Deductive Reasoning about Concurrent Probabilistic Programs

Ira Fesefeldt

Deductive Reasoning about Concurrent Probabilistic Programs

Ira Fesefeldt

DEDUCTIVE REASONING ABOUT CONCURRENT PROBABILISTIC PROGRAMS

Von der Fakultät für Informatik der RWTH Aachen University zur Erlangung des akademischen Grades einer Doktorin der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Ira Fesefeldt, M.Sc.

aus

Aachen

Berichter: Prof. Dr. Joost-Pieter Katoen
Prof. Dr. Lars Birkedal

Tag der mündlichen Prüfung: 5. November 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Acknowledgments

I would like to thank my supervisors Joost-Pieter Katoen and Thomas Noll. Joost-Pieter allowed me to develop my own research interests, and thrive in a friendly work environment. He always had an open ear, when I required it. I am very grateful to have worked at his chair. Thomas provided me with assistance through the hardships of developing my research, through teaching, and through other obligations during my PhD. I am also grateful to Lars Birkedal for being my second examiner. My visit at his chair showed me his inspiring love for sharing knowledge.

I am also very grateful to Christoph Matheja, who supervised my master's thesis in 2019, and afterwards, through productive collaborations, guided me during my first steps in academia. I thank Emma Ahrens, Kevin Batz, Jana Berger, Alexander Bork, Christopher Brix, Christina Gehnen, Darion Haase, Roy Hermanns, Lutz Klinkenberg, Hannah Mertens, Joshua Moerman, Tim Quatmann, Bahare Salmani, Philipp Schroer, Jip Spel, Lena Verscht, Matthias Volk, Tobias Winkler, and all my other colleagues that worked with or next to me during my PhD, and who either provided collaborations or emotional support during stressful moments. I would also like to thank Daniel Cloerkes, Stefan Dollase, Matthias Gehnen, Alexandra Heuschling, Gregor Kobsik, Henri Lotze, Hannah Mertens, Tim Quatmann, Jip Spel and Tobias Winkler for their commitment in organizing the second semester lecture "Datenstrukturen und Algorithmen" during the 4 years that I assisted there.

I would like to thank my students for their valuable work, and the feedback they gave me to improve myself. I would especially like to thank Patrick Arens, Alexander Ferber, Fabian Gasser, Marvin Jansen, Mohamed Khalifa, Patrick Nossol, Dominic Meiser and Jan Tugsbayar, whom I had the honour to supervise during their bachelor's and/or master's theses. I firmly believe that I learned something new during each supervision.

I would also like to thank Nina Štumberger and Tamira Esser for the love and emotional support they gifted me during my PhD. I thank my friends and family for supporting me and allowing me to think about other things than research. I would especially like to thank those, who supported me through hardships during the time of my PhD that were not the PhD itself.

Last but not least, I thank the developers of the LaTeX kaobook class, as this dissertation was typeset with it and Don Kringel for making the cover of this dissertation.

Abstract

In this dissertation, we consider *concurrent programs*, *probabilistic programs* and especially the *combination of both*. To reason about concurrent programs, we facilitate *(classical) separation logic* to derive *axiomatic semantics*. Separation logic allows us to declare a *separation of heaps*. That is, we can declare that predicates hold at *different locations* in the heap. We can thus partition the heap into a sub-heap used by each thread and a sub-heap used by multiple threads: the shared memory. This idea constitutes *concurrent separation logic*. For probabilistic programs, we use *quantitative logics*, which allow us to define a mapping from a random variable and an initial state to the expected value of the respective random variable after executing the given probabilistic program with the given initial state. A quantitative logic differs from a classical logic in that it does not map models to Boolean values, but to real values. Our used quantitative logics also allow for separation operations and thus constitute *quantitative separation logics*. Using this quantitative logic, we can derive axiomatic semantics to prove an *upper bound on the expected value of a random variable*. To finally reason about concurrent and probabilistic programs, we combine the techniques to reason about concurrent programs and to reason about probabilistic programs. We thus introduce a *fuzzy separation logic for concurrent and probabilistic programs* by combining the method to deal with shared memory in concurrent separation logic and the method to deal with expected values in quantitative separation logic. A fuzzy logic maps models to real values between zero and one. This enables us to derive axiomatic semantics to reason about *lower bounds of the probability that the program terminates in a certain set of states or does not terminate*.

The usage of the presented axiomatic semantics requires *entailments* in the respective logic to be proven. A quantitative or fuzzy entailment is the point-wise lifting of the less-than-or-equals relation. This constitutes a challenge as entailments in the logic without restrictions are undecidable. We thus consider one technique to reason about entailments in fuzzy separation logic and one technique to reason about entailments in quantitative separation logic with user-defined predicates. To reason about entailments in fuzzy separation logic, we transform the entailment into an entailment in classical separation logic. This allows the use of a rich landscape of tools designed for entailments in classical separation logic. We present a restricted syntax which allows the *fuzzy part* to be solved automatically, while discharging the *separation logic part* to separation logic entailment checkers. This restricted syntax allows formulae whose semantics has finite image. The second technique allows proving entailments in quantitative separation logic with *quantitative user-defined predicates*. Quantitative user-defined predicates are given using recursive specifications. With these, we can define predicates that describe the shape for data structures or measure their size. To reason about recursively-defined predicates, we use *cyclic proofs*. A cyclic proofs allows back-linking in proofs to reuse previous proof statements. Such a proof is unsound in general. We introduce a *progress criterion* on these proofs to guarantee soundness. It turns out that the classic progress criterion using unfoldings of predicates on the left-hand side of the entailment is *not easily applicable* to quantitative logics. Instead we use *heap sizes as our progress criterion*. In experiments, the progress criterion does not seem to be a limitation in the applicability of the proof system.

Some parts of this dissertation were also formalized in the *proof assistant language Lean* and thus allow the usage of the formalized proof rules to reason about concurrent and probabilistic programs in Lean.

Zusammenfassung

In dieser Dissertation betrachten wir *nebenläufige Programme*, *probabilistische Programme* und insbesondere die Kombination beider. Um über nebenläufige Programme zu argumentieren, benutzen wir *(klassische) Separationslogik* um eine *axiomatische Semantik* abzuleiten. Separationslogik erlaubt es uns, die *Separation von dynamischem Speicher* zu deklarieren. Das bedeutet, dass wir ausdrücken können, dass Prädikate an *verschiedenen Adressen* im dynamischen Speicher gelten können. Wir können also den dynamischen Speicher in Teil-Speicher unterteilen, eines für jeden Prozess und eines das von allen Prozessen genutzt werden kann: den geteilten Speicher. Diese Idee begründet *nebenläufige Separationslogik*. Für probabilistische Programme benutzen wir eine quantitative Logik. Diese erlaubt es uns eine Abbildung von Zufallsvariablen und einem initialen Zustand des Speichers zu dem Erwartungswert der jeweiligen Zufallsvariable nach Ausführung des probabilistischen Programs im gegebenem Initialzustand zu definieren. Eine quantitative Logik unterscheidet sich von einer klassischen Logik in dem Sinne, dass sie nicht Modelle in Boolesche Werte abbildet, sondern stattdessen in Werte in den reellen Zahlen. Die von uns genutzten quantitative Logiken erlauben außerdem die Benutzung von Separationsoperatoren and bilden damit *quantitative Separationslogiken*. Mit dieser quantitativen Logik können wir nun axiomatische Semantiken ableiten, die es uns erlaubt eine *obere Grenze auf dem Erwartungswert einer Zufallsvariable* zu beweisen. Um schließlich über nebenläufige und probabilistische Programme zu argumentieren, kombinieren wir die Techniken um über nebenläufige und über probabilistische Programme zu argumentieren. Wir führen also eine Fuzzy und Separationslogik für nebenläufige und probabilistische Programme ein, indem wir die Ideen von nebenläufige Separation Logic und von quantitativer Separation Logic vereinen. Eine Fuzzylogik bildet Modelle auf reelle Zahlen zwischen null und eins ab. Dies ermöglicht uns eine axiomatische Semantik abzuleiten, mit der wir über *untere Grenzen der Wahrscheinlichkeit in einer bestimmten Menge von Zustände zu terminieren oder nicht zu terminieren* argumentieren können.

Die Nutzung der vorgestellten axiomatischen Semantiken erfordert es, *Implikationen* in den entsprechenden Logiken zu beweisen. Eine quantitative oder fuzzy Implikation ist die punktweise Erweiterung der Kleiner-Gleich Relation. Dies stellt jedoch eine Herausforderung dar, da Implikationen in diesen Logiken ohne Restriktionen unentscheidbar sind. Daher betrachten wir eine Technik um über Implikationen in Fuzzy- und Separationslogik zu argumentieren und eine Technik um über Implikationen in quantitativer Separationslogik mit Nutzer-definierten Prädikaten zu argumentieren. Für Implikationen in der Fuzzy- und Separationslogik transformieren wir eine fuzzy Implikation in klassische Separationslogik. Dies erlaubt es uns eine reiche Auswahl an Werkzeugen für klassische Separationslogik zu nutzen. Wir präsentieren eine eingeschränkte Syntax für Fuzzy- und Separationslogik, die es uns erlaubt den *fuzzy Anteil* automatisch zu lösen, während wir den *Separationslogik Teil* an ein Programm übergeben, welches Implikationen in Separationslogik beweisen kann. Diese eingeschränkte Syntax erlaubt nur Formeln, deren Semantik eine endliche Wertemenge haben. Die zweite Technik erlaubt es uns Implikationen in der quantitative Separationslogik mit *quantitativen Nutzer-definierten Prädikaten* zu beweisen. Quantitative Nutzer-definierte Prädikate sind mithilfe von rekursiven Spezifikationen gegeben. Mit diesen können wir sowohl Prädikate die die Form von Datenstrukturen beschreiben, als auch für deren Länge definieren. Um über rekursiv-definierte Prädikate zu argumentieren, benutzen wir *zyklische Beweise*. Ein zyklischer Beweis erlaubt es uns Rückverbindungen zu vorherigen Beweis-Ausdrücken zu bilden. Solche Beweise sind im Allgemeinen nicht gültig. Daher führen wir eine festgelegte *Terminierungsbedingung* auf diesen Beweisen ein. Es stellt sich heraus, dass die klassische Terminierungsbedingung, die Ausfaltungen von Prädikaten auf den linken Seiten von Implikationen nutzt, nicht leicht auf quantitative Logiken anwendbar ist. Stattdessen nutzen wir *die Größe des dynamischen Speichers als Terminierungsbedingung*. In Experimenten scheint diese Terminierungsbedingung kein limitierender Faktor für die Anwendbarkeit des Beweissystems zu sein.

Einige Teile dieser Dissertation wurden in der Beweis-Assistenten Sprache Lean geschrieben. Dies

erlaubt es uns die formalisierten Beweisregeln zu nutzen, um über nebenläufige und probabilistische Programme in Lean zu argumentieren.

Declaration of Authorship

Eidesstattliche Erklärung

I, Ira Justus Fesefeldt, hereby declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original researchh.

Hiermit erkläre ich an Eides statt / I solemnly swear that:

1. This work was done wholly during the 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 states;
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 has been published before as outlined in Section 1.4.

Ira Fesefeldt, Mai 2025, Aachen

Contents

Acknowledgments iii

Abstract v

Zusammenfassung vii

Declaration of Authorship ix

Contents xi

1. Introduction 1
- 1.1. A Journey through Probabilistic, Concurrent and Heap-manipulating Programs . . . 1
- 1.2. A History of Deductive Program Verification . . . 2
- 1.3. Synopsis . . . 5
- 1.4. Contributions to Published Papers . . . 7

2. Fixed Point Theory 9
- 2.1. Order Theory . . . 10
- 2.2. Knaster-Tarski's Fixed Point Theorem . . . 14
- 2.3. Folklore's and Cousot's Fixed Point Theorems . . . 16

Separation Logic in Program Verification 23

3. Concurrent Separation Logic 25
- 3.1. Separation Logic . . . 26
- 3.2. Syntax and Operational Semantics of the Programming Language . . . 31
- 3.3. Axiomatic Semantics . . . 36

4. Quantitative Separation Logic 49
- 4.1. Markov Decision Processes . . . 50
- 4.2. Operational Semantics for Probabilistic Programs . . . 57
- 4.3. Quantitative Logic . . . 60
- 4.4. Axiomatic Semantics for Probabilistic Programs . . . 66

5. Fuzzy Separation Logic 75
- 5.1. Truncated Arithmetic on the Unit Interval . . . 76
- 5.2. Fuzzy Separation Logic . . . 84
- 5.3. Proof Rules for Fuzzy Separation Logic . . . 88

6. Concurrent Probabilistic Programs 97
- 6.1. Concurrent Probabilistic Programs . . . 98
- 6.2. Weakest Resource-Safe Liberal Expectation . . . 104
- 6.3. Axiomatic Semantics for Lower Bounds on Concurrent Probabilistic Programs . . . 112
- 6.4. Lossy Producer Consumer Example . . . 119

7. First Epilogue 131

Entailments in Fuzzy and Quantitative Separation Logic 137

8. Transforming Fuzzy Separation Logic into Qualitative Separation Logic 139
8.1. The At-Least Predicate 140
8.2. Proof Rules for the At-Least Predicate 143
8.3. Complexity of a Syntactic Transformation 148

9. Cyclic Proofs for Quantitative Separation Logic 159
9.1. Recursively-Defined Quantitative Separation Logic Predicates 160
9.2. Cyclic Proofs 165
9.3. Heap Size as Global Soundness Criterion 169
9.4. Automatization 178

10. Second Epilogue 185

Bibliography 189

Notation 197

Alphabetical Index 199

List of Figures

1.1. Recommended reading order for this dissertation 5

2.1. Counter example for necessity of Scott-continuity 20

3.1. Operational semantics for statements in cPL 33
3.2. Operational Semantics for flow related program constructs in cPL. 34
3.3. Axiomatic semantics for CSL 40

4.1. Operational semantics for statements in pPL 58
4.2. Operational Semantics for flow related program constructs in pPL. 60
4.3. Axiomatic semantics for QSL 70

6.1. Operational semantics for statements in cpPL 100
6.2. Operational semantics for control flow related program constructs in cpPL. . . 101
6.3. Axiomatic semantics for FSL 114
6.4. Program modeling a producer consumer protocol with a lossy channel 120

9.1. A proof tree demonstrating the necessity for cyclic reasoning. 160
9.2. Semantics for a QSL syntax with generic predicates. 162
9.3. First part of a proof system which uses the heap size as progress criterion . . 171
9.4. Second part of a proof system which uses the heap size as progress criterion . 171
9.5. Cyclic proof for a qualitative entailment with list data structures 174
9.6. Cyclic proof for a quantitative entailment with list data structures 175

List of Tables

3.1. Semantics of qualitative separation logic 29

4.1. Semantics for quantitative separation logic 62

5.1. Semantics for fuzzy separation logic 85

8.1. *SL* fragments with decidable entailment problem 158
8.2. Requirements for SL checkers to obtain an FSL checker. 158

9.1. Evaluation of cyclic proofs in SL 179
9.2. Evaluation of cyclic proofs in QSL without recursively-defined predicates . . . 180
9.3. Evaluation of cyclic proofs in QSL with recursively-defined predicates 184

Introduction | 1.

Erroneous software is both *live-risking* and *costs trillions of dollars* [1, 2]. As such, techniques to reduce errors in software is a means to safe lives and to safe money. Guaranteeing the absence of bugs has been the focus of the research on *deductive verification of programs*. This dissertation focuses on the deductive verification of *probabilistic, concurrent and heap-manipulating programs*. Such programs are thus able to flip coins to make decisions, consist of multiple threads running concurrently and manipulate an addressable memory, which we call heap.

1.1. A Journey through Probabilistic, Concurrent and Heap-manipulating Programs

Deterministic programs are a means to computing. However, programs can be used to model various problems. Program verification is then used to verify certain statements on the problem that we modeled as a program. This application is prevalent in various fields.

In mathematics, the famous Curry-Howard correspondence [3–5] allows us to model proofs of theorems as programs. We will make use of this correspondence in the sense that we use the proof assistant Lean [6] to formalize parts of this dissertation. Lean uses the calculus of constructions [7], which is based on the Curry-Howard correspondence.

Probabilistic Programs Probabilistic programs are programs that may flip a coin to introduce a probabilistic form of non-determinism. Probabilistic programs are widely used to implement efficient randomized algorithms, implement probabilistic protocols, implement artificial intelligence or even to model stochastic experiments.

Probabilistic programs may be used to model stochastic events in fields such as psychology [8, 9]. It is also used in computer graphics to generate randomly-behaving structures such as trees [10]. In artificial intelligence, we can implement agents to solve problems in an uncertain environment in a probabilistic programming language [11].

In computation, we want to use probabilistic programming to solve problems in an approximated way efficiently or to improve the runtime of classical programs. In the first case, we solve the problem approximately correct in *expectation*. An example for such an algorithm is an algorithm to compute the circle constant π: We randomly pick points in the square from -1 to 1. Then we count all points with an euclidean distance smaller than 1. This number multiplied with 4 and divided by the number of all points is an estimate for π. In the second case, we usually obtain an improvement of the runtime in *expectation*. That is, runtime or the correctness of the solution may only be achieved with a *certain probability*.

Probabilistic and Heap-manipulating Programs An example for the second kind of algorithms is randomized quicksort [12]. There we sort an unsorted array by choosing an element from the array to partition it into elements greater or equal than the chosen element and to elements lesser than the chosen element. The sub-arrays are then recursively partitioned with new randomly chosen elements. If we pick by chance bad elements, the runtime of this algorithm may be quadratic, but is optimal in case we randomly choose good elements. Another example involves computing minimal spanning trees. A spanning tree of a graph is a connected sub-graph that contains all vertices. For a weighted graph, a spanning tree is minimal if of all spanning trees, it contains the least sum of weights. The theoretical minimal worse-case running time of a deterministic algorithm to find such a minimal spanning tree is unknown. If we allow probabilistic computation, an algorithm running in expected linear time is, however, known [14]. Such algorithms use data structures (i.e. arrays or graphs). There are also examples of data structures that use randomization to enhance the efficiency of the data structure such as bloom filters [15]. Bloom filters implement sets where we can only query membership with a certain probability and non-membership certainly. Such examples justify the necessity of *probabilistic and heap-manipulation programming languages.*

An optimal algorithm to find a minimal spanning tree is, however, known [13].

Probabilistic and Concurrent Programs When we include concurrency in our programming language, we are able to model communication protocols. Many such protocols use probabilities to either model the environment or to resolve symmetries. For example, the unreliability of a cable can be modeled by a communication failing with a certain probability. Collisions when using a bus system may introduce such a failing communication. The CSMA/CD [16] protocol uses probabilistically chosen waiting times to resolve such collisions. Some decentralized network architectures require the election of a leader node in the network to make decisions. Unfortunately, it is known that such a leader cannot be elected deterministically in any network if the nodes are non-distinguishable [17]. However, with randomized algorithms this is possible [18]. To model probabilistic programs in networks we propose the use of *probabilistic and concurrent programs.*

Probabilistic, Concurrent and Heap-manipulating Programs However, we can also find randomized and parallel programs such as [19], where we use randomization and parallelization to find the shortest path from a single-source. Indeed, here we also have to use data-structures to model the graph on which the algorithm is executed. In turn, we require *probabilistic, concurrent and heap-manipulating programming languages* for such algorithms.

1.2. A History of Deductive Program Verification

We use deductive verification in this dissertation to gain *mathematical guarantees* on the absence of errors in program. We follow here the Floyd-Hoare [20, 21] style of deductive verification. That is, we use specifications of the form

$$\vdash \{\Phi\}\ C\ \{\Psi\}$$

to say that the program C starting in an *initial state satisfying* Φ will either *not terminate* or *terminate in a state satisfying* Ψ. For this reason, we call Φ a *precondition* and Ψ a *postcondition.* A reformulation of such specifications has been done by Dijkstra in terms of *weakest (liberal) preconditions* [22]. That is, for a given program C and a postcondition Ψ, we can compute the maximal set of initial states guaranteeing non-termination of fulfillment of the postcondition — which we call weakest liberal preconditions. The weakest precondition excludes non-termination. The previous Floyd-Hoare specification is then equivalent to an entailment between the given precondition Φ and the weakest liberal precondition $wlp[C](\Psi)$:

$$\vdash \{\Phi\}\ C\ \{\Psi\} \iff \Phi \vDash wlp[C](\Psi)$$

Both, the proof system for Floyd-Hoare specifications and the entailment for weakest liberal preconditions require us to check entailment in the used logic. An entailment between two formulae states that whenever a model satisfies the left-hand side formula, then so should the model satisfy the right-hand side formula. For ordinary programs, we usually use *first order logic* and can thus use off-the-shelf techniques to solve entailments for both proof systems for Floyd-Hoare specifications and weakest liberal preconditions entailments. We will see that checking entailments is a reoccurring challenge for other specifications of this style.

