence, $\mu(\mathfrak{s})(v)$ denotes the probability of sampling value v from the distribution given by evaluating expression μ in stack $\mathfrak{s}$. Once we fix a sampled value v and its probability p, the rule for probabilistic assignments is analogous to standard assignments. It is thus defined as:

$$\frac{\mu(\mathfrak{s})(v) = p > 0}{\langle x :\approx \mu, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{v}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h}\rangle}\ \text{passign}$$

Finally, the sequential composition $C_1 ; C_2$ has to account for the fact that it may contain memory allocations, probabilistic choice, or probabilistic assignment. Its action thus coincides with the action taken when executing program C_1.

We summarize the full definition of $\mathbf{P^4L}$'s operational semantics below.

Definition 5.3 (Operational Semantics of P^{4}L Programs) Let term be a special symbol indicating successful termination. Moreover, let $\langle \mathsf{sink}\rangle$ be a sink state and $\langle \mathsf{fault}\rangle$ be a state indicating unsuccessful termination due to a memory failure. The *operational semantics of* $\mathbf{P^4LA}$ *programs* is given by the transition system $\mathrm{oP^4L} \triangleq \langle\, \mathbf{States}, \rightsquigarrow, \mathbf{States}\,\rangle$, where both the set of states and the set of initial states are defined as

$$\mathbf{States} \triangleq \Big(\big(\mathbf{P^4LA} \cup \{\,\mathsf{term}\,\}\big) \times \mathbf{SHPairs}\Big) \cup \{\,\langle \mathsf{sink}\rangle, \langle \mathsf{fault}\rangle\,\}.$$

Furthermore, the execution relation

$$\rightsquigarrow \subseteq \mathbf{States} \times \mathbb{N} \times \mathbb{Q} \cap [0,1] \times \mathbf{States}$$

is the smallest relation induced by the rules in Figure 5.6, page 163, and Figure 5.7, page 164 (the semantics of $\mathbf{P^3L}$ programs is displayed in black-75).

The *reachable fragment* of $\mathrm{oP^4L}$ with respect to the set of initial states $\mathbf{I} \subseteq \mathbf{States}$ is

$$\mathrm{oP^4L}\,(\mathbf{I}) \triangleq \mathbf{Reach}\,(\langle\, \mathbf{States}, \rightsquigarrow, \mathbf{I}\,\rangle).$$

Example 5.4 Consider the following $\mathbf{P^4L}$ C:

$$x :\approx {}^{2}/_{3} \cdot \langle x+1\rangle + {}^{1}/_{3} \cdot \langle 7\rangle ; \underbrace{\texttt{while}\,(\,x \text{ is odd}\,)\,\{\,\texttt{skip}\,\}}_{= C'}$$

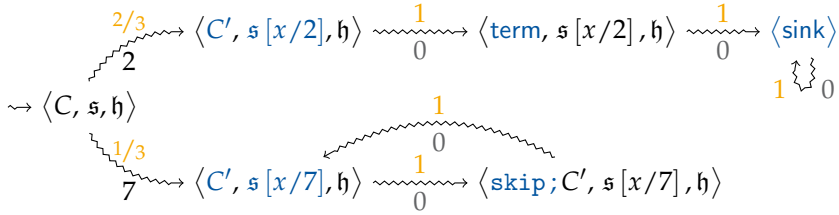


Figure 5.3: Illustration of oP^{4}L $(\{\langle C, \mathfrak{s}, \mathfrak{h}\rangle\})$. Probabilities are colored orange and chosen values are colored black-75, respectively.

This program first flips a biased coin; with probability $2/3$ it increments the value of variable x. Otherwise, it assigns 7 to x. After that, the program loops forever if x is odd or immediately terminates if x is even.
Figure 5.3 on page 153 depicts the transition system oP^{4}L $(\{\langle C, \mathfrak{s}, \mathfrak{h}\rangle\})$, where $(\mathfrak{s}, \mathfrak{h})$ is a stack-heap pair with $\mathfrak{s}(x) = 1$. We observe that there is exactly one execution that terminates, i.e., reaches the sink state. Its probability, which, intuitively, corresponds to the product of all probabilities along that execution, is $2/3$.

5.3.3 Operational Reasoning about P^{4}L Programs

The transition system oP^{4}L precisely defines the set of all executions for every **P^{4}L** program and every stack-heap pair. Furthermore, whenever a value is chosen probabilistically, the chosen value and its probability are made explicit. This allows us to analyze single executions as we did in Example 5.4. When reasoning about probabilistic programs, however, we are interested in properties quantified over all executions. Consider, for example, the **P^{4}L** program C below:

$$
\begin{aligned}
&x := 1\,;\\
&y := 0\,;\\
&\texttt{while}\,(x \neq 0)\,\{\\
&\qquad \{\,x := 0\,\}\,[1/2]\,\{\,y := y + 1\,\}\\
&\}
\end{aligned}
$$

This program keeps flipping a fair coin. As long as it hits tails, it increments variable y. Once it hits heads, it sets variable x to zero and terminates.

Asking whether there exists an execution such that, upon termination, variable y is either less than, equal to, or greater than some natural number does not make much sense; the answer is always yes. Similarly, asking whether every

execution terminates with variable y being either less than, equal to, or greater than some natural number does not make much sense; the answer is always no. In fact, C does not even always terminates, because the coin flip might yield tails all the time. Still, there are sensible questions that contribute to understanding the behavior of program C. For instance:

- What is the probability that C terminates?
- What is the probability that C terminates and variable y is at most 7?
- What is the expected value of variable y upon termination of C?

In this section, we consider how the operational semantics of $\mathbf{P^4L}$ can be applied to answer such questions. Since reasoning about probabilistic programs is full of subtleties, we first take a brief excursion on a well-established mathematical model for probabilistic systems: *Markov decision processes* (MDPs for short). After that, we show that, for every initial state $\langle C, \mathfrak{s}, \mathfrak{h} \rangle$ and every property of interest, the transition system $\mathrm{oP^4L}\left(\{\, \langle C, \mathfrak{s}, \mathfrak{h} \rangle \,\}\right)$ describes an MDP. Hence, reasoning about $\mathbf{P^4L}$ programs amounts to reasoning about the underlying MDP.

While our presentation is roughly based on the textbook by Baier and Katoen [BK08, Chapter 10], some definitions have been simplified with reasoning about $\mathbf{P^4L}$ in mind. For example, we consider reachability probabilities with respect to a single fixed goal state. Puterman's reference book [Put05] provides a comprehensive discussion of MDPs in general.

We assume the reader is familiar with basic concepts of discrete probability theory; a gentle introduction is provided, for example, by Graham, Knuth, and Patashnik [GKP94, Chapter 8]. Furthermore, we refer to [BK08, pp. 754–759, p. 845] for a detailed discussion of the probability spaces underlying MDPs.

Intuitively, an MDP is a transition system that assigns to every state one or more probability distributions over all states; each of these distributions is identified by an action. For every MDP, we would like to reach a fixed goal state from the initial state. Moreover, whenever we leave a state, we collect a reward.

Definition 5.5 (Markov Decision Process (MDP) [BK08, p. 833]) A *Markov Decision Process* is a tuple $\mathcal{M} = \langle S, \mathbf{Act}, \mathsf{Prob}, s_0, s_\mathcal{G}, \mathsf{rew} \rangle$, where

- S is a countable set of *states*,
- **Act** is a countable set of *actions*,

- $\mathsf{Prob}\colon S \times \mathbf{Act} \times S \to \mathbb{Q} \cap [0,1]$ is the *transition probability function* such that, for all states $s \in S$ and actions $a \in \mathbf{Act}$, we have

$$\sum_{s' \in S} \mathsf{Prob}\left(s, a, s'\right) \;=\; 0 \quad \text{or} \quad \sum_{s' \in S} \mathsf{Prob}\left(s, a, s'\right) \;=\; 1,$$

  where we call action *a enabled* in state s in the latter case,

- $s_0 \in S$ is the *initial state*,
- $s_{\mathcal{G}} \in S$ is the *goal state*, and
- $\mathsf{rew}\colon S \to \mathbb{R}^{\infty}_{\geq 0}$ is the *reward function* that assigns a non-negative real number or infinity to every state.

We denote by $\mathbf{Act}\left(s\right)$ the set of actions enabled in state s. For every state $s \in S$ it is required that $\mathbf{Act}\left(s\right) \neq \emptyset$ holds.

Example 5.6 Apart from the reward function rew, all components of an MDP $\mathcal{M} = \langle\, S, \mathbf{Act}, \mathsf{Prob}, s_0, s_4, \mathsf{rew} \,\rangle$ are depicted in Figure 5.4: $\mathcal{M}$ consists of five states, i.e., $S = \{\, s_0, s_1, s_2, s_3, s_4 \,\}$, and two actions, i.e., $\mathbf{Act} = \{\, a, b \,\}$. Moreover, the initial state s_0 is marked with an incoming edge without a source. The goal state s_4 is highlighted by a box drawn around it. The transition probability function Prob is given by the edges labeled with an action and the assigned probability. For instance, if we choose action b the probability to move from state s_1 to s_4 is $\mathsf{Prob}\left(s_1, b, s_4\right) = 1/2$. A missing edge for two states and some action indicates that Prob assigns zero probability to a transition between these states for the chosen action. Hence, for action a, we have $\mathsf{Prob}\left(s_1, a, s_4\right) = 0$. Furthermore, notice that the initial state s_0 is the only state in which both actions a and b are enabled.

We are concerned with two questions when reasoning about MDPs:

1. What is the *probability* to reach goal state $s_{\mathcal{G}}$ from initial state s_0?
2. What is the total *expected reward* collected along all paths reaching goal state $s_{\mathcal{G}}$ from initial state s_0?

Both of these questions only make sense in the absence of nondeterminism. After all, if multiple actions are enabled in a state, this state admits a choice between the probability distributions that govern how to move to the next state. To resolve nondeterministic choices, we employ a *scheduler*, i.e., a function

$$\mathfrak{S}\colon S^{+} \to \mathbf{Dist}\left(\mathbf{Act}\right)$$

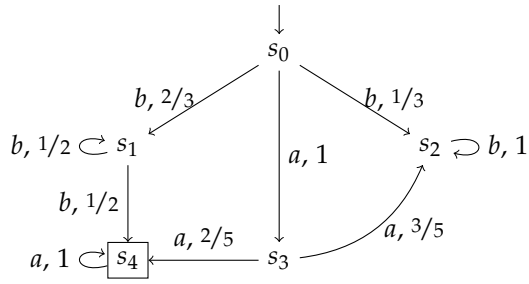


Figure 5.4: Illustration of an MDP with initial state s_0 and goal state s_4.

mapping a non-empty sequence of previously visited states (including the current state) to a probability distribution over the set of actions enabled in the current state. Intuitively, $\mathfrak{S}(s_0 \dots s_k)(a)$ denotes the probability that the next chosen action is a. Notice that, from a theoretical perspective, it suffices to consider *Markovian schedulers*, i.e., schedulers that only depend on the current state [Put05, Proposition 7.1.1]. However, it is often more convenient to work with schedulers that do depend on the history of all previously visited states rather than encoding the same behavior in a probability distribution.

Once we have fixed a scheduler $\mathfrak{S}$, we only consider executions from the initial state to the goal state that are consistent with the actions chosen by $\mathfrak{S}$. Formally, let $\mathcal{M} = \langle S, \mathbf{Act}, \mathsf{Prob}, s_0, s_{\mathcal{G}}, \mathsf{rew} \rangle$ be an MDP. Then the set of all (finite) *paths* of $\mathcal{M}$ induced by scheduler $\mathfrak{S}$ that eventually reach goal state $s_{\mathcal{G}}$ from initial state s_0 is defined as

$$\begin{aligned}\mathbf{Paths}_{\mathcal{M}}(\mathfrak{S}) \;\triangleq\; \{s_0 \dots s_n \mid s_n = s_{\mathcal{G}} \text{ and } \forall k \in [0, n-1] \colon s_k \neq s_{\mathcal{G}} \text{ and } \exists a \in \mathbf{Act} \colon \\ \mathfrak{S}(s_0 \dots s_k)(a) > 0 \text{ and } \mathsf{Prob}\,(s_k, a, s_{k+1}) > 0\}.\end{aligned}$$

The probability of reaching the goal state from the initial state then amounts to the sum of the probabilities of all paths in $\mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})$, where—for a fixed sequence of actions chosen by scheduler $\mathfrak{S}$—the *probability of a path* $s_0 \dots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})$ is the product of its transition probabilities. That is,

$$\mathsf{Prob}_{\mathcal{M}}(s_0 \dots s_n) \;\triangleq\; \sum_{a_0, \dots, a_{n-1} \in \mathbf{Act}} \; \prod_{0 \leq k < n} \mathfrak{S}(s_0 \dots s_k)(a_k) \cdot \mathsf{Prob}\,(s_k, a_k, s_{k+1})\,.$$

To answer the second question, we additionally have to account for the reward collected whenever we leave a state. The *cumulative reward* collected along a

path $s_0 \dots s_n$ amounts to the sum of all rewards assigned to states on that path apart from the last one (which is not left), i.e.,

$$\mathsf{rew}_{\mathcal{M}}(s_0 \dots s_n) \triangleq \sum_{k=0}^{n-1} \mathsf{rew}(s_k).$$

Finally, we sum up the probabilities of all paths eventually reaching the goal state $s_{\mathcal{G}}$ from the initial state s_0 weighted according to the cumulative reward collected along the way. Consequently, the *expected reward* of MDP $\mathcal{M}$ collected by the scheduler $\mathfrak{S}$ is

$$\sum_{s_0 \dots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}}(s_0 \dots s_n) \cdot \mathsf{rew}_{\mathcal{M}}(s_0 \dots s_n).$$

Notice that reasoning about the probability to reach the goal state is a special case of expected rewards in which $\mathsf{rew}_{\mathcal{M}}(s_0 \dots s_n)$ equals one for every path. Since we often do not know precisely which scheduler is used—just as we do not know the behavior of the memory allocator in Chapter 4—we either compute the minimal or the maximal expected reward over all schedulers. Throughout this thesis, we usually compute *minimal* expected rewards. Unless stated otherwise "expected reward" thus always refers to the minimal expected reward over all schedulers. Formally, the *expected reward* of MDP $\mathcal{M}$ is defined as

$$\mathsf{ExpRew}(\mathcal{M}) \triangleq \inf_{\mathfrak{S}} \sum_{s_0 \dots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}}(s_0 \dots s_n) \cdot \mathsf{rew}_{\mathcal{M}}(s_0 \dots s_n).$$

The underlying rationale is that minimal expected rewards correspond to a demonic interpretation of nondeterminism. That is, no matter how nondeterminism is resolved, we can guarantee a lower bound on the expected reward. The same scenario is considered by McIver & Morgan [MM05].

Example 5.7 Consider, again, the MDP $\mathcal{M}$ from Example 5.7 which is depicted in Figure 5.4. Since s_0 is the only state in which two actions are enabled and that state is never visited again, every scheduler represents a probabilistic choice between actions a and b. Let us consider the two schedulers that assign all probability mass to a single action: $\mathfrak{S}_a$ initially chooses action a whereas scheduler $\mathfrak{S}_b$ initially chooses action b, respectively.

For scheduler $\mathfrak{S}_a$, there is exactly one path from initial state s_0 to the goal state s_4, namely $s_0 s_3 s_4$; its probability is $2/5$.

For scheduler $\mathfrak{S}_b$, however, there are infinitely many paths from s_0 to s_4. More precisely, if s_1^k denotes k repetitions of state s_1, then

$$\mathbf{Paths}_{\mathcal{M}}(\mathfrak{S}_b) = \left\{ s_0 s_1^k s_4 \;\middle|\; k \geq 1 \right\}.$$

The probability to reach goal state s_4 from initial state s_0 is then given by:

$$\sum_{s'_0 \dots s'_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S}_b)} \mathsf{Prob}_{\mathcal{M}}(s'_0 \dots s'_n) \;=\; \sum_{k=1}^{\infty} \mathsf{Prob}_{\mathcal{M}}(s_0 s_1^k s_4) \;=\; \sum_{k=1}^{\infty} 2/3 \cdot 1/2^k \;=\; 2/3.$$

For minimizing the probability to reach state s_4 over all schedulers, we have to determine a value $p \in [0,1]$ that minimizes the convex sum of the reachability probabilities induced by schedulers $\mathfrak{S}_a$ and $\mathfrak{S}_b$, i.e.,

$$p \cdot 2/5 + (1-p) \cdot 2/3.$$

The desired minimum is attained for $p = 1$. Hence, always choosing action a yields the minimal expected reward. Notice that the corresponding scheduler $\mathfrak{S}_a$ is both Markovian and deterministic, i.e., it neither depends on previously visited states nor assigns probabilities different from zero and one to any action. This is not a coincidence: For every MDP with finitely many states and actions, there exists a deterministic Markovian scheduler which induces the minimal expected reward [Put05, Theorem 6.2.10].

For later reference, notice that expected rewards can be understood as a (not necessarily unique) solution of a set of optimality equations that are commonly known as Bellman equations [Bel57]. Formally, let us denote by $\mathcal{M}_s$ the MDP $\mathcal{M}$ in which the initial state is set to s. Then:

Theorem 5.8 For every MDP $\mathcal{M} = \langle S, \mathbf{Act}, \mathsf{Prob}, s_0, s_{\mathcal{G}}, \mathsf{rew} \rangle$ and every state $s \in S$, we have

$$\mathsf{ExpRew}(\mathcal{M}_s) \;=\; \inf_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s')=p>0} \mathsf{rew}(s') + p \cdot \mathsf{ExpRew}(\mathcal{M}_{s'}).$$

Proof. The claim is dual to [Put05, Theorem 7.1.3]. A detailed proof is found in Appendix C.1. □

Let us now return to reasoning about **P⁴L** programs. Essentially, the transition system oP⁴L already describes an MDP once we fix an initial state, a goal state, and a reward function. The main difference is that we interpret the execution relation as a transition probability function: Whenever a state represents a probabilistic choice or a probabilistic assignment, we combine all steps into a single distribution for default action 0. Otherwise, a transition $\mathsf{Prob}(s,a,s')$ is performed with probability one for the indicated chosen value if and only if there is a corresponding step $s \overset{1}{\underset{a}{\rightsquigarrow}} s'$; if no such step exists, then we assign the

transition probability zero. Furthermore, the unique sink state $\langle \mathsf{sink} \rangle$, which is entered after *successful* termination, serves us as the goal state. In particular, notice that $\langle \mathsf{sink} \rangle$ is not reached when encountering a memory fault.

The reward function determines the property we would like to reason about. More precisely, it is given by a random variable of the form

$$\mathrm{X} \colon \mathbf{SHPairs} \to \mathbb{R}^{\infty}_{\geq 0}$$

that measures our property of interest in every state $\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \rangle$ indicating successful termination; we assign zero reward to all other states. For instance, to measure the probability of termination, we choose a random variable X mapping every stack-heap pair to one. To measure the expected absolute value of variable y, X maps every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ to $|\mathfrak{s}(y)|$. We use the *Iverson bracket* [GKP94, p. 24] notation $[P]$ to map a predicate P to zero or one. That is,

$$[P] \triangleq \begin{cases} 1, & \text{if } P \text{ evaluates to } \mathsf{true} \\ 0, & \text{if } P \text{ evaluates to } \mathsf{false}. \end{cases}$$

The property that y is at most seven then corresponds to the random variable X mapping every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ to $[\mathfrak{s}(y) \leq 7]$.

Formally, the MDPs for reasoning about **P⁴L** programs induced by our operational semantics are defined as follows (cf. [1, Section 2.3]):

Definition 5.9 (MDP Semantics for Reasoning about P⁴L Programs) Let

$$\mathrm{oP^4L}\left(\langle C, \mathfrak{s}, \mathfrak{h} \rangle\right) = \left\langle S, \to, \{\langle C, \mathfrak{s}, \mathfrak{h} \rangle\} \right\rangle$$

be the reachable fragment of the operational semantics for **P⁴L** program C and stack-heap pair $(\mathfrak{s}, \mathfrak{h})$. Then the *MDP induced by* program C, stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, and random variable X: **SHPairs** $\to \mathbb{R}^{\infty}_{\geq 0}$ is defined as $\mathcal{M} \triangleq \langle S, \mathbb{Z}, \mathsf{Prob}, s_0, s_{\mathcal{G}}, \mathsf{rew} \rangle$, where

- the initial state is $s_0 \triangleq \langle C, \mathfrak{s}, \mathfrak{h} \rangle$,
- the goal state is $s_{\mathcal{G}} \triangleq \langle \mathsf{sink} \rangle$,
- the reward function is defined as

$$\mathsf{rew}(s) \triangleq \begin{cases} \mathrm{X}(\mathfrak{s}', \mathfrak{h}'), & \text{if } s = \langle \mathsf{term}, \mathfrak{s}', \mathfrak{h}' \rangle \\ 0, & \text{otherwise, and} \end{cases}$$

- the transition probability function Prob is determined by two cases:

1. If $s = \langle \{ C_1 \} [p] \{ C_2 \}, \mathfrak{s}', \mathfrak{h}' \rangle$ or $s = \langle x :\approx \mu, \mathfrak{s}', \mathfrak{h}' \rangle$, then

$$\mathsf{Prob}(s, v, s') \triangleq [v = 0] \cdot \sum_{s \xrightarrow[v]{q} s'} q.$$

2. For all other states $s \in S$, we set $\mathsf{Prob}(s, v, s') \triangleq \left[s \xrightarrow[v]{1} s' \right]$.

We denote by $\mathsf{ExpRew}[\mathrm{X}](C, \mathfrak{s}, \mathfrak{h})$ the expected reward of the MDP $\mathcal{M}$ induced by $\mathbf{P^4L}$ program C, stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, and random variable X.

Example 5.10 Recall the $\mathbf{P^4L}$ program below that counts in variable y the number of fair coin flips in which we see tails until we first hit heads:

```
while:   while ( x ≠ 0 ) {
[1/2]:       { x := 0 } [1/2] { y := y + 1 } }
```

Here, both while and $[1/2]$ are shortcuts for the loop and the probabilistic choice, respectively. Assume that we execute program while on a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ with $\mathfrak{s}(x) = 1$ and $\mathfrak{s}(y) = 0$. Then, apart from the reward function, the MDP $\mathcal{M}$ corresponding to the transition system $\mathrm{oP^4L}(\{ \langle \mathsf{while}, \mathfrak{s}, \mathfrak{h} \rangle \})$ is illustrated in Figure 5.5, page 162. Since only the values of variables x and y are relevant, we provide their evaluation in every state instead of all updates of the stack and the heap.

In every state, exactly one action, namely 0, is enabled. Hence, there is only one scheduler $\mathfrak{S}$ to consider. Furthermore, we observe that, for every natural number n, there is exactly one path reaching the goal state $\langle \mathsf{sink} \rangle$: This path first reaches the state $\langle \mathsf{term}, 0, n \rangle$ and then moves on to $\langle \mathsf{sink} \rangle$; its probability is $1/2^{n+1}$. Let us apply $\mathcal{M}$ to answer the three questions raised at the beginning of this section:

First, what is the probability that while terminates on $(\mathfrak{s}, \mathfrak{h})$? In this case, we compute the expected reward of $\mathcal{M}$, where the underlying random variable X maps every stack-heap pair to one. That is,

$$\mathsf{ExpRew}[\mathrm{X}](\mathsf{while}, \mathfrak{s}, \mathfrak{h}) \;=\; \sum_{n=0}^{\infty} 1/2^{n+1} \cdot 1 \;=\; 1.$$

Hence, our program terminates almost-surely on $(\mathfrak{s}, \mathfrak{h})$.

Second, what is the probability that $y < 7$ holds? In this case, the underlying random variable X maps every stack-heap pair $(\mathfrak{s}', \mathfrak{h}')$ to $[\mathfrak{s}'(y) < 7]$,

i.e., the reward is one if we terminate and $y < 7$ holds. Otherwise, it is zero. The expected reward of $\mathcal{M}$ and thus the desired probability is then given by

$$\mathsf{ExpRew}[\mathrm{X}]\,(\mathsf{while}, \mathfrak{s}, \mathfrak{h}) \;=\; \sum_{n=0}^{\infty} 1/2^{n+1} \cdot [n < 7] \;=\; \frac{127}{128} \;\approx\; 0.99212.$$

Finally, what is the expected value of y upon termination? In this case, the underlying random variable X maps every stack-heap pair $(\mathfrak{s}', \mathfrak{h}')$ to the (absolute) value of y, i.e., $|\mathfrak{s}'(y)|$. The expected value of y is then given by the expected reward of $\mathcal{M}$, i.e.,

$$\mathsf{ExpRew}[\mathrm{X}]\,(\mathsf{while}, \mathfrak{s}, \mathfrak{h}) \;=\; \sum_{n=0}^{\infty} 1/2^{n+1} \cdot n \;=\; 1$$

Hence, we can expect $\mathbf{P^4L}$ program while to terminate with y being one, i.e., after two coin flips.

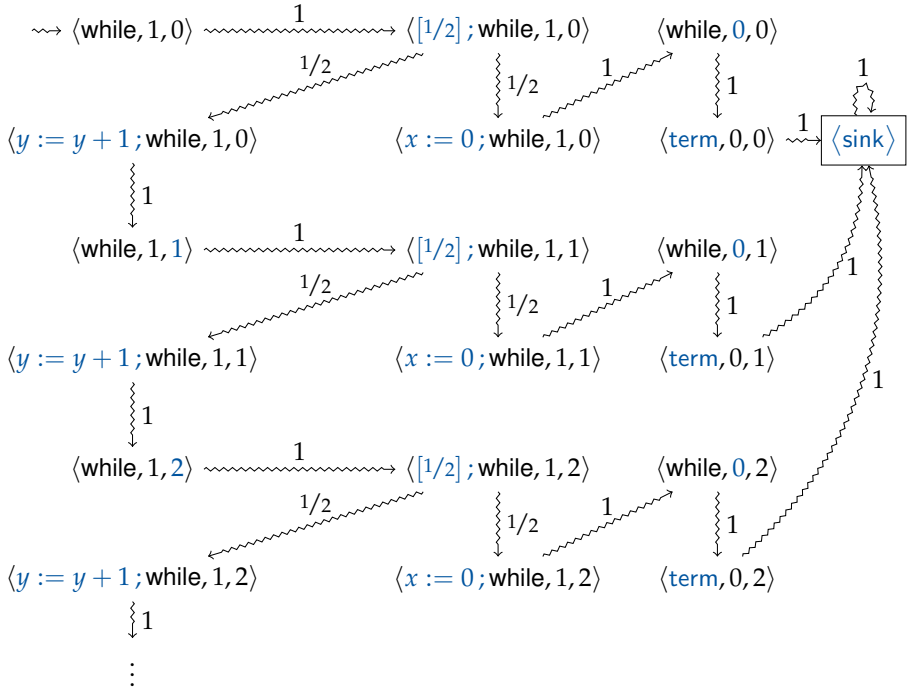


Figure 5.5: Illustration of the MDP corresponding to $\mathrm{oP}^4\mathrm{L}\left(\left\{\left\langle \mathsf{while}, \mathfrak{s}, \mathfrak{h}\right\rangle\right\}\right)$ considered in Example 5.10. For simplicity, we denote each state by a triple $\langle C, u, v\rangle$, where C is the program to execute, u is the value of variable x, and v is the value of variable y, respectively. Since every edge is labeled with action 0, we omit actions to improve readability.

$$\frac{}{\left\langle \{ C_1 \} [p] \{ C_2 \}, \mathfrak{s}, \mathfrak{h} \right\rangle \overset{p}{\underset{0}{\rightsquigarrow}} \left\langle C_1, \mathfrak{s}, \mathfrak{h} \right\rangle} \text{ pchoice-h}$$

$$\frac{}{\left\langle \{ C_1 \} [p] \{ C_2 \}, \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1-p}{\underset{1}{\rightsquigarrow}} \left\langle C_2, \mathfrak{s}, \mathfrak{h} \right\rangle} \text{ pchoice-t}$$

$$\frac{\mu(\mathfrak{s})(v) = p > 0}{\left\langle x :\approx \mu, \mathfrak{s}, \mathfrak{h} \right\rangle \overset{p}{\underset{v}{\rightsquigarrow}} \left\langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \right\rangle} \text{ passign}$$

$$\frac{u, u+1, \ldots, u+n-1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(\mathfrak{h}) \qquad E_1(\mathfrak{s}) = v_1, \ldots, E_n(\mathfrak{s}) = v_n}{\left\langle x := \mathtt{alloc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{u}{\rightsquigarrow}} \left\langle \mathsf{term}, \mathfrak{s}[x/u], \mathfrak{h} \uplus \{ u :: v_1, \ldots, v_n \} \right\rangle} \text{ alloc}$$

$$\frac{E(\mathfrak{s}) = u}{\left\langle \mathtt{free}(E), \mathfrak{s}, \mathfrak{h} \uplus \{ u :: v \} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h} \right\rangle} \text{ free}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(h)}{\left\langle \mathtt{free}(E), \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{fault} \right\rangle} \text{ free-fail}$$

$$\frac{E(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad \mathfrak{h}(u) = v}{\left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h} \right\rangle} \text{ lookup}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\left\langle x := <E>, \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{fault} \right\rangle} \text{ lookup-fail}$$

$$\frac{E(\mathfrak{s}) = u \in \mathsf{dom}(\mathfrak{h}) \quad E'(\mathfrak{s}) = v}{\left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h}[u/v] \right\rangle} \text{ mutation}$$

$$\frac{E(\mathfrak{s}) \notin \mathsf{dom}(\mathfrak{h})}{\left\langle <E> := E', \mathfrak{s}, \mathfrak{h} \right\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \left\langle \mathsf{fault} \right\rangle} \text{ mutation-fail}$$

Figure 5.6: The rules for probabilistic and heap manipulating statements that determine the execution relation of the operational semantics of **P⁴LA** programs.

$$\frac{}{\langle \texttt{skip}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{skip} \qquad \frac{E(\mathfrak{s}) = v}{\langle x := E, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}[x/v], \mathfrak{h}\rangle}\ \text{assign}$$

$$\frac{\langle C_1, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}', \mathfrak{h}'\rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle C_2, \mathfrak{s}', \mathfrak{h}'\rangle}\ \text{seq1} \qquad \frac{\langle C_1, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle C_1', \mathfrak{s}', \mathfrak{h}'\rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle C_1' ; C_2, \mathfrak{s}', \mathfrak{h}'\rangle}\ \text{seq2}$$

$$\frac{\langle C_1, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle \mathsf{fault}\rangle}{\langle C_1 ; C_2, \mathfrak{s}, \mathfrak{h}\rangle \overset{p}{\underset{a}{\rightsquigarrow}} \langle \mathsf{fault}\rangle}\ \text{seq3} \qquad \frac{}{\langle \mathsf{fault}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{fault}\rangle}\ \text{fault}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{if (}B\texttt{) \{}C_1\texttt{\} else \{}C_2\texttt{\}}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle C_1, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{if-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{if (}B\texttt{) \{}C_1\texttt{\} else \{}C_2\texttt{\}}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle C_2, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{if-false}$$

$$\frac{B(\mathfrak{s}) = \mathsf{true}}{\langle \texttt{while (}B\texttt{) \{}C\texttt{\}}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle C\texttt{; while (}B\texttt{) \{}C\texttt{\}}, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{while-true}$$

$$\frac{B(\mathfrak{s}) = \mathsf{false}}{\langle \texttt{while (}B\texttt{) \{}C\texttt{\}}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{while-false}$$

$$\frac{}{\langle \mathsf{term}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{sink}\rangle}\ \text{term} \qquad \frac{}{\langle \mathsf{sink}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{sink}\rangle}\ \text{sink}$$

$$\frac{C = \mathsf{impl}\,(x := \mathrm{Proc}(E_1, \ldots, E_n))}{\langle x := \mathrm{Proc}(E_1, \ldots, E_n), \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle C, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{call} \qquad \frac{C = \mathsf{body}\,(F)}{\langle \texttt{invoke } F, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle C, \mathfrak{s}, \mathfrak{h}\rangle}\ \text{invoke}$$

$$\frac{}{\langle \texttt{enter}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}[+\boxdot], \mathfrak{h}\rangle}\ \text{enter} \qquad \frac{}{\langle \texttt{leave}, \mathfrak{s}, \mathfrak{h}\rangle \overset{1}{\underset{0}{\rightsquigarrow}} \langle \mathsf{term}, \mathfrak{s}[-\boxdot], \mathfrak{h}\rangle}\ \text{leave}$$

Figure 5.7: The remaining rules for **P⁴LA** statements that determine the execution relation of the operational semantics of **P⁴LA** programs.

This chapter is based on prior publications, namely [1; 18], which are presented, discussed, and extended in the broader context of this thesis.

It is—in principle at least—possible to reason about the expected behavior of $\mathbf{P^4L}$ programs through a careful analysis of the Markov decision processes induced by their operational semantics. However, as demonstrated in Section 5.3.3, the involved MDPs typically consist of infinitely many states and might even require infinitely many actions. Computing expected values by considering every single state is thus infeasible in general.

This motivates our study of classical program verification techniques in the context of formal reasoning about the expected behavior of probabilistic pointer programs. Similarly to the weakest precondition calculi studied in Sections 2.3, 3.2 and 4.4, our goal is to compute the expected value of a random variable after program execution *compositionally on the program structure*. Moreover, each individual computation step should amount to simple, i.e., mostly syntactic, operations that facilitate *local reasoning* (cf. Section 4.4.2). To this end, we develop a *quantitative separation logic* (QSL for short). Its distinguished features are:

1. QSL is *quantitative*: It evaluates to a real number instead of a Boolean value. It is thus capable of specifying quantities such as values of program variables, heap sizes, list lengths, etc.

2. QSL is *probabilistic*: It enables reasoning about total correctness of probabilistic programs. This includes the probability of *memory safe termination with a correct result* as well as the *expected size* of heap fragments, e.g., the expected length of a path from the root to a leaf in a tree.

3. QSL is a *separation logic*: It conservatively extends classical separation logic à la Ishtiaq, Reynolds, and O'Hearn [IO01; Rey02]. In particular, we present quantitative analogs of separation logic's key operators, i.e., separating conjunction $\star$ and separating implication $\mathrel{-\!\!\star}$, and show that they preserve virtually all properties of their Boolean counterparts.

On a technical level, QSL is a marriage of separation logic and McIver and Morgan's weakest preexpectation calculus [MM05]. In fact, it conservatively extends both approaches: For programs that never access the heap, we obtain the calculus by McIver and Morgan. Conversely, for Boolean properties of non-probabilistic programs, we recover the weakest precondition calculus for classical separation logic considered in Section 4.4.

As it is common for program logics (cf. Sections 2.3 and 4.4 for two examples), QSL refers to both an *assertion language* and a *verification system*.

QSL's *assertion language* consists of random variables rather than Boolean predicates. The underlying rationale is to combine concepts from two worlds:

1. From separation logic: *separating conjunction* and *separating implication*.

2. From verification of probabilistic programs: *expectations*.

As discussed in Sections 4.2 to 4.4, separating conjunction and separating implication are the two distinguished logical connectives of separation logic.

Expectations [MM05] on the other hand take over the role of predicates when performing *quantitative reasoning* about probabilistic programs.

Throughout this chapter, we gradually develop both a *quantitative separating conjunction* and a *quantitative separating implication* which each connect expectations instead of predicates. Furthermore, we consider various proof rules to discharge "quantitative entailments", i.e., relationships between expectations. These rules are crucial because—as we have observed in previous chapters—Floyd-Hoare style verification boils down to three tasks: Finding suitable invariants, deriving valid triples, e.g., by computing weakest preconditions, and proving all *entailments*, i.e., logical implications, encountered along the way. The rules presented in this chapter form the foundation for a proof system to deal with the last task in a quantitative setting.

Most theorems presented in this chapter are proven by a careful application of their definition, elementary algebra, and predicate logic. We thus omit proofs that—in the author's opinion—do not improve the intuition underlying QSL. Moreover, in cooperation with Max Haslbeck from Technical University Munich, the key theorems have been verified in the theorem prover Isabelle/HOL; the formalization is available online.[1]

We address the other tasks, i.e., invariants and weakest precondition style reasoning about expected values, in Chapter 7 alongside QSL's *verification system*.

[1] https://github.com/maxhaslbeck/QuantSepCon

6.1 Expectations

Floyd-Hoare logic [Flo67; Hoa69] as well as Dijkstra's weakest precondition calculus [Dij76] employ predicates for reasoning about the correctness of programs. In the intensional approach, a popular language to specify predicates is first-order logic. In the extensional approach, which we took in Chapters 2 to 4, we admit every computable predicate.

For probabilistic programs, Kozen [Koz83] was the first to generalize from predicates to measurable functions or, alternatively, arbitrary random variables. Later, McIver and Morgan [MMS96; MM05] coined the term *expectation* for such functions. While we adhere to this terminology, we remind the reader that expectations should not be confused with *expected values*, i.e., the mean of a random variable; rather, *expectations are the quantitative analog to predicates*.

Throughout this thesis, we study two classes of expectations: The set of *expectations* $\mathbb{E}$ enables measuring arbitrary properties of interest, such as the value of a variable, the size of the heap, etc. Moreover, the subset of *one-bounded expectations* $\mathbb{E}_{\leq 1}$ is sufficient to reason about the probability of events, e.g., the probability of terminating without dereferencing a null pointer.

Definition 6.1 (Expectations [MM05, p. 16], [1]) The set $\mathbb{E}$ of *expectations* is

$$\mathbb{E} \triangleq \{ \mathrm{X} \mid \mathrm{X} \colon \mathbf{SHPairs} \to \mathbb{R}^{\infty}_{\geq 0} \}.$$

Moreover, the set $\mathbb{E}_{\leq 1}$ of *one-bounded expectations* is defined as

$$\mathbb{E}_{\leq 1} \triangleq \{ \mathrm{Y} \mid \mathrm{Y} \colon \mathbf{SHPairs} \to [0, 1] \}.$$

An expectation thus maps every stack-heap pair to a non-negative real number or infinity whereas a one-bounded expectation maps to reals within $[0, 1]$. Analogously to predicates (cf. Section 4.3), we lift substitution (Definition 2.6), scoping (Definition 3.6), and the set of relevant variables (Definition 2.39) to expectations. That is, we define:

- $\mathrm{X}[x/E] \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathrm{X}(\mathfrak{s}[x/E(\mathfrak{s})], \mathfrak{h})$,
- $\mathrm{X}[+\boxdot] \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathrm{X}(\mathfrak{s}[+\boxdot], \mathfrak{h})$,
- $\mathrm{X}[-\boxdot] \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathrm{X}(\mathfrak{s}[-\boxdot], \mathfrak{h})$, and
- $\mathbf{Vars}(\mathrm{X}) \triangleq \{ x \in \mathbf{Vars} \mid \exists (\mathfrak{s}, \mathfrak{h})\, \exists u, v \colon \mathrm{X}(\mathfrak{s}[x/u], \mathfrak{h}) \neq \mathrm{X}(\mathfrak{s}[x/v], \mathfrak{h}) \}$.

Example 6.2 Let us consider a few examples of common expectations:

- For every real number $v \in \mathbb{R}^{\infty}_{\geq 0}$, the constant function

$$\lambda(\mathfrak{s},\mathfrak{h}).\, v$$

  is an expectation in $\mathbb{E}$, but not necessarily in $\mathbb{E}_{\leq 1}$. By slight abuse of notation, we often identify an expectation with the expression that is evaluated in a given stack-heap pair to compute its value. For instance, we write v instead of $\lambda(\mathfrak{s},\mathfrak{h}).\, v$.

- The *heap size quantity* **size** , which measures the number of allocated memory cells, is an expectation. It is defined as

$$\textbf{size} \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, |\mathsf{dom}(\mathfrak{h})|,$$

  where $|\mathsf{dom}(\mathfrak{h})|$ denotes the cardinality of the domain of heap $\mathfrak{h}$.

- The absolute value of any variable, say x, i.e., the function

$$|x| \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, |\mathfrak{s}(x)|,$$

  is an expectation. The same holds for the function

$$x^2 + 2 \cdot x \cdot y + y^2 = (x+y)^2 \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, (\mathfrak{s}(x) + \mathfrak{s}(y))^2.$$

- The function $x \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, \mathfrak{s}(x)$ is *not* an expectation, because variable x might evaluate to a negative number. Since we often consider programs in which variables are assigned non-negative values only, it is convenient to use x as if it were an expectation. While it is possible to reason about mixed-sign expectations, it requires more heavy technical machinery [KK17].

- Expectations may depend on both the stack and the heap. For example, an expectation that measures the number of variables representing pointers to null is given by:

$$\#_{\mapsto 0} \triangleq \lambda(\mathfrak{s},\mathfrak{h}).\, |\{\, x \in \textbf{Vars} \mid \mathfrak{s}(x) \in \mathsf{dom}(\mathfrak{h}) \text{ and } \mathfrak{h}(\mathfrak{s}(x)) = 0 \,\}|.$$

Both $\mathbb{E}$ and $\mathbb{E}_{\leq 1}$ form a partial order where the ordering $\preceq$ amounts to pointwise application of the canonical ordering $\leq$ on real numbers with infinity, i.e.,

$$X \preceq Y \qquad \text{iff} \qquad \forall(\mathfrak{s},\mathfrak{h}) \in \textbf{SHPairs}\colon \quad X(\mathfrak{s},\mathfrak{h}) \leq Y(\mathfrak{s},\mathfrak{h}).$$

In contrast to McIver and Morgan [MM05], our expectations are *not* bounded by an arbitrary real number. Consequently,

Lemma 6.3 ([1]) Both $\langle\, \mathbb{E}, \preceq \,\rangle$ and $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$ are complete lattices with least element 0 and greatest element ∞ and 1, respectively.

Proof. Clearly, $\langle\, \mathbb{E}, \preceq \,\rangle$ is a partial order. For every $S \subseteq \mathbb{E}$, the least upper bound of S is then given by the expectation $\lambda(\mathfrak{s}, \mathfrak{h}).\ \sup \{\, X(\mathfrak{s}, \mathfrak{h}) \mid X \in S \,\}$. The least element of $\mathbb{E}$ is the least upper bound of $\emptyset$, i.e., $\lambda(\mathfrak{s}, \mathfrak{h}).\ \sup \emptyset = 0$. Moreover, the greatest element of $\mathbb{E}$ is the least upper bound of $\mathbb{E}$, i.e., ∞. The proof for one-bounded expectations $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$ is analogous. □

In the context of program verification, we are often tasked with discharging *quantitative entailments*, i.e., inequalities between expectations of the form $X \preceq Y$. Clearly, it is neither feasible nor desirable to verify the inequality $X \preceq Y$ by explicitly proving that $X(\mathfrak{s}, \mathfrak{h}) \leq Y(\mathfrak{s}, \mathfrak{h})$ holds for every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$. Throughout the remainder of this chapter, we thus study various algebraic properties of expectations and their connectives. Apart from elementary facts about first-order logic and real-valued functions, which are both well-established in mathematics and supported by common theorem provers, these properties form the foundation of a sound proof system that allows us to establish quantitative entailments step by step through syntactic manipulations of expectations rather than reverting to the underlying formal definitions.

We present most of our results with respect to the domain $\langle\, \mathbb{E}, \preceq \,\rangle$. That is, we develop a logic for reasoning about expected values. A logic for reasoning about probabilities of events can be constructed analogously by considering the complete lattice $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$, which covers a subset of $\langle\, \mathbb{E}, \preceq \,\rangle$, instead.

6.2 Predicates & Separation Logic Atoms

We regularly use the *Iverson bracket* [GKP94, p. 24] notation $[P]$ to associate a predicate $P\colon$ **SHPairs** $\to \{\, \mathsf{true}, \mathsf{false} \,\}$ with its indicator function. Formally, $[P]$ is defined as the function

$$[P] : \textbf{SHPairs} \to \{\, 0, 1 \,\}, \qquad [P]\,(\mathfrak{s}, \mathfrak{h}) \triangleq \begin{cases} 1, & \text{if } P\,(\mathfrak{s}, \mathfrak{h}) = \mathsf{true} \\ 0, & \text{if } P\,(\mathfrak{s}, \mathfrak{h}) = \mathsf{false}. \end{cases}$$

Every predicate P is a (one-bounded) expectation, namely its Iverson bracket $[P]$, that maps only to the set $\{\, 0, 1 \,\}$. The expectation $[x \neq y]$ thus evaluates to one iff variable x is evaluated to a different value than y. Let us take the atoms of separation logic introduced in Section 4.3.1 as a source of further examples:

The *empty-heap predicate* $[\mathbf{emp}]$ evaluates to 1 iff the heap is empty, i.e.,

$$[\mathbf{emp}] \;\triangleq\; \lambda(\mathfrak{s},\mathfrak{h}).\; \begin{cases} 1, & \text{if } \mathsf{dom}(\mathfrak{h}) = \emptyset \\ 0, & \text{otherwise.} \end{cases}$$

Moreover, the *points-to predicate* $[E \mapsto E']$, which evaluates to 1 iff the heap consists of exactly one memory cell with address E and content E', is defined as

$$\left[E \mapsto E'\right] \;\triangleq\; \lambda(\mathfrak{s},\mathfrak{h}).\; \begin{cases} 1, & \text{if } \mathfrak{h} = \{\, E\,(\mathfrak{s}) :: E'\,(\mathfrak{s}) \,\} \\ 0, & \text{otherwise,} \end{cases}$$

where $\{\, E\,(\mathfrak{s}) :: E'\,(\mathfrak{s}) \,\}$ is our notation for heaps introduced in Section 4.1.2. Since an expectation is a random variable, the expected value of $[P]$ corresponds to the probability that predicate P is evaluated to true. Hence, for every expectation $\mathrm{X} \in \mathbb{E}_{\leq 1}$, we interpret $\mathrm{X}(\mathfrak{s},\mathfrak{h}) = 0 = [\mathsf{false}]\,(\mathfrak{s},\mathfrak{h})$ as "X is violated in $(\mathfrak{s},\mathfrak{h})$". Moreover, $\mathrm{X}(\mathfrak{s},\mathfrak{h}) > 0$ means that "X is satisfied in $(\mathfrak{s},\mathfrak{h})$ to some degree".

There are multiple ways to model standard logical connectives between predicates as arithmetic operations on expectations. For instance, we interpret a *standard disjunction* $P \vee Q$ between two predicates as the pointwise maximum between their Iverson brackets. This is backward compatible because

$$[P \vee Q] \;=\; \max\{\,[P],\,[Q]\,\},$$

where the maximum of two expectations X and Y is defined as

$$\max\{\,\mathrm{X},\mathrm{Y}\,\} \;\triangleq\; \lambda(\mathfrak{s},\mathfrak{h}).\; \max\{\,\mathrm{X}(\mathfrak{s},\mathfrak{h}),\mathrm{Y}(\mathfrak{s},\mathfrak{h})\,\}.$$

Notice that we neither claim nor attempt to define a reasonable notion of disjunction for arbitrary expectations $\mathrm{X},\mathrm{Y} \in \mathbb{E}$. One should thus *not* read $\max\{\mathrm{X},\mathrm{Y}\}$ as a disjunction. After all, what is the disjunction between constant values 4 and 17 supposed to mean? While it is an interesting research direction to develop suitable generalizations of logical operators in the realm of expectations, they are not needed for reasoning about probabilistic pointer programs.

We model *standard conjunction* $P \wedge Q$ as the pointwise product between their Iverson brackets.[2] Again, this is backward compatible because

$$[P \wedge Q] \;=\; [P] \cdot [Q]\,,$$

where the multiplication of expectations X and Y is defined as

$$\mathrm{X} \cdot \mathrm{Y} \;\triangleq\; \lambda(\mathfrak{s},\mathfrak{h}).\, \mathrm{X}(\mathfrak{s},\mathfrak{h}) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}).$$

[2] A reader familiar with algebra might be reminded of the Viterbi semiring $([0,1], max, \cdot)$ in which disjunction is interpreted as maximum and conjunction is interpreted as multiplication (cf. [Kui97]).

We could alternatively have chosen to compute a minimum instead of a product. For reasoning about probabilistic programs, however, we often reason about weighted sums in which multiplication is more natural. In particular, there is a useful interpretation of products if one factor is an arbitrary expectation, e.g., $[P] \cdot \mathsf{X}$: We measure the value of random variable X in stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ if and only if $(\mathfrak{s}, \mathfrak{h})$ satisfies predicate P. Otherwise, we measure value 0. For instance, the expectation $[x \neq 0] \cdot \mathbf{size}$ measures the size of the heap if and only if variable x is not equal to 0. More generally, we obtain a first batch of simple rules for reasoning about the expectations $\mathbf{emp}$, $[E \mapsto E']$, and $\mathbf{size}$:

Theorem 6.4 (Conjunction Laws for Atoms) The following properties hold:

(a) $[\mathbf{emp}] \cdot \mathbf{size} = 0$,

(b) $[E \mapsto E'] \cdot \mathbf{size} \;=\; [E \mapsto E'] \;=\; [E \neq 0] \cdot [E \mapsto E']$, and

(c) $[E_1 \mapsto E_2] \cdot [E_3 \mapsto E_4] \;=\; [E_1 = E_3] \cdot [E_2 = E_4] \cdot [E_1 \mapsto E_2]$.

Law 6.4 (a) states that the size of the empty heap is zero. Similarly, Theorem 6.4 (b) states that the size of a single memory cell is one and an allocated address is not equal to null. Put together, both rules yield that $[\mathbf{emp}]$ and $[E \mapsto E']$ cannot be satisfied simultaneously:

$$
\begin{aligned}
& [\mathbf{emp}] \cdot \big[E \mapsto E'\big] \\
= \;& [\![\text{ Theorem 6.4 (b) }]\!] \\
& [\mathbf{emp}] \cdot \big(\big[E \mapsto E'\big] \cdot \mathbf{size}\big) \\
= \;& [\![\text{ elementary algebra (associativity of } \cdot) \text{ }]\!] \\
& \big([\mathbf{emp}] \cdot \mathbf{size}\big) \cdot \big[E \mapsto E'\big] \\
= \;& [\![\text{ Theorem 6.4 (a) }]\!] \\
& 0 \cdot \big[E \mapsto E'\big] \\
= \;& [\![\text{ elementary algebra }]\!] \\
& 0.
\end{aligned}
$$

Theorem 6.4 (c) expresses that two specifications of the same memory cell refer to the same addresses and values.

Furthermore, given a one-bounded expectation $\mathsf{Z} \in \mathbb{E}_{\leq 1}$, we often consider convex sums of the form

$$\mathsf{Z} \cdot \mathsf{X} + (1 - \mathsf{Z}) \cdot \mathsf{Y} \;\triangleq\; \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathsf{Z}(\mathfrak{s}, \mathfrak{h}) \cdot \mathsf{X}(\mathfrak{s}, \mathfrak{h}) + (1 - \mathsf{Z}(\mathfrak{s}, \mathfrak{h})) \cdot \mathsf{Y}(\mathfrak{s}, \mathfrak{h}).$$

In particular, if Z is a predicate, i.e., Z maps to $\{0, 1\}$, then the above sum represents a choice between two mutually exclusive expectations: We either

have $\mathrm{Z}(\mathfrak{s},\mathfrak{h}) = 1$ and measure $\mathrm{X}(\mathfrak{s},\mathfrak{h})$ or $1 - \mathrm{Z}(\mathfrak{s},\mathfrak{h}) = 1$ and measure $\mathrm{Y}(\mathfrak{s},\mathfrak{h})$. In this case, the convex sum coincides with a maximum:

Lemma 6.5 For all expectations X and Y with $\mathrm{X} \cdot \mathrm{Y} = 0$, we have

$$\max\{\,\mathrm{X},\mathrm{Y}\,\} \;=\; \mathrm{X} + \mathrm{Y}.$$

We do not explicitly list other standard facts about arithmetic operations on expectations such as distributivity of multiplication and addition.

6.3 Separating Connectives between Expectations

We now develop quantitative versions of separating conjunction and implication.

6.3.1 Quantitative Separating Conjunction

Let us first recall the classical case: For two predicates P and Q, their separating conjunction $P \star Q$ is given by

$$\mathfrak{s},\mathfrak{h} \models P \star Q \quad \text{iff} \quad \exists \mathfrak{h}_1,\mathfrak{h}_2\colon\ \ \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \text{ and } \mathfrak{s},\mathfrak{h}_1 \models P \text{ and } \mathfrak{s},\mathfrak{h}_2 \models Q.$$

In words, a state $(\mathfrak{s},\mathfrak{h})$ satisfies $P \star Q$ iff there exists a partition of the heap $\mathfrak{h}$ into two heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$ such that the stack $\mathfrak{s}$ together with heap $\mathfrak{h}_1$ satisfies P, and $\mathfrak{s}$ together with $\mathfrak{h}_2$ satisfies Q.

How should we connect two expectations X and Y in a similar fashion? As standard conjunction corresponds to multiplication, we need to find a partition of the heap $\mathfrak{h}$ into $\mathfrak{h}_1 \uplus \mathfrak{h}_2$, measure X in $\mathfrak{h}_1$, measure Y in $\mathfrak{h}_2$, and finally multiply these two measured quantities. The naïve approach, i.e.,

$$\big(\mathrm{X} \star \mathrm{Y}\big)(\mathfrak{s},\,\mathfrak{h}) \quad = \quad \exists\, \mathfrak{h}_1,\mathfrak{h}_2\colon\ \ [\mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2] \cdot \mathrm{X}(\mathfrak{s},\mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2),$$

is not meaningful. At the very least, it is ill-typed. Moreover, what precisely determined quantity would the above express? After all, the existentially quantified partition of $\mathfrak{h}$ need not be unique.

Our key redemptive insight here is that existential quantification ($\exists$) should correspond to max. From an algebraic perspective, this corresponds to the usual interpretation of existential quantifiers in a complete Heyting algebra or Boolean algebra as a disjunction (cf. [Sco08] for an overview) which—at least for predicates—we interpreted as a maximum in the realm of expectations.

In first-order logic, the effect of the quantified predicate $\exists v\colon P(v)$ is so-to-speak to "maximize the truth of $P(v)$" by a suitable choice of v. In QSL, instead of truth, we maximize a quantity: Out of all partitions of the heap $\mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2$, we choose one—out of finitely many for any given $\mathfrak{h}$—that maximizes the product $\mathrm{X}(\mathfrak{s},\mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)$. We thus define the quantitative $\star$ as follows:

Definition 6.6 (Quantitative Separating Conjunction [1]) The *quantitative separating conjunction* $X \star Y$ of two expectations $X, Y \in \mathbb{E}$ is defined as

$$X \star Y \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\ \max\left\{ X(\mathfrak{s}, \mathfrak{h}_1) \cdot Y(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \right\}.$$

As a first sanity check, we observe that the quantitative separating conjunction still prevents unintended aliasing (see also Section 4.3.2):

Theorem 6.7 (Alias-Prevention) $[E \mapsto E'] \star [E \mapsto E''] = 0.$

Example 6.8 Recall from Example 6.2 the expectations **size** and $\#_{\mapsto 0}$ measuring the number of allocated addresses and the number of variables storing the address of a null pointer, respectively. To sharpen our intuition, let us evaluate the quantitative separating conjunction $\#_{\mapsto 0} \star \mathbf{size}$ in a concrete stack heap pair $(\mathfrak{s}, \mathfrak{h})$: $\mathfrak{s}$ is given by $\mathfrak{s}(x) = \mathfrak{s}(y) = 4$ and $\mathfrak{s}(z) = 7$; all other variables are evaluated to zero. Moreover, $\mathfrak{h}$ is defined as

$$\mathfrak{h} \triangleq \{ 1 :: 3 \} \uplus \{ 4 :: 0 \} \uplus \{ 7 :: 0 \}.$$

By definition of the quantitative separating conjunction, we have

$$(\#_{\mapsto 0} \star \mathbf{size})(\mathfrak{s}, \mathfrak{h}) = \max\left\{ \#_{\mapsto 0}(\mathfrak{s}, \mathfrak{h}_1) \cdot \mathbf{size}(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \right\}.$$

Figure 6.1, page 174, depicts the evaluation of the above product for every possible partitioning of $\mathfrak{h}$ into heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$. In particular, the maximum is attained for $\mathfrak{h}_1 = \{ 4 :: 0 \}$ and $\mathfrak{h}_2 = \{ 1 :: 3 \} \uplus \{ 7 :: 0 \}$. Hence,

$$(\#_{\mapsto 0} \star \mathbf{size})(\mathfrak{s}, \mathfrak{h}) = 4.$$

Notice, however, that the maximum is in general not attained for a unique partitioning of $\mathfrak{h}$ into heaps $\mathfrak{h}_1$ and $\mathfrak{h}_2$. The expectation $3 \star 7$, for instance, evaluates to 21 for any partition of the heap.

Furthermore, for predicates, the quantitative separating conjunction is compatible with the classical separating conjunction.

Theorem 6.9 (Backward Compatibility of $\star$ [1]) For all predicates $P, Q \in$ **Pred** and stack-heap pairs $(\mathfrak{s}, \mathfrak{h}) \in$ **SHPairs**, we have:

(a) $([P] \star [Q])(\mathfrak{s}, \mathfrak{h}) \in \{0, 1\}$, and

(b) $([P] \star [Q])(\mathfrak{s}, \mathfrak{h}) = 1$ holds in QSL iff $\mathfrak{s}, \mathfrak{h} \models P \star Q$ holds.

$\mathsf{dom}(\mathfrak{h}_1)$	$\mathsf{dom}(\mathfrak{h}_2)$	$\#_{\mapsto 0}(\mathfrak{s},\mathfrak{h}_1)$	$\mathbf{size}(\mathfrak{s},\mathfrak{h}_2)$	$\#_{\mapsto 0}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathbf{size}(\mathfrak{s},\mathfrak{h}_2)$
$\emptyset$	$\{1,4,7\}$	0	3	0
$\{1\}$	$\{4,7\}$	0	2	0
$\{4\}$	$\{1,7\}$	2	2	4
$\{7\}$	$\{1,4\}$	1	2	2
$\{1,4\}$	$\{7\}$	2	1	2
$\{1,7\}$	$\{4\}$	1	1	1
$\{4,7\}$	$\{1\}$	3	1	3
$\{1,4,7\}$	$\emptyset$	3	0	0

Figure 6.1: Evaluation of $\#_{\mapsto 0}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathbf{size}(\mathfrak{s},\mathfrak{h}_2)$ for all partitions $\mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2$ of the heap $\mathfrak{h}=\{1::3\}\uplus\{4::0\}\uplus\{7::0\}$. Moreover, the stack $\mathfrak{s}$ is given by $\mathfrak{s}(x)=\mathfrak{s}(y)=4$ and $\mathfrak{s}(z)=7$.

Proof. To show 6.9 (a), consider the following:

$$\big([P]\star[Q]\big)(\mathfrak{s},\mathfrak{h}) \;=\; \max\Big\{\underbrace{\underbrace{\underbrace{[P](\mathfrak{s},\mathfrak{h}_1)}_{\in\{0,1\}}\cdot\underbrace{[Q](\mathfrak{s},\mathfrak{h}_2)}_{\in\{0,1\}}}_{\in\{0,1\}}}_{\in 2^{\{0,1\}}\setminus\{\emptyset\}}}_{\in\{0,1\}} \;\Big|\; \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\Big\} \;\in\; \{0,1\}.$$

Moreover, to prove 6.9 (b), consider the following:

$\mathfrak{s},\mathfrak{h}\models P\star Q$
iff ⟦ Definition of $\star$ in classical separation logic ⟧
$\exists\mathfrak{h}_1,\mathfrak{h}_2\colon \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2$ and $\mathfrak{s},\mathfrak{h}_1\models P$ and $\mathfrak{s},\mathfrak{h}_2\models Q$
iff ⟦ Definition of Iverson bracket ⟧
$\exists\mathfrak{h}_1,\mathfrak{h}_2\colon \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2$ and $[P](\mathfrak{s},\mathfrak{h}_1)=1$ and $[Q](\mathfrak{s},\mathfrak{h}_2)=1$
iff ⟦ $[P](\mathfrak{s},\mathfrak{h}_1),[Q](\mathfrak{s},\mathfrak{h}_2)\in\{0,1\}$ ⟧
$\exists\mathfrak{h}_1,\mathfrak{h}_2\colon \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2$ and $[P](\mathfrak{s},\mathfrak{h}_1)\cdot[Q](\mathfrak{s},\mathfrak{h}_2)=1$
iff ⟦ there exist only finitely many partitions of $\mathfrak{h}$ into $\mathfrak{h}_1\uplus\mathfrak{h}_2$ ⟧

$$\begin{aligned} & \max\{\,[P]\,(\mathfrak{s},\mathfrak{h}_1)\cdot[Q]\,(\mathfrak{s},\mathfrak{h}_2)\mid \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\,\} \;=\; 1 \\ \text{iff}\quad & [\![\ \text{Definition of } \star \text{ in QSL}\ ]\!] \\ & ([P]\star[Q])\,(\mathfrak{s},\mathfrak{h}) \;=\; 1. \end{aligned}$$

Hence, the quantitative separating conjunction is backward compatible. □

6.3.2 Quantitative Separating Implication

Recall from Definition 4.15 the definition of separating implication of two predicates P and Q in classical separation logic:

$$\mathfrak{s},\mathfrak{h}\models P\mathrel{-\!\!*} Q \quad \text{iff} \quad \forall\mathfrak{h}'\colon\ \big(\mathfrak{h}\,\#\,\mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}'\models P\big)\ \text{implies}\ \mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}'\models Q,$$

where $\mathfrak{h}\,\#\,\mathfrak{h}'$ denotes that the domains of $\mathfrak{h}$ and $\mathfrak{h}'$ are disjoint. So $(\mathfrak{s},\mathfrak{h})$ satisfies $P\mathrel{-\!\!*} Q$ iff the following holds: Whenever we can find a heap $\mathfrak{h}'$ disjoint from $\mathfrak{h}$ such that stack $\mathfrak{s}$ together with heap $\mathfrak{h}'$ satisfies P, then $\mathfrak{s}$ together with the conjoined heap $\mathfrak{h}\uplus\mathfrak{h}'$ must satisfy Q. In other words: We measure the truth of Q in *extended* heaps $\mathfrak{h}\uplus\mathfrak{h}'$ *given that* $\mathfrak{h}'$ satisfies P. That is, every heap extension is *conditioned* on the truth value of satisfying P.

How should we connect expectations X and Y in a similar fashion? Since the least element of our complete lattice, i.e. 0, corresponds to false when evaluating a predicate, we interpret *satisfying an expectation* X as measuring some positive quantity, i.e. $\mathrm{X}(\mathfrak{s},\mathfrak{h})>0$. Intuitively, $\mathrm{X}\mathrel{-\!\!*}\mathrm{Y}$ then intends to measure Y in extended heaps conditioned on "how much" the extensions satisfy X.

As for the universal quantifier, our key insight is now that—dually to $\exists$ corresponding to max—$\forall$ should correspond to min: Whereas in first-order logic the predicate $\forall v\colon P(v)$ "minimizes the truth of $P(v)$" by requiring that $P(v)$ must be true for all choices of v, in QSL we minimize a quantity: Out of all heap extensions $\mathfrak{h}'$ disjoint from $\mathfrak{h}$ that satisfy expectation X, we choose an extension $\mathfrak{h}'$ that minimizes the quantity $\mathrm{Y}(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')$ weighted by "how much" X is satisfied. Since, for a given expectation X and heap $\mathfrak{h}$, there may be infinitely many (or no) admissible choices of heaps $\mathfrak{h}'$, we define the quantitative separating implication by an infimum:

Definition 6.10 (Quantitative Separating Implication) The *quantitative separating implication* $\mathrm{X}\mathrel{-\!\!*}\mathrm{Y}$ of expectations $\mathrm{X},\mathrm{Y}\in\mathbb{E}$ is defined as

$$\lambda(\mathfrak{s},\mathfrak{h}).\ \inf\left\{\ \frac{\mathrm{Y}(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')}{\mathrm{X}(\mathfrak{s},\mathfrak{h}')}\ \middle|\ \begin{array}{l}\mathfrak{h}\,\#\,\mathfrak{h}' \text{ and } \mathrm{X}(\mathfrak{s},\mathfrak{h}')>0\\ \text{and }\ (\ \mathrm{X}(\mathfrak{s},\mathfrak{h}')<\infty \text{ or } \mathrm{Y}(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')<\infty\ )\end{array}\ \right\},$$

where, for all $r\in\mathbb{R}_{\geq 0}$, we set $r/\infty=0$ and $\infty/r=\infty$.

The constraints on X and Y are needed to avoid the corner case ∞/∞. Alternatively, one could define $\infty/\infty = \infty$: If all choices for $\mathfrak{h}'$ yield both $X(\mathfrak{s}, \mathfrak{h}') = \infty$ and $Y(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') = \infty$, then we take an infimum over the empty set; this results in ∞. Since it is common in probability theory, we assume that $0 \cdot \infty = \infty \cdot 0 = 0$. Hence, no additional constraint to cover this case is necessary.

Notice that our quantitative interpretation of the separating implication is also sensible from an algebraic perspective: As in the classical case (cf. Theorem 4.20), the quantitative separating implication can alternatively be characterized as the adjoint of the quantitative separating conjunction. Further details are discussed in Section 6.3.3 alongside other properties of $\star$ and $\mathrel{-\!\!\star}$.

We often use predicates on the left-hand side of separating implications, i.e., we consider $[P] \mathrel{-\!\!\star} X$. In this case, the quantitative version of $\mathrel{-\!\!\star}$ has a simpler interpretation which served as a definition of $\mathrel{-\!\!\star}$ in [1]: We measure X in the smallest extension of the heap that is compatible with predicate P.

Theorem 6.11 (Guarded Quantitative Separating Implication) For all predicates $P \in \mathbf{Pred}$ and expectations $X \in \mathbb{E}$, we have

$$[P] \mathrel{-\!\!\star} X \;=\; \lambda(\mathfrak{s}, \mathfrak{h}).\ \inf \left\{ X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P \right\}.$$

Proof. For every predicate P, we have

$$\begin{aligned}
&[P] \mathrel{-\!\!\star} X \\
&= \ [\![\text{ Definition of } \mathrel{-\!\!\star}]\!] \\
&\quad \lambda(\mathfrak{s}, \mathfrak{h}).\ \inf \left\{ \frac{X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}')}{[P](\mathfrak{s}, \mathfrak{h}')} \;\middle|\; \begin{array}{l} \mathfrak{h} \# \mathfrak{h}' \text{ and } [P](\mathfrak{s}, \mathfrak{h}') > 0 \\ \text{and } ([P](\mathfrak{s}, \mathfrak{h}') < \infty \text{ or } X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') < \infty) \end{array} \right\} \\
&= \ [\![\ [P](\mathfrak{s}, \mathfrak{h}') \in \{0, 1\} < \infty \ ]\!] \\
&\quad \lambda(\mathfrak{s}, \mathfrak{h}).\ \inf \left\{ \frac{X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}')}{[P](\mathfrak{s}, \mathfrak{h}')} \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } [P](\mathfrak{s}, \mathfrak{h}') = 1 \right\} \\
&= \ [\![\text{ Definition of Iverson bracket; elementary algebra }]\!] \\
&\quad \lambda(\mathfrak{s}, \mathfrak{h}).\ \inf \left\{ X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P \right\}.
\end{aligned}$$

□

Example 6.12 Let $\mathfrak{s}$ be an arbitrary stack. Moreover, let

$$\mathfrak{h} \triangleq \{ 1 :: 3 \} \uplus \{ 4 :: 0 \} \uplus \{ 7 :: 0 \}.$$

The expectation $[5 \mapsto 17] \mathrel{-\!\!\star} \mathbf{size}$ first extends the heap by the unique heap captured by $[5 \mapsto 17]$ and then measures the size of the extended heap. Hence, the measured size is one plus the heap's original size:

$$
\begin{aligned}
& ([5 \mapsto 17] \mathrel{-\!\!*} \mathbf{size})(\mathfrak{s}, \mathfrak{h}) \\
= \; & [\![\text{ Theorem 6.11 }]\!] \\
& \inf \{\, \mathbf{size}(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models [5 \mapsto 17] \,\} \\
= \; & [\![\text{ Definition of } [5 \mapsto 17] \,]\!] \\
& \inf \{\, \mathbf{size}(\mathfrak{s}, \mathfrak{h} \uplus \{\, 5 :: 17 \,\}) \mid \mathfrak{h} \# \{\, 5 :: 17 \,\} \,\} \\
= \; & [\![\text{ Definition of } \mathfrak{h}\text{; elementary algebra }]\!] \\
& \mathbf{size}(\mathfrak{s}, \{\, 1 :: 3 \,\} \uplus \{\, 4 :: 0 \,\} \uplus \{\, 7 :: 0 \,\} \uplus \{\, 5 :: 17 \,\}) \\
= \; & [\![\text{ Definition of } \mathbf{size} \,]\!] \\
& 4 \\
= \; & [\![\, \mathbf{size}(\mathfrak{s}, \mathfrak{h}) = 3 \text{; elementary algebra }]\!] \\
& 1 + \mathbf{size}(\mathfrak{s}, \mathfrak{h}).
\end{aligned}
$$

As a second example, we evaluate the expectation

$$(2 \cdot \mathbf{size}) \mathrel{-\!\!*} \mathbf{size}$$

in the same stack-heap pair $(\mathfrak{s}, \mathfrak{h})$. Notice that neither the left-hand side nor the right-hand side of the above quantitative separating implication is a predicate. We thus evaluate this expectation by applying Definition 6.10:

$$
\begin{aligned}
& ((2 \cdot \mathbf{size}) \mathrel{-\!\!*} \mathbf{size})\,(\mathfrak{s}, \mathfrak{h}) \\
= \; & [\![\text{ Definition 6.10 }]\!] \\
& \inf \left\{ \frac{\mathbf{size}(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}')}{(2 \cdot \mathbf{size})(\mathfrak{s}, \mathfrak{h}')} \;\middle|\; \begin{array}{l} \mathfrak{h} \# \mathfrak{h}' \text{ and } (2 \cdot \mathbf{size})(\mathfrak{s}, \mathfrak{h}') > 0 \\ \text{and } \big(\; (2 \cdot \mathbf{size})(\mathfrak{s}, \mathfrak{h}') < \infty \\ \qquad\qquad \text{or } \mathbf{size}(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') < \infty \;\big) \end{array} \right\} \\
= \; & [\![\text{ Definition of } \mathbf{size}\text{; elementary algebra }]\!] \\
& \inf \left\{ \frac{|\mathsf{dom}(\mathfrak{h} \uplus \mathfrak{h}')|}{2 \cdot |\mathsf{dom}(\mathfrak{h}')|} \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } 2 \cdot |\mathsf{dom}(\mathfrak{h}')| > 0 \right\} \\
= \; & [\![\text{ Definition of } \mathfrak{h}\text{; elementary algebra }]\!] \\
& \inf \left\{ \frac{3 + |\mathsf{dom}(\mathfrak{h}')|}{2 \cdot |\mathsf{dom}(\mathfrak{h}')|} \;\middle|\; 1, 4, 7 \notin \mathsf{dom}(\mathfrak{h}') \text{ and } 2 \cdot |\mathsf{dom}(\mathfrak{h}')| > 0 \right\} \\
= \; & [\![\text{ substitute } |\mathsf{dom}(\mathfrak{h})| \text{ by } n \in \mathbb{N} \,]\!] \\
& \inf \{\, {}^{3+n}\!/_{2 \cdot n} \mid n \in \mathbb{N}_{>0} \,\} \\
= \; & [\![\text{ elementary algebra }]\!] \\
& \inf \{\, {}^{1}\!/_{2} + {}^{3}\!/_{2 \cdot n} \mid n \in \mathbb{N}_{>0} \,\} \;=\; {}^{1}\!/_{2}.
\end{aligned}
$$

Unfortunately, backward compatibility for quantitative separating implication comes with certain reservations: Suppose for a particular stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ that there exists no heap $\mathfrak{h}'$ such that $\mathfrak{s}, \mathfrak{h}' \models P$. Then the set

$$\{\, X(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P \,\}$$

is empty, and the greatest lower bound (within our domain $\mathbb{E}$) of the empty set is ∞ and not 1. In particular, $\mathsf{false} \mathrel{-\!\!*} Q = \mathsf{true}$ holds in classical separation logic, but $0 \mathrel{-\!\!*} [Q] = \infty$ holds in QSL. Since $0 = [\mathsf{false}]$, but $\infty \neq [\mathsf{true}]$, backward compatibility of $\mathrel{-\!\!*}$ breaks for arbitrary expectations.

As a silver lining, true is the greatest element in the complete lattice of predicates and correspondingly ∞ is the greatest element in $\mathbb{E}$. In this light, the above is not surprising. In fact, if we restrict ourselves to the domain $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$, which is used to reason about probabilities, we achieve full compatibility with the classical separating implication.

Theorem 6.13 (Backward Compatibility of $\mathrel{-\!\!*}$ [1]) In the domain $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$ of one-bounded expectations, it holds that, for all predicates $P, Q \in \mathbf{Pred}$ and stack-heap pairs $(\mathfrak{s}, \mathfrak{h}) \in \mathbf{SHPairs}$, we have:

(a) $([P] \mathrel{-\!\!*} [Q])\,(\mathfrak{s}, \mathfrak{h}) \in \{0, 1\}$, and

(b) $([P] \mathrel{-\!\!*} [Q])\,(\mathfrak{s}, \mathfrak{h}) = 1$ holds in QSL iff $\mathfrak{s}, \mathfrak{h} \models P \mathrel{-\!\!*} Q$ holds.

Proof. For proving 6.13 (a), consider the following:

$$\begin{aligned}
& ([P] \mathrel{-\!\!*} [Q])\,(\mathfrak{s}, \mathfrak{h}) \\
= \;& [\![\text{ Theorem 6.11 }]\!] \\
& \underbrace{\inf \{\underbrace{[Q]\,(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}')}_{\in \{0,1\}} \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P\}}_{\in 2^{\{0,1\}}} \\
\in \;& [\![\text{ The domain is restricted to } \mathbb{E}_{\leq 1}, \text{ i.e. } \inf \emptyset = 1 \,]\!] \\
& \{0, 1\}.
\end{aligned}$$

Moreover, for proving 6.13 (b), consider the following:

$$\begin{aligned}
& ([P] \mathrel{-\!\!*} [Q])\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff} \quad & [\![\text{ Theorem 6.11 }]\!] \\
& \underbrace{\inf \{\, [Q]\,(\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}') \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s}, \mathfrak{h}' \models P \,\}}_{= X} \;=\; 1
\end{aligned}$$

iff ⟦ $X = 0$ iff exists $\mathfrak{h}'$ such that $\mathfrak{h} \# \mathfrak{h}'$ and $\mathfrak{s}, \mathfrak{h} \models P$ and $\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \not\models Q$ ⟧
$\neg \exists \mathfrak{h}' \colon \mathfrak{h} \# \mathfrak{h}'$ and $\mathfrak{s}, \mathfrak{h} \models P$ and $\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \not\models Q$
iff ⟦ Pushing negation inside ⟧
$\forall \mathfrak{h}' \colon \neg \mathfrak{h} \# \mathfrak{h}'$ or $\mathfrak{s}, \mathfrak{h} \not\models P$ or $\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \models Q$
iff ⟦ First-order logic ⟧
$\forall \mathfrak{h}' \colon (\mathfrak{h} \# \mathfrak{h}'$ and $\mathfrak{s}, \mathfrak{h} \models P)$ implies $\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \models Q$
iff ⟦ Definition of $\mathrel{-\!\!*}$ in classical separation logic ⟧
$\mathfrak{s}, \mathfrak{h} \models P \mathrel{-\!\!*} Q$. □

6.3.3 Properties of Quantitative Separating Connectives

Besides backward compatibility, the quantitative separating connectives of QSL are well-behaved in the sense that they satisfy most properties of their classical counterparts. To justify this claim, let us revisit the properties of separating conjunction and implication studied in Section 4.3.2. First, the monoidal structure underlying the classical separating conjunction with **emp** as neutral element remains unchanged. Hence, we obtain the following analog to Theorem 4.17:

Theorem 6.14 (Monoidicity of $\star$ [1]) $(\mathbb{E}, \star, [\mathbf{emp}])$ is a commutative monoid. That is, for all expectations $X, Y, Z \in \mathbb{E}$, the following holds:

(a) *Associativity:* $X \star (Y \star Z) = (X \star Y) \star Z$.

(b) *Neutrality of* $[\mathbf{emp}]$: $X \star [\mathbf{emp}] = [\mathbf{emp}] \star X = X$.

(c) *Commutativity:* $X \star Y = Y \star X$.

Furthermore, the quantitative separating conjunction remains monotone:

Theorem 6.15 (Monotonicity of $\star$ [1]) For all expectations $X, Y, Z \in \mathbb{E}$:

$$X \preceq Z \quad \text{implies} \quad X \star Y \preceq Z \star Y.$$

Proof. Assume that $X \preceq Z$. Then, for every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, we have:

$$\begin{aligned}
& (X \star Y)(\mathfrak{s}, \mathfrak{h}) \\
= \; & \text{⟦ Definition of } \star \text{ ⟧} \\
& \max \{ X(\mathfrak{s}, \mathfrak{h}_1) \cdot Y(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \} \\
\leq \; & \text{⟦ by premise } (X \preceq Z) \text{ and monotonicity of } \cdot \text{ ⟧}
\end{aligned}$$

$$\begin{aligned} & \max\{\, Z(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2) \mid \mathfrak{h}=\mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \\ =\ & [\![\text{ Definition of } \star \,]\!] \\ & (Z \star Y)(\mathfrak{s},\mathfrak{h}). \end{aligned}$$ □

Similarly, most subdistributivity laws (cf. Theorem 4.19) are valid in the quantitative case. The only exception is subdistributivity between separating and standard conjunction: For a stack-heap pair $(\mathfrak{s},\mathfrak{h})$ with $|\mathsf{dom}(\mathfrak{h})| = 2$, we have

$$(\mathbf{size} \star (1\cdot 1))(\mathfrak{s},\mathfrak{h}) \;=\; 2 \;\neq\; 4 \;=\; ((\mathbf{size}\star 1)\cdot(\mathbf{size}\star 1))\,(\mathfrak{s},\mathfrak{h}).$$

To restore subdistributivity between $\star$ and $\cdot$, it suffices to require that the separating conjunction's left-hand side is a predicate.

Theorem 6.16 ((Sub)distributivity of $\star$ [1]) For all expectations $X, Y, Z \in \mathbb{E}$ and predicates $P \in \mathbf{Pred}$, we have:

(a) $X \star \max\{\, Y,\, Z \,\} \;=\; \max\{\, X\star Y,\, X\star Z \,\}$,

(b) $X \star \min\{\, Y,\, Z \,\} \;\preceq\; \min\{\, X\star Y,\, X\star Z \,\}$,

(c) $X \star (Y + Z) \;\preceq\; (X\star Y) + (X\star Z)$,

(d) $[P] \star (Y\cdot Z) \;\preceq\; \big([P]\star Y\big)\cdot\big([P]\star Z\big)$,

(e) if $x \notin \mathbf{Vars}(X)$, then $X \star \sup_{v\in\mathbb{Z}} Y\,[x/v] \;=\; \sup_{v\in\mathbb{Z}}(X\star Y)\,[x/v]$, and

(f) if $x \notin \mathbf{Vars}(X)$, then $X \star \inf_{v\in\mathbb{Z}} Y\,[x/v] \;\preceq\; \inf_{v\in\mathbb{Z}}(X\star Y)\,[x/v]$.

As in the classical case, most operations are only subdistributive over $\star$ because there is in general no unique partitioning of the heap. For example, evaluating the expectation $X \triangleq 1 \star ([1\mapsto 2] + [2\mapsto 3])$ in a stack-heap pair $(\mathfrak{s},\mathfrak{h})$, where $\mathfrak{s}$ is an arbitrary stack and $\mathfrak{h} = \{\, 1 :: 2, 3 \,\}$, yields

$$X(\mathfrak{s},\mathfrak{h}) = \max\{\, 1(\mathfrak{s},\mathfrak{h}_1)\cdot([1\mapsto 2](\mathfrak{s},\mathfrak{h}_2) + [2\mapsto 3](\mathfrak{s},\mathfrak{h}_2)) \mid \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \,\} = 1,$$

because we have to choose either $\mathfrak{h}_2 = \{\, 1::2 \,\}$ or $\mathfrak{h}_2 = \{\, 2::3 \,\}$ such that the maximum is attained. However, if we evaluate the expanded expectation $Y \triangleq (1\star[1\mapsto 2]) + (1\star[2\mapsto 3])$ in the same stack-heap pair $(\mathfrak{s},\mathfrak{h})$, we obtain

$$\begin{aligned} Y(\mathfrak{s},\mathfrak{h}) \;=\; & \underbrace{\max\{\, 1(\mathfrak{s},\mathfrak{h}_1)\cdot[1\mapsto 2](\mathfrak{s},\mathfrak{h}_2) \mid \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \,\}}_{=\,1 \text{ (choose } \mathfrak{h}_2=\{\,1::2\,\})} \\ & + \underbrace{\max\{\, 1(\mathfrak{s},\mathfrak{h}_1)\cdot[2\mapsto 3](\mathfrak{s},\mathfrak{h}_2) \mid \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \,\}}_{=\,1 \text{ (choose } \mathfrak{h}_2=\{\,2::3\,\})} \;=\; 2. \end{aligned}$$

Unfortunately, multiplication is general not subdistributive with respect to the quantitative separating conjunction, i.e., there is no law for expectations of the form $Z \cdot (X \star Y)$. We thus present specialized rules for the cases in which Z coincides with one of the atomic expectations $[\mathbf{emp}]$, $[E \mapsto E']$, or **size**:

Theorem 6.17 ((Sub)distributivity of · for Atomic Expectations) For all expectations $X, Y \in \mathbb{E}$, we have:

(a) $[\mathbf{emp}] \cdot (X \star Y) \;=\; ([\mathbf{emp}] \cdot X) \star ([\mathbf{emp}] \cdot Y)$,

(b) $[E \mapsto E'] \cdot (X \star Y) = \max\{([E \mapsto E'] \cdot X) \star ([\mathbf{emp}] \cdot Y),$
$([\mathbf{emp}] \cdot X) \star ([E \mapsto E'] \cdot Y)\}$, and

(c) $\mathbf{size} \cdot (X \star Y) \;\preceq\; ((\mathbf{size} \cdot X) \star Y) + (X \star (\mathbf{size} \cdot Y))$.

Rule 6.17 (c) intuitively states that the size of the heap captured by $X \star Y$ coincides with the sum of the size of the heaps captured by X, i.e., $X \cdot \mathbf{size}$, and Y, i.e., $Y \cdot \mathbf{size}$. However, we have to account for parts of the heap whose size is not measured, i.e., Y if we measure the size of X and vice versa. These parts are "absorbed" by an additional separating conjunction with Y and X, respectively.

Returning to properties of classical separation logic, adjointness of $\star$ and $\mathrel{-\!\!\star}$ is preserved as well. Hence, the following variant of Theorem 4.20 holds:

Theorem 6.18 (Adjointness) For all expectations $X, Y, Z \in \mathbb{E}$, we have

$$X \star Y \;\preceq\; Z \quad \text{iff} \quad X \;\preceq\; Y \mathrel{-\!\!\star} Z.$$

Proof. A detailed proof is found in Appendix C.2. □

A quantitative variant of modus ponens (cf. Corollary 4.21) then immediately holds because it is a special case of adjointness:

Corollary 6.19 (Modus Ponens) $X \star (X \mathrel{-\!\!\star} Y) \;\preceq\; Y$.

Proof. Let X and Y be expectations in $\mathbb{E}$. Then, consider the following:

$$\begin{array}{ll} & X \star (X \mathrel{-\!\!\star} Y) \;\preceq\; Y \\ \text{iff} & \llbracket \text{ Theorem 6.14 (c) } \rrbracket \\ & (X \mathrel{-\!\!\star} Y) \star X \;\preceq\; Y \\ \text{iff} & \llbracket \text{ Theorem 6.18 with X being } X \mathrel{-\!\!\star} Y, \text{ Y being X, and Z being Y } \rrbracket \\ & X \mathrel{-\!\!\star} Y \;\preceq\; X \mathrel{-\!\!\star} Y. \end{array}$$

□

Furthermore, adjointness allows us to derive laws for the separating implication. For example, we can derive a monotonicity property if we fix either the left-hand side or the right-hand side of $\mathrel{-\!\!*}$:

Theorem 6.20 (Monotonicity of $\mathrel{-\!\!*}$) For all expectations $X, Y, X', Y' \in \mathbb{E}$, the quantitative separating implication is:

(a) monotone with respect to its right-hand side, i.e.,

$$Y \preceq Y' \quad \text{implies} \quad X \mathrel{-\!\!*} Y \preceq X \mathrel{-\!\!*} Y', \text{ and}$$

(b) antitone with respect to its left-hand side, i.e.,

$$X \preceq X' \quad \text{implies} \quad X' \mathrel{-\!\!*} Y \preceq X \mathrel{-\!\!*} Y.$$

Proof. For Theorem 6.20 (a), assume $Y \preceq Y'$ and consider the following:

$$\begin{aligned}
& X \mathrel{-\!\!*} Y \preceq X \mathrel{-\!\!*} Y' \\
\text{iff} \quad & [\![\text{ Theorem 6.18 }]\!] \\
& (X \mathrel{-\!\!*} Y) \star X \preceq Y' \\
\text{iff} \quad & [\![\text{ Theorem 6.14 (c); Corollary 6.19 }]\!] \\
& Y \preceq Y',
\end{aligned}$$

where the last line holds by assumption.

For Theorem 6.20 (b), assume $X \preceq X'$ and consider the following:

$$\begin{aligned}
& X' \mathrel{-\!\!*} Y \preceq X \mathrel{-\!\!*} Y \\
\text{iff} \quad & [\![\text{ Theorem 6.18 }]\!] \\
& (X' \mathrel{-\!\!*} Y) \star X \preceq Y \\
\text{iff} \quad & [\![\text{ by assumption and Theorem 6.15 }]\!] \\
& (X' \mathrel{-\!\!*} Y) \star X' \preceq Y \\
\text{iff} \quad & [\![\text{ Theorem 6.14 (c); Corollary 6.19 }]\!] \\
& Y \preceq Y,
\end{aligned}$$

where the last line holds trivially. □

Finally, we collect *super*distributivity laws for the quantitative separating implication. Most of these laws are shown using adjointness between $\star$ and $\mathrel{-\!\!*}$ and the subdistributivity laws for $\star$ presented in Theorem 6.16. The equalities presented below, however, require additional work.

Theorem 6.21 ((Super)distributivity of $\multimap$) For all expectations $X, Y, Z \in \mathbb{E}$ and predicates $P \in \mathbf{Pred}$, we have:

(a) $X \multimap \max\{Y, Z\} \;\succeq\; \max\{X \multimap Y, X \multimap Z\}$,

(b) $X \multimap \min\{Y, Z\} \;=\; \min\{X \multimap Y, X \multimap Z\}$,

(c) $X \multimap (Y + Z) \;\succeq\; (X \multimap Y) + (Y \multimap Z)$,

(d) $[P] \multimap (Y \cdot Z) \;\succeq\; ([P] \multimap Y) \cdot ([P] \cdot Z)$,

(e) if $x \notin \mathbf{Vars}(X)$, then

$$X \multimap (\sup_{v \in \mathbb{Z}} Y\,[x/v]) \;\succeq\; \sup_{v \in \mathbb{Z}} (X \multimap Y)\,[x/v]\,, \text{ and}$$

(f) if $x \notin \mathbf{Vars}(X)$, then

$$X \multimap \inf_{v \in \mathbb{Z}} Y\,[x/v] \;=\; \inf_{v \in \mathbb{Z}} (X \star Y)\,[x/v]\,.$$

Proof. Apart from the two equalities, each property can be proven by exploiting adjointness of the quantitative separating conjunction and the quantitative separating implication. The proof of the two equalities is analogous to the proof of Theorem 6.16. □

6.4 Fragments of Expectations

To effectively reason about expectations, let us study a few fragments of expectations in which quantitative separating conjunction and implication enjoy additional properties. Most of these fragments have already been considered for classical separation logic in Section 4.3.3.

6.4.1 Pure Expectations

Recall that a predicate is *pure* iff its truth does not depend on the heap but only on the stack. Analogously, we call an expectation X *pure* iff

$$\forall\, \mathfrak{s}\; \forall\, \mathfrak{h}_1\; \forall\, \mathfrak{h}_2 \colon \quad X(\mathfrak{s}, \mathfrak{h}_1) \;=\; X(\mathfrak{s}, \mathfrak{h}_2).$$

Examples of pure expectations include constant expectations, e.g., 0 or 1, and expectations over program variables, e.g., x^2 or $[x = y + 2]$. For pure expectations, the quantitative separating connectives are compliant with the algebraic rules for pure predicates which we presented in Theorem 4.22:

Theorem 6.22 (Laws for Pure Expectations [1]) Let $X \in \mathbb{E}$ be a pure expectation. Moreover, let $Y, Z \in \mathbb{E}$ be arbitrary expectations. Then:

(a) $X \cdot Y \preceq X \star Y$,

(b) $(X \cdot Y) \star Z = X \cdot (Y \star Z)$, and

(c) $X \mathrel{-\!\!\star} Y \;\preceq\; [X = 0] \cdot \infty + [X \neq 0] \cdot {}^{Y}\!/_{X}$.

Moreover, if both X and Y are pure expectations, then:

(d) $X \cdot Y \;=\; X \star Y$, and

(e) $X \mathrel{-\!\!\star} Y \;=\; [X = 0] \cdot \infty + [X \neq 0] \cdot {}^{Y}\!/_{X}$.

When considering one-bounded expectations rather than general ones, the same rules apply if ∞ is replaced by 1, i.e., the greatest element of $\mathbb{E}_{\leq 1}$. Rules 6.22 (c) and 6.22 (e) might seem unintuitive at first. It is thus noteworthy that, for one-bounded expectations, $[X = 0] \cdot 1 + [X \neq 0] \cdot {}^{Y}\!/_{X}$ is backward compatible to the standard notion of logical implication. That is, for all predicates P and Q and all stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$, we have:

$$
\begin{aligned}
& \mathfrak{s}, \mathfrak{h} \models P \Rightarrow Q \\
\text{iff} \quad & [\![\text{ elementary logic }]\!] \\
& \mathfrak{s}, \mathfrak{h} \models \neg P \vee Q \\
\text{iff} \quad & [\![\text{ backward compatibility of maximum and Iverson bracket }]\!] \\
& \max\{\, [\neg P],\, [Q] \,\}\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff} \quad & [\![\text{ elementary algebra }]\!] \\
& \max\{\, [\neg P],\, [P] \cdot {}^{[Q]}\!/_{[P]} \,\}\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff} \quad & [\![\text{ Lemma 6.5 }]\!] \\
& ([\neg P] \,+\, [P] \cdot {}^{[Q]}\!/_{[P]})\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff} \quad & [\![\text{ elementary algebra }]\!] \\
& ([[P] = 0] \cdot 1 + [[P] \neq 0] \cdot {}^{[Q]}\!/_{[P]})\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1.
\end{aligned}
$$

6.4.2 Precise Expectations

Analogously to classical separation logic, we call an expectation $X \in \mathbb{E}$ *domain-exact* iff for all stacks $\mathfrak{s} \in \textbf{Stacks}$ and heaps $\mathfrak{h}, \mathfrak{h}' \in \textbf{Heaps}$,

$$
\big(\, X(\mathfrak{s}, \mathfrak{h}) > 0 \text{ and } X(\mathfrak{s}, \mathfrak{h}') > 0 \,\big) \quad \text{implies} \quad \mathsf{dom}(\mathfrak{h}) \;=\; \mathsf{dom}(\mathfrak{h}').
$$

As presented in Theorem 4.23, Reynolds [Rey02] showed that full distributivity of the separating conjunction is achieved for domain-exact predicates. O'Hearn,

Yang, and Reynolds later proved that the more general set of "precise" predicates characterizes all predicates such that separating and standard conjunction distribute (cf. [OYR09; OYR04, Definition 5]). For expectations, it is convenient to consider a similar notion:

Definition 6.23 (Precise Expectations) An expectation $X \in \mathbb{E}$ is *precise* iff

$$\forall \mathfrak{s}\, \forall \mathfrak{h} \colon \quad \left| \left\{ \mathfrak{h}' \in \textbf{Heaps} \;\middle|\; \mathfrak{h}' \subseteq \mathfrak{h} \text{ and } X(\mathfrak{s}, \mathfrak{h}') > 0 \right\} \right| \;\leq\; 1.$$

In other words, for every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, there is at most one heap $\mathfrak{h}'$ included in $\mathfrak{h}$ such that $X(\mathfrak{s}, \mathfrak{h}')$ does not vanish. Notice that the expectation

$$[x = 0] \cdot [\textbf{emp}] + [x \neq 0] \cdot [1 \mapsto 2]$$

is precise, but not domain-exact. The converse direction, however, holds:

Lemma 6.24 Every domain-exact expectation is precise.

Proof. Let $X \in \mathbb{E}$ be domain-exact. Moreover, fix some stack-heap pair $(\mathfrak{s}, \mathfrak{h})$. Towards a contradiction, let us assume that

$$\left| \left\{ \mathfrak{h}' \in \textbf{Heaps} \;\middle|\; \mathfrak{h}' \subseteq \mathfrak{h} \text{ and } X(\mathfrak{s}, \mathfrak{h}') > 0 \right\} \right| > 1.$$

Then, there exist two distinct heaps $\mathfrak{h}', \mathfrak{h}'' \subseteq \mathfrak{h}$ such that $X(\mathfrak{s}, \mathfrak{h}') > 0$ and $X(\mathfrak{s}, \mathfrak{h}'') > 0$. Since X is domain-exact, we have $\mathsf{dom}(\mathfrak{h}') = \mathsf{dom}(\mathfrak{h}'')$.

Both heaps are obtained by restricting the domain of $\mathfrak{h}$ to $\mathsf{dom}(\mathfrak{h}')$ and $\mathsf{dom}(\mathfrak{h}'')$, respectively. Hence, we have $\mathfrak{h}' = \mathfrak{h}''$. However, this contradicts our assumption that $\mathfrak{h}'$ and $\mathfrak{h}''$ are distinct heaps. □

Consequently, the expectations 0, $[\textbf{emp}]$, and $[E \mapsto E']$ are precise whereas 1, ∞, and $[x \neq y]$ are not. Furthermore, precise expectations can be conveniently constructed from existing ones according to the following rules:

- If X and Y are precise expectations, then $X \star Y$ is a precise expectation.
- If X and Y are precise expectations, then $X \cdot Y$ is a precise expectation.
- If X and Y are precise expectations and Q is a pure predicate, then the convex sum $[Q] \cdot X + [\neg Q] \cdot Y$ is a precise expectation.

Similar results have been shown for precise predicates in [OYR09]. For precise expectations, the quantitative separating conjunction becomes fully distributive analogously to domain-exact predicates (cf. Theorem 4.23).

Theorem 6.25 (Distributivity of $\star$ for Precise Expectations) Let $\mathsf{X} \in \mathbb{E}$ be a precise expectation and $P \in \mathbf{Pred}$ be a precise predicate. Then, for all expectations $\mathsf{Y}, \mathsf{Z} \in \mathbb{E}$, we have:

(a) $\mathsf{X} \star \min\{\mathsf{Y}, \mathsf{Z}\} \;=\; \min\{\mathsf{X} \star \mathsf{Y}, \mathsf{X} \star \mathsf{Z}\}$,

(b) $\mathsf{X} \star (\mathsf{Y} + \mathsf{Z}) \;=\; (\mathsf{X} \star \mathsf{Y}) + (\mathsf{X} \star \mathsf{Z})$,

(c) $[P] \star (\mathsf{Y} \cdot \mathsf{Z}) \;=\; \big([P] \star \mathsf{Y}\big) \cdot \big([P] \star \mathsf{Z}\big)$, and

(d) if $x \notin \mathbf{Vars}(\mathsf{X})$, then $\mathsf{X} \star \inf_{v \in \mathbb{Z}} \mathsf{Y}\,[x/v] \;=\; \inf_{v \in \mathbb{Z}} (\mathsf{X} \star \mathsf{Y})\,[x/v]$.

Analogously, multiplication with **size** becomes fully distributive:

Theorem 6.26 (Distributivity of $\cdot$ for size) For all precise expectations $\mathsf{X} \in \mathbb{E}$ and expectations $\mathsf{Y} \in \mathbb{E}$, we have

$$\mathbf{size} \cdot (\mathsf{X} \star \mathsf{Y}) \;=\; ((\mathbf{size} \cdot \mathsf{X}) \star \mathsf{Y}) + (\mathsf{X} \star (\mathbf{size} \cdot \mathsf{Y})).$$

6.4.3 Strictly-Exact Expectations

Unfortunately, neither preciseness nor domain-exactness yields full distributivity of the quantitative separating implication. For example, consider the *allocated pointer predicate* $[E \mapsto -]$ which states that exactly address E is allocated on the heap. Formally, it is syntactic sugar for $\sup_{v \in \mathbb{Z}} [E \mapsto v]$, i.e.,

$$[E \mapsto -] \;\triangleq\; \sup_{v \in \mathbb{Z}} [E \mapsto v] \;=\; \lambda(\mathfrak{s}, \mathfrak{h}).\; \begin{cases} 1, & \text{if } \mathsf{dom}(\mathfrak{h}) = \{\, E\,(\mathfrak{s}) \,\} \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, $[E \mapsto -]$ is domain-exact. Now, consider the expectation

$$\mathsf{X} \;\triangleq\; [1 \mapsto -] \mathrel{-\!\!\star} \big(\sup_{u \in \mathbb{Z}} [u \neq 2] \cdot [1 \mapsto u] \cdot 4 + [1 \mapsto 2] \cdot 3\big).$$

Then, for any stack $\mathfrak{s}$ and the empty heap $\mathfrak{h}_\emptyset$, we have $\mathsf{X}(\mathfrak{s}, \mathfrak{h}_\emptyset) = 3$ as extending the heap by $\{\, 1 :: 2 \,\}$ minimizes the total measured value. However, if we extend the heap by $\{\, 1 :: 2 \,\}$, we obtain

$$\big([1 \mapsto -] \mathrel{-\!\!\star} \big(\sup_{u \in \mathbb{Z}} [u \neq 2] \cdot [1 \mapsto u] \cdot 4\big)\big)(\mathfrak{s}, \mathfrak{h}_\emptyset) \;=\; 0.$$

Analogously, by extending the heap by $\{\, 1 :: 1 \,\}$, we obtain

$$\big([1 \mapsto -] \mathrel{-\!\!\star} \big([1 \mapsto 2] \cdot 3\big)\big(\mathfrak{s}, \mathfrak{h}_\emptyset\big) \;=\; 0.$$

Hence, $\mathrel{-\!\!\star}$ does not distributive over $+$. To achieve full distributivity of $\mathrel{-\!\!\star}$, we consider a more restricted fragment (cf. [Yan01, Section 7.4]):

Definition 6.27 (Strictly-exact Expectations) We call an expectation $\mathrm{X} \in \mathbb{E}$ *strictly-exact* iff

$$\forall \mathfrak{s}\colon \quad |\{\, \mathfrak{h} \in \mathbf{Heaps} \mid \mathrm{X}(\mathfrak{s},\mathfrak{h}) > 0 \,\}| \;\leq\; 1.$$

In other words, for every stack, there is at most one heap such that a strictly-exact expectation does not vanish. For example, the points-to predicate $[E \mapsto E']$ is strictly-exact whereas the predicate $[E \mapsto -]$ is not. Moreover, the separating conjunction of two strictly-exact expectations is strictly-exact as well. Hence, consecutive blocks of memory as specified by

$$\begin{aligned}[E \mapsto E_1, E_2, \ldots, E_n] \;&\triangleq\; [E \mapsto E_1] \star [E+1 \mapsto E_2] \star \ldots \star [E+n-1 \mapsto E_n] \\ &= \lambda(\mathfrak{s},\mathfrak{h}).\, \begin{cases} 1, & \text{if } \mathfrak{h} = \{\, E\,(\mathfrak{s}) :: E_1\,(\mathfrak{s}), E_2\,(\mathfrak{s}), \ldots, E_n\,(\mathfrak{s}) \,\} \\ 0, & \text{otherwise.} \end{cases}\end{aligned}$$

are strictly-exact. Intuitively, for strictly-exact expectations, the quantitative separating implication becomes fully distributive because all possible extensions of the heap are uniquely determined.

Theorem 6.28 (Distributivity of $\mathrel{-\!\!\star}$ for strictly-exact Expectations) Let X be a strictly-exact expectation and $P \in \mathbf{Pred}$ be a strictly-exact predicate. Then, for all expectations $\mathrm{Y}, \mathrm{Z} \in \mathbb{E}$, we have:

(a) $\mathrm{X} \mathrel{-\!\!\star} \max\{\,\mathrm{Y},\,\mathrm{Z}\,\} \;=\; \max\{\,\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y},\, \mathrm{X} \mathrel{-\!\!\star} \mathrm{Z}\,\}$,

(b) $\mathrm{X} \mathrel{-\!\!\star} (\mathrm{Y}+\mathrm{Z}) \;=\; (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}) + (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Z})$,

(c) $[P] \mathrel{-\!\!\star} (\mathrm{Y} \cdot \mathrm{Z}) \;=\; ([P] \mathrel{-\!\!\star} \mathrm{Y}) \cdot ([P] \mathrel{-\!\!\star} \mathrm{Z})$, and

(d) if $x \notin \mathbf{Vars}\,(\mathrm{X})$, then $\mathrm{X} \mathrel{-\!\!\star} (\sup_{v\in\mathbb{Z}} \mathrm{Y}\,[x/v]) = \sup_{v\in\mathbb{Z}} (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y})\,[x/v]$.