Separation Logic To reason about heaps, many successful techniques use *separation logic* [23, 24] instead of first order logic. Separation logic extends first order logic by separating operations, most prominently the *separating conjunction.* The separating conjunction states that the two statements connected by the conjunction hold in *disjoint sub-heaps.* This allows to localize reasoning for heaps containing multiple but separate data and data-structures.

However, the use of separation logic instead of first order logic introduces the challenge that off-the-shelf entailment checkers are not applicable anymore. As such, *new entailment checkers* specialized for separation logic were developed.

Another ground-breaking discovery is the additional use of the separating conjunction to model *data ownership* in the presence of *concurrency.* Indeed, it turns out that we can use separation logic to develop an elegant *proof system for concurrent programs* [25, 26]. To model the shared memory, we introduce another component in the specification, called the *resource invariant* ξ. We thus have that the specification

$$\xi \vdash \{\Phi\}\ C\ \{\Psi\}$$

means that the program C starting in an initial state satisfying Φ

1. has only executions such that for any “change of the shared memory adhering to the resource invariant”, after every execution-step the shared memory still satisfies ξ, and
2. execution either does not terminate or terminates in a state satisfying Ψ.

The first guarantees a certain *inference freeness* of the program and the second guarantees us functional correctness of the program. For the definition of this specification, the shared memory is not included in the used state. Instead we add it before every execution-step and remove it after every execution-step again. This way we can simulate arbitrary changes to the shared memory *adhering to the resource invariant* done by the environment.

We will later call this probabilistic weakest precondition the weakest expectation instead.

The indicator function maps to 1 if the model is in the corresponding set and 0 otherwise.

Probabilistic Weakest Precondition Concurrently to the development of separation logic, there has also been development to extend weakest precondition style reasoning to *probabilistic programs* [27–29]. For this, we use a *quantitative logic* instead of ordinary first order logic. A quantitative logic maps a models not to Booleans, but to (some subset of) the reals. The weakest precondition yields the expected value of a random variable after execution and parameterized by the initial state. There also exists a form of conservativity. If the program is deterministic, the random variable is the indicator function of a formula in first order logic and the parameterized expected value is the indicator function of a formula in first order logic, then the probabilistic weakest precondition and the ordinary weakest precondition coincide. With this probabilistic weakest precondition, we obtain specifications of the form

$$X \bowtie wp[C](Y).$$

We can use this to gain various styles of specifications, for example

- if $\bowtie = \leq$ and $Y(\sigma) = 1$ if $\sigma \in A$ else 0, we specify that the program C starting in the initial state σ terminates in a state contained in A with a probability of at least $X(\sigma)$; and
- if $\bowtie = \geq$ and $Y(\sigma) = \sigma(x)$ for a program variable x, then we specify that the expected value of x after executing the program C in the initial state σ is at most $X(\sigma)$.

Separation Logics for Probabilistic Programs It may not be surprising that in the presence of a logic to reason about probabilistic programs and a logic to reason about heaps and concurrency, there is also interest in a logic to reason about probabilistic programs which may manipulate a heap and act concurrently. By combining the idea of quantitative logics and separation logic, we obtain quantitative separation logic [30, 31] and fuzzy separation logic [31–33]. The first is a separation logic which maps variable assignments and heaps to *non-negative reals with a top element*, the second is a separation logic which maps to reals in the interval between zero and one. This gives rise to weakest precondition calculi for probabilistic and heap-manipulating programs [30] and weakest liberal precondition calculi for probabilistic, concurrent and heap-manipulating programs [33], respectively. In case of the second, we also obtain resource invariants in fuzzy separation logic. It may not be surprisingly that in the probabilistic setting we may not get certain inference freeness by using resource invariants in fuzzy separation logic, but only inference freeness with a certain probability. We will go into more detail for both of these calculi in this dissertation.

Similar to separation logic, we require newly developed techniques to solve entailments in quantitative and fuzzy separation logic. We will see two techniques to verify entailments in this dissertation. The first transforms fuzzy separation logic formulae into classical separation logic formulae, to be able to facilitate entailment checkers for classical separation logic. The second uses cyclic reasoning to allow for a proof system capable to prove entailments on recursively-defined predicates. Cyclic reasoning requires a special soundness criterion. The classic soundness criterion is troublesome for qualitative logics. As such, we use a new soundness criterion based on heap-sizes for quantitative separation logic.

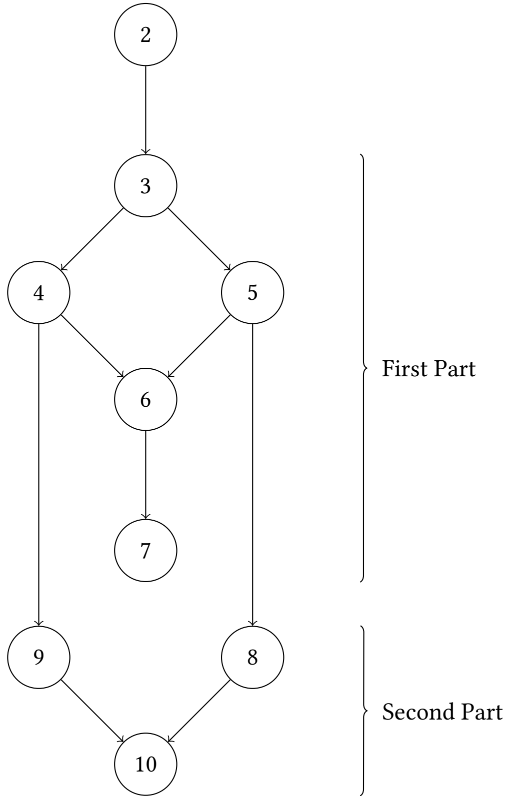


Figure 1.1.: We recommend a reading order of the chapters satisfying the dependency graph depicted here.

1.3. Synopsis

This dissertation is structured in two parts and ten chapters. The first two chapters are introductory and are thus not included in any of the two parts. In Figure 1.1, we present how topics are dependent on each other. In order to read a certain chapter, we recommend reading all chapters it depends on.

You are currently reading Chapter 1, in which we introduce the content of this dissertation. Chapter 2 introduces foundations on order theory and fixed point theory based on monotonic functions in ordered sets that contain least upper bounds and greatest lower bounds for all subsets.

First Part The first part consists of Chapters 3 to 7 and introduces deductive program verification for programs with various features.

In Chapter 3, we introduce seminal work on concurrent separation logic. We first introduce classical separation logic. Then we introduce a programming language supporting heap-manipulation and concurrency and its operational semantics. Finally we define a weakest liberal precondition and Floyd-Hoare style specifications in separation logic for this programming language, and a corresponding proof system.

In Chapter 4 we introduce foundations for probabilistic semantics, namely Markov decision processes. This gives rise to a probabilistic and heap-manipulating programming language, of which the operational semantics is indeed a Markov decision process. Finally we introduce quantitative separation logic, a probabilistic weakest precondition using quantitative separation logic and Floyd-Hoare style specifications for such programs. Finally we introduce a proof system for these specifications.

In Chapter 5 we develop a fuzzy quantitative separation logic and present its formalization in Lean.

A concurrent fuzzy separation logic is introduced in Chapter 6. There we define a programming language supporting probabilistic branching, concurrency and heap-manipulating and define its operational semantics as a Markov decision process. We further develop a weakest liberal precondition calculus using fuzzy separation logic, introduce Floyd-Hoare style specifications for such programs and present a proof system for such specifications. The operational semantics for probabilistic, concurrent and heap-manipulating programs and the corresponding proof system for specifications on such programs are also formalized in Lean.

We conclude this part and present related work in Chapter 7.

My contributions are found in Chapters 5 and 6. In Chapter 5, I contributed the formalization of fuzzy separation logic as a logic of *truncated operations*, as well as a formalization in Lean. I also contributed the content of Chapter 6 and its Lean formalization.

Second Part The second part consists of Chapters 8 to 10 and introduces one technique to solve entailments in fuzzy separation logic and one technique to solve entailments in quantitative separation logic with user-defined predicates.

A transformation from fuzzy separation logic to classical separation logic is introduced in Chapter 8. We first present generic proof rules to transform entailments in fuzzy separation logic to multiple entailments in classical separation logic. Then we present a fragment of fuzzy separation logic that allows automatic discharging the resulting entailments in entailment checker for classical separation logic and analyze the runtime complexity of the transformation.

A proof system for quantitative separation logic is presented in Chapter 9. First we introduce the notion of recursively-defined predicates, which are equations that define predicates. These predicates may be defined mutually and are allowed to occur in entailments to be checked. The proof system allows for cyclic proofs, for which we then introduce an additional soundness criterion. Since the classical soundness criterion is not sufficient, we introduce a soundness criterion based on heap sizes and prove its correctness.

Finally in Chapter 10, we conclude the second part and present related work.

I contributed to the content in Chapter 8 and contributed the Lean formalization of this transformation. Moreover, I contributed the development of proof systems for qualitative logics presented in Chapter 9 and the soundness criterion based on heap sizes.

1.4. Contributions to Published Papers

During my PhD, I published three peer-review papers. In the following I will outline my contribution to all of those.

[34] Ira Fesefeldt, Christoph Matheja, Thomas Noll, and Johannes Schulte. "Automated Checking and Completion of Backward Confluence for Hyperedge Replacement Grammars." In: *Graph Transformation*. Ed. by Fabio Gadducci and Timo Kehrer. Cham: Springer International Publishing, 2021, pp. 283–293.

The initiative for conducting this research came from Christoph Matheja. I have contributed to the evaluation of the presented tool and the writing of the paper.

[32] Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, et al. "Foundations for Entailment Checking in Quantitative Separation Logic." In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Cham: Springer International Publishing, 2022, pp. 57–84. DOI: 10.1007/978-3-030-99336-8_3.

The initiative for conducting this research came from Christoph Matheja and Thomas Noll. I have contributed to the theory, the case studies and the writing of the paper. My contributions to this paper are presented in Chapter 8.

[33] Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. "Towards Concurrent Quantitative Separation Logic." In: *33rd International Conference on Concurrency Theory (CONCUR 2022)*. Ed. by Bartek Klin, Sławomir Lasota, and Anca Muscholl. Vol. 243. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 25:1–25:24. DOI: 10.4230/LIPIcs.CONCUR.2022.25.

The initiative for conducting this research came from me. I have contributed to the theory, the case studies and the writing of the paper. My contributions to this paper are presented in Chapter 6.

Fixed Point Theory 2.

Throughout this thesis, we assume the reader to be familiar with both order theory and fixed point theory. We thus often drop details on these fixed points for brevity and concentrate on new contributions. This chapter aims to give readers unfamiliar to these concepts a gentle introduction.

As an introduction to this chapter's content, we consider a small example. We define the *extended non-negative reals* $\mathbb{R}_{\geq 0}^{\infty}$ as the set of non-negative reals extended with a top element we call infinity. We denote the non-negative reals without that top element as $\mathbb{R}_{\geq 0}$. Let us now consider the function

$$f \colon \mathbb{R}_{\geq 0}^{\infty} \to \mathbb{R}_{\geq 0}^{\infty}, a \mapsto \frac{a+1}{a+2}.$$

Here we define division with infinity as

$$\forall a \in \mathbb{R}_{\geq 0}^{\infty}.\ \frac{a}{\infty} = 0 \quad \text{and} \quad \forall a \in \mathbb{R}_{\geq 0}.\ \frac{\infty}{a} = \infty.$$

We do not need to define division by zero here as $a + 2$ is always positive. The main goal of this chapter is to find methods to prove that a value a is a *fixed point* of f, i.e. $f(a) = a$, and how to describe (and sometimes even compute) the *least* and *greatest fixed points*. In general, a least and greatest fixed point must not exist. However, in Section 2.2 we will see why the function f definitely has them. For now, we observe that f is *monotone*. That is, whenever we take two ordered values $a \leq b$ we also have that $f(a) \leq f(b)$, where $\leq$ is the usual order on $\mathbb{R}_{\geq 0}$ with $\forall a \in \mathbb{R}_{\geq 0}^{\infty}.\ a \leq \infty$. The dual property that for $a \leq b$ we have $f(b) \leq f(a)$ is called antitonicity. Monotonicity of f guarantees us that f has a least and a greatest fixed point due to a fixed point theorem we introduce in this chapter.

A different property, that f can have and which will be helpful, is *(co-)ω-Scott-continuity.* This requires us that f distributes over suprema (infima) using *(co-)ω-chains.* The function f does not distribute over suprema using ω-chains as the following example shows.

Scott-continuity is named after Dana Scott to differentiate it from topological continuity.

An ω-chain $\mathsf{chain} \colon \mathbb{N} \to \mathbb{R}_{\geq 0}^{\infty}$ is a map from natural numbers to values such that it is monotonic. Co-ω-chains are antitone. For the ω-chain

$$\mathsf{chain} \colon \mathbb{N} \to \mathbb{R}_{\geq 0}^{\infty}, n \mapsto n$$

we have that

$$\sup \left\{ \frac{\mathsf{chain}(i) + 1}{\mathsf{chain}(i) + 2} \,\middle|\, i \in \mathbb{N} \right\} = 1,$$

however we also have that

$$\frac{\sup \{ \mathsf{chain}(i) \mid i \in \mathbb{N} \} + 1}{\sup \{ \mathsf{chain}(i) \mid i \in \mathbb{N} \} + 2} = \frac{\infty + 1}{\infty + 2} = 0.$$

Indeed, the problem lies with infinity. It behaves in some sense differently in the finite cases then in the infinite case. This shows that f is not ω-Scott-continuous.

The function f is co-ω-Scott-continuous. For that we require that for every co-ω-chain we have

$$\inf\{f(\mathsf{chain}(i)) \mid i \in \mathbb{N}\} = f(\inf\{\mathsf{chain}(i) \mid i \in \mathbb{N}\}).$$

This does indeed hold as we either have the constant chain $\mathsf{chain}(i) = \infty$ or we have some i for which all $j > i$ satisfy $\mathsf{chain}(j) < \infty$. In the first case, we have

$$\inf\{f(\mathsf{chain}(i)) \mid i \in \mathbb{N}\} = \infty = f(\inf\{\mathsf{chain}(i) \mid i \in \mathbb{N}\}),$$

and in the second case we can use that division distributes over limits (thus also the limit of the chain) in the reals, if they are defined in the reals. They are defined in the reals and equal the infimum because the chain is bounded below by zero and is antitone. Since f is co-ω-Scott-continuous, we can express the greatest fixed point as the infimum of the following infinite iteration

$$\text{greatest fixed point of } f = \inf\left\{f^i(\infty) \,\middle|\, i \in \mathbb{N}\right\},$$

where $f^0(a) = a$ and $f^{i+1}(a) = f(f^i(a))$. Unsurprisingly, we can easily compute this value as ∞ is indeed the greatest fixed point.

We will also learn how to express the least fixed point of f in a similar way using *ordinals*.

2.1. Order Theory

In the introduction we saw the need for *orders* to define *monotonicity*, *least upper* and *greatest lower bounds*. While we gave a high-level overview there, we now go into details starting with the definition of a *partial order*.

Definition 2.1.1 (Partial Order) *We call $(A, \preccurlyeq)$ a partial order if and only if the relation $\preccurlyeq$ is*

- *reflexive* $\forall a \in A.\ a \preccurlyeq a$,
- *transitive* $\forall a, b, c \in A.\ a \preccurlyeq b \Rightarrow b \preccurlyeq c \Rightarrow a \preccurlyeq c$, *and*
- *anti-symmetric* $\forall a, b \in A.\ a \preccurlyeq b \Rightarrow b \preccurlyeq a \Rightarrow a = b$.

Example 2.1.1 The natural numbers form a partial order, where we define

$$a \leq b \iff \exists c \in \mathbb{N}.\ a + c = b.$$

Now we prove that $(\mathbb{N}, \leq)$ is a partial order.

- $(\mathbb{N}, \leq)$ is reflexive: For any $a \in \mathbb{N}$ we have $a + 0 = a$, thus also $a \leq a$.
- $(\mathbb{N}, \leq)$ is transitive: Assume we have $a \leq b$ and $b \leq c$, then there are a' and b' such that

  $$a + a' = b \quad \text{and} \quad b + b' = c.$$

  Thus we also have

  $$a + a' + b' = c,$$

  which proves that we have $a \leq c$.
- $(\mathbb{N}, \leq)$ is anti-symmetric: Assume we have $a \leq b$ and $b \leq a$, then there

are a' and b' such that

$$a + a' = b \quad \text{and} \quad b + b' = a.$$

Thus we have that

$$b + b' + a' = b \iff b' + a' = 0.$$

However, there is only the one solution with $b' = a' = 0$. Thus we have that

$$a + 0 = a = b.$$

Example 2.1.2 Let $(A, \preccurlyeq)$ be a partial order and X a set. The pointwise extensions of the order on A to maps from X to A form a partial order, where we define

$$a \sqsubseteq b \iff \forall x \in X.\, a(x) \preccurlyeq b(x).$$

We now prove that $(X \to A, \sqsubseteq)$ is a partial order.

- $(X \to A, \sqsubseteq)$ is reflexive: Since we have for any $a(x) \preccurlyeq a(x)$, we also have $a \sqsubseteq a$.
- $(X \to A, \sqsubseteq)$ is transitive: Assume we have $a \sqsubseteq b$ and $b \sqsubseteq c$. To prove that $a \sqsubseteq c$ we have to prove that for any $x \in X$ we have $a(x) \preccurlyeq c(x)$. By definition, we have $a(x) \preccurlyeq b(x)$ and $b(x) \preccurlyeq c(x)$ and thus by transitivity of $\preccurlyeq$ we also have $a(x) \preccurlyeq c(x)$.
- $(X \to A, \sqsubseteq)$ is anti-symmetric: Assume we have $a \sqsubseteq b$ and $b \sqsubseteq a$. To prove that $a = b$, we use functional extensionality. That is, we need to prove that

  $$\forall x \in X.\, a(x) = b(x).$$

  We assume an arbitrary $x \in X$. We have by definition that $a(x) \preccurlyeq b(x)$ and $b(x) \preccurlyeq a(x)$. By anti-symmetry of $\preccurlyeq$ we have that $a(x) = b(x)$. Thus we also have that $a = b$.

This thesis involves many pointwise defined functions and the monotonicity behavior of them.

A function is *monotone* if ordered elements are mapped to equally ordered elements. That is, if we have $a \preccurlyeq b$, a function $f\colon A \to A$ is monotone if we have $f(a) \preccurlyeq f(b)$. Since $(A, \preccurlyeq)$ is partial, a function $f\colon A \to A$ that is not monotone has for some values $a \preccurlyeq b$ that either $f(b) \prec f(a)$ or that they are not comparable. A function is *antitone* if ordered elements are mapped to exactly reversed ordered elements.

Definition 2.1.2 (Monotone and Antitone) *Let $(A_1, \preccurlyeq_1)$ and $(A_2, \preccurlyeq_2)$ be partial orders and $f\colon A_1 \to A_2$ a map.*
We call f monotone if and only if

$$\forall a, b \in A_1.\, a \preccurlyeq_1 b \Rightarrow f(a) \preccurlyeq_2 f(b).$$

We call f antitone if and only if

$$\forall a, b \in A_1.\, a \preccurlyeq_1 b \Rightarrow f(b) \preccurlyeq_2 f(a).$$

Example 2.1.3 We define the function f as

$$f\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{N}),\ a \mapsto (b \mapsto a).$$

We prove now that f is monotone using the partial orders from Examples 2.1.1 and 2.1.2. For this, we assume elements $a, b \in \mathbb{N}$ with $a \leq b$. Now we need to prove that $f(a) \sqsubseteq f(b)$. But this holds if we have that

$$\forall c \in \mathbb{N}.\, f(a)(c) \leq f(b)(c).$$

Thus we assume such an element $c \in \mathbb{N}$. Since we have that $f(a)(c) = a$ and $f(b)(c) = b$, it is sufficient to prove that $a \leq b$, which we have as assumption.