6.4.4 Intuitionistic Expectations

Recall from Section 4.3.3.3 that a predicate P is *intuitionistic* iff

$$\forall \mathfrak{s} \in \mathbf{Stacks}\; \forall \mathfrak{h}, \mathfrak{h}' \in \mathbf{Heaps}\colon \quad (\mathfrak{h} \subseteq \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h} \models P) \quad \text{implies} \quad \mathfrak{s},\mathfrak{h}' \models P.$$

So as we extend the heap from $\mathfrak{h}$ to $\mathfrak{h}'$, an intuitionistic predicate can only get *"more true"*. Analogously, in QSL, as we extend the heap from $\mathfrak{h}$ to $\mathfrak{h}'$, the quantity measured by an *intuitionistic expectation* can only *increase*. Formally, an expectation X is called *intuitionistic* iff

$$\forall \mathfrak{s} \in \mathbf{Stacks}\; \forall \mathfrak{h}, \mathfrak{h}' \in \mathbf{Heaps}\colon \quad \mathfrak{h} \subseteq \mathfrak{h}' \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}) \;\leq\; \mathrm{X}(\mathfrak{s},\mathfrak{h}').$$

A natural example of an intuitionistic expectation is the heap size quantity

$$\textbf{size} \;=\; \lambda(\mathfrak{s},\mathfrak{h}).\; |\mathsf{dom}(\mathfrak{h})|.$$

As presented in Theorem 4.24, Reynolds [Rey02] provided a systematic way to construct intuitionistic predicates: For any predicate P,

1. $P \star \mathsf{true}$ is the strongest intuitionistic predicate weaker than P, and
2. $\mathsf{true} \mathrel{-\!\!\star} P$ is the weakest intuitionistic predicate stronger than P.

In QSL, we obtain an analogous result for arbitrary expectations:

Theorem 6.29 (Tightest Intuitionistic Expectations [1]) For every expectation $X \in \mathbb{E}$, the following holds:

(a) $X \star 1$ is the smallest intuitionistic expectation that is at least X, i.e.,

- $X \star 1$ is intuitionistic,
- $X \preceq X \star 1$, and
- for all intuitionistic Y satisfying $X \preceq Y$, we have $X \star 1 \preceq Y$.

(b) $1 \mathrel{-\!\!\star} X$ is the greatest intuitionistic expectation that is at most X, i.e.,

- $1 \mathrel{-\!\!\star} X$ is intuitionistic,
- $1 \mathrel{-\!\!\star} X \preceq X$, and
- for all intuitionistic Y satisfying $Y \preceq X$, we have $Y \preceq 1 \mathrel{-\!\!\star} X$.

The previous theorem allows us to formalize two additional proof rules to deal with quantitative separating implications:

Theorem 6.30 (Intuitionistic Resolution of $\mathrel{-\!\!\star}$) Let $P \in \textbf{Pred}$ be a strictly-exact predicate. For all (not necessarily intuitionistic) predicates $Q \in \textbf{Pred}$ and expectations $X \in \mathbb{E}$, we have:

(a) $[P] \star ([P] \mathrel{-\!\!\star} X) = ([P] \star 1) \cdot X$, and

(b) $[Q] \mathrel{-\!\!\star} X = [Q] \mathrel{-\!\!\star} (([Q] \star 1) \cdot X)$.

Intuitively, 6.30 (a) is an exact version of modus ponens (Corollary 6.19), i.e., $X \star (X \mathrel{-\!\!\star} Y) \preceq Y$. That is, carving out the heap $\mathfrak{h}$ specified by predicate P and then inserting the same heap again before evaluating X is equivalent to evaluating X provided that $\mathfrak{h}$ is included in the original heap. Rule 6.30 (b) states that, as long as the left-hand side of $\mathrel{-\!\!\star}$ is a predicate, an intuitionistic version of the same predicate also holds on the right-hand side of $\mathrel{-\!\!\star}$.

We frequently use an intuitionistic version of the points-to predicate $[E \mapsto E']$. Hence, as in classical separation logic, we define the *contains-pointer predicate* $[E \hookrightarrow E']$ as syntactic sugar for $[E \mapsto E'] \star 1$, i.e.,

$$[E \hookrightarrow E'] \triangleq [E \mapsto E'] \star 1 = \lambda(\mathfrak{s}, \mathfrak{h}). \begin{cases} 1, & \text{if } \{ E(\mathfrak{s}) :: E'(\mathfrak{s}) \} \subseteq \mathfrak{h} \\ 0, & \text{otherwise.} \end{cases}$$

Analogously, $[E \hookrightarrow -]$ is syntactic sugar for $[E \mapsto -] \star 1$, i.e.,

$$[E \hookrightarrow -] \triangleq [E \mapsto -] \star 1 = \lambda(\mathfrak{s}, \mathfrak{h}). \begin{cases} 1, & \text{if } \{ E(\mathfrak{s}) \} \in \mathsf{dom}(\mathfrak{h}) \\ 0, & \text{otherwise.} \end{cases}$$

The contains-pointer predicate allows us to conveniently state additional proof rules for reasoning about $\mathrel{-\!\!\star}$ and the **size** quantity:

Theorem 6.31 (Contains-Pointer Laws) For all expectations $X, Y \in \mathbb{E}$:

(a) $[E \mapsto E'] \star \mathbf{size} = [E \hookrightarrow E'] \cdot (\mathbf{size} - 1)$,

(b) $[E \mapsto E'] \mathrel{-\!\!\star} \mathbf{size} = 1 + \mathbf{size} + [E \hookrightarrow -] \cdot \infty$,

(c) $[E \mapsto E'] \mathrel{-\!\!\star} ([E \mapsto E'] \star X) = X + [E \hookrightarrow -] \cdot \infty$, and

(d) $[E \hookrightarrow E'] \cdot (X \star Y) = \max\{ ([E \hookrightarrow E'] \cdot X) \star Y, X \star ([E \hookrightarrow E'] \cdot Y) \}$.

The first two rules illustrate the role of the quantitative separating connectives $\star$ and $\mathrel{-\!\!\star}$: The quantitative separating conjunction $\star$ removes a part of the heap before its size is measured and consequently decreases the size of the remaining heap. Dually, the quantitative separating implication $\mathrel{-\!\!\star}$ extends the heap and hence increases its size. If the heap cannot be extended appropriately, the infimum in the definition of $\mathrel{-\!\!\star}$ yields ∞.

To understand Theorem 6.31 (c), let us consider a special case of adjointness (cf. Theorem 6.18), i.e., for all $X, Y \in \mathbb{E}$,

$$Y \star (Y \star X) \succeq X.$$

Intuitively, this means that extending the heap according to Y and then immediately disposing the extension again before evaluating X is greater than or equal to evaluating X in the original heap. Rule 6.31 (c) then explains what is missing on the right-hand side of the above inequality: If the heap cannot be extended as specified by Y, then the quantitative separating implication evaluates to ∞.

Finally, Theorem 6.31 (d) lifts the property in Theorem 6.17 (b) from points-to predicates to contains-pointer predicates.

Example 6.32 The rules collected throughout this chapter provide us with a small proof system for reasoning about expectations. Let us apply these rules to derive another rule which will turn out convenient for reasoning about pointer programs: For all expectations $X \in \mathbb{E}$, we have

$$[E \mapsto E_1] \star ([E \mapsto E_2] \mathrel{-\!\!\star} ([E \mapsto E_2] \star X)) \;=\; [E \mapsto E_1] \star X.$$

Intuitively, this rule captures a typical scenario for pointer programs: Initially, address E is allocated and stores value E_1. This value is updated to E_2 by removing the memory cell described by $[E \mapsto E_1]$ and then inserting a new memory cell $[E \mapsto E_2]$. Finally, to avoid memory leaks, address E is disposed, i.e., we remove the memory cell $[E \mapsto E_2]$ again. Our proof rule then states that measuring X in the remaining heap coincides with measuring X in the initial heap after removing memory cell $[E \mapsto E_1]$. We thus avoid the heap update formalized by the quantitative separating implication. Now, to formally prove that the above rule is correct, consider the following:

$$\begin{aligned}
&[E \mapsto E_1] \star ([E \mapsto E_2] \mathrel{-\!\!\star} ([E \mapsto E_2] \star X)) \\
=\;& [\![\text{ Theorem 6.31 (c) }]\!] \\
&[E \mapsto E_1] \star (X + [E \hookrightarrow -] \cdot \infty) \\
=\;& [\![\text{ Theorem 6.25 (b) }]\!] \\
&([E \mapsto E_1] \star X) + ([E \mapsto E_1] \star ([E \hookrightarrow -] \cdot \infty)) \\
=\;& [\![\text{ Theorem 6.14 (c) }]\!] \\
&([E \mapsto E_1] \star X) + (([E \hookrightarrow -] \cdot \infty) \star [E \mapsto E_1]) \\
=\;& [\![\text{ elementary algebra (distributivity of } \cdot)]\!] \\
&([E \mapsto E_1] \star X) + ((\infty \cdot [E \hookrightarrow -]) \star [E \mapsto E_1]) \\
=\;& [\![\text{ Theorem 6.22 (b) }]\!] \\
&([E \mapsto E_1] \star X) + \infty \cdot ([E \hookrightarrow -] \star [E \mapsto E_1]) \\
=\;& [\![\text{ By definition of } [E \hookrightarrow -]]\!] \\
&([E \mapsto E_1] \star X) + \infty \cdot ([E \mapsto -] \star 1 \star [E \mapsto E_1]) \\
=\;& [\![\text{ Theorem 6.14 (c) }]\!] \\
&([E \mapsto E_1] \star X) + \infty \cdot ([E \mapsto E_1] \star [E \mapsto -] \star 1) \\
=\;& [\![\text{ By definition of } [E \mapsto -]]\!] \\
&([E \mapsto E_1] \star X) + \infty \cdot ([E \mapsto E_1] \star (\sup_{v \in \mathbb{Z}} [E \mapsto v]) \star 1) \\
=\;& [\![\text{ Theorem 6.14 (a) }]\!]
\end{aligned}$$

$$
\begin{aligned}
& ([E \mapsto E_1] \star X) + \infty \cdot ((\,[E \mapsto E_1] \star (\sup_{v \in \mathbb{Z}} [E \mapsto v])) \star 1) \\
= \;& [\![\text{ introduce fresh variable } y \text{ for } v \;]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot (([E \mapsto E_1] \star (\sup_{v \in \mathbb{Z}} [E \mapsto y]\,[y/v])) \star 1) \\
= \;& [\![\text{ Theorem 6.16 (e) }]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot ((\sup_{v \in \mathbb{Z}} ([E \mapsto E_1] \star [E \mapsto y])\,[y/v]) \star 1) \\
= \;& [\![\text{ Theorem 6.7 }]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot ((\sup_{v \in \mathbb{Z}} 0\,[y/v]) \star 1) \\
= \;& [\![\text{ apply substitution }]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot ((\sup_{v \in \mathbb{Z}} 0) \star 1) \\
= \;& [\![\text{ elementary algebra }]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot (0 \star 1) \\
= \;& [\![\text{ Theorem 6.22 (d) }]\!] \\
& ([E \mapsto E_1] \star X) + \infty \cdot (0 \cdot 1) \\
= \;& [\![\text{ elementary algebra (assuming } 0 \cdot \infty = 0) \,]\!] \\
& [E \mapsto E_1] \star X.
\end{aligned}
$$

Hence, our proposed proof rule is sound.

To reason effectively about expectations, it is convenient to consider further proof rules that reflect common idioms encountered when reasoning about pointer programs. These rules are collected and verified in Appendix E; all of them are derived from the rules presented in this chapter. Furthermore, a compact reference sheet with all proof rules is found in Appendix D. A reader might find it convenient to consult this sheet while reading formal proofs.

6.5 Conservativity of QSL as an Assertion Language

The assertion language of QSL conservatively extends the assertion language of classical separation logic. As long as we adhere to the extensional approach this is immediate: For every separation logic predicate P it suffices to compute the (one-bounded) expectation $[P]$. Then, by definition of Iverson brackets, it holds for all stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$ that

- $[P]\,(\mathfrak{s}, \mathfrak{h}) \in \{\,0, 1\,\}$, and
- $[P]\,(\mathfrak{s}, \mathfrak{h}) = 1$ iff $\mathfrak{s}, \mathfrak{h} \models P$.

Predicate P in separation logic	Expectation $\texttt{qsl}[\![P]\!]$ in QSL
P (atomic formula)	$[P]$
$Q[x/E]$	$\texttt{qsl}[\![Q]\!][x/E]$
$Q\,[+\boxdot]$	$\texttt{qsl}[\![Q]\!]\,[+\boxdot]$
$Q\,[-\boxdot]$	$\texttt{qsl}[\![Q]\!]\,[-\boxdot]$
$\neg Q$	$1 - \texttt{qsl}[\![Q]\!]$
$\exists x\, Q$	$\sup_{v\in\mathbb{Z}} \texttt{qsl}[\![Q]\!]\,[x/v]$
$\forall x\, Q$	$\inf_{v\in\mathbb{Z}} \texttt{qsl}[\![Q]\!]\,[x/v]$
$Q \wedge R$	$\texttt{qsl}[\![Q]\!] \cdot \texttt{qsl}[\![R]\!]$
$Q \mathbin{\star} R$	$\texttt{qsl}[\![Q]\!] \mathbin{\star} \texttt{qsl}[\![R]\!]$
$Q \vee R$	$\max\{\, \texttt{qsl}[\![Q]\!],\ \texttt{qsl}[\![R]\!] \,\}$
$Q \mathrel{-\!\!\star} R$	$\texttt{qsl}[\![Q]\!] \mathrel{-\!\!\star} \texttt{qsl}[\![R]\!]$

Table 6.1: Embedding of common separation logic connectives into QSL.

In this section, we briefly argue that an explicit embedding of classical separation logic into QSL can be defined compositionally for most common logical connectives. It then suffices to apply the Iverson bracket to atomic separation logic formulas, such as **emp**, $E \mapsto E'$, $x = y$, etc. For every logical connective, we then use its quantitative analog in QSL. This suggests that we could also construct QSL as a conservative extension of separation logic if we take the intensional approach and fix a concrete syntax. An explicit embedding of separation logic formulas into QSL is given by the function

$$\texttt{qsl}\colon \textbf{Pred} \to \mathbb{E}_{\leq 1}$$

mapping predicates to one-bounded expectations as defined in Table 6.1. Assuming that the syntax of separation logic predicates is—apart from atomic formulas and derived connectives—covered by Table 6.1, we obtain the following:

Theorem 6.33 (Conservativity of QSL as an Assertion Language [1]) For every predicate P and every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, we have

(a) $\texttt{qsl}[\![P]\!](\mathfrak{s},\mathfrak{h}) \in \{0,1\}$, and

(b) $\mathfrak{s},\mathfrak{h} \models P \quad$ iff $\quad \texttt{qsl}[\![P]\!](\mathfrak{s},\mathfrak{h}) \;=\; 1.$

Proof. For (atomic) predicates, the claim is immediate by definition of Iverson brackets. The cases for separating conjunction and separating implication have already been considered in Theorem 6.9 and Theorem 6.13. The remaining cases are proven similarly. □

Conservativity for the expectation domain $(\mathbb{E}, \preceq)$ is achieved analogously except that the embedding of separating implication is defined as

$$\mathtt{qsl}[\![Q \mathrel{-\!\!*} R]\!] \triangleq \min\{\,1, \mathtt{qsl}[\![Q]\!] \mathrel{-\!\!*} \mathtt{qsl}[\![R]\!]\,\}.$$

6.6 Recursive Expectation Definitions

As discussed in Section 4.3.4, classical separation logic relies on recursive predicate definitions to describe unbounded data structures, such as lists or trees. The same mechanism is supported in QSL: Quantitative properties of unbounded data structures are specified by recursive equations of the form

$$\mathbf{X}(\vec{v}) \;=\; \Phi(\mathbf{X})(\vec{v}),$$

where $\vec{v} \in \mathbb{Z}^n$ is a sequence of $n \in \mathbb{N}$ values passed to the parameterized expectation $\mathrm{X}\colon \mathbb{Z}^n \to \mathbb{E}$. Moreover, Φ is a monotone transformer of the form

$$\Phi\colon (\mathbb{Z}^n \to \mathbb{E}) \to (\mathbb{Z}^n \to \mathbb{E}).$$

The expectation $\mathbf{X}(E_1, \ldots, E_n)$ is then given by the least fixed point of Φ, i.e.,

$$\mathbf{X}(E_1, \ldots, E_n)(\mathfrak{s}, \mathfrak{h}) \;=\; \mathsf{lfp}\,(\Phi)\,(E_1\,(\mathfrak{s}), \ldots, E_n\,(\mathfrak{s}))(\mathfrak{s}, \mathfrak{h}).$$

Since Φ is monotone and the set of all functions $\mathbf{X}\colon \mathbb{Z}^n \to \mathbb{E}$ form a complete lattice with respect to our ordering on expectations $\preceq$ (applied pointwise), the existence of the least fixed point is guaranteed by the Knaster-Tarski fixed point theorem (Theorem A.11). In fact, all recursively defined expectations in this thesis are continuous. Hence, by Kleene's fixed point theorem (Theorem A.16) the least fixed point is given by the limit of all finite unfoldings of Φ.

Example 6.34 Every recursively defined separation logic predicate can be interpreted as a recursively defined expectation in QSL by applying the embedding introduced in Section 6.5. For instance, recall from Example 4.26 the definition of a recursive predicate specifying singly-linked list segments with head u and tail u:

$$\boldsymbol{sll}(u, v) \;\triangleq\; (\; u = v \wedge \mathbf{emp}\;) \quad \vee \quad (\; \exists u' \colon u \mapsto u' \star \boldsymbol{sll}(u', v)\;)$$

The corresponding expectation in QSL is provided below, where we applied Lemma 6.5 to replace the maximum corresponding to $\vee$ by a sum:

$$[\textbf{\textit{sll}}\,(u,v)] \triangleq \underbrace{[u=v]\cdot[\textbf{emp}] \quad + \quad [u\neq v]\cdot \sup_{u'\in\mathbb{Z}} \left[u\mapsto u'\right]\star\left[\textbf{\textit{sll}}\,\left(u',v\right)\right]}_{=\,\Phi([\textbf{\textit{sll}}(.,.)])(u,v)}.$$

The predicate $[\textbf{\textit{sll}}\,(E,E')]\,(\mathfrak{s},\mathfrak{h})$ evaluates to one if and only if the heap $\mathfrak{h}$ consists exactly of a singly-linked list with head $E\,(\mathfrak{s})$ and tail $E'\,(\mathfrak{s})$.

For quantitative reasoning, recursively defined predicates are particularly useful in combination with the expectation **size**: To quantify, for example, the length of a singly-linked list segment, we first state that the heap is a singly-linked list segment and then measure the size of the heap. Hence, the length of a singly-linked list segment from E to E' is given by $[\textbf{\textit{sll}}\,(E,E')]\cdot\textbf{size}$. Furthermore, to measure the length of such a list segment contained in some larger heap, we transform this expectation into an intuitionistic one. That is, by Theorem 6.29, it suffices to consider $1\star([\textbf{\textit{sll}}\,(E,E')]\cdot\textbf{size})$.

Recursive definitions in QSL are, however, not limited to predicates. For instance, the length of a singly-linked list segment can be specified without **size**:

$$\textbf{\textit{len}}\,(u,v) \triangleq [u\neq v]\cdot\sup_{w\in\mathbb{Z}}[u\mapsto w]\star([\textbf{\textit{sll}}\,(w,v)]+\textbf{\textit{len}}\,(w,v))$$

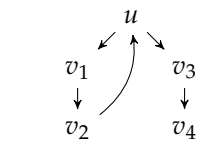
If the heap exclusively consists of a singly-linked list from u to v, then the expectation $\textbf{\textit{len}}\,(u,v)$ evaluates to the length of that list. Otherwise, it evaluates to zero. As the following lemma shows, our recursive definition of list lengths coincides with the previous definition based on the **size** quantity. Moreover, we show that singly-linked list segments can be split into multiple smaller ones. In particular, this means that the expectation $[\textbf{\textit{sll}}\,(E,E')]$ is *not* precise.

Lemma 6.35 (Properties of List Segments [1]) For the singly-linked list predicate $[\textbf{\textit{sll}}\,(u,v)]$ and the list length quantity $\textbf{\textit{len}}\,(u,v)$, we have:

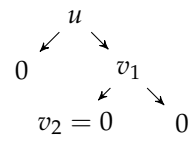
(a) $\textbf{\textit{len}}\,(E,E') = [\textbf{\textit{sll}}\,(E,E')]\cdot\textbf{size}$, and

(b) $[\textbf{\textit{sll}}\,(E,E')] = \sup_{u\in\mathbb{Z}}[\textbf{\textit{sll}}\,(E,u)]\star[\textbf{\textit{sll}}\,(u,E')]$.

The list-length quantity ***len*** actually serves two purposes: It ensures that the heap is a list and if so determines the longest path through the heap. The latter part can be generalized to other data structures. To this end, assume the heap is organized into fixed-size, successive blocks of memory representing *records*, for example the left and right pointer of a binary tree. If the size of records is



$[\textbf{\textit{tree}}\,(u)]\,(\mathfrak{s},\mathfrak{h}) = 0,$
$\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s},\mathfrak{h}) = 5$



$[\textbf{\textit{tree}}\,(u)]\,(\mathfrak{s},\mathfrak{h}) = 1,$
$\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s},\mathfrak{h}) = 2$

Figure 6.2: Evaluation of $[\textbf{\textit{tree}}\,(u)]$ and $\textbf{\textit{path}}[\![2]\!]\,(u)$ in two heaps $\mathfrak{h}$ depicted as directed graphs. An edge $\ell \to \ell'$ denotes either $\mathfrak{h}(\ell) = \ell'$ or $\mathfrak{h}(\ell+1) = \ell'$.

a constant $n \in \mathbb{N}$, then the longest path through these records starting in u is given by the following quantity:

$$\textbf{\textit{path}}[\![n]\!]\,(u) \;\triangleq\; \sup_{v\in\mathbb{Z}} \left(\max_{0\le k<n} [u+k \mapsto v] \right) \star (1 + \textbf{\textit{path}}[\![n]\!]\,(v))$$

Intuitively, $\textbf{\textit{path}}[\![n]\!]\,(u)$ always selects the successor address v among the possible pointers in the record belonging to u which is the source of the longest path through the remaining heap. Notice that no explicit base case is needed, because the length of empty paths is zero. Moreover, the use of the separating conjunction prevents selecting the same pointer twice. The quantity ***path*** is more liberal than ***len*** in the sense that heaps may contain pointers that do not lie on the specified path. Similar to **size**, we may refine the set of possible paths under consideration via additional data structure specifications.

Example 6.36 Let us apply the expectation ***path*** to quantify the height of a binary tree with root u. To this end, we first consider a recursive predicate specifying binary trees:

$$[\textbf{\textit{tree}}\,(u)] \;\triangleq\; [u = 0] \cdot [\mathbf{emp}] \;+\; \sup_{v,w\in\mathbb{Z}} [u \mapsto v, w] \star [\textbf{\textit{tree}}\,(v)] \star [\textbf{\textit{tree}}\,(w)]\,.$$

By convention, every node in a tree consists of two pointers: The first refers to the left child and the second refers to the right child, respectively. $[\textbf{\textit{tree}}\,(u)]$ is a precise expectation: It specifies the largest tree with root u in which every leaf is equipped with two pointers to 0. To measure the height of a tree, we then combine $[\textbf{\textit{tree}}\,(u)]$ with $\textbf{\textit{path}}[\![2]\!]\,(u)$ as follows:

$$\mathsf{treeHeight}(u) \;\triangleq\; [\textbf{\textit{tree}}\,(u)] \cdot \textbf{\textit{path}}[\![2]\!]\,(u)\,.$$

Two evaluations of $\mathsf{treeHeight}(u)$ in some stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ are illustrated in Figure 6.2: First, assume heap $\mathfrak{h}$ is given by the graph on the left-hand side. This graph does *not* constitute a binary tree, because it contains a cycle. Hence, $[\![\textbf{\textit{tree}}\,(u)]\!]\,(\mathfrak{s}, \mathfrak{h}) = 0$. The longest path through the graph is $uv_1v_2uv_3v_4$, i.e., $[\![\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s}, \mathfrak{h}) = 5$. Put together, this yields

$$\mathsf{treeHeight}(u)(\mathfrak{s}, \mathfrak{h}) \;=\; [\![\textbf{\textit{tree}}\,(u)]\!]\,(\mathfrak{s}, \mathfrak{h}) \cdot [\![\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s}, \mathfrak{h}) \;=\; 0 \cdot 5 \;=\; 0.$$

Second, assume heap $\mathfrak{h}$ is described by the graph on the right-hand side. This graph *is* a binary tree with root u, i.e., we have $[\![\textbf{\textit{tree}}\,(u)]\!]\,(\mathfrak{s}, \mathfrak{h}) = 1$. The longest path through this heap is of length two, e.g., uv_1v_2. Hence, $[\![\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s}, \mathfrak{h}) = 2$. Put together, this yields

$$\mathsf{treeHeight}(u)(\mathfrak{s}, \mathfrak{h}) \;=\; [\![\textbf{\textit{tree}}\,(u)]\!]\,(\mathfrak{s}, \mathfrak{h}) \cdot [\![\textbf{\textit{path}}[\![2]\!]\,(u)\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \cdot 2 \;=\; 2.$$

This chapter is based on prior publications, namely [1; 18], which are presented, discussed, and extended in the broader context of this thesis.

We now move from QSL as an assertion language to program verification. In Chapters 2 to 4, we considered program verification through the lens of proving that a Hoare triple is valid. This approach allows to conveniently address most common questions about the correctness of programs. For example:

- The triple $\langle\, x > 0 \,\rangle\ C\ \langle\, \mathsf{true} \,\rangle$ is valid for total correctness iff program C terminates on all inputs in which variable x is positive.
- The triple $\langle\, \mathsf{true} \,\rangle\ C\ \langle\, x = 1 \,\rangle$ is valid for total correctness iff program C terminates with variable x being equal to one.
- The triple $\langle\, \mathit{tree}(x) \,\rangle\ C\ \langle\, \mathbf{emp} \,\rangle$ is valid for total correctness iff the $\mathbf{P^3L}$ program C terminates without a memory fault with an empty heap whenever it is executed on a binary tree with root x.

Den Hartog [Har02; HV02] extended this approach to the verification of probabilistic programs by allowing predicates to refer to the probability that a predicate holds. For instance, his flavor of Hoare logic enables verifying that

$$\{\, x := 0 \,\}\ [\,1/2\,]\ \{\, x := 1 \,\}$$

models a fair coin flip. The precondition of a suitable Hoare triple is true. Moreover, the postcondition states that the probability of $x = 0$ being satisfied is $1/2$ and the probability of $x = 1$ being satisfied is $1/2$, respectively. This approach requires some knowledge of the probabilities that a predicate of interest holds.

When reasoning about probabilistic programs, however, such knowledge is typically not available a priori. Rather, most common questions are concerned with *quantifying* the behavior of a program. For example:

- What is the probability that $x = 1$ holds after execution of program C?

- What is the probability that program C terminates on all inputs?
- What is the expected value of x after execution of program C?

Answering any of the above questions amounts to a special case of the central task of *quantitative reasoning for probabilistic programs*:

> *Given a* $\mathbf{P^4L}$ *program* C *and an expectation* $\mathsf{X} \in \mathbb{E}$*, determine, for every initial stack-heap pair* $(\mathfrak{s}, \mathfrak{h})$*, the expected value of* X *after execution of* C *on* $(\mathfrak{s}, \mathfrak{h})$*.*

The desired expected value is given by the expected reward $\mathsf{ExpRew}[\mathsf{X}]\,(C, \mathfrak{s}, \mathfrak{h})$ introduced in Section 5.3.3 as part of the operational semantics of $\mathbf{P^4L}$ programs. It thus only accounts for both terminating and memory safe executions. Moreover, nondeterminism is resolved by a scheduler that attempts to minimize the overall expected value. In Section 5.3.3, we approached this task by applying the operational semantics of $\mathbf{P^4L}$: For any initial stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, we run program C on $(\mathfrak{s}, \mathfrak{h})$. This yields all stack-heap pairs $(\mathfrak{s}_1, \mathfrak{h}_1), (\mathfrak{s}_2, \mathfrak{h}_2), \ldots$ and probabilities $p_1, p_2, \ldots$ such that program C terminates *successfully* on $(\mathfrak{s}, \mathfrak{h})$ with probability p_k in final stack-heap pair $(\mathfrak{s}_k, \mathfrak{h}_k)$. After that, we determine the expected value of C with respect to expectation X and $(\mathfrak{s}, \mathfrak{h})$, i.e.,

$$\mathsf{ExpRew}[\mathsf{X}]\,(C, \mathfrak{s}, \mathfrak{h}) \;=\; \sum_{k=1}^{\infty} p_k \cdot \mathsf{X}(\mathfrak{s}_k, \mathfrak{h}_k).$$

How do we apply program verification techniques to perform quantitative reasoning compositionally on the structure of $\mathbf{P^4L}$ programs?

Let us first observe that the function mapping stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$ to the expected value of X after execution of C on $(\mathfrak{s}, \mathfrak{h})$, i.e., $\lambda(\mathfrak{s}, \mathfrak{h}).\, \mathsf{ExpRew}[\mathsf{X}]\,(C, \mathfrak{s}, \mathfrak{h})$, is itself an expectation. This leads to a more concise formulation of our task:

> *Given a* $\mathbf{P^4L}$ *program* C *and an expectation* $\mathsf{X} \in \mathbb{E}$*, determine the expectation* $\lambda(\mathfrak{s}, \mathfrak{h}).\, \mathsf{ExpRew}[\mathsf{X}]\,(C, \mathfrak{s}, \mathfrak{h}) \in \mathbb{E}$*.*

The above task suggests to reason on the level of expectations. Kozen [Koz83] was the first to realize close similarities between expected values and weakest preconditions: Expectations take the role of predicates. Moreover, expected values take the role of weakest preconditions. In fact, we show in Section 7.4 that both notions coincide for non-probabilistic programs C and predicates $\mathsf{X} = [Q]$. In the remainder of this section, we study how weakest precondition reasoning can be applied to compute expected values.

Before we proceed, let us briefly argue why we prefer weakest preconditions over Hoare triples when reasoning about expected values. For Hoare triples, there are essentially two approaches to prove their validity: We either apply forward reasoning, i.e., we start at the precondition and derive a predicate

implied by the strongest postcondition, or backward reasoning, i.e., we start at the postcondition and derive a predicate implying the weakest precondition. Unfortunately, as argued by Jones [Jon90, p. 135], the former approach is unsound for reasoning about expected values on the level of expectations. Hence, we are forced to apply backward reasoning. We thus avoid any notion of Hoare triples which might suggest otherwise. This also justifies our need for the separating implication which—in classical forward-directed approaches based on separation logic—is frequently omitted. Rather, one employs a forward transformer that overapproximates the strongest postcondition, i.e. completeness is sacrificed for simplicity.[1]

7.1 The Weakest Preexpectation Calculus

Towards a calculus for formal reasoning about expected values of $\mathbf{P^4L}$ programs, our goal is to compute the expectation

$$\lambda(\mathfrak{s},\mathfrak{h}).\,\mathsf{ExpRew}[\mathrm{X}]\,(C,\mathfrak{s},\mathfrak{h})\,,$$

where $\mathsf{ExpRew}[\mathrm{X}]\,(C,\mathfrak{s},\mathfrak{h})$ denotes the (least) expected value of expectation $\mathrm{X} \in \mathbb{E}$ after successful termination of $\mathbf{P^4L}$ program C on $(\mathfrak{s},\mathfrak{h})$ (cf. Definition 5.9). To this end, we employ a continuation-passing style *expectation transformer*

$$\mathsf{wp}\,[C] : \mathbb{E} \to \mathbb{E}$$

that determines the above expectation through (mostly syntactic) manipulations of the provided postexpectation X. Due to similarities between $\mathsf{wp}\,[C]$ and predicate transformers used to compute weakest preconditions (cf. Chapters 2 and 4), McIver and Morgan [MM05] coined the term *weakest preexpectation transformer* Consequently, we refer to wp as the *weakest preexpectation calculus*.

Definition 7.1 (Weakest Preexpectation Calculus for $\mathbf{P^4L}$ [1]) The *weakest preexpectation calculus* wp is defined by structural induction on $\mathbf{P^4L}$ programs according to the rules in Figure 7.1, page 200.

The weakest preexpectation of most $\mathbf{P^4L}$ statements is defined analogously to the weakest precondition of $\mathbf{P^3L}$ statements as discussed in Section 4.4. However, we have to account for the fact that wp transforms *expectations* rather than predicates. Moreover, we added rules for probabilistic choice and probabilistic assignment. Let us thus briefly go over the rules for wp stated in Figure 7.1.

[1] Nonetheless, it is possible to define the exact strongest postcondition in classical separation logic using the so-called septraction, i.e., $\neg(P \mathrel{-\!\!*} \neg Q)$. Intuitively, the septraction states that there exists an extension of the heap satisfying P such that the entire heap satisfies Q.

C	$\mathsf{wp}\,[C]\,(\mathrm{X})$
`skip`	X
$x := E$	$\mathrm{X}\,[x/E]$
$x :\approx \mu$	$\lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}} \mu(\mathfrak{s})(v)\cdot \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h})$
$x := \texttt{alloc}(E_1,\ldots,E_n)$	$\inf_{v\in\mathbb{Z}} [v\mapsto E_1,\ldots,E_n] \mathrel{-\!\!\!\star} \mathrm{X}\,[x/v]$
$\texttt{free}(E)$	$[E\mapsto -]\star\mathrm{X}$
$x := <E>$	$\sup_{v\in\mathbb{Z}} [E\mapsto v]\star([E\mapsto v]\mathrel{-\!\!\!\star}\mathrm{X}\,[x/v])$
$<E> := E'$	$[E\mapsto -]\star([E\mapsto E']\mathrel{-\!\!\!\star}\mathrm{X})$
$C_1\,;\,C_2$	$\mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(\mathrm{X}))$
$\texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}$	$[B]\cdot\mathsf{wp}\,[C_1]\,(\mathrm{X}) \;+\; [\neg B]\cdot\mathsf{wp}\,[C_2]\,(\mathrm{X})$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p\cdot\mathsf{wp}\,[C_1]\,(\mathrm{X}) \;+\; (1-p)\cdot\mathsf{wp}\,[C_2]\,(\mathrm{X})$
$\texttt{while}\,(\,B\,)\,\{\,C'\,\}$	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda\mathrm{I}.\ [B]\cdot\mathsf{wp}\,[C']\,(\mathrm{I}) \;+\; [\neg B]\cdot\mathrm{X}$
$x := F(E_1,\ldots,E_n)$	$\mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1,\ldots,E_n)(\mathrm{X})$, where
$\mathfrak{P}_F \triangleq \lambda\theta\lambda x'\lambda(E_1',\ldots,E_n')\lambda\mathrm{X}'.$	
$\mathsf{wp}^F_\theta\,[\mathsf{body}\,(F)]\,(\mathrm{X}'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}])\,[x_1/E_1'\,[-\boxdot]]\ldots[x_n/E_n'\,[-\boxdot]]\,[+\boxdot]$	

Figure 7.1: Rules of the weakest preexpectation calculus for **P⁴L** programs. Here, $\mu(\mathfrak{s})(v)$ denotes the probability that distribution expression μ evaluates to integer value v for stack $\mathfrak{s}$. Apart from procedure calls, the auxiliary weakest preexpectation calculus wp^F_θ is defined analogously. A formal definition is found in Figure 7.5, page 228.

Since `skip` modifies neither the stack nor the heap, the expected value of expectation X remains unchanged. Hence, $\mathsf{wp}\,[\texttt{skip}]$ is the identity.

For the assignment, $\mathsf{wp}\,[x := E]\,(\mathrm{X})$ yields $\mathrm{X}\,[x/E]$, i.e., we syntactically substitute variable x by expression E in expectation X (cf. Section 6.5).

The rule for sequential composition is the same as for weakest preconditions: $\mathsf{wp}\,[C_1\,;\,C_2]\,(\mathrm{X})$ obtains a preexpectation of the program $C_1\,;\,C_2$ by applying $\mathsf{wp}\,[C_1]$ to the intermediate expectation obtained from $\mathsf{wp}\,[C_2]\,(\mathrm{X})$.

For the conditional choice, there are two approaches: First, analogously to classical weakest preconditions, $\mathsf{wp}\,[\texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}]\,(\mathrm{X})$ selects either $\mathsf{wp}\,[C_1]\,(\mathrm{X})$ if condition B evaluates to true or $\mathsf{wp}\,[C_2]\,(\mathrm{X})$ if condition B evaluates to false (cf. Section 2.3.4). Since—at least as long as predicates are involved—we interpret standard conjunction as pointwise multiplication and disjunction as pointwise maximum (cf. Section 6.2), a translation of the weakest precondition of conditional choice into the realm of expectations yields

$$\max\left\{\,[B]\cdot\mathsf{wp}\,[C_1]\,(\mathrm{X})\,,\,[\neg B]\cdot\mathsf{wp}\,[C_2]\,(\mathrm{X})\right\}.$$

Second, we may interpret condition B as a degenerate distribution expression that assigns all probability mass either to $\mathsf{wp}\,[C_1]\,(\mathrm{X})$ or to $\mathsf{wp}\,[C_2]\,(\mathrm{X})$. Hence, the expected value of X after execution of a conditional choice is given by the convex sum of the expectations $\mathsf{wp}\,[C_1]\,(\mathrm{X})$ and $\mathsf{wp}\,[C_2]\,(\mathrm{X})$ weighted by their probabilities, i.e., $[B]$ and $1-[B]=[\neg B]$. This yields the following rule:

$$\mathsf{wp}\,[\texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}]\,(\mathrm{X})\ \triangleq\ [B]\cdot\mathsf{wp}\,[C_1]\,(\mathrm{X})\ +\ [\neg B]\cdot\mathsf{wp}\,[C_2]\,(\mathrm{X})\,.$$

In fact, both interpretations are equivalent due to Lemma 6.5. We prefer the latter one as it highlights that we intend to compute expected values.

The expected value of expectation X after execution of a probabilistic choice $\{\,C_1\,\}\ [\,p\,]\ \{\,C_2\,\}$ is given by a convex sum that weighs $\mathsf{wp}\,[C_1]\,(\mathrm{X})$ and $\mathsf{wp}\,[C_2]\,(\mathrm{X})$ by probabilities p and $(1-p)$, respectively.

For the probabilistic assignment, the weakest preexpectation $\mathsf{wp}\,[x :\approx \mu]\,(\mathrm{X})$ is the expected value of X with respect to distribution expression μ. Since μ depends on the stack, let us fix a stack-heap pair $(\mathfrak{s},\mathfrak{h})$. We then compute the convex sum over all possible assignments of integer v to variable x weighted by the probability $\mu(\mathfrak{s})(v)$ of sampling v, i.e.,

$$\begin{aligned}\mathsf{wp}\,[x :\approx \mu]\,(\mathrm{X}) &\triangleq \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}}\mu(\mathfrak{s})(v)\cdot\mathsf{wp}\,[x:=v]\,(\mathrm{X})\\ &= \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}}\mu(\mathfrak{s})(v)\cdot\mathrm{X}\,[x/v]\,.\end{aligned}$$

For loops, $\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C'\,\}]\,(\mathrm{X})$ is characterized as the least fixed point of loop unrollings (cf. Sections 2.3.4 and 4.4). We take a closer look at weakest preexpectations of loops in Section 7.2.1.

Further details regarding weakest preexpectations of the above statements are found in the textbook of McIver and Morgan [MM05].

Our treatment of procedure calls is completely analogous to weakest preconditions (cf. Section 3.2.2). That is, $\mathsf{wp}\,[x:=F(E_1,\ldots,E_n)]\,(\mathrm{X})$ is characterized as the least higher-order fixed point of executing the procedure body after taking scoping, parameter assignment, and the return value into account. To deal with

recursion, we employ a monotone auxiliary transformer wp_θ^F that resolves calls of procedure F by applying the (monotone) expectation transformer

$$\theta \colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E} \to \mathbb{E}).$$

Formally, the higher-order expectation transformer wp_θ^F is given by the rules in Figure 7.5 at the end of this chapter. We discuss reasoning about weakest preexpectations of procedure calls in Section 7.2.2.

It remains to consider the heap manipulating statements, i.e., allocation, deallocation, lookup, and mutation. The rules for computing their weakest preexpectations are adaptions of the backward reasoning rules for classical separation logic (cf. Section 4.4). However, we use the quantitative separating connectives of QSL instead of standard separating conjunction and implication. Furthermore, every separation logic predicate, e.g., the points-to predicate $E \mapsto E'$, is replaced by its Iverson bracket, e.g., $[E \mapsto E']$.

More precisely, for the allocation, $\mathsf{wp}\,[x := \texttt{alloc}(E_1, \ldots, E_n)]\,(\mathrm{X})$ extends the heap by a consecutive block of n addresses with first address v before measuring the updated expectation $\mathrm{X}\,[x/v]$ in the extended heap. As in the classical case, we assume that address v is chosen nondeterministically. Throughout this thesis, we resolve nondeterminism by a "demonic" scheduler that attempts to minimize the expected value. Hence, we take the infimum over all possible integers $v \in \mathbb{Z}$. Notice that any value $v \leq 0$ leads to the case

$$[v \mapsto E_1, \ldots, E_n] \mathrel{-\!\!*} \mathrm{X}\,[x/v] \;=\; 0 \mathrel{-\!\!*} \mathrm{X}\,[x/v] \;=\; \infty.$$

Since we assume an infinite amount of available addresses, it is thus guaranteed—unless X already equals infinity—that a valid address $v > 0$ is chosen.

We remark that our resolution of nondeterminism is a design choice that is compatible with classical separation logic, where all addresses are considered for allocation. However, other choices are possible. For example, one might attempt to maximize the expected value or fix a possibly randomized scheduler governing memory allocation. To ensure that memory allocation never fails, one might have to restrict v to addresses that have not already been allocated. Such a restriction is not required for our demonic scheduler:

Theorem 7.2 (Safe Demonic Allocation) Let $\mathrm{addr} \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathbb{N}_{>0} \setminus \mathsf{dom}(\mathfrak{h})$ be the set of all addresses that are not allocated. Then, for all $\mathrm{X} \in \mathbb{E}$:

(a) $\inf_{v \in \mathbb{Z}} [v \mapsto E] \mathrel{-\!\!*} \mathrm{X} \;=\; \inf_{v \in \mathrm{addr}} [v \mapsto E] \mathrel{-\!\!*} \mathrm{X}$, and

(b) $\inf_{v \in \mathrm{addr}} [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot \mathrm{X} \;=\; \inf_{v \in \mathrm{addr}} \mathrm{X}.$

For the memory deallocation, $\mathsf{wp}\,[\texttt{free}(E)]\,(\mathrm{X})$ measures the expectation X in a heap in which address E has been disposed. To this end, we first separate the

memory cell captured by $[E \mapsto -]$ from the rest of the heap and then measure X in that rest. In particular, if $[E \mapsto -]$ evaluates to zero, then the expected value of X after *successful* termination of $\texttt{free}(E)$ is zero as well. This reflects the fact that $\texttt{free}(E)$ crashed by attempting to dispose an address that is not allocated.

For the lookup, $\mathsf{wp}\,[x := <E>]\,(\mathrm{X})$ measures the expectation X after substituting variable x by the value at address E. If address E is not allocated, the program crashes and we measure zero. We use the intuitionistic predicate $[E \hookrightarrow v]$ to express that E points to v in a possibly larger heap:

$$\begin{aligned}\mathsf{wp}\,[x := <E>]\,(\mathrm{X}) \;&\triangleq\; \sup_{v\in\mathbb{Z}} [E \hookrightarrow v] \cdot \mathrm{X}\,[x/v] \\ &=\; [\![\ \text{Theorem 6.30 (a)}\ ]\!] \\ &\quad\; \sup_{v\in\mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \mathrel{-\!\!\star} \mathrm{X}\,[x/v])\,.\end{aligned}$$

We prefer the second version of the above rule because it precisely captures the memory cells touched by a lookup statement. It is noteworthy that the value v at address E is *selected* rather than maximized by the supremum $\sup_{v\in\mathbb{Z}}$, because either address E is not allocated, i.e., $[E \hookrightarrow v]$ evaluates to zero for every choice of v, or there is a unique value v at address E which is chosen. The supremum thus takes the role of an existential quantifier.

Finally, for the mutation, $\mathsf{wp}\,[<E> := E']\,(\mathrm{X})$ measures expectation X after updating the value at address E to E'. To this end, we first carve out the original pointer $[E \mapsto -]$ using a separating conjunction. This also guarantees that we measure zero if address E is not allocated. After that, we extend the address by the updated pointer $[E \mapsto E']$ using a separating implication and measure X. Hence, the weakest preexpectation of mutations is defined as

$$\mathsf{wp}\,\big[<E> := E'\big]\,(\mathrm{X}) \;\triangleq\; [E \mapsto -] \star \big([E \mapsto E'] \mathrel{-\!\!\star} \mathrm{X}\big)\,.$$

Example 7.3 Let us consider a few examples that highlight the features of weakest preexpectations with QSL.

The **P⁴L** program $\{\,\texttt{skip}\,\}\ [1/3]\ \{\,\texttt{free}(x)\,;\,\texttt{free}(x)\,\}$ flips a biased coin and either immediately terminates or encounters a memory fault because it attempts to dispose the same memory cell twice. Intuitively, this program terminates successfully with probability $1/3$. To confirm this hypothesis, we compute the weakest preexpectation with respect to postexpectation 1:

$$\begin{aligned}&\mathsf{wp}\,[\{\,\texttt{skip}\,\}\ [1/3]\ \{\,\texttt{free}(x)\,;\,\texttt{free}(x)\,\}]\,(1) \\ &= 1/3 \cdot 1 + 2/3 \cdot \mathsf{wp}\,[\texttt{free}(x)\,;\,\texttt{free}(x)]\,(1) \\ &= 1/3 + 2/3 \cdot (\underbrace{[x \mapsto -] \star [x \mapsto -]}_{=\,0} \star 1) \;=\; 1/3.\end{aligned}$$

In contrast to classical separation logic, where memory safety of all executions is built into the definition of Hoare triples (cf. Definition 4.12), *QSL thus enables us to quantify "how memory safe" a program is*.

Similarly, we can reason about the *probability of leaking memory*: To determine the probability of successfully terminating with an empty heap, it suffices to choose $[\mathbf{emp}]$ as a postexpectation. For instance, for the program $\{\, x := \texttt{alloc}(0) \,\}\ [1/2]\ \{\, \texttt{free}(x) \,\}$, we have:

$$\begin{aligned}
&\mathsf{wp}\,[\{\, x := \texttt{alloc}(0) \,\}\ [1/2]\ \{\, \texttt{free}(x) \,\}]\,([\mathbf{emp}]) \\
&= 1/2 \cdot \underbrace{\inf_{v\in\mathbb{Z}} [x \mapsto v] \mathrel{-\!\!*} [\mathbf{emp}]}_{=\,0 \text{ using Theorem 6.30 (a)}} + 1/2 \cdot \underbrace{([x \mapsto -] \star [\mathbf{emp}])}_{=\,[x\mapsto -] \text{ by Theorem 6.14}} \\
&= 1/2 \cdot [x \mapsto -]\,.
\end{aligned}$$

Hence, the probability that the above program terminates successfully without leaking memory is $1/2$ if exactly the memory cell at address x is initially allocated. Otherwise, it is zero. By choosing the postexpectation **size**, it also possible to determine *how much memory is leaked*:

$$\begin{aligned}
&\mathsf{wp}\,[\{\, x := \texttt{alloc}(0) \,\}\ [1/2]\ \{\, \texttt{free}(x) \,\}]\,(\mathbf{size}) \\
&= 1/2 \cdot \underbrace{\inf_{v\in\mathbb{Z}} [x \mapsto v] \mathrel{-\!\!*} \mathbf{size}}_{=\,1+\mathbf{size} \text{ using Theorem 6.31 (b)}} + 1/2 \cdot \underbrace{([x \mapsto -] \star \mathbf{size})}_{=\,[x\hookrightarrow -]\cdot(\mathbf{size}-1) \text{ by Theorem 6.31 (a)}} \\
&= 1/2 \cdot (1 + \mathbf{size}) + 1/2 \cdot \underbrace{[x \hookrightarrow -]}_{\preceq\, 1} \cdot (\mathbf{size} - 1) \ \preceq\ \mathbf{size}.
\end{aligned}$$

In other words, the above program leaks—in expectation—at most all memory cells that have been initially allocated. Finally, consider the **P⁴L** program

$$C\colon \qquad x := \texttt{alloc}(0)\,;\, y :\approx \mathsf{uniform}\,(1, x)\ ;\, \texttt{free}(x).$$

This program first allocates an address and stores it in variable x. After that, it samples an integer between one and x and stores the result in variable y. To compute the expected value of variable y, we proceed as illustrated in Figure 7.2 (read from bottom to top as in Example 2.29). Hence, the expected value of variable y after successful termination of C is the mean value of all integers between 1 and the first address v that is not already allocated. This is caused by our choice of a "demonic" scheduler that minimizes expectations. Other resolutions of the nondeterministic choice of address v lead to dramatically different results. A scheduler that maximizes

$$
\begin{array}{l}
\text{// } \quad \inf_{v \in \mathbb{Z}} [v \mapsto 0] \mathrel{-\!\!\star} \left([v \mapsto -] \star \frac{v+1}{2} \right) \\
x := \texttt{alloc}(0)\,; \\
\text{// } \quad [x \mapsto -] \star \frac{x+1}{2} \\
\text{// } \quad \text{algebra} \\
\text{// } \quad [x \mapsto -] \star \sum_{k=0}^{x-1} \frac{1}{x} \cdot \langle 1+k \rangle \\
y :\approx \mathsf{uniform}\,(1, x)\,; \\
\text{// } \quad [x \mapsto -] \star y \\
\texttt{free}(x) \\
\text{// } \quad y
\end{array}
$$

Figure 7.2: Computation of $\mathsf{wp}\,[C]\,(y)$.

expectations, for instance, yields in an infinite expected value—even if it is forbidden to choose already allocated addresses. It is thus important to evaluate whether the chosen interpretation of nondeterminism is sensible for the application at hand.

7.2 Proof Rules

Since weakest preexpectations—just like weakest preconditions—have to account for all possible program executions, they quickly tend to yield large expressions that are hardly human-readable. Moreover, in the presence of loops and recursion, they are often infeasible to compute. In previous chapters, we often settled for proving that a conceptually simpler precondition, say P, is covered by the weakest precondition of a program, say $\mathsf{wp}\,[C]\,(Q)$. That is, we proved that P implies $\mathsf{wp}\,[C]\,(Q)$. Here, the implication serves as the ordering of the complete lattice $\langle\, \mathbf{Pred}, \Rightarrow \,\rangle$ underlying our notion of weakest preconditions. Whenever we discharged that $P \Rightarrow \mathsf{wp}\,[C]\,(Q)$ holds, we thus actually showed that P is a *lower bound* of $\mathsf{wp}\,[C]\,(Q)$ in the complete lattice of predicates.

In the realm of expectations, we consider the complete lattices $\langle\, \mathbb{E}, \preceq \,\rangle$ and $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$, where $\preceq$ is the canonical ordering on real numbers applied pointwise. Applying the same principle as for weakest preconditions, it is reasonable to prove that an expectation is a lower bound (with respect to $\preceq$) of the weak-

est preexpectation of a program. This allows, for example, to test whether a program's probability of termination is at least $1/2$. In contrast to weakest preconditions, however, *upper bounds* on weakest preexpectations are of interest as well: We have, for instance, computed an upper bound on the expected number of memory cells leaked by a probabilistic pointer program in Example 7.3.

In this section, we collect proof rules for computing both upper and lower bounds on weakest preexpectations. These rules are applied afterward in the case studies presented in Chapter 8. Let us begin with a few basic properties:

Theorem 7.4 (Basic Properties of wp [1]) For all $\mathbf{P^4L}$-programs C, expectations $X, Y \in \mathbb{E}$, predicates P and constants $k \in \mathbb{R}_{\geq 0}$, we have:

(a) Monotonicity: $X \preceq Y$ implies $\mathsf{wp}\,[C]\,(X) \preceq \mathsf{wp}\,[C]\,(Y)$.

(b) Super–linearity: $\mathsf{wp}\,[C]\,(k \cdot X + Y) \succeq k \cdot \mathsf{wp}\,[C]\,(X) + \mathsf{wp}\,[C]\,(Y)$.

(c) Strictness: $\mathsf{wp}\,[C]\,(0) = 0$.

(d) 1-Boundedness of Predicates: $\mathsf{wp}\,[C]\,([P]) \preceq 1$.

Additionally, if C contains no allocation statements $x := \texttt{alloc}(\vec{E})$, we have:

(e) ω-continuity: For every increasing ω-chain $X_1 \preceq X_2 \preceq \ldots$ in $\mathbb{E}$,

$$\sup_n \mathsf{wp}\,[C]\,(X_n) = \mathsf{wp}\,[C]\,(\sup_n X_n)\,.$$

(f) Linearity: $\mathsf{wp}\,[C]\,(k \cdot X + Y) = k \cdot \mathsf{wp}\,[C]\,(X) + \mathsf{wp}\,[C]\,(Y)$.

Proof. Monotonicity, (super-)linearity, and ω-continuity are shown by induction on the program structure; see [1] for details. Since the constant function 0 is the least element of the complete lattice $(\mathbb{E}, \preceq)$, strictness follows directly from super-linearity:

$$\mathsf{wp}\,[C]\,(0) = \mathsf{wp}\,[C]\,(0 \cdot 0) \preceq 0 \cdot \mathsf{wp}\,[c]\,(0) = 0.$$

For 1-Boundedness, a straightforward induction on the program structure yields that $\mathsf{wp}\,[C]\,(1) \preceq 1$. By monotonicity of wp and the fact $[P] \leq 1$, we then conclude that $\mathsf{wp}\,[C]\,([P]) \preceq 1$. □

Monotonicity is a fundamental property that ensures well-definedness of the rules for weakest preexpectations of loops and recursive procedure calls. Moreover, (super-)linearity allows considering sums of expectations independently. Similarly to Theorem 4.32, it thus facilitates compositional reasoning. For weakest preexpectations, the constant function 0 indicates a failure, e.g., a predicate

is violated, nontermination, or a memory fault. Strictness then guarantees that we cannot miraculously recover from the occurrence of a failure. We have occasionally stated that choosing a predicate as a postexpectation allows us to reason about the probability of that predicate being satisfied after program execution. Due to 1-Boundedness, the weakest preexpectation of a predicate is indeed a probability, i.e., a real number in the interval $[0, 1]$. As discussed in Lemma 4.31 for classical separation logic, wp is not continuous in general. However, continuity is restored for programs that do not allocate memory.

The same properties also hold for the higher-order expectation transformer $\mathsf{wp}_\theta^F[C]$ as long as the expectation transformer θ is itself monotone (or continuous when showing ω-continuity). The proofs are completely analogous.

7.2.1 Proof Rules for Loops

We now turn to reasoning about weakest preexpectations of loops. Similarly to weakest preconditions (cf. Section 2.3.4), the weakest preexpectation of loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to postexpectation $\mathrm{X} \in \mathbb{E}$ is defined as the least fixed point of its *characteristic function*

$$\mathfrak{W} \;\triangleq\; \lambda \mathrm{I}.\; [B] \cdot \mathsf{wp}\,[C]\,(\mathrm{I}) \;+\; [\neg B] \cdot \mathrm{X}.$$

Due to monotonicity of wp (by Theorem 7.4 (a)), addition, and multiplication, the characteristic function $\mathfrak{W}$ is monotone as well. Furthermore, $\langle\, \mathbb{E}, \preceq \,\rangle$ is a complete lattice as shown in Lemma 6.3. In addition to guaranteeing the existence of the least fixed point of $\mathfrak{W}$, a constructive version of Tarski's and Knaster's fixed point theorem (cf. Theorem A.11) then allows us to iteratively compute the least fixed point of characteristic function $\mathfrak{W}$:

Theorem 7.5 ([1]) For every loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and expectation $\mathrm{X} \in \mathbb{E}$ with characteristic function $\mathfrak{W}$, there exists an ordinal α such that

$$\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;=\; \mathsf{lfp}\,(\mathfrak{W}) \;=\; \mathfrak{W}^\alpha(0).$$

Unfortunately, since the weakest preexpectation calculus is not continuous in general (see Lemma 4.31 for a counterexample), the fixed point iteration is not guaranteed to converge in the limit. We might thus have to take multiple limits until finding a suitable ordinal α such that $\mathfrak{W}$ coincides with the least fixed point. Consequently, it is often infeasible to compute the exact weakest preexpectation of loops by fixed point iteration. If no memory is allocated within the body of a loop, however, the corresponding point iteration converges in the limit:

Theorem 7.6 Let C be a $\mathbf{P^4L}$ program that contains no allocation statements. Then, for every loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and expectation $\mathrm{X} \in \mathbb{E}$ with characteristic function $\mathfrak{W}$, we have

$$\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;=\; \mathsf{lfp}\,(\mathfrak{W}) \;=\; \lim_{n\to\infty} \mathfrak{W}^n(0).$$

Proof. Since C contains no allocation statements, the characteristic function $\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X})$ is continuous. The claim then follows immediately from Theorem 7.5 and Kleene's fixed point theorem (Theorem A.16). □

The above theorem suggests a scheme for reasoning about the weakest preexpectations of the loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to expectation $\mathrm{X} \in \mathbb{E}$: Let $\bowtie \in \{\preceq, \succeq, =\}$ be a relation indicating whether we want to reason about upper bounds ($\preceq$), lower bounds ($\succeq$) or exact weakest preexpectations ($=$). Moreover, let $\mathrm{I} \in \mathbb{E}$ be an expectation. To show that

$$\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;\bowtie\; \mathrm{I}$$

holds, it suffices to find a family of expectations $(\mathrm{I}_n)_{n\in\mathbb{N}}$ such that

- $\mathrm{I}_0 \bowtie 0$,
- $\mathrm{I}_{n+1} \bowtie \mathfrak{W}(\mathrm{I}_n)$ for all $n \in \mathbb{N}$, and
- $\mathrm{I} \bowtie \lim_{n\to\infty} \mathrm{I}_n$.

What about weakest preexpectations of general loops? As long as we reason about *upper bounds*, we are equipped with a convenient invariant-based rule:

Theorem 7.7 ([1]) Let $\mathrm{I} \in \mathbb{E}$ be an expectation. For every loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and expectation $\mathrm{X} \in \mathbb{E}$ with characteristic function $\mathfrak{W}$, we have

$$\mathfrak{W}(\mathrm{I}) \;\preceq\; \mathrm{I} \quad \text{implies} \quad \mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;\preceq\; \mathrm{I}.$$

In this case, we call I an (upper) *invariant* of $\mathfrak{W}$.

Proof. By Theorem A.11, $\mathsf{lfp}\,(\mathfrak{W})$ is the smallest pre-fixed point of $\mathfrak{W}$. It is thus the smallest expectation I satisfying $\mathfrak{W}(\mathrm{I}) \preceq \mathrm{I}$. Consequently, by Theorem 7.5, $\mathsf{wp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) = \mathsf{lfp}\,(\mathfrak{W}) \preceq \mathrm{I}$. □

It is noteworthy that a similar rule is available when reasoning about total correctness of pointer programs with weakest preconditions in classical separation logic (cf. Section 4.4). In contrast to reasoning about expected values, however,

proving upper rather than lower bounds on predicates, i.e., reasoning about necessary conditions instead of sufficient ones, is usually not of interest.

Frohn et al. [Fro+17], show that the converse direction of the above rule enables reasoning about termination of deterministic programs. For *quantitative* reasoning about lower bounds, however, the converse direction of our invariant-based rule is unsound. That is, even for deterministic programs, $\mathfrak{W}(\mathrm{I}) \succeq \mathrm{I}$, does *not* imply that I is a *lower bound* on the weakest preexpectation of loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ with respect to postexpectation X.

Consider, for instance, the loop $\texttt{while}\,(\,x > 0\,)\,\{\,\texttt{skip}\,\}$. For postexpectation $\mathrm{X} = 0$ and invariant $\mathrm{I} = [x > 0] \cdot \infty$, we have

$$\mathfrak{W}(\mathrm{I}) \;=\; [x > 0] \cdot \mathrm{I} + [x \leq 0] \cdot 0 \;=\; [x > 0] \cdot \infty \;=\; \mathrm{I}.$$

However, by Theorem 7.4 (c), we know that the exact weakest preexpectation is

$$\mathsf{wp}\,[\texttt{while}\,(\,x > 0\,)\,\{\,\texttt{skip}\,\}]\,(0) \;=\; 0 \;\not\succeq\; \mathrm{I} \;=\; [x > 0] \cdot \infty.$$

Hence, I is *not* a correct lower bound. The search for easy-to-apply proof rules to derive lower bounds on weakest preexpectations for total correctness is an interesting direction for future research.

A promising starting point is a rule recently proposed by Hark et al. [Har+19] which allows reasoning about lower bounds on weakest preexpectations of probabilistic programs as long as the program in question terminates almost-surely. Furthermore, McIver and Morgan [MM05] considered proof rules in the alternative domain $\langle\, \mathbb{E}_{\leq 1}, \preceq \,\rangle$. Finally, Chatterjee and Fu [CF17] presented a rule for deriving lower bounds on expected runtimes.

7.2.2 Proof Rules for Procedure Calls

Weakest preexpectation reasoning about recursive procedures is mostly analogous to classical weakest precondition reasoning. We thus only briefly highlight differences and present key theorems. An in-depth discussion of reasoning about recursive procedures is found in Chapter 3.

The rationale underlying weakest preexpectation reasoning about recursive procedures is similar to reasoning about loops. That is, least fixed points exist and can—in principle—be computed by iterative application of the procedure's characteristic function due to the monotonicity of wp and the Tarski-Knaster fixed point theorem (cf. Theorem A.11). Moreover, as long as all involved procedures do not allocate memory, the fixed point iteration converges in the limit. In contrast to reasoning about loops, however, the characteristic function of procedure calls does not transform expectations. Rather, it transforms monotone *expectation transformers* of the form

$$\theta\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E} \to \mathbb{E}).$$

Here, the first argument represents the variable storing the procedure's return value upon termination. The second argument consists of n arithmetic expressions which are passed as parameters to an expectation transformer capturing the procedure's effect. By slight abuse of notation, we denote the constant function $\lambda x \lambda \langle E_1, \dots, E_n \rangle \lambda \mathrm{X}.\, 0$ by 0.

Formally, the characteristic function $\mathfrak{P}_F$ of procedure call $x := F(E_1, \dots, E_n)$ with respect to transformer θ is defined as

$$\begin{aligned}\mathfrak{P}_F(\theta) \;\triangleq\; & \lambda x' \lambda(E'_1, \dots, E'_n) \lambda \mathrm{X}'. \\ & \mathsf{wp}^F_\theta\,[\mathsf{body}\,(F)]\,\big(\mathrm{X}'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,[x_1/E'_1\,[-\boxdot]]\dots[x_n/E'_n\,[-\boxdot]]\,[+\boxdot]\,.\end{aligned}$$

The higher-order transformer wp^F_θ is the same as wp except that calls of procedure F are resolved by applying function θ. That is,

$$\mathsf{wp}^F_\theta\,[x := F(E_1, \dots, E_n)]\,(\mathrm{X}) \;=\; \theta(x)(E_1, \dots, E_n)(\mathrm{X}).$$

The complete definition of wp^F_θ is found in Figure 7.5, page 228.

Intuitively, the characteristic function $\mathfrak{P}_F$ applies all steps involved in executing a procedure call in reverse order: We first leave the scope by transforming postexpectation X to $\mathrm{X}\,[-\boxdot]$. After that, we account for the return value represented by variable out; this yields the actual postexpectation $\mathrm{X}\,[-\boxdot][\boxdot x'/\mathsf{out}]$. We then compute the weakest preexpectation of the procedure body. Finally, we assign each parameter the value of the expression passed to procedure F and account for entering a new scope.

In summary, we obtain the following theorem for reasoning about weakest preexpectations of procedure calls:

Theorem 7.8 For every procedure call $x := F(E_1, \dots, E_n)$ with characteristic function $\mathfrak{P}_F$, there exists an ordinal α such that

$$\begin{aligned}\mathsf{wp}\,[x := F(E_1, \dots, E_n)] \;&= \mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \dots, E_n) \\ &= \mathfrak{P}^\alpha_F(0)(x)(E_1, \dots, E_n).\end{aligned}$$

Moreover, if no procedure body called by procedure F (including the body of F itself) contains an allocation statement, we have

$$\mathsf{wp}\,[x := F(E_1, \dots, E_n)] \;=\; \lim_{k\to\infty} \mathfrak{P}^k_F(0)(x)(E_1, \dots, E_n).$$

Proof. Clearly, the constant expectation transformer

$$0 \;\triangleq\; \lambda x \lambda \langle E_1, \dots, E_n \rangle \lambda \mathrm{X}.\, 0$$

is both monotone and continuous. Furthermore, we notice that the sets

- $\{\, \theta \mid \theta\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E} \to \mathbb{E}) \text{ monotone} \,\}$ and
- $\{\, \theta \mid \theta\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E} \to \mathbb{E}) \text{ continuous} \,\}$

of monotone and continuous expectation transformers form complete lattices with respect to the ordering $\preceq$ of expectations applied pointwise, i.e.,

$$\theta \preceq \rho \text{ iff } \forall x \forall E_1, \ldots, E_n \forall \mathrm{X}\colon \theta(x)(E_1, \ldots, E_n)(\mathrm{X}) \preceq \rho(x)(E_1, \ldots, E_n)(\mathrm{X}).$$

To prove the first claim, we notice that—for monotone θ—the characteristic function $\mathfrak{P}_F$ is monotone due to monotonicity of wp^F_θ (by Theorem 7.4 (a)), and the fact that scoping and substitution preserve monotonicity. Then:

$$\begin{aligned}
&\mathsf{wp}\,[x := F(E_1, \ldots, E_n)] \\
=\ &[\![\text{ Figure 7.1 }]\!] \\
&\mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \ldots, E_n) \\
=\ &[\![\text{ Theorem A.11 (Tarski-Knaster) for a suitable ordinal } \alpha \,]\!] \\
&\mathfrak{P}_F^\alpha(0)(x)(E_1, \ldots, E_n).
\end{aligned}$$

To prove the second claim, we notice that—for *continuous* θ—the characteristic function $\mathfrak{P}_F$ is continuous due to continuity of wp^F_θ (by Theorem 7.4 (e)), and the fact that scoping and substitution preserve continuity. Then:

$$\begin{aligned}
&\mathsf{wp}\,[x := F(E_1, \ldots, E_n)] \\
=\ &[\![\text{ Figure 7.1 }]\!] \\
&\mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \ldots, E_n) \\
=\ &[\![\text{ Theorem A.16 (Kleene) }]\!] \\
&\lim_{k \to \infty} \mathfrak{P}_F^k(0)(x)(E_1, \ldots, E_n). \qquad \square
\end{aligned}$$

Analogously to our proof rules for loops, we can also derive an invariant-based rule for reasoning about upper bounds of weakest preexpectations:

Theorem 7.9 For every procedure call $x := F(E_1, \ldots, E_n)$ with characteristic function $\mathfrak{P}_F$ and every monotone expectation transformer of the form $\rho\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E} \to \mathbb{E})$, we have

$$\mathfrak{P}_F(\rho) \preceq \rho \quad \text{implies} \quad \mathsf{wp}\,[x := F(E_1, \ldots, E_n)] \preceq \theta(x)(E_1, \ldots, E_n).$$

Proof. By the Tarski and Knaster fixed point theorem (cf. Theorem A.11), the least fixed point of $\mathfrak{P}_F$ is the smallest pre-fixed point of $\mathfrak{P}_F$. That is, θ is the smallest transformer satisfying $\mathfrak{P}_F(\theta) \preceq \theta$. Then, consider the following:

$$\begin{aligned}
& \mathsf{wp}\,[x := F(E_1, \ldots, E_n)] \\
= \;& [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1, \ldots, E_n) \\
= \;& [\![\,\mathsf{lfp}\,(\mathfrak{P}_F) \text{ is the as smallest pre-fixed point } \theta \text{ of } \mathfrak{P}_F \,]\!] \\
& \theta(x)(E_1, \ldots, E_n) \\
\preceq \;& [\![\,\rho \text{ is some pre-fixed point of } \mathfrak{P}_F \,]\!] \\
& \rho(x)(E_1, \ldots, E_n).
\end{aligned}$$

□

7.2.3 The Quantitative Frame Rule

As discussed in Sections 4.2 and 4.4.2, classical separation logic is designed to facilitate local reasoning about the heap. This is reflected by two facts:

First, every rule only refers to precisely those memory cells that are accessed by a program. The same holds for our weakest preexpectation calculus.

Second, the *frame rule* allows us to ignore memory cells that are captured by a postcondition but not modified by a program. This raises the question whether the frame rule remains valid for weakest preexpectations.

Recall from Theorem 4.33 the classical frame rule for separation logic formulated in terms of weakest preconditions: For all $\mathbf{P^3L}$ programs C and predicates Q, R with $\mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(R) = \emptyset$, we have

$$\mathsf{wp}\,[C]\,(Q) \star R \quad \Rightarrow \quad \mathsf{wp}\,[C]\,(Q \star R)\,.$$

Towards a frame rule for quantitative separation logic, let us first extend the set $\mathbf{Mod}\,(C)$ of variables updated by program C to $\mathbf{P^4L}$ programs:

$$\mathbf{Mod}\,(\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}) \triangleq \mathbf{Mod}\,(C_1) \cup \mathbf{Mod}\,(C_2) \text{ and } \mathbf{Mod}\,(x :\approx \mu) \triangleq \{\,x\,\}\,.$$

A complete definition of $\mathbf{Mod}\,(C)$ for $\mathbf{P^4L}$ programs is found in Figure 7.3, page 226. Now, in the realm of expectations, logical implications corresponds to the ordering $\preceq$. Then the frame rule for QSL reads as follows:

Theorem 7.10 (Quantitative Frame Rule [1]) For all $\mathbf{P^4L}$ programs C and expectations $X, Y \in \mathbb{E}$ with $\mathbf{Mod}(C) \cap \mathbf{Vars}(Y) = \emptyset$, we have

$$\mathsf{wp}[C](X) \star Y \preceq \mathsf{wp}[C](X \star Y).$$

Proof. By induction on the structure of $\mathbf{P^4L}$ programs. For loops and procedure calls, we apply Theorems 7.5 and 7.8 and then show by transfinite induction that the frame rule holds for any ordinal number of applications of the characteristic function. Details are found in Appendix C.3. □

The quantitative frame rule enables us to push parts of the postexpectation outside of the weakest preexpectation as long as we are reasoning about *lower bounds*. Does the frame rule also hold for *upper bounds*? Unfortunately, the answer is no for multiple reasons: The frame rule breaks in the quantitative case for probabilistic choice because separating conjunction and addition do not distribute in general (cf. Theorem 6.16). We can resolve this issue by restricting ourselves to precise expectations. However, the "converse frame rule" even breaks in the qualitative case for deterministic programs. Consider, for example, the weakest preexpectation of program $C \triangleq \langle x \rangle := 0$ with respect to $[\mathbf{emp}]$:

$$\mathsf{wp}[\langle x \rangle := 0]([\mathbf{emp}]) = [x \mapsto -] \star ([x \mapsto 0] \mathrel{-\!\!\star} [\mathbf{emp}]) = 0.$$

Moreover, for $Y = [x \hookrightarrow 0]$, we have $\mathbf{Mod}(C) \cap \mathbf{Vars}(Y) = \emptyset$. Then:

$$\begin{aligned}\mathsf{wp}[\langle x \rangle := 0]([\mathbf{emp}] \star [x \hookrightarrow 0]) &= [x \mapsto -] \star ([x \mapsto 0] \mathrel{-\!\!\star} ([\mathbf{emp}] \star [x \hookrightarrow 0])) \\ &= [x \hookrightarrow -].\end{aligned}$$

Put together, we obtain a counterexample to the converse frame rule:

$$\mathsf{wp}[\langle x \rangle := 0]([\mathbf{emp}]) \star [x \hookrightarrow 0] \not\preceq \mathsf{wp}[\langle x \rangle := 0]([\mathbf{emp}] \star [x \hookrightarrow 0]).$$

Similar counterexamples can be constructed using any heap manipulating statement except deallocation. Hence, the frame rule is not available when reasoning about upper bounds on weakest preexpectations.

As a silver lining, we rarely need the frame rule for weakest preexpectation reasoning because locality is already built into the rules of wp. In fact, the frame rule is mainly needed to discharge invariants in the sense of Theorem 7.9 for recursive procedures: If our proposed invariant provides us with the weakest preexpectation of a procedure call with respect to postexpectation X and—during its verification—we have to compute the weakest preexpectation of a call of the same procedure with respect to postexpectation $X \star Y$, then the frame rule would allow us to push Y out and apply our invariant.

In the above scenario, the lack of the frame rule can often be compensated by strengthening invariants: We propose a *template of invariants* that allows us to verify the original invariant *and* the frame rule for our particular case in parallel. That is, rather than proposing an invariant for a single postexpectation, say

$$\rho(x)(\vec{E})(\mathrm{X}) \;\preceq\; \mathrm{X}',$$

we propose a *template of invariants*, say

$$\rho(x)(\vec{E})(\mathrm{X} \star \mathrm{Y}) \;\preceq\; \mathrm{X}' \star \mathrm{Y},$$

for all expectations of the form $\mathrm{X} \star \mathrm{Y}$ such that $\mathbf{Mod}(C) \cap \mathbf{Vars}(\mathrm{Y}) = \emptyset$; we may also impose additional restrictions on Y, e.g., preciseness or purity. By Theorem 7.9, it then suffices to verify our proposed template for an arbitrary, but fixed, expectation Y that is compliant with our imposed restrictions. While discharging this proof obligation, we may apply our template to any procedure call with respect to any postexpectation of the form $\mathrm{X} \star \mathrm{Z}$ as long as Z satisfies the imposed restrictions. Hence, at the cost of a slightly more complex proof obligation, it is possible to reason about recursive procedures as if the frame rule was available. We take the previously outlined approach when reasoning about randomized meldable heaps in Section 8.5.

7.2.4 Other Proof Rules

To conclude our collection of proof rules, we discuss a few less prominent rules.

The Conjunction Rule First, let us revisit the conjunction rule for weakest preconditions (cf. Theorems 2.32, 3.19 and 4.32), i.e.,

$$\mathsf{wp}\,[C]\,(Q \wedge R) \;=\; \mathsf{wp}\,[C]\,(Q) \wedge \mathsf{wp}\,[C]\,(R)\,.$$

Unfortunately, the quantitative analog to the conjunction rule, i.e.,

$$\mathsf{wp}\,[C]\,(\mathrm{X} \cdot \mathrm{Y}) \;=\; \mathsf{wp}\,[C]\,(\mathrm{X}) \cdot \mathsf{wp}\,[C]\,(\mathrm{Y})\,,$$

does *not* hold in general. Towards a counterexample, consider the $\mathbf{P^4L}$ program

$$C\colon \quad \{\, x := 1 \,\}\ [\,1/3\,]\ \{\, \texttt{skip} \,\}\,.$$

Moreover, we choose the postexpectations $X = Y = |x|$. Then:

$$
\begin{aligned}
& \mathsf{wp}\,[C]\,(X \cdot Y) \\
= \;& \left[\!\!\left[\, X \cdot Y = |x|^2 \,\right]\!\!\right] \\
& \mathsf{wp}\,[C]\,\left(|x|^2\right) \\
= \;& [\![\, \text{Definition of } \mathsf{wp} \text{ (Figure 7.1)} \,]\!] \\
& 1/3 \cdot |1|^2 + 2/3 \cdot |x|^2 \\
= \;& [\![\, \text{elementary algebra} \,]\!] \\
& 1/3 + 2/3 \cdot |x|^2.
\end{aligned}
\qquad
\begin{aligned}
& \mathsf{wp}\,[C]\,(X) \cdot \mathsf{wp}\,[C]\,(Y) \\
= \;& [\![\, X = Y = |x| \,]\!] \\
& \mathsf{wp}\,[C]\,(|x|)^2 \\
= \;& [\![\, \text{Definition of } \mathsf{wp} \text{ (Figure 7.1)} \,]\!] \\
& (1/3 \cdot |1| + 2/3 \cdot |x|)^2 \\
= \;& [\![\, \text{elementary algebra} \,]\!] \\
& 1/9 + 4/9 \cdot |x| + 4/9 \cdot |x|^2.
\end{aligned}
$$

Hence, the quantitative conjunction rule is unsound for weakest preexpectations because $\mathsf{wp}\,[C]\,(X \cdot Y) \neq \mathsf{wp}\,[C]\,(X) \cdot \mathsf{wp}\,[C]\,(Y)$.

Weak Rules of Constancy As discussed in Section 4.2, one of the prime motivations for developing separation logic is to recover Reynolds' rule of constancy [Rey81] when reasoning about pointer programs, i.e.,

$$\mathsf{wp}\,[C]\,(Q) \wedge R \Rightarrow \mathsf{wp}\,[C]\,(Q \wedge R) \qquad \text{if } \mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(R) = \emptyset.$$

In separation logic, the frame rule, in which standard conjunction is replaced by separating conjunction, takes over the role of the rule of constancy. In fact, the rule of constancy can be derived from the frame rule as long as the provided postconditions are pure. The same holds in the quantitative case: Let C be a $\mathbf{P^4L}$ program. Moreover, let X and Y be pure expectations such that $\mathbf{Mod}\,(C) \cap \mathbf{Vars}\,(Y) = \emptyset$. We then obtain a quantitative rule of constancy:

$$
\begin{aligned}
& \mathsf{wp}\,[C]\,(X \cdot Y) \\
= \;& [\![\, \text{Theorem 6.22 (d)} \,]\!] \\
& \mathsf{wp}\,[C]\,(X \star Y) \\
\succeq \;& [\![\, \text{Theorem 7.10} \,]\!] \\
& \mathsf{wp}\,[C]\,(X) \star Y \\
\succeq \;& [\![\, \text{Theorem 6.22 (a)} \,]\!] \\
& \mathsf{wp}\,[C]\,(X) \cdot Y.
\end{aligned}
$$

In the realm of expectations, we are interested in approximating weakest preexpectations from both below and above. Although the quantitative frame rule holds for lower bounds only (cf. Theorem 7.10), the rule of constancy—for a pure postexpectation Y—is applicable in both directions.

Theorem 7.11 (Pure Rule of Constancy [18]) For all $\mathbf{P^4L}$ programs C and expectations X, Y such that Y is pure and $\mathbf{Mod}(C) \cap \mathbf{Vars}(Y) = \emptyset$, we have

$$\mathsf{wp}[C](X \cdot Y) \;=\; \mathsf{wp}[C](X) \cdot Y.$$

Proof. By induction on the structure of $\mathbf{P^4L}$ programs. □

Another version of the rule of constancy, which imposes no further restrictions on postexpectations, is obtained for programs that do not modify the heap.

Theorem 7.12 (Weak Rule of Constancy [18]) For every $\mathbf{P^4L}$ program C that contains neither allocation, deallocation, nor mutation statements and expectations $X, Y \in \mathbb{E}$ with $\mathbf{Mod}(C) \cap \mathbf{Vars}(Y) = \emptyset$, we have

$$\mathsf{wp}[C](X \cdot Y) \;=\; \mathsf{wp}[C](X) \cdot Y.$$

Proof. By induction on the structure of $\mathbf{P^4L}$ programs. Since we do not have to consider cases for allocation, deallocation, and mutation by the theorem's premise, lookups are the only statements that access the heap. In this case, the proof proceeds as follows:

$$\begin{aligned}
&\mathsf{wp}[x := <E>](X \cdot Y) \\
=\;& [\![\text{ Figure 7.1 }]\!] \\
&\sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \mathrel{-\!\!\star} (X \cdot Y)[x/v]) \\
=\;& [\![\text{ Theorem 6.30 (a) }]\!] \\
&\sup_{v \in \mathbb{Z}} [E \hookrightarrow v] \cdot (X \cdot Y)[x/v]) \\
=\;& [\![\, x \notin \mathbf{Vars}(Y) \text{ as } x \in \mathbf{Mod}(x := <E>) \,]\!] \\
&\sup_{v \in \mathbb{Z}} [E \hookrightarrow v] \cdot X[x/v] \cdot Y \\
=\;& [\![\, v \text{ does not occur in } Y \,]\!] \\
&\left(\sup_{v \in \mathbb{Z}} [E \hookrightarrow v] \cdot X[x/v] \right) \cdot Y \\
=\;& [\![\text{ Theorem 6.30 (a) }]\!] \\
&\left(\sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \mathrel{-\!\!\star} X[x/v]) \right) \cdot Y \\
=\;& [\![\text{ Figure 7.1 }]\!] \\
&\mathsf{wp}[x := <E>](X) \cdot Y.
\end{aligned}$$

□

7.3 Soundness of Weakest Preexpectations

In order to justify that the weakest preexpectation $\mathsf{wp}\,[C]\,(X)$ (evaluated in initial states) captures the expected value of X (evaluated in final states) after successful termination of $\mathbf{P^4L}$ program C, we show that the weakest preexpectation calculus is sound with respect to the operational semantics based on Markov decision processes presented in Section 5.3.3. Formally, our goal is to prove that[2]

$$\forall(\mathfrak{s},\mathfrak{h}) \in \mathbf{SHPairs}\colon \qquad \mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h}) \;=\; \mathsf{ExpRew}[X]\,(C,\mathfrak{s},\mathfrak{h})\,.$$

Before we present a formal proof, a few preparatory definitions are needed to set the stage: First, an *expectation calculus* is a function

$$\mathsf{ec}\colon \mathbf{P^4L} \to (\mathbb{E} \to \mathbb{E})$$

mapping $\mathbf{P^4L}$ programs to expectation transformers. We are concerned with two particular expectation calculi: The *weakest preexpectation calculus*

$$\mathsf{wp}\colon \mathbf{P^4L} \to (\mathbb{E} \to \mathbb{E}), \quad \mathsf{wp} \;\triangleq\; \lambda C\;\lambda X\;\lambda(\mathfrak{s},\mathfrak{h}).\,\mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h})$$

and the *operational expectation calculus*

$$\mathsf{op}\colon \mathbf{P^4L} \to (\mathbb{E} \to \mathbb{E}), \quad \mathsf{op} \;\triangleq\; \lambda C\;\lambda X\;\lambda(\mathfrak{s},\mathfrak{h}).\,\mathsf{ExpRew}[X]\,(C,\mathfrak{s},\mathfrak{h})\,,$$

where the expected reward $\mathsf{ExpRew}[X]\,(C,\mathfrak{s},\mathfrak{h})$ determined by our operational semantics has been introduced in Definition 5.9.

Second, the *termination completion* $\lceil \mathsf{ec} \rceil$ of expectation calculus ec is obtained by additionally assigning the identity function to the symbol term indicating successful termination and the function $\lambda X.\,0$ to the symbol $\langle \mathsf{fault} \rangle$ indicating unsuccessful termination due to a memory fault, respectively. Formally, the termination completion $\lceil \mathsf{ec} \rceil$ is defined as the function

$$\lceil \mathsf{ec} \rceil\colon \mathbf{P^4L} \cup \{\, \mathsf{term}, \langle \mathsf{fault} \rangle \,\} \to (\mathbb{E} \to \mathbb{E}),$$

$$\lceil \mathsf{ec} \rceil[C](X) \;\triangleq\; \begin{cases} X, & \text{if } C = \mathsf{term} \\ 0, & \text{if } C = \langle \mathsf{fault} \rangle \\ \mathsf{ec}[C](X), & \text{otherwise.} \end{cases}$$

Third, an expectation calculus ec is *Bellman compliant* if and only if for all $\mathbf{P^4L}$ programs C, expectations $X \in \mathbb{E}$, and stack-heap pairs $(\mathfrak{s},\mathfrak{h}) \in \mathbf{SHPairs}$, we have

$$\lceil \mathsf{ec} \rceil[C](X)(\mathfrak{s},\mathfrak{h}) \;=\; \inf_{a \in \mathbf{Act}\left(\langle C,\mathfrak{s},\mathfrak{h}\rangle\right)} \; \sum_{\langle C,\mathfrak{s},\mathfrak{h}\rangle \overset{a}{\underset{p}{\rightsquigarrow}} \langle C',\mathfrak{s}',\mathfrak{h}'\rangle} p \cdot \lceil \mathsf{ec} \rceil[C'](X)(\mathfrak{s}',\mathfrak{h}').$$

[2] We give a more detailed and cleaned up version of the original soundness proof in [1; 18].

Here, $\mathbf{Act}\left(\langle C, \mathfrak{s}, \mathfrak{h}\rangle\right)$ denotes the set of all actions enabled in the state $\langle C, \mathfrak{s}, \mathfrak{h}\rangle$ of the transition system $\mathrm{oP}^4\mathrm{L}$ that determines the operational semantics of $\mathbf{P^4L}$ programs (cf. Section 5.3.3). Moreover, as introduced in Definition 5.3, we write

$$\langle C, \mathfrak{s}, \mathfrak{h}\rangle \overset{a}{\underset{p}{\rightsquigarrow}} \langle C', \mathfrak{s}', \mathfrak{h}'\rangle$$

to denote a step of $\mathrm{oP}^4\mathrm{L}$ that is performed with action a and probability p.

The term "Bellman compliant" is a reference to Richard Bellman who developed the optimality equations commonly used for reasoning about Markov decision processes [Bel57]. It is thus not surprising that

Lemma 7.13 The operational expectation calculus op is Bellman compliant.

Proof. Since the operational expectation calculus is defined as

$$\mathsf{op} \triangleq \lambda C\ \lambda X\ \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathsf{ExpRew}[X]\,(C, \mathfrak{s}, \mathfrak{h})\,,$$

this follows immediately from Theorem 5.8. □

In fact, as shown below, op is the *least* Bellman compliant expectation calculus with respect to the ordering $\preceq$ on expectations applied pointwise:

Lemma 7.14 op is the least Bellman compliant expectation calculus.

Proof. Let ec be a Bellman compliant expectation calculus. Moreover, let $\mathrm{oP}^4\mathrm{L}\left(\langle C, \mathfrak{s}, \mathfrak{h}\rangle\right)$ be the reachable fragment of the operational semantics of $\mathbf{P^4L}$ for some fixed program C and stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ (cf. Definition 5.3). We denote by $\mathbf{R}$ the set of all executions that eventually reach a state indicating successful termination. That is,

$$\mathbf{R} \triangleq \left\{\, s_0 \dots s_n \in \mathbf{Exec}[C](\mathfrak{s}, \mathfrak{h}) \;\middle|\; \exists k \in \{\,0, \dots, n\,\} : s_k = \langle \mathsf{term}, \dots \rangle \,\right\}$$

By construction of the reward function of the MDP induced by initial state $\langle C, \mathfrak{s}, \mathfrak{h}\rangle$ (cf. Definition 5.9), only states of the form $\langle \mathsf{term}, \dots \rangle$ are assigned non-zero rewards. Hence, any path that is not also an execution in $\mathbf{R}$ contributes zero reward; it thus does not affect the value of $\mathsf{op}[C](\mathfrak{s}, \mathfrak{h})$. It then suffices to show that, for every expectation $X \in \mathbb{E}$, we have

$$\lceil \mathsf{op} \rceil[C](X)(\mathfrak{s}, \mathfrak{h}) \;\leq\; \lceil \mathsf{ec} \rceil[C](X)(\mathfrak{s}, \mathfrak{h})$$

if the execution relation $\rightsquigarrow$ of oP^{4}L $\left(\left\langle C, \mathfrak{s}, \mathfrak{h}\right\rangle\right)$ is restricted to states belonging to some execution in $\mathbf{R}$. To this end, we first notice that this restriction results in a well-founded relation with $\left\langle C, \mathfrak{s}, \mathfrak{h}\right\rangle$ as least element. We then proceed by well-founded induction on this relation.

There are two cases in which $\rightsquigarrow$ is applied at most once (to move to the sink state $\left\langle \mathsf{sink}\right\rangle$): term and $\left\langle \mathsf{fault}\right\rangle$. Both cases are straightforward by definition of termination completion, i.e.,

$$\begin{aligned}\lceil \mathsf{op}\rceil[\mathsf{term}](X)(\mathfrak{s},\mathfrak{h}) &= X(\mathfrak{s},\mathfrak{h}) = \lceil \mathsf{ec}\rceil[\mathsf{term}](X)(\mathfrak{s},\mathfrak{h}),\\ \lceil \mathsf{op}\rceil[\left\langle \mathsf{fault}\right\rangle](X)(\mathfrak{s},\mathfrak{h}) &= 0 = \lceil \mathsf{ec}\rceil[\left\langle \mathsf{fault}\right\rangle](X)(\mathfrak{s},\mathfrak{h}).\end{aligned}$$

Furthermore, for the single case in which the execution relation $\rightsquigarrow$ is applied more than once, consider the following:

$$\begin{aligned}&\lceil \mathsf{op}\rceil[C](X)(\mathfrak{s},\mathfrak{h})\\ =\ &[\![\text{ Lemma 7.13 }]\!]\\ &\inf_{a\in\mathbf{Act}\left(\left\langle C,\mathfrak{s},\mathfrak{h}\right\rangle\right)} \sum_{\left\langle C,\mathfrak{s},\mathfrak{h}\right\rangle \overset{a}{\underset{p}{\rightsquigarrow}} \left\langle C',\mathfrak{s}',\mathfrak{h}'\right\rangle} p\cdot \lceil \mathsf{op}\rceil[C'](X)(\mathfrak{s}',\mathfrak{h}')\\ \leq\ &[\![\text{ I.H. }]\!]\\ &\inf_{a\in\mathbf{Act}\left(\left\langle C,\mathfrak{s},\mathfrak{h}\right\rangle\right)} \sum_{\left\langle C,\mathfrak{s},\mathfrak{h}\right\rangle \overset{a}{\underset{p}{\rightsquigarrow}} \left\langle C',\mathfrak{s}',\mathfrak{h}'\right\rangle} p\cdot \lceil \mathsf{ec}\rceil[C'](X)(\mathfrak{s}',\mathfrak{h}')\\ =\ &[\![\text{ ec is Bellman compliant }]\!]\\ &\lceil \mathsf{ec}\rceil[C](X)(\mathfrak{s},\mathfrak{h}).\end{aligned}$$

Hence, op is the least Bellman compliant expectation calculus. □

We are now in a position to prove that our weakest preexpectation calculus is sound with respect to the operational semantics presented in Chapter 5:

Theorem 7.15 (Soundness of Weakest Preexpectation Calculus [1]) For all **P^{4}L** programs C, expectations $X \in \mathbb{E}$, and stack-heap pairs $(\mathfrak{s},\mathfrak{h})$, we have

$$\mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h}) \;=\; \mathsf{ExpRew}[X]\,(C,\mathfrak{s},\mathfrak{h})\,.$$

Proof. By definition of the weakest preexpectation calculus wp and the operational expectation calculus, i.e.,

$$\mathsf{op} \triangleq \lambda C\ \lambda X\ \lambda(\mathfrak{s},\mathfrak{h}).\, \mathsf{ExpRew}[X]\,(C,\mathfrak{s},\mathfrak{h})\,,$$

our proof obligation can be conveniently restated as $\mathsf{wp} = \mathsf{op}$.

We first show by induction on the structure of $\mathbf{P^4L}$ programs that wp is Bellman compliant.

By Lemma 7.14, this immediately yields $\mathsf{op} \preceq \mathsf{wp}$ because op is the least Bellman compliant expectation transformer.

The converse direction, i.e., $\mathsf{wp} \preceq \mathsf{op}$, is shown directly by induction on the structure of $\mathbf{P^4L}$ programs. □

7.4 Conservativity of QSL as a Verification System

Quantitative separation logic is a conservative extension of both the weakest preexpectation calculus of McIver & Morgan [MM05] and classical separation logic à la O'Hearn and Reynolds [IO01; Rey02]. Since, for programs that never access the heap, we use the same rules as McIver & Morgan, it is immediate that QSL conservatively extends weakest preexpectations.

In this section, we show that QSL also conservatively extends separation logic. To this end, recall from Section 6.5 the embedding $\mathtt{qsl}[\![.]\!]$ of predicates in classical separation logic into expectations in QSL. We first notice that, for all non-probabilistic programs, computing the weakest preexpectation with respect to a predicate Q embedded into QSL coincides with embedding the weakest precondition (cf. Definition 4.28) with respect to Q into QSL:

Lemma 7.16 For all programs $C \in \mathbf{P^3L}$ and predicates $Q \in \mathbf{Pred}$, we have

$$\underbrace{\mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])}_{\in\,\mathbb{E}} \;=\; \mathtt{qsl}[\![\underbrace{\mathsf{wp}\,[C]\,(Q)}_{\in\,\mathbf{Pred}}]\!].$$

Proof. By induction on the structure of $\mathbf{P^3L}$ programs. □

Our quantitative separation logic is then a conservative extension of classical separation logic in the following sense:

Theorem 7.17 (Conservativity of QSL as a Verification System [1]) For all $\mathbf{P^3L}$ programs C and predicates $P, Q \in \mathbf{Pred}$, we have

$$\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle \text{ is valid for total correctness} \quad \text{iff} \quad \mathtt{qsl}[\![P]\!] \preceq \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,.$$

Proof. Recall from Theorem 4.29 that the Hoare triple $\langle\, P \,\rangle\ C\ \langle\, Q \,\rangle$ is valid for total correctness if and only if P implies $\mathsf{wp}\,[C]\,(Q)$. It thus suffices to show that, for all stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$, we have

$$\mathfrak{s}, \mathfrak{h} \models P \;\Rightarrow\; \mathsf{wp}\,[C]\,(Q) \quad \text{iff} \quad \mathtt{qsl}[\![P]\!](\mathfrak{s}, \mathfrak{h}) \;\leq\; \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}).$$

We distinguish two cases. First, assume that $\mathfrak{s}, \mathfrak{h} \models P$ holds. Then:

$$\begin{aligned}
& (\mathfrak{s}, \mathfrak{h}) \models P \;\Rightarrow\; \mathsf{wp}\,[C]\,(Q) \\
\text{iff}\quad & [\![\ \text{assumption: } \mathfrak{s}, \mathfrak{h} \models P\ ]\!] \\
& \mathfrak{s}, \mathfrak{h} \models \mathsf{wp}\,[C]\,(Q) \\
\text{iff}\quad & [\![\ \text{Theorem 6.33}\ ]\!] \\
& \mathtt{qsl}[\![\mathsf{wp}\,[C]\,(Q)]\!](\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff}\quad & [\![\ \text{Lemma 7.16}\ ]\!] \\
& \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}) \;=\; 1 \\
\text{iff}\quad & [\![\ \text{Theorem 7.4 (d)}\ ]\!] \\
& 1 \;\leq\; \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}) \\
\text{iff}\quad & [\![\ \text{assumption: } \mathfrak{s}, \mathfrak{h} \models P\text{; Theorem 6.33}\ ]\!] \\
& \mathtt{qsl}[\![P]\!](\mathfrak{s}, \mathfrak{h}) \;\leq\; \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}).
\end{aligned}$$

Conversely, assume that $\mathfrak{s}, \mathfrak{h} \not\models P$ holds. In this case, we have

$$\begin{aligned}
& (\mathfrak{s}, \mathfrak{h}) \models P \;\Rightarrow\; \mathsf{wp}\,[C]\,(Q) \\
\text{iff}\quad & [\![\ \text{assumption: } \mathfrak{s}, \mathfrak{h} \not\models P\text{; false} \Rightarrow \ldots = \mathsf{true}\ ]\!] \\
& \mathsf{true} \\
\text{iff}\quad & [\![\ 0 \text{ is the least element of both } \mathbb{E} \text{ and } \mathbb{E}_{\leq 1}\ ]\!] \\
& 0 \;\leq\; \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}) \\
\text{iff}\quad & [\![\ \text{assumption: } \mathfrak{s}, \mathfrak{h} \not\models P\text{; Theorem 6.33}\ ]\!] \\
& \mathtt{qsl}[\![P]\!](\mathfrak{s}, \mathfrak{h}) \;\leq\; \mathsf{wp}\,[C]\,(\mathtt{qsl}[\![Q]\!])\,(\mathfrak{s}, \mathfrak{h}).
\end{aligned}$$

□

A key principle underlying separation logic is that correct programs are memory safe [YO02], i.e., no program execution on a stack-heap pair satisfying the precondition leads to a memory fault. By the conservativity theorem, the same

holds for weakest preexpectations when considering non-probabilistic programs with respect to predicates.

For probabilistic programs, however, we get a more fine-grained view in which the probability of encountering an error can be quantified. This allows to evaluate programs if failures are unavoidable, e.g., due to unreliable hardware. In particular, the weakest preexpectation $\mathsf{wp}\,[C]\,(1)$ measures the probability that program C terminate without a memory fault. For example,

$$\mathsf{wp}\,[\{\,\mathtt{skip}\,\}\,[1/2]\,\{\,\mathtt{free}(0)\,\}]\,(1) \;=\; 1/2 \cdot (1 + \underbrace{[0 \mapsto -]}_{=\,0}) \;=\; 1/2.$$

Hence, the above program terminates successfully with probability one half.

This situation is similar to termination. Weakest preexpectations can be used to prove that a program terminates *almost-surely*, i.e., with probability one. However, there might be non-terminating executions that occur with probability zero. In fact, weakest preexpectations are unable to prove *certain* termination, i.e., that there exists no non-terminating execution. This is not specific to QSL but already holds for McIver & Morgan's weakest preexpectation calculus [MM05].

One might suspect that the same holds when considering memory safety instead of termination. In other words, is it possible that weakest preexpectations can prove that a program is almost-surely memory safe, but not that it is certainly memory safe? Fortunately, the answer to this question is *no*: Assume there is some execution of a program C on an initial stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ that encounters a memory fault. By Theorem 7.15, this means that there is a finite path from $\langle C, \mathfrak{s}, \mathfrak{h}\rangle$ to some error state $\langle \mathsf{fault}\rangle$ indicating unsuccessful termination. Since this path is finite, it has a positive probability. Consequently, the probability that program C encounters a memory error is positive. In other words:

Corollary 7.18 (Qualitative Memory Safety [1]) A **P^{4}L** program is almost-surely memory safe if and only if it is certainly memory safe.

7.5 Weakest Liberal Preexpectations

The weakest preexpectation $\mathsf{wp}\,[C]\,(X)\,(\mathfrak{s}, \mathfrak{h})$ is the (minimal) expected value of random variable X *after successful termination* of program C on initial state $(\mathfrak{s}, \mathfrak{h})$. It is thus—as shown in Theorem 7.17—an extension of classical weakest preconditions for reasoning about *total* correctness. In this section, we briefly introduce weakest *liberal* preexpectations, which is an extension of classical weakest preconditions for reasoning about *partial* correctness (cf. Section 4.4).

7.5.1 Reasoning about Expectations

It is, unfortunately, unclear what partial correctness means within the realm of potentially unbounded expectations. As we already observed in previous chapters, the difference between weakest liberal preconditions and weakest preconditions is that the former includes initial states on which a given program does not terminate. In terms of computing weakest liberal preconditions, this difference is reflected by considering greatest rather than least fixed points. We might thus expect the same for reasoning about expected values with weakest liberal preexpectations. However, consider the program

$$C\colon \qquad \texttt{while}\,(\,x = 1\,)\,\{\ \{\,y := y+1\,\}\,[\,1/2\,]\,\{\,x := 0\,\}\ \}\,.$$

Program C terminates almost-surely because incrementing y in every iteration has probability zero. Since weakest preexpectations cannot distinguish between certain termination and almost-sure termination, there should be no difference between the weakest liberal preexpectation with respect to postexpectation y and $\mathsf{wp}\,[C]\,(y)$. Hence, the least and greatest fixed points of the loop's characteristic function $\mathfrak{W}$ should coincide. This is not the case:

$$\mathsf{lfp}\,(\mathfrak{W}) \;=\; [x \neq 1]\cdot y \quad\neq\quad [x \neq 1]\cdot y + [x = 1]\cdot\infty \;=\; \mathsf{gfp}\,(\mathfrak{W})\,.$$

Notice that program C never accesses the heap. The underlying issue is thus not specific to QSL, but applies to weakest preexpectation calculi à la McIver & Morgan [MM05] in general. To the best of our knowledge, it remains an open question how to develop expectation transformers for reasoning about potentially unbounded expected values in a partial correctness setting.

7.5.2 Reasoning about Probabilities

If we choose $\mathbb{E}_{\leq 1}$ as our underlying domain instead of $\mathbb{E}$, we reason about *probabilities* rather than expected values. That is, the weakest preexpectation $\mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h})$ is the (minimal) probability that X holds after successful termination of program C on initial state $(\mathfrak{s},\mathfrak{h})$.

When reasoning about probabilities, there is a useful notion of partial correctness: The weakest *liberal* preexpectation $\mathsf{wlp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h})$ coincides with the probability $\mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h})$ that program C terminates successfully on initial state $(\mathfrak{s},\mathfrak{h})$ plus the probability p_{diverge} that C does not terminate on $(\mathfrak{s},\mathfrak{h})$, i.e.,

$$\mathsf{wlp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h}) \;=\; \mathsf{wp}\,[C]\,(X)\,(\mathfrak{s},\mathfrak{h}) \;+\; p_{\mathit{diverge}}.$$

In this section, we briefly summarize how a weakest liberal preexpectation calculus for **P⁴L** programs differs from the calculus in Chapter 7. Following McIver & Morgan [MM05], the weakest liberal preexpectation transformer

$$\mathsf{wlp}\,[C] : \mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1}$$

is defined analogously to the weakest preexpectation transformer $\mathsf{wp}\,[C]$ (cf. Definition 7.1) with two exceptions:

1. we compute *greatest* fixed points instead of least fixed points, and
2. we substitute every occurrence of wp by wlp.