A linear oder is sometimes also called total order.

In contrast to partial orders, an order is *linear* if all elements are always ordered in one way or the other. Total orders grant access to the binary minimum and maximum operations, which maps elements to the lower and greater one respectively. It does not guarantee the existence of minima and maxima of infinitely many elements.

Definition 2.1.3 (Linear Order) *A partial order* $(A, \preccurlyeq)$ *is linear if and only if*

$$\forall a, b \in A.\, a \preccurlyeq b \vee b \preccurlyeq b.$$

Example 2.1.4 The order $(\mathbb{N}, \leq)$ is a linear order. To prove this, we let $a, b \in \mathbb{N}$. We prove the claim by induction on a.
First we let $a = 0$. Then we have that $a + b = 0 + b = b$ and thus $a \leq b$.
Next we let $a = a' + 1$. By the induction hypothesis we have that $a' \leq b \vee b \leq a'$. We do case distinction on this

- If $a' \leq b$, then we have that there is c with $a' + c = b$. Again, we do a case distinction on c.
  If $c = 0$ then $a' = b$ and thus we also have $a' + 1 = b + 1$ by congruence. Then we have by definition that $b \leq a' + 1 = a$.
  If $c = c' + 1$ then $a' + c' + 1 = b$ then also $(a' + 1) + c' = a + c' = b$ and therefore also $a \leq b$.
- If $b \leq a'$, then we have that there is c with $b + c = a'$. By congruence we also have that $b + c + 1 = a' + 1 = a$, and therefore by definition we have $b \leq a$.

Linear orders yield binary minima $\wedge$ and maxima $\vee$, but not arbitrary ones. We will not consider maxima and minima of arbitrarily many elements, but instead consider the more useful notion of the *least upper* and *greatest lower bound* of arbitrarily many elements. A complete lattice guarantees the existence of these. We call the least upper bound the supremum and the greatest lower bound the infimum. An order that guarantees their existence for arbitrary elements is called a *complete lattice*. It suffices to show that either the supremum or the infimum always exists for a complete lattice, as we can construct one from the other one.

We can construct the infimum $\bigwedge$ from the supremum $\bigvee$ by

$$\bigwedge X = \bigvee \{ a \in A \mid \forall x \in X.\, a \preccurlyeq x \}.$$

As $\bigvee$ always exists, the right side is defined and is the least upper bound of that

set. It follows immediately that it is greater than or equal to any lower bound. Thus we require that this element is also itself a lower bound. That is, we need to prove

$$\forall x \in X.\ \bigvee \{ a \in A \mid \forall x \in X.\ a \preccurlyeq x \} \preccurlyeq x.$$

Assuming any $x \in X$, we need to prove that for any $a \in A$ with $\forall x \in X.\ a \preccurlyeq x$ we have $a \preccurlyeq x$. But due to $\forall x \in X.\ a \preccurlyeq x$, we already have that $a \preccurlyeq x$.

Similarly we can construct the supremum $\bigvee$ from the infimum $\bigwedge$ by

$$\bigvee X = \bigwedge \{ a \in A \mid \forall x \in X.\ x \preccurlyeq a \}.$$

Definition 2.1.4 (Complete Lattice and Complete Linear Order) *A partial order $(A, \preccurlyeq)$ is a complete lattice if and only if for any subset $X \subseteq A$ there is an infimum $\bigwedge X$ such that*

$$\forall x \in X.\ \bigwedge X \preccurlyeq x \quad \text{and} \quad \forall a \in A.\ (\forall x \in X.\ a \preccurlyeq x) \Rightarrow a \preccurlyeq \bigwedge X,$$

and there is a supremum $\bigvee X$ such that

$$\forall x \in X.\ x \preccurlyeq \bigvee X \quad \text{and} \quad \forall a \in A.\ (\forall x \in X.\ x \preccurlyeq a) \Rightarrow \bigvee X \preccurlyeq a.$$

A complete lattice that is additionally a total order is called a complete linear order.

Example 2.1.5 The natural numbers $(\mathbb{N}, \leq)$ are not a complete lattice (and thus also not a complete linear order). As a counterexample, we consider the set $\mathbb{N}$, which is trivially a subset of $\mathbb{N}$.
We have for this set the infimum $\inf \mathbb{N} = 0$ as $\forall a \in \mathbb{N}.\ 0 \leq a$ and there is no other lower bound. To prove the second, we can consider any other lower bound a and disprove that it is a lower bound. Since $a \neq 0$ we have $a = a' + 1$. But then we have $a' \leq a$, disproving that a is a lower bound.
But we do not have a supremum $\sup \mathbb{N}$, because there is no upper bound for $\mathbb{N}$. Assume a is an upper bound in order to lead this to a contradiction. We have that $a \leq a + 1$, disproving the assumption.

Complete lattices require the set A to have a least element (bottom element) and a greatest element (top element) as they require the existence of the infimum of the whole set $\bigwedge A$ and the existence of the supremum of the whole set $\bigvee A$.

Example 2.1.6 We can extend the natural numbers with a top element $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. We extend the order $\leq$ with $\forall a \in \mathbb{N}^\infty.\ a \leq \infty$ and like before if the right side is not ∞. To prove that this is a complete linear order, we now need to adapt all previous proofs.

- $(\mathbb{N}^\infty, \leq)$ is reflexive: Let $a \in \mathbb{N}^\infty$. If $a \in \mathbb{N}$, the proof is already given. If $a = \infty$ we have by definition that $a = \infty \leq \infty$.
- $(\mathbb{N}^\infty, \leq)$ is transitive: We have that $a \leq b$ and $b \leq c$. If $c = \infty$ we directly have $a \leq \infty = c$. If $b = \infty$, we have $\infty \leq c \iff c = \infty$ and thus $a \leq \infty = c$. Similarly for $a = \infty$. If neither is infinity, the reason from before holds.
- $(\mathbb{N}^\infty, \leq)$ is anti-symmetric: We have that $a \leq b$ and $b \leq a$. If $a = \infty$ we have that $\infty = a \leq b \iff b = \infty$, proving the statement. Analogous

for $b = \infty$. If neither are infinite, we have the same reason as before.

- $(\mathbb{N}^\infty, \leq)$ is linear: We have $a, b \in \mathbb{N}^\infty$. Let be $a = \infty$, then we have $b \leq \infty = a$. We have an analogous proof for $b = \infty$. If neither are infinite, we previous proof can be applied

Lastly we need to show that the supremum and infimum always exist. As discussed before, it suffices to show that one exist. We will thus show that the infimum always exists. For this, let $X \subseteq \mathbb{N}^\infty$ be some subset. If $X = \emptyset$ we have that *every* element is an lower bound. The greatest element of all is ∞, therefore we have $\inf X = \inf \emptyset = \infty$. If $X \neq \emptyset$, than we can find the least element in X be counting up from 0 to a number a such that $a \in X$. By counting up, we know that a is the least element in X, making it trivially the greatest lower bound.

2.2. Knaster-Tarski's Fixed Point Theorem

In this section, we will take a closer look at *fixed points* and the closely related notion of *pre-* and *post-fixed points*. A fixed point of a function $f : A \to A$ is an element $a \in A$ such that $f(a) = a$. A function may not have any fixed points, can have exactly one and can have multiple fixed points. However, if the function is monotone and its domain is a complete lattice, it is *guaranteed* to have at least one fixed point. A pre-fixed point of a function f is an element $a \in A$ such that $f(a) \preccurlyeq a$. A pre-fixed point is an element which is decreased by applying the function on it. Dually, a post-fixed point of a function f is an element $a \in A$ such that $a \preccurlyeq f(a)$, that is an element which is increased by applying the function on it. The intuition behind the nomenclature is that a pre-fixed point has the function placed *before* the order sign, a post-fixed point has the function placed *after* the order sign.

We need to ignore the possibility of mirroring the symbol — or consider the mirrored version a different relation.

Definition 2.2.1 (Fixed, Pre-Fixed and Post-Fixed Points) *Let $(A, \preccurlyeq)$ be a partial order and $f : A \to A$ be a function.*

- *An element $a \in A$ is a fixed point if and only if $f(a) = a$.*
- *An element $a \in A$ is a pre-fixed point if and only if $f(a) \preccurlyeq a$.*
- *An element $a \in A$ is a post-fixed point if and only if $a \preccurlyeq f(a)$.*

Example 2.2.1 We consider again the natural numbers with top element $(\mathbb{N}^\infty, \leq)$ and the function

$$f : \mathbb{N}^\infty \to \mathbb{N}^\infty, a \mapsto \text{if } (a \leq 5) \text{ then } 5 \text{ else } 6.$$

The function f has two fixed points, six post-fixed points and infinitely many pre-fixed points. The fixed points of the function are $f(5) = 5$ and $f(6) = 6$. The post-fixed points are $a = 0, 1, 2, 3, 4, 5 \leq 5 = f(a)$ and $a = 6 \leq 6 = f(a)$. Finally, the pre-fixed points are all numbers a greater than or equal to 6, i.e. $f(a) = 6 \leq a$, or equal to 5 as $f(a) = 5 \leq 5$. The bottom element is always trivially a post-fixed point and the top element is always trivially a pre-fixed point.

A fixed point is always also a pre-fixed and a post-fixed point due to reflexivity of the partial order. If the order is a complete lattice, we can also take the infimum of all pre-fixed points and the supremum of all post-fixed points. It turns out that these are *fixed points* if the function is monotone. The infimum of all pre-fixed points is the *least fixed point* and the supremum of all post-fixed points is the *greatest fixed point*.

We show a proof sketch for the least fixed point here. For that, we prove

$$f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}) = \bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}.$$

First we apply anti-symmetry and are required to prove that

$$f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}) \preccurlyeq \bigwedge \{ a \in A \mid f(a) \preccurlyeq a \},$$

which follows by first eliminating the right infimum to receive one arbitrary pre-fixed point b and applying transitivity together with the pre-fixed property to receive

$$f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}) \preccurlyeq f(b).$$

Now we can use monotonicity of f and obtain the obviously true statement as b is a pre-fixed point that

$$\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \} \preccurlyeq b.$$

Next we need to prove the second premise of anti-symmetry, i.e. that

$$\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \} \preccurlyeq f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}).$$

We observe that if

$$f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}) \in \{ a \in A \mid f(a) \preccurlyeq a \},$$

the statement holds. We thus need to prove that applying the function on the infimum still yields a pre-fixed point, i.e. that

$$f(f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \})) \preccurlyeq f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}).$$

By applying monotonicity, we obtain the statement we already proved first

$$f(\bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}) \preccurlyeq \bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}.$$

Theorem 2.2.1 (Least and Greatest Fixed Points [35, 36]) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ a monotone function.*
The least fixed point of f (denoted as $\mathsf{lfp}(f)$) is

$$\mathsf{lfp}(f) \quad = \quad \bigwedge \{ a \in A \mid f(a) \preccurlyeq a \}.$$

The greatest fixed point of f (denoted as $\mathsf{gfp}(f)$) is

$$\mathsf{gfp}(f) \quad = \quad \bigvee \{ a \in A \mid a \preccurlyeq f(a) \}.$$

Example 2.2.2 We consider again the natural numbers with top element $(\mathbb{N}^\infty, \leq)$ and the function

$$f\colon \mathbb{N}^\infty \to \mathbb{N}^\infty, a \mapsto \text{if } (a \leq 5) \text{ then } 5 \text{ else } 6.$$

We have that

$$\mathsf{lfp}(f) \quad = \quad \inf\{\, a \in \mathbb{N}^\infty \mid f(a) \le a \,\} = \inf\{\, a \in \mathbb{N}^\infty \mid 5 \le a \,\} = 5,$$

as all values greater than or equal to 5 are pre-fixed points. We also have that

$$\mathsf{gfp}(f) \quad = \quad \sup\{\, a \in \mathbb{N}^\infty \mid a \le f(a) \,\} = \sup\{\, 0, 1, 2, 3, 4, 5, 6 \,\} = 6,$$

as these values are all post-fixed points.

Although interesting, this will only play a minor role in thesis. We remark it here for completeness.

Knaster-Tarski's theorem states something even stronger. If we take a set of fixed points on a monotone function f, their suprema and infima are always defined. That is, the fixed points of a monotone function form again a complete lattice.

Theorem 2.2.2 (Knaster-Tarski's Theorem [35, 36]) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ a monotone function. The set of fixed points on f $(\{\, a \in A \mid f(a) = a \,\}, \preccurlyeq)$ is a complete lattice.*

As a last part of this section, we will investigate a theorem allowing approximations using pre- and post-fixed points. Least and greatest fixed points are often used to define the semantics of recursively defined objects. Gaining lower and upper bounds on these objects is an often occurring problem. In terms of program verification, we often define the semantics of programs as least or greatest fixed points of characteristic functions. To approximate these semantics, we use so called *invariants*, *sub-invariants* or *super-invariants*. These refer to pre-fixed and post-fixed points in our order theoretical sense. If we need to over-approximate a least fixed point, it is sufficient to guess a pre-fixed point and prove that it is a pre-fixed point and dually for greatest fixed points with post-fixed points.

The word invariant is sometimes used interchangably for sub-invariant.

Theorem 2.2.3 (Park's Induction [37]) *Let $(A, \preccurlyeq)$ be a complete lattice, $f\colon A \to A$ a monotonic function, $a \in A$ a pre-fixed point of f and $b \in A$ a post-fixed point of f. We have that*

$$\mathsf{lfp}(f) \preccurlyeq a \quad \textit{and} \quad b \preccurlyeq \mathsf{gfp}(f).$$

We have that for any fixed point c of f that $\mathsf{lfp}(f) \preccurlyeq c \preccurlyeq \mathsf{gfp}(f)$. This does not hold for pre-fixed and post-fixed points. A pre-fixed point may be greater than the greatest fixed point and a post-fixed point may be less than the least fixed point.

2.3. Folklore's and Cousot's Fixed Point Theorems

In this section, we will consider two theorems about defining least and greatest fixed points as the supremum or infimum of chains. Sometimes this is called a *constructive* approach to defining these fixed points. In practice, these are often useful as they allow connecting the fixed point to other concepts and to allow for induction on the chain's index.

We will use the constructive approach in Chapter 6 to prove the adequacy of the presented semantics.

A chain is usually considered a monotone or antitone function where the codomain is a linear order. This extracts a linear subset of a potential partial order. Moreover, chains are often especially considered monotone or antitone functions mapping from *natural numbers* to a partial order. However, for our second fixed

point theorem in this section, we will see chains which use *ordinals* as indices instead. We will call chains on natural numbers ω-chains and chains on ordinals ordinal-chains.

The first kind of chain we will consider is the iterated application of a function to some initial value.

Definition 2.3.1 (Natural Function Iteration) *Let $f\colon A \to A$ be a function. We define the natural function iteration for $n \in \mathbb{N}$ steps of f on $a \in A$ as*

$$f^n(a) \quad = \quad \begin{cases} a & \text{if } n = 0 \\ f(f^{n-1}(a)) & \text{else.} \end{cases}$$

If f is monotone and a is a post-fixed point of f, then the natural function iteration of f on a yields a (monotone) ω-chain. Dually, if f is monotone and a is a pre-fixed point of f, then the natural function iteration of f on a yields an (antitone) co-ω-chain. If the function satisfies a property called *ω-Scott-continuity*, then the supremum of the chain from a pre-fixed point yields the least fixed point greater than or equal to a and dually if the function is *co-ω-Scott-continuous*, the infimum of the chain from a post-fixed point yields the greatest fixed point less than or equal to a.

Definition 2.3.2 (Co- and ω-Scott-Continuity) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ a function.*
The function f is ω-Scott-continuous if and only if for any ω-chain, that is a monotone function chain: $\mathbb{N} \to A$, *we have*

$$\bigvee \{ f(\mathsf{chain}(i)) \mid i \in \mathbb{N} \} = f(\bigvee \{ \mathsf{chain}(i) \mid i \in \mathbb{N} \}).$$

The function f is co-ω-Scott-continuous if and only if for any ω-chain, that is an antitone function chain: $\mathbb{N} \to A$, *we have*

$$\bigwedge \{ f(\mathsf{chain}(i)) \mid i \in \mathbb{N} \} = f(\bigwedge \{ \mathsf{chain}(i) \mid i \in \mathbb{N} \}).$$

From ω-Scott-continuity and co-ω-Scott-continuity of a function f follows that f is monotone. As such, Knaster-Tarski's theorem applies as soon as the function is one of them. The fixed point theorem we present here is usually called *Kleene's fixed point theorem*. However, its origin is unclear [38]. As such we agree with Lassez, Nguyen and Sonenberg and consider it a folklore theorem.

Theorem 2.3.1 (The Folklore's Fixed Point Theorem [38]) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ be a function.*
If f is ω-Scott-continues, we have that

$$\mathsf{lfp}(f) \quad = \quad \bigvee \left\{ f^i(\bot) \,\middle|\, i \in \mathbb{N} \right\},$$

where $\bot$ is the least element in A.
If f is co-ω-Scott-continues, we have that

$$\mathsf{gfp}(f) \quad = \quad \bigwedge \left\{ f^i(\top) \,\middle|\, i \in \mathbb{N} \right\},$$

where $\top$ is the greatest element in A.

$f^i(\bot)$ is a monotone ω-chain and $f^i(\top)$ is an antitone ω-chain.

You may also want to reconsider the example from the introduction of this chapter with the new gained insights.

Example 2.3.1 We consider again the natural numbers with top element $(\mathbb{N}^\infty, \leq)$ and the function

$$f\colon \mathbb{N}^\infty \to \mathbb{N}^\infty, a \mapsto \text{if } (a \leq 5) \text{ then } 5 \text{ else } 6.$$

First we need to check whether f is ω-Scott-continuous and co-ω-Scott-continuous.
f is ω-Scott-continues as for any chain $\mathsf{chain}\colon \mathbb{N} \to \mathbb{N}^\infty$ we have that if $\sup\{\,\mathsf{chain}(i) \mid i \in \mathbb{N}\,\} \leq 5$ we have

$$\sup\{\, f(\mathsf{chain}(i)) \mid i \in \mathbb{N}\,\} = 5 = f(\sup\{\,\mathsf{chain}(i) \mid i \in \mathbb{N}\,\}),$$

and if $6 \leq \sup\{\,\mathsf{chain}(i) \mid i \in \mathbb{N}\,\}$ we have

$$\sup\{\, f(\mathsf{chain}(i)) \mid i \in \mathbb{N}\,\} = 6 = f(\sup\{\,\mathsf{chain}(i) \mid i \in \mathbb{N}\,\}).$$

For similar reasons, f is also co-ω-Scott-continues.
Using Folklore's fixed point theorem, we obtain that

$$\mathsf{lfp}(f) = \sup\left\{\, f^i(0) \,\middle|\, i \in \mathbb{N}\,\right\} = \sup\{\,0, 5\,\} = 5$$

as expected, and that

$$\mathsf{gfp}(f) = \inf\left\{\, f^i(\infty) \,\middle|\, i \in \mathbb{N}\,\right\} = \inf\{\,\infty, 6\,\} = 6,$$

also as expected.

Our next fixed point theorem requires a new type of indices. These are an extension to natural numbers, similar to how integers are an extension of natural numbers. Instead of continuing below zero, as we do for integers, we continue with a limit after all natural numbers. This new number, exceeding every natural number, is called ω. From there, we can again count upwards. These constitute the *ordinals.* There are many ways to construct ordinals.

In set theory, ordinals are usually constructed by first constructing natural numbers as sets. For this, we say that

$$0 = \emptyset \quad \text{and} \quad n + 1 = \{\, n \,\} \cup \bigcup n.$$

Using this rule, one can construct the numbers from 0 to 3 as

$$0 = \emptyset, \quad 1 = \{\,\emptyset\,\}, \quad 2 = \{\,\{\,\emptyset\,\}, \emptyset\,\}, \quad 3 = \{\,\{\,\{\,\emptyset\,\}, \emptyset\,\}, \{\,\emptyset\,\}, \emptyset\,\}.$$

This has the nice property that we can define the less-than-or-equal relation by the set membership relation. However, since we are in set theory we can also take the supremum — that is the union — of all natural numbers and obtain the ordinal ω.

In the most widely applied set theory ZFC, the collection of all ordinals is however not a set.

$$\omega = \bigcup\{\, n \mid n \in \mathbb{N}\,\} = \mathbb{N}.$$

From ω, we can again construct their successors $\omega + 1, \omega + 2, \ldots$ to obtain a new sequence of which we construct a new *limit ordinal.* This process may be repeated arbitrarily often.

Definition 2.3.3 (Ordinals) *We define the collection of ordinals, denoted as* Ordinal*, as the least collection such that*

$$\emptyset \in \mathsf{Ordinal}$$
$$\textit{if } a \in \mathsf{Ordinal} \textit{ then } \{a\} \cup \bigcup a \in \mathsf{Ordinal}$$
$$\textit{if } B \subset \mathsf{Ordinal} \textit{ then } \bigcup B \in \mathsf{Ordinal}$$

The order on ordinals is constructed using the "set membership" relation. Ordinals are special in that they generalize all *well orders*. A well order is a linear order for which all nonempty sets contain their infimum. This is equivalent to saying that the order has no infinitely strictly decreasing co-chain. Any well order is equivalent to an *initial segment* of the ordinals. An initial segment are all ordinals smaller than a certain ordinal. This also constitutes a second construction of the ordinals. Instead one may define the ordinals as the equivalence classes of well orders, where well orders are equivalent if there exists a bijection mapping ordered elements to equally ordered elements.

Lemma 2.3.2 (Partial and Well Order on Ordinals [39]) *The order* $(\mathsf{Ordinal}, \leq)$*, where*

$$a \leq b \iff a \in b \vee a = b,$$

is linear and for every non-empty set of ordinals, their infimum is contained.

We compare ordinals also by cardinality. The cardinality of an ordinal a is less than or equal to b if there exists an injection $\mathsf{inj}\colon a \to b$. Their cardinality is equal if there exists a bijection $\mathsf{bij}\colon a \to b$. A cardinal is the least ordinal with equal cardinality. Since the ordinals are well ordered, there always exists such a cardinal. We also have cardinality for other sets. Luckily, every set has a cardinality equal to some cardinal.

Definition 2.3.4 (Cardinals) *We say A has less or equal cardinality than A' if and only if there exists an injection* $\mathsf{inj}\colon A \to A'$*, and A and A' have the same cardinality if and only if there exists a bijection* $\mathsf{bij}\colon A \to A'$*.*
We define the collection of cardinals, denoted as Cardinal*, as the greatest collection such that for all* $a \in \mathsf{Cardinal}$ *and for all* $b \in \mathsf{Ordinal}$*, if a and b have same cardinality, then $a \leq b$.*
The successor of a cardinal a, denoted as $\mathsf{succ}(a)$*, is the smallest cardinal b greater than a.*
We say the cardinality of a set A, denoted as $\#A$, is the (unique) cardinal with cardinality equal to A.

We will use ordinals now as indices of chains, which approximate the least and greatest fixed point similar to Theorem 2.3.1. This time, however, we do not require continuity of the function. For this we introduce *upward* and *downard function iterations on ordinals*. The reason we now need two function iteration maps is because ordinals, in contrast to natural numbers, also feature limit numbers, for which we either take a supremum or an infimum respectively.

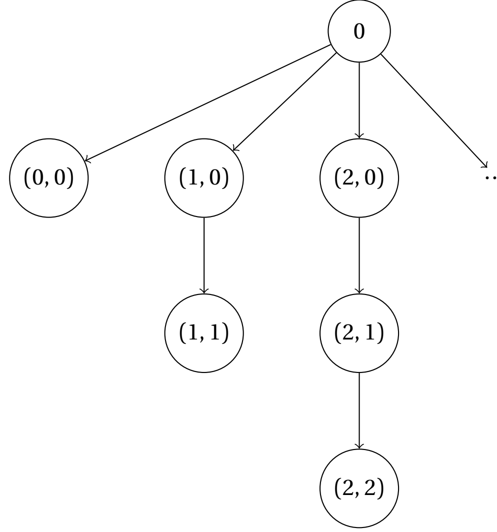


Figure 2.1.: Visualization of the counterexample from Example 2.3.2 depicting a transition system starting from a state **0** and branching infinitely to finite paths of arbitrary length.

Remark that $<$ is equivalent to $\in$ for ordinals and cardinals.

Definition 2.3.5 (Upward and Downward Function Iterations) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ be a monotone function.*
We define the upward function iteration of f as

$$\uparrow f^a \quad = \quad \sup \left\{ f(\uparrow f^b) \in A \,\middle|\, b < a \right\}.$$

We define the downward function iteration of f as

$$\downarrow f^a \quad = \quad \inf \left\{ f(\downarrow f^b) \in A \,\middle|\, b < a \right\}.$$

The upward and downward function iterations take the supremum of values indexes by smaller ordinals. Using this definition, we also have backed in a base case since for the least ordinal, the set is empty and thus we always start with the least and greatest element respectively. Using these new function iterations, we can express the generalization for the Folklore's fixed point theorem to the one shaped by Cousot and Cousot.

Theorem 2.3.3 (Cousot's Fixed Point Theorem [40]) *Let $(A, \preccurlyeq)$ be a complete lattice and $f\colon A \to A$ be a monotone function.*
The least fixed point of f is

$$\mathsf{lfp}(f) \quad = \quad \uparrow f^{\mathsf{succ}(\#A)},$$

and the greatest fixed point if f is

$$\mathsf{gfp}(f) \quad = \quad \downarrow f^{\mathsf{succ}(\#A)}.$$

With $\mathsf{Pow}(\Sigma)$ we denote the powerset of Σ.

Example 2.3.2 Let $(\Sigma, \to)$ be a transition system. We have that $(\mathsf{Pow}(\Sigma), \subseteq)$ is a complete lattice. Let further $\mathsf{allstep}\colon \mathsf{Pow}(\Sigma) \to \mathsf{Pow}(\Sigma)$ be the function that maps a set of goal states to the set of states which can reach only goal states in one step, i.e. $\sigma \in \mathsf{allstep}(B)$ if and only if for all states $\sigma' \in \Sigma$ with

$\sigma \to \sigma'$, we have $\sigma' \in B$.
allstep is not ω-Scott-continues for all transition systems. For a counterexample consider the transition system defined as

$$\Sigma = \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\} \cup \{\,0\,\},$$
$$\forall i \in \mathbb{N}.\, 0 \to (i,0) \quad \text{and} \quad \forall i \in \mathbb{N}.\, \forall j < i.\, (i,j) \to (i,j+1)$$

and visualized in Figure 2.1.
Now we consider the chain

$$\mathsf{chain}\colon \mathbb{N} \to \mathrm{Pow}(\mathbb{N}),\, n \to \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i \leq n\,\},$$

for which the supremum

$$\bigcup \{\,\mathsf{chain}(i) \mid i \in \mathbb{N}\,\} = \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\}$$

are all tuples, but not the 0 state. When now applying allstep to each element of the chain, we obtain

$$\mathsf{allstep}(\mathsf{chain}(n)) = \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i \leq n\,\} \cup \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid i = j\,\},$$

as states without any outgoing transitions are always included in allstep and other only have one transition going to a goal state. The supremum of all of these are all tuples, but not the 0 state, i.e.

$$\bigcup \{\,\mathsf{allstep}(\mathsf{chain}(n)) \mid n \in \mathbb{N}\,\} = \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\}.$$

But when we apply allstep on the supremum of the chain, we obtain

$$\mathsf{allstep}(\{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\}) = \{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\} \cup \{\,0\,\},$$

as all successor states of 0 are contained in $\{\,(i,j) \in \mathbb{N} \times \mathbb{N} \mid j \leq i\,\}$. Since

$$\bigcup \{\,\mathsf{allstep}(\mathsf{chain}(n)) \mid n \in \mathbb{N}\,\} \neq \mathsf{allstep}(\bigcup \{\,\mathsf{chain}(n) \mid n \in \mathbb{N}\,\}),$$

the function allstep is not ω-Scott-continues.
However, allstep is monotone. For this, consider $B \subseteq B'$. We now prove that if $\sigma \in \mathsf{allstep}(B)$ then also $\sigma \in \mathsf{allstep}(B')$. Since we have $\sigma \in \mathsf{allstep}(B)$, we also have that for all states $\sigma' \in \Sigma$ with $\sigma \to \sigma'$, we have $\sigma' \in B$. Since $B \subseteq B'$, we have $\sigma' \in B'$ and thus also $\sigma \in \mathsf{allstep}(B')$.
Since allstep is monotone, it has a least fixed point. We can approximate the least fixed point using Theorem 2.3.3. Let us find the least fixed point of the before mentioned transition system using this fixed point theorem. For this, we need to find the value of $\uparrow\mathsf{allstep}^{\mathsf{succ}(\#\mathbb{N})}$. Assuming the continuum hypothesis, we have that the least fixed point is $\uparrow\mathsf{allstep}^{\#\mathbb{R}}$. To not be irritated about the reals popping up, we will use the standard notation $\aleph_1 = \#\mathbb{R}$ for the cardinality of the reals. We thus now need to find the value of $\uparrow\mathsf{allstep}^{\aleph_1}$. We have for $n \in \mathbb{N}$ that

$$\uparrow\mathsf{allstep}^{n} = \{\,(i,j) \in \mathbb{N} \mid 0 \leq i - j < n\,\}.$$

Taking the supremum of all these, we obtain the value for the ordinal ω, i.e. we have

$$\uparrow\mathsf{allstep}^{\omega} = \bigcup \{\,\uparrow\mathsf{allstep}^{n} \mid n \in \mathbb{N}\,\} = \{\,(i,j) \in \mathbb{N} \mid j \leq i\,\}.$$

The famous but unprovable continuum hypothesis states that $\mathsf{succ}(\#\mathbb{N}) = \#\mathbb{R}$.

When we proceed with one more step, i.e. to $\omega + 1$ we finally have that

$$\uparrow\mathsf{allstep}^{\omega+1} = \mathsf{allstep}(\{\,(i,j) \in \mathbb{N} \mid j \leq i\,\}) = \Sigma,$$

and since Σ is also the greatest element, it necessarily is the least and greatest (and thus unique) fixed point of allstep. Since $\uparrow\mathsf{allstep}^{\alpha}$ stabilizes at the fixed point, we also have that

$$\uparrow\mathsf{allstep}^{\aleph_1} = \uparrow\mathsf{allstep}^{\omega+1} = \Sigma.$$

Separation Logic in Program Verification

3. Concurrent Separation Logic

For didactic reasons and in order to increase complexity slowly we will first investigate reasoning about *non-probabilistic concurrent programs*. This has the benefit that we can first concentrate on separation logic (SL) as a *propositional logic* and later deal with *quantitative separation logic* and *fuzzy separation logic*. We will here explain a version of concurrent separation logic that is similar to the original version from O'Hearn in 2004 [25] and formalized by Vafeiadis in 2011 [26] in Isabelle and later in Coq. This formalization is very different from the formalization of modern logics such as *Iris* [41].

The basic idea of concurrent separation logic is to divide memory in form of the *heap* (and read here especially data-structures and their data) in smaller heaps for every thread of the concurrent program. This allows every thread to have exclusive access to "its heap", but also limits any communication between threads to shared parameters. In order to relax this restriction, we also allow a heap that is used for shared memory over which threads can communicate.

In order to reason about multiple executions, we abstract sets of heaps by logical propositions in separation logic. We do not differentiate between formulae in separation logic and their semantics. Instead we call the composition of logical operations propositions as it is custom when dealing with theorem provers. For sake of concreteness, let us consider a simple example.

If you struggle with the syntax of this program, you may like to peak in the description of this programming syntax in Section 3.2. We generally follow here the syntax of WHILE-languages with a basic feature set for our purpose.

$$\texttt{<}r\texttt{>} := -1;$$
$$\texttt{<}r\texttt{>} := 0 \quad \left\| \begin{array}{l} x := \texttt{<}r\texttt{>}; \\ \texttt{while}(x = -1)\{x := \texttt{<}r\texttt{>}\}; \end{array} \right.$$

This program implements a simple busy-waiting communication between two threads. We first set the value at the heap location r to -1. We thus assume that the heap is initially at least allocated at location r. Next we spawn two threads. The left thread only manipulates the value in location r to 0. The right thread reads the value at location r in the variable x and waits for the initial value at r to change.

Indeed, the challenge of the above program lies in the fact that the value at location r *may* change, but we do not know when it changes. Nevertheless, we try now to solve this by defining propositions that describe the state of the memory between every two statements. Let us start with the initial state. It does not actually matter which value the variables x and r have, it only matters that the heap is allocated at location r. Next we change the value at location r. The proposition thus changes to $r \mapsto -1$. Now we split the heap in three parts: one part for the left thread, one part for the right thread and one part for both threads. Since the heap only contains this allocated location r and this is used by both threads, we give the threads the *empty heap*, which we capture by the predicate emp. We denote this splitting operation as the proposition $\mathsf{emp} * \mathsf{emp} * r \mapsto -1$. It means: I can split the heap in three disjoint parts. One is empty, the next is also empty and the last one only contains the location r with value -1.

We will introduce the notation mentioned here such as $r \mapsto -1$ and $*$ in all necessary detail in Section 3.1. If this explanation leaves you confused, you may want to come back to it later.

We can now proceed by considering every thread on its own. The left thread changes the value of the heap at location r to 0. Thus we change the shared memory proposition to $r \mapsto 0$. For the right thread we now have a problem.

We have two kinds of propositions that describe the shared memory, the initial one $r \mapsto -1$ and $r \mapsto 0$, which we obtain after executing the left thread. Indeed, if we (falsely) considered only one of these, we get a wrong result. Choosing $r \mapsto -1$ yields that the left thread will never (under *every thread scheduling*) finish. Choosing $r \mapsto 0$ yields that the left thread always (again under *every scheduling*) terminates. However, both answers are wrong. If we let the left thread (unfairly) starve, we will never leave the loop in the right thread. If we let the left thread execute, the right thread will terminate.

The idea to fix this, is to establish an *invariant* that holds for both threads. We thus need: (1) change the proposition for the shared memory to the invariant $r \mapsto -1 \vee r \mapsto 0$ and (2) guarantee its invariance, i.e. that no manipulation of the heap invalidates the invariant. Luckily, this proposition also holds between every two program statements. This new invariant also allows us to see the possibility for both, to never terminate and to actually terminate. Although we can now see the possibility of both the terminating and the non-terminating scenario ad-hoc, the soundness theorems for this framework unfortunately can not differentiate between successful execution and non-termination and therefore reasons about *partial-correctness.* A fully annotated program with concurrent separation logic annotations is the following:

In this notation, a proposition behind $/\!\!/$ means that — given that the initial state satisfies the first condition — everytime the execution reaches that position in the program, the state of the program satisfies this proposition. Later we will see a different interpretation of this notation, that is more useful in the probabilistic case.

$$
\begin{array}{l}
/\!\!/ \; \exists a.\, r \mapsto a \\
\texttt{<}r\texttt{>} \; \texttt{:=} \; -1\texttt{;} \\
/\!\!/ \; r \mapsto -1 \\
/\!\!/ \; \mathsf{emp} * \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0) \\
\left.\begin{array}{l}
/\!\!/ \; \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0) \\
\texttt{<}r\texttt{>} \; \texttt{:=} \; 0 \\
/\!\!/ \; \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0)
\end{array}\right\| \begin{array}{l}
/\!\!/ \; \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0) \\
x \; \texttt{:=} \; \texttt{<}r\texttt{>;} \\
/\!\!/ \; \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0) \\
\texttt{while(}x = -1\texttt{)\{}x \; \texttt{:=} \; \texttt{<}r\texttt{>\};} \\
/\!\!/ \; \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0)
\end{array} \\
/\!\!/ \; \mathsf{emp} * \mathsf{emp} * (r \mapsto -1 \vee r \mapsto 0)
\end{array}
$$

The annotations over-approximates the possible memory configuration as the value at location r is never -1 after termination when executing the program. We will not go into details how to circumvent this problem (but see for example [41, 42]) and instead concentrate on the foundations.

We call classical separation logic here qualitative in order to differentiate it from the two different versions we will see later. Usually, we drop the descriptor qualitative, but not the descriptors quantitative and fuzzy.

3.1. Separation Logic

(Qualitative) separation logic is an extension of first order logic introducing the separating operator. In our setting, we will also consider the magic wand, which is the adjoint operation of the separating operator.

The separating operator — in the setting of qualitative logic we call it the separating conjunction — takes an object of the domain and asserts that it can be split in two disjoint objects such that they satisfy the left and right side of the operator respectively. In our setting, the objects of interest are heaps, that is partial functions mapping from locations to values. Disjointness of heaps refers to the disjointness of the domains of the heaps. Other works also include the assumption that heaps are finite. We drop this to ease the proofs and reintroduce

finiteness when convenient. Usually, this requirement is used to guarantee that the memory allocation never fails. However, we assume instead that memory allocation deadlocks when the heap has no free locations remaining instead of failing and thus make our lives a bit easier by not assuming the defined domain of heaps to be finite. We model the partiality of our heap by allowing two kinds of values for heaps: (1) actual rational values that are stored and (2) undefined values which are open for allocation. Our union prioritizes the left side. That is, if both sides are allocated at a location, we take the value on the left.

Definition 3.1.1 (Heaps) *A heap $h : \mathbb{N}_{>0} \to \mathbb{Q} \cup \{\mathit{undef}\}$ is a partial mapping from locations to values. We denote an undefined value at location ℓ in a heap h with $h(\ell) = \mathit{undef}$. We denote the set of all heaps as* Heaps. *We denote the heap where every location is undef as $\emptyset$.*

We adopt a set notation to denote defined mappings in the heap. That is, the set $\{1 \mapsto 0, 2 \mapsto 1\}$ denotes the heap h with $h(1) = 0, h(2) = 1$ and $h(a) = \mathrm{undef}$ else.

For the values of program variables, we use a function mapping program variables to their value. We consider the initial values of the program variables as inputs to the program. We call these functions *stacks*. When we define our logic in some meta logic (like Lean), we will generally be able to use variables that are connected to this meta logic and not the program. Formalizations of separation logic not using this kind of notation of meta logic usually encode variables that are bound by quantifiers as stack variables. We will differ here and explicitly use logical variables to differentiate program variables and logical variables. Logical variables yield their value by the function η mapping logical variables from LVars to values. Since the logical variables are borrowed from the meta logic, we generally have the expressivity of the meta logic here.

Definition 3.1.2 (Stacks, Program States and Logical Variables) *Let* Vars *be a (possibly infinite) set of program variables.*

- *A stack s:* Vars $\to \mathbb{Q}$ *is a mapping from variables to values. We denote the set of all stacks as* Stacks.
- *We call a program state $\sigma \in$* Stacks × Heaps *a stack heap pair. We denote the set of all stack heap pairs* States.
- *Let* LVars *be a set of logical variables. The function η:* LVars $\to \mathbb{Q}$ *maps logical variables to values. We denote the set of all functions mapping logical variables to values as* LStacks.

For simplicity, we assume that locations are positive natural numbers and values are rational numbers. In programs, we use expressions which we abstract as total functions from stacks to some value. We only let the set of program and logical variables be generic. Logical variables are used to represent variables that are bound by quantifiers. When defining a quantifier, we will need to change the value of certain variables. We call this operation a *substitution*.

We use lambda expressions such as $\lambda a. \ldots$ for an anonymous function with one argument named a.

Definition 3.1.3 (Substitution) *The substitution of a variable $a \in \mathsf{LVars}$ by the value $v \in \mathsf{LVars}$ in $\eta \in \mathsf{LStacks}$ is*

$$\eta\,[a := v] \quad = \quad \lambda b. \begin{cases} v & \text{if } b = a \\ \eta(b) & \text{else} \end{cases}$$

and analogous for $x \in \mathsf{Vars}$, $s \in \mathsf{Stacks}$ and $s\,[x := v]$ as well as for $\ell \in \mathbb{N}_{>0}$, $h \in \mathsf{Heaps}$ and $h\,[\ell := v]$.

Disjointness of two heaps is defined by requiring that for any location at least one heap is undefined. We define the union as left-prioritizing. That is, a location maps to undefined if both heaps are undefined at that location, maps to the value of the defined location if only one heap is defined at that location and maps to the left value if both locations are defined.