In particular, the rules of wlp for loops and procedure calls are

$$\mathsf{wlp}\left[\texttt{while}\,(\,B\,)\,\{\,C'\,\}\right](\mathrm{X}) \;\triangleq\; \mathsf{gfp}\,(\mathfrak{LW})\;\text{ and}$$
$$\mathsf{wlp}\,[x := F(E_1,\ldots,E_n)]\,(\mathrm{X}) \;\triangleq\; \mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(E_1,\ldots,E_n)(\mathrm{X}),$$

where the liberal characteristic function $\mathfrak{LW}$ of loop $\texttt{while}\,(\,B\,)\,\{\,C'\,\}$ is

$$\mathfrak{LW} \;\triangleq\; [\neg B]\cdot \mathrm{X} + [B]\cdot \mathsf{wlp}\left[C'\right](\mathrm{Y})$$

and the liberal characteristic function $\mathfrak{LP}_F$ of procedure F is defined as

$$\mathfrak{LP}_F(\theta) \;\triangleq\; \lambda x'\,\lambda(E_1',\ldots,E_n')\,\lambda \mathrm{X}'.$$
$$\mathsf{wlp}_F^{\theta}\left[\mathsf{body}\,(F)\right]\left(\mathrm{X}'\left[-\boxdot\right]\left[\boxdot x'/\mathsf{out}\right]\right)\left[x_1/E_1'\left[-\boxdot\right]\right]\ldots\left[x_n/E_n'\left[-\boxdot\right]\right]\left[+\boxdot\right].$$

A complete definition of the rules that determine the weakest liberal preexpectation calculus wlp is found in Figure 7.4. Most properties of weakest preexpectations considered in Chapter 7 also hold for weakest liberal preexpectations. However, we occasionally have to dualize theorems to account for taking greatest rather than least fixed points. We briefly list the most important properties of wlp; the proofs are analogous to the theorems presented in Chapter 7.

Let us first collect basic properties of wlp (cf. Theorem 7.4):

Theorem 7.19 (Basic Properties of wlp [1]) For all $\mathbf{P^4L}$-programs C, expectations $\mathrm{X},\mathrm{Y}\in\mathbb{E}_{\leq 1}$, predicates P and constants $k\in\mathbb{R}_{\geq 0}$, we have:

(a) Monotonicity: $\mathrm{X}\preceq\mathrm{Y}$ implies $\mathsf{wlp}\,[C]\,(\mathrm{X})\preceq\mathsf{wlp}\,[C]\,(\mathrm{Y})$.

Additionally, if C contains no allocation statements, we have:

(b) ω-continuity: For every increasing ω-chain $\mathrm{X}_1\preceq\mathrm{X}_2\preceq\ldots$ in $\mathbb{E}$,

$$\sup_n \mathsf{wlp}\,[C]\,(\mathrm{X}_n) \;=\; \mathsf{wlp}\,[C]\,(\sup_n \mathrm{X}_n)\,,\text{ and}$$

(c) Sub–Linearity: $\mathsf{wlp}\,[C]\,(k\cdot\mathrm{X}+\mathrm{Y})\preceq k\cdot\mathsf{wlp}\,[C]\,(\mathrm{X})+\mathsf{wlp}\,[C]\,(\mathrm{Y})$.

Moreover, the weakest liberal preexpectation of loops is well-defined and can be characterized by fixed point iteration (cf. Theorem 7.5):

Theorem 7.20 For every loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and expectation $\mathrm{X} \in \mathbb{E}_{\leq 1}$ with liberal characteristic function $\mathfrak{LW}$, there exists an ordinal α such that

$$\mathsf{wlp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;=\; \mathsf{gfp}\,(\mathfrak{LW}) \;=\; \mathfrak{LW}^{\alpha}(1).$$

We then obtain a proof rule dual to Theorem 7.7 for reasoning about *lower bounds* on weakest liberal preexpectations of loops:

Theorem 7.21 For every loop $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and expectations $\mathrm{X}, \mathrm{I} \in \mathbb{E}_{\leq 1}$ with liberal characteristic function $\mathfrak{LW}$, we have

$$\mathfrak{LW}(\mathrm{I}) \;\succeq\; \mathrm{I} \quad \text{implies} \quad \mathsf{wlp}\,[\texttt{while}\,(\,B\,)\,\{\,C\,\}]\,(\mathrm{X}) \;\succeq\; \mathrm{I}.$$

In this case, we call I a (lower) *invariant* of $\texttt{while}\,(\,B\,)\,\{\,C\,\}$ and X.

Analogously, we obtain the following dual versions of Theorems 7.8 and 7.9 for reasoning about procedures calls:

Theorem 7.22 For every call $x := F(E_1, \ldots, E_n)$ and expectation $\mathrm{X} \in \mathbb{E}_{\leq 1}$ with liberal characteristic function $\mathfrak{LP}_F$, there exists an ordinal α such that

$$\begin{aligned}\mathsf{wlp}\,[x := F(E_1, \ldots, E_n)]\,(\mathrm{X}) \;&= \mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(E_1, \ldots, E_n)(\mathrm{X}) \\ &= \mathfrak{LP}_F^{\alpha}(0)(x)(E_1, \ldots, E_n)(\mathrm{X}).\end{aligned}$$

Theorem 7.23 ([1]) For every call $x := F(E_1, \ldots, E_n)$, expectation $\mathrm{X} \in \mathbb{E}_{\leq 1}$, and monotone transformer $\theta\colon \mathbf{Vars} \to \mathbf{AE}^n \to (\mathbb{E}_{\leq 1} \to \mathbb{E}_{\leq 1})$, we have

$$\mathfrak{LP}_F(\theta) \;\succeq\; \theta \quad \text{implies} \quad \mathsf{wlp}\,[x := F(E_1, \ldots, E_n)]\,(\mathrm{X}) \;\succeq\; \theta(x)(E_1, \ldots, E_n)(\mathrm{X}).$$

Finally, the quantitative frame rule, which has been discussed in detail for wp in Theorem 7.10, applies to weakest liberal preexpectations as well.

Theorem 7.24 (Quantitative Frame Rule for wlp [1]) For every $\mathbf{P^4L}$-program C and expectations $\mathrm{X}, \mathrm{Y} \in \mathbb{E}_{\leq 1}$ with $\mathbf{Mod}\,(C) \cap \mathbf{Vars}(\mathrm{Y}) = \emptyset$, we have

$$\mathsf{wlp}\,[C]\,(\mathrm{X}) \star \mathrm{Y} \;\preceq\; \mathsf{wlp}\,[C]\,(\mathrm{X} \star \mathrm{Y})\,.$$

C	$\mathbf{Mod}(C)$
$\{ C_1 \}\ [p]\ \{ C_2 \}$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
$x :\approx \mu$	$\{ x \}$
`skip`	$\emptyset$
$x := E$	$\{ x \}$
$x :=$ `alloc`$(E_1, \ldots, E_n)$	$\{ x \}$
`free`(E)	$\emptyset$
$x := E$	$\{ x \}$
$<E> := E'$	$\emptyset$
$C_1 \,;\, C_2$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`if` (B) $\{ C_1 \}$ `else` $\{ C_2 \}$	$\mathbf{Mod}(C_1) \cup \mathbf{Mod}(C_2)$
`while` (B) $\{ C' \}$	$\mathbf{Mod}(C')$
$x := F(E_1, \ldots, E_n)$	$\{ x \}$

Figure 7.3: Inductive definition of $\mathbf{Mod}(C)$ for $\mathbf{P^4L}$ programs.

C	$\mathsf{wlp}\,[C]\,(\mathrm{X})$
`skip`	X
$x := E$	$\mathrm{X}\,[x/E]$
$x :\approx \mu$	$\lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}} \mu(\mathfrak{s})(v)\cdot \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h})$
$x := \texttt{alloc}(E_1,\ldots,E_n)$	$\inf_{v\in\mathbb{Z}} [v\mapsto E_1,\ldots,E_n] \mathrel{-\!\!*} \mathrm{X}\,[x/v]$
$\texttt{free}(E)$	$[E\mapsto -]\star\mathrm{X}$
$x := <E>$	$\sup_{v\in\mathbb{Z}} [E\mapsto v]\star([E\mapsto v]\mathrel{-\!\!*}\mathrm{X}\,[x/v])$
$<E> := E'$	$[E\mapsto -]\star([E\mapsto E']\mathrel{-\!\!*}\mathrm{X})$
$C_1\,;\,C_2$	$\mathsf{wlp}\,[C_1]\,(\mathsf{wlp}\,[C_2]\,(\mathrm{X}))$
$\texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}$	$[B]\cdot\mathsf{wlp}\,[C_1]\,(\mathrm{X}) \;+\; [\neg B]\cdot\mathsf{wlp}\,[C_2]\,(\mathrm{X})$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p\cdot\mathsf{wlp}\,[C_1]\,(\mathrm{X}) \;+\; (1-p)\cdot\mathsf{wlp}\,[C_2]\,(\mathrm{X})$
$\texttt{while}\,(\,B\,)\,\{\,C'\,\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where
	$\mathfrak{LW} \triangleq \lambda\mathrm{I}.\ [B]\cdot\mathsf{wlp}\,[C']\,(\mathrm{I}) \;+\; [\neg B]\cdot\mathrm{X}$
$x := F(E_1,\ldots,E_n)$	$\mathsf{gfp}\,(\mathfrak{LP}_F)\,(x)(E_1,\ldots,E_n)(\mathrm{X})$, where
$\mathfrak{LP}_F \triangleq \lambda\theta\lambda x'\lambda(E_1',\ldots,E_n')\lambda\mathrm{X}'.$	
$\mathsf{wlp}^F_\theta\,[\mathsf{body}\,(F)]\,\big(\mathrm{X}'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,[x_1/E_1'\,[-\boxdot]]\ldots[x_n/E_n'\,[-\boxdot]]\,[+\boxdot]$	

Figure 7.4: Rules of the weakest liberal preexpectation transformer $\mathsf{wlp}\,[C]$ for **P⁴L** programs. The higher-order transformer $\mathsf{wlp}^F_\theta\,[C]$ is found in Figure 7.6, page 229.

C	$\mathsf{wp}_\theta^F\,[C]\,(\mathrm{X})$
`skip`	X
$x := E$	$\mathrm{X}\,[x/E]$
$x :\approx \mu$	$\lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}} \mu(\mathfrak{s})(v)\cdot \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h})$
$x := \texttt{alloc}(E_1,\ldots,E_n)$	$\inf_{v\in\mathbb{Z}}\,[v\mapsto E_1,\ldots,E_n] \mathrel{-\!\!\!\star} \mathrm{X}\,[x/v]$
$\texttt{free}(E)$	$[E\mapsto -]\star\mathrm{X}$
$x := <E>$	$\sup_{v\in\mathbb{Z}}\,[E\mapsto v]\star([E\mapsto v]\mathrel{-\!\!\!\star}\mathrm{X}\,[x/v])$
$<E> := E'$	$[E\mapsto -]\star([E\mapsto E']\mathrel{-\!\!\!\star}\mathrm{X})$
$C_1\,;\,C_2$	$\mathsf{wp}_\theta^F\,[C_1]\,(\mathsf{wp}_\theta^F\,[C_2]\,(\mathrm{X}))$
$\texttt{if}\,(\,B\,)\,\{\,C_1\,\}\,\texttt{else}\,\{\,C_2\,\}$	$[B]\cdot\mathsf{wp}_\theta^F\,[C_1]\,(\mathrm{X}) \;+\; [\neg B]\cdot\mathsf{wp}_\theta^F\,[C_2]\,(\mathrm{X})$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p\cdot\mathsf{wp}_\theta^F\,[C_1]\,(\mathrm{X}) \;+\; (1-p)\cdot\mathsf{wp}_\theta^F\,[C_2]\,(\mathrm{X})$
$\texttt{while}\,(\,B\,)\,\{\,C'\,\}$	$\mathsf{lfp}\,(\mathfrak{W})$, where
	$\mathfrak{W} \triangleq \lambda\mathrm{I}.\ [B]\cdot\mathsf{wp}_\theta^F\,[C']\,(\mathrm{I}) \;+\; [\neg B]\cdot\mathrm{X}$
$x := F(E_1,\ldots,E_n)$	$\theta\ (x)(E_1,\ldots,E_n)(\mathrm{X})$
$x := G(E_1,\ldots,E_m)$	$\mathsf{lfp}\,(\mathfrak{P}_G)\ (x)(E_1,\ldots,E_m)(\mathrm{X})$, where
$\mathfrak{P}_G \;\triangleq\; \lambda\theta\lambda x'\lambda(E_1',\ldots,E_m')\lambda\mathrm{X}'.$	
$\mathsf{wp}_\theta^G\,[\mathsf{body}\,(G)]\,\big(\mathrm{X}'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}]\big)\,\big[x_1/E_1'\,[-\boxdot]\big]\ldots\big[x_m/E_m'\,[-\boxdot]\big]\,[+\boxdot]$	

Figure 7.5: Rules of the higher-order weakest preexpectation transformer $\mathsf{wp}_\theta^F\,[C]$ that depends on the transformer θ to resolve calls of procedure F.

C	$\mathsf{wlp}^F_\theta\,[C]\,(\mathrm{X})$
`skip`	X
$x := E$	$\mathrm{X}\,[x/E]$
$x :\approx \mu$	$\lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v\in\mathbb{Z}} \mu(\mathfrak{s})(v)\cdot \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h})$
$x := \texttt{alloc}(E_1,\ldots,E_n)$	$\inf_{v\in\mathbb{Z}} [v\mapsto E_1,\ldots,E_n] \mathrel{-\!\!\!\circledast} \mathrm{X}\,[x/v]$
$\texttt{free}(E)$	$[E\mapsto -]\star\mathrm{X}$
$x := <E>$	$\sup_{v\in\mathbb{Z}} [E\mapsto v]\star([E\mapsto v]\mathrel{-\!\!\!\circledast}\mathrm{X}\,[x/v])$
$<E> := E'$	$[E\mapsto -]\star([E\mapsto E']\mathrel{-\!\!\!\circledast}\mathrm{X})$
$C_1\,;\,C_2$	$\mathsf{wlp}^F_\theta\,[C_1]\left(\mathsf{wlp}^F_\theta\,[C_2]\,(\mathrm{X})\right)$
$\texttt{if}\ (\,B\,)\ \{\,C_1\,\}\ \texttt{else}\ \{\,C_2\,\}$	$[B]\cdot\mathsf{wlp}^F_\theta\,[C_1]\,(\mathrm{X}) \;+\; [\neg B]\cdot\mathsf{wlp}^F_\theta\,[C_2]\,(\mathrm{X})$
$\{\,C_1\,\}\,[\,p\,]\,\{\,C_2\,\}$	$p\cdot\mathsf{wlp}^F_\theta\,[C_1]\,(\mathrm{X}) \;+\; (1-p)\cdot\mathsf{wlp}^F_\theta\,[C_2]\,(\mathrm{X})$
$\texttt{while}\,(\,B\,)\,\{\,C'\,\}$	$\mathsf{gfp}\,(\mathfrak{LW})$, where
	$\mathfrak{LW} \triangleq \lambda\mathrm{I}.\ [B]\cdot\mathsf{wlp}^F_\theta\,[C']\,(\mathrm{I}) \;+\; [\neg B]\cdot\mathrm{X}$
$x := F(E_1,\ldots,E_n)$	$\theta\ (x)(E_1,\ldots,E_n)(\mathrm{X})$
$x := G(E_1,\ldots,E_m)$	$\mathsf{gfp}\,(\mathfrak{LP}_G)\,(x)(E_1,\ldots,E_m)(\mathrm{X})$, where
$\mathfrak{LP}_G \triangleq \lambda\theta\lambda x'\lambda(E'_1,\ldots,E'_m)\lambda X'.$	
$\mathsf{wlp}^G_\theta\,[\mathsf{body}\,(G)]\,(\mathrm{X}'\,[-\boxdot]\,[\boxdot x'/\mathsf{out}])\,[x_1/E'_1\,[-\boxdot]]\ldots[x_m/E'_m\,[-\boxdot]]\,[+\boxdot]$	

Figure 7.6: Rules of the higher-order weakest liberal preexpectation transformer $\mathsf{wlp}^F_\theta\,[C]$ that depends on the transformer θ to resolve calls of procedure F.

Chapter 8

Case Studies

This chapter is based on prior publications, namely [1; 18], which are presented, discussed, and extended in the broader context of this thesis.

In this chapter, we demonstrate how quantitative separation logic enables formal reasoning about both expected values and probabilities of probabilistic pointer programs. To this end, we apply the weakest preexpectation calculi introduced in the previous chapter to five case studies:

First, in Section 8.1 we consider a simple randomized algorithm that extends a given list. This example is sufficiently small such that we can discuss all involved calculations in detail. After that, we analyze randomized variants of two classical examples from the separation logic literature (cf. [Bor00; Rey02; OHe12; KTB17]): In Section 8.2, we reason about a lossy list reversal algorithm. Moreover, in Section 8.3, we study a faulty version of the garbage collection procedure presented in Example 4.34. Section 8.4 is concerned with a recurring problem in the design of randomized algorithms: How do we randomize a given input? More precisely, we prove the correctness of a textbook algorithm, which is taken from [Cor+09], for computing random permutations of arrays. Finally, in Section 8.5, we formally verify a complexity result by Gambin and Malinowski [GM98] on randomized meldable priority queues.

Each of our case studies roughly follows the same structure:

1. We briefly motivate the $\mathbf{P}^4\mathbf{L}$ program under consideration and explain its usage of randomization (cf. Section 5.1.1).
2. We outline the property that is analyzed and explain how it is formalized as a (post)expectation. Occasionally, this involves the introduction of specialized expectations, such as a recursive definition.
3. We discuss all invariants required to approximate weakest (liberal) preexpectations of loops and recursive procedure calls.
4. We compute either an upper or a lower bound on the weakest (liberal) preexpectation of the program in question with respect to the chosen

postexpectation. The computation is presented in an annotation style on source code level as introduced in Example 2.29, page 45.

5. Finally, we apply the proof rules presented in Chapter 6 (as well as elementary facts on predicate logic and real-valued functions) to discharge all quantitative entailments that arise during the computation.

To enable reading proofs in a top-down style, we present technical lemmas after the first proof in which they have been applied. In order to understand the main idea, it is thus safe to skip proofs at the end of each section. Apart from exemplary excerpts, the detailed (and rather tedious) calculations required to verify quantitative entailments are found in Appendix F. A reader might find the succinct collection of proof rules in Appendix D—to which we refer whenever a rule is applied—a helpful companion while reading formal proofs.

Assumption 8.1 To avoid brackets and keep expectations readable, let us agree on the following order of precedence $\gg$ for most common operators:

$$[x/E], [+\boxdot], [-\boxdot] \;\gg\; \star \;\gg\; \cdot \;\gg\; \mathrel{-\!\!\star} \;\gg\; +, - \;\gg\; \inf_{v\in\mathbb{Z}}, \sup_{v\in\mathbb{Z}}.$$

Hence, the expectation

$$\sup_{v\in\mathbb{Z}} [x \neq v] \cdot \mathsf{X}\,[x/3]\,[+\boxdot] \star [z \mapsto 0] + \mathbf{size} \star [v \mapsto -] \mathrel{-\!\!\star} \mathbf{emp}$$

is meant to be read as

$$\sup_{v\in\mathbb{Z}} (([x \neq v] \cdot (((\mathsf{X}\,[x/3])\,[+\boxdot]) \star [z \mapsto 0])) + ((\mathbf{size} \star [v \mapsto -]) \mathrel{-\!\!\star} \mathbf{emp})).$$

8.1 Randomized List Extension

Let us begin with a toy example that is simple enough such that we can discuss its verification in full detail. We consider a randomized algorithm that takes a null-terminated singly-linked list with head hd and produces a list with head hd which has been extended by a random number of elements. An implementation of this algorithm is given by the $\mathbf{P^4L}$ program C_{extend} in Figure 8.1. Intuitively, C_{extend} keeps flipping a fair coin. As long as the coin flip yields heads, i.e., $\mathsf{coin} = 1$ holds, we allocate a new address, say v, point it to the current head of the list, i.e., the address stored in variable hd, and assign v to hd afterward. Once a coin flip yields tails, i.e., $\mathsf{coin} \neq 1$ holds, the program terminates. Notice that program C_{extend} does not certainly terminate, because the coin flip might yield

$$
\begin{array}{ll}
 & \mathsf{coin} := 1\,; \\
 & \texttt{while}\,(\,\mathsf{coin} = 1\,)\,\{ \\
 & \quad \{ \\
 & \quad\quad \mathsf{coin} := 0 \\
C_{\mathsf{body}}: & \quad \}\,[\,1/2\,]\,\{ \\
 & \quad\quad \mathsf{hd} := \texttt{alloc}(\mathsf{hd}) \\
 & \quad \} \\
 & \}
\end{array}
$$

Figure 8.1: $\mathbf{P^4L}$ program C_{extend}.

heads all the time. It does, however, terminate almost-surely as the probability of seeing heads all the time is zero. Furthermore, we observe that C_{extend} produces a list whose length is an arbitrary natural number greater than or equal to the length of the original list. How large is the mean of this natural number?

Our goal is to determine an upper bound on the expected length of the list with head hd produced by program C_{extend}. Hence, we have to compute the weakest preexpectation of C_{extend} with respect to postexpectation

$$\mathrm{X} \triangleq \boldsymbol{len}\,(\mathsf{hd}, 0)\,,$$

where $\boldsymbol{len}\,(\mathsf{hd}, 0)$ measures the length of a singly-linked list with head hd and tail 0. Formally, as discussed in Section 6.6, the quantity $\boldsymbol{len}\,(u, v)$ is given by

$$\boldsymbol{len}\,(u, v) \triangleq [u \neq v] \cdot \sup_{w \in \mathbb{Z}} [u \mapsto w] \star ([\boldsymbol{sll}\,(w, v)] + \boldsymbol{len}\,(w, v))\,.$$

To reason about the loop in program C_{extend}, we propose the invariant

$$\mathrm{I} \triangleq \boldsymbol{len}\,(\mathsf{hd}, 0) + [\mathsf{coin} = 1]\,.$$

In other words the length of the list is increased, on average, by one if variable coin equals one. Let us assume, for the moment, that I is a suitable upper invariant in the sense of Theorem 7.7. That is, we are equipped with the following lemma:

Lemma 8.2 ([18]) $\mathsf{wp}\,[\texttt{while}\,(\,\mathsf{coin} = 1\,)\,\{\,C_{\mathsf{body}}\,\}]\,(\boldsymbol{len}\,(\mathsf{hd}, 0)) \preceq \mathrm{I}$.

With this loop variant at hand, the computation of an upper bound on the weakest preexpectation $\mathsf{wp}\,[C_{\mathsf{extend}}]\,(X)$ proceeds as follows:

$$
\begin{array}{l}
\text{// } \quad \mathit{len}\,(\mathsf{hd},0)+1 \\
\text{// } \quad \succeq \quad [\![\text{ elementary algebra }]\!] \\
\text{// } \quad \mathit{len}\,(\mathsf{hd},0)+[1=1] \\
\mathsf{coin} := 1\,; \\
\text{// } \quad \mathit{len}\,(\mathsf{hd},0)+[\mathsf{coin}=1] \\
\text{// } \quad \succeq \quad [\![\text{ Lemma 8.2 }]\!] \\
\text{// } \quad \mathsf{wp}\,[\texttt{while}\,(\,\mathsf{coin}=1\,)\,\{\,C_{\mathsf{body}}\,\}]\,(\mathit{len}\,(\mathsf{hd},0)\,) \\
\texttt{while}\,(\,\mathsf{coin}=1\,)\,\{\,C_{\mathsf{body}}\,\} \\
\text{// } \quad \mathit{len}\,(\mathsf{hd},0) \;=\; X
\end{array}
$$

Hence, *the expected length of the list after execution of program C_{extend} is at most one plus its original length.* To complete our analysis of program C_{extend}, it remains to verify that I is indeed an invariant, i.e., we have to prove Lemma 8.2.

Proof (of Lemma 8.2). Let us denote by $\mathfrak{W}$ the characteristic function of loop $\texttt{while}\,(\,\mathsf{coin}=1\,)\,\{\,C_{\mathsf{body}}\,\}$ with respect to postexpectation $\mathit{len}\,(\mathsf{hd},0)$. By Theorem 7.7, it suffices to verify that $\mathfrak{W}(\mathrm{I}) \preceq \mathrm{I}$ holds to conclude that

$$\mathsf{wp}\,[\texttt{while}\,(\,\mathsf{coin}=1\,)\,\{\,C_{\mathsf{body}}\,\}]\,(\mathit{len}\,(\mathsf{hd},0)) \;\preceq\; \mathrm{I}.$$

To this end, consider the following calculations:

$$
\begin{array}{ll}
 & \mathfrak{W}(\mathrm{I}) \\
= & [\![\text{ Definition of characteristic function } \mathfrak{W}(\mathrm{I}) \text{ }]\!] \\
 & [\mathsf{coin}=1]\cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I}) + [\mathsf{coin}\neq 1]\cdot \mathit{len}\,(\mathsf{hd},0) \\
= & [\![\text{ Computation of } \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I}) \text{ in Figure 8.2 (page 235) }]\!] \\
 & [\mathsf{coin}=1]\cdot (1/2\cdot \mathit{len}\,(\mathsf{hd},0) \\
 & \qquad + 1/2\cdot \inf_{v\in\mathbb{Z}} [v\mapsto \mathsf{hd}] \mathrel{-\!\!\!\rightarrow} (\mathit{len}\,(v,0)+[\mathsf{coin}=1])) \\
 & \qquad + [\mathsf{coin}\neq 1]\cdot \mathit{len}\,(\mathsf{hd},0) \\
\preceq & [\![\text{ D.6.1; D.1.5 }]\!] \\
 & [\mathsf{coin}=1]\cdot (1/2\cdot \mathit{len}\,(\mathsf{hd},0) + 1/2\cdot \inf_{v\in\mathbb{Z}} [v\mapsto \mathsf{hd}] \mathrel{-\!\!\!\rightarrow} (\mathit{len}\,(v,0)+1)) \\
 & \qquad + [\mathsf{coin}\neq 1]\cdot \mathit{len}\,(\mathsf{hd},0) \\
\preceq & [\![\text{ Lemma 8.3 (page 235) }]\!]
\end{array}
$$

```
//  1/2 · len(hd, 0)
//  + 1/2 · inf_{v∈ℤ} [v ↦ hd] −⋆ (len(v, 0) + [coin = 1])
{
    //  len(hd, 0)
    //  ⪰   ⟦ elementary algebra ⟧
    //  len(hd, 0) + [0 = 1]
    coin := 0
    //  len(hd, 0) + [coin = 1]
} [1/2] {
    //  inf_{v∈ℤ} [v ↦ hd] −⋆ (len(v, 0) + [coin = 1])
    hd := alloc(hd)
    //  len(hd, 0) + [coin = 1]
}
//  len(hd, 0) + [coin = 1]  =  I
```

Figure 8.2: Computation of $\mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I})$.

$$
\begin{aligned}
&[\mathsf{coin} = 1] \cdot (1/2 \cdot \mathit{len}\,(\mathsf{hd}, 0) + 1/2 \cdot (\mathit{len}\,(\mathsf{hd}, 0) + 2)) \\
&\qquad + [\mathsf{coin} \neq 1] \cdot \mathit{len}\,(\mathsf{hd}, 0) \\
=\ & [\![\text{ elementary algebra }]\!] \\
&[\mathsf{coin} = 1] \cdot \mathit{len}\,(\mathsf{hd}, 0) + [\mathsf{coin} \neq 1] \cdot \mathit{len}\,(\mathsf{hd}, 0) + [\mathsf{coin} = 1] \\
=\ & [\![\text{ D.6.4 }]\!] \\
&\mathit{len}\,(\mathsf{hd}, 0) + [\mathsf{coin} = 1] \\
=\ & [\![\text{ Definition of I }]\!] \\
&\mathrm{I}.
\end{aligned}
$$

□

Our last proof obligation is to verify all quantitative entailments which we previously exploited without providing a proof. In this case study, we deferred one such inequality to a separate lemma (in the fourth step of the above proof).

Lemma 8.3 ([18]) $\inf_{v \in \mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!\star} (\mathit{len}\,(v, 0) + 1) \preceq \mathit{len}\,(\mathsf{hd}, 0) + 2$.

Proof. Consider the following:

$$
\begin{aligned}
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} (\mathit{len}\,(v,0)+1) \\
= \;& [\![\text{ Definition of } \mathit{len}\,(\mathsf{hd},0) \text{ }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} ([v \neq 0] \cdot \sup_{w\in\mathbb{Z}} [v \mapsto w] \star ([\mathit{sll}\,(w,0)] + \mathit{len}\,(w,0)) + 1) \\
\preceq \;& [\![\text{ D.1.4; D.1.5; D.6.1 }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} (1 \cdot \sup_{w\in\mathbb{Z}} [v \mapsto w] \star (1 + \mathit{len}\,(w,0)) + 1) \\
= \;& [\![\text{ D.1.26 }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} ([v \hookrightarrow \mathsf{hd}] \cdot 1 \cdot \sup_{w\in\mathbb{Z}} [v \mapsto w] \star (1 + \mathit{len}\,(w,0)) + 1) \\
= \;& [\![\text{ introduce fresh variable } z \text{ for } w \text{ }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} (\\
& \qquad [v \hookrightarrow \mathsf{hd}] \cdot \sup_{w\in\mathbb{Z}} ([v \mapsto z] \star (1 + \mathit{len}\,(z,0)) + 1)\,[z/w]) \\
= \;& [\![\text{ D.6.10 }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} (\\
& \qquad \sup_{w\in\mathbb{Z}} ([v \hookrightarrow \mathsf{hd}] \cdot [v \mapsto z] \star (1 + \mathit{len}\,(z,0)) + 1)\,[z/w]) \\
= \;& [\![\text{ D.7.7 }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} (\\
& \qquad \sup_{w\in\mathbb{Z}} ([\mathsf{hd} = z] \cdot [v \mapsto z] \star (1 + \mathit{len}\,(z,0)) + 1)\,[z/w]) \\
= \;& [\![\text{ D.6.11 }]\!] \\
& \inf_{v\in\mathbb{Z}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} ([v \mapsto \mathsf{hd}] \star (1 + \mathit{len}\,(\mathsf{hd},0)) + 1) \\
= \;& [\![\text{ D.5.10 }]\!] \\
& \inf_{v\in\mathrm{addr}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} ([v \mapsto \mathsf{hd}] \star (1 + \mathit{len}\,(\mathsf{hd},0)) + 1) \\
= \;& [\![\text{ D.4.2 }]\!] \\
& \inf_{v\in\mathrm{addr}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} [v \mapsto \mathsf{hd}] \star (1 + \mathit{len}\,(\mathsf{hd},0)) + [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} 1 \\
= \;& [\![\text{ D.7.9 }]\!] \\
& \inf_{v\in\mathrm{addr}} [v \mapsto \mathsf{hd}] \mathrel{-\!\!*} [v \mapsto \mathsf{hd}] \star (1 + \mathit{len}\,(\mathsf{hd},0))
\end{aligned}
$$

$$
\begin{aligned}
&\quad + [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot 1 \\
&= \llbracket \text{ D.1.26 } \rrbracket \\
&\quad \inf_{v \in \mathrm{addr}} [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot (1 + \mathit{len}(\mathsf{hd}, 0)) \\
&\quad\quad + [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot 1 \\
&= \llbracket \text{ elementary algebra (notice that } \infty + \infty = \infty) \rrbracket \\
&\quad \inf_{v \in \mathrm{addr}} [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot (2 + \mathit{len}(\mathsf{hd}, 0)) \\
&= \llbracket \text{ D.5.11 } \rrbracket \\
&\quad \inf_{v \in \mathrm{addr}} 2 + \mathit{len}(\mathsf{hd}, 0) \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad \mathit{len}(\mathsf{hd}, 0) \ +2.
\end{aligned}
$$

□

8.2 Lossy List Reversal

List reversal algorithms are a prominent class of benchmarks when analyzing heap manipulating programs (cf. [Bor00; Rey02; Mag+06; BIP10; Atk11; KTB17]). The **P⁴L** program C_{lrev} below models a *lossy* list reversal that is performed in-place, i.e., the input list is reversed without copying elements:

C_{body}:

```
rv := 0;
while ( hd ≠ 0 ) {
    x := hd.n;
    {
        hd.n := rv;
        rv := hd
    } [1/2] {
        free(hd)
    };
    hd := x
}
```

Program C_{lrev} traverses a list with head hd and attempts to move each element to the front of an initially empty list with head rv. Consequently, variable rv points to the head of a reversed list upon termination. However, during each iteration, another behavior is possible: Rather than moving an element from the initial list to the reversed list, the current element might be dropped with probability

$1/2$. Notice that dropping an element is not necessarily caused by a hardware failure. For example, it is conceivable that another process running in parallel to a standard list reversal algorithm occasionally consumes a list element.

The lossy list reversal raises several questions which we could attempt to verify using weakest preexpectations: Does C_{lrev} encounter a memory failure with some probability? What is the probability that C_{lrev} returns a reversed version of the original list without dropping any elements? What is the *expected length* of the reversed list upon termination of C_{lrev}? In this section, we concentrate on the last question. That is, our goal is to compute an upper bound on the weakest preexpectation of C_{lrev} with respect to postexpectation $\mathrm{X} \triangleq \boldsymbol{len}\,(\mathsf{rv}, 0)$, where the length $\boldsymbol{len}\,(u, v)$ of a list segment with head u and tail v is given by

$$\boldsymbol{len}\,(u,v) \;\triangleq\; [u \neq v] \cdot \sup_{w \in \mathbb{Z}} [u \mapsto w] \star ([\boldsymbol{sll}\,(w,v)] + \boldsymbol{len}\,(w,v))$$

as introduced in Section 6.6. To this end, we propose the loop invariant:

$$\mathrm{I} \;\triangleq\; \boldsymbol{len}\,(\mathsf{rv},0) \star [\boldsymbol{sll}\,(\mathsf{hd},0)] + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \boldsymbol{len}\,(\mathsf{hd},0) \star [\boldsymbol{sll}\,(\mathsf{rv},0)]\,.$$

Intuitively, program C_{lrev} manages two lists with heads rv and hd, respectively. The list with head rv consists of all list elements which have already been reversed. Moreover, the list with head hd consists of the remaining elements of the original list. Invariant I then states that during each loop iteration, the expected length of the reversed list is its current length $\boldsymbol{len}\,(\mathsf{rv},0)$ plus half the length of the remaining list. Furthermore, we added predicates $[\boldsymbol{sll}\,(\mathsf{hd},0)]$ and $[\boldsymbol{sll}\,(\mathsf{rv},0)]$ to account for the exact layout of the heap. Let us assume, for the moment, that I is an invariant in the sense of Theorem 7.7. That is,

Lemma 8.4 ([18]) $\mathsf{wp}\,[\texttt{while}\,(\,\mathsf{coin} = 1\,)\,\{\,C_{\mathsf{body}}\,\}]\,(\boldsymbol{len}\,(\mathsf{rv},0)) \;\preceq\; \mathrm{I}.$

The calculations below then yield our desired upper bound on $\mathsf{wp}\,[C_{\mathsf{lrev}}]\,(\mathrm{X})$:

```
//  1/2 · [hd ≠ 0] · len (hd, 0)
//    ⪰    ⟦ Lemma 8.5; D.7.2; Lemma 8.6; D.1.2 ⟧
//  len (0, 0) ⋆ [sll (hd, 0)] + 1/2 · [hd ≠ 0] · len (hd, 0) ⋆ [sll (0, 0)]
rv := 0;
//  len (rv, 0) ⋆ [sll (hd, 0)] + 1/2 · [hd ≠ 0] · len (hd, 0) ⋆ [sll (rv, 0)]
//    ⪰    ⟦ Lemma 8.4 ⟧
//  wp [while ( hd ≠ 0 ) { C_body }] ( len (rv, 0) )
while ( hd ≠ 0 ) { C_body }
//  len (rv, 0)  =  X
```

Hence, *in expectation, the reversed list produced by program C_{lrev} is at most half as long as the original one.* Apart from applying invariant I, we used two lemmas which state that the list from 0 to 0 is empty and thus has length zero:

Lemma 8.5 ([18]) $\mathit{len}\,(0,0) \;=\; 0.$

Proof.

$$
\begin{aligned}
&\mathit{len}\,(0,0)\\
&= \;[\![\ \text{Definition of } \mathit{len}\,(0,0)\ ]\!]\\
&\quad \underbrace{[0 \neq 0]}_{=\,0} \cdot \sup_{v \in \mathbb{Z}} [0 \mapsto v] \star ([\mathit{sll}\,(v,0)] + \mathit{len}\,(v,0))\\
&= \;[\![\ \text{elementary algebra}\ ]\!]\\
&\quad 0.
\end{aligned}
$$

□

Lemma 8.6 ([18]) $[\mathit{sll}\,(0,0)] \;=\; [\mathbf{emp}].$

Proof.

$$
\begin{aligned}
&[\mathit{sll}\,(0,0)]\\
&= \;[\![\ \text{Definition of } [\mathit{sll}\,(0,0)]\ ]\!]\\
&\quad \underbrace{[0 = 0]}_{=\,1} \cdot [\mathbf{emp}] + \underbrace{[0 \neq 0]}_{=\,0} \cdot \sup_{v \in \mathbb{Z}} [0 \mapsto v] \star [\mathit{sll}\,(v,0)]\\
&= \;[\![\ \text{elementary algebra}\ ]\!]\\
&\quad [\mathbf{emp}].
\end{aligned}
$$

□

It remains to verify our proposed invariant.

Proof (of Lemma 8.4). By Theorem 7.7, it suffices to prove for the characteristic function $\mathfrak{W}$ of the above loop with respect to postexpectation $\mathit{len}\,(\mathsf{rv},0)$ that $\mathfrak{W}(\mathrm{I}) \preceq \mathrm{I}$ holds. To this end, consider the following:

$$
\begin{aligned}
&\mathfrak{W}(\mathrm{I})\\
&= \;[\![\ \text{Definition of characteristic function } \mathfrak{W}(\mathrm{I})\ ]\!]\\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv},0) + [\mathsf{hd} \neq 0] \cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I})\\
&= \;[\![\ \text{Computation of } \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I}) \text{ in Figure 8.3, page 243}\ ]\!]\\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv},0) + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \big([\mathsf{hd} \mapsto v] \mathrel{-\!\!\star} \big(
\end{aligned}
$$

$$
\begin{aligned}
&\quad 1/2 \cdot [\mathsf{hd} \mapsto -] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad +1/2 \cdot [\mathsf{hd} \mapsto -] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,)\,) \\
=\ & [\![\ \text{D.4.5}\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \hookrightarrow v] \cdot (\\
&\quad 1/2 \cdot [\mathsf{hd} \mapsto -] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad + 1/2 \cdot [\mathsf{hd} \mapsto -] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
=\ & [\![\ \text{elementary algebra (factor out } 1/2)\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \hookrightarrow v] \cdot (\\
&\quad [\mathsf{hd} \mapsto -] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad + [\mathsf{hd} \mapsto -] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
=\ & [\![\ \text{D.3.2}\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \hookrightarrow v] \cdot [\mathsf{hd} \mapsto -] \star (\\
&\quad [\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v] + \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
=\ & [\![\ \text{D.7.8}\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star (\\
&\quad [\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v] + \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
=\ & [\![\ \text{D.3.2}\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} \\
&\quad [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad + [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v] \\
\preceq\ & [\![\ \text{D.6.12}\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot (\\
&\quad \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad + \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
=\ & [\![\ \text{elementary algebra (notice that } [P] = [P] \cdot [P])\ ]\!] \\
&[\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot (
\end{aligned}
$$

$$
\begin{aligned}
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\quad + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]\,) \\
&\preceq \quad [\![\text{Lemma 8.7; Lemma 8.8}]\!] \\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot (\\
&\qquad \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\qquad\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) + [\mathit{sll}\,(\mathsf{hd}, 0)]) \\
&\qquad + \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\qquad\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) - [\mathit{sll}\,(\mathsf{hd}, 0)])) \\
&= \quad [\![\text{elementary algebra; D.3.2}]\!] \\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) \;+\; 1/2 \cdot [\mathsf{hd} \neq 0] \cdot (\\
&\qquad 2 \cdot \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \;+\; 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (2 \cdot \mathit{len}\,(\mathsf{hd}, 0))) \\
&= \quad [\![\text{D.1.2; elementary algebra}]\!] \\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) \star [\mathbf{emp}] \;+\; [\mathsf{hd} \neq 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\qquad + \; 1/2 \cdot [\mathsf{hd} \neq 0] \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \mathit{len}\,(\mathsf{hd}, 0) \\
&= \quad [\![\text{Lemma 8.6; D.1.3}]\!] \\
&\quad [\mathsf{hd} = 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \;+\; [\mathsf{hd} \neq 0] \cdot \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\qquad + \; 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \mathit{len}\,(\mathsf{hd}, 0) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
&= \quad [\![\text{D.6.4}]\!] \\
&\quad \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \mathit{len}\,(\mathsf{hd}, 0) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
&= \quad [\![\text{Definition of I}]\!] \\
&\quad \mathrm{I}. \qquad \square
\end{aligned}
$$

To complete this section, let us provide the missing quantitative entailments that we applied during verification of our invariant.

Lemma 8.7 ([18])

$$
\begin{aligned}
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\
&\preceq \; \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) + [\mathit{sll}\,(\mathsf{hd}, 0)]).
\end{aligned}
$$

Proof. See Appendix F.1.1, page 403. □

Lemma 8.8 ([18])

$$[\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v]$$
$$\preceq \ \boldsymbol{len}\,(\mathsf{rv}, 0) \star [\boldsymbol{sll}\,(\mathsf{hd}, 0)] \ + \ 1/2 \cdot [\boldsymbol{sll}\,(\mathsf{rv}, 0)] \star (\boldsymbol{len}\,(\mathsf{hd}, 0) - [\boldsymbol{sll}\,(\mathsf{hd}, 0)]).$$

Proof. See Appendix F.1.2, page 406. □

8.3 Faulty Garbage Collector

Recall from Example 4.34 the procedure delete which takes a binary tree with root x as an input and—if x is not a null pointer—recursively deletes all elements in the tree. In this case study, we consider a *faulty* variant of this garbage collection procedure. That is, with some positive probability p, we assume that the test $x \neq 0$ evaluates to false although x is not a null pointer. This scenario is modeled by the $\mathbf{P^4L}$ program in Figure 8.4: In contrast to the original garbage collection procedure, we introduced a fresh variable fail that evaluates to true with probability p. Moreover, we require that both $x \neq 0$ and $\mathsf{fail} = \mathsf{false}$ hold before we delete the remaining tree with root x.

In this case study, our goal is to establish a *lower* bound on the probability that the faulty procedure delete successfully deletes the whole tree. More precisely, we approximate from below the weakest *liberal* preexpectation of the procedure call $\mathsf{void} := \mathrm{delete}(x)$ with respect to postexpectation

$$\mathrm{X} \ \triangleq \ [\mathbf{emp}]\,.$$

To this end, we claim that

$$\mathsf{wlp}\,[\mathsf{void} := \mathrm{delete}(x)]\,([\mathbf{emp}]) \ \succeq \ [\boldsymbol{tree}\,(x)] \cdot (1-p)^{1/2 \cdot \mathbf{size}},$$

where $[\boldsymbol{tree}\,(u)]$ is a recursive predicate specifying binary trees with root u, i.e.,

$$[\boldsymbol{tree}\,(u)] \ \triangleq \ [u = 0] \cdot [\mathbf{emp}] \ + \ \sup_{v,w \in \mathbb{Z}} [u \mapsto v, w] \star [\boldsymbol{tree}\,(v)] \star [\boldsymbol{tree}\,(w)]\,.$$

Moreover, for every probability $q \in [0,1]$ and expectation $\mathrm{Y} \in \mathbb{E}$, we define the one-bounded expectation

$$q^{\mathrm{Y}} \ \triangleq \ \lambda(\mathfrak{s}, \mathfrak{h}).\, q^{\mathrm{Y}(\mathfrak{s},\mathfrak{h})}.$$

Intuitively, our claim then states that *whenever x is initially the root of a binary tree with n nodes—the probability that procedure* delete *successfully deletes the whole tree*

```
// sup_{v∈ℤ} [hd ↦ v] ⋆ ([hd ↦ v] −⋆ (
//       1/2 · [hd ↦ −] ⋆ ([hd ↦ rv] −⋆ I [hd/x] [rv/hd] [x/v])
//       +1/2 · [hd ↦ −] ⋆ I [hd/x] [x/v])
x := hd.n;
// 1/2 · [hd ↦ −] ⋆ ([hd ↦ rv] −⋆ I [hd/x] [rv/hd])
//   +1/2 · [hd ↦ −] ⋆ I [hd/x]
{
    // [hd ↦ −] ⋆ ([hd ↦ rv] −⋆ I [hd/x] [rv/hd])
    hd.n := rv;
    // I [hd/x] [rv/hd]
    rv := hd
    // I [hd/x]
} [1/2] {
    // [hd ↦ −] ⋆ I [hd/x]
    free(hd)
    // I [hd/x]
};
// I[hd/x]
hd := x
// I
```

Figure 8.3: Computation of $\mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I})$.

```
delete(x) {
    fail :≈ p · ⟨true⟩ + (1 − p) · ⟨false⟩ ;
    if ( x ≠ 0 and fail = false ) {
        left := <x> ;
        right := <x + 1> ;
        void := delete(left) ;
        void := delete(right) ;
        free(x) ;
        free(x + 1)
    } else {
        skip
    } ;
    out := 0
}
```

Figure 8.4: A $\mathbf{P^4L}$ procedure modeling a faulty garbage collector.

is at least $(1-p)^n$. Notice that our model of binary trees uses two memory cells for each node. Hence, the number of nodes coincides with half the number of addresses allocated on the heap, i.e., $1/2 \cdot \mathbf{size}$.

In order to formally verify our claim, we propose the following higher-order invariant in the sense of Theorem 7.23:

$$\rho \;\triangleq\; \lambda y \lambda E \lambda \mathrm{Y}.\; \begin{cases} [\textit{tree}\,(E)] \cdot (1-p)^{1/2 \cdot \mathbf{size}}, & \text{if } y = \mathsf{void} \text{ and } \mathrm{Y} = [\mathbf{emp}] \\ \mathsf{gfp}\,(\mathfrak{LP}_{\mathrm{delete}})\,(y)(E)(\mathrm{Y}), & \text{otherwise.} \end{cases}$$

Two cases arise: First, we assume that our claim holds for all calls of the form $\mathsf{void} := \mathrm{delete}(E)$. Second, since we never encounter calls with respect to postexpectations different from $[\mathbf{emp}]$, we assign the exact semantics of procedure calls to all other cases. That is, as discussed in Section 7.5.2, our invariant evaluates to $\mathsf{gfp}\,(\mathfrak{LP}_{\mathrm{delete}})\,(y)(E)(\mathrm{Y})$, where the liberal characteristic function $\mathfrak{LP}_{\mathrm{delete}}$ of procedure delete is given by:

$$\mathfrak{LP}_{\mathrm{delete}} \;\triangleq\; \lambda\theta\lambda y\lambda E\lambda \mathrm{Y}.\; \mathsf{wlp}_{\theta}^{\mathrm{delete}}\,[\mathsf{body}\,(\mathrm{delete})]\,\big(\mathrm{Y}\,[-\boxdot]\,[\boxdot y/\mathsf{out}]\big)\,[x/E\,[-\boxdot]]\,[+\boxdot]\,.$$

By Theorem 7.23, it then suffices to show that $\mathfrak{LP}_{\mathsf{delete}}(\rho) \succeq \rho$ to conclude that

$$\mathsf{wlp}\,[\mathsf{void} := \mathsf{delete}(x)]\,([\mathbf{emp}]) \;\succeq\; [\mathit{tree}\,(x)] \cdot (1-p)^{1/2\cdot\mathbf{size}}.$$

Figures 8.5 and 8.6 on pages 247 and 248 depict a proof that $\mathfrak{LP}_{\mathsf{delete}}(\rho) \succeq \rho$ holds for the first case in the definition of ρ, i.e., for all procedure calls of the form $\mathsf{void} := \mathsf{delete}(E)$ and postexpectation $[\mathbf{emp}]$. We briefly comment on the involved lemmas at the end of this section. Furthermore, $\mathfrak{LP}_{\mathsf{delete}}(\rho) \succeq \rho$ holds trivially for the second case in which ρ is the greatest fixed point of $\mathfrak{LP}_{\mathsf{delete}}$.

Hence, our claim indeed yields a lower bound on the probability that procedure delete successfully deletes a binary tree.

Notice that establishing an *upper* bound is more involved because procedure delete is capable of successfully deleting more general graph structures than trees with some positive probability. In particular, cyclic structures can be deleted without encountering a memory fault if a coin flip sets fail to true whenever a pointer has already been disposed in a previous invocation of delete.

Quantitative Entailments Our proof in Figures 8.5 and 8.6 relies on the following two quantitative entailments that are proven separately:

Lemma 8.9 ([18])

$$\begin{aligned} & p \cdot [\mathbf{emp}] + (1-p) \cdot \big([x \neq 0] \cdot (1-p)^{1/2\cdot\mathbf{size}-1} \cdot \\ & \qquad \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] + [x = 0] \cdot [\mathbf{emp}]\big) \\ \succeq\; & (1-p)^{1/2\cdot\mathbf{size}} \cdot [\mathit{tree}\,(x)]\,. \end{aligned}$$

Proof. See Appendix F.2.1, page 408. □

Lemma 8.10 ([18])

$$\begin{aligned} & \sup_{u\in\mathbb{Z}} [x \mapsto u] \star \big([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star \big([x+1 \mapsto v] \mathrel{-\!\!\star} \\ & \quad [x \mapsto -] \star [x+1 \mapsto -] \star \\ & \qquad \big([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}\big) \star \big([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot\mathbf{size}}\big)\big)\big) \\ = \; & (1-p)^{1/2\cdot\mathbf{size}-1} \cdot \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)]\,. \end{aligned}$$

Proof. See Appendix F.2.2, page 409. □

Moreover, to verify these inequalities, we added three rules to our proof system in order to reason about the novel expectation $q^{\mathrm{Y}} = \lambda(\mathfrak{s},\mathfrak{h}).\, q^{\mathrm{Y}(\mathfrak{s},\mathfrak{h})}$:

Lemma 8.11 ([18]) For all $v \in \mathbb{R}_{\geq 0}$, $p \in [0,1]$, and expectations $\mathrm{Y}, \mathrm{Z} \in \mathbb{E}_{\leq 1}$,

$$(\mathrm{Y} \star \mathrm{Z}) \cdot p^{v \cdot \mathbf{size}} \;=\; \left(\mathrm{Y} \cdot p^{v \cdot \mathbf{size}}\right) \star \left(\mathrm{Z} \cdot p^{v \cdot \mathbf{size}}\right).$$

Proof. See Appendix F.2.3, page 411. □

Lemma 8.12 ([18]) For all $v \in \mathbb{R}_{\geq 0}$ and $p \in [0,1]$, we have

$$[E \mapsto E_1, \ldots, E_n] \cdot p^{v \cdot \mathbf{size}} \;=\; [E \mapsto E_1, \ldots, E_n] \cdot p^{v \cdot n}.$$

Proof. See Appendix F.2.4, page 412. □

Lemma 8.13 ([18]) For all $p \in [0,1]$, we have $[\mathbf{emp}] = [\mathbf{emp}] \cdot p^{\mathbf{size}}$.

Proof. See Appendix F.2.5, page 412. □

8.4 Array Randomization

A common approach for designing randomized algorithms is to first compute a random permutation of the input and then apply a deterministic algorithm to solve the actual problem. This approach relies on algorithms that produce every possible permutation of an input with the same probability. In this section, we verify that an algorithm for computing random permutations of arrays satisfies this property. This algorithm is taken from [Cor+09, Chapter 5.3] and has been briefly considered in Example 5.2.

More precisely, our goal is to prove that the $\mathbf{P^4L}$ procedure randomize depicted in Figure 8.7 computes a uniform distribution over all permutations of its input. In other words, it produces every permutation of the provided input array of length n with probability at most $1/n!$, where $n!$ denotes the factorial of n. Since there are exactly $n!$ possible permutations of an array consisting of n elements, this means that each permutation is produced with probability $1/n!$.

How do we verify this property with our weakest preexpectation calculus? We first introduce some convenient notation to describe arrays whose length is determined by an expression rather than a constant. Similar to notation for sums, e.g., $\sum_{k=1}^{n} a_k$, the *iterated separating conjunction* [Rey02] is defined as:

$$\bigstar_{k=E}^{E'} \mathrm{X}_k \;\triangleq\; \lambda(\mathfrak{s},\mathfrak{h}).\; \begin{cases} \left(\mathrm{X}_u \star \mathrm{X}_{u+1} \star \ldots \star \mathrm{X}_v\right)(\mathfrak{s},\mathfrak{h}), & \text{if } u = E\,(\mathfrak{s}) \leq E'\,(\mathfrak{s}) = v \\ [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}), & \text{otherwise.} \end{cases}$$

```
//  (1 − p)^{1/2·size} · [tree(E)]
delete(x) {  //  enter scope & set parameters
    //  (1 − p)^{1/2·size} · [tree(x)]
    //   ⪯   ⟦ Lemma 8.9 ⟧
    //  p · [emp] + (1 − p) · ( [x ≠ 0] · (1 − p)^{1/2·size−1} ·
    //        sup_{u,v∈ℤ} [x ↦ u, v] ⋆ [tree(u)] ⋆ [tree(v)] + [x = 0] · [emp])
    fail :≈ p · ⟨true⟩ + (1 − p) · ⟨false⟩ ;
    //  [x ≠ 0 ∧ fail = false] · (1 − p)^{1/2·size−1} ·
    //        sup_{u,v∈ℤ} [x ↦ u, v] ⋆ [tree(u)] ⋆ [tree(v)]
    //    + [¬(x ≠ 0 ∧ fail = false)] · [emp]
    if ( x ≠ 0 and fail = false ) {
        //  (1 − p)^{1/2·size−1} · sup_{u,v∈ℤ} [x ↦ u, v] ⋆ [tree(u)] ⋆ [tree(v)]
        //   ⪯   ⟦ Figure 8.6 ⟧
        //  wlp [...] ( [emp] )
        left := <x> ; right := <x + 1> ;
        void := delete(left) ; void := delete(right) ;
        free(x) ; free(x + 1)
        //  [emp]
    } else {
        //  [emp]
        skip
        //  [emp]
    } ;
    //  [emp]
    out := 0
    //  [emp]
} //  set return value & leave scope
//  [emp]  =  X
```

Figure 8.5: Proof of $\mathfrak{LP}_{\text{delete}}(\rho)(\text{void})(E)([\mathbf{emp}]) \succeq \rho(\text{void})(E)([\mathbf{emp}])$.

$// \quad (1-p)^{1/2\cdot\mathbf{size}-1} \cdot \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)]$

$// \quad \preceq \quad$ ⟦ Lemma 8.10 ⟧

$// \quad \sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star ([x+1 \mapsto v] \mathrel{-\!\!\star}$

$// \qquad [x \mapsto -] \star [x+1 \mapsto -] \star ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star$

$// \qquad\quad ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot\mathbf{size}})))$

left := <x>;

$// \quad \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star ([x+1 \mapsto v] \mathrel{-\!\!\star} [x \mapsto -] \star [x+1 \mapsto -] \star$

$// \qquad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star ([\mathit{tree}\,(\mathsf{left})] \cdot (1-p)^{1/2\cdot\mathbf{size}}))$

right := <$x+1$>;

$// \quad [x \mapsto -] \star [x+1 \mapsto -] \star ([\mathit{tree}\,(\mathsf{right})] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star$

$// \qquad ([\mathit{tree}\,(\mathsf{left})] \cdot (1-p)^{1/2\cdot\mathbf{size}})$

$// \quad \preceq \quad$ ⟦ apply invariant ρ ⟧

$// \quad [x \mapsto -] \star [x+1 \mapsto -] \star ([\mathit{tree}\,(\mathsf{right})] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star$

$// \qquad \mathsf{wlp}\,[\mathsf{void} := \mathsf{delete}(\mathsf{right})]\,([\mathbf{emp}])$

$// \quad \preceq \quad$ ⟦ D.1.2; Theorem 7.24 (frame rule) ⟧

$// \quad \mathsf{wlp}\,[\mathsf{void} := \mathsf{delete}(\mathsf{left})]\,(\ldots)$

void := delete(left);

$// \quad [x \mapsto -] \star [x+1 \mapsto -] \star ([\mathit{tree}\,(\mathsf{right})] \cdot (1-p)^{1/2\cdot\mathbf{size}})$

$// \quad \preceq \quad$ ⟦ apply invariant ρ ⟧

$// \quad [x \mapsto -] \star [x+1 \mapsto -] \star \mathsf{wlp}\,[\mathsf{void} := \mathsf{delete}(\mathsf{right})]\,([\mathbf{emp}])$

$// \quad \preceq \quad$ ⟦ D.1.3; Theorem 7.24 (frame rule) ⟧

$// \quad \mathsf{wlp}\,[\mathsf{void} := \mathsf{delete}(\mathsf{right})]\,([x \mapsto -] \star [x+1 \mapsto -] \star [\mathbf{emp}])$

void := delete(right);

$// \quad [x \mapsto -] \star [x+1 \mapsto -] \star [\mathbf{emp}]$

`free`(x);

$// \quad [x+1 \mapsto -] \star [\mathbf{emp}]$

`free`$(x+1)$

$// \quad [\mathbf{emp}]$

Figure 8.6: Proof for the branch $x \neq 0$ and $\mathsf{fail} = \mathsf{false}$.

```
                randomize(array, n) {                    swap(array, i, j) {
                    i := 0;                                  y := array[i];
                    while (0 ≤ i < n) {                      z := array[j];
C_body:                 j :≈ uniform(i, n − 1);              array[i] := z;
                        void := swap(array, i, j);           array[j] := y;
                        i := i + 1                           out := 0
                    };                                   }
                    out := 0
                }
```

Figure 8.7: A $\mathbf{P^4L}$ program that computes random array permutations.

With this notation at hand, we choose a postexpectation X that specifies an arbitrary, but fixed, permutation of the input array, i.e.,

$$\mathrm{X} \triangleq \bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m],$$

where $a_0, a_1, a_2, \ldots$ is a sequence of fixed integers. The weakest preexpectation

$$\mathsf{wp}\,[\mathsf{void} := \mathrm{randomize}(\mathsf{array}, n)]\,(\mathrm{X})$$

then maps every stack-heap pair to the probability that procedure randomize produces the a priori fixed permutation described by X upon termination.

Before we proceed, some notation is needed. A *permutation* π over a finite set S is a bijective function $\pi\colon S \to S$. Given two integers u and v, we denote by $\mathsf{Perm}(u, v)$ the set of all permutations over the set $\{u, u+1, \ldots, v\}$. In particular, $\mathsf{Perm}(u, v)$ is the empty set if $u > v$. Analogously, we define the set of permutations determined by expressions E and E' as follows:

$$\mathsf{Perm}(E, E') \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\, \mathsf{Perm}(E(\mathfrak{s}), E'(\mathfrak{s})).$$

Now, to compute an upper bound on the desired weakest preexpectation, we propose the following invariant to deal with the loop in procedure randomize:

$$\mathrm{I} \triangleq [\neg(0 \le i < n)] \cdot \bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m] + \frac{[0 \le i < n]}{(n-i)!} \cdot \sum_{\pi \in \mathsf{Perm}(i,n-1)} \underbrace{\bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right]}_{\triangleq\, \mathrm{Z}^i_\pi}$$

Intuitively, due to backward reasoning, I measures the probability of producing the fixed permutation captured by postexpectation X for *i remaining* loop iterations: All but the first i array elements have already been swapped consistently with X. In our preexpectation, these $n - i$ array elements are thus arbitrarily permuted. Moreover, the probability of hitting exactly the permutation consistent with postexpectation X is $1/(n-i)!$. Since the remaining i iterations still have to be executed, the first i array elements coincide with our postexpectation.

Lemma 8.14 ([18])

$$\mathsf{wp}\left[\texttt{while}\,(\,0 \le i < n\,)\,\{\,\mathsf{C}_{\mathsf{body}}\,\}\right]\left(\bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m]\right) \preceq \mathrm{I}.$$

Proof. A detailed proof that I is an invariant in the sense of Theorem 7.7 is found in Appendix F.3. This includes an analysis of the procedure swap. □

The remaining computation of $\mathsf{wp}\,[\mathsf{void} := \mathrm{randomize}(\mathsf{array}, n)]\,(\mathrm{X})$ is found in Figure 8.8, page 253. Two aspects deserve special attention:

First, the resulting upper bound, i.e.,

$$[0 \le n] \cdot \frac{1}{n!} \cdot \sum_{\pi \in \mathsf{Perm}(0,n-1)} \bigstar_{k=0}^{n-1} \left[\mathsf{array} + k \mapsto a_{\pi(k)}\right],$$

confirms that the probability of satisfying postexpectation X is at most $1/n!$ if the input array is an arbitrary permutation of the fixed array specified by X. Otherwise, it is 0. Hence, *procedure* randomize *indeed computes a uniform distributions over all permutations of the input array.*

Second, we cannot immediately apply our loop invariant, because the variables array and n are out of scope. To address this issue, we perform a case distinction on whether $\boxdot\mathsf{array}$ and array as well as $\boxdot n$ and n coincide or not. We then apply linearity of weakest preexpectations to consider both cases separately. If both variables coincide with their unscoped versions, our invariant is applicable. Otherwise, we show that the remaining weakest preexpectation vanishes as we continue with the proof in Figure 8.8. Formally:

Lemma 8.15 ([18]) Let $C \triangleq \texttt{while}\,(\,0 \leq i < n\,)\,\{\,C_{\mathsf{body}}\,\}$. Moreover, let

$$\mathrm{Y} \;\triangleq\; \mathsf{wp}\,[C]\left(\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right).$$

Then:

$$\mathrm{Y} \;\preceq\; [\boxdot\mathsf{array}=\mathsf{array} \wedge \boxdot n = n]\cdot \mathrm{I} \;+\; [\neg(\boxdot\mathsf{array}=\mathsf{array} \wedge \boxdot n = n)]\cdot \mathrm{Y}.$$

Proof. Let $P \triangleq [\boxdot\mathsf{array}=\mathsf{array} \wedge \boxdot n = n]$. Then, consider the following:

$$\begin{aligned}
&\mathrm{Y}\\
=\;& [\![\text{ Definition of Y }]\!]\\
&\mathsf{wp}\,[C]\left(\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
=\;& [\![\text{ D.6.4; D.6.2 }]\!]\\
&\mathsf{wp}\,[C]\left([P]\cdot[P]\cdot\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right.\\
&\qquad\left.+\,[\neg P]\cdot\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
=\;& [\![\text{ Theorem 7.4 (f) }]\!]\\
&\mathsf{wp}\,[C]\left([P]\cdot[P]\cdot\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
&+\mathsf{wp}\,[C]\left([\neg P]\cdot\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
=\;& [\![\text{ Theorem 7.11 }]\!]\\
&[P]\cdot\mathsf{wp}\,[C]\left([P]\cdot\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
&+\,[\neg P]\cdot\mathsf{wp}\,[C]\left(\bigstar_{m=0}^{\boxdot n-1}\,[\boxdot\mathsf{array}+m \mapsto a_m]\right)\\
\preceq\;& [\![\text{ Theorem 7.4 (a); elementary algebra; D.6.1 }]\!]\\
&[P]\cdot\mathsf{wp}\,[C]\left(\bigstar_{m=0}^{n-1}\,[\mathsf{array}+m \mapsto a_m]\right)
\end{aligned}$$

$$
\begin{aligned}
&\quad + [\neg P] \cdot \mathsf{wp}\,[C] \left(\bigstar_{m=0}^{\boxdot n-1} [\boxdot\mathsf{array} + m \mapsto a_m] \right) \\
&\preceq \quad [\![\text{ Lemma 8.14 }]\!] \\
&\quad [P] \cdot \mathrm{I} + [\neg P] \cdot \mathsf{wp}\,[C] \left(\bigstar_{m=0}^{\boxdot n-1} [\boxdot\mathsf{array} + m \mapsto a_m] \right) \\
&= \quad [\![\text{ Definition of } P \text{ and } \mathrm{Y}]\!] \\
&\quad [\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n] \cdot \mathrm{I} \\
&\qquad + \ [\neg(\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n)] \cdot \mathrm{Y}. \qquad \square
\end{aligned}
$$

8.5 Randomized Meldable Heaps

Meldable heaps are priority queues with a dedicated operation `meld` to combine two disjoint queues into a single one [MT90]. Their implementation is usually based on max-heaps, i.e. binary trees in which the priority of every node is greater than or equal to all of its children.

Gambin and Malinowski [GM98] proposed an efficient *randomized* implementation of meldable heaps: While many deterministic implementations, such as [ST86; Dri+88; Knu98], require additional bookkeeping such that the underlying trees remain somewhat balanced for efficiency reasons, randomized meldable heaps work on standard binary trees; they are thus simple to implement and space efficient. Moreover, the worst-case execution time of each individual operation on a randomized meldable heap is—with high probability—at most logarithmic in the size of the underlying tree.

Various operations on randomized meldable heaps are expressed in terms of the `meld` operation which is performed along a randomly chosen path, say π, from the root to some leaf. Consequently, the complexity of `meld` depends on the *expected length* of π. In fact, the central theorem of Gambin and Malinowski states that *the expected length of a randomly chosen path π is at most logarithmic in the total number of nodes of the underlying tree [GM98, Theorem 1].*

Let us apply QSL to verify their result.[1] To this end, consider the recursive procedure rleaf(root) depicted in Figure 8.9 which models a random walk through a binary tree with root root. Procedure rleaf first determines the left child l and the right child r of root. If both children are equal to 0, then we have hit a leaf and return it. Otherwise, we flip a fair coin to decide whether we continue our walk on a subtree with root l or r, respectively. For simplicity, we assume that the initial tree is non-empty and that every node has exactly two children. Consequently, we omitted conditionals that check whether root $= 0$

[1] The same result has been previously verified with QSL in [Arn19]. However, the proof uses a different invariant which considers the whole tree in each step.

$$\text{// } \quad [0 \le n] \cdot \frac{1}{n!} \cdot \sum_{\pi \in \mathsf{Perm}(0,n-1)} \overset{n-1}{\underset{k=0}{\bigstar}} \Big[\mathsf{array} + k \mapsto a_{\pi(k)}\Big]$$

// $\succeq$ ⟦ elementary algebra; D.1.2 ⟧

$$\text{// } \quad [\neg(0 \le 0 < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] + \frac{[0 \le 0 < n]}{(n-0)!} \cdot$$

$$\text{// } \quad \sum_{\pi \in \mathsf{Perm}(0,n-1)} \overset{0-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \star \overset{n-1}{\underset{m=0}{\bigstar}} \Big[\mathsf{array} + m \mapsto a_{\pi(m)}\Big]$$

// $\succeq$ ⟦ Definition of I; apply substitution; apply scoping ⟧

// $\mathrm{I}\,[i/0]\,[\mathsf{array}/\boxdot\mathsf{array}]\,[n/\boxdot n]\,[+\boxdot]$

// $\succeq$ ⟦ elementary algebra ⟧

// $[\mathsf{array} = \mathsf{array} \wedge n = n] \cdot \mathrm{I}\,[i/0]\,[\mathsf{array}/\boxdot\mathsf{array}]\,[n/\boxdot n]\,[+\boxdot]$

// $+\ [\neg(\mathsf{array} = \mathsf{array} \wedge n = n)] \cdot \mathrm{Y}\,[i/0]\,[\mathsf{array}/\boxdot\mathsf{array}]\,[n/\boxdot n]\,[+\boxdot]$

randomize(array, n) { // enter scope & set parameters

// $[\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n] \cdot \mathrm{I}[i/0] + [\neg(\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n)] \cdot \mathrm{Y}[i/0]$

$i := 0$;

// $[\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n] \cdot \mathrm{I} + [\neg(\boxdot\mathsf{array} = \mathsf{array} \wedge \boxdot n = n)] \cdot \mathrm{Y}$

// $\succeq$ ⟦ Lemma 8.15 ⟧

// $\underbrace{\mathsf{wp}\,[\texttt{while}\,(0 \le i < n)\,\{\,C_{\mathsf{body}}\,\}]\left(\overset{\boxdot n-1}{\underset{m=0}{\bigstar}} [\boxdot\mathsf{array} + m \mapsto a_m]\right)}_{\triangleq\, \mathrm{Y}}$

`while` $(0 \le i < n)$ {

$j :\approx \mathsf{uniform}\,(i, n-1)$; void := swap(array, i, j) ; $i := i+1$

};

// $\overset{\boxdot n-1}{\underset{m=0}{\bigstar}} [\boxdot\mathsf{array} + m \mapsto a_m]$

out := 0

// $\overset{\boxdot n-1}{\underset{m=0}{\bigstar}} [\boxdot\mathsf{array} + m \mapsto a_m]$

} // leave procedure scope & set return value

// $\overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \;=\; \mathrm{X}$

Figure 8.8: Approximation of wp [void := randomize(array, n)] (X).

```
rleaf(root) {
    l := <root> ;
    r := <root + 1> ;
    if (l = 0 and r = 0) {
        out := root
    } else {
        n :≈ 1/2 · ⟨l⟩ + 1/2 · ⟨r⟩ ;
        out := rleaf(n)
    }
}
```

Figure 8.9: A procedure implementing a random walk through a binary tree in which every node has two children.

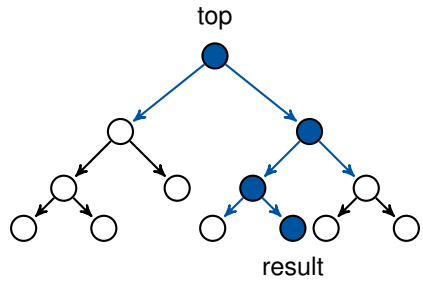


Figure 8.10: Illustration of a heap in which expectation $X = [\textit{tree}\,(\mathsf{top})] \cdot 1 \star ([\textit{path}\,(\mathsf{top}, \mathsf{result})] \cdot 1/2 \cdot \textbf{size})$ evaluates to 3.

holds and that prevent the random assignment from selecting a child equal to 0. If one of the above assumptions is violated, procedure rleaf(root) will encounter a memory fault.

In order to apply weakest preexpectations to reason about the procedure call result := rleaf(top), where variable result stores the returned leaf and top is the root of the whole tree, we have to find a suitable postexpectation $X \in \mathbb{E}$ that measures the length of a path from top to result. The following two recursively defined predicates enable us to concisely formalize this quantity:

$$[\textit{tree}\,(u)] \triangleq [u = 0] \cdot [\textbf{emp}] + \sup_{v,w \in \mathbb{Z}} [u \mapsto v, w] \star [\textit{tree}\,(v)] \star [\textit{tree}\,(w)]\,, \text{ and}$$

$$[\textit{path}\,(u, v)] \triangleq [u = v] \cdot [u \mapsto 0, 0] + \sup_{w,w' \in \mathbb{Z}} [u \mapsto w, w'] \star \max\left\{ [\textit{path}\,(w, v)]\,, [\textit{path}\,(w', v)] \right\}.$$

The first predicate specifies binary trees as introduced in Section 6.6. The second predicate specifies the longest path from node u to node v. Every node consists of two memory cells and a path may choose either of these two cells to determine the next element in a path. In particular, notice that $[\textit{path}\,(u, v)]$ does not admit the heap to contain any memory cells that do not belong to a node on the specified path. Its *length* is given by the expectation $[\textit{path}\,(u, v)] \cdot 1/2 \cdot \textbf{size}$, where

the factor $1/2$ accounts for the fact that every node consists of two memory cells. To describe the length of a path in some larger heap, we use the intuitionistic expectation $1 \star ([\textit{path}\,(u,v)] \cdot 1/2 \cdot \textbf{size})$.

Our postexpectation then measures the length of a path from the root top to the leaf result in a binary tree. Hence, it is defined as

$$\mathrm{X} \;\triangleq\; [\textit{tree}\,(\mathsf{top})] \;\cdot\; 1 \star ([\textit{path}\,(\mathsf{top},\mathsf{result})] \cdot 1/2 \cdot \textbf{size}),$$

where $\star$ takes precendence over $\cdot$ due to Assumption 8.1. Figure 8.10 illustrates the evaluation of X on a stack-heap pair $(\mathfrak{s},\mathfrak{h})$. Clearly, $(\mathfrak{s},\mathfrak{h})$ represents a binary tree with root top; it thus satisfies $[\textit{tree}\,(\mathsf{top})]$. The blue edges highlight the part of the heap captured by $[\textit{path}\,(\mathsf{top},\mathsf{result})]$. The length, i.e. the number of edges, of the unique path from top to result is 3. Hence, $\mathrm{X}(\mathfrak{s},\mathfrak{h}) = 3$.

Our goal is to show that the expected length of a path chosen by procedure rleaf(root) is at most logarithmic in the size of the initially provided tree, i.e.,

$$\mathsf{wp}\,[\mathsf{result} := \mathrm{rleaf}(\mathsf{top})]\,(\mathrm{X}) \;\preceq\; [\mathsf{top} \neq 0] \cdot [\textit{tree}\,(\mathsf{top})] \cdot \log\,(1 + 1/2 \cdot \textbf{size})\,,$$

where we take the logarithm with respect to base two. Recall from Section 7.2.2, that the weakest preexpectation of procedure call result := rleaf(top) with respect to postexpectation X is defined in terms of the least fixed point of procedure rleaf's characteristic function $\mathfrak{P}_{\mathrm{rleaf}}$:

$$\mathsf{wp}\,[\mathsf{result} := \mathrm{rleaf}(\mathsf{top})]\,(\mathrm{X}) \;=\; \mathsf{lfp}\,(\mathfrak{P}_{\mathrm{rleaf}})\,(\mathsf{result})(\mathsf{top})(\mathrm{X}).$$

To verify our desired upper bound, we propose an invariant ρ and then show for all variables result, top $\in$ **Vars** and expectations Z that:

$$\begin{aligned}
&\mathfrak{P}_{\mathrm{rleaf}}(\rho)(\mathsf{result})(\mathsf{top})(\mathrm{Z}) \\
&= \mathsf{wp}_{\rho}^{\mathrm{rleaf}}\,[\mathsf{body}\,(\mathrm{rleaf})]\,(\mathrm{Z}\,[-\boxdot]\,[\boxdot\mathsf{result}/\mathsf{out}])\,[\mathsf{root}/\boxdot\mathsf{top}] \\
&\preceq\; \rho(\mathsf{result})(\mathsf{top})(\mathrm{Z}).
\end{aligned}$$

Our invariant-based proof rule, i.e. Theorem 7.9, then yields

$$\mathsf{wp}\,[\mathsf{result} := \mathrm{rleaf}(\mathsf{top})]\,(\mathrm{X}) \;\preceq\; \rho(\mathsf{result})(\mathsf{top})(\mathrm{Z}).$$

So how do we choose invariant ρ? We consider four distinct cases:

1. For $\mathrm{Z} = \mathrm{Y} \star ([\textit{tree}\,(\mathsf{top})] \cdot 1 \star ([\textit{path}\,(\mathsf{top},\mathsf{result})] \cdot 1/2 \cdot \textbf{size}))$, where Y is a precise expectation satisfying $\mathsf{result} \notin \textbf{Vars}(\mathrm{Y})$, we define

   $$\rho(\mathsf{result})(\mathsf{top})(\mathrm{Z}) \quad\triangleq\quad \mathrm{Y} \star ([\mathsf{top} \neq 0] \cdot [\textit{tree}\,(\mathsf{top})] \cdot \log\,(1 + 1/2 \cdot \textbf{size})).$$

   In this case, Z generalizes postexpectation X. In contrast to X, it admits additional memory cells that are captured by the precise expectation Y. Intuitively, Y specifies all parts of the initially provided tree that do not belong to the subtree processed by the currently executed recursive call.

2. For $Z = [\mathit{tree}\,(\mathsf{top})] \star ([\mathsf{result} \hookrightarrow -] \cdot \infty)$, we set $\rho(\mathsf{result})(\mathsf{top})(Z) \triangleq 0$. This case is an auxiliary that is applied during the verification of the first case. It states that procedure rleaf never returns an address outside of the binary tree passed via its parameter.

3. For $Z \in \mathbb{E}$ with $\mathsf{result} \notin \mathbf{Vars}(Z)$, we set $\rho(\mathsf{result})(\mathsf{top})(Z) \triangleq Z \cdot [\mathsf{top} \neq 0]$. This case is an auxiliary that is applied during the verification of the first case. It states that procedure rleaf does not change the heap and crashes if $\mathsf{top} = 0$.

4. Since no other possibility for Z is encountered during the verification of the first three cases, we set $\rho(\mathsf{result})(\mathsf{top})(Z)$ to the exact fixed point in all other cases. This trivially satisfies the premise of Theorem 7.9.

It remains to verify that ρ is an invariant in the sense of Theorem 7.9: A proof of the first case is provided in Figure 8.11. Moreover, we present a detailed treatment of the recursive call and the probabilistic assignment in Figure 8.12. This requires a few case distinctions on the values of local variables n, l, and r. The most relevant case, i.e. $\mathsf{n} = \mathsf{l}$, is analyzed in Figure 8.13. The case $\mathsf{n} = \mathsf{r}$ is symmetric. Detailed calculations of all involved quantitative entailments are referenced within the proofs and are found in Appendix F.4. The same holds for proofs of the remaining cases and a verification of the second and third case of our proposed invariant ρ. With invariant ρ at hand, we then conclude that

$$
\begin{aligned}
& \mathsf{wp}\,[\mathsf{result} := \mathrm{rleaf}(\mathsf{top})]\,(X) \\
= \;& [\![\text{ Definition of } X \text{ }]\!] \\
& \mathsf{wp}\,[\mathsf{result} := \mathrm{rleaf}(\mathsf{top})]\,([\mathit{tree}\,(\mathsf{top})] \cdot 1 \star ([\mathit{path}\,(\mathsf{top}, \mathsf{result})] \cdot 1/2 \cdot \mathbf{size})) \\
\preceq \;& [\![\text{ Theorem 7.9 }]\!] \\
& \rho(\mathsf{result})(\mathsf{top})([\mathit{tree}\,(\mathsf{top})] \cdot 1 \star ([\mathit{path}\,(\mathsf{top}, \mathsf{result})] \cdot 1/2 \cdot \mathbf{size})) \\
= \;& [\![\text{ invariant case (1) with } Y = [\mathbf{emp}] \text{ }]\!] \\
& [\mathsf{top} \neq 0] \cdot [\mathit{tree}\,(\mathsf{top})] \cdot \log\,(1 + 1/2 \cdot \mathbf{size}).
\end{aligned}
$$

Hence, the expected length of a path from root top to the returned leaf result is indeed at most logarithmic in the number of nodes contained in the tree.

```
// Y ⋆ ([top ≠ 0] · [tree (top)] · log (1 + 1/2 · size))
// ⪰ ⟦ apply [Q] · X + [¬Q] · X = X; [top ↪ u, v] ⪯ 1; elementary algebra ⟧
// sup_{u,v∈ℤ} [top ↪ u, v] · ( [u = 0 ∧ v = 0] · Y ⋆ ([top ≠ 0] · [tree (top)] · log (1 + 1/2 · size))
//   + [u ≠ 0 ∨ v ≠ 0] · Y ⋆ ([top ≠ 0] · [tree (top)] · log (1 + 1/2 · size)))
// ⪰ ⟦ Lemma F.15; Lemma F.16 ⟧
// sup_{u,v∈ℤ} [top ↪ u, v] · ( [u = 0 ∧ v = 0] · X_if [r/v] [l/u] [root/▣top] [+▣]
//   + [u ≠ 0 ∨ v ≠ 0] · X_else [r/v] [l/u] [root/▣top] [+▣])
rleaf(root) { // enter scope & set parameters
  // sup_{u,v∈ℤ} [root ↪ u, v] · ( [u = 0 ∧ v = 0] · X_if [r/v] [l/u] + [u ≠ 0 ∨ v ≠ 0] · X_else [r/v] [l/u])
  l := <root>;
  // sup_{v∈ℤ} [root + 1 ↪ v] · ( [l = 0 ∧ v = 0] · X_if [r/v] + [l ≠ 0 ∨ v ≠ 0] · X_else [r/v] )
  r := <root + 1>; // use alternative lookup rule (D.4.5)
  // [l = 0 ∧ r = 0] · X_if + [l ≠ 0 ∨ r ≠ 0] · X_else
  if (l = 0 and r = 0) {
    // Y [−▣] [out/root] ⋆ ([tree (▣top)] · 1 ⋆ ([path (▣top, root)] · 1/2 · size))
    //   ≜ X_if
    out := root
    // Y [−▣] ⋆ ([tree (▣top)] · 1 ⋆ ([path (▣top, out)] · 1/2 · size))
  } else {
    // 1/2 · [l ≠ r] · Y [−▣] [n/l] ⋆ [▣top ↦ l, r] ⋆ [tree (r)]
    //   ⋆ ([l ≠ 0] · [tree (l)] · (1 + log (1 + 1/2 · size)))
    // + 1/2 · [l ≠ r] · Y [−▣] [n/r] ⋆ [▣top ↦ l, r] ⋆ [tree (l)]
    //   ⋆ ([r ≠ 0] · [tree (r)] · (1 + log (1 + 1/2 · size)))
    // + [l = 0 ∧ r = 0] · ∞
    // + [¬ ▣ top ↪ l, r] · wp^rleaf_ρ [n :≈ 1/2 · ... ; out := rleaf(n)] (...)
    //   ≜ X_else
    // ⪰ ⟦ Figure 8.12 ⟧
    // wp^rleaf_ρ [n :≈ 1/2 · ... ; out := rleaf(n)] (...)
    n :≈ 1/2 · ⟨l⟩ + 1/2 · ⟨r⟩;
    out := rleaf(n)
    // Y [−▣] ⋆ ([tree (▣top)] · 1 ⋆ ([path (▣top, out)] · 1/2 · size))
  }
  // Y [−▣] ⋆ ([tree (▣top)] · 1 ⋆ ([path (▣top, out)] · 1/2 · size))
} // set return value & leave scope
// Y ⋆ ([tree (top)] · 1 ⋆ ([path (top, result)] · 1/2 · size))
```

Figure 8.11: Proof of invariant case (1)

$$
\begin{aligned}
&// \quad 1/2 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot Y\,[-\boxdot]\,[\mathsf{n}/\mathsf{l}] \star [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{r})] \\
&// \qquad \star ([\mathsf{l} \neq 0] \cdot [\textit{tree}\,(\mathsf{l})] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&// \quad + 1/2 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot Y\,[-\boxdot]\,[\mathsf{n}/\mathsf{r}] \star [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \\
&// \qquad \star ([\mathsf{r} \neq 0] \cdot [\textit{tree}\,(\mathsf{r})] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&// \quad + 0 \\
&// \quad + [\mathsf{l} = 0 \wedge \mathsf{r} = 0] \cdot \infty \\
&// \quad + [\neg\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(X_1) \\
&// \quad \succeq \quad [\![\ \text{Figures 8.13, F.3 and F.4 + Figure 8.13 (symmetric case) + Theorem 7.12}\ ]\!] \\
&// \quad \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot X_1) \\
&// \quad + \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot X_1) \\
&// \quad + \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot X_1) \\
&// \quad + \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot X_1) \\
&// \quad + \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]([\neg\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot X_1) \\
&// \quad \succeq \quad [\![\ \text{Linearity of weakest preexpectations (Theorem 7.4 (f))}\ ]\!] \\
&// \quad \mathrm{wp}_\rho^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\,\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(X_2) \\
&\mathsf{n} :\approx 1/2 \cdot \langle \mathsf{l} \rangle + 1/2 \cdot \langle \mathsf{r} \rangle\,; \\
&\mathsf{out} := \mathrm{rleaf}(\mathsf{n}) \\
&// \quad [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot X_1 \\
&// \qquad + [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot X_1 \\
&// \qquad + [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot X_1 \\
&// \qquad + [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot X_1 \\
&// \qquad + [\neg\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot X_1 \\
&\qquad\qquad \triangleq X_2 \\
&// \quad \succeq \quad [\![\ \text{Case distinction on } [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}],\ [\mathsf{n} = \mathsf{l}],\ [\mathsf{n} = \mathsf{r}]\ ]\!] \\
&// \quad Y\,[-\boxdot] \star ([\textit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\textit{path}\,(\boxdot\mathsf{top},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \\
&\qquad\qquad \triangleq X_1
\end{aligned}
$$

Figure 8.12: Proof of the `else` branch of invariant case (1)

$$
\begin{array}{ll}
// & 1/2 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot Y\,[-\boxdot]\,[\mathsf{n}/\mathsf{l}] \star [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})] \\
// & \qquad \star([\mathsf{l} \neq 0] \cdot [\mathit{tree}\,(\mathsf{l})] \cdot (1 + \log\,(1 + 1/2 \cdot \mathbf{size}))) \\
// & \succeq \quad [\![\text{ D.3.2; elementary algebra }]\!] \\
// & 1/2 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot Y\,[-\boxdot]\,[\mathsf{n}/\mathsf{l}] \star [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})] \star ([\mathsf{l} \neq 0] \cdot [\mathit{tree}\,(\mathsf{l})]) \\
// & + 1/2 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot Y\,[-\boxdot]\,[\mathsf{n}/\mathsf{l}] \star [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})] \\
// & \qquad \star\,([\mathsf{l} \neq 0] \cdot [\mathit{tree}\,(\mathsf{l})] \cdot \log\,(1 + 1/2 \cdot \mathbf{size})) \\
\mathsf{n} :\approx & \ 1/2 \cdot \langle \mathsf{l} \rangle + 1/2 \cdot \langle \mathsf{r} \rangle\,; \\
// & [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot Y\,[-\boxdot] \star [\boxdot\mathsf{top} \mapsto \mathsf{n},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})] \star ([\mathsf{n} \neq 0] \cdot [\mathit{tree}\,(\mathsf{n})]) \\
// & \ + [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot Y\,[-\boxdot] \star [\boxdot\mathsf{top} \mapsto \mathsf{n},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})] \\
// & \qquad \star\,([\mathsf{n} \neq 0] \cdot [\mathit{tree}\,(\mathsf{n})] \cdot \log\,(1 + 1/2 \cdot \mathbf{size})) \\
// & \ + 0 \\
// & \succeq \quad [\![\text{ Apply invariant } \rho \text{ to each summand }]\!] \\
// & \mathsf{wp}^{\mathsf{rleaf}}_{\rho}\,[\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(Y_1 \star [\mathit{tree}\,(\mathsf{n})]\,) \\
// & \ + \mathsf{wp}^{\mathsf{rleaf}}_{\rho}\,[\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(Y_2 \star ([\mathit{tree}\,(\mathsf{n})] \cdot 1 \star ([\mathit{path}\,(\mathsf{n},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size}))) \\
// & \ + \mathsf{wp}^{\mathsf{rleaf}}_{\rho}\,[\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,([\mathit{tree}\,(\mathsf{n})] \star ([\mathsf{out} \hookrightarrow 0,0] \cdot \infty)) \\
// & \succeq \quad [\![\text{ Linearity of weakest preexpectations (Theorem 7.4 (f)) }]\!] \\
// & \mathsf{wp}^{\mathsf{rleaf}}_{\rho}\,[\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(\ \ldots\ ) \\
\mathsf{out} := & \ \mathrm{rleaf}(\mathsf{n}) \\
// & \underbrace{[\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot Y\,[-\boxdot] \star [\boxdot\mathsf{top} \mapsto \mathsf{n},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})]}_{= Y_1} \star\ [\mathit{tree}\,(\mathsf{n})] \\
// & \ + \underbrace{[\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot Y\,[-\boxdot] \star [\boxdot\mathsf{top} \mapsto \mathsf{n},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{r})]}_{= Y_2} \\
// & \qquad \star([\mathit{tree}\,(\mathsf{n})] \cdot 1 \star ([\mathit{path}\,(\mathsf{n},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \\
// & \ + \underbrace{[\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot [\mathit{tree}\,(\mathsf{l})]}_{\preceq\ [\mathit{tree}(\mathsf{n})]}\ \star([\mathsf{out} \hookrightarrow 0,0] \cdot \infty) \\
// & \succeq \quad [\![\text{ Lemma F.10; elementary algebra } (\mathsf{n} = \mathsf{l};\ \mathsf{l} \neq \mathsf{r})]\!] \\
// & [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot (\\
// & \qquad Y\,[-\boxdot] \star ([\mathit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\mathit{path}\,(\boxdot\mathsf{top},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size})))
\end{array}
$$

Figure 8.13: Proof of the `else` branch of invariant case (1) for $\mathsf{n} = \mathsf{l}$. The case $\mathsf{n} = \mathsf{r}$ is symmetric.

We developed QSL—a quantitative variant of separation logic—which serves as both an assertion language that evaluates to real numbers instead of truth values and a verification system for reasoning about probabilistic pointer programs.

Regarding QSL as an assertion language, expectations replace predicates as the main object of interest. We introduced quantitative analogs to separating conjunction and separating implication and showed that they enjoy virtually the same properties as the classical separating connectives. Moreover, we studied various rules for reasoning about QSL formulas which have been formalized in the theorem prover Isabelle/HOL.[1]

Regarding QSL as a verification system, we developed a weakest preexpectation calculus for reasoning about total correctness of probabilistic pointer programs that conservatively extends both classical weakest preconditions based on separation logic and McIver and Morgan's weakest preexpectation calculus. We proved the soundness of QSL with respect to the operational model introduced in Chapter 5. Our calculus facilitates local reasoning in a similar fashion to separation logic. In particular, all rules only touch parts of the heap actually accessed by a program. Furthermore, the frame rule is preserved by QSL. Finally, we presented a liberal version of QSL that enables reasoning about probabilities in a partial correctness setting. We demonstrated that QSL provides a foundation for formal reasoning about randomized algorithms manipulating dynamic data structures on source code level.

9.1 Future Work

An interesting direction for future research is to extend the expected runtime calculus developed in [10; 6] to reason about probabilistic pointer programs. Let us briefly sketch one possible approach: Since we reason about runtimes, we flip the ordering underlying our domain of expectations. Hence, while the weakest

[1] In cooperation with Max Haslbeck from TU Munich; further details are available online at https://github.com/maxhaslbeck/QuantSepCon.

preexpectation of a nonterminating program is zero, its expected runtime is infinite. Analogously, since a memory failure leads to undefined behavior, programs encountering a memory failure are assigned infinite expected runtime. Instead of the quantitative separating conjunction, which attempts to maximize the expected value of a product over all partitions of the heap, we then interpret the separating conjunction as an *addition*. That is, we define

$$X \star Y \triangleq \lambda(\mathfrak{s}, \mathfrak{h}).\ \inf\{\, X(\mathfrak{s}, \mathfrak{h}_1) + X(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\}.$$

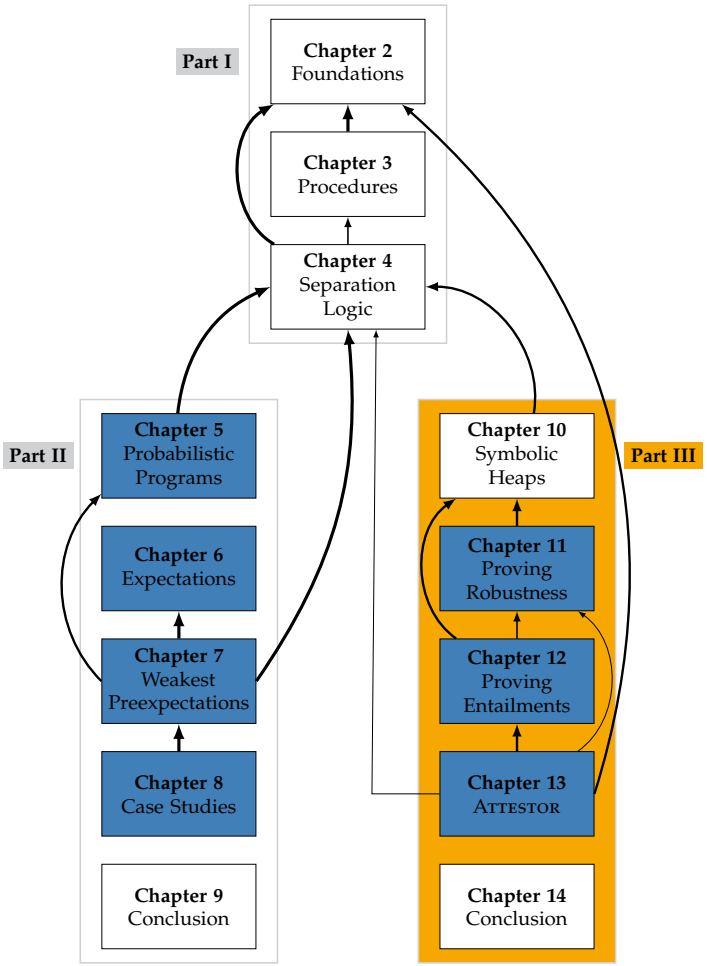
This notion of separating conjunction—provided it leads to a reasonable notion of separation logic—would enable us to reason compositionally about the runtime of randomized divide-and-conquer algorithms, such as randomized quicksort. In particular, a frame rule for the above "separating addition" would support compositional reasoning about expected runtimes in the sense that we only have to consider those parts of the heap on which a program actually operates. The technical details, however, have not been worked out yet.

More generally, the above alternative definition of a quantitative separating conjunction raises another question: What are the minimal requirements that have to be imposed to obtain a sensible calculus that retains the key properties of separation logic, i.e., local reasoning, the frame rule, etc. In the qualitative setting this question has been answered by Calcagno, O'Hearn, and Yang [COY07]. They introduced the notion of separation algebras, i.e., cancellative, partial commutative monoids, and proved that, for any separation algebra, one can derive a separating conjunction between predicates that admits local reasoning for common programming language statements. It would be interesting to derive similar general results in a quantitative setting, i.e., for expectations and probabilistic programs.

Other future work includes developing a richer set of rules for discharging quantitative entailments and investigating *syntactical* fragments of expectations that are suitable for automated reasoning. Moreover, it remains unclear how to reason about potentially unbounded expectations in a partial correctness setting.

Part III

Automated Reasoning



Chapter 10

Towards Automated Reasoning

Throughout Parts I and II, we studied reasoning about programs featuring recursion, pointers, and sampling from probability distributions. Broadly speaking, the presented techniques boil down to either *program analysis* or *Floyd-Hoare style verification*. Let us briefly summarize the essential tasks of both approaches:

- For *program analysis*, we consider properties of program executions. Proving a property involves the overapproximation of the reachable fragment of the operational semantics for a given program and a set of initial states followed by an exhaustive state space exploration.
- For *Floyd-Hoare style verification*, a specification consists of a pre- and a postcondition. Proving it involves the computation of the weakest precondition of a given program with respect to the postcondition. The specification then holds iff the precondition entails the weakest precondition.

At first glance, both approaches largely amount to the mechanical application of inference rules; this suggests that they are amenable to automation. In fact, provided that applying the operational semantics yields a finite transition system and that we can decide for every state which atomic propositions hold, model checking enables us to automatically reason about a large class of properties (cf. [BK08]). Furthermore, provided that we are equipped with suitable invariants, weakest preconditions can be computed automatically.

However, both program analysis and program verification are undecidable in general [Tur37; Ric53; CGR18]. Every attempt to automate formal reasoning about a sufficiently expressive class of programs and specifications is thus either *unsound*, i.e., we might encounter a wrong result, or incomplete, i.e., we may not encounter a result at all. In particular, we face the following challenges:

- The reachable fragment of the operational semantics for a given program and a set of initial states is typically *infinite*; it thus cannot be computed. This may be caused by certain program features, such as loops, recursion, or even memory allocation. Alternatively, we might be interested in an infinite set of initial states. As discussed in Section 2.2, we are thus forced

to apply some kind of *abstraction* to determine a finite overapproximation of the transition system induced by the operational semantics.

- We have to decide the *state-labeling problem*, i.e., does a given state in a transition system satisfy an atomic proposition? When reasoning about heap manipulating programs, sensible atomic propositions include aliasing and reachability conditions, e.g., the heap contains no garbage, or the heap is acyclic. In fact, we might want to use any separation logic formula as an atomic proposition. Without abstraction, solving the state-labeling problem then amounts to solving the *model-checking problem* for separation logic. The situation becomes even more involved in the presence of abstraction because we have to check the same properties for abstract states that capture potentially infinite sets of concrete states.

- We have to decide the *entailment problem*, i.e., the satisfiability of logical implications between predicates. This problem is essential when applying Floyd-Hoare style verification because we have to check whether our precondition entails the previously determined weakest precondition. Furthermore, it is needed to discharge verification conditions which arise when applying invariants. If we use predicates for abstraction, then the entailment problem is also relevant for program analysis. More precisely, we have to solve entailments whenever we check whether an abstract state is already covered by an existing one.

- We have to *synthesize suitable invariants*. For program analysis, we automatically obtain invariants provided that our abstraction is strong enough to yield a finite transition system. However, this shifts the task to finding suitable abstractions. Depending on the desired degree of automation, invariants or abstractions may be supplied by the user or have to be inferred automatically from a given program and the specification.

Goal of Part III & Outline The goal of this part is to *advance automated reasoning about (non-probabilistic) heap manipulating programs with separation logic* by addressing the aforementioned challenges. This involves both reasoning about programs *with* separation logic and reasoning *about* separation logic itself.

More precisely, we focus on the popular symbolic heap fragment of separation logic with user-defined inductive definitions (**SHSL** for short). To pin down **SHSL** in the vast landscape of separation logic fragments investigated in the literature, we first give a brief overview in Section 10.1. The decidability status of various of its fragments is discussed alongside related work in Section 10.2.

In Chapter 11, we consider reasoning *about* separation logic. To this end, we develop a framework based on *heap automata* to decide common *robustness*

properties of **SHSL** [2], e.g., **SHSL**-satisfiability, reachability, etc. Apart from decision procedures, our framework supports debugging of inductive definitions: If an **SHSL** formula is not robust, a counterexample is generated. Furthermore, one can synthesize robust inductive definitions through refinement. Heap automata can also be applied to discharge entailments. We take a closer look at decision procedures for the entailment problem in Chapter 12. In particular, we present a pragmatic decision procedure in NP for a fragment of symbolic heaps with user-supplied inductive definitions.

In Chapter 13, we consider reasoning about programs *with* separation logic. To this end, we describe the implementation of the model checker Attestor [4]. Attestor attempts to check whether a formula in linear temporal logic (cf. [Pnu77; BK08]) with heap-specific atomic propositions holds for an abstract transition system that overapproximates the operational semantics of a given Java program. In particular, abstraction is guided by inductive definitions in **SHSL** and heap automata are applied to deal with the state-labeling problem.

Finally, Chapter 14 concludes and presents future work.

10.1 A Syntax for Separation Logic

In previous chapters, we took the extensional approach to program semantics by admitting any computable predicate as an assertion. While this is convenient for studying weakest precondition calculi, it is inadequate for automated reasoning. Let us thus switch to the intensional approach and fix an explicit syntax for fully-fledged separation logic. We then derive the fragment of symbolic heap separation logic which is considered throughout the remaining chapters. We first introduce a syntax for expressions that may occur in assertions.

Definition 10.1 (Syntax of Expressions) From now on, the syntax of *expressions* E is given by the following context-free grammar:

$$E \to 0 \mid 1 \mid x \mid E + E \mid E \cdot E,$$

where $x \in \mathbf{Vars}$ is a variable.

Furthermore, let us fix some notation for predicates representing recursively defined data structures (Section 4.3.4).

Definition 10.2 (Predicate Symbols) Let **PSym** be a set of *predicate symbols* which is equipped with a function $\mathsf{param} \colon \mathbf{PSym} \to \mathbb{N}$ assigning to each predicate symbol its number of parameters.

A *predicate call* is then a term of the form $\boldsymbol{P}(E_1, \ldots, E_n)$,where $\boldsymbol{P} \in \mathbf{PSym}$, $\mathsf{param}(\boldsymbol{P}) = n$, and $E_1, \ldots, E_n$ are expressions.

We are now in a position to define the syntax of separation logic assertions. Even without predicate symbols, these assertions are expressive enough such that every precondition of an assertion (without predicate symbols) can be written as an assertion again [TC14; TCA19].

Definition 10.3 (Syntax of Separation Logic Assertions) The set of *syntactic separation logic assertions* is given by the following context-free grammar:

$$\begin{aligned}\varphi \ ::= \ & \mathsf{false} \mid \mathsf{true} \mid E = E \mid E \neq E \mid E < E && \text{(pure atoms)}\\ \mid \ & \mathbf{emp} \mid E \mapsto E \mid \boldsymbol{P}(\vec{E}) && \text{(spatial atoms)}\\ \mid \ & \varphi \star \varphi \mid \varphi \mathrel{-\!\!\star} \varphi && \text{(separating connectives)}\\ \mid \ & \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \exists x\colon \varphi \mid \forall x\colon \varphi, && \text{(first-order logic)}\end{aligned}$$

where x is a variable, E is an expression, and $\boldsymbol{P}(\vec{E})$ is a predicate call.