Definition 3.1.4 (Disjointness and Union) *Let $h_1, h_2 \in \mathsf{Heaps}$. We say h_1 and h_2 are disjoint, written $h_1 \perp h_2$, if for all $\ell \in \mathbb{N}_{>0}$ we have $h_1(\ell) = undef$ or $h_2(\ell) = undef$. The (left-prioritizing) union of h_1 and h_2 is defined as*

$$h_1 \cup h_2 \quad = \quad \lambda \ell. \begin{cases} undef & \text{if } h_1(\ell) = undef \text{ and } h_2(\ell) = undef \\ h_1(\ell) & \text{if } h_1(\ell) \neq undef \\ h_2(\ell) & \text{if } h_1(\ell) = undef \text{ and } h_2(\ell) \neq undef \end{cases}$$

Lemma 3.1.1 *The following statements hold for $h_1, hh_2, h_3 \in \mathsf{Heaps}$:*

$$h_1 \perp h_2 \iff h_2 \perp h_1 \qquad \text{(Symmetry of Disjointness)}$$
$$h_1 \cup (h_2 \cup h_3) = (h_1 \cup h_2) \cup h_3 \qquad \text{(Associativity of Union)}$$
$$\text{if } h_1 \perp h_2 \text{ then } h_1 \cup h_2 = h_2 \cup h_1 \qquad \text{(Commutativity of Union)}$$

Proof. See [43] at `LeanFSL.Program.State`. ☐

Using disjointness and union of heaps, we can introduce a partial ordering on heaps that orders two heaps if the lesser has a subset of defined locations and they agree on the values of the subset of locations.

Definition 3.1.5 (Subset on Heaps) *Let $h_1, h_2 \in \mathsf{Heaps}$ be heaps. We say that hh_1 is a subset of h_2, written as $h_1 \subseteq h_2$ if*

$$\exists h' \in \mathsf{Heaps}.\ h_1 \perp h' \wedge h_2 = h_1 \cup h'$$

Lemma 3.1.2 (Partial Order on Heaps) *The relation $h \subseteq h'$ is a partial order.*

Proof. See [43] at `LeanFSL.Program.State`. ☐

Statements in separation logic (SL) are maps from states together with values for values for logical variables to truth and false values. We take a model theoretical

Table 3.1.: This is a list for the semantics for various operations in qualitative separation logic. (s, h) is a stack heap pair. For the expressions e and e' of the atoms we will not give semantics for brevity. We also have first order connectives and the two separation operators the separating conjunction and the qualitative magic wand. $\Theta\colon \mathbb{N} \to \mathsf{SLProp}$ is a map from natural numbers to SL propositions.

Φ	$(s,h) \models_\eta \Phi$
true	always holds
false	never holds
emp	$h = \emptyset$
$e \mapsto e'$	$h(e(\eta,s)) = e'(\eta,s)$ and $h(\ell) = \mathrm{undef}$ for $\ell \neq e(\eta,s)$
$e = e'$	$e(\eta,s) = e'(\eta,s)$
$\Phi\,[x := e]$	$(s\,[x := e(s)]\,, h) \models_\eta \Phi$
$\neg\Psi$	not $(s,h) \models_\eta \Psi$
$\Psi_1 \land \Psi_2$	$(s,h) \models_\eta \Psi_1$ and $(s,h) \models_\eta \Psi_2$
$\Psi_1 \lor \Psi_2$	$(s,h) \models_\eta \Psi_1$ or $(s,h) \models_\eta \Psi_2$
$\exists a.\,\Psi$	exists $v \in \mathbb{Q}$ with $(s,h) \models_{\eta[a:=v]} \Psi$
$\forall a.\,\Psi$	for all $v \in \mathbb{Q}$ we have $(s,h) \models_{\eta[a:=v]} \Psi$
$\Psi_1 * \Psi_2$	exists h_1, h_2 with $h_1 \perp h_2$, $h_1 \cup h_2 = h$, $(s,h_1) \models_\eta \Psi_1$ and $(s,h_2) \models_\eta \Psi_2$
$\mathop{\Huge *}_{i=0}^{n} \Theta$	$\begin{cases} \mathsf{emp} & \text{if } n = -1 \\ \Theta(n) * \mathop{*}_{i=0}^{n-1} \Theta & \text{if } n > -1 \end{cases}$
$\Psi_1 \mathrel{-\!\!*} \Psi_2$	for all h' with $(s,h') \models_\eta \Psi_1$ and $h \perp h'$ we have $(s, h \cup h') \models_\eta \Psi_2$

view on these statements, which we will call propositions. The usage of propositions originally come from type theory, where the type proposition is used to express logical statements. In type theory, this is useful as these propositions can now be a type themselves and have proofs as elements if they hold. Since this thesis is partly formalized in the dependent type theory Lean, we will use this nomenclature and call statements in Separating Logic SLProp.

Definition 3.1.6 (Propositions in Separation Logic) *A proposition in separation logic is a map* $\Phi\colon \mathsf{States} \times \mathsf{LStacks} \to \{\mathsf{true}, \mathsf{false}\}$. *We call the set of all propositions in separation logic* SLProp. *We write* $(s,h) \models_\eta \Phi$ *if* Φ *yields true for* (s,h) *and* η.

We now have all ingredients for defining the key operator in separation logic: the separating conjunction. Given a stack s and a heap h, a proposition with the separating conjunction $\Phi * \Psi$ requires that h can be split in two disjoint heaps h_1, h_2, i.e. that $h_1 \perp h_2$, such that their union is again h, i.e. $h = h_1 \cup h_2$ and such that the sub heaps satisfy the left and right side respectively, i.e. $(s,h_1) \models_\eta \Phi$ and $(s,h_2) \models_\eta \Psi$.

The magic wand $\mathrel{-\!\!*}$ is the adjoint operator of the separating conjunction. That is, $\Phi_1 * \Phi_2$ implies Φ_3 if and only if Φ_1 implies $\Phi_2 \mathrel{-\!\!*} \Phi_3$. For a stack heap pair (s,h) to satisfy the magic wand $\Phi \mathrel{-\!\!*} \Psi$, we require for all heaps h' which are disjoint to h, i.e. $h \perp h'$, and satisfy the left side of the proposition, i.e. $(s,h') \models_\eta \Phi$, that the union of both satisfy the right side, i.e. $(s, h \cup h') \models_\eta \Psi$. A comprehensive list of atoms, first order connectives and the separation logic connectives can be found in Table 3.1. For the atoms, we omit semantics for their expressions. These expressions include arithmetic operations, variables and rounding operators.

Here the word "implies" refers to the entailment between separation logic propositions.

In order to avoid many brackets in our propositions, we will use traditional precedence rules for first order operations. Separating conjunctions bind the same as regular conjunctions and the magic wand bounds the same as regular implication. That is, negation binds the strongest, separating conjunction and regular conjunction bind equally strong and are right-associative, conjunction binds stronger than disjunction, disjunction binds stronger than magic wand and quantifiers bind the strongest. Thus the following propositions are the same:

$$\begin{aligned} &\forall a.\, \mathsf{emp} * \mathsf{emp} \wedge \mathsf{emp} \mathrel{-\!\!*} \mathsf{emp} \mathrel{-\!\!*} \mathsf{emp} \\ =\ &\forall a.\, ((\mathsf{emp} * (\mathsf{emp} \wedge \mathsf{emp})) \mathrel{-\!\!*} (\mathsf{emp} \mathrel{-\!\!*} \mathsf{emp})) \end{aligned}$$

We need to be careful when using both regular conjunction and separating conjunction, as they do not commute, but have the same precedence. We will stick with this precedence for the other versions of separation logic as well.

Example 3.1.1 Let $h = \{1 \mapsto 2, 2 \mapsto 3\}$ be a heap. Furthermore we let $s(x) = 1$ for all x. We want to check whether this holds:

$$(s, h) \models_\eta x \mapsto 2 * 2 \mapsto 3$$

- We have that $(s, h) \models_\eta x \mapsto 2 * 2 \mapsto 3$ as we can split h into the heap $h_1 = \{1 \mapsto 2\}$ and the heap $h_2 = \{2 \mapsto 3\}$. We also have that $h_1 \perp h_2$ and $h = h_1 \cup h_2$.
- Next, we check that $(s, h_1) \models_\eta x \mapsto 2$. Since $s(x) = 1$ we need to check whether $h_1 = \{1 \mapsto 2\}$. Indeed, this is the case.
- Lastly, we check that $(s, h_2) \models_\eta 2 \mapsto 3$. For this we check that $h_2 = \{2 \mapsto 3\}$. Indeed, this is the case.

We shorten the expression $\lambda(s,\eta).\, s(x) \mapsto \lambda(s,\eta).\, 2$ here and elsewhere as the notation $x \mapsto 2$.

Example 3.1.2 Let $h = \{1 \mapsto 2\}$ be a heap. Furthermore we let $s(x) = 1$ for all x. We want to check whether this holds:

$$(s, h) \models_\eta 2 \mapsto 3 \mathrel{-\!\!*} (1 \mapsto 2 * 2 \mapsto 3)$$

- We have that h' with $h' = \{2 \mapsto 3\}$ is disjoint with h, i.e. $h \perp h'$. We will prove that h' is the only heap satisfying the left side of the magic wand.
- For $(s, h') \models_\eta 2 \mapsto 3$ we need to check that h' is the only heap with $h' = \{2 \mapsto 3\}$, which is the case.
- We have that $(s, h \cup h') \models_\eta 1 \mapsto 2 * 2 \mapsto 3$ holds because of Example 3.1.1.

Commonly, we are interested in entailments between propositions. An entailment encodes that for a stack heap pair and values for logical variables, given the proposition on the left side satisfies these, then also the proposition on the right side satisfies these. This allows us to prove and conclude theorems given various assumptions and is thus a key ingredient for logical reasoning.

Definition 3.1.7 (Qualitative Entailments) *Let $\Phi, \Psi \in \mathsf{SLProp}$ be SL propositions. We define the entailment between these as*

$$\Phi \models \Psi \qquad \textit{iff} \qquad \textit{for all } (s, h), \eta, \textit{ we have if } (s, h) \models_\eta \Phi \textit{ then } (s, h) \models_\eta \Psi.$$

We have commutativity and associativity for separating conjunction. This justi-

fies leaving out brackets around separating conjunctions — we still stick with the convention that separating conjunction and regular conjunction bind equally, even though they are not commutativity and associativity with each other.

Theorem 3.1.3 (Commutativity and Associativity) *Let* $\Phi_1, \Phi_2, \Phi_3 \in \mathsf{SLProp}$ *be SL propositions. The following statements hold:*

$$\Phi_1 * \Phi_2 \vDash \Phi_2 * \Phi_1 \qquad \text{(Commutativity)}$$
$$\Phi_1 * (\Phi_2 * \Phi_3) \vDash (\Phi_1 * \Phi_2) * \Phi_3 \qquad \text{(Associativity)}$$

Proof. See [43] at `LeanFSL.SL.ClassicalProofrules`. □

A key theorem of the magic wand in separation logic is called adjointness. This property allows us to introduce the left side of a magic wand on the right side of an entailment into a separating conjunction on the left side of an entailment. Another important property is modus ponens, which allows us to eliminate magic wands on the left side of an entailment, given that the left side includes the antecedent. Transforming magic wands into propositions without the magic wand is in praxis how we deal with it. Indeed, this is not surprising as this is also how we deal with regular implication. The only difference is that in the separating version of modus ponens, we loose that the antecedent still holds. Lastly we also have monotonicity of the separating conjunction.

The antecedent is the left side of an implication.

Theorem 3.1.4 (Adjointness and Modus Ponens) *Let* $\Phi_1, \Phi_2, \Phi_3, \Phi_4 \in \mathsf{SLProp}$ *be SL propositions. The following statements hold:*

$$\Phi_1 \vDash \Phi_2 \quad \textit{and} \quad \Phi_3 \vDash \Phi_4 \quad \textit{then} \quad \Phi_1 * \Phi_3 \vDash \Phi_2 * \Phi_4 \qquad \text{(Monotonicity)}$$
$$\Phi_1 * \Phi_2 \vDash \Phi_3 \quad \textit{iff} \quad \Phi_1 \vDash \Phi_2 \mathrel{-\!\!*} \Phi_3 \qquad \text{(Adjointness)}$$
$$\Phi_1 * (\Phi_1 \mathrel{-\!\!*} \Phi_2) \vDash \Phi_2 \qquad \text{(Modus Ponens)}$$

Proof. See [43] at `LeanFSL.SL.ClassicalProofrules`. □

3.2. Syntax and Operational Semantics of the Programming Language

We will now formally define the syntax of a concurrent programming language supporting heap manipulation (but not probabilistic execution). This language supports assignments to variables, manipulation of a heap, including allocation and deallocation, conditional branching, loops and concurrent execution. We will define an operational semantics for this programming language, on which we base the soundness of our axiomatic semantics. We leave the syntax for expressions open and instead use appropriate functions, which we call expressions. We will abuse notation when dealing with expressions. That is, we write in expressions x instead of $s(x)$ and do not use lambda functions to explicitly state them as anonymous functions. The syntax of the programming language is defined as a context-free language in *Backus-Naur form*.

Definition 3.2.1 (Concurrent Programming Language) *We define the concurrent programming language* cPL *as the language generated by this grammar:*

$\mathscr{C}$	::=	$\texttt{skip}$	*empty program*
	|	$x \texttt{ := } e$	*assignment*
	|	$\texttt{<}e\texttt{>} \texttt{ := } e'$	*heap mutation*
	|	$x \texttt{ := } \texttt{<}e\texttt{>}$	*heap lookup*
	|	$x \texttt{ := CAS}(e_1, e_2, e_3)$	*compare and set*
	|	$x \texttt{ := new}(e)$	*allocation*
	|	$\texttt{free}(e, e')$	*deallocation*
	|	$\texttt{if}\ (e_b)\ \{\mathscr{C}\}\ \texttt{else}\ \{\mathscr{C}\}$	*conditional branching*
	|	$\texttt{while}(e_b)\{\mathscr{C}\}$	*looping*
	|	$\mathscr{C} ; \mathscr{C}$	*program composition*
	|	$\mathscr{C} \parallel \mathscr{C}$	*concurrent execution*

where $e, e', e_1, e_2, e_3 : \mathsf{Stacks} \times \mathsf{LStacks} \to \mathbb{Q}$ *are value expression and* $e_b : \mathsf{Stacks} \times \mathsf{LStacks} \to \{\mathsf{true}, \mathsf{false}\}$ *are Boolean expressions. We denote the set of value expressions as* ValExpr *and the set of Boolean expressions as* BoolExpr*. We use* $\downarrow$ *to depict terminated programs and* $\lightning$ *to depict erroneous programs.*

Both $\downarrow$ and $\lightning$ are deadlocking statements. When reaching, the program does not further execute. The difference is their interpretation. A program reaching $\downarrow$ has successfully terminated, but a program reaching $\lightning$ has unsuccessfully terminated — you may say the program crashed. We will usually not write actual code with these constructs. Instead, we use these to define program transformations enabling operational semantics. The programs $\texttt{skip}$, $x \texttt{ := } e$, $\texttt{<}e\texttt{>} \texttt{ := } e'$, $x \texttt{ := } \texttt{<}e\texttt{>}$, $x \texttt{ := CAS}(e_1, e_2, e_3)$, $x \texttt{ := new}(e)$ and $\texttt{free}(e, e')$ terminate after one step. $\texttt{skip}$ does nothing in this one step. $x \texttt{ := } e$ changes the value of x to the value after evaluating e on the programs state. $\texttt{<}e\texttt{>} \texttt{ := } e'$ changes the value at the location to which e evaluates on the current program state to the value which e' evaluates on the current program state. $x \texttt{ := } \texttt{<}e\texttt{>}$ looks up the value at the location to which e evaluates in the current program state and assigns x this value. $x \texttt{ := CAS}(e_1, e_2, e_3)$ compares the value of e_2 with the value at the location e_1. If these values coincide, it assigns 1 to x and changes the value at location e_1 to e_3, else it assigns 0 to x. All of this happens atomically. Compare and set is crucial to implement mutual exclusion locks in our programming language. Allocation $x \texttt{ := new}(e)$ takes an arbitrary group of e free consecutive locations, assigns them 0 as initial value and returns the starting location to x. If e does not evaluate to a natural number, the program crashes. The program $\texttt{free}(e, e')$ does the opposite and sets the values of the locations $e + 0, \ldots, e + (e' - 1)$ to undef if they were all defined. All programs accessing the heap and finding an undef location when they expected an allocated one will crash. The program constructs $\texttt{if}\ (e_b)\ \{C_1\}\ \texttt{else}\ \{C_2\}$, $\texttt{while}(e_b)\{C\}$, $C_1 ; C_2$ and $C_1 \parallel C_2$ allow controlling the flow of the atomic programs from before. The conditional branching statement $\texttt{if}\ (e_b)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ evaluates the Boolean condition e_b. If it is evaluated to true, we execute C_1, if it is evaluated to false, we execute C_2. The looping statement $\texttt{while}(e_b)\{C\}$ executes C as long as e_b evaluates to true and terminates otherwise. Sequential composition $C_1 ; C_2$ executes first C_1 and then C_2 provided C_1 successfully terminates. Finally, the concurrent execution $C_1 \parallel C_2$ interleaves the execution of C_1 and C_2 in some arbitrary way. If one part of the program crashes, we immediately terminate the

Compare and set is sometimes also referred to as compare and swap with same or similar semantics. Arguably, compare and set is a better name for that operation.

$$\frac{}{\texttt{skip}, (s,h) \to \downarrow, (s,h)}\ \text{SKIP} \qquad \frac{}{x := e, (s,h) \to \downarrow, (s[x := e(s)], h)}\ \text{ASSIGN}$$

$$\frac{e(s) \in \mathsf{dom}(h)}{x := \texttt{<}e\texttt{>}, (s,h) \to \downarrow, (s[x := h(e(s))], h)}\ \text{LOOKUP} \qquad \frac{e(s) \notin \mathsf{dom}(h)}{x := \texttt{<}e\texttt{>}, (s,h) \to \lightning, (s,h)}\ \text{LOOKUP-ABT}$$

$$\frac{e(s) \in \mathsf{dom}(h)}{\texttt{<}e\texttt{>} := e', (s,h) \to \downarrow, (s, h[e(s) := e'(s)])}\ \text{MUT} \qquad \frac{e(s) \notin \mathsf{dom}(h)}{\texttt{<}e\texttt{>} := e', (s,h) \to \lightning, (s,h)}\ \text{MUT-ABT}$$

$$\frac{e_1(s) \in \mathsf{dom}(h) \qquad h(e_1) \neq e_2(s)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \to \downarrow, (s[x := 0], h)}\ \text{CAS-FALSE}$$

$$\frac{e_1(s) \in \mathsf{dom}(h) \qquad h(e_1) = e_2(s)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \to \downarrow, (s[x := 1], h[e_1(s) := e_3(s)])}\ \text{CAS-TRUE}$$

$$\frac{e_1(s) \notin \mathsf{dom}(h)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \to \lightning, (s,h)}\ \text{CAS-ABT}$$

$$\frac{e(s) = n \in \mathbb{N} \qquad \ell, \ldots, \ell + n - 1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(h) \qquad h' = h \cup \{\ell \mapsto 0, \ldots, \ell + n - 1 \mapsto 0\}}{x := \texttt{new}(e), (s,h) \to \downarrow, (s[x := \ell], h')}\ \text{ALLOC}$$

$$\frac{e(s) \notin \mathbb{N}}{x := \texttt{new}(e), (s,h) \to \lightning, (s,h)}\ \text{ALLOC-ABT}$$

$$\frac{e(s) + 0, \ldots e(s) + e'(s) - 1 \in \mathsf{dom}(h) \qquad h' = h[e(s) + 0 := \mathrm{undef}] \ldots [e(s) + e'(s) - 1 := \mathrm{undef}]}{\texttt{free}(e, e'), (s,h) \to \downarrow, (s, h')}\ \text{FREE}$$

$$\frac{\exists i \in \{0, \ldots, e'(s) - 1\}.\ e(s) + i \notin \mathsf{dom}(h)}{\texttt{free}(e,,)(s,h) \to \lightning, (s,h)}\ \text{FREE-ABT}$$

Figure 3.1.: Operational semantics for all statements in cPL. Heap manipulating statements may abort due to the accessed location not being allocated. Compare and set changes the value at a certain location only if the value is as expected.

execution in the failing state $\lightning$.

Instead of a compare and set statement as we introduced it here, we could also have supported atomic regions. These are sub-programs which are guaranteed to run in one execution steps. Indeed, this is the solution of [26]. We do not use arbitrary atomic regions as it increases complexity. Many programming languages usually only offer a limited number of atomic statements instead of general atomic regions anyway. In principle, adding atomic regions is possible — even for probabilistic programs where atomic programs are not probabilistic [33].

Example 3.2.1 We will consider the following example as a running example throughout this chapter:

$$\begin{array}{l} r := \texttt{new}(1)\,; \\ \texttt{<}r\texttt{>} := 1 \;\Big\|\; \begin{array}{l} x := 0; \\ \texttt{while}(x = 0)\{x := \texttt{<}r\texttt{>}\} \end{array} \end{array}$$

The program first allocates r with a size of one. By assumption this is initialized with 0. Then we concurrently execute two programs. The first overrides the value at location r with 1. The other loops until the value at location r is 1. Lastly, we read the value at location r into the variable x, thus we expect x to

$$\frac{C_1' \neq \lightning \qquad C_1, (s,h) \rightarrow C_1', (s',h')}{C_1 \,;\, C_2, (s,h) \rightarrow C_1' \,;\, C_2, (s',h')}\ \text{SEQ} \qquad \frac{}{\downarrow \,;\, C_2, (s,h) \rightarrow C_2, (s,h)}\ \text{SEQ-END}$$

$$\frac{C_1, (s,h) \rightarrow \lightning, (s,h)}{C_1 \,;\, C_2, (s,h) \rightarrow \lightning, (s,h)}\ \text{SEQ-ABT} \qquad \frac{}{\lightning \,;\, C_2, (s,h) \rightarrow \lightning, (s,h)}\ \text{SEQ-ABT-2}$$

$$\frac{e_b(s) = \mathsf{true}}{\texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\}, (s,h) \rightarrow C_1, (s,h)}\ \text{IF-T} \qquad \frac{e_b(s) = \mathsf{false}}{\texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\}, (s,h) \rightarrow C_2, (s,h)}\ \text{IF-F}$$

$$\frac{e_b(s) = \mathsf{true}}{\texttt{while}\,(\,e_b\,)\{C_1\}, (s,h) \rightarrow C_1 \,;\, \texttt{while}\,(\,e_b\,)\{C_1\}, (s,h)}\ \text{WHILE-T}$$

$$\frac{e_b(s) = \mathsf{false}}{\texttt{while}\,(\,e_b\,)\{C_1\}, (s,h) \rightarrow \downarrow, (s,h)}\ \text{WHILE-F}$$

$$\frac{C_1' \neq \lightning \qquad C_1, (s,h) \rightarrow C_1', (s',h')}{C_1 \parallel C_2\,, (s,h) \rightarrow C_1' \parallel C_2\,, (s',h')}\ \text{CON-L} \qquad \frac{C_2' \neq \lightning \qquad C_2, (s,h) \rightarrow C_2', (s',h')}{C_1 \parallel C_2\,, (s,h) \rightarrow C_1 \parallel C_2'\,, (s',h')}\ \text{CON-R}$$

$$\frac{C_1, (s,h) \rightarrow \lightning, (s,h)}{C_1 \parallel C_2\,, (s,h) \rightarrow \lightning, (s,h)}\ \text{CON-L-ABT} \qquad \frac{C_2, (s,h) \rightarrow \lightning, (s,h)}{C_1 \parallel C_2\,, (s,h) \rightarrow \lightning, (s,h)}\ \text{CON-R-ABT}$$

$$\frac{}{\lightning \parallel C_2\,, (s,h) \rightarrow \lightning, (s,h)}\ \text{CON-L-ABT-2} \qquad \frac{}{C_1 \parallel \lightning\,, (s,h) \rightarrow \lightning, (s,h)}\ \text{CON-R-ABT-2}$$

$$\frac{}{\downarrow \parallel \ \downarrow, (s,h) \rightarrow \downarrow, (s,h)}\ \text{CON-END}$$

Figure 3.2.: Operational Semantics for all flow related program constructs in cPL. Sequencing and concurrency are defined inductively, branching and looping are defined transitionally. In any case, we immediately transition to an abort statement when an underlying command fails.

have the value 1 after terminating. The program may however not terminate when the interleaving is unfair and only the loop is executed.

We will next introduce the first of two kinds of semantics. This is common praxis for precondition-postcondition semantics as we will introduce them in Section 3.3. Usually, compilers are based on operational semantics. It is thus easier to verify that the operational semantics is the "correct" one. After that, we can prove that the axiomatic semantics match the operational semantics. We will leave this proof out for the semantics from Chapters 3 and 4 for the sake of brevity and refer to the relevant sources where this proof is given instead. The formal operational semantics are given in Figures 3.1 and 3.2.

Example 3.2.2 We will now provide the operational semantics for the program from Example 3.2.1 denoted by C on the input (s, h) where $s(x) = 0$, $s(r) = 0$ and $h = \emptyset$. For convenience we let s_n be such that $s_n(x) = 0$ and $s_n(r) = n$ and s_n' be such that $s_n'(x) = 1$ and $s_n'(r) = n$.
We start with the state $C, (s, h)$. We like to verify that at every terminating state $\downarrow, (s', h')$ we have $s'(x) = 1$. That is, the value 1 was successfully transmitted to x.

We first apply the sequencing rule and thus have

$$x := \texttt{new}(1), (s, h) \rightarrow \downarrow, (s_n, \{ n \mapsto 0 \})$$

for any n. Now we need to apply every possible interleaving. First we consider the case where only the right thread is executed, as it will lead us into an infinite loop:

The proof here is only possible since the program has only finitely many states. If the program has infinitely many states, we need to use different semantics or proof rules.

$$
\begin{aligned}
& <r> := 1 \parallel x := 0; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* \; & <r> := 1 \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* \; & <r> := 1 \parallel x := <r>; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* \; & <r> := 1 \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \})
\end{aligned}
$$

Remark, that we skip some steps involving the rule SEQ-END and thus take the transitive closure of $\rightarrow$, which we denote as $\rightarrow^*$. At any of the above depicted states, the left side of the concurrent execution can be executed. Let us consider all of them one after the other:

- If at state
$$<r> := 1 \parallel x := 0; \ldots, (s_n, \{ n \mapsto 0 \})$$
the left side is executed, we get the following sequence of transitions:
$$
\begin{aligned}
& <r> := 1 \parallel x := 0; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* \; & \downarrow \parallel x := 0; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 1 \}) \\
\rightarrow^* \; & \downarrow \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 1 \}) \\
\rightarrow^* \; & \downarrow \parallel x := <r>; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 1 \}) \\
\rightarrow^* \; & \downarrow \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s'_n, \{ n \mapsto 1 \}) \\
\rightarrow^* \; & \downarrow \parallel \; \downarrow, (s'_n, \{ n \mapsto 1 \}) \rightarrow \downarrow, (s'_n, \{ n \mapsto 1 \})
\end{aligned}
$$
We thus reach our goal that x is assigned 1 in the final state of this interleaving. Now we consider the next possible interleaving.
- If at state
$$<r> := 1 \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \})$$
the left side is executed, we get the following transition:
$$
\begin{aligned}
& <r> := 1 \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* \; & \downarrow \parallel \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 1 \})
\end{aligned}
$$
This is followed by the transitions depicted above.
- If at state
$$<r> := 1 \parallel x := <r>; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \})$$
the left side is executed, we get the following transition:
$$
\begin{aligned}
& <r> := 1 \parallel x := <r>; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 0 \}) \\
\rightarrow^* & \downarrow \parallel x := <r>; \texttt{while}(x = 0)\{ x := <r> \}, (s_n, \{ n \mapsto 1 \})
\end{aligned}
$$
This is followed by the transitions depicted above.

Thus, all possible (terminating) transition sequences end up in a state s'_n.

3.3. Axiomatic Semantics

Axiomatic semantics define proof rules to prove certain specifications. This is in contrast to the operational semantics in the previous section, which lets us examine the possible program's executions. The sort of specifications we want to investigate are partial correctness properties consisting of pre- and postconditions expressed in separation logic. If an initial state satisfies the precondition, we want every terminating path to lead to a state that satisfies the postcondition. Partial correctness here means that we ignore non-terminating paths in our specification. We denote this specification as triples. However, when dealing with concurrency, these triples are usually not enough to prove non-trivial specifications in the presence of shared memory. To describe the shared memory, we use another proposition in separation logic in the specification called *resource invariant*. Thus we will define our axiomatic semantics over quadruples consisting of the resource invariant, the program, the pre- and the postcondition. For sake of this, we first define the *weakest resource-safe liberal precondition*. We can then define the quadruples as an implication between our precondition and the weakest resource-safe liberal precondition.

These triples are usually called *Hoare triples* in literature.

Weakest liberal precondition is usually referred to as *denotational semantics*. We will use it here to define axiomatic semantics to deal with concurrency.

A weakest liberal precondition takes a postcondition and a program and gives us the precondition for which we have exactly the initial states for which the program's execution will either non-terminate or terminate in a state satisfying the postcondition. Often we have even tighter specifications than the ones given by the weakest liberal preconditions. Thus, checking that this specification is stronger requires checking an entailment. Weakest resource-safe liberal preconditions require a special resource condition. It states — intuitively speaking — that the resource invariant must hold after every manipulation operation to the shared memory. We enforce this by not including the shared memory into our state, but instead adding every possible shared memory into the state before the operation and removing it again after the operation. This way, we require an invariance over every execution on the shared memory. One may wonder now, how we can change shared memory in the presence of such invariants anymore. However, the resource invariants can express multiple options for the value of shared memory. Changes to the shared memory is thus permitted as long as the resource invariant is still guaranteed after any atomic step.

Definition 3.3.1 (Weakest Resource-Safe Liberal Precondition) *Let C be a* cPL *program,* $\Phi \in \mathsf{SLProp}$ *a postcondition and* $\xi \in \mathsf{SLProp}$ *a resource invariant. The weakest resource-safe liberal precondition* **wrlp** *is the greatest solution of the equivalence*

$$\mathsf{wrlp}[\![C]\!]\,(\Phi \mid \xi) \iff \begin{cases} \mathsf{false} & \text{if } C = \lightning \\ \Phi & \text{if } C = \downarrow \\ \xi \mathrel{-\!\!*} \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\Phi \mid \xi) * \xi\right) & \text{else} \end{cases}$$

where $\mathsf{step}[\![C]\!]\,(f)$ *for a* cPL *program C,* $(s,h) \in \mathsf{States}$, $\eta \in \mathsf{LStacks}$ *and a function f mapping programs to SL is defined as*

$$\begin{aligned} & (s,h) \models_\eta \mathsf{step}[\![C]\!]\,(f) \\ \iff\quad & \forall C', s', h'.\quad C,(s,h) \to C',(s',h') \Rightarrow C' \neq \lightning \wedge (s',h') \models_\eta f(C'). \end{aligned}$$

We set the value of executions leading to an abort statement to false in two

instances: When terminating and in the step function. This eases some of the proofs. Definition 3.3.1 is a reformulated and slightly changed version of the one presented in [26]. We formulated it here like this to have as much separation logic in the definition. The formulation here also matches more closely logics such as Iris [41], where the full definition is formulated inside of separation logic. Intuitively, when we still have a program left to execute, we first *introduce* a *possible shared heap* into the current (possible local) heap using the magic wand, *execute* one step of the program and finally *remove* a *valid shared heap* again using the separating conjunction. Both, a possible and a valid shared heap is one that satisfies the resource invariant. However, the interpretation is different. When we introduce a shared heap, we consider all possible heaps satisfying the resource invariant. If we remove the shared heap, we want to validate that the resulting heap satisfies the resource invariant. The required fixed point does exist as the underlying function defining the equation is monotone, that is for the precondition $\Phi \in \mathsf{SLProp}$ and a resource invariant $\xi \in \mathsf{SLProp}$, the function

$$\lambda f.\, \lambda C.\, \xi \mathrel{-\!\!*} \mathsf{step}[\![C]\!]\left(\lambda C'.\, f(C') * \xi\right)$$

is monotone and thus the greatest fixed point exists.

Example 3.3.1 To make weakest resource-safe preconditions a bit more tangible, we will consider a very easy example with the following program C:

$$\begin{aligned} &<\!r_1\!> \;:=\; 1\,; \\ &<\!r_2\!> \;:=\; 1 \end{aligned}$$

We consider the resource invariant

$$\xi = r_1 \mapsto 0 * r_2 \mapsto 0 \vee r_1 \mapsto 1 * r_2 \mapsto 1$$

and check whether the stack with $s(r_1) = 1, s(r_2) = 2$ and heap $h = \emptyset$ satisfies the weakest resource-safe liberal precondition $\mathsf{wrlp}[\![C]\!]\,(\mathsf{true} \mid \xi)$. Remark that since we use resource invariants, the shared heap is not part of the initial state. The stack heap pair (s, h) will satisfy the weakest resource-safe liberal precondition exactly when the resource invariant holds, since our postcondition is only asking for true. Applying the equation from Definition 3.3.1 we have:

$$\begin{aligned} & (s,h) \models_\eta \mathsf{wrlp}[\![C]\!]\,(\mathsf{true} \mid \xi) \\ \iff & (s,h) \models_\eta \xi \mathrel{-\!\!*} \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\ \iff & (s,\{1 \mapsto 0, 2 \mapsto 0\}) \models_\eta \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\ & \text{and } (s,\{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \end{aligned}$$

Note, that the disjunction in the resource invariant converts to a conjunction into our proof obligation as the magic wand requires us to prove the statement for every heap satisfying the resource invariant. Thus it needs to hold for *both* satisfying heaps. If we now also resolve the step function and the separating

conjunction, we get:

$$
\begin{aligned}
& (s, \{1 \mapsto 0, 2 \mapsto 0\}) \models_\eta \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
\Rightarrow\ & (s, \{1 \mapsto 1, 2 \mapsto 0\}) \models_\eta \mathsf{wrlp}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\,(\mathsf{true} \mid \xi) * \xi \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
\Rightarrow\ & \mathsf{false} \text{ and } (s, \emptyset) \models_\eta \mathsf{wrlp}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\,(\mathsf{true} \mid \xi)
\end{aligned}
$$

Since the heap $\{1 \mapsto 1, 2 \mapsto 0\}$ has neither the form $r_1 \mapsto 0 * r_2 \mapsto 0$ nor the form $r_1 \mapsto 1 * r_2 \mapsto 1$, the resource invariant is violated after the first step and the specification does not hold.
We consider now instead the resource invariant

$$\xi = (r_1 \mapsto 0 \vee r_1 \mapsto 1) * (r_2 \mapsto 0 \vee r_2 \mapsto 1)$$

and check whether the stack with $s(r_1) = 1, s(r_2) = 2$ and heap $h = \emptyset$ satisfies the weakest resource-safe liberal precondition $\mathsf{wrlp}[\![C]\!]\,(\mathsf{true} \mid \xi)$. Applying the equation from Definition 3.3.1 and resolving the magic wand, we obtain:

$$
\begin{aligned}
& (s, h) \models_\eta \mathsf{wrlp}[\![C]\!]\,(\mathsf{true} \mid \xi) \\
\Rightarrow\ & (s, \{1 \mapsto 0, 2 \mapsto 0\}) \models_\eta \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\
& \text{and } (s, \{1 \mapsto 0, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 0\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—}
\end{aligned}
$$

Further resolving similarly to the example above yields:

$$
\begin{aligned}
& (s, \{1 \mapsto 0, 2 \mapsto 0\}) \models_\eta \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\
& \text{and } (s, \{1 \mapsto 0, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 0\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
\Rightarrow\ & (s, \{1 \mapsto 1, 2 \mapsto 0\}) \models_\eta \mathsf{wrlp}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\,(\mathsf{true} \mid \xi) * \xi \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
\Rightarrow\ & (s, \emptyset) \models_\eta \mathsf{wrlp}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\,(\mathsf{true} \mid \xi)
\end{aligned}
$$

This time, every possible heap after the transformation has a form that is permitted in the resource invariant. The very same can be done for the rest of the program, yielding:

$$
\begin{aligned}
& (s, \emptyset) \models_\eta \mathsf{wrlp}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\,(\mathsf{true} \mid \xi) \\
\Rightarrow\ & (s, \{1 \mapsto 0, 2 \mapsto 0\}) \models_\eta \mathsf{step}[\![\texttt{<}r_2\texttt{>}\ := 1]\!]\left(\lambda C'.\, \mathsf{wrlp}[\![C']\!]\,(\mathsf{true} \mid \xi) * \xi\right) \\
& \text{and } (s, \{1 \mapsto 0, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 0\}) \models_\eta \qquad \text{—"—} \\
& \text{and } (s, \{1 \mapsto 1, 2 \mapsto 1\}) \models_\eta \qquad \text{—"—} \\
\Rightarrow\ & (s, \emptyset) \models_\eta \mathsf{wrlp}[\![\downarrow]\!]\,(\mathsf{true} \mid \xi) \\
\Rightarrow\ & (s, \emptyset) \models_\eta \mathsf{true}
\end{aligned}
$$

We denote specifications in concurrent separation logic as $\xi \vdash \{\Phi\}\ C\ \{\Psi\}$, which states that for every initial state satisfying the precondition Φ every terminating run of C terminates in a state satisfying the postcondition Ψ, such that between every transitions the current shared state (which is not part of the input) satisfies ξ. The shared state is always introduced between one atomic step and then removed again after the step. We define this as the entailment of the precondition and the weakest resource-safe liberal precondition as below.

Definition 3.3.2 (Specifications for cPL programs) *Let C be a* cPL *program, $\Phi \in \mathsf{SLProp}$ a precondition, $\Psi \in \mathsf{SLProp}$ a postcondition and $\xi \in \mathsf{SLProp}$ a resource invariant. We define:*

$$\xi \vdash \{\Phi\}\ C\ \{\Psi\} \qquad \Longleftrightarrow \qquad \Phi \Rightarrow \mathsf{wrlp}[\![C]\!]\,(\Psi \mid \xi)\,.$$

We can also express now for some of these specifications, what they actually mean semantically:

Theorem 3.3.1 (Adequacy [26]) *Let C be a* cPL *program, $\Phi \in \mathsf{SLProp}$ a precondition and $\Psi \in \mathsf{SLProp}$ a postcondition. If we have*

$$\mathsf{emp} \vdash \{\Phi\}\ C\ \{\Psi\}$$

then for every initial $(s,h) \in \mathsf{States}$ and $\eta \in \mathsf{LVars}$ if $(s,h) \vDash_\eta \Phi$ then for every possible execution of C terminating with the stack-heap pair (s',h') we have $(s',h') \vDash_\eta \Psi$.

For this quadruple, we define various proof rules which are given in Figure 3.3 and explain each of them in the following.

Skip programs `skip` coincide on precondition and postcondition, since they do not change the state.

Assignment programs $x := e$ change the value of variable x to the value of e in the stack. We can represent this in the postcondition by swapping every occurrence of the variable x on the left side of the assignment by the value e on the right side of the assignment.

Lookup programs $x := \langle e \rangle$ change the value of variable x in the stack corresponding to the value at the evaluated location e in the heap. To access a value in the heap, we ask for the value a such that this is the value of the heap at the location e. In separation logic, we first remove the value from the heap using the separating conjunction $e \mapsto a * \ldots$ and afterwards add it again using the magic wand $e \mapsto a \mathrel{-\!\!*} \ldots$ This is the reverse order we did to introduce shared memory. Now we can update the postcondition with the value a using substitution similar to assignments.