Every syntactic assertion specifies a predicate. We thus adopt most notions introduced for separation logic predicates in Section 4.3. In particular:

- **Vars** (φ) denotes the set of all free variables in assertion φ, i.e., all variables that are not bound by a quantifier.
- $\varphi\,[x/E]$ is the assertion obtained by substituting every free occurrence of variable x in assertion φ by expression E.
- The semantics of a syntactic assertion is determined by the semantics of the corresponding predicate. That is, for every syntactic assertion φ and every stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, we have $\mathfrak{s}, \mathfrak{h} \models \varphi$ iff $(\mathfrak{s}, \mathfrak{h})$ satisfies the predicate given by φ (cf. Definition 4.11).

Formal definitions by structural induction of both the semantics (apart from predicate calls) and the set of free variables of syntactic assertions are found in Figure 10.1. The semantics of predicate calls is identical to the least fixed point semantics considered in Section 4.3.4. We will address the semantics of predicate calls for a fragment of separation logic assertions in Section 10.1.2.

Other notation for expressions and assertions used in previous chapters is definable in our syntax; it is thus considered syntactic sugar. For example:

- For every $n \in \mathbb{N}_{>0}$, we write n rather than $1 + 1 + \ldots + 1$ (n times).
- The points-to assertion $E \mapsto E_1, E_2, \ldots, E_n$ is defined as
$$E \mapsto E_1 \star E + 1 \mapsto E_2 \star \ldots \star E + n - 1 \mapsto E_n.$$
- The contains-pointer assertion $E \hookrightarrow E'$ is a shortcut for $E \mapsto E' \star \mathsf{true}$.

E	$E(\mathfrak{s})$	$\mathbf{Vars}(E)$
0	0	$\emptyset$
1	1	$\emptyset$
x	$\mathfrak{s}(x)$	$\{x\}$
$E_1 + E_2$	$E_1(\mathfrak{s}) + E_2(\mathfrak{s})$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$
$E_1 \cdot E_2$	$E_1(\mathfrak{s}) \cdot E_2(\mathfrak{s})$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$

φ	$\mathfrak{s}, \mathfrak{h} \models \varphi$ iff	$\mathbf{Vars}(\varphi)$
false	never	$\emptyset$
true	always	$\emptyset$
$E_1 = E_2$	$E_1(\mathfrak{s}) = E_2(\mathfrak{s})$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$
$E_1 \neq E_2$	$E_1(\mathfrak{s}) \neq E_2(\mathfrak{s})$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$
$E_1 < E_2$	$E_1(\mathfrak{s}) < E_2(\mathfrak{s})$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$
$\mathbf{emp}$	$\mathsf{dom}(\mathfrak{h}) = \emptyset$	$\emptyset$
$E_1 \mapsto E_2$	$\mathfrak{h} = \{E_1(\mathfrak{s}) :: E_2(\mathfrak{s})\}$	$\mathbf{Vars}(E_1) \cup \mathbf{Vars}(E_2)$
$\boldsymbol{P}(E_1, \ldots, E_n)$	see Section 10.1.2	$\bigcup_{k=1}^{n} \mathbf{Vars}(E_k)$
$\psi \star \vartheta$	$\exists \mathfrak{h}_1, \mathfrak{h}_2 \colon \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2$ and $\mathfrak{s}, \mathfrak{h}_1 \models \psi$ and $\mathfrak{s}, \mathfrak{h}_2 \models \vartheta$	$\mathbf{Vars}(\psi) \cup \mathbf{Vars}(\vartheta)$
$\psi \mathrel{-\!\!\!\star} \vartheta$	$\forall \mathfrak{h}' \colon (\mathfrak{h} \# \mathfrak{h}'$ and $\mathfrak{s}, \mathfrak{h} \models \psi)$ implies $\mathfrak{s}, \mathfrak{h} \uplus \mathfrak{h}' \models \vartheta$	$\mathbf{Vars}(\psi) \cup \mathbf{Vars}(\vartheta)$
$\neg\psi$	not $\mathfrak{s}, \mathfrak{h} \models \psi$	$\mathbf{Vars}(\psi)$
$\psi \wedge \vartheta$	$\mathfrak{s}, \mathfrak{h} \models \psi$ and $\mathfrak{s}, \mathfrak{h} \models \vartheta$	$\mathbf{Vars}(\psi) \cup \mathbf{Vars}(\vartheta)$
$\psi \vee \vartheta$	$\mathfrak{s}, \mathfrak{h} \models \psi$ or $\mathfrak{s}, \mathfrak{h} \models \vartheta$	$\mathbf{Vars}(\psi) \cup \mathbf{Vars}(\vartheta)$
$\exists x \colon \psi$	$\exists v \in \mathbb{Z} \colon \mathfrak{s}[x/v], \mathfrak{h} \models \psi$	$\mathbf{Vars}(\psi) \setminus \{x\}$
$\forall x \colon \psi$	$\forall v \in \mathbb{Z} \colon \mathfrak{s}[x/v], \mathfrak{h} \models \psi$	$\mathbf{Vars}(\psi) \setminus \{x\}$

Figure 10.1: Inductive definitions of both the semantics and the set of free variables of expressions and separation logic assertions.

- Given a finite sequence of selectors $\mathbf{Sel} = \langle sel_0, \ldots, sel_n \rangle$, we denote by $x.sel_i \mapsto E$ the points-to assertion $x + i \mapsto E$.

Assumption 10.4 We assume the reader is familiar with standard results from computability and complexity theory, e.g., (polynomial time) reductions and the complexity classes NP, coNP, and ExpTime. A comprehensive introduction of these topics is found, for example, in [Pap07; AB09].

Furthermore, when reasoning about the complexity of decision procedures, we frequently refer to the *size* of mathematical objects, e.g., assertions, sets, tuples, etc. Rather than explicitly encoding these objects as bitstrings, we assume the presence of a function $\|.\|$ which assigns a sensible notion of size to a given object. For instance, if S is a finite set, $\|S\|$ could be defined as the product of the cardinality of S and the size of its largest element.

Any attempt towards automated reasoning with separation logic requires techniques to discharge one or more decision problems about assertions. Let us collect the most prominent ones. Apart from expressiveness, the presence and complexity of (semi-)decision procedures for these problems serves as a criterion for choosing a fragment of separation logic which is amenable to automation.

Definition 10.5 (Decision Problems for Separation Logic) We define the following decision problems for separation logic assertions:

(a) The *model-checking problem*: Given an assertion φ and a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, does $\mathfrak{s}, \mathfrak{h} \models \varphi$ hold?

(b) The *satisfiability problem*: Given an assertion φ, does there exist a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ such that $\mathfrak{s}, \mathfrak{h} \models \varphi$ holds?

(c) The *entailment problem*: Given assertions φ and ψ, does $\varphi \models \psi$, i.e., for all stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$, we have $\mathfrak{s}, \mathfrak{h} \models \varphi$ implies $\mathfrak{s}, \mathfrak{h} \models \psi$, hold?

For example, as discussed initially, we encounter the model-checking problem as an instance of the state-labeling problem. Moreover, it is encountered in runtime verification and software testing [Bro+16]. The satisfiability problem allows us to check whether a specification or an automatically computed abstract state is sensible. We have already observed throughout previous chapters that the entailment problem is essential for Floyd-Hoare style verification.

Unfortunately, none of the above decision problems is decidable for general separation logic assertions. This is hardly surprising. After all, our syntax for assertions covers all formulas in first-order logic. Undecidability of the satisfiability problem is thus immediate (cf. [Tur37]). Furthermore, by exploiting the separating implication and closure under Boolean operations, undecidability

of the remaining decision problems can be reduced to the undecidability of the satisfiability problem. It is noteworthy that the undecidability of separation logic assertions is more deeply rooted than one might expect. In fact, there is an ongoing quest for identifying smaller and smaller undecidable fragments of separation logic, e.g., by restricting the number of variables in assertions (cf. [DD15; Dem+17]). In particular, Calcagno, Yang, and O'Hearn proved that satisfiability remains undecidable if we only admit contains-pointer assertions (for records of size at least five), equality, false, standard implication, and universal quantifiers [CYO01, Theorem 1]. The same holds for standard contains-pointer assertions $E \hookrightarrow E'$ and at most two variables [DD16, Corollary 5.15]. Similarly, Echenim, Iosif, and Peltier showed that satisfiability is undecidable if we restrict formulas in the above fragment to prenex normal form with an $\exists^*\forall^*$-prefix, but admit the separating connectives [EIP19, Theorem 1]. Finally, Demri, Lozes, and Mansutti showed that combining $\mathrel{-\!\!*}$ with a predicate for singly-linked list segments—and allowing $E = E'$, **emp**, $E \hookrightarrow E'$, and $\star$, but no quantifiers—leads to a separation logic fragment with an undecidable satisfiability problem [DLM18, Theorem 1].

10.1.1 The Symbolic Heap Fragment

Separation logic assertions are in general too expressive to allow for effective decision procedures. However, reasoning about many programs does not require the full expressive power of separation logic. In particular, *symbolic heaps* emerged as an idiomatic form of assertions that occur naturally in hand-written proofs [ORY01; BCO05b; BCO05a]. Intuitively, every concrete heap, say

$$\mathfrak{h} \triangleq \{1 :: 6\} \uplus \{6 :: 0, 17\},$$

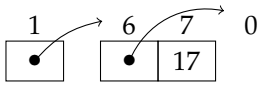
can be characterized by a separation logic assertion using only points-to assertions and separating conjunctions. For example, the formula

$$\varphi \triangleq 1 \mapsto 6 \star 6 \mapsto 0, 17$$

is satisfied iff the heap coincides with $\mathfrak{h}$. When reasoning about programs, we rarely refer to concrete addresses. Rather, memory cells are accessed through variables, e.g., $<x + 1>$ or $x.next$. A *symbolic* heap thus abstracts from concrete addresses by substituting every address by a (not necessarily free) variable or the constant 0 which represents a value that cannot serve as an address itself. For instance, a symbolic heap corresponding to the assertion φ from above is

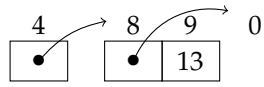
$$\psi \triangleq \exists z \colon x \mapsto y \star y \mapsto 0, z.$$

In contrast to φ, the symbolic heap ψ is satisfied by multiple stack-heap pairs as we may choose different evaluations of the variables x, y, and z. These



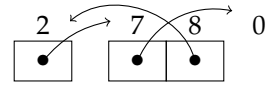
$\mathfrak{s}(x) = 1\ \mathfrak{s}(y) = 6$

Figure 10.2: A stack-heap pair satisfying φ, ψ, and ϑ.



$\mathfrak{s}(x) = 4\ \mathfrak{s}(y) = 8$

Figure 10.3: A stack-heap pair satisfying ψ and ϑ, but not φ.



$\mathfrak{s}(x) = 2\ \mathfrak{s}(y) = 7$

Figure 10.4: A stack-heap pair satisfying ψ, but not φ, ϑ.

stack-heap pairs do not necessarily have the same shape due to aliasing. For example, three stack-heap pairs satisfying ψ are depicted in Figures 10.2 to 10.4. Figure 10.2 illustrates the heap $\mathfrak{h}$; it thus also satisfies assertion φ. The heap in Figure 10.3 has the same shape as $\mathfrak{h}$, but uses different addresses and values; it thus violates φ. Finally, in Figure 10.4, variables x and z are aliases which leads to a cycle that is not present in $\mathfrak{h}$. To prevent aliasing in ψ, it suffices to add pure formulas. This leads to the following symbolic heap which is satisfied for the heaps in Figures 10.2 and 10.3, but not for the heap in Figure 10.4:

$$\vartheta \;\triangleq\; \exists z\colon x \mapsto y \star y \mapsto 0, z \star (\mathbf{emp} \wedge x \neq z \wedge y \neq z).$$

Notice that pure formulas are guarded by the empty-heap predicate **emp**. Otherwise, assertion ϑ would become intuitionistic. This would again admit heaps whose shape differs from $\mathfrak{h}$. Consequently, the number of addresses allocated in any heap satisfying the symbolic heaps ψ and ϑ is fixed. Since this is inadequate for reasoning about programs manipulating unbounded linked data structures, such as lists or trees, we also allow symbolic heaps to contain predicate calls. We now formally introduce the symbolic heap fragment of separation logic.

Definition 10.6 (Syntax of Symbolic Heaps) From now on, let E be an expression that is either the constant 0 or a variable in **Vars**. Moreover, let $\mathbf{Sel} = \langle\, sel_0, \ldots, sel_n \,\rangle$ be a fixed sequence of selectors. Then the set **SHSL** of *symbolic heaps* is given by the following context-free grammar:

$$\varphi \;\rightarrow\; x.sel \mapsto E \mid \mathbf{emp} \wedge E = E' \mid \mathbf{emp} \wedge E \neq E' \mid \boldsymbol{P}(\vec{E}) \mid \varphi \star \varphi \mid \exists x\colon \varphi,$$

where x is a variable, *sel* is a selector, and $\boldsymbol{P}(\vec{E})$ is a predicate call.

Since symbolic heaps are a syntactic fragment of separation logic assertions, their semantics is determined by the semantics of general assertions (see Figure 10.1). We write **emp** and false as shortcuts for $\mathbf{emp} \wedge 0 = 0$ and $\mathbf{emp} \wedge 0 \neq 0$, respec-

tively. There is, however, no symbolic heap which is equivalent to true. In fact, every symbolic heap containing no predicate calls is precise (cf. Definition 6.23).

A reader familiar with separation logic might notice that our points-to assertions differ slightly from various common definitions of symbolic heaps (cf. [IRS13; Ant+14; Bro+14]): Rather than having points-to assertions for fixed-size records, e.g., $x \mapsto y, z, 0$, we allow to specify points-to assertions for single selectors, e.g., $x.sel_0 \mapsto y$. This notion is slightly more liberal. While complete records can always be defined by providing a points-to assertion for every selector, our notion also enables us to define *incomplete* records. For example,

$$x.sel_0 \mapsto y \star x.sel_2 \mapsto z$$

specifies a record in which the selector sel_1 is missing. We will exploit this ability in Chapter 13 in order to efficiently discharge certain entailments.

Example 10.7 In Section 4.3, we have already encountered various separation logic predicates which can be written as symbolic heaps:

- A symbolic heap specifying a singly-linked list segment from x to y of length at least two (cf. Example 4.26) is given by:

$$\varphi \triangleq \exists z\colon \exists z'\colon x.n \mapsto z \star z.n \mapsto z' \star \mathbf{sll}(z', y).$$

- Since variable y might alias with x, z, or z', some stack-heap pairs satisfying φ contain cycles. For example, the following symbolic heap specifies a "lasso", i.e., a list that ends in a cycle.

$$\psi \triangleq \underbrace{\exists z\colon \exists z'\colon x.n \mapsto z \star z.n \mapsto z' \star \mathbf{sll}(z', y)}_{= \varphi} \star (\mathbf{emp} \wedge y = z).$$

- To prevent cycles, we could, for example, require that the list is null-terminated:

$$\vartheta \triangleq \exists z\colon \exists z'\colon x.n \mapsto z \star z.n \mapsto z' \star \mathbf{sll}(z', y) \star (\mathbf{emp} \wedge y = 0).$$

  Variables x and z are not aliases, because both variables are allocated, i.e., occur on the left-hand side of a points-to assertion. Moreover, z and z' are not aliases, because either $z' = 0$ or z' is allocated.

- Finally, the symbolic heap below specifies doubly-linked lists of length two (cf. Example 4.27):

$$\eta \;\triangleq\; \exists x\colon \exists y\colon \exists z\colon x.p \mapsto 0 \star x.n \mapsto z \\ \star\, z.p \mapsto x \star z.n \mapsto y \star y.p \mapsto z \star y.n \mapsto 0.$$

When working with symbolic heaps, it is convenient to assume that formulas are in prenex normal form. Moreover, due to associativity and commutativity of the separating conjunction (see Theorem 4.17), we may collect points-to assertions, predicate calls, equalities, and inequalities in separate blocks.

Definition 10.8 (Normal Form) A *symbolic heap* φ is in *normal form* if and only if it adheres to the following context-free grammar:

$$\begin{aligned} \varphi &\;\to\; \exists x\colon \varphi \mid \mathit{Ptr} \star \mathit{Call} \star \mathit{Eq} \star \mathit{Ineq} \\ \mathit{Ptr} &\;\to\; \mathbf{emp} \mid x.\mathit{sel} \mapsto E \star \mathit{Ptr} \\ \mathit{Call} &\;\to\; \mathbf{emp} \mid \boldsymbol{P}(\vec{E}) \star \mathit{Call} \\ \mathit{Eq} &\;\to\; \mathbf{emp} \mid (\mathbf{emp} \wedge E = E') \star \mathit{Eq} \\ \mathit{Ineq} &\;\to\; \mathbf{emp} \mid (\mathbf{emp} \wedge E \neq E') \star \mathit{Ineq}. \end{aligned}$$

Every variable that appears in a symbolic heap φ in normal form is either free or existentially quantified. Similarly to the set $\mathbf{Vars}(\varphi)$ of free variables of φ, we denote by $\mathbf{BV}(\varphi)$ the set of all variables that are bound by a quantifier in φ.

All symbolic heaps in Example 10.7 are *almost* in normal form. In fact, each symbolic heap misses a few **emp** predicates which—since **emp** is the neutral element of $\star$—can be added or removed at will. The following lemma justifies that we focus on symbolic heaps in normal form:

Lemma 10.9 For every symbolic heap φ there is a symbolic heap ψ in normal form such that, for all stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$, we have

$$\mathfrak{s}, \mathfrak{h} \models \varphi \quad \text{iff} \quad \mathfrak{s}, \mathfrak{h} \models \psi.$$

Proof. By structural induction on the syntax of symbolic heaps. □

While it is useful to assume that a symbolic heap is in normal form, the exact syntax of our normal form is a bit clumsy. For instance, transforming the symbolic heap $\mathbf{emp} \wedge x = 0$ into normal form yields

$$\mathbf{emp} \star \mathbf{emp} \star (\mathbf{emp} \wedge x = 0) \star \mathbf{emp}.$$

This motivates the introduction of more compact notation for symbolic heaps in normal form in which pure formulas are collected in sets and both points-to

assertions and predicate calls are collected in *multisets*.[1] To this end, let us first introduce some convenient notation for variables and predicate calls:

Assumption 10.10 We assume an arbitrary, but fixed, total order $\preccurlyeq$ on the set of all variables in **Vars** and on the (surely disjoint) set of all predicate calls $\boldsymbol{P}(\vec{E})$, where $\boldsymbol{P} \in \mathbf{PSym}$ is a predicate symbol and $\vec{E}$ is a sequence of expressions of length $\mathsf{param}(\boldsymbol{P})$.

Moreover, we denote by $\mathbf{Vars}^{\preccurlyeq}(\varphi)$ the *ordered sequence of free variables* of symbolic heap φ with respect to $\preccurlyeq$. That is, if $\mathbf{Vars}(\varphi) = \{ x_1, x_2, \ldots, x_n \}$ and $x_1 \preccurlyeq x_2 \preccurlyeq \ldots \preccurlyeq x_n$, then $\mathbf{Vars}^{\preccurlyeq}(\varphi) \triangleq \langle x_1, x_2, \ldots, x_n \rangle$.

We occasionally write $\varphi(\vec{x})$ to highlight that the ordered sequence of free variables of symbolic heap φ is $\vec{x} = \mathbf{Vars}^{\preccurlyeq}(\varphi)$.

Similar to [BCO05b], we omit both separating and standard conjunctions and denote symbolic heaps in normal form by their individual components; each component is separated by the symbol $\mid$. Formally, we denote symbolic heaps in normal form as follows:

Definition 10.11 (Multiset Notation for Symbolic Heaps) We identify every symbolic heap φ in normal form with a *symbolic heap in multiset notation*

$$\mathfrak{M}(\varphi) \triangleq \mathbf{Vars}^{\preccurlyeq}(\varphi) \mid \mathbf{PT}(\varphi) \mid \mathbf{PC}(\varphi) \mid \mathbf{EQ}(\varphi) \mid \mathbf{NE}(\varphi),$$

where each individual component is defined as follows:

- The sequence $\mathbf{Vars}^{\preccurlyeq}(\varphi)$ collects all free variables of φ arranged in ascending order with respect to the total order $\preccurlyeq$.
- The multiset $\mathbf{PT}(\varphi)$ consists of all *points-to* assertions in φ.
- The multiset $\mathbf{PC}(\varphi)$ consists of all *predicate calls* in φ.
- The set $\mathbf{EQ}(\varphi)$ consists of all *equalities* in φ.
- The set $\mathbf{NE}(\varphi)$ consists of all *inequalities* in φ.

Formally, each of the above components is defined by structural induction on the syntax of symbolic heaps in normal form as shown in Figure 10.5.

Furthermore, we denote by $\mathbf{PC}^{\preccurlyeq}(\varphi)$ the sequence consisting of all predicate calls in $\mathbf{PC}(\varphi)$ in ascending order with respect to $\preccurlyeq$.

[1] We write $\{\{ a, a, b, c \}\}$ to represent a finite multiset with elements a, a, b, and c. We denote by $M \uplus M'$ the union between multisets M and M'. Moreover, we denote by $M \setminus\!\setminus M'$ the removal of all elements in multiset M' from multiset M. Formal definitions are found in Appendix G.

φ	$\mathbf{PT}(\varphi)$	$\mathbf{PC}(\varphi)$	$\mathbf{EQ}(\varphi)$	$\mathbf{NE}(\varphi)$
$\exists x \colon \psi$	$\mathbf{PT}(\psi)$	$\mathbf{PC}\,(\psi)$	$\mathbf{EQ}(\psi)$	$\mathbf{NE}(\psi)$
$\mathit{Ptr} \star \mathit{Call} \star \mathit{Eq} \star \mathit{Ineq}$	$\mathbf{PT}(\mathit{Ptr})$	$\mathbf{PC}(\mathit{Call})$	$\mathbf{EQ}(\mathit{Eq})$	$\mathbf{NE}(\mathit{Ineq})$

Ptr / Call / Eq / Ineq	$\mathbf{PT}(\mathit{Ptr})$ / $\mathbf{PC}(\mathit{Call})$ / $\mathbf{EQ}(\mathit{Eq})$ / $\mathbf{NE}(\mathit{Ineq})$
$\mathbf{emp}$	$\emptyset$
$x.\mathit{sel} \mapsto E \star \psi$	$\{\!\{ x.\mathit{sel} \mapsto E \}\!\} \uplus \mathbf{PT}(\psi)$
$\boldsymbol{P}(\vec{E}) \star \psi$	$\{\!\{ \boldsymbol{P}(\vec{E}) \}\!\} \uplus \mathbf{PC}(\psi)$
$(\mathbf{emp} \wedge E = E') \star \psi$	$\{ E = E' \} \cup \mathbf{EQ}(\psi)$
$(\mathbf{emp} \wedge E \neq E') \star \psi$	$\{ E \neq E' \} \cup \mathbf{NE}(\psi)$

Figure 10.5: Inductive definitions of the (multi-)sets $\mathbf{PT}(\varphi)$, $\mathbf{PC}(\varphi)$, $\mathbf{EQ}(\varphi)$, and $\mathbf{NE}(\varphi)$ for a given symbolic heap φ in normal form.

It is straightforward to construct a (unique up to commutativity and associativity of the separating conjunction) symbolic heap in normal form for any given symbolic heap in multiset notation: Intuitively, it suffices to apply the inductive definitions in Figure 10.5 backward. In particular, all variables which are not explicitly listed in the sequence of free variables are existentially quantified. Consequently, we adopt all notions defined for symbolic heaps, e.g., substitution of variables by expressions or the set of free variables of an assertion, by applying them to the underlying symbolic heap.

To improve readability, we omit brackets indicating tuples, multisets, and sets because the type of each component of a symbolic in multiset notation is clear by Definition 10.11. Moreover, as long as it is clear from the context that we use multiset notation, we often omit empty components rather than explicitly writing ε for the empty sequence or $\emptyset$ for the empty (multi-)set.

Example 10.12 Recall from Example 10.7 the symbolic heaps φ, ψ, ϑ, and η. Apart from missing **emp** formulas, these symbolic heaps are already in normal form. Their representation in multiset notation is shown below:

- $\mathfrak{M}(\varphi) = x, y \mid x.n \mapsto z, z.n \mapsto z' \mid \boldsymbol{sll}(z', y)$,

$E/\ \varphi$	$f(.)$	
0	0	
x	$f(x)$	if $x \in \mathbf{BV}(\varphi)$
x	x	if $x \notin \mathbf{BV}(\varphi)$
$x.sel \mapsto E$	$f(x).sel \mapsto f(E)$	
$\mathbf{emp} \wedge E = E'$	$\mathbf{emp} \wedge f(E) = f(E')$	
$\mathbf{emp} \wedge E \neq E'$	$\mathbf{emp} \wedge f(E) \neq f(E')$	
$\boldsymbol{P}(E_1, \ldots, E_n)$	$\boldsymbol{P}(f(E_1), \ldots, f(E_n))$	
$\psi \star \vartheta$	$f(\psi) \star f(\vartheta)$	
$\exists x \colon \psi$	$\exists f(x) \colon f(\psi)$	

Figure 10.6: Inductive definition of the variable renaming $f(\varphi)$.

- $\mathfrak{M}(\psi) = x, y \mid x.n \mapsto z, z.n \mapsto z' \mid \boldsymbol{sll}(z', y) \mid y = z$,
- $\mathfrak{M}(\vartheta) = x, y \mid x.n \mapsto z, z.n \mapsto z' \mid \boldsymbol{sll}(z', y) \mid y \neq x, y \neq z, y \neq z'$, and
- $\mathfrak{M}(\eta) = x.p \mapsto 0, x.n \mapsto z, z.p \mapsto x, z.n \mapsto y, y.p \mapsto z, y.n \mapsto 0$.

Before we consider predicate calls, we remark that our results are up to isomorphism of symbolic heaps, i.e., up to reordering of separating conjunctions, renaming of existentially quantified variables, and adding or removing **emp**. Since free variables are understood as program variables, however, they may not be renamed: A list from x to y is different from a list from y to x.

Definition 10.13 (Symbolic Heap Isomorphism) Two symbolic heaps φ and ψ are *isomorphic*, written $\varphi \cong \psi$, iff there exists a bijective function

$$f \colon \mathbf{BV}(\varphi) \to \mathbf{BV}(\psi)$$

such that $\mathfrak{M}(f(\varphi)) = \mathfrak{M}(\psi)$, where $f(\varphi)$ denotes the application of function f to every occurrence of a quantified variable in symbolic heap φ; a formal definition is found in Figure 10.6.

Lemma 10.14 For all symbolic heaps φ, ψ, we have

$$\varphi \cong \psi \qquad \text{implies} \qquad \forall(\mathfrak{s}, \mathfrak{h}) \colon \quad \mathfrak{s}, \mathfrak{h} \models \varphi \text{ iff } \mathfrak{s}, \mathfrak{h} \models \psi.$$

Proof. By structural induction on the syntax of symbolic heaps. □

The converse direction does not hold in general, because symbolic heaps might contain redundant pure formulas which are not taken into account by our notion of isomorphic symbolic heaps. Lemma 10.14 justifies the following assumption:

Assumption 10.15 Throughout the remainder of this thesis, *we do not distinguish between isomorphic symbolic heaps*. Since all symbolic heaps φ with the same multiset notation $\mathfrak{M}(\varphi)$ are isomorphic (choose f as the identity), *we also do not distinguish between symbolic heaps and their representation in multiset notation.* We will thus use both notations interchangeably.

10.1.2 Inductive Predicate Definitions

In Section 4.3.4, we formalized the semantics of a predicate call, say $\boldsymbol{P}(\vec{E})$, by means of a recursive equation of the form

$$\boldsymbol{P}(\vec{v}) \;=\; \Phi(\boldsymbol{P})(\vec{v}),$$

where Φ is a monotone predicate transformer. Similarly, to deal with multiple predicate symbols, it suffices to consider a system of recursive equations. The semantics of $\boldsymbol{P}(\vec{E})$ is then defined as the least fixed point of Φ.

The same approach remains adequate for reasoning about symbolic heaps. However, now that we have an explicit syntax, we would like to reuse the same syntax for defining suitable monotone—and in fact even continuous—predicate transformers Φ rather than reverting to arbitrary predicates. To this end, we consider systems of inductive definitions in **SHSL**:

Definition 10.16 (Systems of Inductive Definitions [Bro+14]) A *system of inductive definitions* Ψ (SID for short) is a finite set of *rules* of the form

$$\boldsymbol{P} \;\Leftarrow\; \varphi,$$

where $\boldsymbol{P} \in \mathbf{PSym}$, $\varphi \in \mathbf{SHSL}$, and $\mathsf{param}(\boldsymbol{P}) = |\mathbf{Vars}\,(\varphi)\,|$.

We denote by $\Psi(\boldsymbol{P})$ the set of all symbolic heaps appearing on the right-hand side of a rule in Ψ with left-hand side $\boldsymbol{P}$, i.e.,

$$\Psi(\boldsymbol{P}) \;\triangleq\; \{\, \varphi \mid (\boldsymbol{P} \Leftarrow \varphi) \in \Psi \,\}.$$

Moreover, the set $\mathbf{PSym}(\Psi)$ collects all left-hand sides of rules in Ψ, i.e.,

$$\mathbf{PSym}(\Psi) \;\triangleq\; \{\, \boldsymbol{P} \in \mathbf{PSym} \mid \Psi(\boldsymbol{P}) \neq \emptyset \,\}.$$

How do SIDs specify the semantics of predicate calls? Intuitively, every SID Ψ describes a system of equations, where, for every predicate symbol $\boldsymbol{P} \in \mathbf{PSym}(\Psi)$, we introduce the following equation:

$$\boldsymbol{P}(v_1, \ldots, v_n) \;\triangleq\; \bigvee_{\varphi(x_1,\ldots,x_n) \in \Psi(\boldsymbol{P})} \varphi\,[x_1/v_1] \ldots [x_n/v_n]\,.$$

The resulting equation system yields a monotone and continuous predicate transformer Φ which, for every predicate symbol $\boldsymbol{P}$, is of the form

$$\Phi(\boldsymbol{P})\colon \quad (\mathbb{Z}^{\mathsf{param}(\boldsymbol{P})} \to \mathbf{Pred}) \;\to\; (\mathbb{Z}^{\mathsf{param}(\boldsymbol{P})} \to \mathbf{Pred}).$$

To determine the semantics of symbolic heaps with respect to Φ, we introduce a dedicated satisfaction relation $\models_\Phi$. Apart from predicate calls, $\models_\Phi$ is defined analogously to the semantics of assertions introduced in Figure 10.1, page 269. Moreover, for predicate calls $\boldsymbol{P}(E_1, \ldots, E_n)$, it is defined as

$$\mathfrak{s}, \mathfrak{h} \models_\Phi \boldsymbol{P}(E_1, \ldots, E_n) \quad \text{iff} \quad \mathfrak{s}, \mathfrak{h} \models \mathsf{lfp}\,(\Phi)\,(\boldsymbol{P})(E_1\,(\mathfrak{s})\,, \ldots, E_n\,(\mathfrak{s})).$$

Example 10.17 Let us define SIDs resulting in the same semantics for list and tree predicates as the recursive definitions in Section 4.3.4:

- An SID Ψ_{sll} for singly-linked segments is given by the rules

$$\boldsymbol{sll} \Leftarrow x, y \mid x = y, \quad \boldsymbol{sll} \Leftarrow x, y \mid x.\mathit{next} \mapsto z \mid \boldsymbol{sll}(z, y).$$

- An SID Ψ_{tree} for binary trees is given by the rules

$$\boldsymbol{tree} \Leftarrow x \mid x = 0, \quad \boldsymbol{tree} \Leftarrow x \mid x.\mathit{left} \mapsto y, x.\mathit{right} \mapsto z \mid \boldsymbol{tree}(y), \boldsymbol{tree}(z).$$

- An SID Ψ_{dll} for doubly-linked list segments is given by the rules

$$\begin{aligned} \boldsymbol{dll} &\Leftarrow x', x, y, y' \mid x = y', y = x', \\ \boldsymbol{dll} &\Leftarrow x', x, y, y' \mid x.\mathit{prev} \mapsto x', x.\mathit{next} \mapsto z \mid \boldsymbol{dll}(x, z, y, y'). \end{aligned}$$

Unfortunately, the least fixed point of a predicate transformer Φ is, in general, not a symbolic heap. In fact, it is not even a syntactic separation logic assertion. For example, the least fixed point characterized by each of the above SIDs is an *infinite* disjunction of symbolic heaps.

However, a single disjunct suffices in order to prove that a predicate call is satisfied by a given stack-heap pair. To conclude this section, let us thus briefly consider an alternative semantics that stays within the realm of symbolic heaps. This semantics is based on the notion of iteratively substituting predicate calls by symbolic heaps according to the rules of an SID.

Definition 10.18 (Predicate Substitution) Let φ be a symbolic heap containing a predicate call $\boldsymbol{P}(E_1, \ldots, E_n) \in \mathbf{PC}(\varphi)$. Moreover, let $\psi(x_1, \ldots, x_n)$ be a symbolic heap such that $\mathbf{BV}(\varphi) \cap \mathbf{BV}(\psi) = \emptyset$. Then the *substitution* of predicate call $\boldsymbol{P}(E_1, \ldots, E_n)$ by symbolic heap ψ in φ is defined as

$$\varphi\,[\boldsymbol{P}(E_1, \ldots, E_n)/\psi] \;\triangleq\; \mathbf{Vars}^{\preccurlyeq}(\varphi) \mid \mathbf{PT} \mid \mathbf{PC} \mid \mathbf{EQ} \mid \mathbf{NE},$$

where, for $\vartheta \triangleq \psi\,[x_1/E_1] \ldots [x_n/E_n]$, the individual components are:

- $\mathbf{PT} \triangleq \mathbf{PT}(\varphi) \uplus \mathbf{PT}(\vartheta)$,
- $\mathbf{PC} \triangleq (\mathbf{PC}(\varphi) \setminus\!\setminus \{\!\{ \boldsymbol{P}(E_1, \ldots, E_n) \}\!\}) \uplus \mathbf{PC}(\vartheta)$,
- $\mathbf{EQ} = \mathbf{EQ}(\varphi) \cup \mathbf{EQ}(\vartheta)$, and
- $\mathbf{NE} = \mathbf{NE}(\varphi) \cup \mathbf{NE}(\vartheta)$.

Example 10.19 Consider the following three symbolic heaps φ, ψ, and ϑ:

$$\varphi \;\triangleq\; x'.prev \mapsto 0, x'.next \mapsto x, y'.prev \mapsto y, y'.next \mapsto 0 \mid \boldsymbol{dll}(x', x, y, y'),$$
$$\psi \;\triangleq\; x', x, y, y' \mid x.prev \mapsto x', x.next \mapsto z \mid \boldsymbol{dll}(x, z, y, y'), \text{ and}$$
$$\vartheta \;\triangleq\; x', x, y, y' \mid x = y', y = x',$$

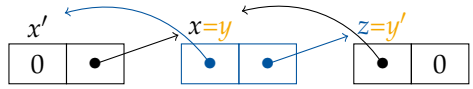
where ψ and ϑ correspond to rules of the SID Ψ_{dll} in Example 10.17. Then the symbolic heap $\eta \;\triangleq\; \varphi\,[\boldsymbol{dll}(x', x, y, y')/\psi]$ is given by:

$$x'.prev \mapsto 0, x'.next \mapsto x, y'.prev \mapsto y, y'.next \mapsto 0,$$
$$x.prev \mapsto x', x.next \mapsto z \mid \boldsymbol{dll}(x, z, y, y').$$

Moreover, the symbolic heap $\eta' \triangleq \eta\,[\boldsymbol{dll}(x, z, y, y')/\vartheta]$ is given by:

$$x'.prev \mapsto 0, x'.next \mapsto x, y'.prev \mapsto y, y'.next \mapsto 0,$$
$$x.prev \mapsto x', x.next \mapsto z \mid z = y', y = x.$$

As illustrated below, η' specifies doubly-linked lists of length three:



Intuitively, an unfolding step then amounts to the substitution of a predicate call by a symbolic heap.

Definition 10.20 (Folding and Unfolding Steps) Let Ψ be an SID. An *unfolding step* of symbolic heap φ with respect to Ψ yields the symbolic heap ψ iff there exists a predicate call $\boldsymbol{P}(\vec{E}) \in \mathbf{PC}(\varphi)$, a symbolic heap $\vartheta \in \Psi(\boldsymbol{P})$, and a symbolic heap $\eta \cong \vartheta$ such that $\psi \cong \varphi\left[\boldsymbol{P}(\vec{E}) \middle/ \eta\right]$. In this case, we write $\varphi \underset{\Psi}{\Leftarrow} \psi$. We denote by $\overset{*}{\underset{\Psi}{\Leftarrow}}$ the reflexive and transitive closure of $\underset{\Psi}{\Leftarrow}$.

Moreover, we frequently write $\varphi \underset{\Psi}{\Rightarrow} \psi$ instead of $\psi \underset{\Psi}{\Leftarrow} \varphi$ to denote a *folding step* starting with symbolic heap ψ. Analogously, $\overset{*}{\underset{\Psi}{\Rightarrow}}$ denotes the reflexive and transitive closure of $\underset{\Psi}{\Rightarrow}$.

For instance, for the symbolic heaps φ and η' in Example 10.19, we have

$$\varphi \overset{*}{\underset{\Psi_{dll}}{\Longleftarrow}} \eta'.$$

Here, the symbolic heap η' contains no predicate calls itself. We call such a symbolic heap an *unfolding* of φ.

Definition 10.21 (Unfoldings) The set $\mathbf{Unf}_\Psi(\varphi)$ of *unfoldings* of symbolic heap φ with respect to SID Ψ is defined as the set of symbolic heaps

$$\mathbf{Unf}_\Psi(\varphi) \triangleq \left\{ \psi \;\middle|\; \varphi \overset{*}{\underset{\Psi}{\Leftarrow}} \psi \text{ and } \mathbf{PC}(\psi) = \emptyset \right\}.$$

Assumption 10.22 Without loss of generality, we assume that all rules of any SID Ψ considered in this thesis are *productive*, i.e., for all rules $(\boldsymbol{P} \Leftarrow \varphi) \in \Psi$, we have $\mathbf{Unf}_\Psi(\varphi) \neq \emptyset$. This can always be achieved by iteratively removing unproductive rules.

To determine whether a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ satisfies a predicate call $\boldsymbol{P}(\vec{E})$ with respect to SID Ψ, it suffices to check whether there exists some unfolding of that predicate call which is satisfied by $(\mathfrak{s}, \mathfrak{h})$. Formally, we define the alternative semantics of predicate calls as

$$\mathfrak{s}, \mathfrak{h} \models_\Psi \boldsymbol{P}(\vec{E}) \quad \text{iff} \quad \exists \psi \in \mathbf{Unf}_\Psi(\boldsymbol{P}(\vec{E})) \colon \mathfrak{s}, \mathfrak{h} \models \psi.$$

The theorem below states that the set of unfoldings of a predicate call with respect to an SID Ψ collects exactly the disjuncts of the corresponding transformer's least fixed point. Consequently, both semantics coincide.

Theorem 10.23 (Equivalence of SHSL Semantics) Let Ψ be an SID. Moreover, let Φ be the predicate transformer corresponding to Ψ. Then:

(a) $\mathsf{lfp}\,(\Phi)\,(\boldsymbol{P})(v_1,\ldots,v_n) = \bigvee_{\varphi\in\mathbf{Unf}_\Psi(\boldsymbol{P}(x_1,\ldots,x_n))} \varphi\,[x_1/v_1]\ldots[x_n/v_n]$, and

(b) for all symbolic heaps φ, we have $\mathfrak{s},\mathfrak{h}\models_\Psi \varphi$ iff $\mathfrak{s},\mathfrak{h}\models_\Phi \varphi$.

Proof (Sketch). To prove Theorem 10.23 (a), notice that the predicate

$$Q \triangleq \mathsf{lfp}\,(\Phi)\,(\boldsymbol{P})(v_1,\ldots,v_n) \;=\; \lim_{n\to\infty} \Phi^n(\lambda \boldsymbol{Q}.\,\mathsf{false})$$

can be written as a potentially infinite disjunction of (satisfiable) assertions by construction of Φ. We then have to show that for each disjunct ψ of Q there is an unfolding $\varphi \in \mathbf{Unf}_\Psi(\boldsymbol{P}(x_1,\ldots,x_n))$ such that ψ and $\varphi\,[x_1/v_1]\ldots[x_n/v_n]$ are equivalent and vice versa.

We show that for each disjunction there exists an unfolding by complete induction on the number n of fixed point iterations. The converse direction is shown by complete induction on the number of unfolding steps applied to derive an unfolding $\varphi \in \mathbf{Unf}_\Psi(\boldsymbol{P}(x_1,\ldots,x_n))$.

To prove Theorem 10.23 (b), we first observe that—due to Theorem 10.23 (a) and the semantics of disjunctions $\vee$—for all predicate calls $\boldsymbol{P}(\vec{E})$, we have

$$\mathfrak{s},\mathfrak{h}\models_\Psi \boldsymbol{P}(\vec{E}) \quad \text{iff} \quad \mathfrak{s},\mathfrak{h}\models_\Phi \boldsymbol{P}(\vec{E}).$$

Since both semantics coincide except for predicate calls, the claim then follows by structural induction on the syntax of symbolic heaps. □

Finally, we remark that the order in which predicates are unfolded is irrelevant. In particular, this means that—to decide whether a stack-heap pair satisfies a given symbolic heap φ—it suffices to consider an unfolding of φ rather than individual unfoldings of its predicate calls:

Lemma 10.24 For all SIDs Ψ and symbolic heaps φ, we have:

$$\mathfrak{s},\mathfrak{h}\models_\Psi \varphi \quad \text{iff} \quad \exists \vartheta \in \mathbf{Unf}_\Psi(\varphi)\colon \mathfrak{s},\mathfrak{h}\models \vartheta.$$

Proof. By structural induction on the syntax of symbolic heaps. □

10.2 Related Work

A detailed discussion of separation logic in general is found in Chapter 4. In this section, we focus on work related to *automated* reasoning with separation logic; in particular its symbolic heap fragment. However, there exist other logics for reasoning about heap manipulating programs (cf. [DKR04; MPQ11; Qiu+13; PQM14]). Further related work will be discussed where appropriate.

10.2.1 Reasoning about Symbolic Heaps

Although symbolic heaps have been extensively studied in the literature, there is not a single symbolic heap fragment. In particular, definitions of symbolic heaps differ in the following aspects:

- the exact form of points-to assertions (for example, compare [BCO05b; BDP11; Coo+11; JGN14]),
- the form and treatment of predicate calls (for example, compare [BCO04; Ber+07; BDP11; IRS13; IRV14; TK15]),
- the usage of existential quantifiers in assertions and inductive definitions (for example, compare [BCO04; IRS13; Bro+14; Ant+14]),
- closure under (possibly restricted) Boolean operations (for example, compare [BCO04; Coo+11; IRS13; IRV14]), and
- whether intuitionistic assertions are definable (compare [BDP11; Bro+14]).

To the best of the author's knowledge, the term symbolic heap itself was—at least in the context of separation logic—coined by Berdine, Calcagno, and O'Hearn [BCO05b]; the same fragment is, however, already considered in [BCO04]. It is restricted to fixed predicates for singly-linked lists (and later also binary trees) rather than allowing user-supplied SIDs. Apart from that, their fragment coincides with our notion of symbolic heaps. In particular, they also use selector-based points-to assertions. Furthermore, our multiset notation (see Definition 10.11) is inspired by their notation for symbolic heaps. Regarding decidability, the same authors show in [BCO04, Lemma 1, Proposition 16] that model checking is decidable in linear time and entailment is decidable in coNP. For singly-linked list predicates, it is noteworthy that the entailment problem becomes polynomial if points-to assertions are additionally restricted to a single selector [Coo+11]. The situation changes dramatically once one admits user-supplied SIDs: Antonopoulos et al. [Ant+14] showed that the entailment problem for symbolic heaps with user-supplied SIDs is undecidable. Their fragment also admits defining intuitionistic assertions. Still, the entailment

problem remains undecidable for our fragment This can be shown by reducing the undecidable inclusion problem for context-free grammars (cf. [BPS64]) to the entailment problem (see [IRV14, Theorem 2] or Appendix H.3).

Since our fragment of symbolic heaps is not closed under Boolean operations, the entailment problem does *not* reduce to the satisfiability problem. In fact, Brotherston et al. [Bro+14; Bro+16] proved that both satisfiability and model checking are decidable even for user-defined SIDs. Moreover, they showed that both problems are ExpTime-complete unless further restrictions are imposed. Their fragment of symbolic heaps is comparable to ours with two exceptions: First, they allow existential quantifiers in SIDs but not in assertions. Second, points-to assertions have to specify complete records, i.e., are of the form $E \mapsto E_1, \ldots, E_n$. The approach underlying their decision procedures can be considered a special case of our framework for reasoning about robustness of SIDs presented in Chapter 11 (cf. [2; 19]). Their base-pair construction to prove satisfiability is similar to our construction of a suitable heap automaton. However, they do not consider refinement of SIDs. We use their results for analyzing the complexity of deciding robustness properties.

While the entailment problem is undecidable for arbitrary SIDs, various decidable fragments have been identified. Iosif, Rogalewicz, and Simacek [IRS13] study symbolic heaps (with closure under Boolean operations) for a restricted class of user-supplied SIDs such that every assertion can be defined in monadic second-order logic over graphs of bounded tree width (cf. [CE12]). As a consequence, both satisfiability and entailment are decidable with non-elementary complexity. Moreover, the entailment problem for this fragment is ExpTime-hard (cf. [Ant+14, Theorem 5] and [IRV14, Theorem 3]). If inequalities are removed from symbolic heaps and SIDs are further restricted to "local edges", which in effect ensures that all models are tree-shaped if read as undirected graphs, the entailment problem becomes ExpTime-complete [IRV14, Theorem 3]. Similar reductions to monadic second-order logic over graphs of bounded tree width have been used to prove decidability of the entailment problem for variations of their fragment [17; TK15]. We consider one of their restrictions on SIDs—called establishment—as an example of a robustness property.

Apart from decision procedures, a pragmatic approach is taken by Brotherston, Distefano, and Petersen [BDP11]: They present a sound, but incomplete cyclic proof system to discharge entailments between symbolic heaps with arbitrary user-supplied inductive predicate definitions. Their approach has recently been extended to a complete proof system for a restricted set of user-defined inductive predicates by Iosif and Serban [IS18]. Furthermore, in [BG14], cyclic provers are used to automatically infer SIDs from programs which then serve as preconditions. We use these automatically generated SIDs to evaluate our framework for reasoning about robustness properties.

Finally, we remark that other fragments than symbolic heaps have been studied in the literature. This includes combining fragments of separation logic with theories to reason about data stored in dynamic data structures [PWZ13; PWZ14; GCW16; LSC16; Le+17; KJW18; LSQ18]. In particular, Le et al. [LSC16; Le+17] present a decision procedure for satisfiability of symbolic heaps extended by data constraints in Presburger arithmetic; their approach can be considered as an extension of our decision procedure for satisfiability based on heap automata.

10.2.2 Automated Verification Tools

In Chapter 13, we present a model checker for Java pointer programs called Attestor [4; 3]. Let us briefly discuss how Attestor relates to other verification tools. There exists a landscape of tools for analyzing heap manipulating programs based on different paradigms such as separation logic, automata or graph transformations. These tools differ heavily in their input, the degree of automation, the properties they allow to establish about pointer programs, and their presentation of results. In the following, we thus restrict ourselves to a choice of tools that are similar to Attestor in more than one of these categories.

Juggrnaut Juggrnaut [HNR10; Hei+15; Hei15] is a tool for analyzing heap manipulating programs using the abstraction mechanism based on hyperedge replacement grammars (HRG for short) proposed by Rieger and Noll [RN08]. Tree grammars were already applied by Lee, Yang, and Yi [LYY05]. HRGs, however, allow to specify richer classes of data structures.

Attestor can be seen as the successor of Juggrnaut. It remains based on abstractions guided by HRGs. The computation of these abstractions, however, is different and imposes fewer constraints on the form of grammars. Furthermore, Attestor improves on Juggrnaut in the following points:

- It supports verification of a rich set of properties that are specified in linear temporal logic with properties of heaps, e.g., reachability between locations, as atomic propositions. These properties are checked using heap automata [2] and tableau-based model checking [CGH94; CGH97].
- It is able to generate concrete, non-spurious counterexamples for debugging purposes whenever a safety property is violated.
- It implements a contract-based, procedure modular approach to verify recursive programs (cf. [JGN14]).
- It ships with a graphical frontend to allow for manual exploration of generated abstract state spaces and counterexamples.

- It supports a fragment of symbolic heap separation logic as an alternative input format to graphs and HRGs.
- It is equipped with a decision procedure for proving that the abstraction induced by a given HRG is confluent—a key property to ensure decidability of the language inclusion (think: entailment) problem.

The precise relationship between HRGs and symbolic heap separation logic has been worked out by Dodds [Dod08a; Dod08b] and Jansen et al. [JGN14; Jan17]. Intuitively, one can think of an HRG as an SID whose unfoldings are satisfied by exactly one canonical stack-heap pair up to renaming of locations and aliasing of free variables. In Section 12.3, we study the set of *graphical symbolic heaps* which is equivalent to the set of graphs captured by HRGs considered in Attestor.

It is noteworthy that the notion of robustness properties (cf. [2]) was originally motivated by this close relationship. In fact, the refinement theorem for symbolic heaps can be considered as the analog to the filter theorem in the realm of HRGs [Hab92, Theorem 5.1].

Separation Logic Tools Various automated verification tools for pointer programs either perform a symbolic execution [BCO05b] or a shape analysis [Ber+07] with symbolic heap separation logic as its underlying domain.

Early tools based on separation logic, such as Smallfoot [BCO05a], SpaceInvader [DOY06], and jStar [DP08], relied on specialized syntactic rewrite rules to normalize symbolic heaps and resolve entailments for a fixed set of predicate symbols modeling singly-linked lists and binary trees.

Similarly, Attestor's abstraction mechanism is based on systematic rewriting. However, since its rewriting rules are based on a dedicated and well-established formalism for (graph) rewriting, namely HRGs [Hab92], it appears to be more natural. In particular, various concepts, such as confluence or termination of abstraction, are readily available. Furthermore, all rewrite steps are inherently up to isomorphism of the underlying graphs. Berdine, Cook, and Ishtiaq noted later that "syntactic variable occurrence conditions [...] proved too fragile when using a more general assertion logic" [BCI11, p. 182]. They thus proposed to take formulas "modulo provable equality" into account which appears in part to be an axiomatization of isomorphism.

While rewriting has a less prominent rule in modern tools for reasoning about separation logic—which rely more heavily on SMT solvers (see, e.g., [PR13; GCW16; Ta+16; Ene+17b; IS18; LSQ18])—it appears in proof systems in the form of fold and unfold rules to deal with inductive predicates (cf. for instance, [BDP11; Chi+12]).

The Infer [CD11; Cal+11] static analyzer is perhaps the most prominent tool using separation logic for automatic verification. It automatically synthesizes

specifications in order to prove memory safety on industrial codebases in a fully automatic fashion. However, it does not support reasoning about the kind of temporal specifications considered by Attestor, such as "every element of a doubly-linked list has been accessed upon termination".

Forester Forester [Hol+13] is a shape analysis tool for C programs, that operates on the domain of generalized tree automata. It combines the local reasoning aspects of separation logic with abstract regular tree model checking [Bou+12]. The tool is fully automatic. In contrast to Attestor, it tries to detect the data structures emerging from the analyzed programs. Although the theoretical approach would allow for richer properties, Forester is limited to checking memory safety only. Forester is a long-term participant in the heap-related categories of SV-COMP and comes with a library of benchmarks. After transferring these benchmarks to Java, Attestor was not only able to cover large amounts of them, but also showing additionally shape and reachability properties.

TVLA The Three-Valued-Logic Analyzer (Tvla) [Bog+07] implements a popular abstraction technique for pointer programs, namely parametric shape analysis [SRW99; SRW02]. It is capable of proving memory safety, shape properties, reachability properties, and sortedness of data structures. Most of these are derivable from Attestor's heap representation. Tvla relies on user-supplied instrumentation predicates to improve the analysis, in particular to refine the abstract heap representation. For example, to verify Lindstrom's algorithm, 24 predicates were used [LRS06]. A key difference is that Attestor's heap representation is not tailored to a specific property. Rather, the graph grammars guiding abstraction are applicable to a wide range of properties.

Groove The Groove tool [Gha+12] implements a technique for verifying graph transformation systems based on model checking. While it supports an automatic state space generation, the integration of abstraction mechanisms is still ongoing. As already noted in [Hei+15], Groove is capable of simulating Attestor's abstraction mechanism by manually encoding it in graph transformation systems. This manual approach allowed us to verify a non-recursive list reversal in a reasonable amount of time. However, we were unable to apply Groove to more complicated case studies, such as Lindstrom's algorithm.[2]

[2]The encoding was developed together with Victor Lanvin from ENS Cachan during his time as an intern at RWTH Aachen and Christina Jansen.

Automated Reasoning about Robustness of Symbolic Heaps

This chapter is based on prior publications, namely [2; 19; 9], which are presented, discussed, and extended in the broader context of this thesis.

To handle a wide range of data structures in separation logic assertions, there is an ongoing trend to support custom systems of inductive definitions (SID for short, see Definition 10.16) that are either defined manually [Jac+11; Chi+12] or inferred automatically [BG14]. Allowing for arbitrary SIDs, however, raises questions about their *robustness*. A user-supplied or automatically generated SID might, for example, be *inconsistent*, introduce *unallocated logical variables*, specify data structures that contain undesired *cycles*, or produce *garbage*, i.e., parts of the heap that are unreachable from any program variable.

The above issues can harm the performance, completeness, and even soundness of verification algorithms:

- Brotherston et al. [Bro+14] point out that tools might waste time on inconsistent scenarios due to the *unsatisfiability* of specifications.
- The absence of unallocated logical variables, also known as *establishment*, is required by the approach of Iosif et al. [IRS13; IRV14] to obtain a decidable fragment of symbolic heaps.
- Other verification approaches, such as [Hab+11; Hab+12], assume that *no garbage* is introduced by data structure specifications.
- Zanardini and Genaim [ZG14] argue that *acyclicity* of the heap is crucial in automated termination proofs.
- More generally, Berdine, Calcagno, and O'Hearn [BCO04] notice that "one of the most treacherous passes in pointer verification and analysis is *reachability*. To describe common loop invariants, and even some pre- and postconditions, one needs to be able to assert that there is a path in the heap from one value to another; a fragment that cannot account for reachability in some way will be of very limited use".

Being able to check whether an SID is *robust* in the above sense is thus crucial for debugging specifications prior to verification and during the verification process itself. While individual robustness properties have been studied in detail, e.g., satisfiability [Bro+14], other properties, such as establishment [IRS13], have been addressed with ad-hoc solutions whenever they arose.

Motivating Example Let us consider the problem of acyclicity. That is, we would like to decide whether all stack-heap pairs satisfying a symbolic heap φ—whose predicate calls are determined by the SID Ψ_{asll} below—are acyclic.

$$\Psi_{\mathit{asll}} \triangleq \{\, \mathbf{asll} \Leftarrow x, y \mid x = y, \quad \mathbf{asll} \Leftarrow x, y \mid x.\mathit{next} \mapsto z \mid \mathbf{asll}(z, y) \mid x \neq y \,\}$$

Intuitively, Ψ_{asll} specifies acyclic singly-linked lists with head x and tail y, where the inequality $x \neq y$ ensures acyclicity. Now, assume we have to decide whether

$$\varphi \triangleq z.\mathit{next} \mapsto y \mid \mathbf{asll}(x, z), \mathbf{asll}(y, x)$$

is acyclic as well. We can do so by inductive reasoning as follows:

- We first analyze the predicate call $\mathbf{asll}(x, z)$. If it is unfolded according to the first rule of Ψ_{asll}, then there is no cycle in $\mathbf{asll}(x, z)$. Moreover, we notice that z is reachable from x.
- If we already know for some predicate call $\mathbf{asll}(x', z)$ that all stack-heap pairs satisfying it are acyclic and that z is reachable from x', then z is also reachable from x in the symbolic heap

  $$\psi \triangleq x, z \mid x.\mathit{next} \mapsto x' \mid \mathbf{asll}(x', z) \mid x \neq z$$

  obtained by unfolding the predicate call $\mathbf{asll}(x, z)$ according to the second rule of Ψ_{asll}. Furthermore, we notice that ψ is acyclic.
- By induction, $\mathbf{asll}(x, z)$ is thus acyclic and z is reachable from x.
- Likewise, $\mathbf{asll}(y, x)$ is acyclic and x is reachable from y.
- Now, based on this information, we examine the symbolic heap φ and conclude that *it is cyclic* because z is reachable from x, y is reachable from z, and x is reachable from y.

In summary, we examine how a symbolic heap is unfolded with respect to an SID *bottom-up*, starting from the non-recursive base-case. At each stage of this analysis, we remember only a fixed amount of information—namely what we discovered about acyclicity and reachability between parameters so far—which is then available at the next stage.

Other questions arise for the above example. For instance, how does a stack-heap pair serving as a counterexample to acyclicity look like? How do we obtain a new SID that does guarantee acyclicity? A systematic treatment of robustness properties should cover these tasks in general, not just for acyclicity.

Problem Statement We are confronted with the following challenges:

1. *Decision procedures for robustness properties.* SIDs specify potentially unbounded data structures; typically, they thus specify infinitely many stack-heap pairs. Our decision procedures need to be able to decide whether all, or some, of these stack-heap pairs satisfy a given robustness property.
2. *Generation of counterexamples* that violate a desired robustness property.
3. *Refinement* of SIDs. Whenever an SID violates a robustness property, it should be possible to synthesize a new SID that respects it.

To address these tasks, we develop an algorithmic framework around the notion of *heap automata*—a special kind of tree automata [Com+07] running on infinite alphabets—that traverses symbolic heaps as they are unfolded according to the rules of an SID. Intuitively, heap automata exploit the inductive structure inherent to SIDs in order to compositionally recognize robustness properties. We then derive suitable decision and refinement procedures from their properties, e.g., closure under Boolean operations and decidability of the emptiness problem.

Outline We study heap automata and their properties in Section 11.1. From these properties, we then derive decision procedures and analyze their complexity. The resulting algorithmic framework is applied to the aforementioned robustness properties in Section 11.2. Finally, we briefly report on a prototypical implementation of our framework in Section 11.3.

11.1 An Algorithmic Framework for Robustness Properties

In this section, we study reasoning about robustness properties. By robustness property, we mean any set of symbolic heaps which is accepted by a *heap automaton*, i.e., a device that recognizes symbolic heaps as they are unfolded.

11.1.1 Unfolding Trees

Towards a precise definition of heap automata, we first arrange the unfolding steps (see Definition 10.20) applied to derive an unfolding from a symbolic heap in *unfolding trees*. To this end, a few preliminaries regarding labeled trees

are needed. Our presentation of these preliminaries is roughly based on the standard textbook on tree automata by Comon et al. [Com+07].

A *ranked alphabet* is a (possibly infinite) set $\mathbf{RA}$ with an associated function

$$\mathsf{rank}\colon \mathbf{RA} \to \mathbb{N}$$

that assigns a *rank* to each symbol in $\mathbf{RA}$. For every $n \in \mathbb{N}$, we denote by $\mathbf{RA}_n$ the set of all symbols of rank n, i.e., $\mathbf{RA}_n \triangleq \{\, a \in \mathbf{RA} \mid \mathsf{rank}(a) = n \,\}$.

We consider the set $\mathbf{SHSL}$ of all symbolic heaps as a ranked alphabet, where $\mathsf{rank}(\varphi)$ is the rank of the symbolic heap φ is given by its number of predicate calls. That is, $\mathsf{rank}(\varphi) \triangleq |\mathbf{PC}(\varphi)|$. Consequently, the set $\mathbf{SHSL}_0$ collects all symbolic heaps that contain no predicate calls and thus cannot be further unfolded by applying SID rules. We arrange symbols over a ranked alphabet in trees, where a symbol's rank determines its number of children. Formally:

Definition 11.1 (Trees [Com+07, pp. 15–17]) Let $\mathbf{RA}$ be a ranked alphabet.

(a) A *tree* over $\mathbf{RA}$ is a partial function $t\colon \mathbb{N}^* \to \mathbf{RA}$ such that

- $\mathsf{dom}(t)$ is non-empty,
- $\mathsf{dom}(t)$ is prefix-closed, i.e., if $uv \in \mathsf{dom}(t)$, then $u \in \mathsf{dom}(t)$, and
- $\forall u\colon t(u) \in \mathbf{RA}_n$ implies $\{\, i \in \mathbb{N} \mid ui \in \mathsf{dom}(t) \,\} = \{\, 1, \ldots, n \,\}$.

(b) We denote by $\mathbf{Trees}(\mathbf{RA})$ the set of all trees over $\mathbf{RA}$.

(c) The *subtree* $t|_u$ of tree t with root $u \in \mathsf{dom}(t)$ is defined as

$$\mathsf{dom}(t|_u) \triangleq \{\, v \mid u\,v \in \mathsf{dom}(t) \,\}, \qquad t|_u(v) \triangleq t(uv).$$

(d) The *height* $h(t)$ of a tree $t \in \mathbf{Trees}(\mathbf{RA})$ is given by

$$h(t) \triangleq \begin{cases} 1, & \text{if } \mathsf{rank}(t(\varepsilon)) = 0 \\ 1 + \max\{\, h(t|_1), \ldots, h(t|_n) \,\}, & \text{if } \mathsf{rank}(t(\varepsilon)) = n > 0. \end{cases}$$

(e) For each $a \in \mathbf{RA}_0$, we identify a with the tree t given by $t(\varepsilon) = a$.

(f) Analogously, for each $n > 0$, $a \in \mathbf{RA}_n$, and $t_1, \ldots, t_n \in \mathbf{Trees}(\mathbf{RA})$, we identify the term $a(t_1, \ldots, t_n)$ with the tree t given by

$$t(\varepsilon) = a,\ t|_1 = t_1,\ \ldots,\ t|_n = t_n.$$

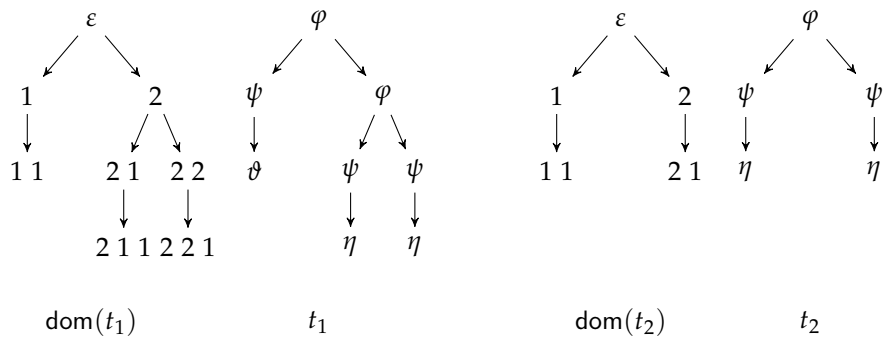


Figure 11.1: Illustration of trees t_1 and t_2 and their domains.

Example 11.2 As a running example, let us consider trees over the ranked alphabet **SHSL**. To this end, we define the following symbolic heaps:

- $\varphi \triangleq x, y \mid z.next \mapsto z' \mid \mathit{asll}(x, z), \mathit{asll}(z', y) \in \mathbf{SHSL}_2$,
- $\psi \triangleq x, y \mid x.next \mapsto z \mid \mathit{asll}(z, y) \mid x \neq y \in \mathbf{SHSL}_1$,
- $\vartheta \triangleq x \mid x.left \mapsto 0, x.right \mapsto 0 \in \mathbf{SHSL}_0$, and
- $\eta \triangleq x, y \mid x = y \in \mathbf{SHSL}_0$.

Figure 11.1 depicts two trees t_1, t_2 over **SHSL** together with their domains. In our alternative notation for trees, these correspond to:

$$t_1 = \varphi(\psi(\vartheta), \varphi(\psi(\eta), \psi(\eta))) \quad \text{and}$$
$$t_2 = \varphi(\psi(\eta), \psi(\eta)).$$

Furthermore, notice that t_2 is a subtree of t_1, namely $t_2 = t_1|_2$.

Our primary motivation for introducing trees over **SHSL** is to systematically describe the unfolding steps applied to derive an unfolding of a symbolic heap, say φ, with respect to an SID, say Ψ (cf. Definitions 10.20 and 10.21). Intuitively, such a tree t has φ at its root, i.e., $t(\varepsilon) = \varphi$. Moreover, every child corresponds to the right-hand side of a rule of Ψ which is used to unfold a predicate call of φ. Their children, in turn, are used to unfold predicate calls introduced through previous rule applications. We call trees with this property *unfolding trees*.

Definition 11.3 (Unfolding Trees [2]) Let Ψ be an SID and φ be a symbolic heap. Then the set $\mathbf{Trees}_\Psi(\varphi)$ of *unfolding trees* of φ with respect to Ψ is the least set of trees in $\mathbf{Trees}(\mathbf{SHSL})$ defined by:

- if $\mathbf{PC}(\varphi) = \emptyset$, then $\varphi \in \mathbf{Trees}_\Psi(\varphi)$, and
- if $\mathbf{PC}^{\preccurlyeq}(\varphi) = \left\langle \boldsymbol{P}_1(\vec{E}_1), \ldots, \boldsymbol{P}_n(\vec{E}_n) \right\rangle$ and, for each $i \in [1, n]$, we have $\psi_i \in \Psi(\boldsymbol{P}_i)$ and $t_i \in \mathbf{Trees}_\Psi(\psi_i)$, then $\varphi(t_1, \ldots, t_n) \in \mathbf{Trees}_\Psi(\varphi)$.

Example 11.4 Recall from Example 11.2 the symbolic heaps φ, ψ, ϑ, and η. Then the tree t_2 in Figure 11.1 is an unfolding tree of φ with respect to the SID $\Psi_{\boldsymbol{asll}}$ consisting of the rules below:

$$\boldsymbol{asll} \Leftarrow \underbrace{x, y \mid x = y}_{=\eta} \quad \text{and} \quad \boldsymbol{asll} \Leftarrow \underbrace{x, y \mid x.next \mapsto z \mid \boldsymbol{asll}(z, y) \mid x \neq y}_{=\psi}.$$

The tree t_1 in Figure 11.1, however, is *not* an unfolding tree for *any* SID, because it contains $\psi(\vartheta)$ as a subtree: The predicate call $\boldsymbol{asll}(z, y)$ in ψ contains two parameters. In contrast, the symbolic heap ϑ has only a single free variable. It thus cannot be an unfolding tree of $\boldsymbol{asll}(z, y)$.

Every unfolding tree describes a unique (up to isomorphism) unfolding which is obtained by recursively substituting predicate calls by the unfoldings specified by its subtrees:[1]

Definition 11.5 (Composition of Unfolding Trees [2]) Let φ be a symbolic heap with $\mathbf{PC}^{\preccurlyeq}(\varphi) = \left\langle \boldsymbol{P}_1(\vec{E}_1), \ldots, \boldsymbol{P}_n(\vec{E}_n) \right\rangle$ for some $n \geq 0$. Then the *composition* $[\![t]\!]$ of unfolding tree $t \in \mathbf{Trees}_\Psi(\varphi)$ is the symbolic heap

$$[\![t]\!] \triangleq \begin{cases} \varphi, & \text{if } t = \varphi \\ \varphi \left[\boldsymbol{P}_1(\vec{E}_1) \Big/ [\![t_1]\!] \right] \ldots \left[\boldsymbol{P}_n(\vec{E}_n) \Big/ [\![t_n]\!] \right], & \text{if } t = \varphi(t_1, \ldots, t_n). \end{cases}$$

Example 11.6 The composition $[\![t_2]\!]$ of the unfolding tree t_2 considered in Example 11.4 is computed as follows:

$$\begin{aligned} & [\![t_2]\!] \\ = \; & [\![\text{ Definition 11.5 }]\!] \end{aligned}$$

[1] We can safely assume that existentially quantified variables are renamed before applying predicate substitution to avoid name clashes due to Assumption 10.15.

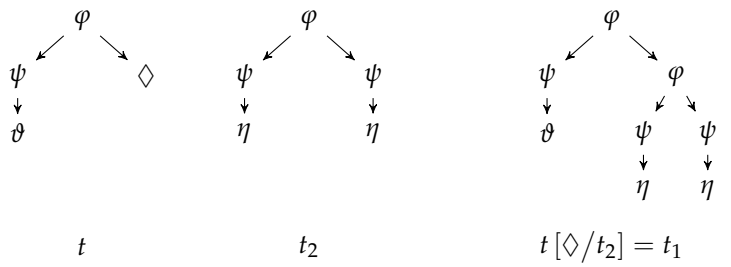


Figure 11.2: Illustration of a context t and the substitution by the tree t_2.

$$
\begin{aligned}
& \varphi\,[\mathbf{asll}(x,z)/[\![\psi(\eta)]\!]]\,[\mathbf{asll}(z',y)/[\![\psi(\eta)]\!]] \\
= \;& [\![\text{ Definition 11.5 }]\!] \\
& \varphi\,[\mathbf{asll}(x,z)/\psi\,[\mathbf{asll}(z,y)/[\![\eta]\!]]]\,[\mathbf{asll}(z',y)/\psi\,[\mathbf{asll}(z,y)/[\![\eta]\!]]] \\
= \;& [\![\text{ Definition 11.5 } ([\![\eta]\!] = \eta) \,]\!] \\
& \varphi\,[\mathbf{asll}(x,z)/\psi\,[\mathbf{asll}(z,y)/\eta]]\,[\mathbf{asll}(z',y)/\psi\,[\mathbf{asll}(z,y)/\eta]] \\
= \;& [\![\text{ Definition 10.18 } (\psi\,[\mathbf{asll}(z,y)/\eta] = x,y \mid x.next \mapsto z \mid z = y \mid x \neq y) \,]\!] \\
& x,y \mid z.next \mapsto z', x.next \mapsto z'', z'.next \mapsto z''' \mid z'' = z, z''' = y \mid x \neq z, z' \neq y \\
= \;& [\![\text{ Remove quantified variables } z'' \text{ and } z''\text{; reorder points-to assertions }]\!] \\
& x,y \mid x.next \mapsto z, z.next \mapsto z', z'.next \mapsto y \mid x \neq z, z' \neq y.
\end{aligned}
$$

Recall from Definitions 10.20 and 10.21 that the set $\mathbf{Unf}_\Psi(\varphi)$ of unfoldings of symbolic heap φ with respect to SID Ψ is obtained by exhaustive application of unfolding steps. A more structured alternative to characterize the same set is to compute the composition of all unfolding trees in $\mathbf{Trees}_\Psi(\varphi)$. In fact, the semantics of symbolic heaps is occasionally defined via unfolding trees in the literature, e.g., in [IRS13; IRV14; 2]. This is justified by the lemma below.

Lemma 11.7 ([2]) For every SID Ψ and every symbolic heap φ, we have

$$\mathbf{Unf}_\Psi(\varphi) \;=\; \{\, [\![t]\!] \mid t \in \mathbf{Trees}_\Psi(\varphi) \,\}$$

Proof. The inclusion "$\subseteq$" is shown by complete induction on the length of the number of unfolding steps.

The inclusion "$\supseteq$" is proven similarly by induction on the height of unfolding trees. □

Finally, it is convenient to extend a given tree such that the original tree is as a subtree of the extended one. To this end, a *context* is a tree with a "hole", i.e., a symbol $\Diamond$ of rank zero, at which some subtree may be inserted. Formally:

Definition 11.8 (Contexts [Com+07, p. 17]) Let **RA** be a ranked alphabet. Moreover, let $\Diamond$ be a special symbol of rank zero which is not in **RA**.

(a) A (linear) *context* over **RA** is a tree $t \in \mathbf{Trees}(\mathbf{RA} \cup \{ \Diamond \})$ such that there exists exactly one $u \in \mathsf{dom}(t)$ with $t(u) = \Diamond$.

(b) We denote by $t\left[\Diamond / t'\right]$ the *substitution* of $\Diamond$ by the tree $t' \in \mathbf{Trees}(\mathbf{RA})$ in context t. That is, if $t(u) = \Diamond$, then we define:

$$\mathsf{dom}\left(t\left[\Diamond / t'\right]\right) \triangleq \mathsf{dom}(t) \cup \left\{ uv \mid v \in \mathsf{dom}(t') \right\},$$

$$t\left[\Diamond / t'\right](v) \triangleq \begin{cases} t(v), & \text{if } v \in \mathsf{dom}(t) \setminus \{ u \} \\ t'(w), & \text{if } v = uw. \end{cases}$$

Example 11.9 Recall from Example 11.2 the trees t_1 and t_2 over the symbolic heaps φ, ψ, ϑ, and η. As illustrated in Figure 11.2, there is a context t such that the tree t_1 results from substituting the symbol $\Diamond$ by the tree t_2 in t.

11.1.2 Heap Automata

A *heap automaton* is a device that traverses an unfolding tree in order to check whether its composition satisfies a property of interest. As such it is a special kind of (bottom-up) tree automaton (cf. [Com+07, Section 1.1]) running on an infinite ranked alphabet. Let us thus briefly consider tree automata first:

Definition 11.10 (Tree Automata [Com+07]) A *tree automaton* $\mathcal{A}$ is a tuple

$$\mathcal{A} \triangleq \langle \mathcal{Q}, \mathbf{RA}, \mathcal{F}, \Delta \rangle,$$

where $\mathcal{Q}$ is a finite ranked alphabet of *states* of rank zero, **RA** is a ranked alphabet which is disjoint from $\mathcal{Q}$, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of *final states*, and Δ is a (decidable) *set of transition rules* of the form

$$a(\mathrm{q}_1, \ldots, \mathrm{q}_n) \to \mathrm{q},$$

where $a \in \mathbf{RA}$ with $\mathsf{rank}(a) = n \geq 0$, and $\mathrm{q}, \mathrm{q}_1, \ldots, \mathrm{q}_n \in \mathcal{Q}$.

Notice that the ranked alphabet **RA** is in general *not* finite. To discharge whether a transition rule is possible, we thus assume that the set Δ is characterized by

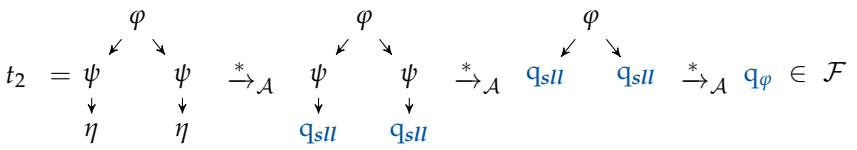


Figure 11.3: Illustration of the moves proving that $t \xrightarrow{*}_{\mathcal{A}} \mathrm{q}_{\varphi}$.

a decidable predicate. Intuitively, a tree automaton rewrites a given tree by iteratively applying its transition rules. That is, it first substitutes every symbol of rank zero, say b, by one of its states, say q, provided that it contains a transition rule $b \to \mathrm{q}$. After that, the automaton keeps replacing subtrees of the form $a(\mathrm{q}_1, \ldots, \mathrm{q}_n)$ according to its transition rules until either only a tree consisting of a single state $\mathrm{q} \in \mathcal{Q}$ remains or no further rule is applicable.

Definition 11.11 (Moves and Languages [Com+07]) Let $\mathcal{A} \triangleq \langle\, \mathcal{Q}, \mathbf{RA}, \mathcal{F}, \Delta \,\rangle$ be a tree automaton. Moreover, let $t, t' \in \mathbf{Trees}(\mathbf{RA} \cup \mathcal{Q})$ be trees over $\mathbf{RA}$ and $\mathcal{Q}$. Then $\mathcal{A}$ may perform a *move* $t \to_{\mathcal{A}} t'$ from t to t' iff there exists a context t'' over $\mathbf{RA}$ and a transition rule $a(\mathrm{q}_1, \ldots, \mathrm{q}_n) \to \mathrm{q} \in \Delta$ such that

- $t = t''\,[\Diamond / a(\mathrm{q}_1, \ldots, \mathrm{q}_n)]$, and
- $t' = t''\,[\Diamond / \mathrm{q}]$.

We denote by $\xrightarrow{*}_{\mathcal{A}}$ the reflexive, and transitive closure of move relation $\to_{\mathcal{A}}$. Furthermore, the *language* $\mathbf{L}(\mathcal{A})$ of tree automaton $\mathcal{A}$ consists of all trees *accepted by* $\mathcal{A}$, i.e., all trees such that $\mathcal{A}$ can eventually move to a final state:

$$\mathbf{L}(\mathcal{A}) \triangleq \left\{ t \in \mathbf{Trees}(\mathbf{RA}) \;\middle|\; \exists \mathrm{q} \in \mathcal{F} \colon t \xrightarrow{*}_{\mathcal{A}} \mathrm{q} \right\}.$$

Example 11.12 Recall from Example 11.2 the symbolic heaps φ, ψ, η. Moreover, consider the tree automaton $\mathcal{A} = \langle\, \mathcal{Q}, \mathbf{SHSL}, \mathcal{F}, \Delta \,\rangle$ given by:

- the set of states $\mathcal{Q} \triangleq \{\, \mathrm{q}_{sll}, \mathrm{q}_{\varphi} \,\}$, the set of final states $\mathcal{F} \triangleq \{\, \mathrm{q}_{\varphi} \,\}$, and
- the transition rules $\Delta \triangleq \{\, \eta \to \mathrm{q}_{sll}, \psi(\mathrm{q}_{sll}) \to \mathrm{q}_{sll}, \varphi(\mathrm{q}_{sll}, \mathrm{q}_{sll}) \to \mathrm{q}_{\varphi} \,\}$.

Figure 11.3 illustrates how $\mathcal{A}$ moves through the unfolding tree $t_2 \in \mathbf{Trees}_{\Psi_{asll}}(\varphi)$ from Example 11.4 to the final state q_{φ}. Hence, $t_2 \in \mathbf{L}(\mathcal{A})$. In fact, the language of $\mathcal{A}$ consists of all unfolding trees of φ with respect to SID Ψ_{asll}, i.e., $\mathbf{L}(\mathcal{A}) = \mathbf{Trees}_{\Psi_{asll}}(\varphi)$.

More generally, for every SID Ψ and every symbolic heap φ, it is possible to construct a tree automaton $\mathcal{A} = \langle\, \mathcal{Q}, \mathbf{SHSL}, \mathcal{F}, \Delta \,\rangle$ whose language $\mathbf{L}(\mathcal{A})$ is the set $\mathbf{Trees}_\Psi(\varphi)$ of unfolding trees of φ with respect to Ψ: Its set of states consists of all predicate symbols of SID Ψ and a dedicated state corresponding to φ, i.e.,

$$\mathcal{Q} \triangleq \mathbf{PSym}(\Psi) \cup \{\, \mathrm{q}_\varphi \,\}.$$

Furthermore, its set of final states is $\mathcal{F} = \{\, \mathrm{q}_\varphi \,\}$. Finally, its set of transition rules Δ is the least set determined by:

- if $\mathbf{PC}^{\preccurlyeq}(\varphi) = \left\langle\, \boldsymbol{P}_1(\vec{E}_1), \ldots, \boldsymbol{P}_n(\vec{E}_n) \,\right\rangle$, then Δ contains the transition rule $\varphi(\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n) \to \mathrm{q}_\varphi$, and
- if $\psi \in \mathbf{SHSL}_n$ with $\mathbf{PC}^{\preccurlyeq}(\psi) = \left\langle\, \boldsymbol{P}_1(\vec{E}_1), \ldots, \boldsymbol{P}_n(\vec{E}_n) \,\right\rangle$ and there exists a rule $(\boldsymbol{P} \Leftarrow \psi) \in \Psi$, then Δ contains the transition rule $\psi(\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n) \to \boldsymbol{P}$.

In the above construction, we introduced one transition rule per rule of SID Ψ. Moreover, we added a single rule to move to the unique final state. Consequently, the size $\|\mathcal{A}\|$ (cf. Assumption 10.4) of the resulting tree automaton is linear in the size of Ψ and φ. Since the emptiness problem for (finite) tree automata is decidable in linear time (cf. [Com+07, Theorem 1.7.4]), we conclude that:

Lemma 11.13 For every SID Ψ and every symbolic heap φ, it is decidable in $\mathcal{O}(\|\Psi\| + \|\varphi\|)$ whether $\mathbf{Trees}_\Psi(\varphi) = \emptyset$ holds.

Our ultimate goal is to reason about properties of *symbolic heaps* rather than their unfolding trees. Tree automata are in general not suited to describe properties of symbolic heaps. After all, there might be two different unfolding trees, say t_1 and t_2, with respect to different SIDs whose composition leads to the same symbolic heap ψ, i.e., $\psi \cong [\![t_1]\!] \cong [\![t_2]\!]$. Now, if a tree automaton $\mathcal{A}$ accepts t_1, but not t_2, does this mean that the symbolic heap ψ satisfies the property specified by $\mathcal{A}$? To circumvent such phenomena, one could restrict the SIDs under consideration: If no SID admits the unfolding tree t_2, it seems reasonable to conclude that ψ satisfies the property specified by tree automaton $\mathcal{A}$.

Definition 11.14 (Unfoldable Symbolic Heaps) For a set $\mathbf{C}$ of SIDs, we denote by $\mathbf{SHC}$ the set of all symbolic heaps that can be obtained via unfolding with respect to some SID in $\mathbf{C}$. That is,

$$\mathbf{SHC} \triangleq \left\{\, \varphi \;\middle|\; \exists \Psi \in \mathbf{C}\ \exists \boldsymbol{P}(\vec{E}) \colon \boldsymbol{P} \in \mathbf{PSym}(\Psi) \text{ and } \boldsymbol{P}(\vec{E}) \overset{*}{\underset{\Psi}{\Leftarrow}} \varphi \,\right\}.$$

For example, we often consider the set $\mathbf{FV}^{\leq k}$ of all SIDs in which the number of free variables in all rules is bounded by some natural number k.

Unfortunately, even if we take only a single SID into account, we encounter undesirable phenomena: Recall from Example 11.12 the tree automaton $\mathcal{A}$ accepting all unfolding trees of symbolic heap φ with respect to the SID Ψ_{asll}. We may conceive $\mathcal{A}$ as a tree automaton over the ranked alphabet $\mathbf{SH}\{\Psi_{\mathit{asll}}\}$. Moreover, as illustrated in Figure 11.3, there is a tree t_2 accepted by $\mathcal{A}$. We have already computed the composition $[\![t_2]\!]$ of t_2 in Example 11.6. Clearly, $[\![t_2]\!]$ is a symbolic heap of rank zero in $\mathbf{SH}\{\Psi_{\mathit{asll}}\}$; in particular, we may use it as a tree itself. However, the tree given by $[\![t_2]\!]$ is *not* accepted by $\mathcal{A}$. In other words, we still have to reason about unfolding trees rather than symbolic heaps. The same issue propagates to subtrees of unfolding trees: If the tree $\varphi(t, t')$ is accepted by $\mathcal{A}$, this does not guarantee that the tree $\varphi([\![t]\!], [\![t']\!])$ is accepted by $\mathcal{A}$ as well.

To reason about properties of symbolic heaps, we thus restrict ourselves to tree automata that do not discriminate between unfolding trees and their composed symbolic heaps. We call such devices *heap automata*:

Definition 11.15 (Heap Automata and Robustness Properties) Let $\mathbf{C}$ be a set of SIDs. A *heap automaton* over $\mathbf{C}$ is a tree automaton

$$\mathcal{A} \triangleq \langle\, \mathcal{Q}, \mathbf{SHC}, \mathcal{F}, \Delta \,\rangle$$

that additionally satisfies the following *compositionality property*:

$$\forall \Psi \in \mathbf{C}\ \ \forall t \in \mathbf{Trees}_{\Psi}(\mathbf{SHC})\ \ \forall \mathrm{q} \in \mathcal{Q}\colon \qquad t \xrightarrow{*}_{\mathcal{A}} \mathrm{q} \ \text{ iff } \ [\![t]\!] \to_{\mathcal{A}} \mathrm{q}.$$

Furthermore, the *robustness property* $[\![\mathbf{L}(\mathcal{A})]\!]$ specified by heap automaton $\mathcal{A}$ consists of all compositions of trees in the language of $\mathcal{A}$, i.e.,

$$[\![\mathbf{L}(\mathcal{A})]\!] \triangleq \{\, [\![t]\!] \mid t \in \mathbf{L}(\mathcal{A}) \,\}.$$

The compositionality property automatically holds for every tree t of height zero because $[\![t]\!]$ coincides with t. In fact, a simple induction on the height of unfolding trees reveals that it suffices to consider trees of height one in order to check whether the compositionality property holds. This leads to the equivalent, but less readable, notion of the compositionality property originally presented in [2; 19, Definition 6].

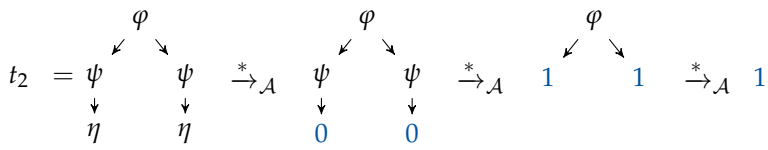


Figure 11.4: Illustration of the moves proving that $t_2 \xrightarrow{*}_{\mathcal{A}} 1$.

Example 11.16 Let us construct a heap automaton $\mathcal{A}$ which checks whether a symbolic heap contains points-to assertions. To this end, let $\mathbf{C}$ be the set of all SIDs, i.e., $\mathbf{SHC} = \mathbf{SHSL}$. The heap automaton $\mathcal{A}$ is then defined as

$$\mathcal{A} \triangleq \langle \{0,1\}, \mathbf{SHSL}, \{1\}, \Delta \rangle,$$

where the state 0 indicates that a symbolic heap contains no and the state 1 indicates that a symbolic heap contains at least one points-to assertion, respectively. Moreover, the set Δ of transition rules is given by:

$$\psi(q_1, \ldots, q_n) \to q \quad \text{iff} \quad q = \begin{cases} 1, & \text{if } |\mathbf{PT}(\psi)| + \sum_{i=1}^{n} q_i > 0 \\ 0, & \text{otherwise.} \end{cases}$$

While $\mathcal{A}$ is a toy example, it illustrates the compositionality property. After all, some symbolic heap in an unfolding tree contains a points-to assertion iff the tree's composition contains a points-to assertion as well.

More concretely, recall from Examples 11.2 and 11.6 the unfolding tree t_2 and its composition $[\![t_2]\!]$. Clearly, $[\![t_2]\!] \to_{\mathcal{A}} 1$. Moreover, as illustrated in Figure 11.4, page 300, we have $t_2 \xrightarrow{*}_{\mathcal{A}} 1$.

11.1.3 Reasoning about Robustness Properties

Our main application of heap automata is to reason about robustness properties, i.e., *infinite* sets of symbolic heaps. Given a heap automaton $\mathcal{A}$, symbolic heap φ, and SID Ψ, we would thus like to answer the following questions:

1. Does there *exist* an unfolding of φ with respect to Ψ that is accepted by $\mathcal{A}$?
2. Are *all* unfoldings of φ with respect to Ψ accepted by $\mathcal{A}$?

To answer these questions, let us begin with a special case of the first question in which φ is a single predicate call. The key idea behind our corresponding decision procedure is to *refine* the SID Ψ in a way such that all unfoldings which

are *not* accepted by $\mathcal{A}$ are *filtered out*. The theorem below guarantees that such a refinement is always computable (cf. [2, Theorem 1]).

Theorem 11.17 (Refinement Theorem [2]) Let $\mathcal{A}$ be a heap automaton over the set of SIDs $\mathbf{C}$. Moreover, let $\Psi \in \mathbf{C}$. Then one can effectively construct a refined SID Γ such that, for each predicate symbol $\boldsymbol{P} \in \mathbf{PSym}(\Psi)$, we have

$$\mathbf{Unf}_\Gamma(\boldsymbol{P}(\vec{x})) \;=\; \mathbf{Unf}_\Psi(\boldsymbol{P}(\vec{x})) \cap [\![\mathbf{L}(\mathcal{A})]\!].$$

Proof. We construct the SID Γ as follows: For every rule $(\boldsymbol{P} \Leftarrow \varphi \in \Psi)$ and every transition rule $\varphi(q_1, \ldots, q_n) \to q$ of heap automaton $\mathcal{A}$ with

$$\mathbf{PC}^{\preccurlyeq}(\varphi) \;=\; \left\langle \boldsymbol{P}_1(\vec{E}_1), \ldots, \boldsymbol{P}_n(\vec{E}_n) \right\rangle,$$

we add a rule $\langle \boldsymbol{P}, q \rangle \Leftarrow \psi$ to Γ, where the symbolic heap ψ coincides with φ except that its predicate calls are given by:

$$\mathbf{PC}^{\preccurlyeq}(\psi) \;=\; \left\langle \langle \boldsymbol{P}_1, q_1 \rangle (\vec{E}_1), \ldots, \langle \boldsymbol{P}_n, q_n \rangle (\vec{E}_n) \right\rangle.$$

Furthermore, for each final state q of heap automaton $\mathcal{A}$, we add a rule $\boldsymbol{P} \Leftarrow \langle \boldsymbol{P}, q \rangle (\vec{x})$ to Γ. To show that our construction of Γ is correct, it suffices to prove by induction on the height of unfolding trees that, for all predicate symbols $\boldsymbol{P} \in \mathbf{PSym}(\Psi)$ and states q of heap automaton $\mathcal{A}$, we have:

$$\varphi \in \mathbf{Unf}_\Gamma(\boldsymbol{P}(\vec{x})) \quad \text{iff} \quad \varphi \in \mathbf{Unf}_\Psi(\boldsymbol{P}(\vec{x})) \text{ and } \varphi \in [\![\mathbf{L}(\mathcal{A})]\!].$$

A detailed proof is found in [19, Lemma 14]. □

Example 11.18 Applying the refinement theorem to the heap automaton $\mathcal{A}$ and the SID $\Psi_{\boldsymbol{asll}}$ in Examples 11.4 and 11.16 yields the refined SID Γ below, where $\boldsymbol{asll}$ specifies all *non-empty* acyclic singly-linked list segments.

$$\begin{aligned}
\boldsymbol{asll} &\Leftarrow x, y \mid \langle \boldsymbol{asll}, 1 \rangle (x, y) \\
\langle \boldsymbol{asll}, 0 \rangle &\Leftarrow x, y \mid x = y \\
\langle \boldsymbol{asll}, 1 \rangle &\Leftarrow x, y \mid x.next \mapsto z \mid \langle \boldsymbol{asll}, 0 \rangle (z, y) \mid x \neq y \\
\langle \boldsymbol{asll}, 1 \rangle &\Leftarrow x, y \mid x.next \mapsto z \mid \langle \boldsymbol{asll}, 1 \rangle (z, y) \mid x \neq y
\end{aligned}$$

To answer the first question raised at the beginning of this section, it then suffices to refine a given SID and subsequently check whether its set of unfolding trees is non-empty. By Theorem 11.17, we can always compute a suitable refined SID. Moreover, by Lemma 11.13, the last step is decidable. Hence, the first question

Input: SID Ψ, $P \in \mathbf{PSym}(\Psi)$, and heap automaton $\mathcal{A} = \langle\, \mathcal{Q}, \mathbf{SHC}, \mathcal{F}, \Delta \,\rangle$
Output: yes iff $\mathbf{Unf}_\Psi(P(\vec{x})) \cap [\![\mathbf{L}(\mathcal{A})]\!] = \emptyset$

reachable $:= \emptyset$; // set of reachable refined states
repeat {
  if (reachable $\cap$ ($\{ P \} \times \mathcal{F}$) $\neq \emptyset$) { return *no* };
  pick a state q *in* $\mathcal{Q}$; *pick a rule* $R \Leftarrow \psi$ *in* Ψ ;
  states $:= \varepsilon$; // empty list of states
  for $P'(\vec{E})$ in $\mathbf{PC}^{\preccurlyeq}(\psi)$ { // base case if $\mathbf{PC}^{\preccurlyeq}(\psi) = \varepsilon$
    pick $\langle\, P', \mathrm{p} \,\rangle$ *in* reachable ;
    states := states $\cdot$ p // append p to states
  };
  if (ψ(states) $\rightarrow$ q *in* Δ) {
    reachable := reachable $\cup$ { $\langle\, Q, \mathrm{q} \,\rangle$ } ;
  }
} until reachable *reaches a fixed point (w.r.t. all choices of rules)*;
return *yes*

Algorithm 1: On-the-fly construction of a refined SID with emptiness check.

is decidable. To extend this result from unfoldings of single predicate calls to unfoldings of arbitrary symbolic heaps φ, we just add a rule $P \Leftarrow \varphi$ to our SID, where P is a fresh predicate symbol, and proceed as before. In summary:

Corollary 11.19 (Decidability of Existence of Robust Unfoldings [2]) Let $\mathcal{A}$ be a heap automaton over a set of SIDs $\mathbf{C}$ and $\Psi \in \mathbf{C}$. Then, for each symbolic heap $\varphi \in \mathbf{SHC}$, it is decidable whether there exists an unfolding $\psi \in \mathbf{Unf}_\Psi(\varphi)$ such that $\psi \in [\![\mathbf{L}(\mathcal{A})]\!]$.

The refinement and emptiness check can also be integrated: Algorithm 1 displays a procedure that constructs the refined SID Γ from Theorem 11.17 on-the-fly while checking whether its set of unfoldings is empty for a given predicate symbol. Termination is guaranteed because the size of the set reachable increases in every iteration and is bounded by the finite number of predicate symbols and the finite number of states of the supplied heap automaton.

Regarding complexity, the size of a refined SID Γ obtained from Ψ and a heap automaton $\mathcal{A}$ is bounded by $\|\Psi\| \cdot \|\mathcal{Q}_\mathcal{A}\|^{m+1}$, where m is the maximal number of predicate calls occurring in any rule of Ψ. Consequently, if we denote

by $\|\Delta_{\mathcal{A}}\|$ the complexity of deciding whether a given transition rule belongs to the set of transition rules of heap automaton $\mathcal{A}$, then Algorithm 1 runs in time

$$\mathcal{O}\left(\|\Psi\| \cdot \|\mathcal{Q}_{\mathcal{A}}\|^{m+1} \cdot \|\Delta_{\mathcal{A}}\|\right).$$

While we do not give a lower complexity bound, we show in Section 11.2 that the above problem is ExpTime-complete for various fixed heap automata.

Example 11.20 Let us apply Algorithm 1 to check whether all unfoldings of the predicate ***asll*** defined in Example 11.4 contain no points-to assertions. That is, for the heap automaton $\mathcal{A}$ in Example 11.16, we decide whether

$$\mathbf{Unf}_{\Psi_{asll}}(\textbf{\textit{asll}}(x,y)) \cap [\![\mathbf{L}(\mathcal{A})]\!] = \emptyset.$$

Algorithm 1 first picks the rule $\textbf{\textit{asll}} \Leftarrow \eta$, where the symbolic heap $\eta = x, y \mid x = y$ contains no points-to assertions. It consequently adds the predicate symbol $\langle\, \textbf{\textit{asll}}, 0 \,\rangle$ to the set reachable of reachable predicate-state pairs. In the next iteration, it picks the rule

$$\textbf{\textit{asll}} \Leftarrow \psi \triangleq x, y \mid x.next \mapsto z \mid \textbf{\textit{asll}}(z, y) \mid x \neq y.$$

Since $\langle\, \textbf{\textit{asll}}, 0 \,\rangle \in$ reachable, the sequence states is set to 0 in the `for` loop. After that, we conclude that $\psi(\mathsf{states}) \to 1$ is a transition rule in Δ. Next, the refined predicate symbol $\langle\, \textbf{\textit{asll}}, 1 \,\rangle$ is added to the set reachable. Finally, *no* is returned because 1 is a final state of $\mathcal{A}$, i.e.,

$$\langle\, \textbf{\textit{asll}}, 1 \,\rangle \in \mathsf{reachable} \cap (\{\, \textbf{\textit{asll}} \,\} \times \mathcal{F}) \neq \emptyset.$$

Consequently, some unfolding of ***asll*** is accepted by $\mathcal{A}$.

We now revisit the second question raised at the beginning of this section: Are all unfoldings of a symbolic φ heap with respect to an SID Ψ accepted by a given heap automaton? To answer this question, we first observe that heap automata enjoy the same closure properties as standard finite tree automata.

Theorem 11.21 (Closure Properties [2]) For all heap automata $\mathcal{A}$, $\mathcal{B}$ over $\mathbf{C}$, we can construct heap automata $\mathcal{C}_\cup$, $\mathcal{C}_\cap$, and $\mathcal{C}_\setminus$ over $\mathbf{C}$ such that:

(a) $[\![\mathbf{L}(\mathcal{C}_\cup)]\!] = [\![\mathbf{L}(\mathcal{A})]\!] \cup [\![\mathbf{L}(\mathcal{B})]\!]$,

(b) $[\![\mathbf{L}(\mathcal{C}_\cap)]\!] = [\![\mathbf{L}(\mathcal{A})]\!] \cap [\![\mathbf{L}(\mathcal{B})]\!]$, and

(c) $[\![\mathbf{L}(\mathcal{C}_\setminus)]\!] = \mathbf{SHC} \setminus [\![\mathbf{L}(\mathcal{A})]\!]$.

Proof. The construction of suitable heap automata is analogous to standard constructions for finite tree automata (over finite alphabets). Notice, however, that the correctness of these constructions depends on the fact that we consider heap automata rather than arbitrary tree automata over infinite alphabets. In particular, to prove that the constructed automata yield the required robustness property, it suffices to consider symbolic heaps of rank zero due to the compositionality property. Detailed constructions as well as correctness proofs are found in [19, Theorem 2]. □

Then, by the equivalence $X \subseteq Y$ iff $X \cap (\mathbf{SHC} \setminus Y) = \emptyset$ and Corollary 11.19, it is also decidable whether *all* unfoldings of a symbolic heap with respect to a given SID are accepted by a heap automaton. Hence:

Corollary 11.22 (Decidability of Robust Unfolding Inclusion [2]) Let $\mathcal{A}$ be a heap automaton over a set of SIDs $\mathbf{C}$ and $\Psi \in \mathbf{C}$. Then, for each symbolic heap $\varphi \in \mathbf{SHC}$, it is decidable whether $\mathbf{Unf}_\Psi(\varphi) \subseteq [\![\mathbf{L}(\mathcal{A})]\!]$ holds.

Note that complementation of heap automata in general requires a determinization step which results in an exponentially larger state space and an exponentially higher complexity of evaluating whether a transition rule belongs to $\Delta_\mathcal{A}$. Thus, the question whether $\mathbf{Unf}_\Psi(\varphi) \subseteq [\![\mathbf{L}(\mathcal{A})]\!]$ holds is decidable in time

$$\mathcal{O}\left((\|\varphi\| + \|\Psi\|) \cdot \|2^{\mathcal{Q}_\mathcal{A}}\|^{2\cdot(m+1)} \cdot \|\Delta_\mathcal{A}\|\right),$$

where, again, m denotes the maximal number of predicate calls in both φ and all rules in Ψ. In many cases, however, it is possible to directly construct smaller heap automata for the complement to obtain more efficient decision procedures. For example, this is the case for most heap automata considered in Section 11.2. We do not provide a precise lower complexity bound which depends on the form of (infinite) sets of transition rules admitted in the definition of heap automata. However, we show in Section 11.2 that the above problem is ExpTime-complete for various fixed instances of heap automata.

Apart from decision procedures, Theorem 11.17 enables systematic refinement of SIDs in order to establish a robustness property of interest. For instance, in Section 11.2.2 we will construct a heap automaton accepting all satisfiable symbolic heaps. Consequently, it is possible to construct a refined SID in which every unfolding is satisfiable. If a robustness property is violated, a counterexample is obtained by constructing the refined SID of the complement of its corresponding heap automaton. In this sense, heap automata also support debugging of SIDs that are manually written as data structure specifications or automatically generated.

11.2 A Zoo of Robustness Properties

Heap automata are widely applicable to both decide and establish common properties of SIDs. To justify this claim, we construct heap automata for the robustness properties informally presented at the beginning of this chapter: satisfiability, establishment, reachability, garbage-freedom, and acyclicity.

Assumption 11.23 All of our heap automata are constructed over the set $\mathbf{FV}^{\leq k}$ of SIDs in which, for some arbitrary, but fixed constant $k \in \mathbb{N}$, the number of free variables in all symbolic heaps is bounded by k. To simplify notation, we additionally fix the names of these free variables. That is, we assume for every symbolic heap φ that $\mathbf{Vars}(\varphi) \subseteq \mathbf{Vars}_k \triangleq \{ x_1, \ldots, x_k \}$ and $\mathbf{BV}(\varphi) \cap \mathbf{Vars}_k = \emptyset$. Furthermore, given a set of equalities $\mathbf{EQ}$, we denote by $\mathbf{EQ}_k$, the set of equalities over variables in $\mathbf{Vars}_k$ or 0, i.e.,

$$\mathbf{EQ}_k \triangleq \{ (E = E') \in \mathbf{EQ} \mid E, E' \in \mathbf{Vars}_k \cup \{ 0 \} \}.$$

Analogously, $\mathbf{NE}_k$ denotes the same restriction for a set of inequalities $\mathbf{NE}$.

11.2.1 Tracking Pure Formulas and Allocation

Consider the symbolic heap

$$\varphi \triangleq x_1, x_2 \mid \boldsymbol{P}(x_1, x_2), \boldsymbol{Q}(x_2, z) \mid x_1 = z.$$

Clearly, φ is unsatisfiable if $x_1 = x_2$ holds for every unfolding of $\boldsymbol{P}(x_1, x_2)$ and $x_2 \neq z$ holds for every unfolding of $\boldsymbol{Q}(x_2, z)$. Moreover, φ is unsatisfiable if variable x_1 is allocated, i.e., appears on the left-hand side of a points-to assertion, in every unfolding of $\boldsymbol{P}(x_1, x_2)$ and variable z is allocated in every unfolding of $\boldsymbol{Q}(x_2, z)$: φ requires that $x_1 = z$, but $x_1 \mapsto - \star z \mapsto -$ implies $x_1 \neq z$.

The above example illustrates that reasoning about the satisfiability of symbolic heaps requires detailed knowledge about the relationships between parameters for all unfoldings of predicate calls. We thus first construct a heap automaton which keeps track of these relationships. This automaton is also a useful building block when considering other robustness properties.

We begin with symbolic heaps in $\mathbf{SHFV}_0^{\leq k}$, i.e., symbolic heaps φ with at most k free variables and no predicate calls ($\mathsf{rank}(\varphi) = 0$). In this case, all relationships between variables of a symbolic heap φ are discovered as follows:

First, observe that every points-to assertion $x.sel \mapsto E$ implies $x \neq 0$. Furthermore, every separating conjunction of points-to assertions $x.sel \mapsto E \star y.sel \mapsto E'$ implies $x \neq y$. After adding these inequalities to the set of inequalities $\mathbf{NE}(\varphi)$, we compute the reflexive, symmetric, and transitive closure of its set of equalities

$\mathbf{EQ}(\varphi)$. Finally, we compute the symmetric closure of the set of inequalities $\mathbf{NE}(\varphi)$ to which we additionally add an inequality $E \neq E'$ whenever the previously computed set of equalities contains $E = E_1$ and $E' = E_2$ such that $E_1 \neq E_2$ is in $\mathbf{NE}(\varphi)$. We call the resulting symbolic heap the *closure* of φ:

Definition 11.24 (Closure of Rank-Zero Symbolic Heaps) The *closure*

$$\overline{\varphi} \triangleq \mathbf{Vars}^{\preccurlyeq}(\varphi) \mid \mathbf{PT}(\varphi) \mid \mathbf{EQ} \mid \mathbf{NE}$$

of a symbolic heap φ of rank zero is given by the smallest (with respect to set inclusion $\subseteq$) sets $\mathbf{EQ}$ and $\mathbf{NE}$ satisfying the following rules:

- $\mathbf{EQ}(\varphi) \subseteq \mathbf{EQ}$ and $\mathbf{NE}(\varphi) \subseteq \mathbf{NE}$,
- if $x.sel \mapsto E \in \mathbf{PT}(\varphi)$, then $x \neq 0 \in \mathbf{NE}$,
- if $\{\{ x.sel \mapsto E, y.sel \mapsto E' \}\} \subseteq \mathbf{PT}$, then $x \neq y \in \mathbf{NE}$,
- if $E \in \mathbf{Vars}(\varphi) \cup \mathbf{BV}(\varphi) \cup \{ 0 \}$, then $E = E \in \mathbf{EQ}$,
- if $E = E' \in \mathbf{EQ}$ and $E' \neq E'' \in \mathbf{NE}$, then $E \neq E'' \in \mathbf{NE}$,
- if $E = E', E' = E'' \in \mathbf{EQ}$, then $E = E'' \in \mathbf{EQ}$,
- if $E = E' \in \mathbf{EQ}$, then $E' = E \in \mathbf{EQ}$,
- if $E \neq E' \in \mathbf{NE}$, then $E' \neq E \in \mathbf{NE}$, and
- if $E \neq E \in \mathbf{NE}$, then $E' = E'' \in \mathbf{EQ}$ and $E' \neq E'' \in \mathbf{NE}$ for all E', E''.

Moreover, we denote by $\mathsf{alloc}(\varphi)$ the set of *allocated variables* in φ, i.e.,

$$\mathsf{alloc}(\varphi) \triangleq \{ x \mid \exists (y.sel \mapsto E) \in \mathbf{PT}(\varphi) \colon (x = y) \in \mathbf{EQ}(\overline{\varphi}) \}.$$

The set of all *free allocated variables* is $\mathsf{alloc}_k(\varphi) \triangleq \mathsf{alloc}(\varphi) \cap \mathbf{Vars}_k$.

The last rule intuitively states that an inconsistent inequality leads to an unsatisfiable symbolic heap; we may thus add all possible pure formulas at once.

Example 11.25 The closure of $x_1, x_2 \mid x_1.sel \mapsto 0, z.sel \mapsto 0 \mid x_1 = x_2$ is

$$x_1, x_2 \mid x_1.sel \mapsto 0, z.sel \mapsto 0 \mid 0 = 0, x_1 = x_1, x_2 = x_2, z = z, x_1 = x_2, x_2 = x_1 \\ \mid x_1 \neq 0, 0 \neq x_1, x_2 \neq 0, 0 \neq x_2, z \neq 0, 0 \neq z, x_1 \neq z, z \neq x_1.$$

Notice that the closure of a symbolic heap can be computed in polynomial time by successively applying the above rules until a fixed point is reached. Since

there are at most quadratically many equalities and inequalities between the variables occurring in a symbolic heap (and 0), the fixed point is attained after a polynomial number of iterations. Furthermore, all rules applied to determine the closure of a symbolic heap lead to equivalent symbolic heaps.

Lemma 11.26 For all symbolic heaps φ of rank zero, we have:

$$\forall(\mathfrak{s},\mathfrak{h}): \qquad \mathfrak{s},\mathfrak{h} \models \varphi \quad \text{iff} \quad \mathfrak{s},\mathfrak{h} \models \overline{\varphi}.$$

Provided that a symbolic heap φ is satisfiable at all, its closure $\overline{\varphi}$ explicitly collects all equalities and inequalities that hold for all stack-heap pairs satisfying φ.[2] Our first robustness property then checks whether the closure of a symbolic heap is consistent with a given set of equalities, inequalities, and allocated variables. We formalize this property in terms of "tracking sets":

Definition 11.27 (Tracking Sets) Let $\mathbf{V} \subseteq \mathbf{Vars}_k$ be a set of free variables. Moreover, let $\mathbf{EQ}$ and $\mathbf{NE}$ be sets of equalities and inequalities over $\mathbf{Vars}_k \cup \{0\}$, respectively. Then the *tracking set* $\mathbf{Track}_k(\mathbf{V},\mathbf{EQ},\mathbf{NE})$ is defined as

$$\left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \mathsf{alloc}_k(\varphi) = \mathbf{V}, \mathbf{EQ}_k(\overline{\varphi}) = \mathbf{EQ}, \mathbf{NE}_k(\overline{\varphi}) = \mathbf{NE} \right\}.$$

How do we construct a heap automaton $\mathcal{A}$ which accepts tracking sets? Intuitively, $\mathcal{A}$ stores in its state space which free variables are equal, unequal, and allocated. Its transition relation then enforces that the stored information is correct, i.e., assuming correctness of the information assigned to predicate calls, we verify correctness of the information assigned to the whole symbolic heap.

Towards a formal construction, we fix a symbolic heap for every tracking set encoding all required relationships between free variables:

Definition 11.28 (Kernel) Let $\mathbf{Track}_k(\mathbf{V},\mathbf{EQ},\mathbf{NE})$ be a tracking set. Moreover, let $U = \{ x_i \in \mathbf{V} \mid \neg\exists x_j \in \mathbf{V} : j < i \}$ be the set of minimal allocated free variables and *sel* be a fixed selector. Then the *kernel* of $\mathbf{Track}_k(\mathbf{V},\mathbf{EQ},\mathbf{NE})$ is defined as the symbolic heap (which is itself contained in $\mathbf{Track}_k(\mathbf{V},\mathbf{EQ},\mathbf{NE})$)

$$\varphi \;\triangleq\; \mathbf{Vars}^{\preccurlyeq}(\varphi) \mathbin{:} \{\{ x.sel \mapsto 0 \mid x \in U \}\} \mathbin{:} \mathbf{EQ} \mathbin{:} \mathbf{NE}.$$

Notice that we have to take the minimal (with respect to their index) set of allocated free variables because introducing a points-to assertion for every variable in $\mathbf{V}$ might lead to unsatisfiable kernels due to points-to assertions with equal left-hand sides. There are at most $2^{2\cdot(k+1)^2+k}$ tracking sets. Since there is a

[2] This approach is taken in [2] rather than providing a syntactic definition of the closure.

one-to-one correspondence between tracking sets and their kernels, we use the set of all kernels as the set of states of a heap automaton.

Definition 11.29 (Tracking Automaton) The *tracking automaton* of a tracking set $\mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$ is defined as $\mathcal{A}_{\mathsf{track}} \triangleq \left\langle \mathcal{Q}, \mathbf{FV}^{\leq k}, \mathcal{F}, \Delta \right\rangle$, where

- the set of states is $\mathcal{Q} \triangleq \{ \varphi \mid \varphi \text{ is the kernel of a tracking set} \}$,
- the set of final states is $\mathcal{F} \triangleq \mathcal{Q} \cap \mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$, and
- the transition relation Δ is given by

$$\varphi(\psi_1, \ldots, \psi_n) \to \psi \text{ iff } [\![\varphi(\psi_1, \ldots, \psi_n)]\!] \in \mathbf{Track}_k(\mathsf{alloc}(\psi), \mathbf{EQ}(\psi), \mathbf{NE}(\psi)).$$

A detailed proof that $\mathcal{A}_{\mathsf{track}}$ satisfies the compositionality property and thus is indeed a heap automaton is found in [19, Lemma 17]. Furthermore, since for every symbolic heap φ of rank zero, we have $[\![\varphi]\!] = \varphi$ (cf. Definition 11.5), it follows immediately that $[\![\mathbf{L}(\mathcal{A}_{\mathsf{track}})]\!] = \mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$. In summary:

Lemma 11.30 For all $k \in \mathbb{N}$ and all tracking sets $\mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$, there exists a heap automaton $\mathcal{A}_{\mathsf{track}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{2 \cdot (k+1)^2 + k}$ states such that $[\![\mathbf{L}(\mathcal{A}_{\mathsf{track}})]\!] = \mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$.

We remind the reader that robustness properties, such as $[\![\mathbf{L}(\mathcal{A}_{\mathsf{track}})]\!]$, are defined as subsets of $\mathbf{SHFV}_0^{\leq k}$, i.e., they consist of symbolic heaps without predicate calls. We thus reason about *unfoldings* of symbolic heaps. This is not a restriction due to the compositionality property of heap automata (Definition 11.15). In particular, Corollary 11.22 enables us to prove that a symbolic heap with predicate calls is robust in the sense that all of its unfoldings are robust.

11.2.2 Satisfiability

Next, we address the satisfiability problem for symbolic heaps (see Definition 10.5). Let us first observe that a symbolic heap φ of rank zero is *unsatisfiable* if and only if its closure $\overline{\varphi}$ contains an inconsistent inequality of the form $E \neq E$; by Definition 11.24, it even contains the inequality $0 \neq 0$. The set of all of these symbolic heaps can be characterized by a heap automaton. In fact, it suffices to employ the tracking automaton $\mathcal{A}_{\mathsf{track}}$ introduced in Definition 11.29 except that the set $\mathcal{F}$ of final states is set to

$$\mathcal{F} \triangleq \{ \varphi \in \mathcal{Q}_{\mathcal{A}_{\mathsf{track}}} \mid (0 \neq 0) \in \mathbf{NE}(\varphi) \}.$$

A heap automaton accepting the complement is constructed analogously by choosing all states that do *not* contain the inequality $0 \neq 0$. Consequently:

Theorem 11.31 ([2; 19, Theorem 3]) For all $k \in \mathbb{N}$, there exists a heap automaton $\mathcal{A}_{\mathsf{sat}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{2\cdot(k+1)^2+k}$ states such that

$$[\![\mathbf{L}(\mathcal{A}_{\mathsf{sat}})]\!] \;=\; \left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \varphi \text{ is satisfiable} \right\}.$$

By putting $\mathcal{A}_{\mathsf{sat}}$ together with Corollary 11.19, we immediately obtain a decision procedure for the satisfiability problem: To decide whether the symbolic heap φ is satisfiable with respect to SID Ψ, it suffices to apply Algorithm 1 to the SID $\Psi \cup \{ \boldsymbol{P} \Leftarrow \varphi \}$, the fresh predicate symbol $\boldsymbol{P}$, and the heap automaton $\mathcal{A}_{\mathsf{sat}}$. Regarding complexity, the heap automaton $\mathcal{A}_{\mathsf{sat}}$ has at most $2^{2\cdot(k+1)^2+k}$ many states. Hence, Algorithm 1 runs on the provided input in time

$$\mathcal{O}\left((\|\varphi\| + \|\Psi\|) \cdot 2^{(2\cdot k^2+k)^{m+1}} \cdot \|\Delta_{\mathcal{A}_{\mathsf{sat}}}\| \right),$$

where membership in $\Delta_{\mathcal{A}_{\mathsf{sat}}}$ is decidable in polynomial time as it amounts to computing the closure of a symbolic heap of rank zero and comparing the resulting sets of allocated variables, equalities, and inequalities. Overall, we thus obtain an *exponential-time decision procedure*.

In fact, if the number of free variables k is bounded by a constant, we also obtain a decision procedure in NP: It suffices to guess an unfolding tree t of height at most $\|\mathcal{Q}_{\mathcal{A}_{\mathsf{sat}}}\|$—which is now a constant—and then check whether $\mathcal{A}_{\mathsf{sat}}$ accepts t.[3] This is in line with results by Brotherston et al. [Bro+14], where the satisfiability problem for symbolic heaps is shown to be ExpTime-complete in general and NP-complete if the number of free variables k is bounded. In fact, their complexity results even hold for the following special case:

Definition 11.32 (Restricted Satisfiability Problem (RSH-SAT) [Bro+16]) Given an SID Ψ in which no rule contains points-to assertions and a predicate call $\boldsymbol{P}(\vec{x})$, is $\boldsymbol{P}(\vec{x})$ satisfiable?

11.2.3 Establishment

A symbolic heap φ is *established* [IRS13, Definition 5] if every existentially quantified variable is "eventually allocated". That is, every variable of every unfolding ψ of φ is either equal to 0, equal to a free variable in $\mathbf{Vars}(\psi)$, or contained in $\mathsf{alloc}(\psi)$. Establishment is a natural property of symbolic heaps that specify

[3] A proof is found in [19, Lemma 24]; the approach is similar to the proof of Lemma 11.34.

data structures. For instance, all unfoldings of the SID Ψ_{asll} in Example 11.4 are established. Similarly, all SIDs considered in Example 10.17 describe established symbolic heaps. Furthermore, establishment is occasionally required to obtain fragments of symbolic heaps with a decidable entailment problem [IRS13; IRV14; 17; TK15] (see also Section 12.3). Iosif, Rogalewicz, and Simacek [IRS13] already sketched a decision procedure for establishment.

In this section, we show that establishment can be checked with heap automata and analyze its complexity.

Theorem 11.33 ([2; 19, Theorem 4]) For all $k \in \mathbb{N}$, there exists a heap automaton $\mathcal{A}_{\mathsf{est}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{2\cdot(k+1)^2+k+1}$ states such that

$$[\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!] \;=\; \left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \varphi \text{ is established} \right\}.$$

Proof. For a symbolic heap φ of rank zero, we write $\mathsf{est}(\varphi)$ if φ is established. That is, $\mathsf{est}(\varphi)$ holds if and only if

$$\forall y \in \mathbf{BV}(\varphi)\colon \;\; y \in \mathsf{alloc}_k(\varphi) \;\text{ or }\; \exists E \in \mathbf{Vars}\,(\varphi) \cup \{\,0\,\} : (y = E) \in \mathbf{EQ}(\overline{\varphi}).$$

The main idea for constructing a suitable heap automaton $\mathcal{A}_{\mathsf{est}}$ is to verify establishment of symbolic heaps while running the tracking automaton $\mathcal{A}_{\mathsf{track}}$ (cf. Definition 11.29) in parallel to account for relationships between parameters of predicate calls. To this end, an additional flag $\mathrm{p} \in \{\,\mathsf{false}, \mathsf{true}\,\}$ is attached to each state of $\mathcal{A}_{\mathsf{track}}$. This flag indicates whether the establishment condition is known to be violated by some unfolding of a predicate call ($\mathrm{p} = \mathsf{false}$) or holds in all unfoldings considered so far ($\mathrm{p} = \mathsf{true}$). Formally, we define the automaton $\mathcal{A}_{\mathsf{est}} \triangleq \left\langle Q, \mathbf{FV}^{\leq k}, \mathcal{F}, \Delta \right\rangle$, where

- the set of states is $Q \triangleq Q_{\mathcal{A}_{\mathsf{track}}} \times \{\,\mathsf{false}, \mathsf{true}\,\}$,
- the set of final states is $\mathcal{F} \triangleq Q_{\mathcal{A}_{\mathsf{track}}} \times \{\,\mathsf{true}\,\}$, and
- the set of transition rules Δ is given by:

$$\begin{aligned} & \varphi(\langle\, \mathrm{q}_1, \mathrm{p}_1 \,\rangle, \ldots, \langle\, \mathrm{q}_n, \mathrm{p}_n \,\rangle) \to \langle\, \mathrm{q}, \mathrm{p} \,\rangle \\ \text{iff} \quad & \varphi(\mathrm{q}_1, \ldots, \mathrm{q}_n) \to \mathrm{q} \in \Delta_{\mathcal{A}_{\mathsf{track}}} \\ & \text{and } p = \mathrm{p}_1 \wedge \ldots \wedge \mathrm{p}_n \wedge \mathsf{est}([\![\varphi(\mathrm{q}_1, \ldots, \mathrm{q}_n)]\!]). \end{aligned}$$

A formal correctness proof of this construction is found in [19, Lemma 26 and Lemma 27]. Regarding the number of states, we have

$$\|\mathcal{Q}\| \;\leq\; 2 \cdot \|\mathcal{Q}_{\mathcal{A}_{\mathsf{track}}}\| \;\leq\; 2^{2\cdot(k+1)^2+k+1}. \qquad \square$$

As for the tracking automaton, it suffices to swap the final and non-final states of $\mathcal{A}_{\mathsf{est}}$ to obtain a heap automaton accepting the complement of $[\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$. Consequently, by Corollary 11.19, we obtain an ExpTime decision procedure for the *establishment problem* (SH-EST for short): Given an SID Ψ and a symbolic heap φ, decide whether every unfolding $\psi \in \mathbf{Unf}_\Psi(\varphi)$ is established. The complexity can be improved if the number k of free variables is bounded:

Lemma 11.34 ([2; 19, Lemma 5]) The decision problem SH-EST is in coNP if the maximal number k of free variables is bounded.

Proof. Let (Ψ, φ) be an instance of the establishment problem. Moreover, let $n = \|\Psi\| + \|\varphi\|$. Clearly, the maximal number m of predicate calls in φ and Ψ is bounded by n, i.e., $m \leq n$. Furthermore, let

$$c \;\triangleq\; \|\mathcal{Q}_{\mathcal{A}_{\mathsf{est}}}\| \;\leq\; 2^{2\cdot(k+1)^2+k+1}.$$

be the number of states of the heap automaton $\mathcal{A}_{\mathsf{est}}$ (see Theorem 11.33). Since the number of free variables k is bounded, c is a constant.

Now, let us denote by $\mathbf{Trees}_\Psi(\varphi)^{\leq c}$ the set of all unfolding trees of φ with respect to SID Ψ of height at most c. Each of these trees t is of size

$$\|t\| \leq m^c \leq n^c$$

That is, the size of t is polynomial in n. To prove membership of SH-EST in coNP we claim that:

1. $(\Psi, \varphi) \in$ SH-EST iff $\forall t \in \mathbf{Trees}_\Psi(\varphi)^{\leq c}\colon t \in [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$, and
2. it is decidable in polynomial time whether $t \in [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$ holds.

To prove the first claim, consider the following:

- If $(\Psi, \varphi) \in$ SH-EST, then, for every unfolding tree $t \in \mathbf{Trees}_\Psi(\varphi)$, its composition $[\![t]\!]$ is established. By Theorem 11.33, we have $t \in [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$. In particular, this holds for all

$$t \in \mathbf{Trees}_\Psi(\varphi)^{\leq c} \subseteq \mathbf{Trees}_\Psi(\varphi).$$

- Towards a contradiction, assume $(\Psi, \varphi) \notin \mathsf{SH\text{-}EST}$, but $t \in [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$ holds for all $t \in \mathbf{Trees}_\Psi(\varphi)^{\leq c}$. Then there exist unfolding trees in $t \in \mathbf{Trees}_\Psi(\varphi)$, of height larger than c such that $[\![t]\!]$ is not established. By Theorem 11.33, we have $t \notin [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$. Let t be a minimal (with respect to the number of elements $|\mathsf{dom}(t)|$) such tree. By a standard pumping argument, there exists a position $v \in t$ and a prefix u of v which are assigned the same state by heap automaton $\mathcal{A}_{\mathsf{est}}$. However, this means that the tree t' obtained from substituting at position u the subtree $t|_u$ by $t|_v$ is an even smaller unfolding tree in $\mathbf{Trees}_\Psi(\varphi)$ with $t' \notin [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$. This contradicts the minimality of t.

To prove the second claim, notice that we can verify whether $t \in [\![\mathbf{L}(\mathcal{A}_{\mathsf{est}})]\!]$ as follows: For every function $\rho\colon \mathsf{dom}(t) \to \mathcal{Q}_{\mathcal{A}_{\mathsf{est}}}$ mapping each node of tree t to a state of heap automaton $\mathcal{A}_{\mathsf{est}}$ and every position $u \in \mathsf{dom}(t)$ with $\mathsf{rank}(t(u)) = \ell$, we check whether

$$t(u)(\rho(u\,1), \ldots, \rho(u\,\ell)) \to \rho(u) \;\in\; \Delta_{\mathcal{A}_{\mathsf{est}}}.$$

Since the size of ρ is polynomial in n, namely $\|\rho\| \leq c \cdot n^c$, this procedure can be performed in polynomial time; as such in

$$\mathcal{O}\left(c \cdot n^{2c} \cdot \|\Delta_{\mathcal{A}_{\mathsf{est}}}\|\right). \qquad \square$$

In fact, both complexity bounds are asymptotically optimal.

Theorem 11.35 (Complexity of Establishment [2; 19, Theorem 5]) The decision problem SH-EST is ExpTime-complete in general and coNP-complete if the maximal number k of free variables is bounded.

Proof. We have already described decision procedures for SH-EST which yield upper complexity bounds in ExpTime and coNP, respectively.

To derive the corresponding lower bounds, we reduce the complement $\overline{\mathsf{RSH\text{-}SAT}}$ of the restricted satisfiability problem (Definition 11.32) to SH-EST. Given an instance $\langle \Psi, \boldsymbol{P}(\vec{y}) \rangle$ of $\overline{\mathsf{RSH\text{-}SAT}}$, we define an instance (Ψ, φ) of SH-EST, where, for some fresh variable z, the symbolic heap φ is given by

$$\varphi \;\triangleq\; x_1 \mid \boldsymbol{P}(\vec{y}) \mid x_1 = 0 \mid z \neq 0.$$

Variable z is neither allocated nor occurs in $\boldsymbol{P}(\vec{y})$. Consequently, φ is established iff $z = x_1$ holds in every unfolding of φ. We then have:

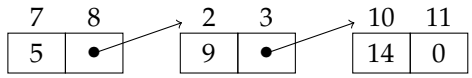


Figure 11.5: A heap $\mathfrak{h}$ in which location 11 is reachable from location 7.

$\langle \Psi, \varphi \rangle \in \mathsf{SH\text{-}EST}$
iff $[\![$ Definition of SH-EST $]\!]$
$\forall \psi \in \mathbf{Unf}_\Psi(\varphi)$: ψ is established
iff $[\![$ By construction of φ $]\!]$
$\forall \psi \in \mathbf{Unf}_\Psi(\varphi)$: $(z = x_1) \in \overline{\psi}$
iff $[\![$ Definition 11.24 $]\!]$
$\forall \psi \in \mathbf{Unf}_\Psi(\varphi)$: $(0 \neq 0) \in \overline{\psi}$
iff $[\![$ Lemma 11.26 $]\!]$
$\forall \psi \in \mathbf{Unf}_\Psi(\varphi)$: ψ is unsatisfiable
iff $[\![$ Definition of $\overline{\mathsf{RSH\text{-}SAT}}$ $]\!]$
$\langle \Psi, \boldsymbol{P}(\vec{y}) \rangle \in \overline{\mathsf{RSH\text{-}SAT}}$.

Since the restricted satisfiability problem for symbolic heaps is ExpTime-complete in general, we conclude that the establishment problem is ExpTime-hard. Analogously, since the former problem is NP-complete for a bounded number k of free variables, we conclude that the establishment problem is coNP-hard for bounded k. □

11.2.4 Reachability

Another family of robustness properties is concerned with reachability problems which have been characterized as "one of the most treacherous passes in pointer verification" [BCO04]. For example, common questions include "is a specified heap free of garbage, i.e., is every location of every unfolding of a symbolic heap reachable from a location assigned to a program variable?" or "is every unfolding of a symbolic heap acyclic?" Intuitively, a location ℓ is reachable from a location s in a heap $\mathfrak{h}$ if the directed graph underlying $\mathfrak{h}$ contains a path from ℓ to s. In this graph, all locations corresponding to the same record are considered as a single node. Hence, in Figure 11.5, both location 10 and 11 are reachable from locations 7 and 8 if we assume that every record has two selectors.

In the above questions, however, we ask for reachability in *all* unfoldings of a symbolic heap. Hence, we express reachability syntactically in terms of a path of points-to assertions in an assertion rather than a path of edges in a graph:

Definition 11.36 (Definite Reachability) Let φ be a symbolic heap of rank zero. Then E_2 is *definitely reachable* from E_1 in φ, written $E_1 \rightsquigarrow^{\varphi} E_2$, iff there exist expressions E', E'' and a selector *sel* such that $(E_1 = E') \in \mathbf{EQ}(\overline{\varphi})$ and

- $(E_2 = E'') \in \mathbf{EQ}(\overline{\varphi})$ and $E'.sel \mapsto E'' \in \mathbf{PT}(\varphi)$, or
- $E'.sel \mapsto E''$ and $E'' \rightsquigarrow^{\varphi} E_2$.

Notice that the definite reachability relation is transitive, but not reflexive. That is, for two expressions to be reachable from one another, there must be at least one pointer between them. Consequently, for the symbolic heap

$$\varphi \;\triangleq\; x_1, x_2 \mid x_1.sel \mapsto x_1, x_2.sel \mapsto y, y.sel \mapsto z,$$

we have both $x_2 \rightsquigarrow^{\varphi} z$ and $x_1 \rightsquigarrow^{\varphi} x_1$. Furthermore, x_2 is reachable from itself for *some* stack-heap pairs $(\mathfrak{s}, \mathfrak{h})$, e.g., if $\mathfrak{s}(z) = \mathfrak{s}(x_2)$. However, this is not the case for all stack-heap pairs satisfying φ; in fact, $x_2 \rightsquigarrow^{\varphi} x_2$ does *not* hold. While reachability introduced through unallocated variables, such as z in the above example, is not detected, the absence (or existence) of such variables can be checked due to Theorem 11.33.

For any fixed constant $k \in \mathbb{N}$, let us denote by $\rightsquigarrow^{\varphi}_{k}$ the restriction of the definite reachability relation to free variables in $\mathbf{Vars}_k$ and 0, i.e.,

$$\rightsquigarrow^{\varphi}_{k} \;\triangleq\; \rightsquigarrow^{\varphi} \;\cap\; ((\mathbf{Vars}_k \cup \{0\}) \times (\mathbf{Vars}_k \cup \{0\})).$$

Definite reachability is a robustness property:

Theorem 11.37 ([2; 19, Theorem 6]) Let $k \in \mathbb{N}$. Moreover, let

$$\mathbf{R} \subseteq (\mathbf{Vars}_k \cup \{0\}) \times (\mathbf{Vars}_k \cup \{0\})$$

be a binary relation capturing reachability relationships between free variables and 0. Then there exists a heap automaton $\mathcal{A}_{\mathsf{reach}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{3 \cdot k^2 + k}$ states such that

$$[\![\mathbf{L}(\mathcal{A}_{\mathsf{reach}})]\!] \;=\; \left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \rightsquigarrow^{\varphi}_{k} = \mathbf{R} \right\}.$$

Proof. The main idea is to extend the heap automaton $\mathcal{A}_{\mathsf{track}}$ (cf. Definition 11.29) such that it additionally stores reachability information. In fact, it suffices to enrich tracking sets (Definition 11.27) by relations

$$\mathbf{S} \subseteq (\mathbf{Vars}_k \cup \{\, 0 \,\}) \times (\mathbf{Vars}_k \cup \{\, 0 \,\})$$

which collect all reachability relationships between free variables and 0. Moreover, we have to update the symbolic heap representing the kernel of a tracking set (Definition 11.27). Apart from the final states, $\mathcal{A}_{\mathsf{reach}}$ is then constructed analogously to $\mathcal{A}_{\mathsf{track}}$.

More precisely, given a relation $\mathbf{S}$ as above, the *reachability sensitive tracking set* $\mathbf{Track}_k(\mathbf{S}, \mathbf{V}, \mathbf{EQ}, \mathbf{NE})$ is defined as

$$\left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \varphi \in \mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE}) \text{ and } \rightsquigarrow_k^{\varphi} = \mathbf{S} \right\}.$$

To extend the kernel φ of a tracking set $\mathbf{Track}_k(\mathbf{V}, \mathbf{EQ}, \mathbf{NE})$ to a kernel of a reachability sensitive tracking set $\mathbf{Track}_k(\mathbf{S}, \mathbf{V}, \mathbf{EQ}, \mathbf{NE})$, let us assume that there are additional selectors $sel_1, \ldots, sel_{k+1}$ (as long as there are at least two selectors, we can alternatively encode these as a linked list). It then suffices to add a points-to assertion $x_i.sel_j \mapsto x_j$ whenever $\langle\, x_i, x_j \,\rangle \in \mathbf{S}$. Moreover, if $\langle\, x_i, 0 \,\rangle \in \mathbf{S}$, we add the points-to assertion $x_i.sel_{k+1} \mapsto 0$. Hence, our updated kernel is given by the closure of the symbolic heap

$$\begin{aligned} \psi \;\triangleq\; \mathbf{Vars}^{\preccurlyeq}(\varphi) \,\mathbin{¦}\, \mathbf{PT}(\varphi) \Cup \{\{\, x_i.sel_j \mapsto x_j \mid \langle\, x_i, x_j \,\rangle \in \mathbf{S} \,\}\} \\ \Cup \{\{\, x_i.sel_{k+1} \mapsto 0 \mid \langle\, x_i, 0 \,\rangle \in \mathbf{S} \,\}\} \,\mathbin{¦}\, \mathbf{EQ}(\varphi) \,\mathbin{¦}\, \mathbf{NE}(\varphi). \end{aligned}$$

To define the heap automaton $\mathcal{A}_{\mathsf{reach}} \triangleq \left\langle\, \mathcal{Q}, \mathbf{FV}^{\leq k}, \mathcal{F}, \Delta \,\right\rangle$, we then choose the set of states $\mathcal{Q}$ as the set of all kernels of reachability sensitive tracking sets. The set of final states is $\mathcal{F} \triangleq \mathcal{Q} \cap \mathbf{Track}_k(\mathbf{R}, \_, \_, \_)$, i.e., it consists all kernels of reachability sensitive tracking sets over the fixed relation $\mathbf{R}$. Finally, the transition relation Δ is given by $\varphi(\psi_1, \ldots, \psi_n) \to \psi$ iff

$$[\![\varphi(\psi_1, \ldots, \psi_n)]\!] \in \mathbf{Track}_k(\rightsquigarrow_k^{\psi}, \mathsf{alloc}(\psi), \mathbf{EQ}(\psi), \mathbf{NE}(\psi)).$$

A formal correctness proof is found in [19, Appendix A.15]. Since we effectively added another binary relation over $\mathbf{Vars}_k \cup \{\, 0 \,\}$ to the notion of tracking sets, the number of states in $\mathcal{Q}$ is bounded by:

$$\|\mathcal{Q}\| \;\leq\; 2^{(k+1)^2} \cdot \|\mathcal{Q}_{\mathcal{A}_{\mathsf{track}}}\| \;\leq\; 2^{3 \cdot (k+1)^2 + k}. \qquad \square$$

Now that we have a heap automaton to check reachability relationships, let us consider the related *reachability problem for symbolic heaps* (SH-Reach for short): Given an SID Ψ, a symbolic heap φ, and expressions E_1, E_2, decide whether, for all $\psi \in \mathbf{Unf}_\Psi(\varphi)$, we have $E_1 \rightsquigarrow^\psi E_2$.

Theorem 11.38 (Complexity of Reachability [2; 19, Theorem 7]) The decision problem SH-Reach is ExpTime-complete in general and coNP-complete if the number k of free variables is bounded.

Proof. We first observe that—as all other heap automata presented in this section—the complement of heap automaton $\mathcal{A}_{\mathsf{reach}}$ can be computed by swapping final and non-final states. Membership in ExpTime then follows from the upper complexity bound for Algorithm 1, the size of the state space of $\mathcal{A}_{\mathsf{reach}}$ (see Theorem 11.37), and the fact that membership in a tracking set is decidable in polynomial time. Membership in coNP for bounded k is shown analogously to Lemma 11.34. To prove both lower bounds, we reduce the complement $\overline{\text{RSH-SAT}}$ of the restricted satisfiability problem to SH-REACH. Formally, let $\langle \Psi, \boldsymbol{P}(\vec{y_1}) \rangle$ be an instance of $\overline{\text{RSH-SAT}}$. Moreover, let $\mathbf{Sel} = sel_1, \ldots, sel_n$ be the sequence of selectors under consideration. We then define an instance $\langle \Psi, \varphi, x_1, x_2 \rangle$ of SH-REACH as follows:

$$\varphi \triangleq x_1, x_2 \mid x_1.sel_1 \mapsto 0, \ldots, x_1.sel_n \mapsto 0 \mid \boldsymbol{P}(\vec{y}) \mid x_2 \neq 0.$$

Assume that $\langle \Psi, \varphi, x_1, x_2 \rangle \in$ SH-REACH. Then, for every unfolding ψ of φ, we have $x_1 \rightsquigarrow^\psi x_2$. We distinguish two cases: First, if ψ contains more than n points-to assertions with x_1 on the left-hand side, then ψ is immediately unsatisfiable due to double allocation. Second, if ψ contains exactly n such points to assertions, namely the ones in φ, then, by definition of $\rightsquigarrow^\psi$ (see Definition 11.36), two cases are possible:

1. $(x_2 = 0) \in \mathbf{EQ}(\overline{\psi})$. Then ψ is unsatisfiable as $(x_2 \neq 0) \in \mathbf{EQ}(\varphi)$.
2. There exists a variable z such that $(z = 0) \in \mathbf{EQ}(\overline{\psi})$ and $z \rightsquigarrow^\psi x_2$. In this case, z is both equal to 0 and allocated as it occurs on the left-hand side of a points-to assertion. Thus, ψ is unsatisfiable.

In total, every unfolding ψ of φ is unsatisfiable. Hence, φ is unsatisfiable.

Conversely, if φ is unsatisfiable, then every unfolding ψ of φ is unsatisfiable. Consequently, $\overline{\psi}$ contains some inconsistent inequality $E \neq E$. By Definition 11.24, we have $(x_2 = 0) \in \mathbf{EQ}(\overline{\psi})$. By Definition 11.36, this means that, for every unfolding ψ of φ, we have $x_1 \rightsquigarrow^\psi x_2$. Hence, $\langle \Psi, \varphi, x_1, x_2 \rangle \in$ SH-REACH. □

11.2.5 Garbage-Freedom

Similar to the tracking automaton $\mathcal{A}_{\mathsf{track}}$, the heap automaton $\mathcal{A}_{\mathsf{reach}}$ is a useful ingredient for reasoning about robustness properties. For instance, it can easily be modified to check whether a symbolic heap is *garbage-free*. As it is common in programming languages such as Java, we consider the heap to be garbage-free if every allocated location is reachable from some program variable. For symbolic heaps, we usually identify program variables with free variables.

Hence, a symbolic heap φ of rank zero is garbage-free, written $\mathsf{gfree}(\varphi)$, iff

$$\forall y \in \mathbf{BV}(\varphi) \colon\ \exists x \in \mathbf{Vars}(\varphi) : (x = y) \in \mathbf{EQ}(\varphi) \text{ or } x \rightsquigarrow^{\varphi} y.$$

For instance, recall from Example 11.6 the symbolic heap

$$\varphi \ = \ x, y \mid x.next \mapsto z, z.next \mapsto z', z'.next \mapsto y \mid x \neq z, z' \neq y.$$

Since both $x \rightsquigarrow^{\varphi} z$ and $x \rightsquigarrow^{\varphi} z'$ hold, φ is garbage-free, i.e., $\mathsf{gfree}(\varphi)$ holds.

Theorem 11.39 ([2; 19, Lemma 6]) For all $k \in \mathbb{N}$, there exists a heap automaton $\mathcal{A}_{\mathsf{gfree}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{3\cdot(k+1)^2+k+1}$ states such that

$$[\![\mathbf{L}(\mathcal{A}_{\mathsf{gfree}})]\!] \ = \ \left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \mathsf{gfree}(\varphi) \right\}.$$

Proof. $\mathcal{A}_{\mathsf{gfree}}$ is constructed analogously to the automaton $\mathcal{A}_{\mathsf{est}}$ in the proof of Theorem 11.33. However, rather than running the tracking automaton $\mathcal{A}_{\mathsf{track}}$ and checking whether $\mathsf{est}(\varphi)$ holds in parallel, we use the heap automaton $\mathcal{A}_{\mathsf{reach}}$ (cf. Theorem 11.37) and check whether $\mathsf{gfree}(\varphi)$ holds. For further details, see [19, Appendix A.16]. □

To check whether a symbolic heap contains garbage, we have to discharge the *garbage-freedom problem* (SH-Gfree for short): Given an SID Ψ and a symbolic heap φ, decide whether every unfolding $\psi \in \mathbf{Unf}_{\Phi}(\varphi)$ is garbage-free.

Theorem 11.40 (Complexity of Garbage-Freedom [2; 19, Theorem 8]) The decision problem SH-Gfree is ExpTime-complete in general and in coNP-complete if the number of free variables is bounded.

Proof. Analogous to Theorem 11.35. □

11.2.6 Acyclicity

Automatic termination proofs frequently rely on the acyclicity of data structures employed by a program. That is, they assume that no variable is reachable from

itself. In fact, Zanardini and Genaim [ZG14] claim that "proving termination needs acyclicity, unless program-specific or non-automated reasoning is performed." Our last example of robustness properties is thus concerned with proving acyclicity of symbolic heaps. Formally, a symbolic heap φ of rank zero is *weakly acyclic*, written $\mathsf{acyclic}(\varphi)$, iff

$$\forall x \in (\mathbf{Vars}(\varphi) \cup \mathbf{BV}(\varphi)) \colon \quad \neg(x \rightsquigarrow^{\varphi} x).$$

The above notion of acyclicity is weak in the sense that some stack-heap pairs satisfying a symbolic heap might still contain cycles. For example,

$$\varphi \triangleq x \mid x.sel \mapsto y$$

is weakly acyclic, but admits cyclic stack-heap pairs if x and y are aliases. However, weak acyclicity successfully prevents cycles in the absence of dangling pointers. In particular, this is the case for established symbolic heaps—a robustness property which we considered in Section 11.2.3.

Theorem 11.41 ([2; 19, Lemma 7]) For all $k \in \mathbb{N}$, there exists a heap automaton $\mathcal{A}_{\mathsf{acyclic}}$ over $\mathbf{FV}^{\leq k}$ with at most $2^{3\cdot(k+1)^2+k+1}$ states such that

$$[\![\mathbf{L}(\mathcal{A}_{\mathsf{acyclic}})]\!] \;=\; \left\{ \varphi \in \mathbf{SHFV}_0^{\leq k} \;\middle|\; \mathsf{acyclic}(\varphi) \right\}$$

Proof. $\mathcal{A}_{\mathsf{acyclic}}$ is constructed analogously to the automaton $\mathcal{A}_{\mathsf{est}}$ in the proof of Theorem 11.33. However, rather than running the tracking automaton $\mathcal{A}_{\mathsf{track}}$ and checking whether $\mathsf{est}(\varphi)$ holds in parallel, we use the heap automaton $\mathcal{A}_{\mathsf{reach}}$ (cf. Theorem 11.37) and check whether $\mathsf{acyclic}(\varphi)$ holds. For further details, see [19, Appendix A.19]. □

To check whether a symbolic heap is (weakly) acyclic, we have to discharge the *acyclicity problem* (SH-AC for short): Given an SID Ψ and a symbolic heap φ, decide whether every unfolding $\psi \in \mathbf{Unf}_{\Phi}(\varphi)$ is weakly acyclic.

Theorem 11.42 (Complexity of Weak Acyclicity [2; 19, Theorem 9]) The decision problem SH-AC is ExpTime-complete in general and in coNP-complete if the number of free variables is bounded.

Proof. Analogous to Theorem 11.35. □

11.2.7 Overview

To conclude, Table 11.1 summarizes the decision problems considered throughout this section together with their complexity in general and if the number of

Decision Problem	Description	Complexity	for $\mathbf{SHFV}^{\leq k}$
RSH-SAT	(restricted) satisfiability	ExpTime	NP
SH-EST	establishment	ExpTime	coNP
SH-Reach	reachability	ExpTime	coNP
SH-Gfree	garbage-freedom	ExpTime	coNP
SH-AC	(weak) acyclicity	ExpTime	coNP

Table 11.1: Summary of the complexity of deciding robustness properties.

free variables k is bounded. While all of the decision problems turned out to be ExpTime-complete in general, the complexity for a bounded number of free variables depends on the robustness property in question: If the existence of one suitable unfolding suffices to satisfy a given property, the corresponding decision problem becomes NP-complete. Conversely, if all unfoldings have to satisfy a property, the corresponding decision problem becomes coNP-complete.

11.3 Implementation

Our algorithmic framework has been implemented in a tool called *Heap Automata for Reasoning about Robustness of Symbolic Heaps* (Harrsh for short) in cooperation with Jens Katelaan and Florian Zuleger at the Technical University of Vienna. The tool, its source code, and all experiments are available under MIT license on GitHub.[4] At the moment of writing this thesis,[5] the implementation consists of roughly 13000 lines of Scala code.

Harrsh implements both refinement of SIDs in the sense of Theorem 11.17 as well as the decision procedure via on-the-fly refinement with an integrated emptiness check presented in Algorithm 1. Whenever checking a robustness property yields that an SID is not robust, it is also able to generate a witness, i.e., a concrete unfolding, for the property violation.

Each of the aforementioned algorithms is implemented with respect to a generic interface for heap automata. Adding a decision procedure for a new robustness property thus only requires a definition of the corresponding heap

[4] https://github.com/katelaan/harrsh/

[5] i.e., commit `be8fbf6354c786d13c357c87b9a717d96458279b` on 04.05.2019.

automaton. In particular, heap automata for all robustness properties and their complements presented in Section 11.2 are already defined in Harrsh.

Harrsh accepts SIDs either in its own input format, the input format of the separation logic prover Cyclist [Bro+14], or an extension of the SMT-LIB format to account for separation logic (cf. [Ios+]). The latter also serves as the input format of the separation logic competition SL-COMP[6] in which Harrsh has participated since 2018. In the satisfiability checking category with user-supplied SIDs, i.e., `qf_shid_sat`, the generic approach based on heap automata implemented in Harrsh turned out to be competitive with dedicated state-of-the-art satisfiability checkers, such as Cyclist [Bro+14], Songbird [Ta+16; Ta+19], or Sleek [CDG11]. In particular, in the 2018 edition of SL-COMP, it was the fastest tool for satisfiability checking which did not produce any wrong results.[7] Moreover, since 2019, Harrsh is also capable of deciding entailments (see Section 12.1). In its first participation in the entailment checking category in the 2019 edition of SL-COMP [13] for symbolic heaps with user-supplied SIDs, i.e., `qf_shid_entl`, Harrsh competed successfully among the top three tools (out of seven tools in total over both runs) with respect to both runtime and number of solved entailment queries.

To evaluate the performance of Harrsh with respect to deciding the robustness properties in Section 11.2, which—apart from satisfiability—are not supported by other tools, we considered a large collection of benchmarks that is distributed with Cyclist. This collection includes the following problem sets:

1. A set of handwritten data structure specifications compiled from the separation logic literature.

2. 45945 problem instances that have been automatically generated from source code by the specification inference tool Caber [BG14].

All experiments reported below were performed on an Intel Core i5-3317U at 1.70GHz with 4GB of RAM.

For the standard specifications in the first problem set, Harrsh runs in approximately 300ms to check *all* robustness properties on *all* specifications, i.e., a total of 45 problem instances. This is not a surprise, because the SIDs under consideration have been carefully handcrafted. They are thus expected to be robust. For the more realistic set of automatically inferred SIDs in the second problem set, Table 11.2 on page 321 shows the accumulated runtime of Harrsh on all instances for each robustness property. In each case, Harrsh finished reasoning below 20 seconds in total. This demonstrates the applicability of the heap automaton approach for reasoning about SIDs that occur in practice.

[6] https://sl-comp.github.io/

[7] https://www.irif.fr/~sighirea/sl-comp/18/qf_shid_sat.html

Robustness Property	Definition	Analysis Time (ms)
No points-to asssertions	Example 11.16	7230
Tracking property	Section 11.2.1	11459
Satisfiability	Section 11.2.2	12460
Complement of satisfiability	Section 11.2.2	11980
Establishment	Section 11.2.3	18055
Complement of establishment	Section 11.2.3	17272
Reachability	Section 11.2.4	14897
Garbage-Freedom	Section 11.2.5	18192
Weak acyclicity	Section 11.2.6	18505

Table 11.2: Total analysis time for checking the robustness properties presented in this chapter on the second set of problem instances [2].

This chapter is based on prior publications, namely [17; 3; 20], which are presented, discussed, and extended in the broader context of this thesis.

The most prominent decision problem in the context of separation logic is the *entailment problem* (cf. Definition 10.5 (c)) which is key to Floyd-Hoare style verification techniques. As such, it has been characterized as being "at the foundation of automatic verification based on separation logic" [BDP11]. In fact, all examples on weakest preconditions (or weakest preexpectations) presented in this thesis, e.g., the case studies in Chapter 8, involve solving entailments in order to prove that a proposed invariant is correct or to discharge that a given precondition is covered by the computed weakest precondition.

For symbolic heaps (cf. Section 10.1.1), the entailment problem is the following question: Given an SID Ψ and symbolic heaps φ and ψ, does $\varphi \models_\Psi \psi$ hold, i.e., does every stack-heap pair satisfying φ also satisfy ψ? Formally, that is

$$\varphi \models_\Psi \psi \quad \text{iff} \quad \forall (\mathfrak{s}, \mathfrak{h}) \colon \mathfrak{s}, \mathfrak{h} \models_\Psi \varphi \text{ implies } \mathfrak{s}, \mathfrak{h} \models_\Psi \psi.$$

Notice that the entailment problem for symbolic heaps *cannot* be reduced to the satisfiability problem because symbolic heaps are not closed under negation; a logical implication is thus not definable (cf. Definition 10.6). Furthermore, we remark that every attempt to automatically discharge entailments is necessarily unsound or incomplete because the entailment problem for symbolic heaps with user-supplied SIDs is undecidable in general [Ant+14].

In this chapter, we explore two approaches to discharge entailments for fragments of symbolic heaps. First, in Section 12.1, we consider entailments as robustness properties in the sense of Definition 11.15 and sketch how heap automata are applicable to the entailment problem. Second, in Section 12.2, we discuss how entailments are discharged by syntactic rewriting which is commonly known as fold/unfold reasoning (cf. , for example, [BCO05b; BDP11; Chi+12]). In particular, in Section 12.3, we study the fragment of *graphical* symbolic heaps and present a pragmatic decision procedure that is complete for all entailments encountered by the software model checker Attestor (see Chapter 13).

12.1 Entailments as Robustness Properties

Recall from Lemma 10.24 that a stack-heap pair $(\mathfrak{s}, \mathfrak{h})$ satisfies a symbolic heap φ with respect to an SID Ψ iff $(\mathfrak{s}, \mathfrak{h})$ satisfies some unfolding ϑ of φ. That is,

$$\mathfrak{s}, \mathfrak{h} \models_\Psi \varphi \quad \text{iff} \quad \exists \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \mathfrak{s}, \mathfrak{h} \models \vartheta.$$

To discharge an entailment of the form $\varphi \models_\Psi \psi$, we can—in principle—attempt to construct a heap automaton $\mathcal{A}_\psi$ which accepts all unfoldings with respect to Ψ that entail some unfolding of ψ. That is, $\mathcal{A}_\psi$ accepts the robustness property

$$[\![\mathbf{L}(\mathcal{A}_\psi)]\!] \;=\; \{\, \vartheta \in \mathbf{SHC}_0 \mid \vartheta \models_\Psi \psi \,\}$$

for some reasonable set of SIDs $\mathbf{C}$ including Ψ. Notice that the entailment $\vartheta \models_\Psi \psi$ in the above robustness property is conceptually simpler than a general entailment because its left-hand side is of rank zero; it thus has exactly one unfolding, namely itself. By Corollary 11.22, it is then decidable whether all unfoldings of the symbolic heap $\varphi \in \mathbf{SHC}$ are accepted by $\mathcal{A}_\psi$, i.e., whether $\mathbf{Unf}_\Psi(\varphi) \subseteq [\![\mathbf{L}(\mathcal{A}_\psi)]\!]$ holds. This property is equivalent to checking the entailment $\varphi \models_\Psi \psi$:

$\varphi \models_\Psi \psi$
iff $[\![$ Definition of entailments $]\!]$
$\forall (\mathfrak{s}, \mathfrak{h}) \colon \mathfrak{s}, \mathfrak{h} \models_\Psi \varphi$ implies $\mathfrak{s}, \mathfrak{h} \models_\Psi \psi$
iff $[\![$ Definition of logical implication $]\!]$
$\forall (\mathfrak{s}, \mathfrak{h}) \colon \neg(\mathfrak{s}, \mathfrak{h} \models_\Psi \varphi)$ or $\mathfrak{s}, \mathfrak{h} \models_\Psi \psi$
iff $[\![$ Lemma 10.24 $]\!]$
$\forall (\mathfrak{s}, \mathfrak{h}) \colon \neg(\exists \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \mathfrak{s}, \mathfrak{h} \models \vartheta)$ or $\mathfrak{s}, \mathfrak{h} \models_\Psi \psi$
iff $[\![$ Elementary first-order logic $]\!]$
$\forall \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \forall (\mathfrak{s}, \mathfrak{h}) \colon \mathfrak{s}, \mathfrak{h} \models \vartheta$ implies $\mathfrak{s}, \mathfrak{h} \models_\Psi \psi$
iff $[\![$ Since ϑ contains no predicate calls, we have $\mathfrak{s}, \mathfrak{h} \models \vartheta$ iff $\mathfrak{s}, \mathfrak{h} \models_\Psi \vartheta$ $]\!]$
$\forall \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \forall (\mathfrak{s}, \mathfrak{h}) \colon \mathfrak{s}, \mathfrak{h} \models_\Psi \vartheta$ implies $\mathfrak{s}, \mathfrak{h} \models_\Psi \psi$
iff $[\![$ Definition of entailments $]\!]$
$\forall \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \vartheta \models_\Psi \psi$
iff $[\![\,[\![$ Definition of $[\![\mathbf{L}(\mathcal{A}_\psi)]\!]$ $]\!]\,]\!]$
$\forall \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \vartheta \in [\![\mathbf{L}(\mathcal{A}_\psi)]\!]$
iff $[\![$ Elementary first-order logic $]\!]$
$\mathbf{Unf}_\Psi(\varphi) \;\subseteq\; [\![\mathbf{L}(\mathcal{A}_\psi)]\!]$.

Hence, we can discharge entailments as long as we find suitable heap automata.

Example 12.1 Consider the SID Ψ below which specifies non-empty, null-terminated singly-linked lists with head x:

$$\Psi \triangleq \{ \mathit{sll0} \Leftarrow x \nmid x.\mathit{next} \mapsto 0, \quad \mathit{sll0} \Leftarrow x \nmid x.\mathit{next} \mapsto z \nmid \mathit{sll0}(z) \}$$

Let us construct a heap automaton $\mathcal{A}_{\mathit{sll0}} \triangleq \left\langle \mathcal{Q}, \mathbf{SHFV}^{\leq 1}, \mathcal{F}, \Delta \right\rangle$ over all SIDs in which symbolic heaps have at most one free variable which accepts all symbolic heaps entailing the predicate call $\mathit{sll0}(x)$. That is,

$$[\![\mathbf{L}(\mathcal{A}_{\mathit{sll0}})]\!] = \left\{ \vartheta \in \mathbf{SHFV}_0^{\leq 1} \;\middle|\; \vartheta \models_\Psi \mathit{sll0}(x) \right\}.$$

Intuitively, $\mathcal{A}_{\mathit{sll0}}$ assigns a state to every symbolic heap $\vartheta \in \mathbf{SHFV}_0^{\leq 1}$ of rank zero according to the following two criteria:

1. Does $\vartheta \models_\Psi \mathit{sll0}(x)$ hold? For instance, this is the case for $\vartheta = x \nmid x.\mathit{next} \mapsto 0$, but neither for $\vartheta = x \nmid x \neq 0$ nor $\vartheta = x \nmid z.\mathit{next} \mapsto z$.
2. Does $\vartheta \star x.\mathit{next} \mapsto 0 \models_\Psi \mathit{sll0}(x)$ hold, i.e., can every stack-heap pair satisfying ϑ be extended such that it satisfies $\mathit{sll0}(x)$? For instance, this is the case for $\vartheta = x \nmid x.\mathit{next} \mapsto 0$ (as ϑ becomes unsatisfiable) and $\vartheta = x \nmid x \neq 0$, but not for $\vartheta = x \nmid z.\mathit{next} \mapsto z$.

Since it is impossible to meet the first, but not the second criterion, the above criteria partition all symbolic heaps in $\mathbf{SHFV}_0^{\leq 1}$ into three equivalence classes. The state space of $\mathcal{A}_{\mathit{sll0}}$ then consists of a representative of each equivalence class. Hence, we define the set of states as

$$\mathcal{Q} \triangleq \{ x_1 \nmid x_1.n \mapsto 0, \quad x_1 \nmid x_1 \neq 0, \quad x \nmid z.\mathit{next} \mapsto z \}.$$

Moreover, the set $\mathcal{F}$ of final states consists of all states which meet the first criterion, i.e., $\mathcal{F} \triangleq \{ x_1 \nmid x_1.n \mapsto 0 \}$. Finally, the transition rules in Δ are given by all rules of the form $\varphi(\psi_1, \ldots, \psi_m) \to \psi$ iff $\vartheta = [\![\varphi(\psi_1, \ldots, \psi_m)]\!]$, where ψ is the symbolic heap determined by the table below:

ψ	$\vartheta \models_\Psi \mathit{sll0}(x)$?	$\vartheta \star x.\mathit{next} \mapsto 0 \models_\Psi \mathit{sll0}(x)$?
$x \nmid x.\mathit{next} \mapsto 0$	yes	yes
$x \nmid x \neq 0$	no	yes
$x \nmid z.\mathit{next} \mapsto z$	no	no

In fact, it suffices to construct heap automata to discharge entailments with single predicate calls on the right-hand side. As shown in [2, Theorem 10], one can always construct suitable heap automata for single points-to assertions $x.sel \mapsto y$ as well as pure formulas $\mathbf{emp} \wedge E = E'$ and $\mathbf{emp} \wedge E \neq E'$. Furthermore, if we already know suitable heap automata for the symbolic heaps φ and ψ, then we can also construct one for the symbolic heap $\varphi \star \psi$.

Consequently, whenever we find heap automata over SIDs in $\mathbf{C}$ for all predicate symbols specified by the SID Ψ, it is decidable whether $\varphi \models_\Psi \psi$ holds for all symbolic heaps $\varphi, \psi \in \mathbf{SHC}$ that contain no existentially quantified variables. For instance, the heap automaton $\mathcal{A}_{\mathit{sll0}}$ in Example 12.1 allows us to discharge entailments between symbolic heaps with at most one (free) variable that may contain predicate calls of the form $\mathit{sll0}(E)$. The construction of the heap automaton $\mathcal{A}_{\mathit{sll0}}$ can be generalized to construct heap automata $\mathcal{A}$ over a set of SIDs $\mathbf{C}$ such that, for some fixed $\Psi \in \mathbf{C}$ and a predicate symbol $\boldsymbol{P}$, we have

$$[\![\mathbf{L}(\mathcal{A})]\!] \;=\; \{\, \vartheta \in \mathbf{SHC}_0 \mid \vartheta \models_\Psi \psi \,\}.$$

Similar to the Myhill-Nerode construction for finite automata over words (cf. [HMU07, Section 3.15]), this amounts to finding a decidable equivalence relation

$$\equiv \;\subseteq\; \mathbf{SHC} \times \mathbf{SHC}$$

with finitely many equivalence classes such that $\varphi \equiv \psi$ holds iff

$$\forall \vartheta \in \mathbf{SHC}_1 : \qquad [\![\vartheta(\varphi)]\!] \models_\Psi \boldsymbol{P}(\vec{x}) \quad \text{iff} \quad [\![\vartheta(\psi)]\!] \models_\Psi \boldsymbol{P}(\vec{x}).$$

Given such an equivalence relation $\equiv$, it suffices to pick one symbolic heap of rank zero out of every equivalence class of $\equiv$ to determine the set of states $\mathcal{Q}_\mathcal{A}$ of heap automaton $\mathcal{A}$. The set of final states then consists of all states that entail $\boldsymbol{P}(\vec{x})$, i.e., $\mathcal{F}_\mathcal{A} \triangleq \{\, \psi \in \mathcal{Q} \mid \psi \models_\Psi \boldsymbol{P}(\vec{x}) \,\}$. Finally, the transition rules of $\mathcal{A}$ first compose a given symbolic heap with the states assigned to each of its predicate calls and then determine its equivalence class. Hence, $\Delta_\mathcal{A}$ is given by

$$\varphi(\psi_1, \ldots, \psi_n) \to \psi \quad \text{iff} \quad [\![\varphi(\psi_1, \ldots, \psi_n)]\!] \equiv \psi.$$

Notice that—due to undecidability of the entailment problem for symbolic heaps in general [Ant+14]—there do *not* always exist suitable heap automata covering all symbolic heaps entailing a given predicate call. Unfortunately, even if such automata exist, their construction quickly becomes fairly complex. For example, extending the heap automaton in Example 12.1 to more than one free variable immediately leads to an explosion of the state space.

Nonetheless, the author conjectures that heap automata allow to derive decision procedures for large fragments of symbolic heap separation logic. Further details will be found in the forthcoming PhD thesis of Jens Katelaan at the Technical University of Vienna.

12.2 Deciding Entailments through Folding

We previously reduced the entailment problem to the question whether all unfoldings of a symbolic heap are covered by the robustness property of a heap automaton. Unfortunately, constructing suitable heap automata is quite complex. As an alternative, we might thus syntactically analyze unfoldings. That is, to verify the entailment $\varphi \models_\Psi \psi$, we attempt to prove that every unfolding of φ is *included* in the set of unfoldings of ψ with respect to Ψ.

Lemma 12.2 For all SIDs Ψ and symbolic heaps $\varphi, \psi \in \mathbf{SHSL}$, we have

$$\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi) \quad \text{implies} \quad \varphi \models_\Psi \psi.$$

Proof. Let $(\mathfrak{s}, \mathfrak{h})$ be an arbitrary, but fixed stack-heap pair. Then:

$$\begin{aligned}
& \mathfrak{s}, \mathfrak{h} \models_\Psi \varphi \\
\text{iff} \quad & [\![\text{ Lemma 10.24 }]\!] \\
& \exists \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \mathfrak{s}, \mathfrak{h} \models \vartheta \\
\text{implies} \quad & [\![\text{ Premise: } \mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi) \text{ }]\!] \\
& \exists \vartheta \in \mathbf{Unf}_\Psi(\psi) \colon \mathfrak{s}, \mathfrak{h} \models \vartheta \\
\text{iff} \quad & [\![\text{ Lemma 10.24 }]\!] \\
& \mathfrak{s}, \mathfrak{h} \models_\Psi \psi
\end{aligned}$$

□

Reasoning about unfoldings is, in fact, commonly found in (semi-)decision procedures for separation logic, such as [BCO05b; BDP11; Chi+12], where it appears in the form of rules for folding and unfolding symbolic heaps. In this section, we briefly discuss the rationale underlying these rules.

Example 12.3 As a running example, we consider an SID Ψ_{pdll} specifying non-empty partial doubly-linked list segments between head x and tail y, where the predecessor of x and the successor of y are unspecified (cf. Example 4.27, page 119). To this end, we fix selectors for the previous and the next element, i.e., $\mathbf{Sel} \triangleq \langle \mathit{prev}, \mathit{next} \rangle$. The SID Ψ_{pdll} is then given by:

$$\begin{aligned}
\Psi_{pdll} \triangleq \{ \; & \mathit{pdll} \Leftarrow x, y \mid y.\mathit{prev} \mapsto x, x.\mathit{next} \mapsto y, \\
& \mathit{pdll} \Leftarrow \underbrace{x, y \mid z.\mathit{prev} \mapsto x, x.\mathit{next} \mapsto z \mid \mathit{pdll}(z, y)}_{= \varphi}, \\
& \mathit{pdll} \Leftarrow \underbrace{x, y \mid \mathit{pdll}(x, z), \mathit{pdll}(z, y)}_{= \psi} \; \}.
\end{aligned}$$

Intuitively, every unfolding of the symbolic heap φ is also an unfolding of ψ because we obtain φ from ψ by unfolding the predicate call $\boldsymbol{pdll}(x, z)$ according to the first rule of $\Psi_{\boldsymbol{pdll}}$. Hence, we have $\mathbf{Unf}_{\Psi_{\boldsymbol{pdll}}}(\varphi) \subseteq \mathbf{Unf}_{\Psi_{\boldsymbol{pdll}}}(\psi)$. By Lemma 12.2, this means that $\varphi \models_{\Psi_{\boldsymbol{pdll}}} \psi$ holds.

Example 12.3 indicates a simple—yet incomplete—approach to check inclusions between unfoldings: Whenever the symbolic heap φ can be folded into the symbolic heap ψ by means of applying the folding relation $\overset{*}{\underset{\Psi}{\Rightarrow}}$ introduced in Definition 10.20, then the inclusion $\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$ holds.

Lemma 12.4 For all SIDs Ψ and symbolic heaps $\varphi, \psi \in \mathbf{SHSL}$, we have

$$\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi \quad \text{implies} \quad \mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi).$$

Proof. By complete induction on the number of folding steps. □

The entailment $\varphi \models_{\Psi_{\boldsymbol{pdll}}} \psi$ considered in Example 12.3 then follows from Lemmas 12.2 and 12.4 and the fact

$$\varphi \overset{*}{\underset{\Psi_{\boldsymbol{pdll}}}{\Longrightarrow}} \psi.$$

To automatically check whether a symbolic heap can be folded into another one, we restrict ourselves to SIDs in which every rule maps more than a single predicate call to a predicate symbol.

Definition 12.5 (Shrinking SID) An SID Ψ is *shrinking* if and only if

$$\forall (\boldsymbol{P} \Leftarrow \varphi) \in \Psi\colon \quad \begin{cases} |\mathbf{PT}(\varphi)| + |\mathbf{EQ}(\varphi)| + |\mathbf{NE}(\varphi)| > 0, \text{ or} \\ |\mathbf{PC}(\varphi)| > 1. \end{cases}$$

For instance, the SID $\Psi_{\boldsymbol{pdll}}$ in Example 12.3 is shrinking. Moreover, notice that every SID can be transformed into a shrinking one by adding rules in which all rules consisting of nothing but a single predicate call have already been applied.

Intuitively, folding a symbolic heap φ with respect to a shrinking SID reduces the size of φ in each step. Consequently, the length of any sequence of folding steps starting in φ is bounded by $2 \cdot \|\varphi\|$, where the factor two stems from the fact that every separation logic atom might first be folded into a predicate call.

This property can be exploited to decide whether $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ holds: Starting with the symbolic heap ψ, Algorithm 2 iteratively guesses up to $2 \cdot \|\varphi\|$ unfolding steps and computes the resulting symbolic heap ϑ via predicate substitution

(Definition 10.18). After that, it checks whether φ and ϑ are isomorphic by guessing a possible isomorphism and applying Definition 10.13.

Notice that the size of each guess is at most polynomial in the size of the algorithm's input, i.e., $n = \|\Psi\| + \|\varphi\| + \|\vartheta\|$. Apart from the nondeterministic guesses, every step of Algorithm 2—including predicate unfolding and computing the multiset notation of a symbolic heap—can be performed in polynomial time. In particular, variables can be renamed prior to predicate unfolding without an isomorphism check. Hence, $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ is decidable in NP. In fact, we cannot expect to obtain a more efficient algorithm:

Theorem 12.6 (Decidability of Folding) Given a shrinking SID Ψ and symbolic heaps φ and ψ, deciding whether $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ holds is NP-complete.

Proof. Algorithm 2 is a nondeterministic algorithm deciding $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ in polynomial time; it thus proves membership in NP.

NP-hardness is shown by a polynomial-time reduction from the NP-complete 3-Partition problem [GJ75] to the question whether $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ holds.

The reduction is inspired by [ARE86, Theorem 1]. A detailed proof is found in Appendix H.1. □

Example 12.7 Resuming our running Example 12.3 involving the SID Ψ_{pdll}, let us apply Algorithm 2 to check whether the symbolic heap

$$\begin{aligned}\varphi \;\triangleq\; & x, y \mid z_1.prev \mapsto 0, z_1.next \mapsto z_2, z_2.prev \mapsto z_1, z_2.next \mapsto z_3, \\ & z_3.prev \mapsto z_2, z_3.next \mapsto z_4, z_4.prev \mapsto z_3, z_4.next \mapsto 0 \\ & \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x\end{aligned}$$

entails the symbolic heap

$$\begin{aligned}\psi \;\triangleq\; & x, y \mid z_1.prev \mapsto 0, z_3.next \mapsto 0 \mid \mathit{pdll}(z_1, z_2), \mathit{pdll}(z_2, z_3) \\ & \mid x = z_1, y = z_2, x \neq 0, y \neq 0, x \neq y, y \neq x.\end{aligned}$$

To verify this entailment, Algorithm 2 guesses that three unfolding steps are required. The computation of symbolic heap ϑ then proceeds as follows:

$$\psi \underset{\Psi_{pdll}}{\Longleftarrow} [\![\text{ apply first rule to } \mathit{pdll}(z_2, z_3)]\!]$$

Input: A growing SID Ψ, symbolic heaps φ and ψ.
Output: yes iff $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ holds.

$\vartheta := \psi$;
pick a number $\mathsf{steps} \in \{0, 1, \ldots, 2 \cdot \|\psi\|\}$;
`// guess symbolic heap` ϑ `with` $\vartheta \overset{\mathsf{steps}}{\underset{\Psi}{\Longrightarrow}} \psi$
`while` ($\mathsf{steps} > 0$) {
    pick a predicate call $\boldsymbol{P}(\vec{E}) \in \mathbf{PC}(\vartheta)$;
    pick a rule $\boldsymbol{P} \Leftarrow \eta$ *in* Ψ ;
    rename η *such that* ϑ *and* η *have no common variables* ;
    $\vartheta := \vartheta \left[\boldsymbol{P}(\vec{E}) \middle/ \eta \right]$;
    $\mathsf{steps} := \mathsf{steps} - 1$
};
`// return yes iff` $\varphi \cong \vartheta$ `holds`
`if` ($|\mathbf{BV}(\varphi)| = |\mathbf{BV}(\vartheta)|$) {
    `// guess potential isomorphism`
    pick a bijective function $f \colon \mathbf{BV}(\varphi) \to \mathbf{BV}(\vartheta)$;
    `// verify isomorphism by comparing multiset notation`
    `if` ($\mathfrak{M}(f(\varphi)) = \mathfrak{M}(\vartheta)$) { $\mathsf{out} :=$ *yes* } `else` { $\mathsf{out} :=$ *no* }
} `else` {
    $\mathsf{out} :=$ *no*
}

Algorithm 2: Nondeterministic decision procedure for the folding problem.

$$x, y \mid z_1.\mathit{prev} \mapsto 0, z_3.\mathit{next} \mapsto 0, \underbrace{z_3.\mathit{prev} \mapsto z_2, z_2.\mathit{next} \mapsto z_3}_{\text{added through unfolding}} \mid \mathit{pdll}(z_1, z_2)$$
$$\mid x = z_1, y = z_2, x \neq 0, y \neq 0, x \neq y, y \neq x$$
$$\Longleftarrow_{\Psi_{\mathit{pdll}}} [\![\text{ apply second rule to } \mathit{pdll}(z_1, z_2)]\!]$$
$$x, y \mid z_1.\mathit{prev} \mapsto 0, z_3.\mathit{next} \mapsto 0, z_3.\mathit{prev} \mapsto z_2, z_2.\mathit{next} \mapsto z_3,$$
$$\underbrace{z_4.\mathit{prev} \mapsto z_1, z_1.\mathit{next} \mapsto z_4,}_{\text{added through unfolding}} \mid \mathit{pdll}(z_4, z_2)$$
$$\mid x = z_1, y = z_2, x \neq 0, y \neq 0, x \neq y, y \neq x$$
$$\Longleftarrow_{\Psi_{\mathit{pdll}}} [\![\text{ apply first rule to } \mathit{pdll}(z_4, z_2)]\!]$$
$$x, y \mid z_1.\mathit{prev} \mapsto 0, z_3.\mathit{next} \mapsto 0, z_3.\mathit{prev} \mapsto z_2, z_2.\mathit{next} \mapsto z_3,$$
$$z_4.\mathit{prev} \mapsto z_1, z_1.\mathit{next} \mapsto z_4, \underbrace{z_2.\mathit{prev} \mapsto z_4, z_4.\mathit{next} \mapsto z_2,}_{\text{added through unfolding}}$$
$$\mid x = z_1, y = z_2, x \neq 0, y \neq 0, x \neq y, y \neq x.$$

The last symbolic heap is isomorphic to φ as witnessed by the function

$$f\colon \{ z_1, z_2, z_3, z_4 \} \to \{ z_1, z_2, z_3, z_4 \},\ z_1 \mapsto z_1, z_2 \mapsto z_4, z_3 \mapsto z_2, z_4 \mapsto z_3.$$

We remark that purely syntactical reasoning by means of folding symbolic heaps is, in general, highly incomplete. Redundant pure formulas, e.g., $x \neq 0$ if x appears on the left-hand side of a points-to assertion, for instance, are not taken into account unless a normal form is imposed or additional proof rules are provided. Furthermore, it is possible that an inclusion $\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$ holds although φ cannot be folded into ψ.

Example 12.8 Consider the SID Ψ given by the following rules:

$$\boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.\mathit{sel} \mapsto x_2,$$
$$\boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.\mathit{sel} \mapsto z \mid \boldsymbol{P}(z, x_2), \text{ and}$$
$$\boldsymbol{Q} \Leftarrow x_1, x_2 \mid x_1.\mathit{sel} \mapsto x_2.$$

Moreover, consider the symbolic heaps φ and ψ below:

$$\varphi \triangleq y \mid z'.\mathit{sel} \mapsto z'', z.\mathit{sel} \mapsto z'' \mid \boldsymbol{Q}(y, z), \text{ and}$$
$$\psi \triangleq y \mid z'.\mathit{sel} \mapsto z \mid \boldsymbol{P}(y, z).$$

Clearly, $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ does *not* hold. However, the single unfolding of φ, namely

$$\vartheta \;\triangleq\; y \mid z'.sel \mapsto z'', z.sel \mapsto z'', y.sel \mapsto z,$$

can be folded into ψ. Hence, we have $\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$.

In the previous example, we reasoned by first *unfolding* the symbolic heap φ in all possible ways before folding each of the resulting heaps into ψ again. This argument is indeed sound as justified by the following lemma:

Lemma 12.9 For all SIDs Ψ and symbolic heaps $\varphi \in \mathbf{SHSL}$, we have

$$\mathbf{Unf}_\Psi(\varphi) \;=\; \begin{cases} \{\, \varphi \,\}, & \text{if } \mathbf{PC}(\varphi) = \emptyset \\ \bigcup_{\varphi \underset{\Psi}{\Leftarrow} \psi} \mathbf{Unf}_\Psi(\psi), & \text{otherwise.} \end{cases}$$

Proof. Similar to [20, Theorem 1.2]. Details are found in Appendix H.2. □

12.3 A Decision Procedure for Graphical Symbolic Heaps

In this section, we present a pragmatic decision procedure for entailments which are actually encountered while running the software model checker Attestor (see Chapter 13). Intuitively, Attestor's abstraction mechanism involves folding symbolic heaps as much as possible. Its state space thus consists of symbolic heaps which are (maximally) folded in the following sense:

Definition 12.10 (Folding Sets) The *folding set* $\mathbf{Fold}_\Psi(\varphi)$ of a symbolic heap φ with respect to SID Ψ is defined as

$$\mathbf{Fold}_\Psi(\varphi) \;\triangleq\; \left\{ \psi \;\middle|\; \varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi \text{ and } \neg\exists \vartheta \colon \psi \underset{\Psi}{\Rightarrow} \vartheta \right\}.$$

We call the symbolic heap φ *folded* with respect to Ψ iff $\mathbf{Fold}_\Psi(\varphi) = \{\, \varphi \,\}$.

We are then not concerned with arbitrary entailments. Rather, the right-hand side of every entailment which arises within Attestor is always a folded symbolic heap. Moreover, both sides of every entailment have the same free variables. Our goal is thus to decide the following restricted entailment problem:

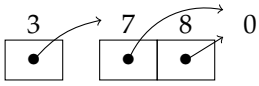


Figure 12.1: A concrete heap $\mathfrak{h}$.

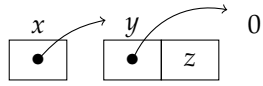


Figure 12.2: A symbolic heap φ.

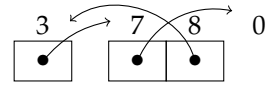


Figure 12.3: A concrete heap $\mathfrak{h}'$.

Definition 12.11 (The Folded Entailment Problem) Given an SID Ψ, a symbolic heap φ, and a folded symbolic heap ψ with $\mathbf{Vars}(\varphi) = \mathbf{Vars}(\psi)$, decide whether the entailment $\varphi \models_\Psi \psi$ holds.

Notice that the undecidability proof by Antonopoulos et al. [Ant+14, Theorem 3] considers an instance of the folded entailment problem. Hence, the above problem is just as undecidable as its unrestricted version. Attestor thus employs a restricted fragment of symbolic heaps in which entailments can be verified by syntactically folding symbolic heaps as presented in Section 12.2.

As a first step towards this fragment, we impose a normal form on symbolic heaps without predicate calls. To this end, recall from Section 10.1.1 that symbolic heaps are intuitively obtained from a single heap, say $\mathfrak{h}$, by substituting all of its addresses and values by variables. Analogously to the illustration of the heap $\mathfrak{h}$ in Figure 12.1, we might thus want to draw the symbolic heap

$$\varphi \triangleq x, y \mid x.next \mapsto y, y.next \mapsto 0, y.sel \mapsto z$$

as a graph consisting of memory cells as shown in Figure 12.2. In fact, for a stack $\mathfrak{s}$ mapping variables x and y to addresses 3 and 7, respectively, we have $\mathfrak{s}, \mathfrak{h} \models \varphi$. As already mentioned in Section 10.1.1, however, drawing such graphs is somewhat deceiving: For example, the shape of the heap $\mathfrak{h}'$ depicted in Figure 12.3 is different from $\mathfrak{h}$. In particular, if we consider addresses 7 and 8 as a single memory block, then it contains a cycle which is not shown in Figure 12.2. Nonetheless, $\mathfrak{s}, \mathfrak{h}' \models \varphi$ holds. To avoid such hidden aliasing, we restrict ourselves to *established* symbolic heaps (cf. Section 11.2.3). Consequently, potential aliases are restricted to the free variables of symbolic heaps.

Assumption 12.12 From now on, we require that *all unfoldings of all symbolic heaps under consideration are both satisfiable and established*. That is, every unfolding of a symbolic heap is satisfied by at least one stack-heap pair and every existentially quantified variable is allocated or equal to a free variable or 0. Both properties can be checked automatically for a given symbolic heap and enforced for a given SID by applying Corollaries 11.19 and 11.22 and Theorem 11.17 for the heap automata in Sections 11.2.2 and 11.2.3.

Attestor imposes strict syntactic restrictions such that symbolic heaps can be unambiguously represented by directed graphs, where addresses correspond to nodes and pointers correspond to labeled edges, respectively. In particular, distinct nodes in these graphs represent distinct addresses and values. We thus refer to the resulting fragment of separation logic as *graphical* symbolic heaps.

Definition 12.13 (Graphical Symbolic Heaps) A symbolic heap φ is *graphical* if and only if it has the following properties:

- points-to assertions and predicate calls contain no free variables, i.e.,

$$\forall \psi \in (\mathbf{PT}(\varphi) \cup \mathbf{PC}(\varphi)) : \quad \mathbf{Vars}(\psi) \subseteq \mathbf{BV}(\varphi),$$

- every free variable is either equal to 0 or a quantified variable, i.e.,

$$\forall x \in \mathbf{Vars}(x)\ \exists E \in (\mathbf{BV}(\varphi) \cup \{0\}) : \quad (x = E) \in \mathbf{EQ}(\varphi),$$

- φ contains no other equalities, i.e., $|\mathbf{EQ}(\varphi)| = |\mathbf{Vars}(\varphi)|$, and
- $\mathbf{NE}(\varphi)$ is the least set satisfying the following rules:
  - for all $x \in \mathbf{Vars}(\varphi)$, if $x = 0 \notin \mathbf{EQ}(\varphi)$, then $x \neq 0 \in \mathbf{NE}(\varphi)$, and
  - for all $x, y \in \mathbf{Vars}(\varphi)$, if there is no $E \in (\mathbf{BV}(\varphi) \cup \{0\})$ such that $x = E, y = E \in \mathbf{EQ}(\varphi)$, then $x \neq y, y \neq x \in \mathbf{NE}(\varphi)$.

It is noteworthy that the free variables of a graphical symbolic heap, say φ, have *no effect on the shape of heaps* $\mathfrak{h}$ specified by φ. Their sole purpose is to describe which stacks $\mathfrak{s}$ are admitted such that $\mathfrak{s}, \mathfrak{h} \models \varphi$ holds.

Example 12.14 A key property of graphical symbolic heaps is that they can be unambiguously represented as a graph. For instance, the symbolic heap φ below, which already appeared in Example 12.7, is graphical.

$$\begin{aligned}\varphi \triangleq\ & x, y \mid z_1.\mathit{prev} \mapsto 0, z_1.\mathit{next} \mapsto z_2, z_2.\mathit{prev} \mapsto z_1, z_2.\mathit{next} \mapsto z_3, \\ & z_3.\mathit{prev} \mapsto z_2, z_3.\mathit{next} \mapsto z_4, z_4.\mathit{prev} \mapsto z_3, z_4.\mathit{next} \mapsto 0 \\ & \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x\end{aligned}$$

φ is precisely captured by the directed graph in Figure 12.4: Every existentially quantified variable $z_1, \ldots, z_4$ as well as the constant 0 is drawn as a circle. The free variables x and y are drawn as boxes with a single edge to the node representing the expression they are equal to according to the equalities $x = z_1$ and $y = z_3$, respectively. Moreover, every points-to

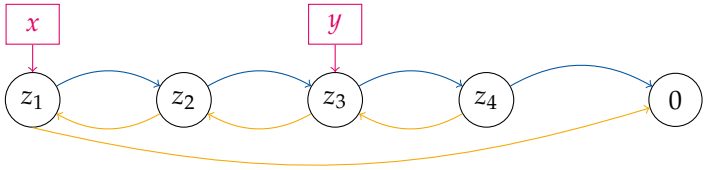


Figure 12.4: Illustration of the graphical symbolic heap in Example 12.14.

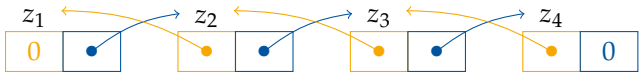


Figure 12.5: Illustration of all heaps satisfying the graphical symbolic heap φ in Example 12.14 for some stack.

assertion corresponds to a directed edge whose color indicates the selector. All inequalities are implicit because two distinct nodes are guaranteed to represent distinct addresses and values. Moreover, Figure 12.5 illustrates the shape of all heaps satisfying φ for some stack.

Formally, "having no effect on the shape of the heap" means that two graphical symbolic heaps without predicate calls are isomorphic whenever they are satisfied by some common stack-heap pair.

Lemma 12.15 For all graphical symbolic heaps φ, ψ of rank zero over the same set of free variables, i.e., $\mathbf{Vars}(\varphi) = \mathbf{Vars}(\psi)$, we have:

$$(\exists(\mathfrak{s},\mathfrak{h}) \colon \mathfrak{s},\mathfrak{h} \models \varphi \text{ and } \mathfrak{s},\mathfrak{h} \models \psi) \qquad \text{implies} \qquad \varphi \cong \psi.$$

Proof. We first observe that φ and ψ are not satisfied by a common stack-heap pair $(\mathfrak{s},\mathfrak{h})$ unless both contain the same number of points-to assertions. The claim is then shown by complete induction on the number of free variables and points-to assertions in φ and ψ. □

Since the converse direction holds for arbitrary satisfiable symbolic heaps due to Lemma 10.14, it follows immediately that *an entailment $\varphi \models \psi$ between graphical symbolic heaps of rank zero (over the same free variables) holds iff φ and ψ are isomorphic*. To reason about graphical symbolic heaps with predicate calls, we would like to preserve this property. Hence, we also impose syntactic restrictions on the SIDs admitted to determine the semantics of predicate calls.

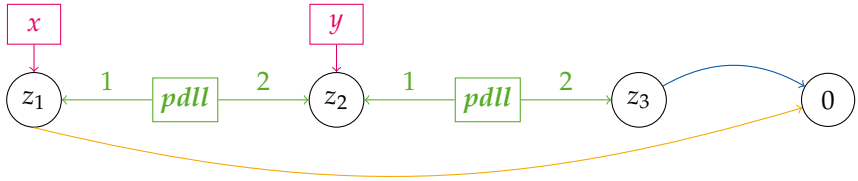


Figure 12.6: Illustration of the graphical symbolic heap ψ in Example 12.17.

Definition 12.16 (Graphical Systems of Inductive Definitions) An SID Ψ is *graphical* iff it is shrinking and, for every rule $(\boldsymbol{P} \Leftarrow \varphi) \in \Psi$, the symbolic heap φ contains no pure formulas, i.e., $\mathbf{EQ}(\varphi) = \mathbf{NE}(\varphi) = \emptyset$.

The rules of a graphical SID do *not* map to graphical symbolic heaps because the role of free variables is different. For graphical symbolic heaps, free variables are used to specify the stack, i.e., the values assigned to program variables. In contrast, for rules of graphical SIDs, free variables represent the parameters of predicate calls. Since a variable might occur twice as a parameter of the same call, e.g., $\boldsymbol{P}(z, z)$, aliasing between free variables is explicitly permitted.

Example 12.17 Recall from Example 12.3 the SID Ψ_{pdll}

$$\begin{aligned}\Psi_{\mathit{pdll}} \triangleq \{\ & \mathit{pdll} \Leftarrow x, y \mid y.\mathit{prev} \mapsto x, x.\mathit{next} \mapsto y, \\ & \mathit{pdll} \Leftarrow x, y \mid z.\mathit{prev} \mapsto x, x.\mathit{next} \mapsto z \mid \mathit{pdll}(z, y), \\ & \mathit{pdll} \Leftarrow x, y \mid \mathit{pdll}(x, z), \mathit{pdll}(z, y)\ \}.\end{aligned}$$

Ψ_{pdll} is graphical. Moreover, the symbolic heap ψ below, which is taken from Example 12.7, is graphical as well.

$$\begin{aligned}\psi \triangleq\ & x, y \mid z_1.\mathit{prev} \mapsto 0, z_3.\mathit{next} \mapsto 0 \mid \mathit{pdll}(z_1, z_2), \mathit{pdll}(z_2, z_3) \\ & \mid x = z_1, y = z_2, x \neq 0, y \neq 0, x \neq y, y \neq x.\end{aligned}$$

Figure 12.6 depicts the graph associated with ψ, where predicate calls are drawn as labeled boxes. The parameters are indicated by numbered edges.

Intuitively, a graphical symbolic heap associates every stack-heap pair with at most one of its unfoldings. As a consequence, entailment checking for graphical symbolic heaps is equivalent to inclusion-checking for their unfoldings.

Lemma 12.18 For all graphical SIDs Ψ and graphical symbolic heaps φ and ψ with $\mathbf{Vars}(\varphi) = \mathbf{Vars}(\psi)$, we have

$$\varphi \models_\Psi \psi \quad \text{iff} \quad \mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi).$$

Proof. If $\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$ holds, then $\varphi \models_\Psi \psi$ holds due to Lemma 12.2. For the converse direction, consider the following:

$\varphi \models_\Psi \psi$
implies ⟦ Lemma 10.24 ⟧
$\forall(\mathfrak{s},\mathfrak{h})\colon \quad (\exists \vartheta \in \mathbf{Unf}_\Psi(\varphi)\colon \mathfrak{s},\mathfrak{h} \models \vartheta)$
$\qquad$ implies $\quad \exists \eta \in \mathbf{Unf}_\Psi(\psi)\colon \mathfrak{s},\mathfrak{h} \models \eta$
implies ⟦ Elementary first-order logic ⟧
$\forall(\mathfrak{s},\mathfrak{h})\colon \forall \vartheta \in \mathbf{Unf}_\Psi(\varphi)\colon$
$\quad \mathfrak{s},\mathfrak{h} \models \vartheta \quad$ implies $\quad \exists \eta \in \mathbf{Unf}_\Psi(\psi)\colon \mathfrak{s},\mathfrak{h} \models \eta$
implies ⟦ Lemma 12.15 ⟧
$\forall(\mathfrak{s},\mathfrak{h})\colon \forall \vartheta \in \mathbf{Unf}_\Psi(\varphi)\colon$
$\quad \mathfrak{s},\mathfrak{h} \models \vartheta \quad$ implies $\quad \exists \eta \in \mathbf{Unf}_\Psi(\psi)\colon \vartheta \cong \eta$
implies ⟦ Every unfolding is satisfiable due to Assumption 12.12 ⟧
$\forall \vartheta \in \mathbf{Unf}_\Psi(\varphi)\colon \exists \eta \in \mathbf{Unf}_\Psi(\psi)\colon \vartheta \cong \eta$
implies ⟦ Elementary first-order logic ⟧
$\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$. □

Unfortunately, the folded entailment problem remains undecidable for graphical SIDs and graphical symbolic heaps. This can be shown, for instance, by reducing the undecidable inclusion problem for context-free string grammars (cf. [BPS64; Gre68]) to the question whether $\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$ holds for a graphical SID Ψ and graphical symbolic heaps φ and ψ. A detailed proof is found in Appendix H.3. To obtain a decidable fragment, we additionally require that the graphical SID underlying an entailment query is *confluent*, i.e., the folding set of any graphical symbolic heap is a singleton. Formally:

Definition 12.19 (Confluent Graphical SIDs) A graphical SID Ψ is *confluent* iff for all graphical symbolic heaps φ, we have $|\mathbf{Fold}_\Psi(\varphi)| = 1$.

Confluence is, admittedly, not a syntactic restriction of SIDs. However, it can be checked automatically. In particular, a suitable decision procedure has been implemented in Attestor.

Theorem 12.20 (Decidability of Confluence [Plu10], [Hei15, p. 30]) It is decidable whether a given graphical SID is confluent.

Proof. A concrete decision procedure for deciding this property for hyperedge replacement grammars (HRG for short) rather than graphical SIDs is found in [Sch19] (notice that folding corresponds to applying HRG derivations backward). This procedure is based on more general results for deciding confluence of (coverable) graph transformation systems (cf. [Plu10, Corollary 1] and [Hri18, pp. 43–46]) by checking strong joinability of critical pairs. The claim then follows from the close relationship between graphical SIDs and HRGs. Details are found in [JGN14] and [Jan17, Proposition 6.1], where graphical SIDs are included in the considered fragment of SIDs. □

Confluence is also practically motivated: Program verification tools, such as Attestor, frequently have to fold and unfold symbolic heaps. If an SID is confluent, then these tools do not have to consider all possible ways in which a symbolic heap can be folded; rather it suffices to fold symbolic heaps in an arbitrary order until no further folding step is possible.

Example 12.21 The SID Ψ_{pdll} in Examples 12.3 and 12.17 is confluent. Hence, exhaustively folding the symbolic heap φ with respect to Ψ_{pdll} always leads to the symbolic heap ψ as demonstrated in Example 12.7. Notice that all rules of Ψ_{pdll} are needed to ensure confluence. As demonstrated on the next page, omitting the third rule, for example, admits another folding of φ.

$$
\begin{aligned}
\varphi \;= x, y \mid\; & z_1.\mathit{prev} \mapsto 0, z_1.\mathit{next} \mapsto z_2, z_2.\mathit{prev} \mapsto z_1, z_2.\mathit{next} \mapsto z_3, \\
& z_3.\mathit{prev} \mapsto z_2, \underbrace{z_3.\mathit{next} \mapsto z_4, z_4.\mathit{prev} \mapsto z_3}_{\text{removed through folding}}, z_4.\mathit{next} \mapsto 0 \\
& \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x \\
\underset{\Psi_{pdll}}{\Longrightarrow} & [\![\text{ fold according to the first rule }]\!] \\
x, y \mid\; & z_1.\mathit{prev} \mapsto 0, \underbrace{z_1.\mathit{next} \mapsto z_2, z_2.\mathit{prev} \mapsto z_1}_{\text{removed through folding}}, z_2.\mathit{next} \mapsto z_3, \\
& z_3.\mathit{prev} \mapsto z_2, z_4.\mathit{next} \mapsto 0 \mid \mathit{pdll}(z_3, z_4) \\
& \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x \\
\underset{\Psi_{pdll}}{\Longrightarrow} & [\![\text{ fold according to the first rule }]\!]
\end{aligned}
$$

$$
\begin{aligned}
&x, y \mid z_1.\mathit{prev} \mapsto 0, \underbrace{z_2.\mathit{next} \mapsto z_3, z_3.\mathit{prev} \mapsto z_2}_{\text{removed through folding}}, z_4.\mathit{next} \mapsto 0 \\
&\quad \mid \mathit{pdll}(z_3, z_4), \mathit{pdll}(z_1, z_2) \\
&\quad \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x \\
&\xRightarrow[\Psi_{\mathit{pdll}}]{} [\![\text{ fold according to the first rule }]\!] \\
&\quad x, y \mid z_1.\mathit{prev} \mapsto 0, z_4.\mathit{next} \mapsto 0 \\
&\quad \mid \mathit{pdll}(z_3, z_4), \mathit{pdll}(z_1, z_2), \mathit{pdll}(z_2, z_3) \\
&\quad \mid x = z_1, y = z_3, x \neq 0, y \neq 0, x \neq y, y \neq x.
\end{aligned}
$$

Hence, the third rule of Ψ_{pdll} is necessary to ensure confluence.

We remark that finding confluent SIDs is nontrivial and depends on the number of free variables. For example, only three rules are needed to define a confluent SID for (partial) doubly-linked list segments with two free variables; the third rule intuitively states that two concatenated doubly-linked list segments specify a doubly-linked list segment as well. However, the same approach does not work to obtain a confluent SID for (complete) doubly-linked list segments with four free variables (cf. Example 4.27, page 119). It is also noteworthy that confluence is indeed a restriction of SIDs.

Theorem 12.22 (Expressiveness of Confluent SIDs) There exists a graphical SID Ψ and a predicate symbol $\boldsymbol{P}$ such that the set of unfoldings $\mathbf{Unf}_\Psi(\boldsymbol{P}(\vec{x}))$ cannot be captured by a confluent graphical SID.

Proof. The construction is similar to [20, Theorem 6]. Let $\mathbf{Sel} = \langle a, b \rangle$ be a sequence of selectors. Moreover, let Ψ be the graphical SID given by:

$$
\begin{aligned}
\boldsymbol{P} &\Leftarrow x, y \mid x.a \mapsto y, \\
\boldsymbol{P} &\Leftarrow x, y \mid x.b \mapsto y, \text{ and} \\
\boldsymbol{P} &\Leftarrow x, y \mid x.a \mapsto z \mid \boldsymbol{P}(z, y)
\end{aligned}
$$

Towards a contradiction, assume that there exists a confluent graphical SID Γ such that $\mathbf{Unf}_\Psi(\boldsymbol{P}(x, y)) = \mathbf{Unf}_\Gamma(\boldsymbol{P}(x, y))$. Then, since each of the three symbolic heaps below is in $\mathbf{Unf}_\Psi(\boldsymbol{P}(x, y))$, the following foldings are possible with respect to Γ:

1. $x, y \mid x.a \mapsto z \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, y),$

2. $x, y \mid x.a \mapsto z, z.a \mapsto y \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, y)$, and

3. $x, y \mid x.a \mapsto z, z.b \mapsto y \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, y)$.

Now, Γ is confluent by assumption. Hence, by applying either the first folding from above twice or the second folding once, the following folding is possible as well:

$$x, y \mid x.a \mapsto z, z.a \mapsto y \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, z), \boldsymbol{P}(z, y) \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, y)$$

By applying the third folding from above twice, this means that we can also perform the following folding:

$$\underbrace{x, y \mid x.a \mapsto z_1, z_1.b \mapsto z_2, z_2.a \mapsto z_3, z_3.b \mapsto y}_{= \varphi}$$
$$\overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, z), \boldsymbol{P}(z, y) \quad \overset{*}{\underset{\Gamma}{\Rightarrow}} \quad x, y \mid \boldsymbol{P}(x, y)$$

Consequently, we have $\varphi \in \mathbf{Unf}_\Gamma(\boldsymbol{P}(x, y))$. However, this contradicts our assumption because $\varphi \notin \mathbf{Unf}_\Psi(\boldsymbol{P}(x, y))$. □

We now show that the folded entailment problem is decidable for confluent graphical SIDs. The key idea is to extend Lemma 12.15 such that two graphical symbolic heaps are isomorphic whenever they share a common unfolding.

Lemma 12.23 Let Ψ be a confluent graphical SID. Moreover, let φ and ψ be folded graphical symbolic heaps with $\mathbf{Vars}(\varphi) = \mathbf{Vars}(\psi)$. Then, we have

$$\mathbf{Unf}_\Psi(\varphi) \cap \mathbf{Unf}_\Psi(\psi) \neq \emptyset \quad \text{iff} \quad \varphi \cong \psi.$$

Proof. The proof is similar to [20, Theorem 7]. More precisely, we have:

$\mathbf{Unf}_\Psi(\varphi) \cap \mathbf{Unf}_\Psi(\psi) \neq \emptyset$

iff ⟦ Definition 10.21 ⟧

$\exists \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \varphi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta$ and $\psi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta$

iff ⟦ Definition 10.20; φ and ψ are folded (Definition 12.10) ⟧

$\exists \vartheta \in \mathbf{Unf}_\Psi(\varphi) \colon \varphi \in \mathbf{Fold}_\Psi(\vartheta)$ and $\psi \in \mathbf{Fold}_\Psi(\vartheta)$

iff ⟦ Ψ is confluent and productive (Assumption 12.12) ⟧

$\varphi \cong \psi$. □

Folding symbolic heaps and checking for isomorphism, i.e., applying Algorithm 2, then yields a complete decision procedure.

Theorem 12.24 (Complexity of the Folded Entailment Problem) The folded entailment problem for confluent graphical SIDs and graphical symbolic heaps is decidable in NP.

Proof. Let Ψ be a confluent graphical SID Ψ. Moreover, let φ be a graphical symbolic heap and ψ be a folded graphical symbolic heap. Our goal is to decide the folded entailment $\varphi \models_\Psi \psi$. To this end, we claim that

$$\varphi \models_\Psi \psi \quad \text{iff} \quad \varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi.$$

Decidability in NP then follows immediately from NP-completeness of the latter problem (see Theorem 12.6).

To verify our claim, let us first assume that $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ does hold. Then $\varphi \models_\Psi \psi$ holds as well due to Lemma 12.4.

Conversely, assume that $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ does *not* hold. Since ψ is folded, we have $\mathbf{Fold}_\Psi(\psi) = \{\psi\}$. Furthermore, since Ψ is confluent, there exists a graphical symbolic heap ϑ such that $\mathbf{Fold}_\Psi(\varphi) = \{\vartheta\}$. By assumption, ψ and ϑ are not isomorphic.

We then proceed as follows:

$$\begin{aligned}
& \psi \ncong \vartheta \\
\text{implies} \quad & [\![\text{ Lemma 12.23 }]\!] \\
& \mathbf{Unf}_\Psi(\vartheta) \cap \mathbf{Unf}_\Psi(\psi) = \emptyset \\
\text{implies} \quad & [\![\text{ Lemma 12.4 }]\!] \\
& \mathbf{Unf}_\Psi(\varphi) \cap \mathbf{Unf}_\Psi(\psi) = \emptyset \\
\text{implies} \quad & [\![\text{ elementary algebra; Assumption 12.12 }]\!] \\
& \mathbf{Unf}_\Psi(\varphi) \not\subseteq \mathbf{Unf}_\Psi(\psi) \\
\text{implies} \quad & [\![\text{ Lemma 12.18 }]\!] \\
& \varphi \models_\Psi \psi \text{ does not hold.}
\end{aligned}$$

Hence, the folded entailment problem is in NP. □

It remains an open problem whether the folded entailment problem for confluent graphical SIDs is NP-complete. The main difficulty is to construct a suitable *confluent* SID when attempting to prove NP-hardness. In particular, the SID in the hardness proof of Theorem 12.6 is graphical, but not confluent. While we

were unable to show NP-completeness, we notice that the empty SID is both confluent and graphical. Hence, given two (folded) graphical symbolic heaps φ,ψ without predicate calls, we have

$$\varphi \models_\emptyset \psi \quad \text{iff} \quad \varphi \cong \psi.$$

Due to the close relationship between directed graphs and graphical symbolic heaps (cf. Example 12.14) it is then straightforward to show that the folded entailment problem is at least as hard as the graph isomorphism problem (cf. [ZKT85, Section 15]). Since it is a longstanding open problem whether the graph isomorphism problem is decidable in polynomial time or NP-complete, we did not further explore whether folded entailments can be solved in polynomial time. Furthermore, we remark that a confluent SID might be exponentially larger than a non-confluent one specifying the same set of unfoldings. If we start with an arbitrary SID and attempt to make it confluent first, we thus do not necessarily obtain an efficient decision procedure.

To conclude this section, we observe that—despite the previous remarks—folded entailments can be solved rather efficiently in practice as SIDs tend to be rather small. For example, Attestor was able to solve all 88326 instances of the folded entailment problem which arose when running its benchmark suite (cf. Chapter 13) in roughly 35.424 seconds.

Chapter 13

Attestor: Model Checking Java Pointer Programs

This chapter is based on prior publications, namely [4; 3], which are presented, discussed, and extended in the broader context of this thesis.

In this chapter, we give a brief tour through Attestor—an automated verification tool for reasoning about Java pointer programs.[1] The tool, its source code, documentation, and all experiments are open-source and available under GPL 3.0 license on both GitHub[2] and the Maven central repository.[3] In a nutshell, Attestor takes a Java program and attempts to verify a specification in linear temporal logic [Pnu77; BK08] (LTL) with support for pointer-specific properties, such as reachability, for all executions which initially satisfy a given precondition. To this end, it proceeds along the line of most program analyses as described in Chapter 2 (Section 2.2, pages 28 to 33). That is, it applies the program's operational semantics to generate a transition system that is subject to further analysis. Since programs manipulating unbounded dynamic data structures typically lead to infinite state spaces, Attestor performs abstraction to compute a finite overapproximation of all possible executions. The abstraction is derived automatically from a user-supplied specification of the involved data structures. The resulting finite abstract transition system is then passed to an integrated LTL model checker which—provided that the abstraction mechanism is strong enough—either confirms that the specification is satisfied or provides a counterexample in terms of the program's concrete operational semantics. Both counterexamples and the full abstract state space can be explored graphically.

Although we will describe Attestor in terms of graphical symbolic heaps and SIDs (cf. Section 12.3), its internal representation uses the equivalent notions of hypergraphs and context-free graph grammars which have originally been proposed as a formalism for heap abstraction by Rieger and Noll [RN08]. A detailed account of the theoretical underpinnings is found in [Hei+15; 3].

[1] The name is a reference to the tool's ability to generate counterexamples. If one insists on a meaningful acronym, it is a bizarre abbreviation for "AuTomaTEd analysiS of poinTer prOgRams".

[2] https://moves-rwth.github.io/attestor/

[3] https://mvnrepository.com/artifact/de.rwth.i2/attestor

In summary, ATTESTOR's main features can be characterized as follows:

- Once supplied with a graphical SID or a context-free graph grammar that specifies the data structures accessed by a program, it automatically derives implicit abstract semantics.
- Apart from SIDs or graph grammars to guide abstraction, the analysis is fully automated. In particular, no program annotations are required.
- Specifications are given by LTL formulas that support a rich set of program properties, ranging from memory safety over shape, reachability or balancedness (for an extension of graphical SIDs [3]) to temporal properties such as full traversal or preservation of the exact heap structure.
- Modular reasoning is supported in the form of contracts that summarize the effect of executing (recursive) procedures. These contracts can be automatically derived or manually specified.
- Valuable feedback is provided through a comprehensive report including non-spurious counterexamples in case of property violations.
- It allows for graphical exploration of state spaces and counterexamples.