Mutation programs $\langle e \rangle := e'$ change the value at the location e to the value e' in the heap. To verify that the heap location is allocated, we remove it from the heap using the separating conjunction $(\exists a.\, e \mapsto a) * \ldots$ Next, we introduce the location back with the new value e' using the magic wand $e \mapsto e' \mathrel{-\!\!*} \ldots$

For allocation and deallocation programs, we allow to set a dynamic value for the size of the chunk to be allocated or deallocated. To model this, we use a big operation notation for separating conjunction.

$$\frac{}{\xi \vdash \{\Phi\}\ \texttt{skip}\ \{\Phi\}}\ \text{skip} \qquad \frac{x \notin \mathrm{Vars}(\xi)}{\xi \vdash \{\Phi[x := e]\}\ x\ \texttt{:=}\ e\ \{\Phi\}}\ \text{assign}$$

$$\frac{x \notin \mathrm{Vars}(\xi)}{\xi \vdash \{\exists a.\, e \mapsto a * (e \mapsto a \mathrel{-\!\!*} \Phi[x := a])\}\ x\ \texttt{:=}\ \texttt{<}e\texttt{>}\ \{\Phi\}}\ \text{lookup}$$

$$\frac{}{\xi \vdash \left\{(\exists a.\, e \mapsto a) * \left(e \mapsto e' \mathrel{-\!\!*} \Phi\right)\right\}\ \texttt{<}e\texttt{>}\ \texttt{:=}\ e'\ \{\Phi\}}\ \text{mut}$$

$$\frac{x \notin \mathrm{Vars}(\xi)}{\xi \vdash \left\{e \in \mathbb{N} \wedge \forall a.\ \mathop{\ast}_{i=0}^{e-1} a + i \mapsto 0 \mathrel{-\!\!*} \Phi[x := a]\right\}\ x\ \texttt{:=}\ \mathrm{new}(e)\ \{\Phi\}}\ \text{alloc}$$

$$\frac{}{\xi \vdash \left\{e' \in \mathbb{N} \wedge \left(\mathop{\ast}_{i=0}^{e'-1} \exists a.\, e + i \mapsto a\right) * \Phi\right\}\ \texttt{free}(e, e')\ \{\Phi\}}\ \text{dealloc}$$

$$\frac{x \notin \mathrm{Vars}(\xi)}{\xi \vdash \{e_l \mapsto e_c * (e_l \mapsto e_s \mathrel{-\!\!*} \Phi[x := 1])\}\ x\ \texttt{:=}\ \mathrm{CAS}(e_l, e_c, e_s)\ \{\Phi\}}\ \text{cas-true}$$

$$\frac{x \notin \mathrm{Vars}(\xi)}{\xi \vdash \{\exists a.\, (e_l \mapsto a \wedge a \neq e_c) * (e_l \mapsto a \mathrel{-\!\!*} \Phi[x := 0])\}\ x\ \texttt{:=}\ \mathrm{CAS}(e_l, e_c, e_s)\ \{\Phi\}}\ \text{cas-false}$$

$$\frac{\xi \vdash \{\Phi_1\}\ C_1\ \{\Psi\} \qquad \xi \vdash \{\Phi_2\}\ C_2\ \{\Psi\}}{\xi \vdash \{e_b \wedge \Phi_1 \vee \neg e_b \wedge \Phi_2\}\ \texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{\Psi\}}\ \text{if}$$

$$\frac{\xi \vdash \{\Phi\}\ C_1\ \{\Psi\} \qquad \xi \vdash \{\Psi\}\ C_2\ \{\Theta\}}{\xi \vdash \{\Phi\}\ C_1 ; C_2\ \{\Theta\}}\ \text{seq} \qquad \frac{I \vDash e_b \wedge \Phi \vee \neg e_b \wedge \Psi \qquad \xi \vdash \{\Phi\}\ C\ \{I\}}{\xi \vdash \{I\}\ \texttt{while}(\,e_b\,)\{C\}\ \{\Psi\}}\ \text{while}$$

$$\frac{\xi \vdash \{\Phi_1\}\ C_1\ \{\Psi_1\} \qquad \xi \vdash \{\Phi_2\}\ C_2\ \{\Psi_2\} \qquad \forall i \in \{1,2\}.\ \mathrm{Write}(C_i) \cap \mathrm{Vars}(C_{3-i}, \Psi_{3-i}, \xi) = \emptyset}{\xi \vdash \{\Phi_1 * \Phi_2\}\ C_1 \parallel C_2\ \{\Psi_1 * \Psi_2\}}\ \text{concur}$$

$$\frac{\xi * \pi \vdash \{\Phi\}\ C\ \{\Psi\}}{\xi \vdash \{\Phi * \pi\}\ C\ \{\Psi * \pi\}}\ \text{share} \qquad \frac{\mathsf{emp} \vdash \{\Phi * \xi\}\ C\ \{\Psi * \xi\} \qquad C \text{ is a terminating atom}}{\xi \vdash \{\Phi\}\ C\ \{\Psi\}}\ \text{atom}$$

$$\frac{\xi \vdash \{\Phi\}\ C\ \{\Psi\} \qquad \xi \vdash \{\Phi'\}\ C\ \{\Psi'\}}{\xi \vdash \{\Phi \vee \Phi'\}\ C\ \{\Psi \vee \Psi'\}}\ \text{disj}$$

$$\frac{\xi \vdash \{\Phi\}\ C\ \{\Psi\} \qquad \xi \vdash \{\Phi'\}\ C\ \{\Psi'\} \qquad \xi \text{ precise}}{\xi \vdash \{\Phi \wedge \Phi'\}\ C\ \{\Psi \wedge \Psi'\}}\ \text{conj}$$

$$\frac{\xi \vdash \{\Phi\}\ C\ \{\Psi\} \qquad \mathrm{Write}(C) \cap \mathrm{Vars}(\Theta) = \emptyset}{\xi \vdash \{\Phi * \Theta\}\ C\ \{\Psi * \Theta\}}\ \text{frame}$$

$$\frac{\Phi \vDash \Phi' \qquad \xi \vdash \{\Phi'\}\ C\ \{\Psi'\} \qquad \Psi' \vDash \Psi}{\xi \vdash \{\Phi\}\ C\ \{\Psi\}}\ \text{conseq}$$

Figure 3.3.: Axiomatic semantics using the notation defined in Definition 3.3.2. We use here semantics similar to denotation semantics in order to closely match the axiomatic semantics in later chapters. The first proof rules deal with atomic statements, followed by flow related programs, then proof rules dealing with the resource invariant and lastly elimination proof rules.

Allocation programs $x := \texttt{new}(e)$ allocate e many consecutive locations in the heap. Since the starting address is chosen non-deterministically, we need to cover every possibility. We can do this using an all-quantifier and the magic wand to introduce the heap location. Since we can dynamically allocate arbitrary many values, the program may crash if we use non-natural numbers as inputs. Remark here, that we want to allocate e many pointers and use zero as starting index, thus we need to iterate from 0 to $e - 1$.

In the Lean formalization, we define the big operations slightly different to avoid using negative numbers.

Deallocation programs $\texttt{free}(e, e')$ deallocate e' consecutive locations starting from the one at e in the heap. We can encode this by using the separating conjunction to remove that part in the postcondition, i.e. we use a proposition of the form $(*_{i=0}^{e-1} \exists a.\, x + i \mapsto a) * \ldots$ to remove this part of the heap. Again, we have to use $e - 1$ as our index starts with 0.

Compare and set programs $x := \texttt{CAS}(e_l, e_c, e_s)$ have two cases, for which we differentiate. In order to keep the premises small, we split these two cases into two rules. The first rule applies if the value at location e_l is indeed e_c. Again, we verify this using the separating conjunction method. Then we reintroduce the location e_l but with the new value e_s using the magic wand. Lastly, we update x to 1 to indicate success using substitution. In the other case, the value at location e_l does not match e_c. We verify this similarly to the other case. Next, we reintroduce the location e_l but now with the unchanged value using the magic wand and set x to 0 using substitution.

Sequential programs $C_1 ; C_2$ execute first the left program C_1 and then the right program C_2. To express this, we split the proof obligation into two. First we need to prove a specification for some new separation logic proposition Ψ for the left side program C_1, i.e. the proof obligation $\xi \vdash \{\Phi\}\ C_1\ \{\Psi\}$ and then the corresponding one $\xi \vdash \{\Psi\}\ C_2\ \{\Theta\}$ for the right program C_2.

The word conditional branching may sound superficial as we only have one sort of branching command. However, we differntiate later between conditional and probabilistic branching.

Conditional branching programs $\texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ execute either the first or the second program, depending on the evaluation of e_b on the stack. Thus we require three things: (1) We need to differentiate on the evaluation of e_b, (2) prove the proof obligation $\xi \vdash \{\Phi_1\}\ C_2\ \{\Psi\}$ for the left program C_1, (3) prove the proof obligation $\xi \vdash \{\Phi_2\}\ C_2\ \{\Psi\}$ for the right program C_2.

Looping programs $\texttt{while}\,(\,e_b\,)\,\{C\}$ execute C until e_b does not hold. Since we consider a liberal interpretation of looping programs, the semantics of the loop correspond to a greatest fixed point. This allows us to use any pre-fixed point — which we will call loop invariants — to prove a specification on it. Indeed, invariants may only contain less non-terminating states than the greatest invariant. Given such a loop invariant I, we need to prove the invariance, which is $I \vDash e_b \wedge \mathsf{wrlp}[\![C]\!]\,(I \mid \xi) \vee \neg e_b \wedge \Psi$. We split this into the two proof obligations $I \vDash e_b \wedge \Phi \vee \neg e_b \wedge \Psi$ and $\xi \vdash \{\Phi\}\ C\ \{I\}$ for convenience.

Concurrent programs $C_1 \parallel C_2$ execute both programs by interleaving it arbitrarily. This makes reasoning about them a hurdle. Attempting to reason about each thread C_1 and C_2 locally seems difficult. However, here our resource invariants shine. Indeed, the presence of a resource invariant together with a split of the heap into the local heaps for each thread allows us the reason about C_1 and C_2 *locally*! We thus get the two proof obligations $\xi \vdash \{\Phi_1\}\ C_1\ \{\Psi_1\}$ and $\xi \vdash \{\Psi_2\}\ C_2\ \{\Psi_2\}$. Since the resource invariant only depicts shared heap memory, we cannot allow shared variables and require local variables or constants in each thread. To check that variables are either local or constants, we use two helper functions. The first collects all variables that are (syntactically) written to,

The concurrency proof rule is not possible on any programming language. We rely here on the fact that the programming language allows framing of memory.

the second one collects all variables that the separation logic propositions depend on. If their intersection is empty, every variable is either local or a constant.

Definition 3.3.3 *Let C be a* cPL *Program. We define Write(C) as the set of variables which occur on the left side of an assignment or lookup in C. We define Vars(C) as all variables occurring in the program C.*
Let $\Phi \in$ SLProp be an SL proposition, we define Vars(Φ) as the set of variables such that $x \in$ Vars(Φ) if and only if there exists $(s, h) \in$ States, $\eta \in$ LStacks and a value $v \in \mathbb{Q}$ such that

$$(s, h) \models_\eta \Phi \iff (s[x := v], h) \not\models_\eta \Phi.$$

We use the shorthand notation Vars$(C, \Phi, \Psi) =$ Vars$(C) \cup$ Vars$(\Phi) \cup$ Vars(Ψ).

While Write(C) can be checked statically, for Vars(Φ) we can resort to statically calculating an over-approximation.

Sharing allows us to extend the resource invariant in our proof obligation. That is, the specification $\xi * \pi \vdash \{\Phi\}\ C\ \{\Psi\}$ is *stronger* than the specification $\xi \vdash \{\Phi * \pi\}\ C\ \{\Psi * \pi\}$. Thus proving the prior also proves the latter.

Atomic programs can access the shared memory, but need to realize the resource invariant after execution immediately again. This allows us to get access to the resource invariant during the proof, i.e. effectively reversing the share proof rule for exactly one step. This is of course necessary when the program actually operates on the shared memory. We call programs on which we can apply this rule *terminating atomic*. The loop statement for example is also executed in one atomic step, but does not terminate in exactly one step.

Definition 3.3.4 (Terminating Atomic) *We call a program C terminating atomic if for all $C, (s, h) \to C', (s', h')$ we have $C' = \downarrow$.*

Disjunction elimination can be achieved by proving the proof obligation for each part of the conjunction separately. The proof rule does, however, require both the precondition and the postcondition to be a disjunction.

Conjunction elimination is a bit more tricky. Indeed, it turns out that for conjunction contrary to disjunction, we need an additional condition on the resource invariant. This is because we coupled the resource invariant in the weakest resource-safe liberal precondition using the separating conjunction. But the separating conjunction does not distribute over conjunction — except if the resource invariant is *precise*. Preciseness requires that for any heap satisfying the proposition, there is only maximally one *smaller* heap satisfying the proposition.

Definition 3.3.5 (Preciseness [44]) *We call the SL proposition $\Phi \in$ SLProp precise if for every $(s, h) \in$ States and $\eta \in$ LStacks, there exists $h' \in$ Heaps with $h' \subseteq h$, such that we have for every $h'' \in$ Heaps with $h' \subseteq h$ and $h' \neq h''$ we have $(s, h'') \not\models_\eta \Phi$.*

For precise left-hand sides of a separating conjunction, conjunction distributed over the separating conjunction on the right-hand side.

Theorem 3.3.2 (Distributivity with Preciseness [44]) *Let* $\Phi, \Psi_1, \Psi_2 \in \mathrm{SLProp}$ *be SL propositions and* Φ *be precise. We have*

$$\Phi * (\Psi_1 \wedge \Psi_2) \iff (\Phi * \Psi_1) \wedge (\Phi * \Psi_2).$$

Something similar also exists for a combination of a magic wand with a disjunction. As this is not of relevance for us, we will not consider this further here.

Example 3.3.2 The proposition $1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1$ is not precise as we can extend the heap $h = \{1 \mapsto 1\}$ to the heap $h' = \{1 \mapsto 1, 2 \mapsto 1\}$ and both satisfy the proposition. We now do not have the following implication:

$$\begin{aligned} &((1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * 2 \mapsto 1 * 3 \mapsto 1) \\ &\wedge ((1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * 3 \mapsto 1) \\ &\not\Rightarrow (1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * ((2 \mapsto 1 * 3 \mapsto 1) \wedge 3 \mapsto 1) \end{aligned}$$

Let us investigate a counterexample to that implication. We have that for the heap $h = \{1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1\}$ the propositions

$$\begin{aligned} &(1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * 2 \mapsto 1 * 3 \mapsto 1 && \text{is satisfied by } h, \\ &(1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * 3 \mapsto 1 && \text{is satisfied by } h, \\ &(1 \mapsto 1 \vee 1 \mapsto 1 * 2 \mapsto 1) * ((2 \mapsto 1 * 3 \mapsto 1) \wedge 3 \mapsto 1) && \text{is not satisfied by } h, \end{aligned}$$

as $(2 \mapsto 1 * 3 \mapsto 1) \wedge 3 \mapsto 1$ is unsatisfiable. Thus the implication above cannot hold. However the proposition $1 \mapsto 1 \vee 2 \mapsto 1$ is precise as every satisfying heap can not be extended and we have the following equivalence:

$$\begin{aligned} &((1 \mapsto 1 \vee 2 \mapsto 1) * 2 \mapsto 1 * 3 \mapsto 1) \\ &\wedge ((1 \mapsto 1 \vee 2 \mapsto 1) * 3 \mapsto 1) \\ &\Leftrightarrow (1 \mapsto 1 \vee 2 \mapsto 1) * ((2 \mapsto 1 * 3 \mapsto 1) \wedge 3 \mapsto 1) \end{aligned}$$

Here both sides are unsatisfiable, which makes them equivalent.

Framing allows to exclude a part of the heap that is not relevant for the programs execution. Such rules are useful when we reason locally about programs that only operate on a part of the heap. For example, if the heap consists of two lists, we like to reason about inserting into one of these without tracking that the other remains unchanged. Framing allows this. We prove that the functionality does not access the second list, since after framing accessing it yields a runtime error, invalidating our proof.

Framing can also be considered a special case of the concurrency rule. We can transform every program C into the equivalent program $C \parallel \texttt{skip}$. Applying the concurrency rule on this program yields the same result as applying the framing rule on C directly.

The consequence rule allows us make the precondition stronger and the postcondition weaker. We will later call this property monotonicity. Indeed for postconditions the proof rule is derived from the monotonicity of the weakest resource-safe liberal preconditions.

Sound means here that for every proof rule, if the premises (statements over the lines) hold, then also the conclusion (statements under the line) holds.

Theorem 3.3.3 (Soundness [26]) *The proof rules from Figure 3.3 are sound.*

We will not prove the soundness of the axiomatic semantics presented here — even if they differ in some details to the one from [26]. Instead we give a proof for semantics for probabilistic programs, which are conservative towards these and thus supersede most of them.

Sometimes, we will use a special notation that annotates the program. This notation has the style

$$\begin{array}{l} \quad /\!\!/ \;\; \Phi \\ C \\ \quad /\!\!/ \;\; \Psi \end{array}$$

and means $\xi \vdash \{\Phi\}\ C\ \{\Psi\}$ for some ξ that is clear from the context. If we want to denote the resource invariant explicitly, we also write

$$\begin{array}{l} \quad /\!\!/ \;\; \Phi \mid \xi \\ C \\ \quad /\!\!/ \;\; \Psi \mid \xi \end{array}$$

for $\xi \vdash \{\Phi\}\ C\ \{\Psi\}$. Remark, that the resource invariant needs to be equal. When using the rules share and atom, this notation may result in confusing notations, as this example demonstrates:

Example 3.3.3 Consider the following annotated program:

$$\begin{array}{l} \quad /\!\!/ \;\; x \mapsto 1 \mid \mathsf{emp} \\ \quad /\!\!/ \;\; \mathsf{emp} \mid x \mapsto 1 \\ \quad /\!\!/ \;\; x \mapsto 1 \mid \mathsf{emp} \\ \texttt{<}x\texttt{>} := 1 \\ \quad /\!\!/ \;\; x \mapsto 1 \mid \mathsf{emp} \\ \quad /\!\!/ \;\; \mathsf{emp} \mid x \mapsto 1 \\ \quad /\!\!/ \;\; x \mapsto 1 \mid \mathsf{emp} \end{array}$$

The proof of this annotation works like this:

1. We like to prove

   $$\mathsf{emp} \vdash \{x \mapsto 1\}\ \texttt{<}x\texttt{>} := 1\ \{x \mapsto 1\}.$$

2. We apply the share rule and have to prove next

   $$x \mapsto 1 \vdash \{\mathsf{emp}\}\ \texttt{<}x\texttt{>} := 1\ \{\mathsf{emp}\}.$$

3. We apply the atom rule and have to prove

   $$\mathsf{emp} \vdash \{x \mapsto 1\}\ \texttt{<}x\texttt{>} := 1\ \{x \mapsto 1\}.$$

4. We lastly apply the mutation rule and are left to prove that

   $$x \mapsto 1 \vDash (\exists a.\, x \mapsto a) * (x \mapsto 1 \mathrel{-\!\!*} x \mapsto 1),$$

   which also holds.

The notation may even result in wrong interpretations. Consider for this the

following annotated program:

$$
\begin{array}{l}
\quad /\!\!/ \; x \mapsto 1 \mid \mathsf{emp} \\
\quad /\!\!/ \; \mathsf{emp} \mid x \mapsto 1 \\
\texttt{skip} \\
\quad /\!\!/ \; \mathsf{emp} \mid x \mapsto 1 \\
\quad /\!\!/ \; x \mapsto 1 \mid \mathsf{emp} \\
<x> := 2 \\
\quad /\!\!/ \; x \mapsto 2 \mid \mathsf{emp} \\
<x> := 1 \\
\quad /\!\!/ \; x \mapsto 1 \mid \mathsf{emp} \\
\quad /\!\!/ \; \mathsf{emp} \mid x \mapsto 1 \\
\texttt{skip} \\
\quad /\!\!/ \; \mathsf{emp} \mid x \mapsto 1 \\
\quad /\!\!/ \; x \mapsto 1 \mid \mathsf{emp}
\end{array}
$$

What happens here is that for the quadruple $\mathsf{emp} \vdash \{x \mapsto 1\}\ \texttt{skip}\ \{x \mapsto 1\}$ we choose to introduce a resource invariant, but do not use one for the program

$$
\begin{array}{l}
<x> := 2 \\
<x> := 1
\end{array}
$$

as this would be unsound. Indeed, between those mutations, the resource invariant is violated. However, one may think that this notation gives us the quadruple $x \mapsto 1 \vdash \{\mathsf{emp}\}\ <x> := 2;\ <x> := 1\ \{\mathsf{emp}\}$, which is wrong. Thus, we need to be very careful that we apply all proof rules correctly and read the interpretations correctly when using this notation.