13.1 Attestor's Abstraction

ATTESTOR performs a program analysis with abstraction as discussed in Section 2.2. As such, it generates a labeled abstract transition system which is then explored by a model checker. Before we describe the implementation of ATTESTOR, let us briefly summarize the abstractions involved in this process.

Concrete Semantics Conceptually, states of the concrete transition system consist of a program and a stack-heap pair; an example is illustrated in Figure 13.1 (topmost). The concrete program semantics is similar in spirit to the operational semantics of $\mathbf{P^3L}$ programs presented in Section 4.1.3. Since pointer arithmetic is not supported by the Java programming language, we assume that the heap is accessed through selectors (cf. Section 4.1.4).

Symbolic Heap Abstraction ATTESTOR concentrates on reasoning about the heap. Primitive data types—except from finitely many literal constants, e.g., 0—are not supported. We thus drop the values of all variables on the stack which do not represent pointers or one of the aforementioned literals. The resulting stack-heap pairs are then abstracted to graphical symbolic heaps (cf. Definition 12.13) of rank zero, i.e., without any predicate calls. In other words,

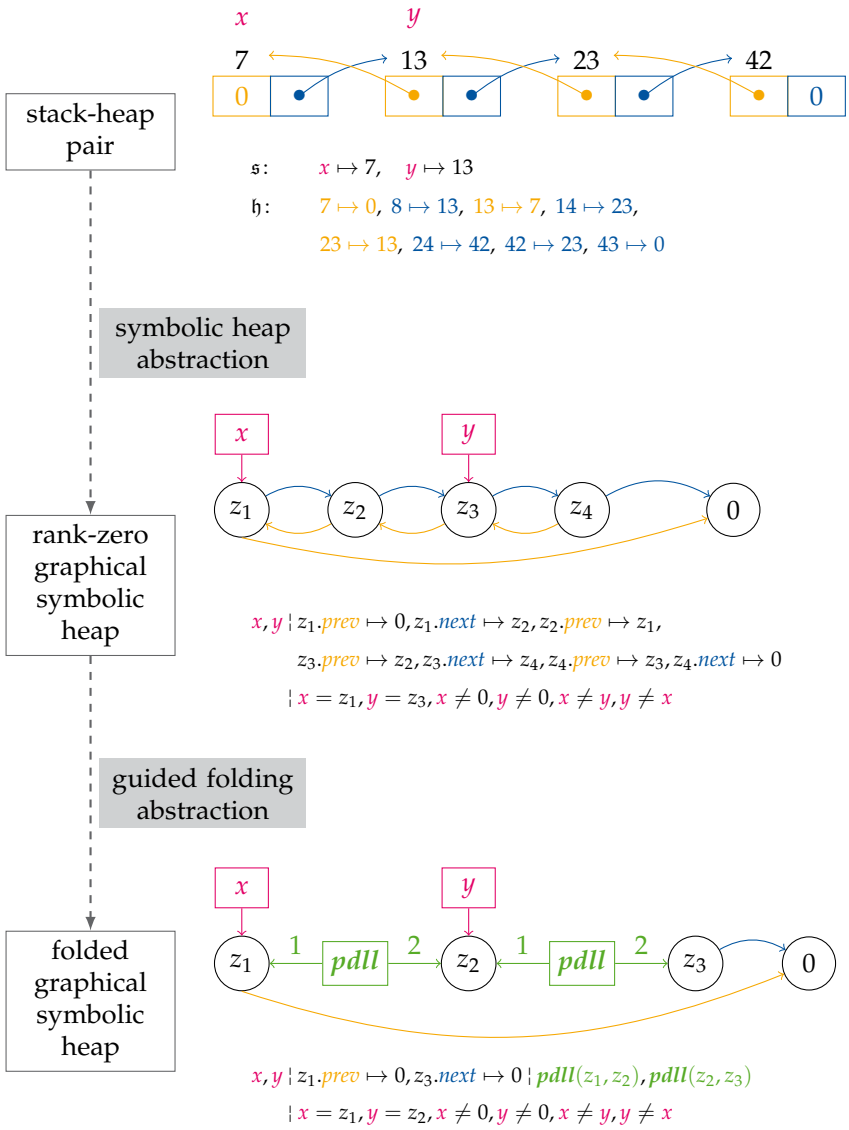


Figure 13.1: Example of ATTESTOR's abstraction.

we abstract from concrete addresses and values. More precisely, the involved abstraction function maps every concrete state $\langle C, \mathfrak{s}, \mathfrak{h} \rangle$ to the state $\langle C, \varphi \rangle$, where φ is the graphical symbolic heap such that $(\mathfrak{s}, \mathfrak{h})$ satisfies φ, i.e., $\mathfrak{s}, \mathfrak{h} \models \varphi$. Notice that, by Lemma 12.15, φ is uniquely determined up to isomorphism. Figure 13.1 (middle) depicts an example of a graphical symbolic heap of rank zero obtained from abstracting a given stack-heap pair.

Regarding the computation of the abstract transition relation, moving from stack-heap pairs to graphical symbolic heaps of rank zero preserves all information about both the shape of the heap and variables storing pointers to memory cells. Hence, the semantics of heap manipulating statements, i.e., lookup, mutation, allocation, and deallocation, is easily lifted to graphical symbolic heaps of rank zero. The semantics of statements involving arithmetic, such as evaluating conditionals and loop guards, is overapproximated by nondeterministically considering all possible executions. Further details are found in [Hei+15, Section 3.3] and [20, Appendix A.3].

Guided Folding Abstraction To obtain a finite abstract state space, another abstraction is applied on top of the previous one. This abstraction is guided by a user-supplied graphical system of inductive definitions (cf. Definition 12.16), say Ψ. We assume that Ψ is confluent (see Definition 12.19). While this assumption is technically not required to obtain a sound analysis, confluence ensures decidability of the folded entailment problem (Theorem 12.24) and simplifies computing abstractions. Intuitively, the abstraction function determined by Ψ folds a given graphical symbolic heap φ as much as possible; it thus maps φ to the single graphical symbolic heap (of arbitrary rank) in its folding set $\mathbf{Fold}_\Psi(\varphi)$ (Definition 12.10). This is exemplified in Figure 13.1 (lowermost), where the underlying SID Ψ models partial doubly-linked list segments (see Example 12.3).

Before we apply this abstraction function, however, we have to ensure that it respects the atomic propositions required for checking our specification. In Attestor, a specification is an LTL formula over a finite set of robustness properties (cf. Chapter 11) which serve as atomic propositions. Consequently, every atomic proposition is a property of graphical symbolic heaps of rank zero that (1) corresponds to a state of and (2) can be checked by a heap automaton, say $\mathcal{A}$. Examples of properties include simple equalities and points-to relationships between program variables as well as reachability conditions. To guarantee that these properties are preserved by the abstraction, we first *refine* our SID Ψ according to the heap automaton $\mathcal{A}$. Our actual abstraction function is then determined by the resulting refined SID.[4]

[4]Notice that the resulting SID remains confluent provided that $\mathcal{A}$ is deterministic, i.e., their move relation is a function, and we remove predicate symbols which are not decorated with an automaton state. In particular, all heap automata considered in Section 11.2 are deterministic.

To effectively compute the abstract transition relation, we adhere to a standard pattern in shape analysis [WRS02; RSW07]. That is, for every state $\langle C, \varphi \rangle$, we compute its successor states as follows:

1. We first *materialize* the graphical symbolic heap φ, i.e., we apply unfolding steps until no selectors accessed by the semantics of C are hidden within a predicate symbol.
2. We execute the (concrete) semantics of C on all graphical symbolic heaps obtained from materialization as if they would have rank zero.
3. We check whether the obtained graphical symbolic heaps entail a previously computed one. To this end, we *canonicalize* them, i.e., fold them as much as possible, and then search for isomorphic graphical symbolic heaps which are already part of the abstract transition system. This is in line with the decision procedure discussed in Section 12.3.

13.2 The Attestor Tool

ATTESTOR is implemented in Java and consists of about 37,000 LOC (excluding comments and benchmarks). An architectural overview is depicted in Figure 13.2. It shows the tool's inputs (left), its outputs (right), and the main phases of its execution (middle). Moreover, a separate frontend allows to graphically navigate through the produced outputs. These elements are discussed in detail below.

13.2.1 Input

As shown in Figure 13.2 (left), a verification task passed to ATTESTOR consists of two obligatory (solid boxes) and up to three optional (dashed boxes) inputs.

Program First, ATTESTOR accepts both Java and Java Bytecode, where the former is translated to the latter prior to the analysis.

Data Structure Specification Second, ATTESTOR requires a specification of the dynamic data structures employed by the given program to guide its abstraction. In order to obtain a finite abstract transition system, this specification is supposed to cover the data structures emerging during program execution. The user may choose from a fixed set of predefined specifications for standard data structures, such as singly- and doubly-linked lists and binary trees. Alternatively, a specification may be supplied manually either as a context-free graph grammar in a JSON-style graph format or as a graphical SID. For instance, Figure 13.3 depicts a graphical SID with three rules specifying singly-linked list

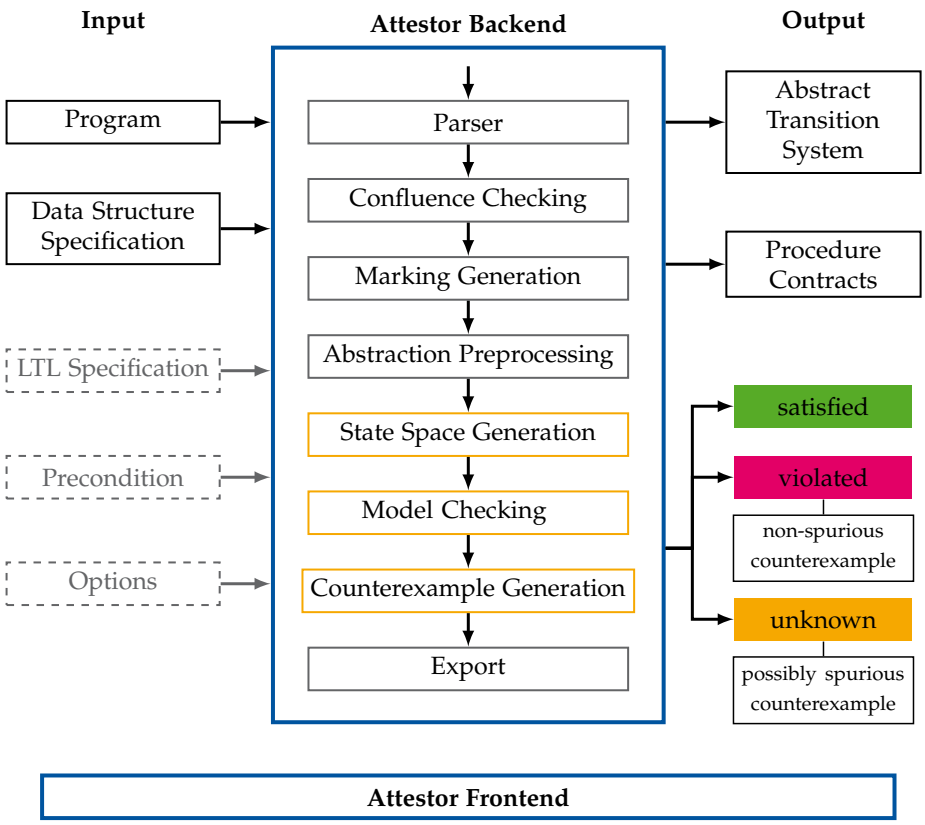


Figure 13.2: The architecture of Attestor.

segments. In contrast to our multiset notation for symbolic heaps introduced in Definition 10.11, the free variables of each rule are provided as parameters of a rule's left-hand side, i.e., to the left of the first "<=". A list of the existentially quantified variables is provided as the first component of each rule's right-hand side, i.e., immediately after "<=". Moreover, notice that every variable is additionally annotated with a type (here: `List`) in curly brackets. This information corresponds to the class of Java objects referenced by a variable. As such, it determines the possible selectors that can be used together with a variable.

Precondition By default, Attestor attempts to overapproximate all program executions starting with an empty heap. Alternatively, a graphical symbolic

```
sll(x{List}, y{List})
<= x.next -> y
<= z{List} | x.next -> z * sll(z,y)
<= z{List} | sll(x,z) * sll(z,y);
```

Figure 13.3: A graphical SID for singly-linked lists in ATTESTOR's syntax.

heap φ may be supplied as a precondition. ATTESTOR will then assume that all program executions start in a stack-heap pair satisfying φ.

LTL Specification ATTESTOR always checks memory safety while generating an abstract transition system. To verify more involved properties, an LTL formula over heap-specific atomic propositions has to be provided. The latter comprise, for example, shape properties, (in)equalities between variables, and reachability of objects identified by program variables. For instance, the LTL formula below shows a possible specification for reversing doubly linked lists:

```
G (X {terminated} -> (
    {isReachable(head,tail,[next])}
    & {isReachable(tail,head,[prev])})
)
```

Intuitively, it states that a doubly-linked list with head `head` and tail `tail` has been reversed, i.e., `head` is reachable from `tail` using `prev`-pointers only and `tail` is reachable from `head` using `next`-pointers only, whenever we terminate in the next state (in which the garbage collector kicks in and cleans up the heap). Further examples of specifications are briefly discussed in Section 13.3 alongside our experimental evaluation.

Options Finally, it is possible to provide additional options to control, amongst others, the garbage collection behavior, the re-use of results from previous analyses in the form of procedure contracts, or the granularity of abstraction. A detailed account of all options is provided as part of ATTESTOR's documentation.

13.2.2 Phases

ATTESTOR attempts to verify a specification for a given program in eight phases as illustrated in Figure 13.2 (middle).

Phase 1: Parser All provided inputs are parsed and preprocessed. In particular, the input program is read and—if necessary—transformed to Java Bytecode. The resulting Java Bytecode program is then translated into the Jimple intermediate language (cf. [Val+99]) using the Soot framework[5] to reduce the instruction set which must be supported by our operational semantics. Moreover, data structure specifications are internally represented as a graph grammar regardless of whether they have been provided as one or as a graphical SID.

Phase 2: Confluence Checking While confluence (see Definition 12.19) is not required for the soundness of Attestor's analysis, it significantly improves both precision and performance. If enabled, the provided data structure specification is thus checked for confluence prior to the actual analysis. To this end, all possible critical pairs, i.e., overlappings of SID rules which enable distinct folding steps, are enumerated and checked for joinability, i.e., whether exhaustively folding them leads to isomorphic symbolic heaps. A detailed account of Attestor's decision procedure for confluence is found in [Sch19]. In particular, we remark that Attestor was capable to decide for various graphical SIDs, e.g., singly-linked list segments, doubly-linked list with two or four free variables, binary trees, in-trees, and binary trees with linked leaves, whether they are confluent within at most 500ms each. Apart from the graphical SID for doubly-linked lists with four free variables, all of the aforementioned SIDs are confluent. This demonstrates an advantage of partial data structure specifications: While it is simple to define a confluent SID for doubly-linked lists with two missing pointers, constructing one for complete doubly-linked lists is non-trivial.

Phase 3: Marking Generation Depending on the provided LTL specification, additional markings [Hei+15, Section 6], i.e., artificial program variables, are added to the graphical symbolic heap determining the initial state. Markings are never modified by the program and prevent abstractions of the marked Java object. As such, they allow tracking object identities during program execution which is required, for instance, to validate that every initially allocated object is eventually accessed by a program. Intuitively, the marking generation algorithm simulates the execution of a simple data structure traversal algorithm. The result of this simulation is an abstract state space in which the marking has been placed on any possible object in the given data structure. The states in which the simulated traversal terminated then serve as initial states of the transition system considered in the remaining phases.

Phase 4: Abstraction Preprocessing Before the actual state space generation, the user-supplied graphical SID guiding abstraction is refined (cf. Theorem 11.17) to respect the atomic propositions appearing in LTL specifications (if

[5] http://sable.github.io/soot/

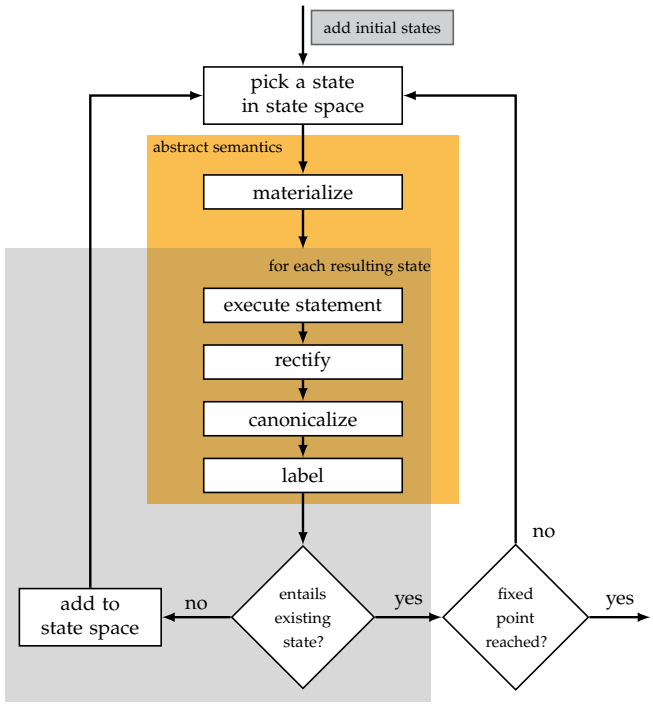


Figure 13.4: Main steps of the state space generation loop.

any is provided). Furthermore, various optimizations—some of which depend on user-defined options—are applied to improve the performance of abstraction and materialization. In particular, this involves precomputing the rules required for materialization of every program statement. Moreover, to reduce the number of expensive operations, such as abstraction and entailment checking, ATTESTOR identifies chains of straight-line code (excluding function calls) which are then treated as a single program statement; they are thus abstracted only once and no intermediate entailment checks are performed. Other optimizations include the removal of dead variables, garbage collection, and placing artificial program variables on the successors of every program variable to prevent abstraction of points-to assertions which are likely to be needed soon for program execution.

Phase 5: State Space Generation The core module of ATTESTOR is the generation of an abstract transition system according to the algorithm illustrated in Figure 13.4. It is provided with a set of initial states which each consist of

a program and a graphical symbolic heap. From these, ATTESTOR picks a state and applies the abstract semantics of the next statement to execute until no new states can be generated, i.e., a fixed point has been reached. First, the graphical symbolic heap contained in the state is materialized such that all selectors required by the statement are accessible. Each of the resulting formulas is then modified according to the concrete operational semantics of the statement; possible null pointer dereferences are automatically detected and reported at this stage. In the rectification step, a graphical symbolic heap is cleaned from dead variables and garbage (unless specified otherwise in the provided options). After that, it is folded as much as possible in the canonicalization step and labeled with atomic propositions by passing it to a suitable heap automaton (see Section 11.1.2). Finally, we check whether the state consisting of the current program and the obtained graphical symbolic heap is covered by one already contained in the state space. That is, we search for a state with the same program whose graphical symbolic heap is entailed by the current one.

Phases 6 and 7: Model Checking & Counterexample Generation Once an abstract transition system has been generated, ATTESTOR checks whether it is satisfied by the provided LTL specification. To this end, ATTESTOR uses an off-the-shelf tableau-based LTL model checking algorithm [BCG95]. There is, however, also prototypical support for both on-the-fly model checking during state space generation and hierarchical model checking in the sense of [AY01] to avoid reasoning about procedure calls twice (cf. [Cha19] for details). Regardless of the chosen model checking algorithm, the following outcomes are possible:

1. The model checker proves that the specification is satisfied. Then the specified property holds for all program executions starting in a stack-heap pair satisfying the precondition.

2. The model checker proves that the specification is violated. As long as the specification in question is a safety-property [BK08, Chapter 4.2], ATTESTOR computes a "bad prefix" of a path in the abstract transition system which serves as a counterexample. After that, it checks whether the counterexample is realizable by an actual program execution or spurious. To this end, the path is *replayed* according to the concrete program semantics:

   a) If ATTESTOR manages to replay the execution, it confirms that the specification is violated. Furthermore, it computes a concrete initial state causing the property violation.

   b) Otherwise, a warning is issued and the result is changed to unknown.

3. For large abstract transition systems, the model checker might run out of memory and thus returns unknown.

Phase 8: Export Finally, the generated abstract state space, the model checking results, and possibly generated counterexamples are exported into JSON files which can be imported by the ATTESTOR frontend for graphical exploration.

13.2.3 Output

As shown in Figure 13.2 (right), we obtain three main outputs once the analysis is completed: the computed abstract transition system, the derived procedure contracts, i.e., pairs of pre- and postconditions for each encountered function, and the model checking results. For each provided LTL formula, results comprise the possible answers satisfied, violated, and unknown obtained in Phase 6. In the latter two cases, ATTESTOR might additionally provide a (non-spurious) counterexample and a concrete initial state causing the property violation for further testing and debugging.

13.2.4 Frontend

ATTESTOR features a graphical frontend that visualizes inputs as well as results. In particular, it enables graphical exploration of state spaces, counterexamples, procedure contracts, and abstraction rules (as rules of graph grammars). Figure 13.5, page 357, depicts a screenshot of the graphical state space exploration component. The topmost pane allows switching to the general report and to search for particular states, e.g., all states in which execution has terminated. The pane below contains information about the currently selected state, such as the program statement which is executed next and the atomic propositions it satisfies. The graphical symbolic heap corresponding to the currently selected state is illustrated in the right pane below (our graphical notation is explained in Section 12.3). Finally, the left pane shows an excerpt from the generated abstract transition system and allows navigating through it. All graphs are rendered using the JavaScript library CYTOSCAPE.JS.[6]

13.3 Evaluation

We conclude this chapter with an experimental evaluation of ATTESTOR against common algorithms on various dynamic data structures and specifications.

Tool comparison While there exists a plethora of verification tools for pointer programs, such as FORESTER [Hol+13], GROOVE [Gha+12], TVLA [Bog+07], JUGGRNAUT [HNR10; Hei+15], and HIP/SLEEK [CDG11; Chi+12], these tools differ in multiple aspects:

[6] http://js.cytoscape.org/

- *Input languages* range from C code without recursion (Forester) and Java-like pseudo-code (Hip/Sleek) over Java/Java Bytecode (Juggrnaut) to assembly code (Tvla) and graph transformation systems (Groove).
- The *degree of automation* differs heavily: Forester only requires source code. Hip/Sleek and Juggrnaut additionally expect general data structure specifications in the form of graph grammars or SIDs. Moreover, Tvla requires additional program-dependent instrumentation predicates.
- *Verifiable properties* cover at least memory safety. Apart from Forester, the aforementioned tools are additionally capable of identifying the shape of the heap. Furthermore, Hip/Sleek is able to reason about shape-numeric properties, e.g. lengths of lists, if a suitable specification is provided. While these properties are not supported by Tvla, it is possible to verify reachability properties. Moreover, Juggrnaut can reason about properties specified in linear temporal logic.

Benchmarks Due to the large degree of diversity outlined above there is no publicly available and representative set of standardized benchmarks to compare the aforementioned tools [Abd+16]. We thus evaluated Attestor on a collection of challenging, pointer intensive algorithms compiled from the literature [Bog+07; Bou+06; Hol+13; LRS06]. The aforementioned algorithms are examples of successful applications of program verification. To assess our counterexample generation, we also considered *invalid* specifications, e.g., that a reversed list is the same list as the input list. Furthermore, we injected faults into our examples by swapping and deleting statements.

Properties *Memory safety (M)* is always checked during state space generation. In addition to that, we considered five classes of properties which have been verified by the built-in LTL model checker:

- The *shape property (S)* establishes that the heap is of a specified shape, e.g. a doubly-linked list or a (balanced) tree (for balancedness an extended notion of graphical SIDs is required; see [3]).
- The *reachability property (R)* checks whether some variable is reachable from another one via specified selectors.
- The *visit property (V)* verifies whether every element of the input is accessed by a specified variable.
- The *neighborhood property (N)* checks whether the input data structure coincides with the output data structure upon termination.

- Finally, we consider other functional *correctness properties (C)*, e.g., the return value is not null upon termination.

Moreover, we occasionally injected errors in the considered programs and checked whether ATTESTOR is capable of finding non-spurious counterexamples. These benchmarks are marked with an (X).

Experiments We conducted experiments on an Intel Core i7-7500U CPU at 2.70GHz with 16GB of RAM with the Java virtual machine (OPENJDK version 1.8.0_151) limited to its default setting of 2GB of RAM. All experiments were run using the Java benchmarking harness JMH.[7] An excerpt from our experimental results is shown in Table 13.1. All provided runtimes are in seconds. Since the total runtime includes starting the Java runtime environment, we also provide the runtimes for abstract state space generation and verification (which includes the former). Additionally, for comparison purposes, we considered Java implementations of benchmarks that have been previously analyzed for memory safety by FORESTER [Hol+13]; the results are found in Table 13.2.

The full benchmark collection together with its documentation and all scripts to reproduce our results is available online.[8]

Our experiments demonstrate that both memory safety (M) and shape (S) are efficiently processed with regard to both state space size and runtime. This is not surprising as these properties are directly handled by the state space generation engine. The most challenging tasks are the visit (V) and neighborhood (N) properties as they require to track objects across program executions by means of markings. The latter have a similar impact as program variables: Increasing their number impedes abstraction as larger parts of the heap have to be kept concrete. This effect can especially be observed for the Lindstrom tree traversal procedure where adding one (V) and three markings (N) both increase the verification effort by an order of magnitude.

[7] https://openjdk.java.net/projects/code-tools/jmh/
[8] https://github.com/moves-rwth/attestor-examples

		Number of States		State Space Gen.		Verification		Total Runtime	
Benchmark	**Properties**	min	max	min	max	min	max	min	max
`SLL.traverse`	M,S,R,V,N,X	13	97	0.030	0.074	0.039	0.097	0.757	0.848
`SLL.reverse`	M,S,R,V,X	46	268	0.050	0.109	0.050	0.127	0.793	0.950
`SLL.reverse` (recursive)	M,S,V,N,X	40	823	0.038	0.100	0.044	0.117	0.720	0.933
`DLL.reverse`	M,S,R,V,N,X	70	1508	0.076	0.646	0.097	0.712	0.831	1.763
`DLL.findLast`	M,C,X	44	44	0.069	0.069	0.079	0.079	0.938	0.938
`SLL.findMiddle`	M,S,R,V,N,X	75	456	0.060	0.184	0.060	0.210	0.767	0.975
`Tree.traverse` (Lindstrom)	M,S,V,N	229	67941	0.119	8.901	0.119	16.52	0.845	17.36
`Tree.traverse` (recursive)	M,S	91	21738	0.075	1.714	0.074	1.765	0.849	2.894
`AVLTree.binarySearch`	M,S	192	192	0.117	0.172	0.118	0.192	0.917	1.039
`AVLTree.searchAndBack`	M,S,C	455	455	0.193	0.229	0.205	0.289	1.081	1.335
`AVLTree.searchAndSwap`	M,S,C	3855	4104	0.955	1.590	1.004	1.677	1.928	2.521
`AVLTree.leftMostInsert`	M,S	6120	6120	1.879	1.942	1.932	1.943	2.813	2.817
`AVLTree.insert`	M,S	10388	10388	3.378	3.676	3.378	3.802	4.284	4.720
`AVLTree.sllToAVLTree`	M,S,C	7166	7166	2.412	2.728	2.440	2.759	3.383	3.762

Table 13.1: The experimental results [4, Table 1]. All runtimes are in seconds.

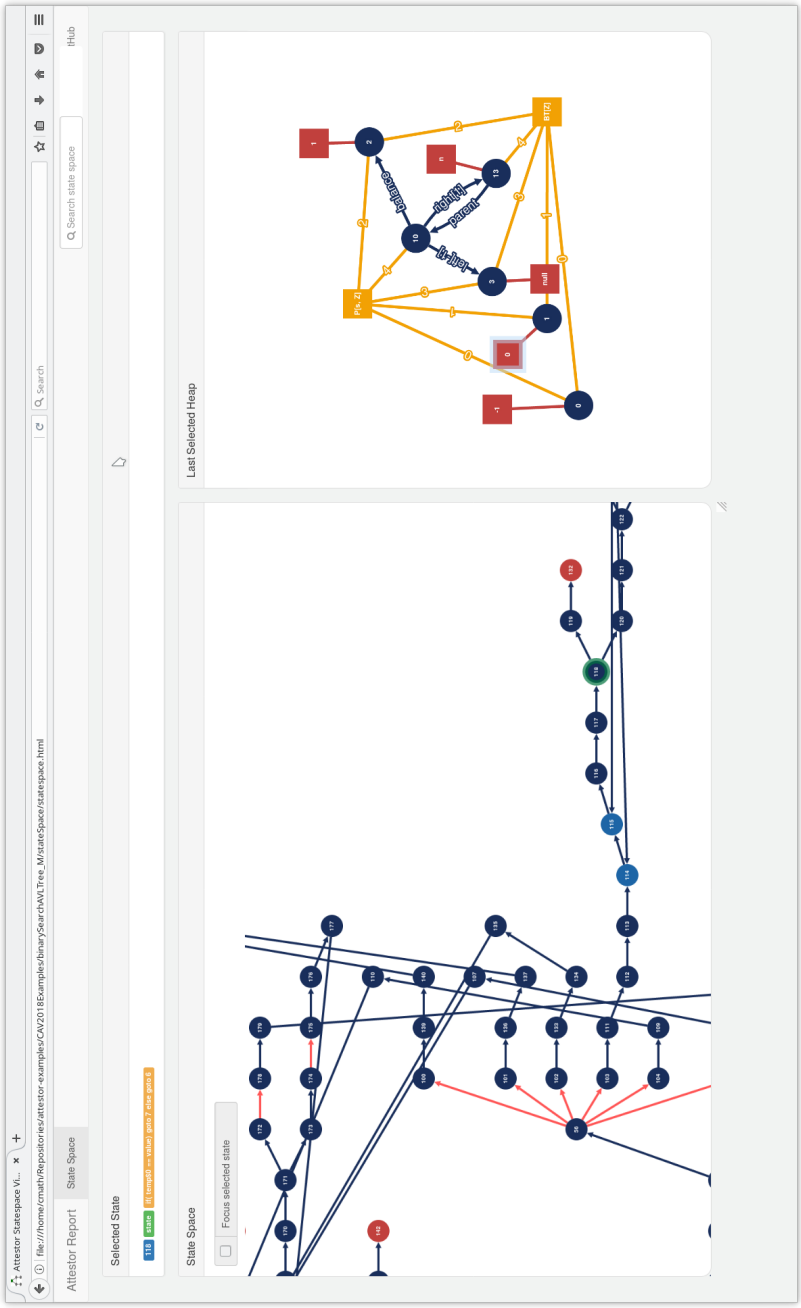


Figure 13.5: Screenshot of ATTESTOR's frontend [20].

Benchmark	Number of States	Verification Time (s)
`SLL.bubblesort`	287	0.134
`SLL.deleteElement`	152	0.096
`SLLHeadPtr` (traverse)	111	0.095
`SLL.insertsort`	369	0.147
`ListOfCyclicLists`	313	0.153
`DLL.insert`	379	0.207
`DLL.insertsort1`	4302	1.467
`DLL.insertsort2`	1332	0.514
`DLL.buildAndReverse`	277	0.164
`CyclicDLL` (traverse)	104	0.108
`Tree.construct`	44	0.062
`Tree.constructAndDSW`	1334	0.365
`SkipList.insert`	302	0.160
`SkipList.build`	330	0.173

Table 13.2: ATTESTOR's performance on FORESTER's benchmarks [4, Table 2].

Chapter 14

Conclusion and Future Work

We studied aspects of automated reasoning *with* and *about* the symbolic heap fragment of separation logic with user-defined inductive predicate definitions.

For reasoning *about* separation logic, we considered a class of *robustness properties* and the *entailment problem* in particular.

For reasoning *with* separation logic, we briefly reported on the implementation of Attestor—a model checker for Java pointer programs.

Robustness Properties We developed an algorithmic framework to decide whether a given system of inductive definitions (SID) satisfies a robustness property, e.g., satisfiability, reachability, or establishment. Our approach is based on the notion of *heap automata*—a specialized kind of tree automaton that is tailored towards reasoning about unfoldings of symbolic heaps. Apart from decision procedures, heap automata also enable systematic refinement of SIDs to guarantee that all of their (remaining) unfoldings are robust. They thus support both debugging and optimizing SIDs before the actual verification process.

We presented several case studies—ranging from the satisfiability problem to reachability problems—that demonstrate the applicability of heap automata. A prototypical implementation of our framework indicates that the decision procedures derived from heap automata are competitive with dedicated decision procedures for, e.g., satisfiability of symbolic heaps.

Furthermore, heap automata are applied within the Attestor tool to discharge the state labeling problem. That is, they decide whether an abstract state, i.e., a graphical symbolic heap, satisfies an atomic proposition.

The entailment problem We outlined how heap automata can be applied to discharge entailments. Moreover, we considered the *folded* entailment problem. While this problem is undecidable in general, we presented a pragmatic decision procedure for the folded entailment problem for graphical symbolic heaps and confluent graphical SIDs. This decision procedure covers all entailments encountered by the Attestor tool. Moreover, it lies in the complexity class NP and is at least as hard as the well-known graph isomorphism problem.

Although we have been unable to prove a tight lower bound, we conjecture that the aforementioned problem is NP-complete.

Attestor We briefly reported on the key features and the implementation of Attestor. In a nutshell, Attestor takes a Java pointer program and attempts to verify specifications in linear temporal logic with dedicated atomic propositions for describing properties of the heap. To this end, it applies abstraction which is guided by a graphical SID. Our experiments demonstrate that Attestor is capable of proving complex properties of pointer intensive algorithms including preservation of the original input by the Lindstrom tree traversal algorithm.

14.1 Future Work

An obvious task is to clarify the exact complexity of the folded entailment problem for confluent graphical SIDs. In this context, it would also be desirable to obtain a mathematical characterization of the kind of data structures that can be specified by confluent SIDs. We list other directions for future work below.

Confluent SIDs We have shown in Theorem 12.22 that there exist graphical SIDs which specify sets of unfoldings that cannot be captured by a confluent graphical SID. At the same time, for every $k \geq 1$, there exists an SID specifying the set of all symbolic heaps without predicate calls and with at most k free variables. While such an SID is useless for both abstraction and discharging folded entailments—after all, everything is folded into a single predicate call—it shows that every graphical SID *can* be extended to a confluent graphical SID describing a potentially larger set of unfoldings. This raises the question of how confluent SIDs which only slightly increase the set of unfoldings described by an originally provided SID can be learned automatically. Another question is whether and how the knowledge of having a confluent SID is beneficial for the performance of other proof systems. For instance, a common example for cyclic proof systems (cf. [BDP11, Example 3]) is their capability of proving that two suitably connected singly-linked list segments entail a single list segment. This fact is immediate for confluent SIDs: Constructing a confluent SID for singly-linked list segments forces us to add a rule which makes this fact explicit:

$$\mathit{sll} \Leftarrow x, y \mid \mathit{sll}(x, z), \mathit{sll}(z, y).$$

Biabduction with Heap Automata An important decision problem for separation logic, which we did not cover in this thesis, is the *biabduction problem*: Given assertions φ and ψ, find assertions ϑ and η such that $\varphi \star \vartheta \models_\Psi \psi \star \eta$ and $\varphi \star \vartheta$ is satisfiable. This problem arises when aiming for full automation,

i.e., even specifications should be inferred automatically (cf. [Cal+11; GKO11; BG14]). In this context, it has been identified as "key to automatic specification inference at industrial scale" [BGK17]. An interesting aspect of using heap automata for solving entailments (see Section 12.1) is that they typically capture how an unfolding must be extended such that it entails a given symbolic heap. This suggests applying heap automata to the biabduction problem, where every possible extension of a symbolic heap is a potential solution.

Support for Data in Attestor The abstraction techniques employed by Attestor presented in Chapter 13 provide no support for payload data stored within dynamic data structures. While there exist extensions to deal with a few shape-numeric properties, such as balancedness [3], it would be interesting to investigate whether the syntactic rewriting approach taken by Attestor works well in the presence of richer data theories.

Part IV

Appendix

We collect a few essential definitions and results from domain theory, in particular complete lattices and various fixed point theorems, which are used throughout this thesis. A more thorough introduction to domain theory is found in various textbooks, such as [Bir40; Win93].

A.1 Partial Orders and Complete Lattices

Definition A.1 (Partial Order) A *partial order* $\langle D, \sqsubseteq \rangle$ consists of a set D and a binary relation $\sqsubseteq \subseteq D \times D$ such that $\sqsubseteq$ is

- reflexive, i.e., $\forall r \in D \in D\colon r \sqsubseteq r$,
- antisymmetric, i.e., $\forall r, s \in D\colon r \sqsubseteq s$ and $s \sqsubseteq r$ implies $r = s$, and
- transitive, i.e., $\forall r, s, t \in D\colon r \sqsubseteq s$ and $s \sqsubseteq t$ implies $r \sqsubseteq t$.

We write $d \sqsubset d'$ as a shortcut for $d \sqsubseteq d'$ and $d' \neq d$.

Definition A.2 (Order Isomorphism) Two partial orders $\langle D, \sqsubseteq \rangle$ and $\langle A, \leq \rangle$ are *isomorphic* iff there exists a bijective function $f\colon D \to A$ such that

$$\forall r, s \in D\colon \qquad r \sqsubseteq s \text{ iff } f(r) \leq f(s).$$

We do not distinguish between isomorphic partial orders.

Definition A.3 (Upper and Lower Bounds) Let $\langle D, \sqsubseteq \rangle$ be a partial order and $S \subseteq D$ be a subset of D.

Then $u \in D$ is an *upper bound* of S if and only if for all $s \in S$, we have $s \sqsubseteq u$. Moreover, u is the *least upper bound* of S if and only if

- u is an upper bound of S, and

- for every upper bound u' of S, we have $u \sqsubseteq u'$.

If the least upper bound, which we also call the *supremum*,of S exists, we denote it by $\sup S$. Furthermore, if $\sup S \in S$, then $\sup S$ is called the *greatest element*, or maximum, of S.

Analogously, $l \in D$ is a *lower bound* of S if and only if for all $s \in S$, we have $l \sqsubseteq s$. Moreover, l is the *greatest lower bound* of S if and only if

- l is a lower bound of S, and
- for every lower bound l' of S, we have $l' \sqsubseteq l$.

If the greatest lower bound, which we also call the *infimum*, of S exists, we denote it by $\inf S$. Furthermore, if $\inf S \in S$, then $\inf S$ is called the *least element*, or minimum, of S.

Definition A.4 (Complete Lattice) A partial order $\langle D, \sqsubseteq \rangle$ is a *complete lattice* if every subset $S \subseteq D$ has a least upper bound.

Lemma A.5 For every set S, $\left(2^S, \subseteq\right)$ is a complete lattice.

Lemma A.6 ([Bir40, Chapter 4]) For every complete lattice $\langle D, \sqsubseteq \rangle$:

- Every subset $S \subseteq D$ has a greatest lower bound.
- The least element of D is $\bot \triangleq \sup \emptyset = \inf D$.
- The greatest element of D is $\top \triangleq \sup D = \inf \emptyset$.

A.2 Transfinite Induction

The following presentation is based on [Bir40, Chapter 3].

Definition A.7 A partial order $\langle D, \sqsubseteq \rangle$ is *well-ordered* if and only if every non-empty subset of D has a least element. Let 0 be the least element of D.

Then $0 \neq \alpha \in D$ is called a *limit ordinal* if and only if

$$\forall \beta\colon \quad \beta \sqsubset \alpha \text{ implies } \exists \gamma\colon \ \beta \sqsubset \gamma \sqsubset \alpha.$$

Otherwise, i.e., if there exists a largest $\beta \in D$ such that $\beta \sqsubset \alpha$, we call α a *successor ordinal* and denote it by $\beta + 1$.

Theorem A.8 (Principle of Transfinite Induction) Let $\langle \{P_\alpha\}, \sqsubset \rangle$ be a well-ordered set of propositions with least element P_0. Then all propositions P_α are true whenever

(a) (induction base) P_0 is true,

(b) (induction step) if P_α is true, then $P_{\alpha+1}$ is true, and

(c) (limit step) if α is a limit ordinal and, for all $\beta \sqsubset \alpha$, P_β is true, then P_α is true.

In particular, the simplest infinite well-ordered set is given by $\langle \mathbb{N}, \leq \rangle$, i.e., the set of natural numbers $\mathbb{N}$ with the canonical ordering $\leq$. Since this well-ordered set contains no limit ordinals, applying the principle of transfinite induction to a countable set is identical to the principle of complete induction.

A.3 Fixed Points

A total function f with domain A and image B is denoted by $f\colon A \to B$.

Definition A.9 (Fixed Points) Let $\langle D, \sqsubseteq \rangle$ be a partial order. Moreover, let $f\colon D \to D$ be a function. Then $d \in D$ is

- a *prefixed point* if $f(d) \sqsubseteq d$,
- a *postfixed point* if $d \sqsubseteq f(d)$, and
- a *fixed point* if $f(d) = d$.

Definition A.10 (Monotone Function) Let $\langle D, \sqsubseteq \rangle$ be a partial order. A function $f\colon D \to D$ is *monotone* if and only if for all $r, s \in D$, we have

$$r \sqsubseteq s \quad \text{implies} \quad f(r) \sqsubseteq f(s).$$

Theorem A.11 (Knaster-Tarski [Kna28; Tar+55]) For a complete lattice $\langle D, \sqsubseteq \rangle$ and a monotone function $f\colon D \to D$, the set $\{ d \mid f(d) = d \}$ of fixed points of f is a complete lattice with ordering $\sqsubseteq$.

In particular, the least fixed point $\mathsf{lfp}(f)$ and the greatest fixed point $\mathsf{gfp}(f)$ are given by

$$\mathsf{lfp}(f) = \inf\{ d \mid f(d) \sqsubseteq d \} \quad \text{and} \quad \mathsf{gfp}(f) = \sup\{ d \mid d \sqsubseteq f(d) \}.$$

Corollary A.12 (Park [Par69]) For a complete lattice $\langle D, \sqsubseteq \rangle$ and a monotone function $f : D \to D$, we have:

$$\begin{array}{lll} \forall d \in D: & f(d) \sqsubseteq d & \text{implies} \quad \mathsf{lfp}\,(f) \sqsubseteq d \\ \forall d \in D: & d \sqsubseteq f(d) & \text{implies} \quad d \sqsubseteq \mathsf{gfp}\,(f) \end{array}$$

Definition A.13 (Function Composition) Let $\langle D, \sqsubseteq \rangle$ be a complete lattice and α be some ordinal number. The α-fold application of a function $f : D \to D$ is defined as

$$f^{\alpha} \triangleq \begin{cases} \lambda d.\, d & \text{, if } \alpha = 0 \\ f(f^{\beta}) & \text{, if } \alpha = \beta + 1 \text{ is a successor ordinal} \\ \sup \{ f^{\beta} \mid \beta \sqsubset \alpha \} & \text{, if } \alpha \text{ is a limit ordinal.} \end{cases}$$

Theorem A.14 (Cousot and Cousot [CC79]) For a complete lattice $\langle D, \sqsubseteq \rangle$ and a monotone function $f : D \to D$, the least fixed point of f is given by

$$\mathsf{lfp}\,(f) \;=\; \sup \{ f^{\alpha}(\bot) \mid \alpha \text{ is an ordinal number} \}.$$

In particular, the supremum is attained for some ordinal number.

Moreover, the greatest fixed point of f is given by

$$\mathsf{gfp}\,(f) \;=\; \inf \{ f^{\alpha}(\top) \mid \alpha \text{ is an ordinal number} \}.$$

In particular, the infimum is attained for some ordinal number.

For a set D, let $2^D \triangleq \{ S \mid S \subseteq D \}$ denote its powerset. Function f is lifted to a function on sets, i.e., $f : 2^D \to 2^D$, by pointwise application:

$$\forall S \in 2^D: \quad f(S) \triangleq \{ f(s) \mid s \in S \}.$$

Definition A.15 (Continuous Function) Let $\langle D, \sqsubseteq \rangle$ be a complete lattice. A function $f : D \to D$ is *continuous* if and only if

$$\forall S \in 2^D: \quad \sup f(S) \;=\; f(\sup S).$$

Theorem A.16 (Kleene [Kle+52]) For a complete lattice $\langle D, \sqsubseteq \rangle$ and a continuous function $f : D \to D$, the least fixed point of f is given by

$$\mathsf{lfp}\,(f) \;=\; \sup \{ f^{n}(\bot) \mid n \in \mathbb{N} \}.$$

Moreover, the greatest fixed point of f is given by

$$\mathsf{gfp}\,(f) \;=\; \inf\{\, f^n(\top) \mid n \in \mathbb{N} \,\}\,.$$

B.1 Proof of the Entailment in Section 4.4.2

Our goal is to show that

$$x \mapsto - \star y \mapsto - \star \mathsf{true}$$

implies

$$x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (y \mapsto - \star (y \mapsto 17 \mathrel{-\!\!\!\star} x \hookrightarrow 3))).$$

To this end, consider the following:

$$\begin{array}{ll}
& x \mapsto - \star y \mapsto - \star \mathsf{true} \\
// \implies & [\![\text{Lemma E.6}]\!] \\
& x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (x \mapsto 3 \star y \mapsto - \star \mathsf{true})) \\
// \implies & [\![\text{Theorem 4.17 (c)}]\!] \\
& x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (y \mapsto - \star x \mapsto 3 \star \mathsf{true})) \\
// \implies & [\![\text{Lemma E.6}]\!] \\
& x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (y \mapsto - \star (y \mapsto 17 \mathrel{-\!\!\!\star} y \mapsto 17 \star x \mapsto 3 \star \mathsf{true}))) \\
// \implies & [\![\text{Theorem 6.30 (b), elementary algebra}]\!] \\
& x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (y \mapsto - \star (y \mapsto 17 \mathrel{-\!\!\!\star} \mathsf{true} \star x \mapsto 3 \star \mathsf{true}))) \\
// \implies & [\![\text{elementary algebra, Definition of } x \hookrightarrow 3]\!] \\
& x \mapsto - \star (x \mapsto 3 \mathrel{-\!\!\!\star} (y \mapsto - \star (y \mapsto 17 \mathrel{-\!\!\!\star} x \hookrightarrow 3)))
\end{array}$$

C.1 Proof of Theorem 5.8

We have to show that, for every MDP $\mathcal{M} = \langle S, \mathbf{Act}, \mathsf{Prob}, s_0, s_G, \mathsf{rew} \rangle$ and every state $s \in S$, we have

$$\mathsf{ExpRew}(\mathcal{M}_s) = \inf_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s')=p>0} \mathsf{rew}(s') + p \cdot \mathsf{ExpRew}(\mathcal{M}_{s'}).$$

Proof. The claim is dual to a well-established result for Markov decision processes that is found in [Put05, Theorem 7.1.3]: Assume that all rewards assigned to states are either non-negative or non-positive. Moreover, let $\overline{\mathsf{ExpRew}(\mathcal{M})}$ denote the *maximal* expected reward of MDP $\mathcal{M}$, i.e.,

$$\overline{\mathsf{ExpRew}(\mathcal{M})} \triangleq \sup_{\mathfrak{S}} \sum_{s_0 \ldots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}}(s_0 \ldots s_n) \cdot \mathsf{rew}_{\mathcal{M}}(s_0 \ldots s_n).$$

Then, for every state $s \in S$, we have

$$\overline{\mathsf{ExpRew}(\mathcal{M}_s)} = \sup_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s')=p>0} \mathsf{rew}(s') + p \cdot \overline{\mathsf{ExpRew}(\mathcal{M}_{s'})}.$$

Now, let $\mathcal{M}'$ be the same MDP as $\mathcal{M}$ except that we multiply every reward assigned to a state with minus one. Then, we have

$$\begin{aligned}
&\overline{\mathsf{ExpRew}(\mathcal{M}')} \\
&= [\![\text{ By definition of maximal expected rewards }]\!] \\
&\sup_{\mathfrak{S}} \sum_{s_0 \ldots s_n \in \mathbf{Paths}_{\mathcal{M}'}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}'}(s_0 \ldots s_n) \cdot \mathsf{rew}_{\mathcal{M}'}(s_0 \ldots s_n) \\
&= [\![\text{ By definition of } \mathcal{M}']\!]
\end{aligned}$$

$$
\begin{aligned}
&\sup_{\mathfrak{S}} \sum_{s_0 \ldots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}}(s_0 \ldots s_n) \cdot -1 \cdot \mathsf{rew}_{\mathcal{M}}(s_0 \ldots s_n) \\
&= \; [\![\text{ factor out } -1; \sup -f = -\inf f \;]\!] \\
&\quad - \inf_{\mathfrak{S}} \sum_{s_0 \ldots s_n \in \mathbf{Paths}_{\mathcal{M}}(\mathfrak{S})} \mathsf{Prob}_{\mathcal{M}}(s_0 \ldots s_n) \cdot \mathsf{rew}_{\mathcal{M}}(s_0 \ldots s_n) \\
&= \; [\![\text{ By definition of minimal expected rewards }]\!] \\
&\quad - \mathsf{ExpRew}\,(\mathcal{M})\,.
\end{aligned}
$$

Hence, for every state $s \in S$, we have $\mathsf{ExpRew}\,(\mathcal{M}_s) = -\overline{\mathsf{ExpRew}\,(\mathcal{M}'_s)}$. To conclude the proof, we then proceed as follows:

$$
\begin{aligned}
&\mathsf{ExpRew}\,(\mathcal{M}_s) \\
&= \; [\![\text{ By the previously shown equality }]\!] \\
&\quad - \overline{\mathsf{ExpRew}\,(\mathcal{M}'_s)} \\
&= \; [\![\text{ By [Put05, Theorem 7.1.3] and definition of } \mathcal{M}' \;]\!] \\
&\quad - \sup_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s') = p > 0} -\mathsf{rew}(s') + p \cdot \overline{\mathsf{ExpRew}\,(\mathcal{M}'_{s'})} \\
&= \; [\![\text{ By the previously shown equality }]\!] \\
&\quad - \sup_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s') = p > 0} -\mathsf{rew}(s') + p \cdot -\mathsf{ExpRew}\,(\mathcal{M}_{s'}) \\
&= \; [\![\text{ factor out } -1; \sup -f = -\inf f; \text{ algebra }]\!] \\
&\quad \inf_{a \in \mathbf{Act}(s)} \sum_{\mathsf{Prob}(s,a,s') = p > 0} \mathsf{rew}(s') + p \cdot \mathsf{ExpRew}\,(\mathcal{M}_{s'})\,. \qquad \square
\end{aligned}
$$

C.2 Proof of Theorem 6.18 (adjointness)

Our goal is to show that, for all expectations $X, Y, Z \in \mathbb{E}$, we have

$$X \star Y \preceq Z \quad \text{iff} \quad X \preceq Y \mathrel{-\!\!\star} Z.$$

The proof relies on the following auxiliary result:

Lemma C.1 Let $X, Y, Z \in \mathbb{E}$ be expectations. Then, for all stacks $\mathfrak{s}$ and heaps $\mathfrak{h}$, $\mathfrak{h}_1$,$\mathfrak{h}_2$ with $\mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2$, $Y(\mathfrak{s}, \mathfrak{h}_2) > 0$, and ($Y(\mathfrak{s}, \mathfrak{h}_2) < \infty$ or $Z(\mathfrak{s}, \mathfrak{h}) < \infty$):

$$X(\mathfrak{s}, \mathfrak{h}_1) \preceq \frac{Z(\mathfrak{s}, \mathfrak{h})}{Y(\mathfrak{s}, \mathfrak{h}_2)} \quad \text{iff} \quad X(\mathfrak{s}, \mathfrak{h}_1) \cdot Y(\mathfrak{s}, \mathfrak{h}_2) \preceq Z(\mathfrak{s}, \mathfrak{h}).$$

Proof. We distinguish three cases:

1. $Y(\mathfrak{s},\mathfrak{h}_2) < \infty$ and $Z(\mathfrak{s},\mathfrak{h}) < \infty$. By standard arithmetic, we have:

$$X(\mathfrak{s},\mathfrak{h}_1) \preceq \frac{Z(\mathfrak{s},\mathfrak{h})}{Y(\mathfrak{s},\mathfrak{h}_2)} \quad \text{iff} \quad X(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2) \preceq Z(\mathfrak{s},\mathfrak{h}).$$

2. $Y(\mathfrak{s},\mathfrak{h}_2) = \infty$ and $Z(\mathfrak{s},\mathfrak{h}) < \infty$. Then:

$$\begin{aligned}
& X(\mathfrak{s},\mathfrak{h}_1) \preceq \frac{Z(\mathfrak{s},\mathfrak{h})}{Y(\mathfrak{s},\mathfrak{h}_2)} \\
\text{iff} \quad & [\![\text{by assumption; } \forall u \in \mathbb{N}_{>0} \colon {}^{u}/_{\infty} = 0]\!] \\
& X(\mathfrak{s},\mathfrak{h}_1) \preceq 0 \\
\text{iff} \quad & [\![\text{by assumption: } Y(\mathfrak{s},\mathfrak{h}_2) = \infty]\!] \\
& X(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2) \preceq 0 \\
\text{iff} \quad & [\![\text{by assumption: } 0 \preceq Z(\mathfrak{s},\mathfrak{h}) \preceq \infty]\!] \\
& X(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2) \preceq Z(\mathfrak{s},\mathfrak{h}).
\end{aligned}$$

3. $Y(\mathfrak{s},\mathfrak{h}_2) < \infty$ and $Z(\mathfrak{s},\mathfrak{h}) = \infty$. Then:

$$\begin{aligned}
& X(\mathfrak{s},\mathfrak{h}_1) \preceq \frac{Z(\mathfrak{s},\mathfrak{h})}{Y(\mathfrak{s},\mathfrak{h}_2)} \\
\text{iff} \quad & [\![\text{by assumption; } \forall u \in \mathbb{N}_{>0} \colon {}^{\infty}/_{u} = \infty]\!] \\
& X(\mathfrak{s},\mathfrak{h}_1) \preceq \infty \\
\text{iff} \quad & [\![\text{by assumption: } Y(\mathfrak{s},\mathfrak{h}_2) < \infty, Z(\mathfrak{s},\mathfrak{h}) = \infty]\!] \\
& X(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2) \preceq Z(\mathfrak{s},\mathfrak{h}).
\end{aligned}$$

□

Proof (of Theorem 6.18). Let $X, Y, Z \in \mathbb{E}$. Then consider the following:

$$\begin{aligned}
& X \preceq Y \mathrel{-\!\!\!\circledast} Z \\
\text{iff} \quad & [\![\text{Definition of } \preceq]\!] \\
& \forall\mathfrak{s}\forall\mathfrak{h}\colon \quad X(\mathfrak{s},\mathfrak{h}) \preceq (Y \mathrel{-\!\!\!\circledast} Z)(\mathfrak{s},\mathfrak{h}) \\
\text{iff} \quad & [\![\text{Definition of } \mathrel{-\!\!\!\circledast}]\!] \\
& \forall\mathfrak{s}\forall\mathfrak{h}\colon \quad X(\mathfrak{s},\mathfrak{h}) \preceq \inf \left\{ \frac{Z(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')}{Y(\mathfrak{s},\mathfrak{h}')} \;\middle|\; \begin{array}{l} \mathfrak{h} \# \mathfrak{h}' \text{ and } Y(\mathfrak{s},\mathfrak{h}') > 0 \\ \text{and } \big(Y(\mathfrak{s},\mathfrak{h}') < \infty \\ \qquad \text{or } Z(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}') < \infty \big) \end{array} \right\}
\end{aligned}$$

iff $[\![$ $\mathrm{X}(\mathfrak{s},\mathfrak{h})$ is a lower bound of $\inf\{\dots\}$; thus all elements of $\dots$ are $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}\forall\mathfrak{h}'\colon \quad \begin{array}{l}\mathfrak{h}\,\#\,\mathfrak{h}' \text{ and } \mathrm{Y}(\mathfrak{s},\mathfrak{h}')>0\\ \text{and } \big(\ \mathrm{Y}(\mathfrak{s},\mathfrak{h}')<\infty \\ \text{or } \mathrm{Z}(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')<\infty\ \big)\end{array} \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}) \preceq \frac{\mathrm{Z}(\mathfrak{s},\mathfrak{h}\uplus\mathfrak{h}')}{\mathrm{Y}(\mathfrak{s},\mathfrak{h}')}$$

iff $[\![$ Rename $\mathfrak{h}$ to $\mathfrak{h}_1$, $\mathfrak{h}'$ to $\mathfrak{h}_2$, and $\mathfrak{h}_1\uplus\mathfrak{h}_2$ to $\mathfrak{h}$ $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}_1\forall\mathfrak{h}_2\forall\mathfrak{h}\colon \quad \begin{array}{l}\mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \text{ and } \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)>0\\ \text{and } \big(\ \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)<\infty \text{ or } \mathrm{Z}(\mathfrak{s},\mathfrak{h})<\infty\ \big)\end{array} \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}_1) \preceq \frac{\mathrm{Z}(\mathfrak{s},\mathfrak{h})}{\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)}$$

iff $[\![$ Lemma C.1 $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}_1\forall\mathfrak{h}_2\forall\mathfrak{h}\colon \quad \begin{array}{l}\mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \text{ and } \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)>0\\ \text{and } \big(\ \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)<\infty \text{ or } \mathrm{Z}(\mathfrak{s},\mathfrak{h})<\infty\ \big)\end{array} \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ $\mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$ satisfied for $\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)=\mathrm{Z}(\mathfrak{s},\mathfrak{h})\cdot\infty$ $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}_1\forall\mathfrak{h}_2\forall\mathfrak{h}\colon \quad \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \text{ and } \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)>0 \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ $\mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$ trivially satisfied for $\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)=0$ $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}_1\forall\mathfrak{h}_2\forall\mathfrak{h}\colon \quad \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ swap quantifiers $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}\forall\mathfrak{h}_1\forall\mathfrak{h}_2\colon \quad \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \quad \text{implies} \quad \mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ $\mathrm{Z}(\mathfrak{s},\mathfrak{h})$ is an upper bound of $\mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2)$ for all $\mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2$ $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}\colon \quad \sup\{\,\mathrm{X}(\mathfrak{s},\mathfrak{h}_1)\cdot\mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \mid \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\,\} \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ Definition of $\star$ $]\!]$

$$\forall\mathfrak{s}\forall\mathfrak{h}\colon \quad (\mathrm{X}\star\mathrm{Y})(\mathfrak{s},\mathfrak{h}) \preceq \mathrm{Z}(\mathfrak{s},\mathfrak{h})$$

iff $[\![$ Definition of $\preceq$ $]\!]$

$$\mathrm{X}\star\mathrm{Y} \preceq \mathrm{Z}.$$

□

C.3 Proof of Theorem 7.10 (Quantitative Frame Rule)

Proof. By induction on the structure of **P⁴L** programs.

The case `skip`.

$$\begin{aligned} & \mathsf{wp}\,[\texttt{skip}]\,(\mathrm{X})\star\mathrm{Y} \\ =\ & [\![\ \text{Figure 7.1}\ ]\!] \end{aligned}$$

$$
\begin{aligned}
& \mathrm{X} \star \mathrm{Y} \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[\texttt{skip}]\,(\mathrm{X} \star \mathrm{Y}).
\end{aligned}
$$

The case $x := E$.

$$
\begin{aligned}
& \mathsf{wp}\,[x := E]\,(\mathrm{X}) \star \mathrm{Y} \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathrm{X}\,[x/E] \star \mathrm{Y} \\
= \; & [\![\; x \in \mathbf{Mod}\,(x := E). \text{ Hence, } x \notin \mathbf{Vars}\,(\mathrm{Y}) \;]\!] \\
& \mathrm{X}\,[x/E] \star \mathrm{Y}\,[x/E] \\
= \; & [\![\text{ elementary algebra }]\!] \\
& (\mathrm{X} \star \mathrm{Y})\,[x/E] \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[x := E]\,(\mathrm{X} \star \mathrm{Y}).
\end{aligned}
$$

The case $x :\approx \mu$.

$$
\begin{aligned}
& \mathsf{wp}\,[x :\approx \mu]\,(\mathrm{X} \star \mathrm{Y}) \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v \in \mathbb{Z}} \mu(\mathfrak{s})(v) \cdot (\mathrm{X} \star \mathrm{Y})\,[x/v]\,(\mathfrak{s},\mathfrak{h}) \\
= \; & [\![\text{ substitution distributes; } x \notin \mathbf{Vars}\,(\mathrm{Y}) \;]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v \in \mathbb{Z}} \mu(\mathfrak{s})(v) \cdot (\mathrm{X}\,[x/v] \star \mathrm{Y})(\mathfrak{s},\mathfrak{h}) \\
= \; & [\![\text{ Theorem 6.22 (b) }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v \in \mathbb{Z}} ((\mu(\mathfrak{s})(v) \cdot \mathrm{X}\,[x/v]) \star \mathrm{Y})(\mathfrak{s},\mathfrak{h}) \\
= \; & [\![\text{ Theorem 6.14 (c) }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v \in \mathbb{Z}} (\mathrm{Y} \star (\mu(\mathfrak{s})(v) \cdot \mathrm{X}\,[x/v]))(\mathfrak{s},\mathfrak{h}) \\
\succeq \; & [\![\text{ Theorem 6.16 (c) }]\!] \\
& \mathrm{Y} \star \left(\lambda(\mathfrak{s},\mathfrak{h}).\ \sum_{v \in \mathbb{Z}} \star(\mu(\mathfrak{s})(v) \cdot \mathrm{X}\,[x/v])(\mathfrak{s},\mathfrak{h}) \right) \\
= \; & [\![\text{ Figure 7.1 }]\!]
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{Y} \star \mathsf{wp}\,[x :\approx \mu]\,(\mathrm{X}) \\
=\ & [\![\ \text{Theorem 6.14 (c)}\ ]\!] \\
& \mathsf{wp}\,[x :\approx \mu]\,(\mathrm{X}) \star \mathrm{Y}.
\end{aligned}
$$

The case $x := <E>$.

$$
\begin{aligned}
& \mathsf{wp}\,[x := <E>]\,(\mathrm{X} \star \mathrm{Y}) \\
=\ & [\![\ \text{Figure 7.1}\ ]\!] \\
& \sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \mathrel{-\!\!\star} (\mathrm{X} \star \mathrm{Y})\,[x/v]) \\
=\ & [\![\ x \notin \mathbf{Vars}\,(\mathrm{Y})\ ]\!] \\
& \sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \mapsto v] \mathrel{-\!\!\star} (\mathrm{X}\,[x/v] \star \mathrm{Y})) \\
=\ & [\![\ \text{Theorem 6.30 (a)}\ ]\!] \\
& \sup_{v \in \mathbb{Z}} [E \hookrightarrow v] \cdot (\mathrm{X}\,[x/v] \star \mathrm{Y}) \\
=\ & [\![\ \text{Definition of } \star\text{; elementary algebra}\ ]\!] \\
& \lambda(\mathfrak{s}, \mathfrak{h}).\ \sup_{v \in \mathbb{Z}} \max \\
& \qquad \{\, [E \hookrightarrow v]\,(\mathfrak{s}, \mathfrak{h}) \cdot (\mathrm{X}\,[x/v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s}, \mathfrak{h}_2)) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \\
=\ & [\![\ \text{elementary algebra}\ ]\!] \\
& \lambda(\mathfrak{s}, \mathfrak{h}).\ \sup_{v \in \mathbb{Z}} \max \\
& \qquad \{\, ([E \hookrightarrow v]\,(\mathfrak{s}, \mathfrak{h}) \cdot \mathrm{X}\,[x/v]\,(\mathfrak{s}, \mathfrak{h}_1)) \cdot \mathrm{Y}(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \\
\succeq\ & [\![\ \text{take subset in which } [E \hookrightarrow v] \text{ is evaluated in } \mathfrak{h}_1 \text{ instead of } \mathfrak{h}\ ]\!] \\
& \lambda(\mathfrak{s}, \mathfrak{h}).\ \sup_{v \in \mathbb{Z}} \max \\
& \qquad \{\, ([E \hookrightarrow v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot \mathrm{X}\,[x/v]\,(\mathfrak{s}, \mathfrak{h}_1)) \cdot \mathrm{Y}(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \\
=\ & [\![\ \text{Definition of } \star\ ]\!] \\
& \sup_{v \in \mathbb{Z}} ([E \hookrightarrow v] \cdot \mathrm{X}\,[x/v]) \star \mathrm{Y} \\
=\ & [\![\ v \text{ fresh, does not occur in } \mathrm{Y}\ ]\!] \\
& \left(\sup_{v \in \mathbb{Z}} E \hookrightarrow v \cdot \mathrm{X}\,[x/v] \right) \star \mathrm{Y} \\
=\ & [\![\ \text{Figure 7.1}\ ]\!] \\
& \mathsf{wp}\,[x := <E>]\,(\mathrm{X}) \star \mathrm{Y}.
\end{aligned}
$$

The case $<E> := E'$.

$$
\begin{aligned}
& \mathsf{wp}\left[<E> := E'\right](X \star Y) \\
= \;& [\![\text{ Figure 7.1 }]\!] \\
& [E \mapsto -] \star \left(\left[E \mapsto E'\right] \mathrel{-\!\!\star} (X \star Y)\right) \\
= \;& [\![\text{ elementary algebra }]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \left(\left[E \mapsto E'\right] \mathrel{-\!\!\star} (X \star Y)\right)(s,h) \\
= \;& [\![\text{ Theorem 6.11 }]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \inf\left\{\, (X \star Y)(s, h \uplus h') \;\middle|\; h \# h', s, h' \models \left[E \mapsto E'\right] \,\right\} \\
= \;& [\![\text{ Definition of } \star \,]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \inf\{ \\
& \qquad \max\left\{\, X(s,h_1) \cdot Y(s,h_2) \;\middle|\; h \uplus h' = h_1 \uplus h_2 \,\right\} \\
& \qquad \mid h \# h', s, h' \models \left[E \mapsto E'\right]\} \\
= \;& [\![\text{ replace max by sup for non-empty finite set }]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \inf\{ \\
& \qquad \sup\left\{\, X(s,h_1) \cdot Y(s,h_2) \;\middle|\; h \uplus h' = h_1 \uplus h_2 \,\right\} \\
& \qquad \mid h \# h', s, h' \models \left[E \mapsto E'\right]\} \\
\succeq \;& [\![\text{ choose } h' \subseteq h_1 \,]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \inf\{ \\
& \qquad \sup\left\{\, X(s,h_1 \star h') \cdot Y(s,h_2) \;\middle|\; h = h_1 \uplus h_2 \,\right\} \\
& \qquad \mid h \# h', s, h' \models \left[E \mapsto E'\right]\} \\
\succeq \;& \left[\!\!\left[\inf_{a \in A} \sup_{b \in B} f(a,b) \geq \sup_{b \in B} \inf_{a \in A} f(a,b) \right]\!\!\right] \\
& [E \mapsto -] \star \lambda(s,h).\, \sup\{ \\
& \qquad \inf\left\{\, X(s,h_1 \star h') \cdot Y(s,h_2) \;\middle|\; h \# h', s, h' \models \left[E \mapsto E'\right] \,\right\} \\
& \qquad \mid h = h_1 \uplus h_2\} \\
= \;& [\![\text{ elementary algebra (} Y \text{ does not depend on } h' \text{) }]\!] \\
& [E \mapsto -] \star \lambda(s,h).\, \sup\{ \\
& \qquad \inf\left\{\, X(s,h_1 \star h') \;\middle|\; h \# h', s, h' \models \left[E \mapsto E'\right] \,\right\} \cdot Y(s,h_2) \\
& \qquad \mid h = h_1 \uplus h_2\} \\
= \;& [\![\text{ Theorem 6.11 }]\!]
\end{aligned}
$$

$$
\begin{aligned}
& [E \mapsto -] \star \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\{ \\
& \qquad (\left[E \mapsto E'\right] \mathrel{-\!\!\star} \mathrm{X})(\mathfrak{s},\mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \\
& \quad \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2\} \\
= \ & [\![\text{ the set of partitions of } \mathfrak{h} \text{ is non-empty and finite }]\!] \\
& [E \mapsto -] \star \lambda(\mathfrak{s},\mathfrak{h}).\ \max\{ \\
& \qquad (\left[E \mapsto E'\right] \mathrel{-\!\!\star} \mathrm{X})(\mathfrak{s},\mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \\
& \quad \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2\} \\
= \ & [\![\text{ Definition of } \star \text{ }]\!] \\
& [E \mapsto -] \star (\left[E \mapsto E'\right] \mathrel{-\!\!\star} \mathrm{X}) \star \mathrm{Y} \\
= \ & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\left[<E> := E'\right](\mathrm{X}) \star \mathrm{Y}.
\end{aligned}
$$

The case $x := \mathtt{alloc}(\vec{E})$.

$$
\begin{aligned}
& \mathsf{wp}\left[x := \mathtt{alloc}(\vec{E})\right](\mathrm{X} \star \mathrm{Y}) \\
= \ & [\![\text{ Figure 7.1 }]\!] \\
& \inf_{v \in \mathbb{Z}} \left[v \mapsto \vec{E}\right] \mathrel{-\!\!\star} (\mathrm{X} \star \mathrm{Y})\,[x/v] \\
= \ & [\![\ x \in \mathbf{Vars}\,(\mathrm{Y}) \ ]\!] \\
& \inf_{v \in \mathbb{Z}} \left[v \mapsto \vec{E}\right] \mathrel{-\!\!\star} (\mathrm{X}\,[x/v] \star \mathrm{Y}) \\
= \ & [\![\text{ Theorem 6.11 }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \inf_{v \in \mathbb{Z}} \inf\{(\mathrm{X}\,[x/v] \star \mathrm{Y})(\mathfrak{s},\mathfrak{h} \uplus \mathfrak{h}') \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right]\} \\
= \ & [\![\text{ Definition of } \star \text{ }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \inf_{v \in \mathbb{Z}} \inf\{ \\
& \qquad \max\left\{\, \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h}_1) \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \;\middle|\; \mathfrak{h} \uplus \mathfrak{h}' = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\right\} \\
& \quad \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right]\} \\
\succeq \ & [\![\text{ choose } \mathfrak{h}' \subseteq \mathfrak{h}_1 \ ]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \inf_{v \in \mathbb{Z}} \inf\{ \\
& \qquad \max\left\{\, \mathrm{X}\,[x/v]\,(\mathfrak{s},\mathfrak{h}_1 \uplus \mathfrak{h}') \cdot \mathrm{Y}(\mathfrak{s},\mathfrak{h}_2) \;\middle|\; \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\right\} \\
& \quad \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right]\}
\end{aligned}
$$

$$
\begin{aligned}
&= [\![\text{ replace max by sup for non-empty finite set }]\!] \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \inf_{v\in\mathbb{Z}} \inf\{ \\
&\qquad\qquad \sup\left\{ X[x/v](\mathfrak{s},\mathfrak{h}_1 \uplus \mathfrak{h}') \cdot Y(\mathfrak{s},\mathfrak{h}_2) \;\middle|\; \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \right\} \\
&\qquad\quad \mid \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right] \} \\
&\succeq \left[\!\!\left[\inf_{a\in A} \sup_{b\in B} f(a,b) \geq \sup_{b\in B} \inf_{a\in A} f(a,b) \text{ twice} \right]\!\!\right] \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\{ \inf_{v\in\mathbb{Z}} \\
&\qquad\quad \inf\left\{ X[x/v](\mathfrak{s},\mathfrak{h}_1 \uplus \mathfrak{h}') \cdot Y(\mathfrak{s},\mathfrak{h}_2) \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right] \right\} \\
&\quad \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \} \\
&= [\![\text{ elementary algebra (Y does not depend on } \mathfrak{h}')]\!] \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\{ \inf_{v\in\mathbb{Z}} \\
&\qquad\quad \inf\left\{ X[x/v](\mathfrak{s},\mathfrak{h}_1 \uplus \mathfrak{h}') \;\middle|\; \mathfrak{h} \# \mathfrak{h}' \text{ and } \mathfrak{s},\mathfrak{h}' \models \left[v \mapsto \vec{E}\right] \right\} \cdot Y(\mathfrak{s},\mathfrak{h}_2) \\
&\quad \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \} \\
&= [\![\text{ Theorem 6.11 }]\!] \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\{ \\
&\qquad\quad \inf_{v\in\mathbb{Z}} \left(\left[v \mapsto \vec{E}\right] \mathrel{-\!\!\!\star} X[x/v] \right)(\mathfrak{s},\mathfrak{h}_1) \cdot Y(\mathfrak{s},\mathfrak{h}_2) \\
&\quad \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \} \\
&= [\![\text{ Definition of } \star]\!] \\
&\quad \left(\inf_{v\in\mathbb{Z}} \left[v \mapsto \vec{E}\right] \mathrel{-\!\!\!\star} X[x/v] \right) \star Y \\
&= [\![\text{ Figure 7.1 }]\!] \\
&\quad \mathsf{wp}\left[x := \mathtt{alloc}(\vec{E})\right](X) \star Y.
\end{aligned}
$$

The case $\mathtt{free}(E)$.

$$
\begin{aligned}
&\mathsf{wp}\,[\mathtt{free}(E)]\,(X) \star Y \\
&= [\![\text{ Figure 7.1 }]\!] \\
&\quad ([E \mapsto -] \star X) \star Y \\
&= [\![\text{ Theorem 6.14 (a) }]\!]
\end{aligned}
$$

$$
\begin{aligned}
& [E \mapsto -] \star (X \star Y) \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[\mathtt{free}(E)]\,(X \star Y).
\end{aligned}
$$

The case $C_1 \,;\, C_2$.

$$
\begin{aligned}
& \mathsf{wp}\,[C_1 \,;\, C_2]\,(X) \star Y \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(X)) \star Y \\
\preceq \; & [\![\text{ I.H. on } C_1]\!] \\
& \mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(X) \star Y) \\
\preceq \; & [\![\text{ I.H. on } C_2]\!] \\
& \mathsf{wp}\,[C_1]\,(\mathsf{wp}\,[C_2]\,(X \star Y)) \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[C_1 \,;\, C_2]\,(X \star Y).
\end{aligned}
$$

The case $\{\, C_1 \,\}\ [\, p \,]\ \{\, C_2 \,\}$.

$$
\begin{aligned}
& \mathsf{wp}\,[\{\, C_1 \,\}\ [\, p \,]\ \{\, C_2 \,\}]\,(X) \star Y \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& (p \cdot \mathsf{wp}\,[C_1]\,(X) + (1-p) \cdot \mathsf{wp}\,[C_2]\,(X)) \star Y \\
\preceq \; & [\![\text{ Theorem 6.16 (c) }]\!] \\
& (p \cdot \mathsf{wp}\,[C_1]\,(X)) \star Y + ((1-p) \cdot \mathsf{wp}\,[C_2]\,(X)) \star Y \\
= \; & [\![\text{ Theorem 6.22 (b) }]\!] \\
& p \cdot (\mathsf{wp}\,[C_1]\,(X) \star Y) + (1-p) \cdot (\mathsf{wp}\,[C_2]\,(X) \star Y) \\
\preceq \; & [\![\text{ I.H. for } C_1 \text{ and } C_2]\!] \\
& p \cdot \mathsf{wp}\,[C_1]\,(X \star Y) + (1-p) \cdot \mathsf{wp}\,[C_2]\,(X \star Y) \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& \mathsf{wp}\,[\{\, C_1 \,\}\ [\, p \,]\ \{\, C_2 \,\}]\,(X \star Y).
\end{aligned}
$$

The case $\mathtt{if}\ (\, B \,)\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\}$.

$$
\begin{aligned}
& \mathsf{wp}\,[\mathtt{if}\ (\, B \,)\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\}]\,(X) \star Y \\
= \; & [\![\text{ Figure 7.1 }]\!] \\
& ([B] \cdot \mathsf{wp}\,[C_1]\,(X) + [\neg B] \cdot \mathsf{wp}\,[C_2]\,(X)) \star Y
\end{aligned}
$$

$$
\begin{aligned}
&\preceq \quad [\![\text{ Theorem 6.16 (c) }]\!] \\
&\quad ([B] \cdot \mathsf{wp}\,[C_1]\,(X)) \star Y + ([\neg B] \cdot \mathsf{wp}\,[C_2]\,(X)) \star Y \\
&= \quad [\![\text{ Theorem 6.22 (b) }]\!] \\
&\quad [B] \cdot (\mathsf{wp}\,[C_1]\,(X) \star Y) + [\neg B] \cdot (\mathsf{wp}\,[C_2]\,(X) \star Y) \\
&\preceq \quad [\![\text{ I.H. for } C_1 \text{ and } C_2 \text{ }]\!] \\
&\quad [B] \cdot \mathsf{wp}\,[C_1]\,(X \star Y) + [\neg B] \cdot \mathsf{wp}\,[C_2]\,(X \star Y) \\
&= \quad [\![\text{ Figure 7.1 }]\!] \\
&\quad \mathsf{wp}\,[\texttt{if (} B \texttt{) \{ } C_1 \texttt{ \} else \{ } C_2 \texttt{ \}}]\,(X \star Y).
\end{aligned}
$$

The case `while ( B ) { C }`. Recall from Section 7.2.1 the characteristic function $\mathfrak{W}$ of loop `while ( B ) { C }` with respect to postexpectation Z:

$$\mathfrak{W}_Z \triangleq \lambda I.\ [B] \cdot \mathsf{wp}\,[C]\,(I) + [\neg B] \cdot Z.$$

By Figure 7.1 and Theorem 7.5, there is some ordinal β such that

$$\mathsf{wp}\,[\texttt{while (} B \texttt{) \{ } C \texttt{ \}}]\,(X \star Y) = \mathsf{lfp}\,(\mathfrak{W}_{X \star Y}) = \mathfrak{W}^{\beta}_{X \star Y}(0).$$

To complete the proof, we show by transfinite induction (see Appendix A.2) that, for all ordinals α, we have:

$$\mathfrak{W}^{\alpha}_{X \star Y}(0) \succeq \mathfrak{W}^{\alpha}_{X}(0) \star Y.$$

The claim then follows for $\alpha = \beta$.

The case $\alpha = 0$ is trivial. For $\alpha = 1$, consider the following:

$$
\begin{aligned}
&\quad \mathfrak{W}_{X \star Y}(0) \\
&= \quad [\![\text{ Definition of characteristic function }]\!] \\
&\quad [B] \cdot \mathsf{wp}\,[C]\,(0) + [\neg B] \cdot (X \star Y) \\
&= \quad [\![\text{ Theorem 7.4 (c) }]\!] \\
&\quad [\neg B] \cdot (X \star Y) \\
&= \quad [\![\text{ Theorem 6.22 (b) }]\!] \\
&\quad ([\neg B] \cdot X) \star Y \\
&= \quad [\![\text{ Definition of characteristic function }]\!] \\
&\quad \mathfrak{W}_X(0) \star Y.
\end{aligned}
$$

For successor ordinals, assume that $\mathfrak{W}^{\alpha}_{X\star Y}(0) \succeq \mathfrak{W}^{\alpha}_{X}(0) \star Y$. Then, consider the following:

$$
\begin{aligned}
& \mathfrak{W}^{\alpha+1}_{X\star Y}(0) \\
= \; & [\![\text{ Definition A.13 }]\!] \\
& \mathfrak{W}_{X\star Y}(\mathfrak{W}^{\alpha}_{X\star Y}(0)) \\
= \; & [\![\text{ Definition of characteristic function }]\!] \\
& [B] \cdot \mathsf{wp}\,[C]\,(\mathfrak{W}^{\alpha}_{X\star Y}(0)) + [\neg B] \cdot (X \star Y) \\
\succeq \; & [\![\text{ I.H. }]\!] \\
& [B] \cdot \mathsf{wp}\,[C]\,(\mathfrak{W}^{\alpha}_{X}(0) \star Y) + [\neg B] \cdot (X \star Y) \\
\succeq \; & [\![\text{ I.H. of outer induction }]\!] \\
& [B] \cdot (\mathsf{wp}\,[C]\,(\mathfrak{W}^{\alpha}_{X}(0)) \star Y) + [\neg B] \cdot (X \star Y) \\
= \; & [\![\text{ Theorem 6.22 (b) }]\!] \\
& ([B] \cdot \mathsf{wp}\,[C]\,(\mathfrak{W}^{\alpha}_{X}(0))) \star Y + ([\neg B] \cdot X) \star Y \\
\succeq \; & [\![\text{ Theorem 6.16 (c) }]\!] \\
& ([B] \cdot \mathsf{wp}\,[C]\,(\mathfrak{W}^{\alpha}_{X}(0)) + [\neg B] \cdot X) \star Y \\
= \; & [\![\text{ Definition of characteristic function }]\!] \\
& \mathfrak{W}_{X}(\mathfrak{W}^{\alpha}_{X}(0)) \star Y \\
= \; & [\![\text{ Definition A.13 }]\!] \\
& \mathfrak{W}^{\alpha+1}_{X}(0) \star Y.
\end{aligned}
$$

Finally, let α be a limit ordinal and assume, for all $\gamma \sqsubset \alpha$, that $\mathfrak{W}^{\gamma}_{X\star Y}(0) \succeq \mathfrak{W}^{\gamma}_{X}(0) \star Y$. Then consider the following:

$$
\begin{aligned}
& \mathfrak{W}^{\alpha}_{X\star Y}(0) \\
= \; & [\![\text{ Definition A.13 for limit ordinal } \alpha \text{ }]\!] \\
& \sup \{\, \mathfrak{W}^{\gamma}_{X\star Y}(0) \mid \gamma \sqsubset \alpha \,\} \\
\succeq \; & [\![\text{ I.H. }]\!] \\
& \sup \{\, \mathfrak{W}^{\gamma}_{X}(0) \star Y \mid \gamma \sqsubset \alpha \,\} \\
= \; & [\![\text{ elementary algebra }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ (\sup \{\, \mathfrak{W}^{\gamma}_{X}(0) \star Y \mid \gamma \sqsubset \alpha \,\})\,(\mathfrak{s},\mathfrak{h}) \\
= \; & [\![\text{ Definition of } \star \text{ }]\!] \\
& \lambda(\mathfrak{s},\mathfrak{h}).\ \sup \{\, \max \{\, \mathfrak{W}^{\gamma}_{X}(0)(\mathfrak{s},\mathfrak{h}_1) \cdot Y(\mathfrak{s},\mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \mid \gamma \sqsubset \alpha \,\}
\end{aligned}
$$

$$
\begin{aligned}
&= \llbracket \text{ replace max by sup for non-empty finite set } \rrbracket \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\left\{\ \sup\left\{\ \mathfrak{W}_X^{\gamma}(0)(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2)\ \middle|\ \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\ \right\}\ \middle|\ \gamma\sqsubset\alpha\ \right\} \\
&= \llbracket \text{ elementary algebra (commute suprema) } \rrbracket \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\left\{\ \sup\left\{\ \mathfrak{W}_X^{\gamma}(0)(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2)\ \middle|\ \gamma\sqsubset\alpha\ \right\}\ \middle|\ \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\ \right\} \\
&= \llbracket \text{ elementary algebra (Y does not depend on } \gamma \rrbracket \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\left\{\ \sup\left\{\ \mathfrak{W}_X^{\gamma}(0)(\mathfrak{s},\mathfrak{h}_1)\ \middle|\ \gamma\sqsubset\alpha\ \right\}\cdot Y(\mathfrak{s},\mathfrak{h}_2)\ \middle|\ \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\ \right\} \\
&= \llbracket \text{ Definition A.13 for limit ordinal } \alpha \rrbracket \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \sup\left\{\ \mathfrak{W}_X^{\alpha}(0)(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2)\ \middle|\ \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\ \right\} \\
&= \llbracket \text{ the set of partitions of } \mathfrak{h} \text{ is non-empty and finite } \rrbracket \\
&\quad \lambda(\mathfrak{s},\mathfrak{h}).\ \max\left\{\ \mathfrak{W}_X^{\alpha}(0)(\mathfrak{s},\mathfrak{h}_1)\cdot Y(\mathfrak{s},\mathfrak{h}_2)\ \middle|\ \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2\ \right\} \\
&= \llbracket \text{ Definition of } \star \rrbracket \\
&\quad \mathfrak{W}_X^{\alpha}(0)\star Y.
\end{aligned}
$$

The case $x := F(\vec{E})$. Let $\vec{E} = (E_1,\ldots,E_n)$ for some natural number n. Recall from Section 7.2.2 the characteristic function $\mathfrak{P}_F$ of procedure F:

$$
\begin{aligned}
\mathfrak{P}_F(\theta) \ \triangleq\ & \lambda x'\,\lambda(E_1',\ldots,E_n')\,\lambda X'. \\
& \mathsf{wp}_\theta^F\left[\mathsf{body}\,(F)\right]\left(X'\left[-\boxdot\right]\left[\boxdot x'/\mathsf{out}\right]\right) \\
& \quad \left[x_1/E_1'\left[-\boxdot\right]\right]\ldots\left[x_n/E_n'\left[-\boxdot\right]\right]\left[+\boxdot\right].
\end{aligned}
$$

By Figure 7.1 and Theorem 7.8, there exists an ordinal β such that

$$
\begin{aligned}
\mathsf{wp}\left[x := F(E_1,\ldots,E_n)\right] &= \mathsf{lfp}\,(\mathfrak{P}_F)\,(x)(E_1,\ldots,E_n) \\
&= \mathfrak{P}_F^{\beta}(0)(x)(E_1,\ldots,E_n).
\end{aligned}
$$

Similarly to our treatment of loops, we show by transfinite induction on α that, for every ordinal α, we have:

$$
\mathfrak{P}_F^{\alpha}(0)(x)(E_1,\ldots,E_n)(X\star Y)\ \succeq\ \mathfrak{P}_F^{\alpha}(0)(x)(E_1,\ldots,E_n)(X)\star Y.
$$

As shown below, the case $\alpha = 0$ is straightforward:

$$
\begin{aligned}
& \mathfrak{P}_F^{0}(0)(x)(E_1,\ldots,E_n)(X\star Y) \\
=\ & \llbracket \text{ Definition A.13 } \rrbracket \\
& 0 \\
=\ & \llbracket \text{ D.7.2 } \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
& 0 \star \mathrm{Y} \\
= \; & [\![\text{ Definition A.13 }]\!] \\
& \mathfrak{P}_F^0(0)(x)(E_1, \ldots, E_n)(\mathrm{X}) \star \mathrm{Y}.
\end{aligned}
$$

For a successor ordinal $\alpha + 1$, consider the following:

$$
\begin{aligned}
& \mathfrak{P}_F^{\alpha+1}(0)(x)(E_1, \ldots, E_n)(\mathrm{X} \star \mathrm{Y}) \\
= \; & [\![\text{ Definition A.13; Definition of } \mathfrak{P}_F \text{ }]\!] \\
& \mathsf{wp}^F_{\mathfrak{P}^\alpha_F}[\mathsf{body}\,(F)]\,((\mathrm{X} \star \mathrm{Y})\,[-\boxdot]\,[\boxdot x/\mathsf{out}]) \\
& \qquad [x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
= \; & [\![\text{ Scoping and substitution distribute }]\!] \\
& \mathsf{wp}^F_{\mathfrak{P}^\alpha_F}[\mathsf{body}\,(F)]\,(\mathrm{X}\,[-\boxdot]\,[\boxdot x/\mathsf{out}] \star \mathrm{Y}\,[-\boxdot]\,[\boxdot x/\mathsf{out}]) \\
& \qquad [x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
= \; & [\![\boxdot x \notin \mathbf{Vars}\,(\mathrm{Y}\,[-\boxdot]) \text{ }]\!] \\
& \mathsf{wp}^F_{\mathfrak{P}^\alpha_F}[\mathsf{body}\,(F)]\,(\mathrm{X}\,[-\boxdot]\,[\boxdot x/\mathsf{out}] \star \mathrm{Y}\,[-\boxdot]) \\
& \qquad [x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
\succeq \; & [\![\text{ I.H. (inner for calls, outer for other statements) }]\!] \\
& \mathsf{wp}^F_{\mathfrak{P}^\alpha_F}[\mathsf{body}\,(F)]\,(\mathrm{X}\,[-\boxdot]\,[\boxdot x/\mathsf{out}])\,[x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
& \qquad \star \mathrm{Y}\,[-\boxdot]\,[x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
= \; & [\![\text{ elementary algebra (apply scoping and substitution) }]\!] \\
& \mathsf{wp}^F_{\mathfrak{P}^\alpha_F}[\mathsf{body}\,(F)]\,(\mathrm{X}\,[-\boxdot]\,[\boxdot x/\mathsf{out}])\,[x_1/E_1\,[-\boxdot]] \ldots [x_n/E_n\,[-\boxdot]]\,[+\boxdot] \\
& \qquad \star \mathrm{Y} \\
= \; & [\![\text{ Definition A.13; Definition of } \mathfrak{P}_F \text{ }]\!] \\
& \mathfrak{P}_F^{\alpha+1}(0)(x)(E_1, \ldots, E_n)(\mathrm{X}) \star \mathrm{Y}.
\end{aligned}
$$

Finally, for a limit ordinal α, consider the following:

$$
\begin{aligned}
& \mathfrak{P}_F^{\alpha}(0)(x)(E_1, \ldots, E_n)(\mathrm{X} \star \mathrm{Y}) \\
= \; & [\![\text{ Definition A.13 for } \alpha \text{ limit ordinal }]\!] \\
& \sup \{\, \mathfrak{P}_F^{\gamma}(0)(x)(E_1, \ldots, E_n)(\mathrm{X} \star \mathrm{Y}) \mid \gamma \sqsubseteq \alpha \,\} \\
\succeq \; & [\![\text{ I.H. }]\!] \\
& \sup \{\, \mathfrak{P}_F^{\gamma}(0)(x)(E_1, \ldots, E_n)(\mathrm{X}) \star \mathrm{Y} \mid \gamma \sqsubseteq \alpha \,\}
\end{aligned}
$$

$$
\begin{aligned}
&= [\![\text{ elementary algebra }]\!] \\
&\quad \sup \left\{ \mathfrak{P}_F^{\gamma}(0)(x)(E_1, \ldots, E_n)(\mathrm{X}) \;\middle|\; \gamma \sqsubseteq \alpha \right\} \star \mathrm{Y} \\
&= [\![\text{ Definition A.13 for } \alpha \text{ limit ordinal }]\!] \\
&\quad \mathfrak{P}_F^{\alpha}(0)(x)(E_1, \ldots, E_n)(\mathrm{X}) \star \mathrm{Y}. \qquad \square
\end{aligned}
$$

We adhere to the notational conventions in Assumption 8.1.

Then, for all *expectations* $\mathrm{X}, \mathrm{Y}, \mathrm{Z}, \mathrm{X}', \mathrm{Y}'$, *precise expectations* X_T, *pure expectations*, $\mathrm{X}_P, \mathrm{Y}_P$, *strictly-exact expectations*, $\mathrm{X}_C, \mathrm{Y}_C$, *predicates* Q, *precise predicates* Q_T, *strictly-exact predicates* Q_C, *variables* x, *integers* v, *natural numbers* n, and *expressions* E, E', E'', E_1, E_2, etc., the following (in)equalities hold:

D.1 Proof Rules for General Expectations

D.1.1 $\mathrm{X} \star (\mathrm{Y} \star \mathrm{Z}) \;=\; (\mathrm{X} \star \mathrm{Y}) \star \mathrm{Z}$ [Theorem 6.14 (a)]

D.1.2 $\mathrm{X} \star [\mathbf{emp}] \;=\; [\mathbf{emp}] \star \mathrm{X} \;=\; \mathrm{X}$ [Theorem 6.14 (b)]

D.1.3 $\mathrm{X} \star \mathrm{Y} \;=\; \mathrm{Y} \star \mathrm{X}$ [Theorem 6.14 (c)]

D.1.4 $\mathrm{X} \preceq \mathrm{Z} \quad \text{implies} \quad \mathrm{X} \star \mathrm{Y} \preceq \mathrm{Z} \star \mathrm{Y}$ [Theorem 6.15]

D.1.5 $\mathrm{Y} \;\preceq\; \mathrm{Y}' \quad \text{implies} \quad \mathrm{X} \mathrel{-\!\!\star} \mathrm{Y} \;\preceq\; \mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}'$ [Theorem 6.20]

D.1.6 $\mathrm{X} \;\preceq\; \mathrm{X}' \quad \text{implies} \quad \mathrm{X}' \mathrel{-\!\!\star} \mathrm{Y} \;\preceq\; \mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}$ [Theorem 6.20]

D.1.7 $\mathrm{X} \star \max\{\,\mathrm{Y},\, \mathrm{Z}\,\} \;=\; \max\{\,\mathrm{X} \star \mathrm{Y},\, \mathrm{X} \star \mathrm{Z}\,\}$ [Theorem 6.16 (a)]

D.1.8 $\mathrm{X} \star \min\{\,\mathrm{Y},\, \mathrm{Z}\,\} \;\preceq\; \min\{\,\mathrm{X} \star \mathrm{Y},\, \mathrm{X} \star \mathrm{Z}\,\}$ [Theorem 6.16 (b)]

D.1.9 $\mathrm{X} \star (\mathrm{Y} + \mathrm{Z}) \;\preceq\; (\mathrm{X} \star \mathrm{Y}) + (\mathrm{X} \star \mathrm{Z})$ [Theorem 6.16 (c)]

D.1.10 $[Q] \star (\mathrm{Y} \cdot \mathrm{Z}) \;\preceq\; \big([Q] \star \mathrm{Y}\big) \cdot \big([Q] \star \mathrm{Z}\big)$ [Theorem 6.16 (d)]

D.1.11 $x \notin \mathbf{Vars}(\mathrm{X})$ implies [Theorem 6.16 (e)]
$\mathrm{X} \star \sup_{v \in \mathbb{Z}} \mathrm{Y}\,[x/v] = \sup_{v \in \mathbb{Z}} (\mathrm{X} \star \mathrm{Y})\,[x/v]$

D.1.12 $x \notin \mathbf{Vars}(\mathrm{X})$ implies [Theorem 6.16 (f)]
$\mathrm{X} \star \inf_{v \in \mathbb{Z}} \mathrm{Y}\,[x/v] \preceq \inf_{v \in \mathbb{Z}} (\mathrm{X} \star \mathrm{Y})\,[x/v]$

D.1.13 $[\mathbf{emp}] \cdot (\mathrm{X} \star \mathrm{Y}) \;=\; ([\mathbf{emp}] \cdot \mathrm{X}) \star ([\mathbf{emp}] \cdot \mathrm{Y})$ [Theorem 6.17 (a)]

D.1.14 $[E \mapsto E'] \cdot (\mathrm{X} \star \mathrm{Y}) = \max\{([E \mapsto E'] \cdot \mathrm{X}) \star ([\mathbf{emp}] \cdot \mathrm{Y}), ([\mathbf{emp}] \cdot \mathrm{X}) \star ([E \mapsto E'] \cdot \mathrm{Y})\}$ [Theorem 6.17 (b)]

D.1.15 $\mathbf{size} \cdot (\mathrm{X} \star \mathrm{Y}) \preceq ((\mathbf{size} \cdot \mathrm{X}) \star \mathrm{Y}) + (\mathrm{X} \star (\mathbf{size} \cdot \mathrm{Y}))$ [Theorem 6.17 (c)]

D.1.16 $\mathrm{X} \star \mathrm{Y} \preceq \mathrm{Z}$ iff $\mathrm{X} \preceq \mathrm{Y} \mathrel{-\!\!\star} \mathrm{Z}$ [Theorem 6.18]

D.1.17 $\mathrm{X} \star (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}) \preceq \mathrm{Y}$ [Corollary 6.19]

D.1.18 $\mathrm{X} \mathrel{-\!\!\star} \max\{\mathrm{Y}, \mathrm{Z}\} \succeq \max\{\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}, \mathrm{X} \mathrel{-\!\!\star} \mathrm{Z}\}$ [Theorem 6.21 (a)]

D.1.19 $\mathrm{X} \mathrel{-\!\!\star} \min\{\mathrm{Y}, \mathrm{Z}\} = \min\{\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}, \mathrm{X} \mathrel{-\!\!\star} \mathrm{Z}\}$ [Theorem 6.21 (b)]

D.1.20 $\mathrm{X} \mathrel{-\!\!\star} (\mathrm{Y} + \mathrm{Z}) \succeq (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y}) + (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Z})$ [Theorem 6.21 (c)]

D.1.21 $[Q] \mathrel{-\!\!\star} (\mathrm{Y} \cdot \mathrm{Z}) \succeq ([Q] \mathrel{-\!\!\star} \mathrm{Y}) \cdot ([Q] \cdot \mathrm{Z})$ [Theorem 6.21 (d)]

D.1.22 $x \notin \mathbf{Vars}(\mathrm{X})$ implies $\mathrm{X} \mathrel{-\!\!\star} (\sup_{v \in \mathbb{Z}} \mathrm{Y}[x/v]) \succeq \sup_{v \in \mathbb{Z}} (\mathrm{X} \mathrel{-\!\!\star} \mathrm{Y})[x/v]$ [Theorem 6.21 (e)]

D.1.23 $x \notin \mathbf{Vars}(\mathrm{X})$ implies $\mathrm{X} \mathrel{-\!\!\star} \inf_{v \in \mathbb{Z}} \mathrm{Y}[x/v] = \inf_{v \in \mathbb{Z}} (\mathrm{X} \star \mathrm{Y})[x/v]$ [Theorem 6.21 (f)]

D.1.24 $\mathrm{X} \preceq \mathrm{X} \star 1$ [Theorem 6.29 (a)]

D.1.25 $1 \mathrel{-\!\!\star} \mathrm{X} \preceq \mathrm{X}$ [Theorem 6.29 (b)]

D.1.26 $[Q] \mathrel{-\!\!\star} \mathrm{X} = [Q] \mathrel{-\!\!\star} (([Q] \star 1) \cdot \mathrm{X})$ [Theorem 6.30 (b)]

D.2 Proof Rules for Pure Expectations

D.2.1 $\mathrm{X}_P \cdot \mathrm{Y} \preceq \mathrm{X}_P \star \mathrm{Y}$ [Theorem 6.22 (a)]

D.2.2 $(\mathrm{X}_P \cdot \mathrm{Y}) \star \mathrm{Z} = \mathrm{X}_P \cdot (\mathrm{Y} \star \mathrm{Z})$ [Theorem 6.22 (b)]

D.2.3 $\mathrm{X}_P \mathrel{-\!\!\star} \mathrm{Y} \preceq [\mathrm{X}_P = 0] \cdot \infty + [\mathrm{X}_P \neq 0] \cdot \mathrm{Y}/\mathrm{X}_P.$ [Theorem 6.22 (c)]

D.2.4 $\mathrm{X}_P \cdot \mathrm{Y}_P = \mathrm{X}_P \star \mathrm{Y}_P$ [Theorem 6.22 (d)]

D.2.5 $\mathrm{X}_P \mathrel{-\!\!\star} \mathrm{Y}_P = [\mathrm{X}_P = 0] \cdot \infty + [\mathrm{X}_P \neq 0] \cdot \mathrm{Y}_P/\mathrm{X}_P$ [Theorem 6.22 (e)]

D.3 Proof Rules for Precise Expectations

D.3.1 $\mathrm{X}_T \star \min\{\,\mathrm{Y},\,\mathrm{Z}\,\} \;=\; \min\{\,\mathrm{X}_T \star \mathrm{Y},\,\mathrm{X}_T \star \mathrm{Z}\,\}$ [Theorem 6.25 (a)]

D.3.2 $\mathrm{X}_T \star (\mathrm{Y} + \mathrm{Z}) \;=\; (\mathrm{X}_T \star \mathrm{Y}) + (\mathrm{X}_T \star \mathrm{Z})$ [Theorem 6.25 (b)]

D.3.3 $[Q_T] \star (\mathrm{Y} \cdot \mathrm{Z}) \;=\; \big([Q_T] \star \mathrm{Y}\big) \cdot \big([Q_T] \star \mathrm{Z}\big)$ [Theorem 6.25 (c)]

D.3.4 $x \notin \mathbf{Vars}\,(\mathrm{X}_T)$ implies [Theorem 6.25 (d)]
$\mathrm{X}_T \star \inf_{v\in\mathbb{Z}} \mathrm{Y}\,[x/v] = \inf_{v\in\mathbb{Z}} (\mathrm{X}_T \star \mathrm{Y})\,[x/v]$

D.3.5 $\mathbf{size} \cdot (\mathrm{X}_T \star \mathrm{Y}) \;=\; ((\mathbf{size} \cdot \mathrm{X}_T) \star \mathrm{Y}) + (\mathrm{X}_T \star (\mathbf{size} \cdot \mathrm{Y}))$ [Theorem 6.26]

D.4 Proof Rules for Strictly-Exact Expectations

D.4.1 $\mathrm{X}_C \longrightarrow\!\!\star \max\{\,\mathrm{Y},\,\mathrm{Z}\,\} \;=\; \max\{\,\mathrm{X}_C \longrightarrow\!\!\star \mathrm{Y},\,\mathrm{X}_C \longrightarrow\!\!\star \mathrm{Z}\,\}$ [Theorem 6.28 (a)]

D.4.2 $\mathrm{X}_C \longrightarrow\!\!\star (\mathrm{Y} + \mathrm{Z}) \;=\; (\mathrm{X}_C \longrightarrow\!\!\star \mathrm{Y}) + (\mathrm{X}_C \longrightarrow\!\!\star \mathrm{Z})$ [Theorem 6.28 (b)]

D.4.3 $[P_C] \longrightarrow\!\!\star (\mathrm{Y} \cdot \mathrm{Z}) \;=\; ([P_C] \longrightarrow\!\!\star \mathrm{Y}) \cdot ([P_C] \longrightarrow\!\!\star \mathrm{Z})$ [Theorem 6.28 (c)]

D.4.4 $x \notin \mathbf{Vars}\,(\mathrm{X}_C)$ implies [Theorem 6.28 (d)]
$\mathrm{X}_C \longrightarrow\!\!\star (\sup_{v\in\mathbb{Z}} \mathrm{Y}\,[x/v]) \;=\; \sup_{v\in\mathbb{Z}} (\mathrm{X}_C \longrightarrow\!\!\star \mathrm{Y})\,[x/v]$

D.4.5 $[P_C] \star ([P_C] \longrightarrow\!\!\star \mathrm{X}) = ([P_C] \star 1) \cdot \mathrm{X}$ [Theorem 6.30 (a)]

D.5 Proof Rules for Atomic Expectations

D.5.1 $[\mathbf{emp}] \cdot \mathbf{size} \;=\; 0$ [Theorem 6.4 (a)]

D.5.2 $[E \mapsto E'] \cdot \mathbf{size} \;=\; [E \mapsto E']$ [Theorem 6.4 (b)]

D.5.3 $[E \mapsto E'] \;=\; [E \neq 0] \cdot [E \mapsto E']$ [Theorem 6.4 (b)]

D.5.4 $[E_1 \mapsto E_2] \cdot [E_3 \mapsto E_4] \;=\; [E_1 = E_3] \cdot [E_2 = E_4] \cdot [E_1 \mapsto E_2]$ [Theorem 6.4 (c)]

D.5.5 $[E \mapsto E'] \star [E \mapsto E''] \;=\; 0$ [Theorem 6.7]

D.5.6 $[E \mapsto E'] \star \mathbf{size} \;=\; [E \hookrightarrow E'] \cdot (\mathbf{size} - 1)$ [Theorem 6.31 (a)]

D.5.7 $[E \mapsto E'] \longrightarrow\!\!\star \mathbf{size} \;=\; 1 + \mathbf{size} + [E \hookrightarrow -] \cdot \infty$ [Theorem 6.31 (b)]

D.5.8 $[E \mapsto E'] \longrightarrow\!\!\star ([E \mapsto E'] \star \mathrm{X}) \;=\; \mathrm{X} + [E \hookrightarrow -] \cdot \infty$ [Theorem 6.31 (c)]

D.5.9 $[E \hookrightarrow E'] \cdot (\mathrm{X} \star \mathrm{Y})$ [Theorem 6.31 (d)]
$= \max\{\,([E \hookrightarrow E'] \cdot \mathrm{X}) \star \mathrm{Y},\, \mathrm{X} \star ([E \hookrightarrow E'] \cdot \mathrm{Y})\,\}$

D.5.10 $\inf_{v\in\mathbb{Z}} [v \mapsto E] \longrightarrow\!\!\star \mathrm{X} \;=\; \inf_{v\in\mathrm{addr}} [v \mapsto E] \longrightarrow\!\!\star \mathrm{X}$ [Theorem 7.2 (a)]

D.5.11 $\inf_{v\in\mathrm{addr}} [v \hookrightarrow -] \cdot \infty + (1 - [v \hookrightarrow -]) \cdot \mathrm{X} \;=\; \inf_{v\in\mathrm{addr}} \mathrm{X}$ [Theorem 7.2 (b)]

D.6 Facts on Expectations from Calculus and Logic

D.6.1 $[Q] \in \{\, 0, 1 \,\}$

D.6.2 $[Q] \cdot [Q] \;=\; [Q]$

D.6.3 $[Q] \cdot [\neg Q] \;=\; 0$

D.6.4 $[Q] \cdot \mathrm{X} + [\neg Q] \cdot \mathrm{X} \;=\; \mathrm{X}$

D.6.5 $0 \;\preceq\; \mathrm{X} \;\preceq\; \infty$

D.6.6 $n > 1$ implies [By definition]
$[E \mapsto E_1, \ldots, E_n] = [E \mapsto E_1] \star [E + 1 \mapsto E_2, \ldots, E_n]$

D.6.7 $[E \hookrightarrow E'] \;=\; [E \mapsto E'] \star 1$ [By definition]

D.6.8 $[E \mapsto -] \;=\; \sup_{v \in \mathbb{Z}} [E \mapsto v]$ [By definition]

D.6.9 $\mathrm{X} \cdot \mathrm{Y} = 0 \quad \text{implies} \quad \max\{\, \mathrm{X}, \mathrm{Y} \,\} = \mathrm{X} + \mathrm{Y}$ [Lemma 6.5]

D.6.10 $x \notin \mathbf{Vars}\,(\mathrm{X})$ implies $\mathrm{X} \cdot \sup_{v \in \mathbb{Z}} \mathrm{Y}\,[x/v] \;=\; \sup_{v \in \mathbb{Z}} (\mathrm{X} \cdot \mathrm{Y})\,[x/v]$

D.6.11 $\sup_{v \in \mathbb{Z}} ([E = v] \cdot \mathrm{X})\,[x/v] \;=\; \mathrm{X}\,[x/E]$

D.6.12 $\sup_{v \in \mathbb{Z}} (\mathrm{X} + \mathrm{Y}) \;\preceq\; (\sup_{v \in \mathbb{Z}} \mathrm{X}) + (\sup_{v \in \mathbb{Z}} \mathrm{Y})$

D.7 Derived Proof Rules

All rules listed below can be derived using the rules presented so far. The corresponding proofs are provided in Appendix E as indicated for each rule. Their purpose is to enable more high-level proofs as they reflect common idioms encountered while reasoning about programs.

D.7.1 $[E \mapsto E'] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \mapsto E''] \star \mathrm{X})) \;=\; [E \mapsto E'] \star \mathrm{X}$ [Example 6.32]

D.7.2 $0 \star \mathrm{X} \;=\; 0$ [Lemma E.1]

D.7.3 $\mathrm{X}_P \cdot (\mathrm{Y} \star \mathrm{Z}) \;=\; (\mathrm{X}_P \cdot \mathrm{Y}) \star \mathrm{Z} \;=\; \mathrm{Y} \star (\mathrm{X}_P \cdot \mathrm{Z})$ [Lemma E.2]

D.7.4 $[E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \mapsto E'''] \star \mathrm{X}))$ [Lemma E.6]
$= \; [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X}$

D.7.5 $[E \hookrightarrow E'] \cdot [E \mapsto E''] \;=\; [E' = E''] \cdot [E \mapsto E'']$ [Lemma E.4]

D.7.6 $[E \hookrightarrow E'] \cdot [\mathbf{emp}] \;=\; 0$ [Lemma E.7]

D.7.7 $[E \hookrightarrow E'] \cdot ([E \mapsto E''] \star \mathrm{X}) \;=\; [E' = E''] \cdot ([E \mapsto E''] \star \mathrm{X})$ [Lemma E.5]

D.7.8 $[E \hookrightarrow E'] \cdot ([E \mapsto -] \star \mathsf{X}) \;=\; [E \mapsto E'] \star \mathsf{X}$ [Lemma E.8]

D.7.9 $[E \mapsto E'] \mathrel{-\!\!\star} \mathsf{X}_P \;=\; [E \hookrightarrow -] \cdot \infty + (1 - [E \hookrightarrow -]) \cdot \mathsf{X}_P$ [Lemma E.9]

D.7.10 $n \geq 1$ implies [Lemma E.10]
$[E \mapsto E_1, \ldots, E_n] \cdot \mathbf{size} \;=\; [E \mapsto E_1, \ldots, E_n] \cdot n$

We show the correctness of proof rules for quantitative entailments that are derived from the existing proof rules studied in Chapter 6. Throughout this appendix, we fix *expectations* X, Y, Z, *precise expectations* X_T, *pure expectations*, X_P, Y_P, *predicates* Q, *precise predicates* Q_T, *variables* x, *integers* v, *natural numbers* n, and *expressions* E, E', etc.

Moreover, we adhere to the notational conventions in Assumption 8.1.

Lemma E.1 $0 \star X = 0$.

Proof.

$$
\begin{aligned}
& 0 \star X \\
= \; & [\![\text{ elementary algebra }]\!] \\
& [0] \star (X \cdot 1) \\
= \; & [\![\text{ D.3.3 }]\!] \\
& ([0] \star X) \cdot ([0] \star 1) \\
= \; & [\![\text{ D.2.4 }]\!] \\
& ([0] \star X) \cdot ([0] \cdot 1) \\
= \; & [\![\text{ elementary algebra }]\!] \\
& ([0] \star X) \cdot 0 \\
= \; & [\![\text{ elementary algebra }]\!] \\
& 0.
\end{aligned}
$$

□

Lemma E.2 $X_P \cdot (Y \star Z) = (X_P \cdot Y) \star Z = Y \star (X_P \cdot Z)$.

Proof.

$$(X_P \cdot Y) \star Z$$

$$
\begin{aligned}
&= [\![\text{D.2.2}]\!] \\
&\quad X_P \cdot (Y \star Z) \\
&= [\![\text{D.1.3}]\!] \\
&\quad X_P \cdot (Z \star Y) \\
&= [\![\text{D.2.2}]\!] \\
&\quad (X_P \cdot Z) \star Y \\
&= [\![\text{D.1.3}]\!] \\
&\quad Y \star (X_P \cdot Z).
\end{aligned}
$$

□

Lemma E.3 $[E \mapsto E'] \star ([E \hookrightarrow E''] \cdot X) = 0.$

Proof.

$$
\begin{aligned}
&\quad [E \mapsto E'] \star ([E \hookrightarrow E''] \cdot X) \\
&= [\![\text{D.3.3}]\!] \\
&\quad ([E \mapsto E'] \star [E \hookrightarrow E'']) \cdot ([E \mapsto E'] \star X) \\
&= [\![\text{D.6.7}]\!] \\
&\quad ([E \mapsto E'] \star ([E \mapsto E''] \star 1)) \cdot ([E \mapsto E'] \star X) \\
&= [\![\text{D.1.1}]\!] \\
&\quad (([E \mapsto E'] \star [E \mapsto E'']) \star 1) \cdot ([E \mapsto E'] \star X) \\
&= [\![\text{D.5.5}]\!] \\
&\quad (0 \star 1) \cdot ([E \mapsto E'] \star X) \\
&= [\![\text{Lemma E.1; elementary algebra}]\!] \\
&\quad 0.
\end{aligned}
$$

□

Lemma E.4 $[E \hookrightarrow E'] \cdot [E \mapsto E''] = [E' = E''] \cdot [E \mapsto E''].$

Proof.

$$
\begin{aligned}
&\quad [E \hookrightarrow E'] \cdot [E \mapsto E''] \\
&= [\![\text{D.6.7}]\!] \\
&\quad ([E \mapsto E'] \star 1) \cdot [E \mapsto E''] \\
&= [\![\text{elementary algebra}]\!] \\
&\quad [E \mapsto E''] \cdot ([E \mapsto E'] \star 1)
\end{aligned}
$$

$$
\begin{aligned}
&= \llbracket \text{ D.1.14 } \rrbracket \\
&\quad \max\{([E \mapsto E''] \cdot [E \mapsto E']) \star ([\mathbf{emp}] \cdot 1), \\
&\qquad\qquad ([\mathbf{emp}] \cdot [E \mapsto E']) \star ([E \mapsto E''] \cdot 1)\} \\
&= \llbracket \text{ elementary algebra; D.5.5; D.1.2 } \rrbracket \\
&\quad \max\{\, [E \mapsto E''] \cdot [E \mapsto E'],\, 0 \,\} \\
&= \llbracket \text{ elementary algebra; D.6.5 } \rrbracket \\
&\quad [E \mapsto E''] \cdot [E \mapsto E'] \\
&= \llbracket \text{ D.5.4 } \rrbracket \\
&\quad [E = E] \cdot [E'' = E'] \cdot [E \mapsto E''] \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad [E' = E''] \cdot [E \mapsto E'']\,. \qquad \square
\end{aligned}
$$

Lemma E.5 $[E \hookrightarrow E'] \cdot ([E \mapsto E''] \star X) \;=\; [E' = E''] \cdot ([E \mapsto E''] \star X)$.

Proof.

$$
\begin{aligned}
&\quad [E \hookrightarrow E'] \cdot ([E \mapsto E''] \star X) \\
&= \llbracket \text{ D.5.9 } \rrbracket \\
&\quad \max\{\, ([E \hookrightarrow E'] \cdot [E \mapsto E'']) \star X,\, [E \mapsto E''] \star ([E \hookrightarrow E'] \cdot X) \,\} \\
&= \llbracket \text{ Lemma E.4 } \rrbracket \\
&\quad \max\{\, ([E' = E''] \cdot [E \mapsto E'']) \star X,\, [E \mapsto E''] \star ([E \hookrightarrow E'] \cdot X) \,\} \\
&= \llbracket \text{ Lemma E.2 } \rrbracket \\
&\quad \max\{\, [E' = E''] \cdot [E \mapsto E''] \star X,\, [E \mapsto E''] \star ([E \hookrightarrow E'] \cdot X) \,\} \\
&= \llbracket \text{ Lemma E.3 } \rrbracket \\
&\quad \max\{\, [E' = E''] \cdot [E \mapsto E''] \star X,\, 0 \,\} \\
&= \llbracket \text{ D.6.5; elementary algebra } \rrbracket \\
&\quad [E' = E''] \cdot ([E \mapsto E''] \star X). \qquad \square
\end{aligned}
$$

Lemma E.6 $[E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \mapsto E'''] \star X)) \;=\; [E'' = E'''] \cdot [E \mapsto -] \star X$.

Proof.

$$[E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \mapsto E'''] \star X))$$

$$
\begin{aligned}
&= [\![\text{D.1.26}]\!] \\
&\quad [E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} (([E \hookrightarrow E''] \star 1) \cdot [E \mapsto E'''] \star \mathrm{X})) \\
&= [\![\text{D.6.7}]\!] \\
&\quad [E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \hookrightarrow E''] \cdot [E \mapsto E'''] \star \mathrm{X})) \\
&= [\![\text{Lemma E.5}]\!] \\
&\quad [E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E'' = E'''] \cdot [E \mapsto E'''] \star \mathrm{X})) \\
&= [\![\text{elementary algebra}]\!] \\
&\quad [E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E'' = E'''] \cdot [E \mapsto E''] \star \mathrm{X})) \\
&= [\![\text{Lemma E.2}]\!] \\
&\quad [E \mapsto -] \star ([E \mapsto E''] \mathrel{-\!\!\star} ([E \mapsto E''] \star ([E'' = E'''] \cdot \mathrm{X}))) \\
&= [\![\text{D.5.8}]\!] \\
&\quad [E \mapsto -] \star (([E'' = E'''] \cdot \mathrm{X}) + [E \hookrightarrow -] \cdot \infty) \\
&= [\![\text{D.3.2}]\!] \\
&\quad [E \mapsto -] \star ([E'' = E'''] \cdot \mathrm{X}) \;+\; [E \mapsto -] \star ([E \hookrightarrow -] \cdot \infty) \\
&= [\![\text{Lemma E.2}]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X} + [E \mapsto -] \star ([E \hookrightarrow -] \cdot \infty) \\
&= [\![\text{D.6.8; elementary algebra } (y,z \text{ fresh})]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X} \\
&\qquad + \sup_{u \in \mathbb{Z}} [E \mapsto y]\,[y/u] \star (\sup_{v \in \mathbb{Z}} [E \hookrightarrow z]\,[z/v] \cdot \infty) \\
&= [\![\text{D.6.10; D.1.11}]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X} \\
&\qquad + \sup_{u \in \mathbb{Z}} \sup_{v \in \mathbb{Z}} [E \mapsto y]\,[y/u] \star ([E \hookrightarrow z]\,[z/v] \cdot \infty) \\
&= [\![\text{elementary algebra}]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X} \\
&\qquad + \sup_{u \in \mathbb{Z}} \sup_{v \in \mathbb{Z}} [E \mapsto u] \star ([E \hookrightarrow v] \cdot \infty) \\
&= [\![\text{Lemma E.3}]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X} + \sup_{u \in \mathbb{Z}} \sup_{v \in \mathbb{Z}} 0 \\
&= [\![\text{elementary algebra}]\!] \\
&\quad [E'' = E'''] \cdot [E \mapsto -] \star \mathrm{X}. \qquad \square
\end{aligned}
$$

Lemma E.7 $[E \hookrightarrow E'] \cdot [\mathbf{emp}] = 0$.

Proof.

$$
\begin{aligned}
& [E \hookrightarrow E'] \cdot [\mathbf{emp}] \\
= \; & [\![\text{D.6.7}]\!] \\
& ([E \mapsto E'] \star 1) \cdot [\mathbf{emp}] \\
= \; & [\![\text{elementary algebra; D.1.13}]\!] \\
& ([\mathbf{emp}] \cdot [E \mapsto E']) \star ([\mathbf{emp}] \cdot 1) \\
= \; & [\![\text{elementary algebra; D.1.2}]\!] \\
& [\mathbf{emp}] \cdot [E \mapsto E'] \\
= \; & [\![\text{D.5.2}]\!] \\
& [\mathbf{emp}] \cdot ([E \mapsto E'] \cdot \mathbf{size}) \\
= \; & [\![\text{elementary algebra}]\!] \\
& ([\mathbf{emp}] \cdot \mathbf{size}) \cdot [E \mapsto E'] \\
= \; & [\![\text{D.5.1}]\!] \\
& 0 \cdot [E \mapsto E'] \\
= \; & [\![\text{elementary algebra}]\!] \\
& 0.
\end{aligned}
$$

□

Lemma E.8 $[E \hookrightarrow E'] \cdot ([E \mapsto -] \star X) = [E \mapsto E'] \star X$.

Proof.

$$
\begin{aligned}
& [E \hookrightarrow E'] \cdot ([E \mapsto -] \star X) \\
= \; & [\![\text{D.5.9}]\!] \\
& \max \{ ([E \hookrightarrow E'] \cdot [E \mapsto -]) \star X, [E \mapsto -] \star ([E \hookrightarrow E'] \cdot X) \} \\
= \; & [\![\text{D.6.8; elementary algebra; D.1.11}]\!] \\
& \max \left\{ ([E \hookrightarrow E'] \cdot [E \mapsto -]) \star X, \sup_{v \in \mathbb{Z}} [E \mapsto v] \star ([E \hookrightarrow E'] \cdot X) \right\} \\
= \; & [\![\text{Lemma E.3}]\!] \\
& \max \left\{ ([E \hookrightarrow E'] \cdot [E \mapsto -]) \star X, \sup_{v \in \mathbb{Z}} 0 \right\}
\end{aligned}
$$

$$
\begin{aligned}
&= [\![\text{ elementary algebra; D.6.5 }]\!] \\
&\quad \left(\left[E \hookrightarrow E'\right] \cdot [E \mapsto -]\right) \star \mathrm{X} \\
&= [\![\text{ D.6.8; elementary algebra; D.6.10 }]\!] \\
&\quad \left(\sup_{v \in \mathbb{Z}} \left[E \hookrightarrow E'\right] \cdot [E \mapsto v]\right) \star \mathrm{X} \\
&= [\![\text{ Lemma E.4 }]\!] \\
&\quad \left(\sup_{v \in \mathbb{Z}} \left[E' = v\right] \cdot [E \mapsto v]\right) \star \mathrm{X} \\
&= [\![\text{ D.6.11; elementary algebra }]\!] \\
&\quad \left[E \mapsto E'\right] \star \mathrm{X}. \qquad \square
\end{aligned}
$$

Lemma E.9 $[E \mapsto E'] \mathrel{-\!\!\star} \mathrm{X}_P \;=\; [E \hookrightarrow -] \cdot \infty + (1 - [E \hookrightarrow -]) \cdot \mathrm{X}_P.$

Proof.

$$
\begin{aligned}
&\left[E \mapsto E'\right] \mathrel{-\!\!\star} \mathrm{X}_P \\
&= [\![\text{ D.1.26; Lemma E.2; elementary algebra }]\!] \\
&\quad \left[E \mapsto E'\right] \mathrel{-\!\!\star} \left[E \mapsto E'\right] \star \mathrm{X}_P \\
&= [\![\text{ D.5.8 }]\!] \\
&\quad \mathrm{X}_P + [E \hookrightarrow -] \cdot \infty \\
&= [\![\text{ elementary algebra }]\!] \\
&\quad ([E \hookrightarrow -] + 1 - [E \hookrightarrow -]) \cdot \mathrm{X}_P + [E \hookrightarrow -] \cdot \infty \\
&= [\![\text{ elementary algebra }]\!] \\
&\quad [E \hookrightarrow -] \cdot \infty + (1 - [E \hookrightarrow -]) \cdot \mathrm{X}_P. \qquad \square
\end{aligned}
$$

Lemma E.10 $n \geq 1$ implies

$$[E \mapsto E_1, \ldots, E_n] \cdot \mathbf{size} \;=\; [E \mapsto E_1, \ldots, E_n] \cdot n.$$

Proof. We first show by induction on the length m of blocks that, for all $1 \leq u \leq v$ with $m = v - u$, we have:

$$[E \mapsto E_u, \ldots, E_v] \cdot \mathbf{size} \;=\; [E \mapsto E_u, \ldots, E_v] \cdot (v - u + 1).$$

For $m = 0$, we have $u = v$. Then consider the following:

$$[E \mapsto E_u] \cdot \mathbf{size}$$

$$
\begin{aligned}
&= \llbracket \text{ D.5.2; elementary algebra } \rrbracket \\
&\quad [E \mapsto E_u] \cdot (v - u + 1).
\end{aligned}
$$

For $m > 0$, we have $u < v$. Then consider the following:

$$
\begin{aligned}
&\quad [E \mapsto E_u, \ldots, E_v] \cdot \mathbf{size} \\
&= \llbracket \text{ D.6.6 } \rrbracket \\
&\quad ([E \mapsto E_u] \star [E + 1 \mapsto E_{u+1}, \ldots, E_v]) \cdot \mathbf{size} \\
&= \llbracket \text{ elementary algebra; D.3.5 } \rrbracket \\
&\quad (\mathbf{size} \cdot [E \mapsto E_u]) \star [E + 1 \mapsto E_{u+1}, \ldots, E_v]) \\
&\qquad + [E \mapsto E_u] \star (\mathbf{size} \cdot [E + 1 \mapsto E_{u+1}, \ldots, E_v]) \\
&= \llbracket \text{ D.5.2 } \rrbracket \\
&\quad [E \mapsto E_u] \star [E + 1 \mapsto E_{u+1}, \ldots, E_v] \\
&\qquad + [E \mapsto E_u] \star (\mathbf{size} \cdot [E + 1 \mapsto E_{u+1}, \ldots, E_v]) \\
&= \llbracket \text{ D.6.6 } \rrbracket \\
&\quad [E \mapsto E_u, \ldots, E_v] \\
&\qquad + [E \mapsto E_u] \star (\mathbf{size} \cdot [E + 1 \mapsto E_{u+1}, \ldots, E_v]) \\
&= \llbracket \text{ elementary algebra; I.H. } \rrbracket \\
&\quad [E \mapsto E_u, \ldots, E_v] \\
&\qquad + [E \mapsto E_u] \star [E + 1 \mapsto E_{u+1}, \ldots, E_v] \cdot (v - (u + 1) + 1) \\
&= \llbracket \text{ elementary algebra; Lemma E.2; D.6.6 } \rrbracket \\
&\quad [E \mapsto E_u, \ldots, E_v] + [E \mapsto E_u, \ldots, E_v] \cdot (v - u) \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad [E \mapsto E_u, \ldots, E_v]) \cdot (v - u + 1).
\end{aligned}
$$

The claim then follows immediately for $u = 1$ and $v = n$. □

F.1 Omitted Calculations in Section 8.2

F.1.1 Proof of Lemma 8.7

We have to show that

$$\begin{aligned}&[\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\ \preceq\ & \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) + [\mathit{sll}\,(\mathsf{hd}, 0)]).\end{aligned}$$

Proof.

$$\begin{aligned}&[\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \mathrm{I}\,[\mathsf{hd}/x]\,[\mathsf{rv}/\mathsf{hd}]\,[x/v]) \\ =\ & [\![\ \text{Definition of I; apply substitution}\ ]\!] \\ & [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \big(\\ & \quad \mathit{len}\,(\mathsf{hd}, 0) \star [\mathit{sll}\,(v, 0)] + 1/2 \cdot \underbrace{[v \neq 0]}_{\preceq 1} \cdot \mathit{len}\,(v, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \big)) \\ \preceq\ & [\![\ \text{Theorem 6.35 (a); D.6.1; D.1.4; D.1.5}\ ]\!] \\ & [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \big(\\ & \quad ([\mathit{sll}\,(\mathsf{hd}, 0)] \cdot \mathbf{size}) \star [\mathit{sll}\,(v, 0)] \\ & \qquad + 1/2 \cdot ([\mathit{sll}\,(v, 0)] \cdot \mathbf{size}) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \big)) \\ \preceq\ & [\![\ \text{D.1.26; D.1.10; D.6.1; D.1.4; D.1.5}\ ]\!] \\ & [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} \big(\\ & \quad (([\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathit{sll}\,(\mathsf{hd}, 0)]) \cdot \mathbf{size}) \star [\mathit{sll}\,(v, 0)]\end{aligned}$$

$$
\begin{aligned}
&\quad + 1/2 \cdot ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star ([\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathbf{sll}\,(\mathsf{hd},0)])) \\
&= [\![\text{ Lemma F.1 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} (\\
&\qquad (([\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathbf{sll}\,(\mathsf{rv},0)]) \cdot \mathbf{size}) \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathbf{sll}\,(\mathsf{rv},0)])) \\
&= [\![\text{ D.3.5; D.5.2; D.3.2 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} (\\
&\qquad [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathbf{sll}\,(\mathsf{rv},0)] \star [\mathbf{sll}\,(v,0)] \\
&\qquad + [\mathsf{hd} \mapsto \mathsf{rv}] \star ([\mathbf{sll}\,(\mathsf{rv},0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathbf{sll}\,(\mathsf{rv},0)])) \\
&= [\![\text{ D.1.3; D.3.2 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathsf{hd} \mapsto \mathsf{rv}] \mathrel{-\!\!\star} ([\mathsf{hd} \mapsto \mathsf{rv}] \star (\\
&\qquad [\mathbf{sll}\,(\mathsf{rv},0)] \star [\mathbf{sll}\,(v,0)] + ([\mathbf{sll}\,(\mathsf{rv},0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(\mathsf{rv},0)]))) \\
&= [\![\text{ D.7.1 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star (\\
&\qquad [\mathbf{sll}\,(\mathsf{rv},0)] \star [\mathbf{sll}\,(v,0)] + ([\mathbf{sll}\,(\mathsf{rv},0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(\mathsf{rv},0)]) \\
&= [\![\text{ D.1.3; D.3.2 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathbf{sll}\,(\mathsf{rv},0)] \star [\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)] \\
&\qquad + ([\mathbf{sll}\,(\mathsf{rv},0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot [\mathsf{hd} \mapsto v] \star ([\mathbf{sll}\,(v,0)] \cdot \mathbf{size}) \star [\mathbf{sll}\,(\mathsf{rv},0)] \\
&= [\![\text{ D.3.5; D.5.2 }]\!] \\
&\quad [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathbf{sll}\,(\mathsf{rv},0)] \star [\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)] \\
&\qquad + ([\mathbf{sll}\,(\mathsf{rv},0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)] \\
&\qquad + 1/2 \cdot ([\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)] \cdot \mathbf{size} \\
&\qquad\quad - [\mathsf{hd} \mapsto v] \star [\mathbf{sll}\,(v,0)]) \star [\mathbf{sll}\,(\mathsf{rv},0)] \\
&= [\![\text{ D.1.3; D.3.2; elementary algebra }]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
&\quad + ([\mathit{sll}\,(\mathsf{rv}, 0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
&\quad + 1/2 \cdot ([\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \cdot \mathbf{size}) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
&\quad - 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
\preceq\ & [\![\text{ D.6.12; elementary algebra }]\!] \\
&1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
&\quad + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathit{sll}\,(\mathsf{rv}, 0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
&\quad + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \cdot \mathbf{size}) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
=\ & [\![\text{ introduce fresh variable } z \text{ for } v \text{ ; D.1.3 }]\!] \\
&1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)])[z/v] \\
&\quad + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} (([\mathit{sll}\,(\mathsf{rv}, 0)] \cdot \mathbf{size}) \star [\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)])[z/v] \\
&\quad + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathit{sll}\,(\mathsf{rv}, 0)] \star ([\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)] \cdot \mathbf{size}))[z/v] \\
=\ & [\![\text{ D.1.11; D.7.3; apply substitution; D.6.10 }]\!] \\
&1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star ([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)]) \\
&\quad + ([\mathit{sll}\,(\mathsf{rv}, 0)] \cdot \mathbf{size}) \star ([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)]) \\
&\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathbf{size} \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)]) \\
\preceq\ & [\![\text{ Definition of } [\mathit{sll}\,(\mathsf{hd}, 0)]]\!] \\
&1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathit{sll}\,(\mathsf{hd}, 0)] + ([\mathit{sll}\,(\mathsf{rv}, 0)] \cdot \mathbf{size}) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathbf{size} \cdot [\mathit{sll}\,(\mathsf{hd}, 0)]) \\
=\ & [\![\text{ Theorem 6.35 (a) }]\!] \\
&1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star [\mathit{sll}\,(\mathsf{hd}, 0)] + \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \mathit{len}\,(\mathsf{hd}, 0) \\
=\ & [\![\text{ elementary algebra; D.3.2 }]\!] \\
&\mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) + [\mathit{sll}\,(\mathsf{hd}, 0)]).
\end{aligned}
$$

□

Lemma F.1 $[\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathit{sll}\,(\mathsf{hd}, 0)] = [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathit{sll}\,(\mathsf{rv}, 0)]$.

Proof.

$$
\begin{aligned}
& [\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathit{sll}\,(\mathsf{hd}, 0)] \\
= \;& [\![\text{ Definition of } [\mathit{sll}\,(\mathsf{hd}, 0)] \,]\!] \\
& [\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot ([\mathsf{hd} = 0] \cdot [\mathbf{emp}] + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)]) \\
= \;& [\![\text{ elementary algebra; introduce fresh variable } z \text{ for } v \,]\!] \\
& [\mathsf{hd} = 0] \cdot [\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathbf{emp}] \\
& \qquad + [\mathsf{hd} \neq 0] \cdot [\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot \sup_{v \in \mathbb{Z}} ([\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)])[z/v] \\
= \;& [\![\text{ D.7.6; D.6.10 }]\!] \\
& [\mathsf{hd} = 0] \cdot 0 + [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathsf{hd} \hookrightarrow \mathsf{rv}] \cdot [\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)])\,[z/v] \\
= \;& [\![\text{ elementary algebra; D.7.7 }]\!] \\
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} ([\mathsf{rv} = z] \cdot [\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)])\,[z/v] \\
= \;& [\![\text{ apply substitution }]\!] \\
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
= \;& [\![\text{ elementary algebra }]\!] \\
& [\mathsf{hd} \neq 0] \cdot [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathit{sll}\,(\mathsf{rv}, 0)] \\
= \;& [\![\text{ D.7.8; D.5.3 }]\!] \\
& [\mathsf{hd} \mapsto \mathsf{rv}] \star [\mathit{sll}\,(\mathsf{rv}, 0)].
\end{aligned}
$$

□

F.1.2 Proof of Lemma 8.8

We have to show that

$$
\begin{aligned}
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v] \\
\preceq \;& \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \;+\; 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) - [\mathit{sll}\,(\mathsf{hd}, 0)]).
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathrm{I}\,[\mathsf{hd}/x]\,[x/v] \\
= \;& [\![\text{ Definition of I; apply substitution }]\!]
\end{aligned}
$$

$$
\begin{array}{ll}
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \\
& \quad \big(\mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(v, 0)] + 1/2 \cdot [v \neq 0] \cdot \mathit{len}\,(v, 0) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \big) \\
\preceq & \llbracket \text{D.4.2; Theorem 6.25 (b); D.6.10} \rrbracket \\
& [\mathsf{hd} \neq 0] \cdot \big(\sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(v, 0)] \\
& \quad + \sup_{v \in \mathbb{Z}} 1/2 \cdot \underbrace{[v \neq 0]}_{\preceq 1} \cdot [\mathsf{hd} \mapsto v] \star \mathit{len}\,(v, 0) \star [\mathit{sll}\,(\mathsf{rv}, 0)] \big) \\
\preceq & \llbracket \text{D.1.3; D.6.1; D.1.4; introduce fresh variable } z \text{ for } v \rrbracket \\
& [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} \mathit{len}\,(\mathsf{rv}, 0) \star \big([\mathsf{hd} \mapsto z] \star [\mathit{sll}\,(z, 0)] \big)\,[z/v] \\
& \quad + 1/2 \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathit{sll}\,(\mathsf{rv}, 0)] \star \big([\mathsf{hd} \mapsto z] \star \mathit{len}\,(z, 0) \big)\,[z/v] \\
= & \llbracket \text{D.1.11; apply substitution; D.7.3} \rrbracket \\
& \mathit{len}\,(\mathsf{rv}, 0) \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \big) \\
& \quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star \mathit{len}\,(v, 0) \big) \\
= & \llbracket \text{Theorem 6.35 (a)} \rrbracket \\
& \mathit{len}\,(\mathsf{rv}, 0) \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \big) \\
& \quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star ([\mathit{sll}\,(v, 0)] \cdot \mathbf{size}) \big) \\
= & \llbracket \text{D.1.15 } ([\mathsf{hd} \mapsto v] \text{ is precise}) \rrbracket \\
& \mathit{len}\,(\mathsf{rv}, 0) \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \big) \\
& \quad + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} \\
& \qquad [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \cdot \mathbf{size} - ([\mathsf{hd} \mapsto v] \cdot \mathbf{size}) \star [\mathit{sll}\,(v, 0)] \big) \\
\preceq & \llbracket \text{D.5.2; elementary algebra; D.6.10} \rrbracket \\
& \mathit{len}\,(\mathsf{rv}, 0) \star \big([\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \big) + 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star \big(\\
& \quad \mathbf{size} \cdot [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \\
& \quad - [\mathsf{hd} \neq 0] \cdot \sup_{v \in \mathbb{Z}} [\mathsf{hd} \mapsto v] \star [\mathit{sll}\,(v, 0)] \big) \\
\preceq & \llbracket \text{Definition of } [\mathit{sll}\,(\mathsf{hd}, 0)] \rrbracket \\
& \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)]
\end{array}
$$

$$
\begin{aligned}
&\quad + \ 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathbf{size} \cdot [\mathit{sll}\,(\mathsf{hd}, 0)] - [\mathit{sll}\,(\mathsf{hd}, 0)]) \\
&= \ [\![\text{ Theorem 6.35 (a) }]\!] \\
&\quad \mathit{len}\,(\mathsf{rv}, 0) \star [\mathit{sll}\,(\mathsf{hd}, 0)] \\
&\quad + \ 1/2 \cdot [\mathit{sll}\,(\mathsf{rv}, 0)] \star (\mathit{len}\,(\mathsf{hd}, 0) - [\mathit{sll}\,(\mathsf{hd}, 0)]).
\end{aligned}
$$

□

F.2 Omitted Calculations in Section 8.3

F.2.1 Proof of Lemma 8.9

We have to show that

$$
\begin{aligned}
& p \cdot [\mathbf{emp}] + (1-p) \cdot ([x \neq 0] \cdot (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \\
& \qquad \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] + [x = 0] \cdot [\mathbf{emp}]) \\
\succeq \ & (1-p)^{1/2 \cdot \mathbf{size}} \cdot [\mathit{tree}\,(x)]\,.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& p \cdot [\mathbf{emp}] + (1-p) \cdot ([x \neq 0] \cdot (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \\
& \qquad \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] + [x = 0] \cdot [\mathbf{emp}]) \\
= \ & [\![\text{ elementary algebra }]\!] \\
& p \cdot [\mathbf{emp}] + (1-p) \cdot [x \neq 0] \cdot (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \\
& \qquad \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] + (1-p) \cdot [x = 0] \cdot [\mathbf{emp}] \\
\succeq \ & [\![\text{ D.6.1 }]\!] \\
& p \cdot [x = 0] \cdot [\mathbf{emp}] + (1-p) \cdot [x \neq 0] \cdot (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \\
& \qquad \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] + (1-p) \cdot [x = 0] \cdot [\mathbf{emp}] \\
= \ & [\![\text{ elementary algebra }]\!] \\
& [x = 0] \cdot [\mathbf{emp}] + (1-p) \cdot [x \neq 0] \cdot (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \\
& \qquad \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] \\
= \ & [\![\text{ elementary algebra }]\!] \\
& [x = 0] \cdot [\mathbf{emp}] + (1-p)^{1/2 \cdot \mathbf{size}} \cdot [x \neq 0] \cdot
\end{aligned}
$$

$$
\begin{aligned}
&\sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\textit{tree}\,(u)] \star [\textit{tree}\,(v)] \\
\succeq\ & \text{⟦ Lemma 8.13; D.6.1 ⟧} \\
& (1-p)^{1/2\cdot\mathbf{size}} \cdot [x = 0] \cdot [\mathbf{emp}] + (1-p)^{1/2\cdot\mathbf{size}} \cdot [x \neq 0] \cdot \\
&\qquad \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\textit{tree}\,(u)] \star [\textit{tree}\,(v)] \\
=\ & \text{⟦ elementary algebra ⟧} \\
& (1-p)^{1/2\cdot\mathbf{size}} \cdot \big([x = 0] \cdot [\mathbf{emp}] + [x \neq 0] \cdot \\
&\qquad \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\textit{tree}\,(u)] \star [\textit{tree}\,(v)] \big) \\
=\ & \text{⟦ Definition of } [\textit{tree}\,(x)] \text{ ⟧} \\
& (1-p)^{1/2\cdot\mathbf{size}} \cdot [\textit{tree}\,(x)].
\end{aligned}
$$

□

F.2.2 Proof of Lemma 8.10

We have to show that

$$
\begin{aligned}
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star ([x+1 \mapsto v] \mathrel{-\!\!\star} \\
&\quad [x \mapsto -] \star [x+1 \mapsto -] \star \\
&\quad\quad ([\textit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star ([\textit{tree}\,(u)] \cdot (1-p)^{1/2\cdot\mathbf{size}}))) \\
&= (1-p)^{1/2\cdot\mathbf{size}-1} \cdot \sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star [\textit{tree}\,(u)] \star [\textit{tree}\,(v)]\,.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star ([x+1 \mapsto v] \mathrel{-\!\!\star} \\
&\quad [x \mapsto -] \star [x+1 \mapsto -] \star \\
&\quad\quad ([\textit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star ([\textit{tree}\,(u)] \cdot (1-p)^{1/2\cdot\mathbf{size}}))) \\
=\ & \text{⟦ D.4.5 ⟧} \\
& \sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \hookrightarrow v] \cdot \\
&\quad [x \mapsto -] \star [x+1 \mapsto -] \star \\
&\quad\quad ([\textit{tree}\,(v)] \cdot (1-p)^{1/2\cdot\mathbf{size}}) \star ([\textit{tree}\,(u)] \cdot (1-p)^{1/2\cdot\mathbf{size}})) \\
=\ & \text{⟦ D.1.3; D.7.8 ⟧}
\end{aligned}
$$

$$
\begin{aligned}
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star [x \mapsto -] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ D.1.3; introduce fresh variable } y \text{ for } v \ ]\!]\\
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} \sup_{v\in\mathbb{Z}} \big([x \mapsto -] \star [x+1 \mapsto y] \star \\
&\quad ([\mathit{tree}\,(y)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\big)\,[y/v]\\
=\ & [\![\text{ D.1.11; apply substitution }]\!]\\
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star ([x \mapsto u] \mathrel{-\!\!\star} [x \mapsto -] \star \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ D.4.5 }]\!]\\
&\sup_{u\in\mathbb{Z}} [x \hookrightarrow u] \cdot [x \mapsto -] \star \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ D.7.8 }]\!]\\
&\sup_{u\in\mathbb{Z}} [x \mapsto u] \star \sup_{v\in\mathbb{Z}} [x+1 \mapsto v] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ introduce fresh variable } y \text{ for } v \text{ D.1.11; apply substitution }]\!]\\
&\sup_{u\in\mathbb{Z}} \sup_{v\in\mathbb{Z}} [x \mapsto u] \star [x+1 \mapsto v] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ elementary algebra; D.6.6 }]\!]\\
&\sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star \\
&\quad ([\mathit{tree}\,(v)] \cdot (1-p)^{1/2\cdot \mathbf{size}}) \star ([\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}))\\
=\ & [\![\text{ Lemma 8.11 }]\!]\\
&\sup_{u,v\in\mathbb{Z}} [x \mapsto u, v] \star \big([\mathit{tree}\,(v)] \star [\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}\big)\\
=\ & [\![\text{ elementary algebra }]\!]\\
&\sup_{u,v\in\mathbb{Z}} ([x \mapsto u, v] \cdot {}^{1-p}/_{1-p}) \star \big([\mathit{tree}\,(v)] \star [\mathit{tree}\,(u)] \cdot (1-p)^{1/2\cdot \mathbf{size}}\big)\\
=\ & [\![\text{ elementary algebra; D.7.3 }]\!]
\end{aligned}
$$

$$
\begin{aligned}
& {}^{1}/_{(1-p)} \cdot \sup_{u,v \in \mathbb{Z}} \left([x \mapsto u, v] \cdot (1-p) \right) \star \\
& \qquad \left([\mathit{tree}\,(v)] \star [\mathit{tree}\,(u)] \cdot (1-p)^{1/2 \cdot \mathbf{size}} \right) \\
= \; & [\![\text{ Lemma 8.12 }]\!] \\
& {}^{1}/_{(1-p)} \cdot \sup_{u,v \in \mathbb{Z}} \left([x \mapsto u, v] \cdot (1-p)^{1/2 \cdot \mathbf{size}} \right) \star \\
& \qquad \left([\mathit{tree}\,(v)] \star [\mathit{tree}\,(u)] \cdot (1-p)^{1/2 \cdot \mathbf{size}} \right) \\
= \; & [\![\text{ Lemma 8.11; elementary algebra }]\!] \\
& {}^{1}/_{(1-p)} \cdot \sup_{u,v \in \mathbb{Z}} (1-p)^{1/2 \cdot \mathbf{size}} \cdot [x \mapsto u, v] \star [\mathit{tree}\,(v)] \star [\mathit{tree}\,(u)] \\
= \; & [\![\text{ elementary algebra }]\!] \\
& (1-p)^{1/2 \cdot \mathbf{size} - 1} \cdot \sup_{u,v \in \mathbb{Z}} [x \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] \,. \qquad \square
\end{aligned}
$$

F.2.3 Proof of Lemma 8.11

We have to show that, for all $v \in \mathbb{R}_{\geq 0}$, $p \in [0,1]$, and expectations $Y, Z \in \mathbb{E}_{\leq 1}$,

$$
(Y \star Z) \cdot p^{v \cdot \mathbf{size}} \;=\; \left(Y \cdot p^{v \cdot \mathbf{size}} \right) \star \left(Z \cdot p^{v \cdot \mathbf{size}} \right).
$$

Proof. For any stack-heap pair $(\mathfrak{s}, \mathfrak{h})$, consider the following:

$$
\begin{aligned}
& \left((Y \star Z) \cdot p^{v \cdot \mathbf{size}} \right) (\mathfrak{s}, \mathfrak{h}) \\
= \; & [\![\text{ elementary algebra }]\!] \\
& (Y \star Z)\,(\mathfrak{s}, \mathfrak{h}) \cdot p^{v \cdot \mathbf{size}}(\mathfrak{s}, \mathfrak{h}) \\
= \; & [\![\text{ Definition 6.6 }]\!] \\
& \max \{\, Y(\mathfrak{s}, \mathfrak{h}_1) \cdot Z(\mathfrak{s}, \mathfrak{h}_2) \mid \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\} \cdot p^{v \cdot \mathbf{size}}(\mathfrak{s}, \mathfrak{h}) \\
= \; & [\![\text{ elementary algebra }]\!] \\
& \max \left\{\, Y(\mathfrak{s}, \mathfrak{h}_1) \cdot Z(\mathfrak{s}, \mathfrak{h}_2) \cdot p^{v \cdot \mathbf{size}}(\mathfrak{s}, \mathfrak{h}) \;\middle|\; \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\right\} \\
= \; & [\![\text{ Definition of } \mathbf{size}(\mathfrak{s}, \mathfrak{h}) \colon |\mathsf{dom}(\mathfrak{h})| = |\mathsf{dom}(\mathfrak{h}_1)| + |\mathsf{dom}(\mathfrak{h}_2)| \,]\!] \\
& \max \left\{\, Y(\mathfrak{s}, \mathfrak{h}_1) \cdot Z(\mathfrak{s}, \mathfrak{h}_2) \cdot p^{v \cdot (|\mathsf{dom}(\mathfrak{h}_1)| + |\mathsf{dom}(\mathfrak{h}_2)|)} \;\middle|\; \mathfrak{h} = \mathfrak{h}_1 \uplus \mathfrak{h}_2 \,\right\} \\
= \; & [\![\text{ elementary algebra }]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\max\left\{ \mathrm{Y}(\mathfrak{s},\mathfrak{h}_1)\cdot p^{v\cdot|\mathsf{dom}(\mathfrak{h}_1)|}\cdot \mathrm{Z}(\mathfrak{s},\mathfrak{h}_2)\cdot p^{v\cdot|\mathsf{dom}(\mathfrak{h}_2)|} \;\middle|\; \mathfrak{h}=\mathfrak{h}_1\uplus\mathfrak{h}_2 \right\}\\
=\;& [\![\ \text{Definition 6.6}\ ]\!]\\
&\left(\left(\mathrm{Y}\cdot p^{v\cdot\mathbf{size}}\right)\star\left(\mathrm{Z}\cdot p^{v\cdot\mathbf{size}}\right)\right)(\mathfrak{s},\mathfrak{h}).
\end{aligned}
$$

□

F.2.4 Proof of Lemma 8.12

We have to show that, for all $v\in\mathbb{R}_{\geq 0}$ and $p\in[0,1]$, we have

$$[E\mapsto E_1,\ldots,E_n]\cdot p^{v\cdot\mathbf{size}} \;=\; [E\mapsto E_1,\ldots,E_n]\cdot p^{v\cdot n}.$$

Proof. For any stack-heap pair $(\mathfrak{s},\mathfrak{h})$, consider the following:

$$
\begin{aligned}
&\left([E\mapsto E_1,\ldots,E_n]\cdot p^{v\cdot\mathbf{size}}\right)(\mathfrak{s},\mathfrak{h})\\
=\;& [\![\ \text{elementary algebra; Definition of } p^{\mathrm{Y}}\ ]\!]\\
&[E\mapsto E_1,\ldots,E_n](\mathfrak{s},\mathfrak{h})\cdot p^{v\cdot\mathbf{size}(\mathfrak{s},\mathfrak{h})}\\
=\;& [\![\ \text{Definition of } \mathbf{size}\ ]\!]\\
&[E\mapsto E_1,\ldots,E_n](\mathfrak{s},\mathfrak{h})\cdot p^{v\cdot|\mathsf{dom}(\mathfrak{h})|}\\
=\;& [\![\ \text{if } \mathsf{dom}(\mathfrak{h})\neq n \text{ then } [E\mapsto E_1,\ldots,E_n](\mathfrak{s},\mathfrak{h})=0\ ]\!]\\
&[E\mapsto E_1,\ldots,E_n](\mathfrak{s},\mathfrak{h})\cdot p^{v\cdot n}\\
=\;& [\![\ \text{elementary algebra}\ ]\!]\\
&\left([E\mapsto E_1,\ldots,E_n]\cdot p^{v\cdot n}\right)(\mathfrak{s},\mathfrak{h}).
\end{aligned}
$$

□

F.2.5 Proof of Lemma 8.13

We have to show that, for all $p\in[0,1]$, we have $[\mathbf{emp}]=[\mathbf{emp}]\cdot p^{\mathbf{size}}$.

Proof. For any stack-heap pair $(\mathfrak{s},\mathfrak{h})$, consider the following:

$$
\begin{aligned}
&\left([\mathbf{emp}]\cdot p^{\mathbf{size}}\right)(\mathfrak{s},\mathfrak{h})\\
=\;& [\![\ \text{elementary algebra}\ ]\!]\\
&[\mathbf{emp}](\mathfrak{s},\mathfrak{h})\cdot p^{\mathbf{size}}(\mathfrak{s},\mathfrak{h})\\
=\;& \left[\!\!\left[\ \text{Definition of } p^{\mathbf{size}}\ \right]\!\!\right]
\end{aligned}
$$

$$
\begin{aligned}
& [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}) \cdot p^{\mathbf{size}(\mathfrak{s},\mathfrak{h})} \\
= \;& [\![\text{ Definition of } \mathbf{size} \,]\!] \\
& [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}) \cdot p^{|\mathsf{dom}(h)|} \\
= \;& [\![\,[\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}) = 1 \text{ iff } \mathsf{dom}(h) = \emptyset \text{ and } [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}) = 0 \text{ otherwise} \,]\!] \\
& [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}) \cdot p^{0} \\
= \;& [\![\text{ elementary algebra }]\!] \\
& [\mathbf{emp}]\,(\mathfrak{s},\mathfrak{h}).
\end{aligned}
$$

□

F.3 Omitted Calculations in Section 8.4

Before we verify the proposed invariant I, let us introduce a convenient fact on permutations. For every permutation $\pi \in \mathsf{Perm}(u+1, v)$ and every integer $k \in \{0, 1, \ldots, v-u\}$, we denote by π_k the permutation in $\mathsf{Perm}(u, v)$ which is defined as π except that u is mapped to $\pi(u+k)$ and $u+k$ is mapped to u, respectively. Formally, the permutation π_k is given by:

$$
\pi_k(w) \triangleq \begin{cases} \pi(u+k), & \text{if } w = u \\ u, & \text{if } w = u+k \\ \pi(w), & \text{otherwise.} \end{cases}
$$

Furthermore, we exploit the following fact on permutations:

Lemma F.2 For all integers $u, v \in \mathbb{Z}$ with $u < v$, we have

$$
\mathsf{Perm}(u, v) \;=\; \bigcup_{\pi \in \mathsf{Perm}(u+1, v)} \{\, \pi_k \mid 0 \le k \le v-u \,\}.
$$

Proof. By complete induction on $m = v - u \ge 1$. □

We are now in a position to show that I is indeed an invariant.

Proof (Lemma 8.14). By Theorem 7.7, it suffices to prove for the characteristic function $\mathfrak{W}$ of the above loop with respect to postexpectation

$$
\mathrm{X} \;=\; \bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m]
$$

that $\mathfrak{W}(\mathrm{I}) \preceq \mathrm{I}$ holds. To this end, consider the following:

$$
\mathfrak{W}(\mathrm{I})
$$

$$
\begin{aligned}
&= \llbracket \text{Definition of characteristic function } \mathfrak{W} \rrbracket \\
&\quad [0 \le i < n] \cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I}) \\
&\quad + \ [\neg(0 \le i < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \\
&= \llbracket \text{D.6.9} \rrbracket \\
&\quad \max\{\, [0 \le i < n] \cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I})\,, \\
&\qquad\qquad [\neg(0 \le i < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \,\} \\
&\preceq \llbracket \text{Lemma F.3} \rrbracket \\
&\quad \max\{\, [0 \le i < n] \cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I})\,, \mathrm{I} \,\} \\
&\preceq \llbracket \text{Lemma F.4} \rrbracket \\
&\quad \max\{\, \mathrm{I}, \mathrm{I} \,\} \\
&= \llbracket \text{elementary algebra} \rrbracket \\
&\quad \mathrm{I}.
\end{aligned}
$$

□

Lemma F.3 $[\neg(0 \le i < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \ \preceq\ \mathrm{I}.$

Proof.

$$
\begin{aligned}
&\quad [\neg(0 \le i < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \\
&\preceq \llbracket \text{elementary algebra } (0 \preceq \mathrm{Z} \text{ for all } \mathrm{Z} \in \mathbb{E}) \rrbracket \\
&\quad [\neg(0 \le i < n)] \cdot \overset{n-1}{\underset{m=0}{\bigstar}} [\mathsf{array} + m \mapsto a_m] \\
&\quad + \frac{[0 \le i < n]}{(n-i)!} \cdot \sum_{\pi \in \mathsf{Perm}(i, n-1)} \mathrm{Z}_\pi^i \\
&= \llbracket \text{Definition of I} \rrbracket \\
&\quad \mathrm{I}.
\end{aligned}
$$

□

Lemma F.4 $[0 \le i < n] \cdot \mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{I}) \ \preceq\ \mathrm{I}.$

Proof.

$$
\begin{aligned}
& [0 \le i < n] \cdot \mathsf{wp}\,[\mathsf{C}_{\mathsf{body}}]\,(\mathrm{I}) \\
= \;& [\![\text{ D.6.2; Theorem 7.11 }]\!] \\
& [0 \le i < n] \cdot \mathsf{wp}\,[\mathsf{C}_{\mathsf{body}}]\,([0 \le i < n] \cdot \mathrm{I}) \\
= \;& [\![\text{ Definition of I }]\!] \\
& [0 \le i < n] \cdot \mathsf{wp}\,[\mathsf{C}_{\mathsf{body}}]\,\big([0 \le i < n] \cdot \big(\\
& \qquad [\neg(0 \le i < n)] \cdot \bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m] \\
& \qquad + \frac{[0 \le i < n]}{(n-i)!} \cdot \sum_{\pi \in \mathsf{Perm}(i,n-1)} \mathsf{Z}_\pi^i \\
& \big) \big) \\
= \;& [\![\text{ elementary algebra; D.6.3 }]\!] \\
& [0 \le i < n] \cdot \mathsf{wp}\,[\mathsf{C}_{\mathsf{body}}] \left(\sum_{\pi \in \mathsf{Perm}(i,n-1)} \frac{[0 \le i < n]}{(n-i)!} \cdot \mathsf{Z}_\pi^i \right) \\
\preceq \;& \left[\!\!\left[\text{ Figure F.1 (page 436) ; apply substitution } \left(j \notin \mathbf{Vars}\left(\mathsf{Z}_\pi^{i+1} \right) \right) \right]\!\!\right] \\
& [0 \le i < n] \cdot \sum_{k=0}^{n-1-i} \frac{1}{n-i} \cdot \\
& \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
& \quad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
& \qquad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
& \qquad\quad [\mathsf{array} + i + k \mapsto -] \star \big([\mathsf{array} + i + k \mapsto u] \mathrel{-\!\!\star} \\
& \qquad\qquad \sum_{\pi \in \mathsf{Perm}(i+1,n-1)} \frac{[0 \le i+1 < n]}{(n-(i+1))!} \cdot \mathsf{Z}_\pi^{i+1} \big) \big) \big) \big) \\
\preceq \;& [\![\text{ notice } \mathsf{Perm}(.,.) \text{ is finite; repeatedly apply D.4.2; D.3.2; D.6.12 }]\!] \\
& [0 \le i < n] \cdot \sum_{k=0}^{n-1-i} \frac{1}{n-i} \sum_{\pi \in \mathsf{Perm}(i+1,n-1)} \\
& \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star}
\end{aligned}
$$

$$
\begin{aligned}
&\sup_{v\in\mathbb{Z}} [\mathsf{array}+i+k\mapsto v] \star \big([\mathsf{array}+i+k\mapsto v] \mathrel{-\!\!\star} \\
&\quad [\mathsf{array}+i\mapsto -] \star \big([\mathsf{array}+i\mapsto v] \mathrel{-\!\!\star} \\
&\quad\quad [\mathsf{array}+i+k\mapsto -] \star \big([\mathsf{array}+i+k\mapsto u] \mathrel{-\!\!\star} \\
&\quad\quad\quad \frac{[0\le i+1<n]}{(n-(i+1))!} \cdot Z_\pi^{i+1}\big)\big)\big)\big)
\end{aligned}
$$

$=$ ⟦ Figure F.1 (page 436); elementary algebra (swap finite sums) ⟧

$$
\begin{aligned}
&[0\le i<n]\cdot \sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \frac{1}{n-i}\cdot \\
&\quad \mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array},i,j)] \left(\frac{[0\le i+1<n]}{(n-(i+1))!} \cdot Z_\pi^{i+1} \right) [j/i+k]
\end{aligned}
$$

$=$ ⟦ Theorem 7.11; apply substitution ⟧

$$
\begin{aligned}
&[0\le i<n]\cdot \sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \frac{1}{n-i} \cdot \frac{[0\le i+1<n]}{(n-(i+1))!} \cdot \\
&\quad \mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array},i,j)] \left(Z_\pi^{i+1} \right) [j/i+k]
\end{aligned}
$$

$=$ ⟦ elementary algebra ⟧

$$
\begin{aligned}
&\frac{[0\le i<n]\cdot[0\le i+1<n]}{(n-i)!} \cdot \sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \\
&\quad \mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array},i,j)] \left(Z_\pi^{i+1} \right) [j/i+k]
\end{aligned}
$$

$\preceq$ ⟦ elementary algebra $(0\le i<n \wedge 0\le i+1<n \Rightarrow 0\le i<n)$ ⟧

$$
\begin{aligned}
&\frac{[0\le i<n]}{(n-i)!} \cdot \sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \\
&\quad \mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array},i,j)] \left(Z_\pi^{i+1} \right) [j/i+k]
\end{aligned}
$$

$=$ ⟦ Lemma F.5; Lemma F.6 ⟧

$$
\frac{[0\le i<n]}{(n-i)!} \cdot \sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} Z_{\pi_k}^{i}
$$

$=$ ⟦ Lemma F.9 ⟧

$$
\frac{[0\le i<n]}{(n-i)!} \cdot \sum_{\pi\in\mathsf{Perm}(i,n-1)} Z_\pi^{i}
$$

$$
\begin{aligned}
&\preceq \quad [\![\text{ elementary algebra } (0 \preceq Z \text{ for all } Z \in \mathbb{E})]\!] \\
&\quad \frac{[0 \le i < n]}{(n-i)!} \cdot \sum_{\pi \in \mathsf{Perm}(i,n-1)} Z_\pi^i \\
&\qquad + [\neg(0 \le i < n)] \cdot \bigstar_{m=0}^{n-1} [\mathsf{array} + m \mapsto a_m] \\
&= [\![\text{ Definition of I }]\!] \\
&\quad \mathrm{I}. \qquad \square
\end{aligned}
$$

Lemma F.5 $\mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)]\,\left(Z_\pi^{i+1}\right)\,[j/i] \;=\; Z_{\pi_0}^{i+1}$.

Proof. First, consider the following:

$$
\begin{aligned}
&Z_\pi^{i+1} \\
&= \left[\!\!\left[\text{ Definition of } Z_\pi^{i+1} \right]\!\!\right] \\
&\quad \bigstar_{m=0}^{i} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \\
&= \left[\!\!\left[\text{ Definition of } \bigstar \text{; D.1.3; D.1.1 } \right]\!\!\right] \\
&\quad [\mathsf{array} + i \mapsto a_i] \star \underbrace{\left(\bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \right)}_{\triangleq Z} \\
&= [\![\text{ Definition of } \pi_0]\!] \\
&\quad \left[\mathsf{array} + i \mapsto a_{\pi_0(i)}\right] \star \left(\bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi_0(m)}\right] \right) \\
&= [\![\text{ D.1.3; D.1.1 }]\!] \\
&\quad \bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \left(\left[\mathsf{array} + i \mapsto a_{\pi_0(i)}\right] \star \bigstar_{m=i+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi_0(m)}\right] \right)
\end{aligned}
$$

$$
\begin{aligned}
&= \left[\!\left[\text{Definition of } \bigstar \right]\!\right] \\
&\quad \bigstar_{m=0}^{i-1} \left[\mathsf{array} + m \mapsto a_m\right] \star \bigstar_{m=i}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi_0(m)}\right] \\
&= \left[\!\left[\text{Definition of } \mathsf{Z}^i_{\pi_0} \right]\!\right] \\
&\quad \mathsf{Z}^i_{\pi_0}.
\end{aligned}
$$

With these calculations at hand, we proceed as follows:

$$
\begin{aligned}
&\mathsf{wp}\left[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)\right] \left(\mathsf{Z}^{i+1}_{\pi}\right) [j/i] \\
&= \left[\!\left[\text{Calculations from above} \right]\!\right] \\
&\quad \mathsf{wp}\left[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)\right] (\mathsf{Z}) [j/i] \\
&= \left[\!\left[\text{Lemma F.7} \right]\!\right] \\
&\quad \mathsf{Z} \\
&= \left[\!\left[\text{Calculations from above} \right]\!\right] \\
&\quad \mathsf{Z}^i_{\pi_0}.
\end{aligned}
$$

□

Lemma F.6 For $0 < k \leq n - 1 - i$, we have:

$$
\mathsf{wp}\left[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)\right] \left(\mathsf{Z}^{i+1}_{\pi}\right) [j/i+k] \;=\; \mathsf{Z}^i_{\pi_k}.
$$

Proof. First, consider the following:

$$
\begin{aligned}
&\mathsf{Z}^{i+1}_{\pi} \\
&= \left[\!\left[\text{Definition of } \mathsf{Z}^{i+1}_{\pi} \right]\!\right] \\
&\quad \bigstar_{m=0}^{i} \left[\mathsf{array} + m \mapsto a_m\right] \star \bigstar_{m=i+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \\
&= \left[\!\left[\text{Definition of } \bigstar\text{; elementary algebra} \right]\!\right] \\
&\quad \bigstar_{m=0}^{i} \left[\mathsf{array} + m \mapsto a_m\right] \star \left(\bigstar_{m=i+1}^{i+k-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \right) \star \\
&\qquad \left[\mathsf{array} + i + k \mapsto a_{\pi(i+k)}\right] \star \left(\bigstar_{m=i+k+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \right)
\end{aligned}
$$

$$= \quad [\![\text{ Definition of } \bigstar\text{; D.1.3 }]\!]$$

$$[\mathsf{array} + i \mapsto a_i] \star \left[\mathsf{array} + i + k \mapsto a_{\pi(i+k)}\right] \star \bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i+1}^{i+k-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \star \bigstar_{m=i+k+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right]$$

Let us denote by Z the last two lines of the above expectation. We then proceed as follows:

$$\begin{aligned}
&\mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)]\left(Z_\pi^{i+1}\right)[j/i+k] \\
&= [\![\text{ Calculations from above }]\!] \\
&\quad \mathsf{wp}\,[\mathsf{void} := \mathsf{swap}(\mathsf{array}, i, j)]\,(\\
&\qquad [\mathsf{array} + i \mapsto a_i] \star \left[\mathsf{array} + i + k \mapsto a_{\pi(i+k)}\right] \star Z \\
&\quad)\,[j/i+k] \\
&= [\![\text{ Lemma F.8 }]\!] \\
&\quad \left[\mathsf{array} + i \mapsto a_{\pi(i+k)}\right] \star [\mathsf{array} + i + k \mapsto a_i] \star Z \\
&= [\![\text{ Definition of Z }]\!] \\
&\quad \left[\mathsf{array} + i \mapsto a_{\pi(i+k)}\right] \star [\mathsf{array} + i + k \mapsto a_i] \star \\
&\qquad \bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m] \star \bigstar_{m=i+1}^{i+k-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \star \\
&\qquad \bigstar_{m=i+k+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi(m)}\right] \\
&= [\![\text{ D.1.3; Definition of } \pi_k \text{ }]\!] \\
&\quad \left(\bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m]\right) \star \\
&\qquad \left[\mathsf{array} + i \mapsto a_{\pi_k(i)}\right] \star \left(\bigstar_{m=i+1}^{i+k-1} \left[\mathsf{array} + m \mapsto a_{\pi_k(m)}\right]\right) \star \\
&\qquad \left[\mathsf{array} + i + k \mapsto a_{\pi_k(i+k)}\right] \star \left(\bigstar_{m=i+k+1}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi_k(m)}\right]\right)
\end{aligned}$$

$$
\begin{aligned}
&= \left[\!\!\left[\text{Definition of } \bigstar \right]\!\!\right] \\
&\quad \left(\bigstar_{m=0}^{i-1} [\mathsf{array} + m \mapsto a_m]\right) \star \left(\bigstar_{m=i}^{n-1} \left[\mathsf{array} + m \mapsto a_{\pi_k(m)}\right]\right) \\
&= \left[\!\!\left[\text{Definition of } Z^i_{\pi_k} \right]\!\!\right] \\
&\quad Z^i_{\pi_k}.
\end{aligned}
$$

□

Lemma F.7 For every precise expectation Z with $j \notin \mathbf{Vars}\,(\mathrm{Z})$:

$$
\begin{aligned}
&\mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,([\mathsf{array} + i \mapsto w] \star \mathrm{Z})\,[j/i] \\
\preceq\ &[\mathsf{array} + i \mapsto w] \star \mathrm{Z}.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,([\mathsf{array} + i \mapsto w] \star \mathrm{Z})\,[j/i] \\
&= [\![\text{ Figure F.2 (page 437); apply substitution } (j \notin \mathbf{Vars}\,(\mathrm{Z}))]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\qquad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad\qquad\quad [\mathsf{array} + i \mapsto w] \star \mathrm{Z}\big)\big)\big)\big) \\
&= [\![\text{ D.1.3; D.7.4 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\qquad [u = w] \cdot [\mathsf{array} + i \mapsto -] \star \mathrm{Z}\big)\big)\big) \\
&= [\![\text{ D.7.3; D.1.26; D.7.8 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star}
\end{aligned}
$$

$$
\begin{aligned}
&\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad [\mathsf{array} + i \mapsto v] \star ([u = w] \cdot \mathrm{Z}) \big) \big) \\
&= [\![\text{ D.7.1 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\quad [\mathsf{array} + i \mapsto -] \star ([u = w] \cdot \mathrm{Z}) \big) \big) \\
&= [\![\text{ D.1.26; D.7.8 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\quad [\mathsf{array} + i \mapsto v] \star ([u = w] \cdot \mathrm{Z}) \big) \big) \\
&= [\![\text{ D.7.1 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i \mapsto v] \star ([u = w] \cdot \mathrm{Z}) \big) \\
&= [\![\text{ D.1.26; D.7.8; introduce fresh variable } y \text{ for } v \text{ ; D.6.11 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v \in \mathbb{Z}} ([u = v] \cdot [\mathsf{array} + i \mapsto y] \star ([u = w] \cdot \mathrm{Z}))\,[y/v] \big) \\
&= [\![\text{ D.6.11; apply substitution }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad [\mathsf{array} + i \mapsto u] \star ([u = w] \cdot \mathrm{Z}) \big) \\
&= [\![\text{ D.7.1 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star ([u = w] \cdot \mathrm{Z}) \\
&= [\![\text{ D.7.3 }]\!] \\
&\quad \sup_{u \in \mathbb{Z}} [u = w] \cdot [\mathsf{array} + i \mapsto u] \star \mathrm{Z} \\
&= [\![\text{ introduce fresh variable } y \text{ for } u \text{ }]\!]
\end{aligned}
$$

$$\begin{aligned}
&\sup_{u\in\mathbb{Z}} \left([u = w] \cdot [\mathsf{array} + i \mapsto y] \star \mathrm{Z}\right)[y/u] \\
=\ & [\![\ \text{D.6.11; apply substitution}\ ]\!] \\
& [\mathsf{array} + i \mapsto w] \star \mathrm{Z}. \qquad \square
\end{aligned}$$

Lemma F.8 For every precise expectation Z with $j \notin \mathbf{Vars}\,(\mathrm{Z})$ and $k > 0$:

$$\begin{aligned}
&\mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,\big(\\
&\qquad [\mathsf{array} + i \mapsto u'] \star [\mathsf{array} + j \mapsto v'] \star \mathrm{Z}\big)\,[j/i + k] \\
=\ & [\mathsf{array} + i \mapsto v'] \star [\mathsf{array} + i + k \mapsto u'] \star \mathrm{Z}.
\end{aligned}$$

Proof.

$$\begin{aligned}
&\mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,\big(\\
&\qquad [\mathsf{array} + i \mapsto u'] \star [\mathsf{array} + j \mapsto v'] \star \mathrm{Z}\big)\,[j/i + k] \\
=\ & [\![\ \text{Figure F.2 (page 437); apply substitution } (\mathsf{void}, j \notin \mathbf{Vars}\,(\mathrm{Z}))\ ]\!] \\
& \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
& \quad \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
& \quad\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
& \quad\quad\quad [\mathsf{array} + i + k \mapsto -] \star \big([\mathsf{array} + i + k \mapsto u] \mathrel{-\!\!\star} \\
& \quad\quad\quad\quad [\mathsf{array} + i \mapsto u'] \star [\mathsf{array} + i + k \mapsto v'] \star \mathrm{Z}\big)\big)\big)\big) \\
=\ & [\![\ \text{D.1.3; D.7.4}\ ]\!] \\
& \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
& \quad \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
& \quad\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
& \quad\quad\quad [u = v'] \cdot [\mathsf{array} + i + k \mapsto -] \star [\mathsf{array} + i \mapsto u'] \star \mathrm{Z}\big)\big)\big) \\
=\ & [\![\ \text{D.7.3; D.1.3; D.7.4}\ ]\!] \\
& \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
& \quad \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
& \quad\quad [v = u'] \cdot [\mathsf{array} + i \mapsto -] \star [\mathsf{array} + i + k \mapsto -] \star \big([u = v'] \cdot \mathrm{Z}\big)\big)\big)
\end{aligned}$$

$$
\begin{aligned}
&= \llbracket \text{ D.7.3; D.1.3; D.1.26; D.7.8 } \rrbracket \\
&\quad \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
&\qquad\quad [\mathsf{array} + i + k \mapsto v] \star [\mathsf{array} + i \mapsto -] \star \big(\big[v = u'\big] \cdot \big[u = v'\big] \cdot \mathrm{Z} \big) \big) \big) \\
&= \llbracket \text{ D.7.1 } \rrbracket \\
&\quad \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star [\mathsf{array} + i \mapsto -] \star \big(\big[v = u'\big] \cdot \big[u = v'\big] \cdot \mathrm{Z} \big) \big) \\
&= \llbracket \text{ D.1.11; D.1.3; D.1.26; D.7.8 } \rrbracket \\
&\quad \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&\qquad [\mathsf{array} + i \mapsto u] \star \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big(\big[v = u'\big] \cdot \big[u = v'\big] \cdot \mathrm{Z} \big) \big) \\
&= \llbracket \text{ D.7.1 } \rrbracket \\
&\quad \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \sup_{v\in\mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big(\big[v = u'\big] \cdot \big[u = v'\big] \cdot \mathrm{Z} \big) \\
&= \llbracket \text{ D.7.3; D.6.11; elementary algebra } \rrbracket \\
&\quad \sup_{u\in\mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big[\mathsf{array} + i + k \mapsto u'\big] \star \big(\big[u = v'\big] \cdot \mathrm{Z} \big) \\
&= \llbracket \text{ D.7.3; D.6.11; elementary algebra } \rrbracket \\
&\quad \big[\mathsf{array} + i \mapsto v'\big] \star \big[\mathsf{array} + i + k \mapsto u'\big] \star \mathrm{Z}. \qquad \square
\end{aligned}
$$

Lemma F.9

$$
\sum_{\pi\in\mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \mathrm{Z}^i_{\pi_k} \preceq \sum_{\pi\in\mathsf{Perm}(i,n-1)} \mathrm{Z}^i_{\pi}.
$$

Proof. We first notice that $\sum_{\pi\in\emptyset} \ldots = 0$. Furthermore, we have $1 = [i < n-1] + [i \geq n-1]$. We consider both cases separately:

1. If $[i < n-1]$ holds, we have

$$
\begin{aligned}
&\sum_{\pi\in\mathsf{Perm}(i,n-1)} \mathrm{Z}^i_{\pi} \\
&= \llbracket \text{ Lemma F.2 } \rrbracket
\end{aligned}
$$

$$\begin{aligned}
&\sum_{\pi \in \bigcup_{\pi' \in \mathsf{Perm}(i,n-1)} \{\, \pi'_k \mid 0 \le k \le n-1-i \,\}} \mathrm{Z}^i_\pi \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad \sum_{\pi \in \mathsf{Perm}(i,n-1)} \sum_{k=0}^{n-1-i} \mathrm{Z}^i_{\pi_k}.
\end{aligned}$$

2. If $[i \ge n-1]$ holds, we have $\mathsf{Perm}(i+1, n-1) = \emptyset$. Hence,

$$\begin{aligned}
&\sum_{\pi \in \mathsf{Perm}(i+1,n-1)} \sum_{k=0}^{n-1-i} \mathrm{Z}^i_{\pi_k} \\
&= \llbracket \text{ By assumption } \rrbracket \\
&\quad \sum_{\pi \in \emptyset} \sum_{k=0}^{n-1-i} \mathrm{Z}^i_{\pi_k} \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad 0 \\
&\preceq \llbracket \text{ 0 is least element of } \mathbb{E} \rrbracket \\
&\quad \sum_{\pi \in \mathsf{Perm}(i,n-1)} \mathrm{Z}^i_\pi.
\end{aligned}$$

□

F.4 Omitted Calculations in Section 8.5

All involved quantitative entailments are proven in separate lemmas below. Furthermore, the missing cases to verify our invariant are provided at the end of this section: Figures F.3 and F.4 contain the two missing cases considered in the `else` branch when verifying case (1) of our proposed invariant. Case (2) of our proposed invariant is verified in Figures F.5 and F.6. Finally, case (3) of our invariant is considered in Figure F.7.

Lemma F.10 For every precise expectation Y', we have

$$\begin{aligned}
&[\boxdot\mathsf{top} \hookrightarrow \mathsf{l}, \mathsf{r}] \cdot \mathrm{Y}' \star ([\textit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\textit{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \textbf{size})) \\
\preceq\ & \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \\
&+ \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star ([\textit{tree}\,(\mathsf{l})] \cdot 1 \star ([\textit{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \textbf{size})) \star [\textit{tree}\,(\mathsf{r})] \\
&+ [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + [\textit{tree}\,(\mathsf{l})] \star [\mathsf{out} \hookrightarrow 0, 0] \cdot \infty.
\end{aligned}$$

Proof.

$$
\begin{aligned}
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\textit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\textit{path}\,(\boxdot\mathsf{top},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size}) \big) \\
= \;& [\![\text{ Def. of } [\textit{tree}\,(\boxdot\mathsf{top})]; \text{ D.6.2; D.7.3; D.6.6; D.7.5; elem. algebra }]\!] \\
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot 1 \star \\
& \quad ([\textit{path}\,(\boxdot\mathsf{top},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size}) \big) \\
= \;& [\![\text{ Def. of } [\textit{path}\,(\boxdot\mathsf{top},\mathsf{out})]; \text{ D.6.2; D.7.3; D.6.6; D.7.5; elem. algebra }]\!] \\
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot 1 \\
& \quad \star \big(([\boxdot\mathsf{top} = \mathsf{out}] \cdot [\boxdot\mathsf{top} \mapsto 0,0] \\
& \qquad + [\boxdot\mathsf{top} \neq \mathsf{out}] \cdot [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \\
& \qquad\quad \star \max\{[\textit{path}\,(\mathsf{l},\mathsf{out})], [\textit{path}\,(\mathsf{r},\mathsf{out})]\}) \cdot 1/2 \cdot \mathbf{size} \big) \big) \\
= \;& [\![\text{ elementary algebra }]\!] \\
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot \\
& \quad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} = \mathsf{out}] \cdot [\boxdot\mathsf{top} \mapsto 0,0] \\
& \qquad + 1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} \neq \mathsf{out}] \cdot \\
& \qquad\quad [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star \max\{ [\textit{path}\,(\mathsf{l},\mathsf{out})], [\textit{path}\,(\mathsf{r},\mathsf{out})] \}) \big) \\
= \;& [\![\text{ elementary algebra; D.3.2 }]\!] \\
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot \\
& \quad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} = \mathsf{out}] \cdot [\boxdot\mathsf{top} \mapsto 0,0]) \big) \\
& + [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot \\
& \quad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} \neq \mathsf{out}] \cdot \\
& \qquad [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star \max\{[\textit{path}\,(\mathsf{l},\mathsf{out})], [\textit{path}\,(\mathsf{r},\mathsf{out})]\}) \big) \\
\preceq \;& [\![\text{ D.7.3; D.7.5; D.6.1; D.6.5; elementary algebra }]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty \\
& + \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot \\
& \quad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} \neq \mathsf{out}] \cdot \\
& \qquad [\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star \max\{[\textit{path}\,(\mathsf{l},\mathsf{out})], [\textit{path}\,(\mathsf{r},\mathsf{out})]\}) \big) \\
= \;& [\![\text{ elementary algebra (max distributes over } +, \cdot); \text{ D.1.7 }]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + \max\{ \\
& \quad \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\textit{tree}\,(\mathsf{l})] \star [\textit{tree}\,(\mathsf{r})] \cdot
\end{aligned}
$$

$$
\begin{aligned}
&\qquad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{path}\,(\mathsf{l}, \boxdot\mathsf{top})])\big), \\
&\quad \mathrm{Y}' \star \big([\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] \cdot \\
&\qquad 1 \star (1/2 \cdot \mathbf{size} \cdot [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{path}\,(\mathsf{r}, \mathsf{out})])\big)\big\} \\
\preceq\ & [\![\text{ Lemma F.11 (twice) }]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + \max\big\{ \\
&\quad \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\qquad + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad\quad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})], \\
&\quad \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\qquad + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad\quad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{r}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})]\big\} \\
=\ & [\![\text{ elementary algebra } (\max\{A + B, A + C\} = A + \max\{B, C\}) \,]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] + \max\big\{ \\
&\quad \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})], \\
&\quad \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{r}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})]\ \big\} \\
\preceq\ & [\![\max\{A, B\} \le A + B \text{ for } A, B \ge 0 \,]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\quad + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\quad + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{r}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})]\ \big\} \\
\preceq\ & [\![\text{ Lemma F.13 }]\!] \\
& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\boldsymbol{tree}\,(\mathsf{l})] \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\quad + \mathrm{Y}' \star [\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad ([\boldsymbol{tree}\,(\mathsf{l})] \cdot 1 \star ([\boldsymbol{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})) \star [\boldsymbol{tree}\,(\mathsf{r})] \\
&\quad + \underbrace{\mathrm{Y}'}_{\preceq \infty} \star \underbrace{[\boxdot\mathsf{top} \mapsto \mathsf{l}, \mathsf{r}]}_{\preceq 1} \star
\end{aligned}
$$

$$
\begin{aligned}
&\big([\mathit{tree}\,(\mathsf{l})]\cdot \underbrace{1\star\big([\mathsf{out}\hookrightarrow 0,0]\cdot 1/2\cdot \mathbf{size}\big)}_{\preceq[\mathsf{out}\hookrightarrow 0,0]\cdot\infty}\big)\star \underbrace{[\mathit{tree}\,(\mathsf{r})]}_{\preceq 1}\\
\preceq\ & [\![\ \text{elementary algebra; } 1\star[x\hookrightarrow 0,0]=[x\hookrightarrow 0,0]\ ]\!]\\
& [\mathsf{l}=0]\cdot[\mathsf{r}=0]\cdot\infty+\mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\\
&\quad+\mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star\\
&\qquad([\mathit{tree}\,(\mathsf{l})]\cdot 1\star([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot\mathbf{size}))\star[\mathit{tree}\,(\mathsf{r})]\\
&\quad+[\mathit{tree}\,(\mathsf{l})]\star[\mathsf{out}\hookrightarrow 0,0]\cdot\infty\\
=\ & [\![\ \text{algebra (rearranging terms)}\ ]\!]\\
& \mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\\
&+\mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star([\mathit{tree}\,(\mathsf{l})]\cdot 1\star([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot\mathbf{size}))\star[\mathit{tree}\,(\mathsf{r})]\\
&+[\mathsf{l}=0]\cdot[\mathsf{r}=0]\cdot\infty+[\mathit{tree}\,(\mathsf{l})]\star[\mathsf{out}\hookrightarrow 0,0]\cdot\infty.
\end{aligned}
$$

□

Lemma F.11 For every precise expectation Y', we have

$$
\begin{aligned}
&\mathrm{Y}'\star\big([\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\cdot\\
&\qquad 1\star(1/2\cdot\mathbf{size}\cdot[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{path}\,(\mathsf{l},\boxdot\mathsf{top})])\big)\\
\preceq\ & \mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]+\mathrm{Y}'\star[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star\\
&\qquad([\mathit{tree}\,(\mathsf{l})]\cdot 1\star([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot\mathbf{size}))\star[\mathit{tree}\,(\mathsf{r})]\,.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\mathrm{Y}'\star\big([\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\cdot\\
&\qquad 1\star(1/2\cdot\mathbf{size}\cdot[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{path}\,(\mathsf{l},\boxdot\mathsf{top})])\big)\\
=\ & [\![\ \text{D.3.5}\ ]\!]\\
&\mathrm{Y}'\star\big([\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\cdot\\
&\quad 1\star\big((1/2\cdot\mathbf{size}\cdot[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}])\star[\mathit{path}\,(\mathsf{l},\mathsf{out})]\\
&\qquad+[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star(1/2\cdot\mathbf{size}\cdot[\mathit{path}\,(\mathsf{l},\mathsf{r})])\big)\big)\\
=\ & [\![\ \text{D.6.6; elementary algebra}\ ]\!]\\
&\mathrm{Y}'\star\big([\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{tree}\,(\mathsf{l})]\star[\mathit{tree}\,(\mathsf{r})]\cdot\\
&\quad 1\star\big([\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star[\mathit{path}\,(\mathsf{l},\mathsf{out})]\\
&\qquad+[\boxdot\mathsf{top}\mapsto\mathsf{l},\mathsf{r}]\star(1/2\cdot\mathbf{size}\cdot[\mathit{path}\,(\mathsf{l},\mathsf{r})])\big)\big)\\
\preceq\ & [\![\ \text{D.1.9}\ ]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Y}' \star \big([\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot \big(\\
&\qquad 1 \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{path}\,(\mathsf{l}, \mathsf{out})] \\
&\qquad\quad + 1 \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \big) \\
\preceq\ & [\![\text{ D.6.1 } (1 \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \preceq 1;\ [\mathit{path}\,(\mathsf{l}, \mathsf{out})] \preceq 1) \,]\!] \\
&\mathrm{Y}' \star \big([\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot \big(\\
&\qquad 1 + 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \big) \\
&\mathrm{Y}' \star \big([\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot \big(\\
&\qquad 1 + 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \big) \\
=\ & [\![\text{ elementary algebra (distributivity of } \cdot \text{ and } +); \text{ D.3.2 }]\!] \\
&\mathrm{Y}' \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \\
&+ \mathrm{Y}' \star \big([\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \\
=\ & [\![\text{ Lemma F.12 }]\!] \\
&\mathrm{Y}' \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] + \mathrm{Y}' \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \\
&\qquad \big([\mathit{tree}\,(\mathsf{l})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \star [\mathit{tree}\,(\mathsf{r})] \,. \qquad \square
\end{aligned}
$$

Lemma F.12 For every precise expectation Y', we have

$$
\begin{aligned}
&\mathrm{Y}' \star \big([\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \\
=\ & \mathrm{Y}' \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star \big([\mathit{tree}\,(\mathsf{l})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\mathsf{l}, \mathsf{r})]) \big) \star [\mathit{tree}\,(\mathsf{r})] \,.
\end{aligned}
$$

Proof. Let $(\mathfrak{s}, \mathfrak{h})$ be a stack-heap pair such that

$$(\mathrm{Y}' \star [\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})])(\mathfrak{s}, \mathfrak{h}) = 1.$$

For every other stack-heap pair, the first expectation evaluations to 0 and the claim is trivially satisfied. Now, since Y', $[\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}]$, $[\mathit{tree}\,(\mathsf{l})]$, and $[\mathit{tree}\,(\mathsf{r})]$ are precise expectations, there exists exactly one partition $\mathfrak{h} = \mathfrak{h}_1 \star \mathfrak{h}_2 \star \mathfrak{h}_3 \star \mathfrak{h}_4$ with $\mathrm{Y}'(\mathfrak{s}, \mathfrak{h}_1) = 1$, $[\boxdot \mathsf{top} \mapsto \mathsf{l}, \mathsf{r}]\,(\mathfrak{s}, \mathfrak{h}_2) = 1$, $[\mathit{tree}\,(\mathsf{l})]\,(\mathfrak{s}, \mathfrak{h}_3) = 1$, $[\mathit{tree}\,(\mathsf{r})]\,(\mathfrak{s}, \mathfrak{h}_4) = 1$. It then remains to show that

$$
\begin{aligned}
&(1 \star ([\mathit{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size}))(\mathfrak{s}, \mathfrak{h}_2 \star \mathfrak{h}_3 \star \mathfrak{h}_4) \\
=\ & [\![\text{ Definition of } \star \text{ for } 1 \star \ldots \,]\!] \\
&\max\{([\mathit{path}\,(\mathsf{l}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size})(\mathfrak{s}, \mathfrak{h}') \mid \mathfrak{h}' \subseteq \mathfrak{h}_2 \star \mathfrak{h}_3 \star \mathfrak{h}_4\} \\
=\ & [\![\text{ shown further below }]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\max\{([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot \mathbf{size})(\mathfrak{s},\mathfrak{h}') \mid \mathfrak{h}'\subseteq \mathfrak{h}_3\}\\
=\;& [\![\text{ Definition of } \star \text{ for } 1\star\ldots\,]\!]\\
&(1\star[\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot \mathbf{size})(\mathfrak{s},\mathfrak{h}_3).
\end{aligned}
$$

To complete the proof, we now show that for every $\mathfrak{h}'\subseteq \mathfrak{h}_2\star\mathfrak{h}_3\star\mathfrak{h}_4$

$$
([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot \mathbf{size})(\mathfrak{s},\mathfrak{h}')>0 \quad \text{implies} \quad \mathfrak{h}'\subseteq\mathfrak{h}_3. \tag{F.1}
$$

First recall that

$$
[\mathit{tree}\,(\mathsf{l})] \;\triangleq\; [\mathsf{l}=0]\cdot\mathbf{emp} + \sup_{u,v\in\mathbb{Z}} [\mathsf{l}\mapsto u,v]\star[\mathit{tree}\,(u)]\star[\mathit{tree}\,(v)]\,.
$$

By D.5.3, $[\mathsf{l}\mapsto u,v\in\mathbb{Z}]$ implies $[\mathsf{l}\neq 0]$.

We proceed by induction on the number of unfoldings of $[\mathit{path}\,(\mathsf{l},\mathsf{out})]$.

If $[\mathit{path}\,(\mathsf{l},\mathsf{out})]$ is unrolled once, we have

$$
\begin{aligned}
&([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot\mathbf{size})(\mathfrak{s},\mathfrak{h}')>0\\
\text{iff}\quad &[\![\text{ Unroll } [\mathit{path}\,(\mathsf{l},\mathsf{out})] \text{ exactly once }]\!]\\
&([\mathsf{l}=\mathsf{out}]\cdot[\mathsf{l}\mapsto 0,0]\cdot\underbrace{1/2\cdot\mathbf{size}}_{=1})(\mathfrak{s},\mathfrak{h}')>0\\
\text{iff}\quad &[\![\text{ Definition of involved expectations }]\!]\\
&\mathfrak{s}(\mathsf{l})=\mathfrak{s}(\mathsf{out}) \text{ and } \mathfrak{h}'=\{\,\mathfrak{s}(\mathsf{l})::0,0\,\}\,.
\end{aligned}
$$

The last equation from above implies $\mathfrak{s}(\mathsf{l})\neq 0$. Then, since $[\mathit{tree}\,(\mathsf{l})]\,(\mathfrak{s},\mathfrak{h}_3)=1$ holds by assumption, we have $\mathfrak{h}'\subseteq\mathfrak{h}_3$.

If $[\mathit{path}\,(\mathsf{l},\mathsf{out})]$ is unrolled more than once, we have

$$
\begin{aligned}
&([\mathit{path}\,(\mathsf{l},\mathsf{out})]\cdot 1/2\cdot\mathbf{size})(\mathfrak{s},\mathfrak{h}')>0\\
\text{iff}\quad &[\![\text{ Unroll } [\mathit{path}\,(\mathsf{l},\mathsf{out})] \text{ at least twice }]\!]\\
&([\mathsf{l}\neq\mathsf{out}]\cdot \sup_{u,v\in\mathbb{Z}} [\mathsf{l}\mapsto u,v]\star\\
&\qquad \max\{[\mathit{path}\,(u,\mathsf{out})]\,,[\mathit{path}\,(v,\mathsf{out})]\}\cdot 1/2\cdot\mathbf{size})(\mathfrak{s},\mathfrak{h}')>0.
\end{aligned}
$$

Now, by definition of $\star$, we have $\mathfrak{h}'=\{\,\mathfrak{s}(\mathsf{l})::u,v\,\}\star\mathfrak{h}''$. Furthermore, since $[\mathit{tree}\,(\mathsf{l})]\,(\mathfrak{s},\mathfrak{h}_3)=1$ holds by assumption, we have $\{\,\mathfrak{s}(\mathsf{l})::u,v\,\}\subseteq\mathfrak{h}_3$. Moreover, by I.H., we have $\mathfrak{h}''\subseteq\mathfrak{h}_3$. Hence $\mathfrak{h}'\subseteq\mathfrak{h}_3$. □

Lemma F.13 $[\mathit{path}\,(\circledcirc\mathsf{top},\mathsf{l})] \;\preceq\; [\mathsf{l}\hookrightarrow 0,0]$.

Proof. By induction on the number of unrollings of $[\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{l})]$.
For exactly one unrolling, we have

$$
\begin{aligned}
& [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{l})] \\
= \;& [\![\text{ Definition of } [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{l})] \text{ for exactly one unrolling }]\!] \\
& [\boxdot\mathsf{top} = \mathsf{l}] \cdot [\boxdot\mathsf{top} \mapsto 0, 0] \\
\preceq \;& [\![\text{ elementary algebra; D.2.2 }]\!] \\
& [\boxdot\mathsf{top} = \mathsf{l}] \star [\mathsf{l} \mapsto 0, 0] \\
\preceq \;& [\![\text{ D.6.1; D.6.7 }]\!] \\
& [\mathsf{l} \hookrightarrow 0, 0].
\end{aligned}
$$

For more than one unrolling, we have

$$
\begin{aligned}
& [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{l})] \\
= \;& [\![\text{ Definition of } [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{l})] \text{ for more than one unrolling }]\!] \\
& [\boxdot\mathsf{top} \neq \mathsf{l}] \cdot \sup_{u,v \in \mathbb{Z}} [\boxdot\mathsf{top} \mapsto u, v] \star \\
& \qquad \max\{[\boldsymbol{path}\,(u, \mathsf{l})]\,, [\boldsymbol{path}\,(v, \mathsf{l})]\} \\
\preceq \;& [\![\text{ I.H. }]\!] \\
& \underbrace{[\boxdot\mathsf{top} \neq \mathsf{l}]}_{\preceq 1} \cdot \sup_{u,v \in \mathbb{Z}} \underbrace{[\boxdot\mathsf{top} \mapsto u, v]}_{\preceq 1} \star \max\{[\mathsf{l} \hookrightarrow 0, 0], [\mathsf{l} \hookrightarrow 0, 0]\} \\
\preceq \;& [\![\text{ D.6.1; elementary algebra }]\!] \\
& 1 \star [\mathsf{l} \hookrightarrow 0, 0] \\
= \;& [\![\text{ D.6.7; elementary algebra }]\!] \\
& [\mathsf{l} \hookrightarrow 0, 0]\,.
\end{aligned}
$$

□

Lemma F.14 For every precise expectation Y', we have

$$
\begin{aligned}
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l}, \mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \mathrm{Y}' \star \\
& \qquad \left([\boldsymbol{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{out})]) \right) \\
\preceq \;& [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& [\boxdot\mathsf{top} \hookrightarrow \mathsf{l}, \mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \mathrm{Y}' \star \\
& \qquad \left([\boldsymbol{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star (1/2 \cdot \mathbf{size} \cdot [\boldsymbol{path}\,(\boxdot\mathsf{top}, \mathsf{out})]) \right)
\end{aligned}
$$

$$
\begin{aligned}
&= \llbracket \text{ Def. of } [\mathit{tree}\,(\boxdot\mathsf{top})]\text{; D.6.2; D.7.3; D.6.6; D.7.5; elem. algebra } \rrbracket \\
&\quad [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \mathrm{Y}' \star \\
&\qquad \big([\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}] \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{r})] \cdot \\
&\qquad\quad 1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\boxdot\mathsf{top}, \mathsf{out})]) \big) \\
&= \llbracket \text{ elementary algebra } \rrbracket \\
&\quad \underbrace{[\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}]}_{\preceq 1} \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \underbrace{\mathrm{Y}'}_{\preceq \infty} \star \\
&\qquad \big(\underbrace{[\boxdot\mathsf{top} \mapsto \mathsf{l},\mathsf{r}]}_{\preceq 1} \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{l})] \big) \cdot \\
&\qquad\quad \underbrace{1 \star (1/2 \cdot \mathbf{size} \cdot [\mathit{path}\,(\boxdot\mathsf{top}, \mathsf{out})])}_{\preceq \infty} \big) \\
&\preceq \llbracket \text{ D.6.1; D.6.5; D.1.3 } \rrbracket \\
&\quad [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \infty \star [\mathit{tree}\,(\mathsf{l})] \star [\mathit{tree}\,(\mathsf{l})] \\
&= \llbracket \text{ Definition of } [\mathit{tree}\,(v)] \text{ (twice) } \rrbracket \\
&\quad [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot \infty \\
&\quad \star \big([\mathsf{l} = 0] \cdot \mathbf{emp} + \underbrace{\sup_{u,v \in \mathbb{Z}} [\mathsf{l} \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)]}_{= \mathrm{Z}'} \big) \\
&\quad \star \big([\mathsf{l} = 0] \cdot \mathbf{emp} + \sup_{u,v \in \mathbb{Z}} [\mathsf{l} \mapsto u, v] \star [\mathit{tree}\,(u)] \star [\mathit{tree}\,(v)] \big) \\
&= \llbracket \text{ D.3.2; } \mathrm{Z}' \star \mathrm{Z}' = 0\text{; } \mathrm{Z}' \star ([\mathsf{l} = 0] \cdot \mathbf{emp}) = 0\text{; elementary algebra } \rrbracket \\
&\quad [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{l} = \mathsf{r}] \cdot \infty \star ([\mathsf{l} = 0] \cdot \mathbf{emp}) \\
&= \llbracket \text{ D.2.2; D.1.2; D.1.3 } \rrbracket \\
&\quad [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \cdot [\mathsf{l} = 0] \cdot \infty \\
&\preceq \llbracket \text{ elementary algebra } \rrbracket \\
&\quad [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty. \qquad \square
\end{aligned}
$$

Lemma F.15 Let X_{if} be defined as in Abbildung 8.11. Then

$$
\begin{aligned}
&\quad [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \mathrm{X}_{\mathsf{if}}\,[\mathsf{r}/v]\,[\mathsf{l}/u]\,[\mathsf{root}/\boxdot\mathsf{top}]\,[+\boxdot] \\
&\preceq\ [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \\
&\qquad \mathrm{Y} \star ([\mathsf{top} \neq 0] \cdot [\mathit{tree}\,(\mathsf{top})] \cdot \log\,(1 + 1/2 \cdot \mathbf{size}))\,.
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \mathsf{X}_{\mathsf{if}}\, [\mathsf{r}/v]\, [\mathsf{l}/u]\, [\mathsf{root}/\boxdot\mathsf{top}]\, [+\boxdot] \\
= \;& [\![\text{ Definition of } \mathsf{X}_{\mathsf{if}} \text{ as in Abbildung 8.11 }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \\
& \quad \underbrace{\mathsf{Y}\, [-\boxdot]\, [\mathsf{out}/\mathsf{root}]\, [\mathsf{r}/v]\, [\mathsf{l}/u]\, [\mathsf{root}/\boxdot\mathsf{top}]\, [+\boxdot]}_{= \mathsf{Y}} \\
& \quad \star\, ([\textit{tree}\,(\mathsf{top})] \cdot 1 \star (\underbrace{[\textit{path}\,(\mathsf{top}, \mathsf{top})]}_{= [\mathsf{top} \mapsto 0,0]} \cdot 1/2 \cdot \mathbf{size})) \\
= \;& [\![\text{ Apply above equalities; } 1 \star ([\mathsf{top} \mapsto 0, 0] \cdot 1/2 \cdot \mathbf{size}) = [\mathsf{top} \hookrightarrow 0, 0]]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \mathsf{Y} \star (\underbrace{[\textit{tree}\,(\mathsf{top})] \cdot [\mathsf{top} \hookrightarrow 0, 0]}_{= [\mathsf{top} \mapsto 0,0]}) \\
= \;& [\![\text{ Definition of } [\textit{tree}\,(\mathsf{top})]]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \mathsf{Y} \star [\mathsf{top} \mapsto 0, 0] \\
= \;& [\![[\mathsf{top} \mapsto 0, 0] \cdot \log(1 + 1/2 \cdot \mathbf{size}) = [\mathsf{top} \mapsto 0, 0]]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \mathsf{Y} \star ([\mathsf{top} \mapsto 0, 0] \cdot \log(1 + 1/2 \cdot \mathbf{size})) \\
= \;& [\![\text{ D.5.3 }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u = 0 \wedge v = 0] \cdot \\
& \quad \mathsf{Y} \star ([\mathsf{top} \neq 0] \cdot [\mathsf{top} \mapsto 0, 0] \cdot \log(1 + 1/2 \cdot \mathbf{size})).
\end{aligned}
$$

□

Lemma F.16 Let $\mathsf{X}_{\mathsf{else}}$ be defined as in Abbildung 8.11. Then

$$
\begin{aligned}
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot \mathsf{X}_{\mathsf{else}}\, [\mathsf{r}/v]\, [\mathsf{l}/u]\, [\mathsf{root}/\boxdot\mathsf{top}]\, [+\boxdot] \\
\preceq \;& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot \\
& \quad \mathsf{Y} \star ([\mathsf{top} \neq 0] \cdot [\textit{tree}\,(\mathsf{top})] \cdot \log(1 + 1/2 \cdot \mathbf{size})).
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot \mathsf{X}_{\mathsf{else}}\, [\mathsf{r}/v]\, [\mathsf{l}/u]\, [\mathsf{root}/\boxdot\mathsf{top}]\, [+\boxdot] \\
= \;& [\![\text{ Definition of } \mathsf{X}_{\mathsf{else}} \text{ as in Abbildung 8.11 }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot (\\
& \quad 1/2 \cdot \underbrace{[u \neq v]}_{\preceq 1} \cdot \underbrace{\mathsf{Y}\, [-\boxdot]\, [\mathsf{n}/\mathsf{l}]\, [\mathsf{r}/v]\, [\mathsf{l}/u]\, [\mathsf{root}/\boxdot\mathsf{top}]\, [+\boxdot]}_{= \mathsf{Y}} \\
& \quad \star\, [\mathsf{top} \mapsto u, v] \star [\textit{tree}\,(v)] \star
\end{aligned}
$$

$$
\begin{aligned}
&\quad ([u \neq 0] \cdot [\mathit{tree}\,(u)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&+ 1/2 \cdot \underbrace{[u \neq v]}_{\preceq 1} \cdot \underbrace{\mathrm{Y}\,[-\boxdot]\,[\mathsf{n}/\mathsf{r}]\,[\mathsf{r}/v]\,[\mathsf{l}/u]\,[\mathsf{root}/\boxdot\mathsf{top}]\,[+\boxdot]}_{= \mathrm{Y}} \\
&\quad \star [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(u)] \star \\
&\quad\quad ([v \neq 0] \cdot [\mathit{tree}\,(v)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&+ \underbrace{[u = 0 \wedge v = 0]}_{= 0} \cdot \infty + \underbrace{[\neg \mathsf{top} \hookrightarrow u, v]}_{= 0} \cdot \\
&\quad\quad \mathrm{wp}_{\rho}^{\mathrm{rleaf}}\,[\mathsf{n} :\approx 1/2 \cdot \ldots\,;\, \mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\,(\ldots) \\
&\quad\quad\quad [\mathsf{r}/v]\,[\mathsf{l}/u]\,[\mathsf{root}/\boxdot\mathsf{top}]\,[+\boxdot]
\end{aligned}
$$

$$
\begin{aligned}
&) \\
\preceq\ & [\![\text{ apply the (in)equalities indicated above }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot (\\
&\quad 1/2 \cdot \mathrm{Y} \star [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(v)] \star \\
&\quad\quad ([u \neq 0] \cdot [\mathit{tree}\,(u)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&\quad + 1/2 \cdot \mathrm{Y} \star [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(u)] \star \\
&\quad\quad ([v \neq 0] \cdot [\mathit{tree}\,(v)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&) \\
=\ & [\![\mathrm{Y} \text{ and } [\mathit{tree}\,(\ldots)] \text{ are precise; D.3.2 }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot \mathrm{Y} \star (\\
&\quad 1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(v)] \star \\
&\quad\quad ([u \neq 0] \cdot [\mathit{tree}\,(u)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&\quad + 1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(u)] \star \\
&\quad\quad ([v \neq 0] \cdot [\mathit{tree}\,(v)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&) \\
\preceq\ & [\![\text{ Lemma F.17 }]\!] \\
& [\mathsf{top} \hookrightarrow u, v] \cdot [u \neq 0 \vee v \neq 0] \cdot \\
&\quad \mathrm{Y} \star ([\mathsf{top} \neq 0] \cdot [\mathit{tree}\,(\mathsf{top})] \cdot \log(1 + 1/2 \cdot \mathbf{size})).
\end{aligned}
$$

□

Lemma F.17

$$
\begin{aligned}
&1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(v)] \star \\
&\quad ([u \neq 0] \cdot [\mathit{tree}\,(u)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size})))
\end{aligned}
$$

$$
\begin{aligned}
&+ 1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(u)] \star \\
&\qquad ([v \neq 0] \cdot [\mathit{tree}\,(v)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
\preceq\ & [\mathsf{top} \neq 0] \cdot [\mathit{tree}\,(\mathsf{top})] \cdot \log(1 + 1/2 \cdot \mathbf{size})\,.
\end{aligned}
$$

Proof. Let X be defined as

$$
\begin{aligned}
&1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(v)] \star \\
&\qquad ([u \neq 0] \cdot [\mathit{tree}\,(u)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))) \\
&+ 1/2 \cdot [\mathsf{top} \mapsto u, v] \star [\mathit{tree}\,(u)] \star \\
&\qquad ([v \neq 0] \cdot [\mathit{tree}\,(v)] \cdot (1 + \log(1 + 1/2 \cdot \mathbf{size}))).
\end{aligned}
$$

Moreover, let $(\mathfrak{s}, \mathfrak{h})$ be a stack-heap pair with $X(\mathfrak{s}, \mathfrak{h}) > 0$ (for all other stack heap pairs the inequality $0 \preceq \ldots$ holds trivially). Since $[\mathsf{top} \mapsto u, v]$, $[\mathit{tree}\,(u)]$, and $[\mathit{tree}\,(v)]$ are precise expectations, there exists exactly one partition, say $\mathfrak{h} = \mathfrak{h}_1 \star \mathfrak{h}_2 \star \mathfrak{h}_3$, such that $[\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) > 0$, $[\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) > 0$ and $[\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) > 0$. With this in mind, consider the following:

$$
\begin{aligned}
& X(\mathfrak{s}, \mathfrak{h}) \\
=\ & [\![\text{ Definition of X; Definition of } \star\text{; Definition of } \mathbf{size} \text{ }]\!] \\
& \max\{1/2 \cdot [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1') \cdot [\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3') \\
& \qquad \cdot [\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2') \cdot (1 + \log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2')|)) \\
& \qquad \mid \mathfrak{h} = \mathfrak{h}_1' \star \mathfrak{h}_2' \star \mathfrak{h}_3'\} \\
& +\ \max\{1/2 \cdot [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1') \cdot [\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2') \\
& \qquad \cdot [\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3') \cdot (1 + \log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3')|)) \\
& \qquad \mid \mathfrak{h} = \mathfrak{h}_1' \star \mathfrak{h}_2' \star \mathfrak{h}_3'\} \\
=\ & [\![\ \mathfrak{h} = \mathfrak{h}_1 \star \mathfrak{h}_2 \star \mathfrak{h}_3 \text{ is the only partition leading to evaluations} > 0 \ ]\!] \\
& 1/2 \cdot [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \cdot \\
& \qquad [\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot (1 + \log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2)|)) \\
& +\ 1/2 \cdot [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot \\
& \qquad [\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \cdot (1 + \log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3)|)) \\
=\ & [\![\text{ elementary algebra }]\!] \\
& [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\mathit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot [\mathit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \\
& \qquad \cdot (1 + 1/2 \cdot (\log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2)|) + \log(1 + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3)|)))
\end{aligned}
$$

$$
\begin{aligned}
&\preceq \quad [\![\text{ Lemma F.18 with } n_1 = 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2)| \text{ and } n_2 = 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3)| \,]\!] \\
&\quad [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\textit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot [\textit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \\
&\qquad \cdot \log\,(1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2)| + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3)| + 2) \\
&= \quad [\![\, |\mathsf{dom}(\mathfrak{h}_1)| = 2 \,]\!] \\
&\quad [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\textit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot [\textit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \\
&\qquad \cdot \log\,(1/2 \cdot |\mathsf{dom}(\mathfrak{h}_2)| + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_3)| + 1/2 \cdot |\mathsf{dom}(\mathfrak{h}_1)| + 1) \\
&= \quad [\![\, (|\mathsf{dom}(\mathfrak{h})| = |\mathsf{dom}(\mathfrak{h}_1)| + |\mathsf{dom}(\mathfrak{h}_2)| + |\mathsf{dom}(\mathfrak{h}_3)| \,]\!] \\
&\quad [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1) \cdot [\textit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2) \cdot [\textit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3) \cdot \\
&\qquad \log\,(1 + |\mathsf{dom}(\mathfrak{h})|) \\
&= \quad [\![\, \mathfrak{h} = \mathfrak{h}_1 \star \mathfrak{h}_2 \star \mathfrak{h}_3 \text{ is the only partition leading to evaluations } > 0 \,]\!] \\
&\quad \max\{\, [\mathsf{top} \mapsto u, v]\,(\mathfrak{s}, \mathfrak{h}_1') \cdot [\textit{tree}\,(u)]\,(\mathfrak{s}, \mathfrak{h}_2') \cdot [\textit{tree}\,(v)]\,(\mathfrak{s}, \mathfrak{h}_3') \cdot \\
&\qquad \log\,(1 + |\mathsf{dom}(\mathfrak{h})|) \;\mid \mathfrak{h} = \mathfrak{h}_1' \star \mathfrak{h}_2' \star \mathfrak{h}_3' \,\} \\
&= \quad [\![\text{ Definition of } \star \text{; Definition of } \mathbf{size} \,]\!] \\
&\quad \big(\underbrace{[\mathsf{top} \mapsto u, v] \star [\textit{tree}\,(u)] \star [\textit{tree}\,(v)]}_{\preceq\, [\mathsf{top} \neq 0] \cdot [\textit{tree}(\mathsf{top})]} \cdot \log\,(1 + \mathbf{size}) \big)\,(\mathfrak{s}, \mathfrak{h}) \\
&\preceq \quad [\![\text{ Definition of } [\textit{tree}\,(\mathsf{top})] \,]\!] \\
&\quad ([\mathsf{top} \neq 0] \cdot [\textit{tree}\,(\mathsf{top})] \cdot \log\,(1 + \mathbf{size}))\,(\mathfrak{s}, \mathfrak{h}). \qquad \square
\end{aligned}
$$

Lemma F.18 For natural numbers $n_1, n_2 \in \mathbb{N}$, we have

$$1 + 1/2 \cdot \log\,(1 + n_1) + 1/2 \cdot \log\,(1 + n_2) \leq \log\,(n_1 + n_2 + 2)\,.$$

Proof. The following calculations are taken from the proof of Theorem 1 in [GM98]:

$$
\begin{aligned}
&\quad 1 + 1/2 \cdot \log\,(1 + n_1) + 1/2 \cdot \log\,(1 + n_2) \\
&= \quad [\![\text{ elementary algebra (logarithm laws) }]\!] \\
&\quad \log\left(2 \cdot \sqrt{(n_1 + 1)(n_2 + 1)}\right) \\
&\leq \quad [\![\text{ elementary algebra }]\!] \\
&\quad \log\left(2 \cdot \frac{(n_1 + 1) + (n_2 + 1)}{2}\right) \\
&= \quad [\![\text{ elementary algebra }]\!] \\
&\quad \log\,(n_1 + n_2 + 2)\,. \qquad \square
\end{aligned}
$$

$$
\begin{aligned}
&// \quad \sum_{k=0}^{n-1-i} \frac{1}{n-i} \cdot \sup_{u \in \mathbb{Z}} [\mathsf{array} + i \mapsto u] \star \big([\mathsf{array} + i \mapsto u] \mathrel{-\!\!\star} \\
&// \quad\quad \sup_{v \in \mathbb{Z}} [\mathsf{array} + i + k \mapsto v] \star \big([\mathsf{array} + i + k \mapsto v] \mathrel{-\!\!\star} \\
&// \quad\quad\quad [\mathsf{array} + i \mapsto -] \star \big([\mathsf{array} + i \mapsto v] \mathrel{-\!\!\star} \\
&// \quad\quad\quad\quad [\mathsf{array} + i + k \mapsto -] \star \big([\mathsf{array} + i + k \mapsto u] \mathrel{-\!\!\star} \\
&// \quad\quad\quad\quad\quad \mathrm{Z}\,[i/i+1]\,[-\boxdot]\,[\boxdot x/\mathsf{out}]\,[\mathsf{out}/0]\,[z/v]\,[y/u] \\
&// \quad\quad\quad\quad\quad\quad [\mathsf{array}/\boxdot\mathsf{array}]\,[i/\boxdot i]\,[j/\boxdot j]\,[+\boxdot]\,[j/i+k]\big)\big)\big)\big) \\
&// \quad \succeq \quad \llbracket \text{ Figure F.2 (page 437) } \rrbracket \\
&// \quad \sum_{k=0}^{n-1-i} \frac{1}{n-i} \cdot \mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,(\mathrm{Z}\,[i/i+1])\,[j/i+k] \\
&j :\approx \mathsf{uniform}\,(i, n-1)\,; \\
&// \quad \mathsf{wp}\,[\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)]\,(\mathrm{Z}\,[i/i+1]\,) \\
&\mathsf{void} := \mathrm{swap}(\mathsf{array}, i, j)\,; \\
&// \quad \mathrm{Z}\,[i/i+1] \\
&i := i+1 \\
&// \quad \mathrm{Z}
\end{aligned}
$$

Figure F.1: Computation of $\mathsf{wp}\,[C_{\mathsf{body}}]\,(\mathrm{Z})$ for every $\mathrm{Z} \in \mathbb{E}$.

```
//  sup_{u∈ℤ} [array + i ↦ u] ⋆ ( [array + i ↦ u] −⋆
//    sup_{v∈ℤ} [array + j ↦ v] ⋆ ( [array + j ↦ v] −⋆
//      [array + i ↦ −] ⋆ ( [array + i ↦ v] −⋆
//        [array + j ↦ −] ⋆ ( [array + j ↦ u] −⋆
//          Z [−⊡] [⊡x/out] [out/0] [z/v] [y/u]
//            [array/⊡array] [i/⊡i] [j/⊡j] [+⊡]))))
swap(array, i, j) {  //  enter scope & set parameters
    //  sup_{u∈ℤ} [array + i ↦ u] ⋆ ( [array + i ↦ u] −⋆
    //    sup_{v∈ℤ} [array + j ↦ v] ⋆ ( [array + j ↦ v] −⋆
    //      [array + i ↦ −] ⋆ ( [array + i ↦ v] −⋆
    //        [array + j ↦ −] ⋆ ( [array + j ↦ u] −⋆
    //          Z [−⊡] [⊡x/out] [out/0] [z/v] [y/u])))))
    y := array [ i ] ;
    //  sup_{v∈ℤ} [array + j ↦ v] ⋆ ( [array + j ↦ v] −⋆
    //    [array + i ↦ −] ⋆ ( [array + i ↦ v] −⋆
    //      [array + j ↦ −] ⋆ ( [array + j ↦ y] −⋆ Z [−⊡] [⊡x/out] [out/0] [z/v])))
    z := array [ j ] ;
    //  [array + i ↦ −] ⋆ ( [array + i ↦ z] −⋆
    //    [array + j ↦ −] ⋆ ( [array + j ↦ y] −⋆ Z [−⊡] [⊡x/out] [out/0] ))
    array [ i ] := z ;
    //  [array + j ↦ −] ⋆ ([array + j ↦ y] −⋆ Z [−⊡] [⊡x/out] [out/0] )
    array [ j ] := y ;
    //  Z [−⊡] [⊡x/out] [out/0]
    out := 0
    //  Z[−⊡] [⊡x/out]
} //  leave scope & set return value
//  Z
```

Figure F.2: Computation of wp [void := swap(array, i, j)] (Z) for any Z ∈ 𝔼.

$$
\begin{array}{l}
\mathit{//}\quad 0 \\
\mathit{//}\quad \succeq \quad [\![\text{ elementary algebra }]\!] \\
\mathit{//}\quad 1/2 \cdot 0 \cdot [\mathsf{l} \neq \mathsf{r}] \cdot \mathsf{wp}_\rho^{\mathsf{rleaf}}\,[\mathsf{out} := \mathsf{rleaf}(\mathsf{n})]\,(\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r} \cdot X_1) \\
\mathit{//}\quad\quad + 1/2 \cdot [\mathsf{r} \neq \mathsf{l}] \cdot 0 \cdot \mathsf{wp}_\rho^{\mathsf{rleaf}}\,[\mathsf{out} := \mathsf{rleaf}(\mathsf{n})]\,(\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r} \cdot X_1) \\
\mathsf{n} :\approx 1/2 \cdot \langle\, \mathsf{l} \,\rangle + 1/2 \cdot \langle\, \mathsf{r} \,\rangle\,; \\
\mathit{//}\quad [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot \mathsf{wp}_\rho^{\mathsf{rleaf}}\,[\mathsf{out} := \mathsf{rleaf}(\mathsf{n})]\,([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot X_1) \\
\mathit{//}\quad \succeq \quad [\![\text{ Theorem 7.11 }]\!] \\
\mathit{//}\quad \mathsf{wp}_\rho^{\mathsf{rleaf}}\,[\mathsf{out} := \mathsf{rleaf}(\mathsf{n})]\,([\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \cdot X_1) \\
\mathsf{out} := \mathsf{rleaf}(\mathsf{n}) \\
\mathit{//}\quad [\boxdot\mathsf{top} \hookrightarrow \mathsf{l},\mathsf{r}] \cdot [\mathsf{n} \neq \mathsf{l}] \cdot [\mathsf{n} \neq \mathsf{r}] \\
\mathit{//}\quad\quad \cdot \underbrace{Y\,[-\boxdot] \star ([\mathit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\mathit{path}\,(\boxdot\mathsf{top},\mathsf{out})] \cdot 1/2 \cdot \mathbf{size}))}_{\triangleq X_1}
\end{array}
$$

Figure F.3: Proof of the `else` branch of invariant case (1) for $\mathsf{n} \neq \mathsf{l}$ and $\mathsf{n} \neq \mathsf{r}$.

$$
\begin{array}{ll}
// & [\mathsf{l} = 0 \wedge \mathsf{r} = 0] \cdot \infty \\
// & \quad \succeq \quad [\![\text{ elementary algebra }]\!] \\
// & [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty \\
\mathsf{n} :\approx 1/2 \cdot \langle \mathsf{l} \rangle + 1/2 \cdot \langle \mathsf{r} \rangle ; & \\
// & [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty \\
// & \quad \succeq \quad [\![\text{ Theorem 7.11; D.6.5 }]\!] \\
// & \mathsf{wp}_{\rho}^{\mathsf{rleaf}} [\mathsf{out} := \mathsf{rleaf}(\mathsf{n})]([\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty) \\
\mathsf{out} := \mathsf{rleaf}(\mathsf{n}) & \\
// & [\mathsf{l} = 0] \cdot [\mathsf{r} = 0] \cdot \infty \\
// & \quad \succeq \quad [\![\text{ Lemma F.14 }]\!] \\
// & [\boxdot\mathsf{top} \hookrightarrow \mathsf{l}, \mathsf{r}] \cdot [\mathsf{n} = \mathsf{l}] \cdot [\mathsf{n} = \mathsf{r}] \\
// & \qquad \cdot \underbrace{\mathsf{Y}\,[-\boxdot] \star ([\mathit{tree}\,(\boxdot\mathsf{top})] \cdot 1 \star ([\mathit{path}\,(\boxdot\mathsf{top}, \mathsf{out})] \cdot 1/2 \cdot \mathbf{size}))}_{\triangleq\, X_1}
\end{array}
$$

Figure F.4: Proof of the `else` branch of invariant case (1) for $\mathsf{n} = \mathsf{l}$ and $\mathsf{n} = \mathsf{r}$.

```
//  0
//   ≥   ⟦ [top ↪ u, v] · [tree (top)] = [top ↦ u, v] ⋆ [tree (u)] ⋆ [tree (v)]; D.6.7; D.5.4 ⟧
//   sup_{u,v∈ℤ} [top ↪ u, v] · (
//     [u = 0 ∧ v = 0] · [tree (top)] ⋆ ([top ↪ −] · ∞) + [u ≠ 0 ∨ v ≠ 0] · (
//       [¬top ↪ u ∧ ¬top + 1 ↪ u] · ∞ + [¬top ↪ v ∧ ¬top + 1 ↪ v] · ∞))
rleaf(root) {  //  enter scope & set parameters
    //  sup_{u,v∈ℤ} [root ↪ u, v] · (
    //    [u = 0 ∧ v = 0] · [tree (▣top)] ⋆ ([root ↪ −] · ∞) + [u ≠ 0 ∨ v ≠ 0] · (
    //      [¬ ▣ top ↪ u ∧ ¬ ▣ top + 1 ↪ u] · ∞ + [¬ ▣ top ↪ v ∧ ¬ ▣ top + 1 ↪ v] · ∞))
    l := <root>;
    //  sup_{v∈ℤ} [root + 1 ↪ v] · (
    //    [l = 0 ∧ v = 0] · ([tree (▣top)] ⋆ [root ↪ −] · ∞) + [l ≠ 0 ∨ v ≠ 0] · (
    //      [¬ ▣ top ↪ l ∧ ¬ ▣ top + 1 ↪ l] · ∞ + [¬ ▣ top ↪ v ∧ ¬ ▣ top + 1 ↪ v] · ∞))
    r := <root + 1>;  //  use alternative lookup rule (D.4.5)
    //  [l = 0 ∧ r = 0] · ( [tree (▣top)] ⋆ ([root ↪ −] · ∞)) + [l ≠ 0 ∨ r ≠ 0] · (
    //    [¬ ▣ top ↪ l ∧ ¬ ▣ top + 1 ↪ l] · ∞ + [¬ ▣ top ↪ r ∧ ¬ ▣ top + 1 ↪ r] · ∞)
    if (l = 0 and r = 0) {
        //  [tree (▣top)] ⋆ ([root ↪ −] · ∞)
        out := root
        //  [tree (▣top)] ⋆ ([out ↪ −] · ∞)
    } else {
        //  [¬ ▣ top ↪ l ∧ ¬ ▣ top + 1 ↪ l] · ∞ +  [¬ ▣ top ↪ r ∧ ¬ ▣ top + 1 ↪ r] · ∞
        n :≈ 1/2 · ⟨ l ⟩ + 1/2 · ⟨ r ⟩;
        //  [¬ ▣ top ↪ n ∧ ¬ ▣ top + 1 ↪ n] · ∞
        //   ≥   ⟦ Figure F.6 ⟧
        //  wp^{rleaf}_ρ [out := rleaf(n)] ( [tree (▣top)] ⋆ [out ↪ −] · ∞)
        out := rleaf(n)
        //  [tree (▣top)] ⋆ ([out ↪ −] · ∞)
    }
    //  [tree (▣top)] ⋆ ([out ↪ −] · ∞)
}  //  set return value & leave scope
//  [tree (top)] ⋆ ([result ↪ −] · ∞)
```

Figure F.5: Proof of invariant case (2)

$$
\begin{array}{ll}
// & [\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot \infty \\
// & \quad \succeq \quad [\![\text{ elementary algebra }]\!] \\
// & 2 \cdot 0 + [\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot \underbrace{\mathsf{wp}_\rho^{\mathsf{rleaf}}\, [\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\, (\infty)}_{\preceq\, \infty} \\
// & \quad \succeq \quad [\![\text{ Apply invariant } \rho\text{; Theorem 7.12 }]\!] \\
// & 2 \cdot \mathsf{wp}_\rho^{\mathsf{rleaf}}\, [\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\, ([\boldsymbol{tree}\, (\mathsf{n})] \star ([\mathsf{out} \hookrightarrow -] \cdot \infty)) \\
// & \quad + \mathsf{wp}_\rho^{\mathsf{rleaf}}\, [\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\, ([\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot \infty) \\
// & \quad \succeq \quad [\![\text{ Linearity of weakest preexpectations (Theorem 7.4 (f)) }]\!] \\
// & \mathsf{wp}_\rho^{\mathsf{rleaf}}\, [\mathsf{out} := \mathrm{rleaf}(\mathsf{n})]\, (X_1) \\
& \mathsf{out} := \mathrm{rleaf}(\mathsf{n}) \\
// & \underbrace{2 \cdot [\boldsymbol{tree}\, (\mathsf{n})] \star ([\mathsf{out} \hookrightarrow -] \cdot \infty) + [\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot \infty}_{\triangleq\, X_1} \\
// & \quad \succeq \quad [\![\text{ elementary algebra; } 1 \star [\boxdot \mathsf{top} \hookrightarrow x] = [\boxdot \mathsf{top} \hookrightarrow x] \,]\!] \\
// & \underbrace{[\boxdot \mathsf{top} \hookrightarrow \mathsf{n}] \cdot [\boldsymbol{tree}\, (\boxdot \mathsf{top})]}_{\preceq\, [\boldsymbol{tree}(\mathsf{n})] \star 1} \star ([\mathsf{out} \hookrightarrow -] \cdot \infty) \\
// & \quad + \underbrace{[\boxdot \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot [\boldsymbol{tree}\, (\boxdot \mathsf{top})]}_{\preceq\, [\boldsymbol{tree}(\mathsf{n})] \star 1} \star ([\mathsf{out} \hookrightarrow -] \cdot \infty) \\
// & \quad + [\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot \underbrace{[\boldsymbol{tree}\, (\boxdot \mathsf{top})]}_{\preceq\, 1} \star ([\mathsf{out} \hookrightarrow -] \cdot \infty) \\
// & \quad \succeq \quad [\![\text{ elementary algebra } ([P \vee Q] \;\preceq\; [P] + [Q]) \,]\!] \\
// & [\boxdot \mathsf{top} \hookrightarrow \mathsf{n} \vee \boxdot \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot [\boldsymbol{tree}\, (\boxdot \mathsf{top})] \star [\mathsf{out} \hookrightarrow -] \cdot \infty \\
// & + [\neg\, \boxdot\, \mathsf{top} \hookrightarrow \mathsf{n} \wedge \neg\, \boxdot\, \mathsf{top}+1 \hookrightarrow \mathsf{n}] \cdot [\boldsymbol{tree}\, (\boxdot \mathsf{top})] \star [\mathsf{out} \hookrightarrow -] \cdot \infty \\
// & \quad \succeq \quad [\![\text{ Case distinction on } [\boxdot \mathsf{top} \hookrightarrow \mathsf{n} \vee \boxdot \mathsf{top}+1 \hookrightarrow \mathsf{n}] \,]\!] \\
// & [\boldsymbol{tree}\, (\boxdot \mathsf{top})] \star ([\mathsf{out} \hookrightarrow -] \cdot \infty)
\end{array}
$$

Figure F.6: $\mathsf{wp}_\rho^{\mathsf{rleaf}}\, [\mathsf{leaf} := \mathrm{rleaf}(\mathsf{top})]\, ([\boldsymbol{tree}\, (\boxdot \mathsf{top})] \star ([\mathsf{leaf} \hookrightarrow -] \cdot \infty))$

```
//  [top ≠ 0] · Z
//    ≽    ⟦ elementary algebra ⟧
//  sup_{u,v∈ℤ} [top ↪ u, v] · ( [u = 0 ∧ v = 0] · Z [−⧈] [out/root] [r/v] [l/u] [root/⧈top] [+⧈]
//                  ≼ [top≠0]                                  ≼ Z
//     + [u ≠ 0 ∨ v ≠ 0] · (1/2 · Z [−⧈] [n/l] [r/v] [l/u] [root/⧈top] [+⧈]
//                                                  ≼ Z
//        +1/2 · Z [−⧈] [n/r] [r/v] [l/u] [root/⧈top] [+⧈] ))
//                                  ≼ Z
rleaf(root) {  //  enter scope & set parameters
    //  sup_{u,v∈ℤ} [root ↪ u, v] · ( [u = 0 ∧ v = 0] · Z [−⧈] [out/root] [r/v] [l/u]
    //     + [u ≠ 0 ∨ v ≠ 0] · (1/2 · Z [−⧈] [n/l] [r/v] [l/u] + 1/2 · Z [−⧈] [n/r] [r/v] [l/u]))
    l := <root> ;
    //  sup_{v∈ℤ} [root + 1 ↪ v] · ( [l = 0 ∧ v = 0] · Z [−⧈] [out/root] [r/v]
    //        + [l ≠ 0 ∨ v ≠ 0] · (1/2 · Z [−⧈] [n/l] [r/v] + 1/2 · Z [−⧈] [n/r] [r/v]))
    r := <root + 1> ;  //  use alternative lookup rule (D.4.5)
    //  [l = 0 ∧ r = 0] · Z [−⧈] [out/root]
    //   + [l ≠ 0 ∨ r ≠ 0] · (1/2 · Z [−⧈] [n/l] + 1/2 · Z [−⧈] [n/r] )
    if (l = 0 and r = 0) {
        //  Z [−⧈] [out/root]
        out := root
        //  Z [−⧈]
    } else {
        //  1/2 · Z [−⧈] [n/l] + 1/2 · Z [−⧈] [n/r]
        n :≈ 1/2 · ⟨ l ⟩ + 1/2 · ⟨ r ⟩ ;
        //  Z [−⧈]
        //    ≽    ⟦ Apply invariant ρ; D.6.1 ⟧
        //  wp_ρ^rleaf [out := rleaf(n)] (Z [−⧈])
        out := rleaf(n)
        //  Z [−⧈]
    }
    //  Z [−⧈]
    //    ≽    ⟦ elementary algebra (out ∉ Vars (Z)) ⟧
    //  Z [−⧈] [⧈out/out]
} //  set return value & leave scope
//  Z
```

Figure F.7: Proof of invariant case (3)

Let $S = \{ s_1, \ldots, s_n \}$ be a finite set. A *multiset* M over S is a function

$$M \colon S \to \mathbb{N},$$

mapping each element in S to its multiplicity. We denote by $\emptyset$ the *empty multiset*, i.e., the function $\lambda s.\, 0$. Furthermore, we often write

$$\{\{ \underbrace{s_1, \ldots, s_1}_{k_1 \text{ times}}, \ldots, \underbrace{s_n, \ldots, s_n}_{k_n \text{ times}} \}\}$$

to denote the multiset

$$M \colon S \to \mathbb{N}, \qquad M(s_i) = k_i,\ i \in \{ 1, \ldots, n \}.$$

Finally, we define the following operators for multisets:

- We say that s is an *element* of multiset M, written $s \in M$, iff $M(s) > 0$.
- The *union* $M_1 \uplus M_2$ of two multisets M_1 and M_2 is defined as

$$M_1 \uplus M_2 \colon S \to \mathbb{N}, \qquad (M_1 \uplus M_2)(s) \triangleq M_1(s) + M_2(s).$$

- The *difference* $M_1 \setminus\!\setminus M_2$ of multisets M_1 and M_2 is defined as

$$M_1 \setminus\!\setminus M_2 \colon S \to \mathbb{N}, \qquad (M_1 \setminus\!\setminus M_2)(s) \triangleq \max \{ 0, M_1(s) - M_2(s) \}.$$

H.1 Proof of Theorem 12.6 (NP-hardness)

Our goal is to show that folding problem defined below is NP-hard.

Definition H.1 (Folding Problem) The *folding problem for symbolic heaps* SH-Fold amounts to the following question: Given a shrinking SID Ψ and symbolic heaps φ and ψ, does $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$ hold?

The proof is inspired by a similar proof for regular DNLC graph grammars by Aalbersberg, Rozenberg, and Ehrenfeucht [ARE86, Theorem 1]. Before we turn to the actual hardness proof, some preparation is needed.

First, we fix the sequence of selectors $\mathbf{Sel} \triangleq \langle a, b \rangle$ and the set of predicate symbols $\mathbf{PSym} \triangleq \{ \boldsymbol{P} \}$. Furthermore, let $\psi \triangleq \exists y\colon \exists z\colon \boldsymbol{P}(y, z)$, where $\boldsymbol{P}$ is determined by the SID Ψ defined below:

$$\begin{aligned}\Psi \triangleq \{\ & \boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.b \mapsto 0, x_2.a \mapsto 0, \\ & \boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.b \mapsto y, x_2.a \mapsto z \mid \boldsymbol{P}(y, z), \\ & \boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.b \mapsto 0, x_2.a \mapsto 0 \mid \boldsymbol{P}(y, z), \\ & \boldsymbol{P} \Leftarrow x_1, x_2 \mid x_1.b \mapsto y, x_2.a \mapsto 0 \mid \boldsymbol{P}(y, z)\ \}.\end{aligned}$$

Intuitively, every unfolding $\vartheta \in \mathbf{Unf}_\Psi(\psi)$ consists of sets of points-to assertions which describe singly-linked lists, where every list contains either only selector a or selector b. We refer to a list containing only a's as an "a-path"; analogously, a "b-path" is a list containing only b's. For any a-path or b-path P, we write $|P|$ to denote its length, i.e., the number points-to assertions that belong to P.

More precisely, Ψ always specifies one a-labeled and one b-labeled path in parallel. During unfolding $\boldsymbol{P}$, Ψ may nondeterministically decide to "terminate" a path by pointing to 0 and start a new one under the proviso that terminating

a b-path implies terminating the corresponding a-path we well. An example of an unfolding $\vartheta \in \mathbf{Unf}_\Psi(\psi)$ is depicted below:

$$\vartheta \;=\; \varepsilon \mid \underbrace{y_1.a \mapsto y_2, y_2.a \mapsto y_3, y_4.a \mapsto 0,}_{a\text{-path of length 3}} \quad \underbrace{y_5.a \mapsto 0,}_{a\text{-path of length 1}} \quad \underbrace{y_6.a \mapsto 0,}_{a\text{-path of length 1}}$$
$$\underbrace{z_1.b \mapsto z_2, z_2.b \mapsto z_3, z_3.b \mapsto z_4, z_4.b \mapsto 0,}_{b\text{-path of length 4}} \quad \underbrace{z_5.b \mapsto 0}_{b\text{-path of length 1}}$$

By construction of Ψ, we observe that the set of unfoldings $\mathbf{Unf}_\Psi(\psi)$ can be characterized as follows (cf. [ARE86, Lemma 1]):

Lemma H.2 Let $\vartheta \in \mathbf{SHSL}_0$ be a graphical symbolic heap with $\mathbf{Vars}(\vartheta) = \emptyset$ which is a non-empty union of a-paths and b-paths. Moreover, let $S_b = \{ B_1, \ldots, B_m \}$ be the set of disjoint b-paths in ϑ; analogously, let S_a be the set of disjoint a-paths in ϑ. Then $\vartheta \in \mathbf{Unf}_\Psi(\psi)$ iff there exists a partition $\langle S_1^a, \ldots, S_m^a \rangle$ of S_a such that

$$\forall i \in \{1, \ldots, m\} : \qquad |B_i| \;=\; \sum_{A \in S_i^a} |A|.$$

We are now in a position to prove that SH-Fold is NP-hard.

Proof (Proof of Theorem 12.6). We construct a polynomial-time reduction of the well-known NP-complete 3-Partition problem to the SH-Fold problem. Formally, the 3-Partition is defined as follows (cf. [GJ75]): An instance of 3-Partition is a tuple $\langle S, n, k, f \rangle$, where

- $n \geq 1$ is a natural number,
- S is a finite set consisting of $3n$ elements,
- k is a positive integer, and
- $f \colon S \to \mathbb{N}$ is a function mapping each element of S to a natural number such that, for every $s \in S$, we have $k/4 < f(s) < k/2$, and $\sum_{s \in S} f(s) = k \cdot n$.

The 3-Partition problem then asks whether S can be partitioned into sets $S_1, \ldots, S_n$ such that, for every $1 \leq i \leq n$, we have $\sum_{s \in S_i} f(s) = k$.

Now, let $\langle S, n, k, f \rangle$ be an instance of the 3-Partition problem. We then define the graphical symbolic heap φ (without free variables) as follows:

- for every $1 \leq i \leq n$, φ contains a b-path of length k, and

- for every $s \in S$, φ contains an a-path of length $f(s)$.

Clearly, φ can be computed in polynomial time in the size of $\langle S, n, k, f \rangle$. Moreover, both Ψ and ψ can be computed in constant time since neither depends on $\langle S, n, k, f \rangle$. Hence, the instance $\langle \Psi, \varphi, \psi \rangle$ of the SH-Fold problem can be constructed in polynomial time.

We then claim that $\langle S, n, k, f \rangle$ satisfies the 3-Partition problem if and only if $\langle \Psi, \varphi, \psi \rangle$ satisfies the SH-Fold problem.

Let us first assume that $\langle S, n, k, f \rangle$ satisfies the 3-Partition problem. Then consider the following:

- By definition, this means that S can be partitioned into sets $S_1, \ldots, S_n$ such that, for every $1 \leq i \leq n$, we have $\sum_{s \in S_i} f(s) = k$.
- Since $n \geq 1$, φ contains at least one a-path and one b-path.
- Let S_a be the set of disjoint a-paths in φ. By construction, there is a one-to-one correspondence between elements of S and a-paths in S_a.
- We can thus partition S_a into $S_1^a, \ldots, S_n^a$ in the same way as we partitioned S into $S_1, \ldots, S_n$ such that $s \in S_i$ iff the corresponding a-path is in S_i^a for all $i \in \{1, \ldots, n\}$.
- Let $S_b = \{B_1, \ldots, B_n\}$ be the set of disjoint b-paths in φ. By construction, there are exactly n such paths—each having length k.
- Consequently, we have:

$$\forall i \in \{1, \ldots, n\} : \qquad |B_i| \;=\; k \;=\; \sum_{s \in S_i} f(s) = \sum_{A \in S_i^a} |A|.$$

- By Lemma H.2 this means that $\varphi \in \mathbf{Unf}_\Psi(\psi)$ and thus also $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$.
- Hence, $\langle \Psi, \varphi, \psi \rangle$ satisfies SH-Fold.

Now, assume $\langle \Psi, \varphi, \psi \rangle$ satisfies SH-Fold and consider the following:

- By definition, $\varphi \overset{*}{\underset{\Psi}{\Rightarrow}} \psi$. In particular, since φ contains no predicate calls by construction, we have $\varphi \in \mathbf{Unf}_\Psi(\psi)$.
- Let $S_b = \{B_1, \ldots, B_n\}$ be the set of disjoint b-paths in φ. By construction, there are exactly n such paths—each having length k.

- By Lemma H.2 the set S_a of a-paths in φ can be partitioned into $\langle S_1^a, \ldots, S_n^a \rangle$ such that

$$\forall i \in \{1, \ldots, n\} : \qquad k = |B_i| = \sum_{A \in S_i^a} |A|.$$

- Since there is a one-to-one correspondence between elements of S and a-paths in S_a, S can be partitioned in the same way as S_a.
- Then there is a partition $\langle S_1, \ldots, S_n \rangle$ of S such that, for every $1 \leq i \leq n$, we have $\sum_{s \in S_i} f(s) = k$.
- Hence, $\langle S, n, k, f \rangle$ satisfies the 3-Partition problem.

Since the 3-Partition problem is (strongly) NP-complete, it follows that SH-Fold is NP-hard. □

H.2 Proof of Lemma 12.9

Our goal is to show that, for all SIDs Ψ and symbolic heaps $\varphi \in \mathbf{SHSL}$, we have

$$\mathbf{Unf}_\Psi(\varphi) = \begin{cases} \{\varphi\}, & \text{if } \mathbf{PC}(\varphi) = \emptyset \\ \bigcup_{\varphi \underset{\Psi}{\Leftarrow} \psi} \mathbf{Unf}_\Psi(\psi), & \text{otherwise.} \end{cases}$$

Proof. We distinguish two cases: $\mathbf{PC}(\varphi) = \emptyset$ and $\mathbf{PC}(\varphi) \neq \emptyset$.

For the case $\mathbf{PC}(\varphi) = \emptyset$, consider the following:

$$\begin{aligned} & \mathbf{Unf}_\Psi(\varphi) \\ = \; & [\![\text{ Definition 10.21 }]\!] \\ & \left\{ \vartheta \;\middle|\; \varphi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta \text{ and } \mathbf{PC}(\vartheta) = \emptyset \right\} \\ = \; & [\![\text{ Since } \mathbf{PC}(\varphi) = \emptyset, \text{ we have } \vartheta \cong \varphi \;]\!] \\ & \{\varphi\}. \end{aligned}$$

For the case $\mathbf{PC}(\varphi) \neq \emptyset$, consider the following:

$$\begin{aligned} & \mathbf{Unf}_\Psi(\varphi) \\ = \; & [\![\text{ Definition 10.21 }]\!] \end{aligned}$$

$$
\begin{aligned}
&\left\{ \vartheta \;\middle|\; \varphi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta \text{ and } \mathbf{PC}(\vartheta) = \emptyset \right\} \\
&= [\![\mathbf{PC}(\varphi) \neq \emptyset \text{ implies } \varphi \notin \mathbf{Unf}_\Psi(\varphi)]\!] \\
&\quad \left\{ \vartheta \;\middle|\; \exists \psi \colon \varphi \underset{\Psi}{\Leftarrow} \psi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta \text{ and } \mathbf{PC}(\vartheta) = \emptyset \right\} \\
&= [\![\text{elementary algebra}]\!] \\
&\quad \bigcup_{\varphi \underset{\Psi}{\Leftarrow} \psi} \left\{ \vartheta \;\middle|\; \psi \overset{*}{\underset{\Psi}{\Leftarrow}} \vartheta \text{ and } \mathbf{PC}(\vartheta) = \emptyset \right\} \\
&= [\![\text{Definition 10.21}]\!] \\
&\quad \bigcup_{\varphi \underset{\Psi}{\Leftarrow} \psi} \mathbf{Unf}_\Psi(\psi). \qquad \square
\end{aligned}
$$

H.3 Undecidability of the Entailment Problem

Our goal is to show that the question whether

$$\mathbf{Unf}_\Psi(\varphi) \subseteq \mathbf{Unf}_\Psi(\psi)$$

holds for an SID Ψ and symbolic heaps φ and ψ is undecidable.

In particular, we show that this even holds if Ψ, φ, and ψ are graphical. Moreover, we construct ψ such that it is folded.

Proof (Sketch). We reduce the undecidable inclusion problem for context-free string grammars (cf. [BPS64; Gre68]) to the above question. To this end, for $i \in \{1, 2\}$, let $G_i = \langle N_i, T, P_i, S_i \rangle$ be a context-free grammar with set of nonterminals N_i, set of terminals T, set of production rules P_i, and starting symbol S_i. Without loss of generality, assume that $N_1 \cap N_2$. Moreover, we assume that both grammars do no generate the empty word and that all production rules are in Greibach normal form, i.e., they are either of the form $X \to a$ or $X \to aYZ$, where $a \in T$ and $X, Y, Z \in N_i$ for $i \in \{1, 2\}$.

To prove undecidability of the unfolding inclusion problem, we define an SID Ψ and predicate calls $\boldsymbol{P}(x_1, x_2)$, $\boldsymbol{Q}(x_1, x_2)$ such that

$$\mathbf{Unf}_\Psi(\boldsymbol{P}(x_1, x_2)) \subseteq \mathbf{Unf}_\Psi(\boldsymbol{Q}(x_1, x_2)) \quad \text{iff} \quad L(G_1) \subseteq L(G_2),$$

where $L(G_i)$ denotes the language generated by grammar G_i.

Let **Sel** be a total ordering of the set of terminal symbols T. Moreover, we choose $N_1 \cup N_2$ as the set of predicate symbols; each symbol being assigned rank two. We then define the SID Ψ as follows:

$$\begin{aligned}\Psi \triangleq\ & \{\, X \Leftarrow x_1, x_2 \,\mathrel{!}\, x_1.a \mapsto x_2 \mid X \to a \in P_i, i \in \{1,2\} \,\} \\ \cup\ & \{\, X \Leftarrow x_1, x_2 \,\mathrel{!}\, x_1.a \mapsto y \,\mathrel{!}\, Y(y,z), Z(z,x_2) \mid X \to aYZ \in P_i, i \in \{1,2\} \,\}\end{aligned}$$

Moreover, we choose the starting symbols S_i for the top-level predicate calls, i.e., we set $\boldsymbol{P}(x_1, x_2) = S_1(x_1, x_2)$ and $\boldsymbol{Q}(x_1, x_2) = S_2(x_1, x_2)$. □

Contributions of the Author to Covered Publications

By the (occasionally byzantine) regulations for doctoral studies at RWTH Aachen University, I am required to indicate my own contributions for all prior peer-reviewed publications covered in this thesis. All publications listed in Section 1.5 resulted from joint work with many different people and involved lots of meetings, teleconferences, discussions about definitions and proofs, programming hours, writing, and—occasionally—groups staring together at a whiteboard. Consequently, I consider it hardly possible to give a detailed account of every individual contribution years after conducting the actual research. I was generally involved in the development of the theory and the writing of all relevant publications. I will thus focus on contributions to the publications most relevant to this thesis which—to the best of my knowledge—appear to be rather clear-cut.

For roughly three years, I have been the main developer of the verification tool Attestor, which is described in [4]. I am grateful to Christina Jansen who contributed her experience with Attestor's predecessor Juggrnaut. She also implemented the first LTL model checking component of Attestor. Apart from that, most components of Attestor have been implemented by myself together with my student assistant Hannah Arndt. Furthermore, I supervised seven students who worked with Attestor either as student assistants, interns, or as part of their bachelor's and master's projects. In particular, Hannah Arndt added support for reasoning about balanced data structures as described in [3]. Johannes Schulte implemented the component for confluence checking (cf. [Sch19]). Sally Chau [Cha19] implemented two alternative model checking algorithms to enable hierarchical and on-the-fly model-checking.

The research resulting in [2] was initiated by myself after I encountered the state labeling problem in Attestor. I also worked out most of the theoretical results presented in the paper. After a visit in Vienna, I got Jens Katelaan from TU Wien on board who implemented the described algorithmic framework. This resulted in the tool Harrsh which is also covered in [9]. Both of us wanted to apply heap automata to solve entailments right from the beginning. After exchanging countless drafts, we came up with the approach in [5]. While I was involved in the formalization, the finally published version is due to Jens.

The initiative for the research conducted in [1] came from me as part of a very productive collaboration with Benjamin Kaminski, Joost-Pieter Katoen, and Federico Olmedo on reasoning about probabilistic programs. In particular, the quantitative separating connectives have been jointly developed by Benjamin and myself. I believe I did a large share of the proofs involving the weakest preexpectation transformer, in particular regarding soundness, the frame rule, and the case studies. Our student assistant Kevin Batz supported us primarily in conducting the case studies. The case study on verifying randomized meldable heaps has been considered by my former student assistant Hannah Arndt as part of her master's thesis [Arn19] which I actively supervised. The actual proof presented in this thesis, however, is different from hers. Finally, upon a suggestion by Hongseok Yang, I generalized the quantitative separating implication to allow for arbitrary expectations on the left-hand side.

Eidesstattliche Erklärung / Declaration of Authorship

According to §12 (3) of the doctoral regulations of RWTH Aachen University, I am required to give the following declaration.

I, Christoph Matheja,
declare that this thesis and the work presented in it are my own and have been generated by me as the result of my own original research.

Hiermit erkläre ich an Eides statt / I do solemnly swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before as indicated in Section 1.5, page 10.

Christoph Matheja, Aachen, January 20, 2020

[1] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative Separation Logic: A Logic for Reasoning about Probabilistic Pointer Programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 34:1–34:29 (cit. on pp. 4, 8, 11, 92, 124, 125, 137, 139, 143, 147, 159, 165, 167, 169, 173, 176, 178–180, 184, 188, 192, 194, 197, 199, 206–208, 213, 217, 219, 221, 222, 224, 225, 231, 452).

[2] Christina Jansen, Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "Unified Reasoning about Robustness Properties of Symbolic-Heap Separation Logic". In: *European Symposium on Programming (ESOP)*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 611–638 (cit. on pp. 9, 11, 118, 120, 267, 284–286, 289, 294, 295, 299, 301–304, 307, 309–312, 314, 316–318, 321, 326, 451).

[3] Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll. "Graph-Based Shape Analysis Beyond Context-Freeness". In: *Software Engineering and Formal Methods (SEFM)*. Vol. 10886. Lecture Notes in Computer Science. Springer, 2018, pp. 271–286 (cit. on pp. 10, 11, 285, 323, 343, 344, 354, 361, 451).

[4] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Let this Graph Be Your Witness! - An Attestor for Verifying Java Pointer Programs". In: *Computer Aided Verification (CAV), Part II*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 3–11 (cit. on pp. 10, 11, 267, 285, 343, 356, 358, 451).

[5] Jens Katelaan, Christoph Matheja, and Florian Zuleger. "Effective Entailment Checking for Separation Logic with Inductive Definitions". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II*. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 319–336 (cit. on pp. 11, 451).

[6] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms". In: *Journal of the ACM* 65.5 (2018), 30:1–30:68 (cit. on pp. 11, 137, 139, 144, 261).

[7] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times". In: *European Symposium on Programming (ESOP)*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213 (cit. on pp. 11, 53).

[8] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "A Program Analysis Perspective on Expected Sampling Times". In: *Extended Abstracts of the First International Conference on Probabilistic Programming (PROBPROG)*. 2018 (cit. on p. 12).

[9] Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "HARRSH: A Tool for Unified Reasoning about Symbolic-Heap Separation Logic". In: *International Workshop on the Implementation of Logics (IWIL)*. Vol. 9. LPAR-22 Workshop and Short Paper Proceedings. 2018, pp. 23–36 (cit. on pp. 12, 289, 451).

[10] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs". In: *European Symposium on Programming (ESOP)*. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 364–389. (best paper award) (cit. on pp. 12, 137, 139, 144, 145, 261).

[11] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Analyzing Expected Runtimes by Program Verification". In: *Book on Probabilistic Programming resulting from the First School on Foundations of Programming and Software systems (FOPPS 2017)*. Ed. by Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. [to appear] (cit. on p. 12).

[12] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "On the Hardness of Analyzing Probabilistic Programs". In: *Acta Informatica* 56.3 (2019), pp. 255–285 (cit. on p. 12).

[13] Mihaela Sighireanu et al. "SL-COMP: Competition of Solvers for Separation Logic". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part III*. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 116–132 (cit. on pp. 12, 320).

[14] Maurice van Keulen, Benjamin Lucien Kaminski, Christoph Matheja, and Joost-Pieter Katoen. "Rule-Based Conditioning of Probabilistic Data". In: *Scalable Uncertainty Management (SUM)*. Vol. 11142. Lecture Notes in Computer Science. Springer, 2018, pp. 290–305. (best paper award) (cit. on p. 12).

[15] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Reasoning about Recursive Probabilistic Programs". In: *Symposium on Logic in Computer Science (LICS)*. ACM, 2016, pp. 672–681 (cit. on pp. 12, 139, 144).

[16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Inferring Covariances for Probabilistic Programs". In: *Quantitative Evaluation of Systems (QEST)*. Vol. 9826. Lecture Notes in Computer Science. Springer, 2016, pp. 191–206 (cit. on pp. 13, 139).

[17] Christoph Matheja, Christina Jansen, and Thomas Noll. "Tree-Like Grammars and Separation Logic". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Vol. 9458. Lecture Notes in Computer Science. Springer, 2015, pp. 90–108 (cit. on pp. 13, 284, 310, 323).

[18] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. "Quantitative Separation Logic". In: *CoRR* abs/1802.10467 (2018) (cit. on pp. 137, 165, 197, 216, 217, 231, 233, 235, 238, 239, 241, 242, 245, 246, 250, 251).

[19] Christina Jansen, Jens Katelaan, Christoph Matheja, Thomas Noll, and Florian Zuleger. "Unified Reasoning about Robustness Properties of Symbolic-Heap Separation Logic". In: *CoRR* abs/1610.07041 (2016) (cit. on pp. 284, 289, 299, 301, 304, 308–312, 314–318).

[20] Hannah Arndt, Christina Jansen, Christoph Matheja, and Thomas Noll. "Heap Abstraction Beyond Context-Freeness". In: *CoRR* abs/1705.03754 (2017) (cit. on pp. 323, 332, 339, 340, 346, 357).

[Aac18] Aachener Zeitung. *Digitales Management-System an der RWTH Aachen: Neue Plattform startet mit Ladehemmungen*. 2018. URL: https://www.aachener-zeitung.de/lokales/aachen/zu-viele-zugriffe-bei-rwthonline-sorgen-fuer-probleme_aid-33555785 (visited on 08/27/2019) (cit. on p. 1).

[ARE86] IJsbrand Jan Aalbersberg, Grzegorz Rozenberg, and Andrzej Ehrenfeucht. "On the membership problem for regular DNLC grammars". In: *Discrete Applied Mathematics* 13.1 (1986), pp. 79–85 (cit. on pp. 329, 445, 446).

[Abd+16] Parosh Aziz Abdulla, Fabio Gadducci, Barbara König, and Viktor Vafeiadis. "Verification of Evolving Graph Structures (Dagstuhl Seminar 15451)". In: *Dagstuhl Reports* 5.11 (2016). Ed. by Parosh Aziz Abdulla, Fabio Gadducci, Barbara König, and Viktor Vafeiadis, pp. 1–28. ISSN: 2192-5283 (cit. on p. 354).

[AY01] Rajeev Alur and Mihalis Yannakakis. "Model checking of hierarchical state machines". In: *ACM Transactions on Programming Languages and Systems* 23.3 (2001), pp. 273–303 (cit. on p. 352).

[Ant+14] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. "Foundations for Decision Problems in Separation Logic with General Inductive Predicates". In: *Foundations of Software Science and Computation Structure (FoSSaCS)*. Vol. 8412. Lecture Notes in Computer Science. Springer, 2014, pp. 411–425 (cit. on pp. 118, 130, 273, 283, 284, 323, 326, 333).

[Apt81] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey - Part 1". In: *ACM Transactions on Programming Languages and Systems* 3.4 (1981), pp. 431–483 (cit. on pp. 54, 56, 70, 106).

[AP86] Krzysztof R. Apt and Gordon D. Plotkin. "Countable nondeterminism and random assignment". In: *Journal of the ACM* 33.4 (1986), pp. 724–767 (cit. on pp. 125, 144).

[Arn19] Hannah Arndt. "Randomized Meldable Heaps: A more formal proof of a less simple probabilistic data structure". MA thesis. Germany: RWTH Aachen University, 2019 (cit. on pp. 252, 452).

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005 (cit. on p. 90).

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009 (cit. on p. 270).

[Atk11] Robert Atkey. "Amortised Resource Analysis with Separation Logic". In: *Logical Methods in Computer Science* 7.2 (2011) (cit. on pp. 145, 237).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cit. on pp. 18, 20, 21, 25, 26, 28, 30–33, 146, 154, 265, 267, 343, 352).

[BPS64] Yehoshua Bar-Hillel, Micha Perles, and Eliahu Shamir. *On formal properties of simple phrase-structure grammars, Z. Phonetik, Sprachwiss. Kommunikationsforsch. 14 (1961) 143–172. Reprinted as Chapter 9 in Y. Bar-Hillel: Language and Information*. 1964 (cit. on pp. 284, 337, 449).

[BGB12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Probabilistic Relational Hoare Logics for Computer-Aided Security Proofs". In: *Mathematics of Program Construction (MPC)*. Vol. 7342. Lecture Notes in Computer Science. Springer, 2012, pp. 1–6 (cit. on p. 145).

[BHL19] Gilles Barthe, Justin Hsu, and Kevin Liao. "A Probabilistic Separation Logic". In: *CoRR* abs/1907.10708 (2019) (cit. on p. 4).

[BBC17] BBC. *'Network issue' causes global airport delays*. 2017. URL: https://www.bbc.com/news/technology-41429376 (visited on 08/27/2019) (cit. on p. 1).

[Bel57] Richard Bellman. "A Markovian decision process". In: *Journal of Mathematics and Mechanics* (1957), pp. 679–684 (cit. on pp. 158, 218).

[Ber+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. "Shape Analysis for Composite Data Structures". In: *Computer Aided Verification (CAV)*. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 178–192 (cit. on pp. 142, 283, 286).

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "A Decidable Fragment of Separation Logic". In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Vol. 3328. Lecture Notes in Computer Science. Springer, 2004, pp. 97–109 (cit. on pp. 130, 283, 289, 313).

[BCO05a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Formal Methods for Components and Objects (FMCO)*. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 115–137 (cit. on pp. 89, 271, 286).

[BCO05b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Symbolic Execution with Separation Logic". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Vol. 3780. Lecture Notes in Computer Science. Springer, 2005, pp. 52–68 (cit. on pp. 271, 275, 283, 286, 323, 327).

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory Safety for Systems-Level Code". In: *Computer Aided Verification (CAV)*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 178–183 (cit. on pp. 89, 286).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004 (cit. on p. 2).

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. "Efficient On-the-Fly Model Checking for CTL*". In: *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 1995, pp. 388–397 (cit. on p. 352).

[Bij89] A. Bijlsma. "Calculating with Pointers". In: *Science of Computer Programming* 12.3 (1989), pp. 191–205 (cit. on p. 103).

[Bin+19] Eli Bingham et al. "Pyro: Deep universal probabilistic programming". In: *The Journal of Machine Learning Research* 20.1 (2019), pp. 973–978 (cit. on pp. 4, 141, 142).

[Bir40] Garrett Birkhoff. *Lattice theory*. Vol. 25. American Mathematical Society, 1940 (cit. on pp. 365, 366).

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007 (cit. on p. 140).

[BG03] Andreas Blass and Yuri Gurevich. "Algorithms: A Quest for Absolute Definitions". In: *Bulletin of the EATCS* 81 (2003), pp. 195–225 (cit. on p. 87).

[Bog+07] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. "Revamping TVLA: Making Parametric Shape Analysis Competitive". In: *Computer Aided Verification (CAV)*. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 221–225 (cit. on pp. 287, 353, 354).

[Bor00] Richard Bornat. "Proving Pointer Programs in Hoare Logic". In: *Mathematics of Program Construction (MPC)*. Vol. 1837. Lecture Notes in Computer Science. Springer, 2000, pp. 102–126 (cit. on pp. 103, 231, 237).

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. "Permission accounting in separation logic". In: *Principles of Programming Languages (POPL)*. Ed. by Jens Palsberg and Martin Abadi. ACM, 2005, pp. 259–270 (cit. on p. 146).

[Bou+06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. "Abstract Regular Tree Model Checking of Complex Dynamic Data Structures". In: *Static Analysis Symposium (SAS)*. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 52–70 (cit. on p. 354).

[Bou+12] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. "Abstract regular (tree) model checking". In: *International Journal on Software Tools for Technology Transfer (STTT)* 14.2 (2012), pp. 167–191 (cit. on p. 287).

[Boy03] John Boyland. "Checking Interference with Fractional Permissions". In: *Static Analysis Symposium (SAS)*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cit. on p. 146).

[BIP10] Marius Bozga, Radu Iosif, and Swann Perarnau. "Quantitative Separation Logic and Programs with Lists". In: *Journal of Automated Reasoning* 45.2 (2010), pp. 131–156 (cit. on pp. 145, 237).

[BB96] Gilles Brassard and Paul Bratley. *Fundamentals of algorithms*. Prentice Hall, 1996 (cit. on p. 148).

[Bro07] James Brotherston. "Formalised Inductive Reasoning in the Logic of Bunched Implications". In: *Static Analysis Symposium (SAS)*. Vol. 4634. Lecture Notes in Computer Science. Springer, 2007, pp. 87–103 (cit. on p. 118).

[BBC08] James Brotherston, Richard Bornat, and Cristiano Calcagno. "Cyclic proofs of program termination in separation logic". In: *Principles of Programming Languages (POPL)*. ACM, 2008, pp. 101–112 (cit. on p. 143).

[BDP11] James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. "Automated Cyclic Entailment Proofs in Separation Logic". In: *Conference on Automated Deduction (CADE)*. Vol. 6803. Lecture Notes in Computer Science. Springer, 2011, pp. 131–146 (cit. on pp. 118, 120, 130, 283, 284, 286, 323, 327, 360).

[Bro+14] James Brotherston, Carsten Fuhs, Juan A. Navarro Perez, and Nikos Gorogiannis. "A decision procedure for satisfiability in separation logic with inductive predicates". In: *Joint Meeting of Computer Science Logic and the Symposium on Logic in Computer Science (CSL-LICS)*. ACM, 2014, 25:1–25:10 (cit. on pp. 118–120, 273, 278, 283, 284, 289, 290, 309, 320).

[BG14] James Brotherston and Nikos Gorogiannis. "Cyclic Abduction of Inductively Defined Safety and Termination Preconditions". In: *Static Analysis Symposium (SAS)*. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 68–84 (cit. on pp. 284, 289, 320, 361).

[BGK17] James Brotherston, Nikos Gorogiannis, and Max I. Kanovich. "Biabduction (and Related Problems) in Array Separation Logic". In: *Conference on Automated Deduction (CADE)*. Vol. 10395. Lecture Notes in Computer Science. Springer, 2017, pp. 472–490 (cit. on p. 361).

[Bro+16] James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. "Model checking for symbolic-heap separation logic with inductive predicates". In: *Principles of Programming Languages (POPL)*. ACM, 2016, pp. 84–96 (cit. on pp. 120, 270, 284, 309).

[Bun15] Bundesministerium der Justiz und für Verbracherschutz. *Gesetz über die Bereitstellung von Produkten auf dem Markt (Produktsicherheitsgesetz - ProdSG) § 34 Ermächtigung zum Erlass von Rechtsverordnungen*. 2015. URL: https://www.gesetze-im-internet.de/prodsg_2011/__34.html (visited on 08/27/2019) (cit. on p. 1).

[Bur72] Rodney M. Burstall. "Some techniques for proving correctness of programs which alter data structures". In: *Machine Intelligence* 7.23-50 (1972), p. 3 (cit. on pp. 3, 89, 103, 108).

[CD11] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NASA Formal Methods (NFM)*. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 459–465 (cit. on pp. 3, 89, 286).

[Cal+11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional Shape Analysis by Means of Bi-Abduction". In: *Journal of the ACM* 58.6 (2011), 26:1–26:66 (cit. on pp. 3, 89, 286, 361).

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. "Local Action and Abstract Separation Logic". In: *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2007, pp. 366–378 (cit. on pp. 93, 262).

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. "Computability and Complexity Results for a Spatial Assertion Language for Data Structures". In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Vol. 2245. Lecture Notes in Computer Science. Springer, 2001, pp. 108–119 (cit. on p. 271).

[CMR16] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. "Verifying quantitative reliability for programs that execute on unreliable hardware". In: *Communications of the ACM* 59.8 (2016), pp. 83–91 (cit. on p. 141).

[Car+17] Bob Carpenter et al. "Stan: A probabilistic programming language". In: *Journal of Statistical Software* 76.1 (2017) (cit. on pp. 4, 140).

[CF17] Krishnendu Chatterjee and Hongfei Fu. "Termination of Nondeterministic Recursive Probabilistic Programs". In: *CoRR* abs/1701.02944 (2017) (cit. on pp. 139, 209).

[CFM17] Krishnendu Chatterjee, Hongfei Fu, and Aniket Murhekar. "Automated Recurrence Analysis for Almost-Linear Expected-Runtime Bounds". In: *Computer Aided Verification (CAV), Part I*. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 118–139 (cit. on p. 144).

[Cha+18] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. "Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs". In: *ACM Transactions on Programming Languages and Systems* 40.2 (2018), 7:1–7:45 (cit. on p. 139).

[CNZ17] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. "Stochastic invariants for probabilistic termination". In: *Principles of Programming Languages (POPL)*. ACM, 2017, pp. 145–160 (cit. on p. 139).

[Cha19] Sally Chau. "Comparing Hierarchical and On-The-Fly Model Checking for Java Pointer Programs". MA thesis. Germany: RWTH Aachen University, 2019 (cit. on pp. 352, 451).

[CDG11] Wei-Ngan Chin, Cristina David, and Cristian Gherghina. "A HIP and SLEEK verification system". In: *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2011, pp. 9–10 (cit. on pp. 320, 353).

[Chi+12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. "Automated verification of shape, size and bag properties via user-defined predicates in separation logic". In: *Science of Computer Programming* 77.9 (2012), pp. 1006–1036 (cit. on pp. 89, 130, 145, 286, 289, 323, 327, 353).

[CGH94] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. "Another Look at LTL Model Checking". In: *Computer Aided Verification (CAV)*. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 415–427 (cit. on p. 285).

[CGH97] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. "Another Look at LTL Model Checking". In: *Formal Methods in System Design* 10.1 (1997), pp. 47–71 (cit. on p. 285).

[Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification (CAV)*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169 (cit. on p. 33).

[Cla+18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*. Springer, 2018 (cit. on p. 33).

[Cla83] Kenneth L. Clarkson. "Fast Algorithms for the All Nearest Neighbors Problem". In: *Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1983, pp. 226–232 (cit. on p. 140).

[Com+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007. 2007 (cit. on pp. 291, 292, 296–298).

[CFN12] Anthony C. Constantinou, Norman E. Fenton, and Martin Neil. "pi–football: A Bayesian Network Model for Forecasting Association Football Match Outcomes". In: *Knowledge Based Systems* 36 (2012), pp. 322–339 (cit. on p. 140).

[Coo+11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. "Tractable Reasoning in a Fragment of Separation Logic". In: *International Conference on Concurrency Theory (CONCUR)*. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 235–249 (cit. on p. 283).

[Coo90] Gregory F. Cooper. "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks". In: *Artificial Intelligence* 42.2-3 (1990), pp. 393–405 (cit. on p. 141).

[Coq19] The Coq Development Team. *The Coq Reference Manual, version 8.9.0*. Available electronically at `https://coq.inria.fr/distrib/current/refman/index.html`. Aug. 2019 (cit. on p. 2).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009 (cit. on pp. 137, 148, 231, 246).

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Vol. 138. Encyclopedia of mathematics and its applications. Cambridge University Press, 2012 (cit. on p. 284).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysisof Programs by Construction or Approximation of Fixpoints". In: *Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252 (cit. on pp. 18, 28).

[CC79] Patrick Cousot and Radhia Cousot. "Constructive versions of Tarski's fixed point theorems". In: *Pacific journal of Mathematics* 82.1 (1979), pp. 43–57 (cit. on p. 368).

[CC92] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *Journal of Logic and Computation* 2.4 (1992), pp. 511–547 (cit. on pp. 18, 28).

[CC14] Patrick Cousot and Radhia Cousot. "Abstract interpretation: past, present and future". In: *Joint Meeting of Computer Science Logic and the Symposium on Logic in Computer Science (CSL-LICS)*. ACM, 2014, 2:1–2:10 (cit. on pp. 18, 28, 33).

[CGR18] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. "Program Analysis Is Harder Than Verification: A Computability Perspective". In: *Computer Aided Verification (CAV), Part II*. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 75–95 (cit. on pp. 33, 265).

[DL93] Paul Dagum and Michael Luby. "Approximating Probabilistic Inference in Bayesian Belief Networks is NP–Hard". In: *Artifical Intelligence* 60.1 (1993), pp. 141–153 (cit. on p. 141).

[Dar09] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009 (cit. on p. 140).

[DD15] Stéphane Demri and Morgan Deters. "Two-Variable Separation Logic and Its Inner Circle". In: *ACM Transactions on Programming Languages and Systems* 16.2 (2015), 15:1–15:36 (cit. on p. 271).

[DD16] Stéphane Demri and Morgan Deters. "Expressive Completeness of Separation Logic with Two Variables and No Separating Conjunction". In: *ACM Transactions on Programming Languages and Systems* 17.2 (2016), 12:1–12:44 (cit. on p. 271).

[Dem+17] Stéphane Demri, Didier Galmiche, Dominique Larchey-Wendling, and Daniel Méry. "Separation Logic with One Quantified Variable". In: *Theory of Computing Systems* 61.2 (2017), pp. 371–461 (cit. on p. 271).

[DLL17] Stéphane Demri, Etienne Lozes, and Denis Lugiez. "On Symbolic Heaps Modulo Permission Theories". In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Ed. by Satya V. Lokam and R. Ramanujam. Vol. 93. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:14. DOI: 10.4230/LIPIcs.FSTTCS.2017.25 (cit. on p. 146).

[DLM18] Stéphane Demri, Étienne Lozes, and Alessio Mansutti. "The Effects of Adding Reachability Predicates in Propositional Separation Logic". In: *Foundations of Software Science and Computation Structure (FoSSaCS)*. Vol. 10803. Lecture Notes in Computer Science. Springer, 2018, pp. 476–493 (cit. on p. 271).

[Dij72] Edsger W. Dijkstra. "The Humble Programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866 (cit. on pp. 1, 17).

[Dij75] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Communications of the ACM* 18.8 (1975), pp. 453–457 (cit. on pp. 39, 41, 49, 53).

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: http://www.worldcat.org/oclc/01958445 (cit. on pp. 4, 5, 18, 41, 44, 49, 53, 167).

[DKR04] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. "Who is Pointing When to Whom?" In: *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Vol. 3328. Lecture Notes in Computer Science. Springer, 2004, pp. 250–262 (cit. on p. 283).

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "A Local Shape Analysis Based on Separation Logic". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Vol. 3920. Lecture Notes in Computer Science. Springer, 2006, pp. 287–302 (cit. on p. 286).

[DP08] Dino Distefano and Matthew J. Parkinson. "jStar: towards practical verification for java". In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2008, pp. 213–226 (cit. on pp. 89, 286).

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. "A Fresh Look at Separation Algebras and Share Accounting". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Ed. by Zhenjiang Hu. Vol. 5904. Lecture Notes in Computer Science. Springer, 2009, pp. 161–177. DOI: 10.1007/978-3-642-10672-9_13 (cit. on p. 146).

[Dod08a] Mike Dodds. "From Separation Logic to Hyperedge Replacement and Back". In: *International Conference on Graph Transformation (ICGT)*. Vol. 5214. Lecture Notes in Computer Science. Springer, 2008, pp. 484–486 (cit. on p. 286).

[Dod08b] Mike Dodds. "Graph transformation and pointer structures". PhD thesis. University of York, UK, 2008 (cit. on p. 286).

[Dor+19] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. "Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs". In: *IEEE Transactions on Software Engineering* 45.3 (2019), pp. 219–236 (cit. on p. 141).

[Dri+88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation". In: *Communications of the ACM* 31.11 (1988), pp. 1343–1354 (cit. on p. 252).

[DPV11] Kamil Dudka, Petr Peringer, and Tomás Vojnar. "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic". In: *Computer Aided Verification (CAV)*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 372–378 (cit. on p. 89).

[EHN18] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. "Verified Analysis of Random Binary Tree Structures". In: *Interactive Theorem Proving (ITP)*. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 196–214 (cit. on p. 144).

[EHN15] Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. "A Verified Compiler for Probability Density Functions". In: *European Symposium on Programming (ESOP)*. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 80–104 (cit. on p. 144).

[EIP19] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. "The Bernays-Schönfinkel-Ramsey Class of Separation Logic on Arbitrary Domains". In: *Foundations of Software Science and Computation Structure (FoSSaCS)*. Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 242–259 (cit. on p. 271).

[Eco18] The Economist. *Why Uber's self-driving car killed a pedestrian*. 2018. URL: https://www.economist.com/the-economist-explains/2018/05/29/why-ubers-self-driving-car-killed-a-pedestrian (visited on 08/27/2019) (cit. on p. 1).

[EC80] E. Allen Emerson and Edmund M. Clarke. "Characterizing Correctness Properties of Parallel Programs Using Fixpoints". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 85. Lecture Notes in Computer Science. Springer, 1980, pp. 169–181 (cit. on p. 33).

[Ene+17a] Constantin Enea, Ondrej Lengal, Mihaela Sighireanu, and Tomas Vojnar. "SPEN: A Solver for Separation Logic". In: *NASA Formal Methods (NFM)*. Vol. 10227. Lecture Notes in Computer Science. 2017, pp. 302–309 (cit. on p. 130).

[Ene+17b] Constantin Enea, Ondrej Lengál, Mihaela Sighireanu, and Tomás Vojnar. "Compositional entailment checking for a fragment of separation logic". In: *Formal Methods in System Design* 51.3 (2017), pp. 575–607 (cit. on p. 286).

[Flo67] Robert W. Floyd. "Assigning meanings to programs". In: *Mathematical Aspects of Computer Science* 19.19-32 (1967), p. 1 (cit. on pp. 4, 18, 34, 38, 55, 70, 103, 167).

[FH79] Steven Fortune and John E. Hopcroft. "A Note on Rabin's Nearest-Neighbor Algorithm". In: *Information Processing Letters* 8.1 (1979), pp. 20–23 (cit. on p. 140).

[Fre79] Rūsiņš Mārtiņš Freivalds. "Fast Probabilistic Algorithms". In: *Mathematical Foundations of Computer Science (MFCS)*. Vol. 74. LNCS. Springer, 1979, pp. 57–69 (cit. on p. 140).

[FP10] Roman Fric and Martin Papco. "A Categorical Approach to Probability Theory". In: *Studia Logica* 94.2 (2010), pp. 215–230 (cit. on p. 144).

[Fro+17] Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder. "Lower Bounds for Runtime Complexity of Term Rewriting". In: *Journal of Automated Reasoning* 59.1 (2017), pp. 121–163 (cit. on p. 209).

[GM98] Anna Gambin and Adam Malinowski. "Randomized Meldable Priority Queues". In: *Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*. Vol. 1521. Lecture Notes in Computer Science. Springer, 1998, pp. 344–349 (cit. on pp. 8, 231, 252, 435).

[GJ75] Michael R. Garey and David S. Johnson. "Complexity Results for Multiprocessor Scheduling under Resource Constraints". In: *SIAM Journal on Computing* 4.4 (1975), pp. 397–411 (cit. on pp. 329, 446).

[Gha+12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. "Modelling and analysis using GROOVE". In: *International Journal on Software Tools for Technology Transfer (STTT)* 14.1 (2012), pp. 15–40 (cit. on pp. 287, 353).

[GS14] Noah D Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org. Accessed: 2019-4-18. 2014 (cit. on pp. 4, 140).

[Gor+14a] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio V. Russo, Johannes Borgström, and John Guiver. "Tabular: a schema-driven probabilistic programming language". In: *Principles of Programming Languages (POPL)*. ACM, 2014, pp. 321–334 (cit. on p. 140).

[Gor+14b] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. "Probabilistic programming". In: *Future of Software Engineering (FOSE)*. ACM, 2014, pp. 167–181 (cit. on pp. 4, 137, 141).

[GKO11] Nikos Gorogiannis, Max I. Kanovich, and Peter W. O'Hearn. "The Complexity of Abduction for Separated Heap Abstractions". In: *Static Analysis Symposium (SAS)*. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 25–42 (cit. on p. 361).

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics - a Foundation for Computer Science (2. ed.)* Addison-Wesley, 1994 (cit. on pp. 154, 159, 169).

[Gre68] Sheila A. Greibach. "A Note on Undecidable Properties of Formal Languages". In: *Mathematical Systems Theory* 2.1 (1968), pp. 1–6 (cit. on pp. 337, 449).

[Gre16] Friedrich Gretz. "Semantics and loop invariant synthesis for probabilistic programs". PhD thesis. RWTH Aachen University, Germany, 2016 (cit. on p. 144).

[GKM14] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. "Operational versus weakest pre-expectation semantics for the probabilistic guarded command language". In: *Performance Evaluation* 73 (2014), pp. 110–132 (cit. on p. 144).

[Gru05] Orna Grumberg. "Abstraction and Refinement in Model Checking". In: *Formal Methods for Components and Objects (FMCO)*. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 219–242 (cit. on p. 33).

[GCW16] Xincai Gu, Taolue Chen, and Zhilin Wu. "A Complete Decision Procedure for Linearly Compositional Separation Logic with Data Constraints". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 532–549 (cit. on pp. 285, 286).

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Vol. 643. Lecture Notes in Computer Science. Springer, 1992 (cit. on p. 286).

[Hab+11] Peter Habermehl, Lukas Holik, Adam Rogalewicz, Jiri Simacek, and Tomas Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Computer Aided Verification (CAV)*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 424–440 (cit. on p. 289).

[Hab+12] Peter Habermehl, Lukas Holik, Adam Rogalewicz, Jiri Simacek, and Tomas Vojnar. "Forest automata for verification of heap manipulation". In: *Formal Methods in System Design* 41.1 (2012), pp. 83–106 (cit. on p. 289).

[Har+19] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. "Aiming Low Is Harder - Inductive Proof Rules for Lower Bounds on Weakest Preexpectations in Probabilistic Program Verification". In: *CoRR* abs/1904.01117 (2019) (cit. on p. 209).

[Har02] Jeremy Ian den Hartog. "Probabilistic Extensions of Semantical Models". Dissertation. Amsterdam: Vrije Universiteit, 2002 (cit. on p. 197).

[HV02] Jerry den Hartog and Erik P. de Vink. "Verifying Probabilistic Programs Using a Hoare Like Logic". In: *International Journal of Foundations of Computer Science* 13.3 (2002), pp. 315–340 (cit. on pp. 138, 197).

[Hec08] David Heckerman. "A Tutorial on Learning with Bayesian Networks". In: *Innovations in Bayesian Networks*. Vol. 156. Studies in Computational Intelligence. Springer, 2008, pp. 33–82 (cit. on p. 140).

[Hei15] Jonathan Heinen. "Verifying Java Programs - A Graph Grammar Approach". PhD thesis. RWTH Aachen University, 2015 (cit. on pp. 285, 338).

[Hei+15] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. "Juggrnaut: using graph grammars for abstracting unbounded heap structures". In: *Formal Methods in System Design* 47.2 (2015), pp. 159–203 (cit. on pp. 285, 287, 343, 346, 350, 353).

[HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. "Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures". In: *Electronic Notes in Theoretical Computer Science* 266 (2010), pp. 93–107 (cit. on pp. 285, 353).

[Hen13] Thomas A. Henzinger. "Quantitative reactive modeling and verification". In: *Computer Science - Research and Development* 28.4 (2013), pp. 331–344 (cit. on p. 141).

[Hes93] Wim H. Hesselink. "Proof Rules for Recursive Procedures". In: *Formal Aspects of Computing* 5.6 (1993), pp. 554–570 (cit. on p. 70).

[Hoa62] Charles Antony Richard Hoare. "Quicksort". In: *The Computer Journal* 5.1 (1962), pp. 10–15 (cit. on p. 140).

[Hoa69] Charles Antony Richard Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on pp. 4, 18, 34, 36, 38, 55, 56, 70, 103, 167).

[Hoa71] Charles Antony Richard Hoare. "Procedures and parameters: An axiomatic approach". In: *Symposium on Semantics of Algorithmic Languages*. Vol. 188. Lecture Notes in Mathematics. Springer, 1971, pp. 102–116 (cit. on pp. 70, 103).

[HW73] Charles Antony Richard Hoare and Niklaus Wirth. "An Axiomatic Definition of the Programming Language PASCAL". In: *Acta Informatica* 2 (1973), pp. 335–355 (cit. on p. 103).

[Hoa09] Tony Hoare. "Null references: The billion dollar mistake". In: *Presentation at QCon London* 298 (2009) (cit. on p. 88).

[HJ03] Martin Hofmann and Steffen Jost. "Static prediction of heap space usage for first-order functional programs". In: *Principles of Programming Languages (POPL)*. ACM, 2003, pp. 185–197 (cit. on p. 145).

[Hol+13] Lukas Holik, Ondrej Lengal, Adam Rogalewicz, Jiri Simacek, and Tomas Vojnar. "Fully Automated Shape Analysis Based on Forest Automata". In: *Computer Aided Verification (CAV)*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 740–755 (cit. on pp. 287, 353–355).

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007 (cit. on p. 326).

[Hri18] Ivaylo Hristakiev. "Confluence analysis for a graph programming language". PhD thesis. University of York, UK, 2018 (cit. on p. 338).

[IRS13] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. "The Tree Width of Separation Logic with Recursive Definitions". In: *Conference on Automated Deduction (CADE)*. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 21–38 (cit. on pp. 118–120, 130, 273, 283, 284, 289, 290, 295, 309, 310).

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomas Vojnar. "Deciding Entailments in Inductive Separation Logic with Tree Automata". In: *Automated Technology for Verification and Analysis (ATVA)*. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 201–218 (cit. on pp. 120, 130, 283, 284, 289, 295, 310).

[IS18] Radu Iosif and Cristina Serban. "An entailment checker for separation logic with inductive definitions". In: *18th International Workshop on Automated Verification of Critical Systems*. 2018 (cit. on pp. 284, 286).

[Ios+] Radu Iosif, Cristina Serban, Andrew Reynolds, and Mihaela Sighireanu. "Encoding Separation Logic in SMT-LIB v2. 5". In: (). URL: https://sl-comp.github.io/docs/smtlib-sl.pdf (cit. on p. 320).

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an Assertion Language for Mutable Data Structures". In: *Principles of Programming Languages (POPL)*. ACM, 2001, pp. 14–26 (cit. on pp. 3, 4, 6, 8, 89, 93, 94, 105, 108, 109, 111, 113, 114, 118, 121, 124, 128, 143, 165, 220).

[Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods (NFM)*. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55 (cit. on p. 289).

[Jan17] Christina Jansen. "Static Analysis of Pointer Programs - Linking Graph Grammars and Separation Logic". PhD thesis. RWTH Aachen University, Germany, 2017 (cit. on pp. 286, 338).

[JGN14] Christina Jansen, Florian Göbe, and Thomas Noll. "Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs". In: *International Conference on Graph Transformation (ICGT)*. Vol. 8571. Lecture Notes in Computer Science. Springer, 2014, pp. 65–80 (cit. on pp. 283, 285, 286, 338).

[JC10] Xia Jiang and Gregory F. Cooper. "A Bayesian Spatio–temporal Method for Disease Outbreak Detection". In: *Journal of the American Medical Informatics Association* 17.4 (2010), pp. 462–471 (cit. on p. 140).

[Jon90] Claire Jones. "Probabilistic non-determinism". PhD thesis. University of Edinburgh, UK, 1990 (cit. on pp. 139, 143, 199).

[Kam19] Benjamin Lucien Kaminski. "Advanced weakest precondition calculi for probabilistic programs". Dissertation. RWTH Aachen University, 2019, p. 363. DOI: 10.18154/RWTH-2019-01829. URL: http://publications.rwth-aachen.de/record/755408 (cit. on p. 139).

[KK17] Benjamin Lucien Kaminski and Joost-Pieter Katoen. "A weakest pre-expectation semantics for mixed-sign expectations". In: *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2017, pp. 1–12 (cit. on p. 168).

[Kar91] Richard M. Karp. "Probabilistic Recurrence Relations". In: *Symposium on the Theory of Computing (STOC)*. ACM, 1991, pp. 190–197 (cit. on p. 138).

[Kar94] Richard M. Karp. "Probabilistic Recurrence Relations". In: *Journal of the ACM* 41.6 (1994), pp. 1136–1150 (cit. on p. 138).

[KJW18] Jens Katelaan, Dejan Jovanovic, and Georg Weissenbacher. "A Separation Logic with Data: Small Models and Automation". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Vol. 10900. Lecture Notes in Computer Science. Springer, 2018, pp. 455–471 (cit. on p. 285).

[Kat+15] Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, and Federico Olmedo. "Understanding Probabilistic Programs". In: *Correct System Design*. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 15–32 (cit. on p. 137).

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988 (cit. on pp. 89, 100).

[Kle+52] Stephen Cole Kleene, NG de Bruijn, J de Groot, and Adriaan Cornelis Zaanen. *Introduction to metamathematics*. Vol. 483. van Nostrand New York, 1952 (cit. on pp. 47, 368).

[Kna28] Bronisław Knaster. "Un theoreme sur les functions d'ensembles". In: *Annales de la Société Polonaise de Mathématique* 6 (1928), pp. 133–134 (cit. on p. 367).

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997 (cit. on pp. 59, 87).

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming, Volume III, 2nd Edition*. Addison-Wesley, 1998 (cit. on p. 252).

[KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009 (cit. on p. 140).

[Kol53] Andrei N. Kolmogorov. "On the concept of algorithm". In: *Uspekhi Matematicheskikh Nauk* 8.4 (1953), pp. 175–176 (cit. on p. 87).

[Koz81] Dexter Kozen. "Semantics of Probabilistic Programs". In: *Journal of Computer and System Sciences* 22.3 (1981), pp. 328–350 (cit. on pp. 5, 138).

[Koz83] Dexter Kozen. "A Probabilistic PDL". In: *Symposium on the Theory of Computing (STOC)*. ACM, 1983, pp. 291–297 (cit. on pp. 5, 8, 139, 142, 167, 198).

[Koz85] Dexter Kozen. "A Probabilistic PDL". In: *Journal of Computer and System Sciences* 30.2 (1985), pp. 162–178 (cit. on pp. 139, 143).

[Kre+17] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic". In: *European Symposium on Programming (ESOP)*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 696–723 (cit. on pp. 3, 39, 143).

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *Principles of Programming Languages (POPL)*. ACM, 2017, pp. 205–217 (cit. on pp. 3, 231, 237).

[Kui97] Werner Kuich. "Semirings and Formal Power Series: Their Relevance to Formal Languages and Automata". In: *Handbook of Formal Languages (1)*. 1997, pp. 609–677 (cit. on p. 170).

[Law67] Harold W. Lawson. "PL/I list processing". In: *Communications of the ACM* 10.6 (1967), pp. 358–367 (cit. on pp. 87, 89, 103).

[LSC16] Quang Loc Le, Jun Sun, and Wei-Ngan Chin. "Satisfiability Modulo Heap-Based Programs". In: *Computer Aided Verification (CAV), Part I*. Vol. 9779. Lecture Notes in Computer Science. Springer, 2016, pp. 382–404 (cit. on p. 285).

[LSQ18] Quang Loc Le, Jun Sun, and Shengchao Qin. "Frame Inference for Inductive Entailment Proofs in Separation Logic". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I*. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 41–60 (cit. on pp. 285, 286).

[Le+17] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. "A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic". In: *Computer Aided Verification (CAV), Part II*. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 495–517 (cit. on pp. 130, 285).

[LYY05] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. "Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis". In: *European Symposium on Programming (ESOP)*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 124–140 (cit. on p. 285).

[LP14] Wonyeol Lee and Sungwoo Park. "A proof system for separation logic with magic wand". In: *Principles of Programming Languages (POPL)*. ACM, 2014, pp. 477–490 (cit. on p. 130).

[Leg19] Legal Tribune Online. *Probleme beim beA: BRAK empfiehlt bei eiligen Sachen "andere Versandmöglichkeiten"*. 2019. URL: https://www.lto.de/persistent/a_id/33663/ (visited on 08/27/2019) (cit. on p. 1).

[LT93] Nancy G. Leveson and Clark S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41 (cit. on p. 1).

[Lio96] Jacques-Louis Lions (on behalf of the inquiry board). *ARIANE 5 – Flight 501 Failure – Report by the Inquiry Board*. 1996. URL: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html (visited on 08/27/2019) (cit. on p. 1).

[LRS06] Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. "Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm". In: *Static Analysis Symposium (SAS)*. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 261–279 (cit. on pp. 287, 354).

[LS79] David C. Luckham and Norihisa Suzuki. "Verification of Array, Record, and Pointer Operations in Pascal". In: *ACM Transactions on Programming Languages and Systems* 1.2 (1979), pp. 226–244 (cit. on pp. 103, 108).

[MPQ11] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. "Decidable logics combining heap structures and data". In: *Principles of Programming Languages (POPL)*. ACM, 2011, pp. 611–622 (cit. on p. 283).

[Mag+06] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. "Inferring invariants in separation logic for imperative list-processing programs". In: *SPACE Workshop* 3 (2006), pp. 5–7 (cit. on p. 237).

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005 (cit. on pp. 5, 8, 139, 143, 157, 166, 167, 169, 199, 201, 209, 220, 222, 223).

[MT90] Kurt Mehlhorn and Athanasios K. Tsakalidis. "Data Structures". In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. 1990, pp. 301–342 (cit. on p. 252).

[Mil71] Robin Milner. "An Algebraic Definition of Simulation Between Programs". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. William Kaufmann, 1971, pp. 481–489 (cit. on pp. 18, 28, 32).

[MW19] Tom Minka and John Winn. *Infer.NET*. Accessed online April 17. 2019. URL: http://infernet.azurewebsites.net/ (cit. on pp. 141, 142).

[MW08] Tom Minka and John M. Winn. "Gates". In: *Neural Information Processing Systems (NIPS)*. Curran Associates, 2008, pp. 1073–1080 (cit. on p. 141).

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005 (cit. on p. 4).

[Mon01] David Monniaux. "Analysis of probabilistic programs by abstract interpretation. (Analyse de programmes probabilistes par interprétation abstraite)". PhD thesis. Paris Dauphine University, France, 2001 (cit. on p. 138).

[MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. "Probabilistic Predicate Transformers". In: *ACM Transactions on Programming Languages and Systems* 18.3 (1996), pp. 325–353 (cit. on pp. 5, 8, 139, 143, 167).

[MJ84] F Lockwood Morris and Clifford B Jones. "An early program proof by Alan Turing". In: *Annals of the History of Computing* 6.2 (1984), pp. 139–143 (cit. on p. 34).

[MR97] Rajeev Motwani and Prabhakar Raghavan. "Randomized Algorithms". In: *The Computer Science and Engineering Handbook*. CRC Press, 1997, pp. 141–161 (cit. on pp. 4, 137, 140, 142).

[NJ10] Richard E Neapolitan and Xia Jiang. *Probabilistic Methods for Financial and Marketing Informatics*. Morgan Kaufmann, 2010 (cit. on p. 140).

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. "Bounded expectations: resource analysis for probabilistic programs". In: *Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 496–512 (cit. on p. 139).

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. URL: https://doi.org/10.1007/978-3-662-03811-6 (cit. on pp. 17, 28).

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992. ISBN: 978-0-471-92980-2 (cit. on pp. 18, 20–23, 27, 35, 36, 38–40, 42, 43, 45, 47, 51, 56).

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007. ISBN: 978-1-84628-691-9. DOI: 10.1007/978-1-84628-692-6. URL: https://doi.org/10.1007/978-1-84628-692-6 (cit. on p. 35).

[Nip02] Tobias Nipkow. "Hoare Logics for Recursive Procedures and Unbounded Nondeterminism". In: *Computer Science Logic (CSL)*. Vol. 2471. Lecture Notes in Computer Science. Springer, 2002, pp. 103–119 (cit. on pp. 55, 70).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002 (cit. on p. 2).

[OHe12] Peter W. O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: *Software Safety and Security* 33 (2012), pp. 286–318 (cit. on pp. 129, 231).

[OHe19] Peter W. O'Hearn. "Separation logic". In: *Communications of the ACM* 62.2 (2019), pp. 86–95 (cit. on pp. 3, 89, 143).

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic (CSL)*. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19 (cit. on pp. 4, 89, 105, 108, 271).

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. "Separation and information hiding". In: *Principles of Programming Languages (POPL)*. ACM, 2004, pp. 268–280 (cit. on p. 185).

[OYR09] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. "Separation and information hiding". In: *ACM Transactions on Programming Languages and Systems* 31.3 (2009), 11:1–11:50 (cit. on p. 185).

[Old83] Ernst-Rüdiger Olderog. "On the Notion of Expressiveness and the Rule of Adaption". In: *Theoretical Computer Science* 24 (1983), pp. 337–347 (cit. on pp. 70, 106).

[Pap07] Christos H. Papadimitriou. *Computational complexity*. Academic Internet Publ., 2007 (cit. on p. 270).

[Par69] David Park. "Fixpoint induction and proofs of program properties". In: *Machine Intelligence* 5 (1969) (cit. on p. 368).

[Pea85] Judea Pearl. "Bayesian Networks: A Model of Self–activated Memory for Evidential Reasoning". In: *Conference of the Cognitive Science Society*. 1985, pp. 329–334 (cit. on p. 140).

[PQM14] Edgar Pek, Xiaokang Qiu, and Parthasarathy Madhusudan. "Natural proofs for data structure manipulation in C using separation logic". In: *Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 440–451 (cit. on p. 283).

[PR13] Juan Antonio Navarro Pérez and Andrey Rybalchenko. "Separation Logic Modulo Theories". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 90–106 (cit. on p. 286).

[Pfe09] Avi Pfeffer. "Figaro: An object-oriented probabilistic programming language". In: *Charles River Analytics Technical Report* 137 (2009), p. 96 (cit. on p. 140).

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating Separation Logic Using SMT". In: *Computer Aided Verification (CAV)*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 773–789 (cit. on p. 285).

[PWZ14] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating Separation Logic with Trees and Data". In: *Computer Aided Verification (CAV)*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 711–728 (cit. on p. 285).

[Plo81] Gordon Plotkin. "Structural operational semantics". In: *Aarhus University, Denmark* (1981) (cit. on pp. 20, 23).

[Plu10] Detlef Plump. "Checking Graph-Transformation Systems for Confluence". In: *Electronic Communication of the European Association of Software Science and Technology* 26 (2010) (cit. on p. 338).

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1977, pp. 46–57 (cit. on pp. 32, 267, 343).

[Put05] Martin Lee Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2005 (cit. on pp. 146, 154, 156, 158, 373, 374).

[Qiu+13] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. "Natural proofs for structure, data, and separation". In: *Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 231–242 (cit. on p. 283).

[Rab76] Michael Oser Rabin. "Probabilistic Algorithms". In: *Algorithms and Complexity: New Directions and Recent Results*. Ed. by Joseph Frederick Traub. Academic Press, 1976, pp. 21–39 (cit. on pp. 4, 139).

[Reg07] The Register. *US Superfighter software glitch fixed*. 2007. URL: https://www.theregister.co.uk/2007/02/28/f22s_working_again/ (visited on 08/27/2019) (cit. on p. 1).

[RSW07] Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. "Shape Analysis and Applications". In: *The Compiler Design Handbook, 2nd ed.* CRC Press, 2007, p. 12 (cit. on p. 347).

[Rey81] John C. Reynolds. *The Craft of Programming*. Prentice Hall International series in computer science. Prentice Hall, 1981 (cit. on pp. 106, 215).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2002, pp. 55–74 (cit. on pp. 3, 4, 8, 89, 92, 94, 106–109, 111, 113–116, 118, 119, 121, 143, 165, 184, 188, 220, 231, 237, 246).

[Ric53] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on p. 265).

[RN08] Stefan Rieger and Thomas Noll. "Abstracting complex data structures by hyperedge replacement". In: *International Conference on Graph Transformation (ICGT)*. Springer. 2008, pp. 69–83 (cit. on pp. 285, 343).

[RP14] Brian E. Ruttenberg and Avi Pfeffer. "Decision-Making with Complex Data Structures using Probabilistic Programming". In: *CoRR* abs/1407.3208 (2014) (cit. on p. 142).

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In: *Principles of Program Analysis (POPL)*. ACM, 1999, pp. 105–118 (cit. on p. 287).

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Transactions on Programming Languages and Systems* 24.3 (2002), pp. 217–298 (cit. on p. 287).

[Sch19] Johannes Bernhard Schulte. "Automated Detection and Completion of Confluence for Graph Grammars". MA thesis. Germany: RWTH Aachen University, 2019 (cit. on pp. 338, 350, 451).

[Sco08] Dana Scott. "The Algebraic Intepretation of Quantifiers. Intuitionistic and Classical". In: *Andrzej Mostowski and Foundational Studies* (2008), pp. 289–312 (cit. on p. 172).

[SGG10] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts, 8th Edition*. Wiley, 2010. ISBN: 978-0-471-69466-3 (cit. on pp. 21, 87).

[ST86] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Heaps". In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69 (cit. on p. 252).

[Suz80] Norihisa Suzuki. "Analysis of Pointer Rotation". In: *Principles of Programming Languages (POPL)*. ACM Press, 1980, pp. 1–11 (cit. on p. 103).

[Ta+16] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. "Automated Mutual Explicit Induction Proof in Separation Logic". In: *Formal Methods (FM)*. Vol. 9995. Lecture Notes in Computer Science. 2016, pp. 659–676 (cit. on pp. 130, 286, 320).

[Ta+18] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. "Automated lemma synthesis in symbolic-heap separation logic". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 9:1–9:29 (cit. on p. 130).

[Ta+19] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. "Automated mutual induction proof in separation logic". In: *Formal Aspects of Computing* 31.2 (2019), pp. 207–230 (cit. on p. 320).

[Tar+55] Alfred Tarski et al. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309 (cit. on p. 367).

[TH18] Joseph Tassarotti and Robert Harper. "Verified Tail Bounds for Randomized Programs". In: *Interactive Theorem Proving (ITP)*. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 560–578 (cit. on p. 138).

[TH19] Joseph Tassarotti and Robert Harper. "A separation logic for concurrent randomized programs". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 64:1–64:30 (cit. on pp. 4, 145).

[TC14] Makoto Tatsuta and Wei-Ngan Chin. "Completeness of Separation Logic with Inductive Definitions for Program Verification". In: *Software Engineering and Formal Methods (SEFM)*. Vol. 8702. Lecture Notes in Computer Science. Springer, 2014, pp. 20–34 (cit. on p. 268).

[TCA19] Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. "Completeness and expressiveness of pointer program verification by separation logic". In: *Information and Computation* 267 (2019), pp. 1–27 (cit. on p. 268).

[TK15] Makoto Tatsuta and Daisuke Kimura. "Separation Logic with Monadic Inductive Definitions and Implicit Existentials". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Vol. 9458. Lecture Notes in Computer Science. Springer, 2015, pp. 69–89 (cit. on pp. 283, 284, 310).

[Tra+17] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. "Deep probabilistic programming". In: *International Conference on Learning Representations*. 2017 (cit. on p. 141).

[Tra+16] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. "Edward: A library for probabilistic modeling, inference, and criticism". In: *arXiv preprint arXiv:1610.09787* (2016) (cit. on p. 141).

[Tra19] Gregory Travis. *How the Boeing 737 Max Disaster Looks to a Software Developer*. 2019. URL: https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer (visited on 08/27/2019) (cit. on p. 1).

[Tur49] Alan M. Turing. "Checking a large routine". In: *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, 1949, pp. 67–69 (cit. on pp. 4, 17, 33).

[Tur37] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.1 (1937), pp. 230–265 (cit. on pp. 2, 265, 270).

[Val+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a Java bytecode optimization framework". In: *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 1999, p. 13 (cit. on p. 350).

[VLC10] Jules Villard, Étienne Lozes, and Cristiano Calcagno. "Tracking Heaps That Hop with Heap-Hop". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 275–279 (cit. on p. 89).

[WRS02] Reinhard Wilhelm, Thomas W. Reps, and Shmuel Sagiv. "Shape Analysis and Applications". In: *The Compiler Design Handbook*. CRC Press, 2002, pp. 175–218 (cit. on p. 347).

[Wil12] Virginia Vassilevska Williams. "Multiplying Matrices faster than Coppersmith-Winograd". In: *Symposium on the Theory of Computing (STOC)*. Vol. 12. Citeseer. 2012, pp. 887–898 (cit. on p. 140).

[Win93] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. ISBN: 978-0-262-23169-5 (cit. on pp. 18–21, 23, 35, 36, 39, 40, 56, 365).

[WMM14] Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. "A New Approach to Probabilistic Programming Inference". In: *Artificial Intelligence and Statistics (AISTATS)*. Vol. 33. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 1024–1032 (cit. on p. 140).

[Yan01] Hongseok Yang. "Local Reasoning for Stateful Programs". PhD thesis. University of Illinois at Urbana-Champaign, 2001 (cit. on pp. 128, 186).

[Yan+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. "Scalable Shape Analysis for Systems Code". In: *Computer Aided Verification (CAV)*. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 385–398 (cit. on p. 89).

[YO02] Hongseok Yang and Peter W. O'Hearn. "A Semantic Basis for Local Reasoning". In: *Foundations of Software Science and Computation Structure (FoSSaCS)*. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 402–416 (cit. on pp. 89, 95, 105, 108, 125, 128, 221).

[ZG14] Damiano Zanardini and Samir Genaim. "Inference of Field-Sensitive Reachability and Cyclicity". In: *ACM Transactions on Programming Languages and Systems* 15.4 (2014), 33:1–33:41 (cit. on pp. 289, 318).

[ZKT85] Viktor N. Zemlyachenko, Nickolay M. Korneenko, and Regina I. Tyshkevich. "Graph Isomorphism Problem". In: *Journal of Soviet Mathematics* 29.4 (1985), pp. 1426–1481 (cit. on p. 342).

[ZR98] Geoffrey Zweig and Stuart J. Russell. "Speech Recognition with Dynamic Bayesian Networks". In: *Innovative Applications of Artificial Intelligence (IAAI)*. AAAI Press / The MIT Press, 1998, pp. 173–180 (cit. on p. 140).

$\_ \mapsto -$, *see* allocated pointer predicate
0, *see* oP^{4}L (action)
0, *see* constant expectation, *see* null pointer
$\_^*$, *see* Kleene star
$2^{\_}$, *see* powerset

a,b, *see* actions
$\mathcal{A}$,$\mathcal{B}$,$\mathcal{C}$, *see* heap automata, *see* tree automata
α,β,γ, *see* limit ordinal, successor ordinal
Act, *see* actions, set of
Act $(\_)$, *see* enabled actions, set of
AE, *see* arithmetic expressions, set of
$\_ :=$ `alloc`$(\_)$, *see* allocation
$\wedge$, *see* predicate conjunction
AP, *see* atomic propositions, set of
$\_ := \_$, *see* assignment

B, *see* Boolean expression
body $(\_)$, *see* procedure body
$\boxdot$, *see* scope
$[-\boxdot]$, *see* scope decrement
$[+\boxdot]$, *see* scope increment
BV$(\_)$, *see* bound variables, set of

C, *see* SID, class of
C, *see* program
χ, *see* execution fragment

D, *see* domain theory
$\triangleq$, *read as* "is defined as"
Δ, *see* transition rules of automaton
$\Diamond$, *see* context over a ranked alphabet
Dist $(\_)$, *see* probability distributions over a set, set of
dom$(_)$, *see* domain of function

$\mathbb{E}$, *see* expectations, set of
E, *see* arithmetic expression
ec$[_](_)$, *see* expectation calculus
emp $\wedge \_ = \_$, *see* normal form of symbolic heaps
emp $\wedge \_ \neq \_$, *see* normal form of symbolic heaps
emp, *see* empty-heap predicate
`enter`, *see* enter scope
$\mathbb{E}_{\leq 1}$, *see* one-bounded expectations, set of
EQ, *see* equalities component
Exec$[\_](\_)$, *see* executions, set of
$\exists$, *see* existential quantifier
ExpRew $(\_)$, *see* expected reward
ExpRew$[_]$ $(\_, \_)$, *see* expected reward of **P^{4}L** program

$\mathcal{F}$, *see* final states of automaton, set of
false, *see* truth value
$\langle$fault$\rangle$, *see* memory fault, state
F,G,H, *see* procedure names
Fold$_{\_}(\_)$, *see* folding set
$\forall$, *see* universal quantifier
`free`$(\_)$, *see* deallocation
FV$^{\leq \_}$, *see* SID with bounded free variables, set of

$s_{\mathcal{G}}$, *see* goal state
gfp $(\_)$, *see* greatest fixed point

$\mathfrak{h}$, *see* heap
Heaps, *see* heaps, set of

I, *see* initial states of operational semantics, set of
I, *see* expectations, invariant
`if` $(\_)$ $\{\_\}$ `else` $\{\_\}$, *see* conditional choice
impl $(\_ := \_(\_))$, *see* implementation of procedure call

P⁴LA, *see* probabilistic procedural pointer programming language with auxiliaries
p,q, *see* state of automaton
p,*q*, *see* probability
P,*Q*,*R*, *see* predicate
P,***Q***,***R***, *see* predicate symbols
Pred, *see* predicates, set of
$\mathsf{Prob}(\_,\_,\_)$, *see* transition probability function
Procs, *see* procedure names, set of
Ψ,Γ, *see* system of inductive definitions
PSym, *see* predicate symbols, set of
PT, *see* points-to component

$\mathcal{Q}$, *see* states of automaton, set of
$\mathbb{Q}$, *read as* "set of rational numbers"
$\mathbb{Q} \cap [0,1]$, *read as* "set of rational probabilities"
$\mathbb{Q} \cap (0,1]$, *read as* "set of positive rational probabilities"
QSL, *see* quantitative separation logic
$\mathtt{qsl}[\![\_]\!]$, *see* embedding of SL into QSL

$\mathbb{R}$, *read as* "set of real numbers"
RA, *see* ranked alphabet
$\mathtt{rank}\_$, *see* rank of a symbol
Reach$(\_)$, *see* reachable fragment
rew, *see* reward function
$\mathbb{R}_{\geq 0}$, *read as* "set of non-negative real numbers"
ρ, *see* higher-order invariant
⇒, *see* predicate, order
$\mathbb{R}^{\infty}_{\geq 0}$, *read as* "set of extended non-negative real numbers"

S, *see* states of transition system, set of
$\mathfrak{S}$, *see* scheduler
$\mathfrak{s}$, *see* stack
S_0, *see* initial states, set of
Sel, *see* selectors, sequence of
sel, *next*, *see* selector
⋆, *see* separating conjunction
−⋆, *see* separating implication
$(\mathfrak{s},\mathfrak{h})$, *see* stack-heap pairs
SH_, *see* symbolic heaps, set of
SHPairs, *see* stack-heap pairs, set of
SHPairs$(\_)$, *see* stack-heap pairs captured by predicate, set of
SHSL, *see* symbolic heaps, set of
s, *see* state
$\langle \mathsf{sink} \rangle$, *see* sink state
size, *see* heap size quantity
skip, *see* effectless program
SL, *see* separation logic
$S^{\#}$, *see* abstract states, set of
$S^{\#}_0$, *see* abstract initial states, set of
Stacks, *see* stacks, set of
Stacks$(\_)$, *see* stacks captured by predicate
States, *see* states of operational semantics, set of
sup, *read as* "supremum"

t, *see* tree over a ranked alphabet
term, *see* successful termination indicator
Terminated, *see* terminating executions, set of
θ, *see* higher-order predicate transformer
TS, *see* transition system
Trees$(\_)$, *see* trees, set of
Trees$_{\_}(\_)$, *see* unfolding trees, set of
true, *see* truth value
$\mathbb{T}$, *see* truth values, set of
$TS_{\_}$, *see* abstract transition system

Index