Using this notation, we will now prove the desired property for the running example from the introduction section.

Example 3.3.4 With the tools explained in this chapter, we will look again at our initial example, the following program C:

$$
\begin{array}{l}
<r> := -1; \\
<r> := 0 \quad \left\| \begin{array}{l} x := <r>; \\ \texttt{while}(x = -1)\{x := <r>\}; \end{array} \right.
\end{array}
$$

We will prove now that the specification

$$\mathsf{emp} \vdash \{r \mapsto 0 * x = 0\}\ C\ \{(r \mapsto -1 \lor r \mapsto 0) * x = 0\}$$

holds. By Theorem 3.3.1 we can then deduce that every terminating execution of C in an initial state satisfying $r \mapsto 0 * x = 0$ will end in a state satisfying $(r \mapsto 0 \lor r \mapsto -1) * x = 0$. We prove this specification in three steps with annotated programs. During the explanation, we will not mention the application of the sequential proof rule, but use it implicitly.

- The first part looks just at the sequential part of the program.

$$
\begin{aligned}
&/\!/\ x = 0 * r \mapsto 0 \mid \mathsf{emp} \\
&<r> := -1\,; \\
&/\!/\ x = 0 * r \mapsto -1 \mid \mathsf{emp} \\
&/\!/\ x = 0 * (r \mapsto -1 \lor r \mapsto 0) \mid \mathsf{emp} \\
&/\!/\ x = 0 \mid (r \mapsto -1 \lor r \mapsto 0) \\
&/\!/\ \mathsf{true} * x = 0 \mid (r \mapsto -1 \lor r \mapsto 0)
\end{aligned}
$$

We explain the program from top to bottom. For other logics we may prefer the other direction. However, for qualitative separation logic this way is usually more intuitive. First, we apply the mutation rule to assign -1 to the location of r. The very next thing is an application of the consequence rule allowing us to weaken the condition on r to also allow for the value 0. We do this, in order to transform the part about r into a useful resource invariant. Indeed, the next rule we apply is the share rule allowing us to move the part about r into the resource invariant. Lastly, we add true in order to enable applying the concurrency rule:

$$
\begin{aligned}
&/\!/\ \mathsf{true} * x = 0 \mid (r \mapsto -1 \lor r \mapsto 0) \\
&<r> := 0 \quad \left\Vert \begin{array}{l} x := <r>\,; \\ \mathtt{while}\,(x = -1)\,\{x := <r>\}\,; \end{array} \right. \\
&/\!/\ \mathsf{true} * x = 0 \mid (r \mapsto -1 \lor r \mapsto 0) \\
&/\!/\ x = 0 \mid (r \mapsto -1 \lor r \mapsto 0) \\
&/\!/\ (r \mapsto -1 \lor r \mapsto 0) * x = 0 \mid \mathsf{emp}
\end{aligned}
$$

From this point on, we will locally consider each thread on its own.

- For the left part of the concurrent execution, we have the following annotated program:

$$
\begin{aligned}
&/\!/\ \mathsf{true} \mid r \mapsto -1 \lor r \mapsto 0 \\
&/\!/\ \mathsf{true} * (r \mapsto -1 \lor r \mapsto 0) \mid \mathsf{emp} \\
&<r> := 0 \\
&/\!/\ \mathsf{true} * r \mapsto 0 \mid \mathsf{emp} \\
&/\!/\ \mathsf{true} * (r \mapsto -1 \lor r \mapsto 0) \mid \mathsf{emp} \\
&/\!/\ \mathsf{true} \mid r \mapsto -1 \lor r \mapsto 0
\end{aligned}
$$

From the concurrency rule we had given the precondition true, the postcondition true and the resource invariant $r \mapsto -1 \lor r \mapsto 0$. In order to reason now about the mutation $<r> := 0$, we apply the atom rule and are left with

$$
\mathsf{emp} \vdash \left\{ \begin{array}{c} \mathsf{true} \\ * \,(r \mapsto -1 \lor r \mapsto 0) \end{array} \right\} <r> := 0 \left\{ \begin{array}{c} \mathsf{true} \\ * \,(r \mapsto -1 \lor r \mapsto 0) \end{array} \right\}.
$$

Applying the mutation rule, we have finally left to prove

$$
\begin{aligned}
&\mathsf{true} * (r \mapsto -1 \lor r \mapsto 0) \\
\models\ &(\exists a.\, r \mapsto a) * (r \mapsto 0 \mathrel{-\!\!*} \mathsf{true} * (r \mapsto -1 \lor r \mapsto 0)),
\end{aligned}
$$

which holds. We can match $\exists a.\, r \mapsto a$ with $r \mapsto -1 \vee r \mapsto 0$. For $r \mapsto 0 \mathrel{-\!\!*} \mathsf{true} * (r \mapsto -1 \vee r \mapsto 0)$, we first introduce $r \mapsto 0$ and match it with $r \mapsto -1 \vee r \mapsto 0$, which leaves us with the tautology true ⊨ true.

▶ For the right part of the concurrent execution, we have the annotated program:

```
  // x = 0 | r ↦ −1 ∨ r ↦ 0
  // x = 0 ∗ (r ↦ −1 ∨ r ↦ 0) | emp
x := <r>;
  // (x = −1 ∨ x = 0) ∗ (r ↦ −1 ∨ r ↦ 0) | emp
  // x = −1 ∨ x = 0 | r ↦ −1 ∨ r ↦ 0
while(x = −1){
  // x = −1 ∨ x = 0 | r ↦ −1 ∨ r ↦ 0
  // (x = −1 ∨ x = 0) ∗ (r ↦ −1 ∨ r ↦ 0) | emp
    x := <r>
  // (x = −1 ∨ x = 0) ∗ (r ↦ −1 ∨ r ↦ 0) | emp
  // x = −1 ∨ x = 0 | r ↦ −1 ∨ r ↦ 0
};
  // x = 0 | r ↦ −1 ∨ r ↦ 0
```

We first consider the lookup `x := <r>`. For this, we apply the atom rule, followed by the lookup rule, which requires us to prove the entailment

$$\begin{aligned} &x = 0 * (r \mapsto -1 \vee r \mapsto 0) \\ \models\, &\exists a.\, r \mapsto a * (r \mapsto a \mathrel{-\!\!*} (a = -1 \vee a = 0) * (r \mapsto -1 \vee r \mapsto 0)), \end{aligned}$$

which holds. We assume the stack heap pair (s, h) has the form restricted by the left side. Then $h(s(r)) = -1$ or $h(s(r)) = 0$. We pick a as one of these. Without loss of generality, we let $a = 0$. Then we remove the heap part with location $s(r)$ and add it again right after. The right part of

$$r \mapsto a \mathrel{-\!\!*} (a = -1 \vee a = 0) * (r \mapsto -1 \vee r \mapsto 0)$$

needs to have the heap $h_2 = \{s(r) \mapsto 0\}$, which is then satisfied. The rest of the heap goes to the left part, where $a = 0$ holds by assumption. For $a = 1$ the reasoning is similar.

Next we consider the loop. For this, we need to guess a loop invariant and check its validity. We guess the loop invariant $x = -1 \vee x = 0$. Then we need to check that

$$r \mapsto -1 \vee r \mapsto 0 \vdash \{x = -1 \vee x = 0\}\ x := \texttt{<}r\texttt{>}\ \{x = -1 \vee x = 0\}.$$

Applying first the atom rule and then the lookup rule, we are required to check the entailment

$$\begin{aligned} &(x = -1 \vee x = 0) * (r \mapsto -1 \vee r \mapsto 0) \\ \models\, &\exists a.\, r \mapsto a * (r \mapsto a \mathrel{-\!\!*} (a = -1 \vee a = 0) * (r \mapsto -1 \vee r \mapsto 0)), \end{aligned}$$

which holds for a very similar reason to the previous lookup. Lastly, we

need to check the entailment

$$\begin{aligned} &x = -1 \lor x = 0 \\ \models\ &x = -1 \land (x = -1 \lor x = 0) \lor \neg x = -1 \land x = 0, \end{aligned}$$

which holds by an easy case distinction over the disjunction on the left.

Quantitative Separation Logic | 4.

When reasoning about probabilistic programs, qualitative logics are often unwieldy. They require expressing probability distributions explicitly, even when the distribution is not of interest. Usually, only certain specifications are of interest, such as "the probability of something good should be at least 0.3" or "The expected value of the random variable is at most 10". For these questions, a different solution is appropriate. Instead of phrasing our specification in logic, we change our axiomatic semantics in such a way, that specifications of this style are naturally encodable. The semantics we will work towards in this chapter are called *expectation-based*. As probabilities are only a special case of expectations, reasoning about expectations is sufficient to reason about both.

As a first example, we consider the following program that chooses with probability 0.3 to execute $x := 1$ and with probability 0.7 to execute $x := 2$:

$$\texttt{pif}\ (0.3)\ \{x := 1\}\ \texttt{else}\ \{x := 2\}$$

We will later see, that the probability distribution of a program's output values also depends on the initial state. We are thus also interested in specifications that are valid for all initial states.

Say, we want to check that if we start in an initial state where x is -1, whether the probability that in the final state the variable x has value 1 is 0.3, phrased formally as $\mathbb{P}(x = 1) = 0.3$. We now calculate $\mathbb{P}(x = 1)$ by going throw the program bottom-up with respect to the control flow. The probabilistic choice tells us, that the left-hand side is executed with probability 0.3. If the left-hand side will validate our condition $x = 1$, we know that this will happen with probability 0.3. The same holds for the right-hand side of the program with probability 0.7. However, only the left program validates the condition $x = 1$, as the right-hand side sets x to 2. Thus we have $\mathbb{P}(x = 1) = 0.3 \cdot \mathsf{true} + 0.7 \cdot \mathsf{false}$. This equation is unfortunately type incorrect. We cannot multiply a rational number with a Boolean value. Instead we use *Iverson brackets* [45], which cast the proposition into a function that is 1 if the state satisfies the propositions and 0 else. This translates our equation into

$$\mathbb{P}(x = 1) = 0.3 \cdot [\mathsf{true}] + 0.7 \cdot [\mathsf{false}] = 0.3.$$

For a more challenging problem, we consider a program which is commonly called the geometric loop:

```
x = 1;
y = 0;
while(x = 1){
    pif (0.5) {x := 0} else {x := 1};
    y := y + 1
}
```

Now, we specify a bound on the *expected value of a random variable*. We like to verify that the expected value of y is exactly 2, phrased formally as $\mathbb{E}(y) = 2$. Again we reason bottom-up. Thus we first need to resolve the loop. Similar to the reasoning style from Chapter 3, we need to find an invariant property. Now our invariant is not a proposition but a *random variable*. This may sound confusing, as we wanted to compute expected values. However, if we parametrize an expected

value by its input, we have a function mapping states to values, which is what we consider a random variable. It turns out, that the invariant random variable is

$$[x = 1] \cdot (y + 2) + [x \neq 1] \cdot y.$$

We will discuss later in more detail why this is correct and instead continue with the calculation. When now using the information that initially $x = 1$ and $y = 0$, we can simplify our result and obtain

$$\mathbb{E}(y) = [1 = 1] \cdot (0 + 2) + [1 \neq 1] \cdot 0 = 2.$$

4.1. Markov Decision Processes

Probabilistic programs are operationally often modelled as Markovian models. Our programming language features two kinds of non-determinism. Probabilistic and non-probabilistic non-determinism differ in the amount of knowledge we have about them. Probabilistic non-determinism follow a certain distribution, while non-probabilistic non-determinism is resolved arbitrarily. We require a model that features both, probabilistic non-determinism and non-probabilistic non-determinism. These models are called Markov decision processes. In order to introduce these models, this section will guide you through the jungle of probability theory.

In probability theory, we consider sets of events to calculate their probability. The sets of events need to form a sigma algebra.

$\mathrm{Pow}(S)$ is the powerset of S and $\bar{A}$ is the complement with respect to S.

Definition 4.1.1 (Sigma Algebra [46]) *We call a set $\mathcal{S} \subseteq \mathrm{Pow}(S)$ a sigma algebra of S, if*

- $\emptyset \in \mathcal{S}$,
- *if* $A \in \mathcal{S}$ *then* $\bar{A} \in \mathcal{S}$, *and*
- *if for all* $n \in \mathbb{N}$ *we have* $A_n \in \mathcal{S}$ *then also* $\bigcup_{n \in \mathbb{N}} A_n \in \mathcal{S}$.

It makes sense to define our possible events like this, as these are the operations we require for common tasks in probability theory. Complements give rise to the negation of a certain condition, countable unions allow us to sum up disjoint events. A probability space now enriches the sigma algebra with a probability distribution sometimes called probability measure due to the foundational work on measure theory.

Definition 4.1.2 (Probability Space [46]) *A probability space is a triple* $(\Omega, \mathcal{F}, \mathbb{P})$ *where* Ω *is a non-empty set,* $\mathcal{F}$ *is a sigma algebra of* Ω *and for* $\mathbb{P}\colon \mathcal{F} \to [0, 1]$ *we have*

- $\mathbb{P}(\emptyset) = 0$,
- $\mathbb{P}(\Omega) = 1$, *and*
- *for pairwise disjoint sets* A_n, *we have* $\mathbb{P}(\bigcup_{n \in \mathbb{N}} A_n) = \sum_{n \in \mathbb{N}} \mathbb{P}(A_n)$.

We call Ω the sample space, $\mathcal{F}$ the event space and $\mathbb{P}$ the probability function or just distribution. A random variable in a probability space is nothing more than a map from events to real values. If we calculate the expected value of something, we always calculate the expected value of some random variable. In

the case where the events are countable and every singleton is contained in the sigma algebra, the expected value is the weighted sum of the probability of an event times the value of the random variable for that event. If we do not have countability, we need to resort to integrals. We will avoid introducing integrals and the accompanied measure theory but refer to [46] for excellent lecture notes on probability theory, measure theory and integrals.

Definition 4.1.3 (Random Variables and Expected Values [46]) *Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. We call a map from events to non-negative extended reals $X\colon \Omega \to \mathbb{R}^{\infty}_{\geq 0}$ a random variable. The expected value of X is*

$$\mathbb{E}(X) = \int_{\Omega} X(\omega)\, \mathbb{P}(d\omega).$$

When Ω is countable and for every $\omega \in \Omega$ we have $\{\omega\} \in \mathcal{F}$, the expected value of X is

$$\mathbb{E}(X) = \sum_{\omega \in \Omega} X(\omega) \cdot \mathbb{P}(\{\omega\}).$$

This integral is called Lebesque integral and is essential to measure theory.

Example 4.1.1 We define the following probability space:

- Let $\Omega = \{a, b, c\}$ be a sigma algebra with
- $\mathcal{F} = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$.
- We define $\mathbb{P}$ as the unique distribution with
  - $\mathbb{P}(\{a\}) = 0.2$,
  - $\mathbb{P}(\{b\}) = 0.5$, and
  - $\mathbb{P}(\{c\}) = 0.3$.

Moreover, we let X be a random variable with $X(a) = 1$, $X(b) = 2$ and $X(c) = 3$.
We have that the expected value of X is

$$\begin{aligned}\mathbb{E}(X) &= 0.2 \cdot 1 + 0.5 \cdot 2 + 0.3 \cdot 3 \\ &= 0.2 + 1 + 0.9 \\ &= 2.1\end{aligned}$$

We model our program operationally as *Markovian models*. These are generalizations of transition systems to the probabilistic world. Since we do not want to resolve concurrency and allocation randomly, but instead want to prove judgments about all possible allocations and all possible thread schedules. The appropriate model for these requirements is called *Markov decision process* and is based on the *(discrete-time) Markov chain*.

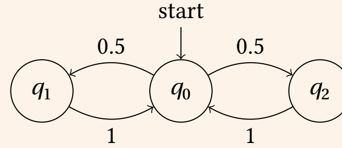
Continous-time Markov chains also exist in literature, but are of no interest in this thesis.

Definition 4.1.4 (Markov Chains [47]) *A Markov chain is a tuple $(S, \mathbf{P}, s_0)$ where*

- *S is a countable nonempty set of states,*
- *$\mathbf{P}\colon S \times S \to [0, 1]$ is the transition probability function such that for all states s:*
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- *$s_0 \in S$ is an initial state.*

Although we limit our Markov Chains to countable state spaces S, the sample space of our resulting probability space is still uncountable. This is because the sample space in a Markov chain is its *infinite paths*, of which there are uncountable many — even if the state space is finite.

Example 4.1.2 The following graph depicts a representation of the Markov chain M with $S = \{q_0, q_1, q_2\}$ and $\mathbf{P}$ as drawn:



We only connect states in this depiction if they have nonzero transition probability. The infinite paths in M with non-zero probability are all infinite sequences composed of the sub-sequences q_0, q_1 and q_0, q_2. As we can represent every real number in the open interval between 0 and 1 in binary form with these sequences, we have uncountable many infinite paths.

Since we are interested in *reachability properties* — that is the probability of eventually reaching a certain state — we need to consider the infinite paths of a Markov chain. Indeed, this is how we define the sample space of a Markov chain.

Definition 4.1.5 (Probability Space of a Markov Chain [47]) *Let $(S, \mathbf{P}, s_0)$ be a Markov chain M and $\hat{\pi} = s_0 \ldots s_n$ be a finite path in M. We define the cylinder set of $\hat{\pi}$ as*

$$Cyl(\hat{\pi}) \quad = \quad \{\pi \text{ is an infinite path in } M \mid \hat{\pi} \text{ is a prefix of } \pi\}.$$

We define the probability space of M as $(\Omega, \mathcal{F}, \mathbb{P})$ where

- ▶ *Ω is the set of all infinite paths,*
- ▶ *$\mathcal{F}$ is the smallest sigma algebra that contains all cylinder sets $Cyl(\hat{\pi})$, and*
- ▶ *$\mathbb{P}$ is the unique probability function with*

$$\mathbb{P}(Cyl(s_0 \ldots s_n)) = \prod_{0 \le i < n} \mathbf{P}(s_i, s_{i+1}).$$

The uniqueness of both the sigma algebra and the probability function follow from common probability theory concepts [47]. The probability space defined here now allows us to compute the reachability probability of a state as well. Say, we want to compute the probability for reaching state s. Then we take all finite sequences of states up to but without s and append s to them. All of these paths are prefix free and thus the cylinder sets are disjoint. Moreover, there are only countable many finite paths. Then we can compute the probability to reach state s by summing over the probability of all these cylinder sets.

Definition 4.1.6 (Reachability Probability in a Markov Chain [47]) *Let $(S, \mathbb{P}, s_0)$ be a Markov chain M and $B \subseteq S$ a set of goal states. The probability to reach a state in B is*

$$\mathbb{P}^M(\diamond B) = \sum_{s_0 \dots s_n \text{ with } \forall i<n.\ s_i \in S \setminus B \text{ and } s_n \in B} \mathbb{P}(Cyl(s_0, \dots s_n)).$$

Example 4.1.3 Revisiting Example 4.1.2, we now want to compute the probability to reach state q_2. Thus we have

$$\begin{aligned} \mathbb{P}(\diamond q_2) &= \sum_{q_0 \dots s_n \text{ with } \forall i<n.\ s_i \in S \setminus \{q_2\} \text{ and } s_n = q_2} \mathbb{P}(Cyl(s_0, \dots s_n)) \\ &= \sum_{i=0}^{\infty} \mathbb{P}(Cyl((q_0, q_1)^n q_0, q_2)) \\ &= \sum_{i=0}^{\infty} 0.5^n \cdot 0.5 \\ &= \frac{0.5}{1 - 0.5} \\ &= \frac{0.5}{0.5} \\ &= 1. \end{aligned}$$

The probability to eventually reach q_2 is thus 1.

As we are only interested in the result of the program, we will consider a special case of reachability probability: the probability for reaching a final state. We introduce dedicated final states from which one may not further traverse.

Definition 4.1.7 (Final States and Final State Probability Space) *Let $(S, \mathbf{P}, s_0)$ be a Markov chain M and $F \subseteq S$ be a set of final states such that for all $s \in F$ we have that they are absorbing: $\mathbf{P}(s, s) = 1$.*
We define the probability space on final states for a Markov chain as $(\Omega, \mathscr{F}, \mathbb{P})$ where

- *the sample set is the set of final states including a symbol for non-termination $\Omega = F \cup \{\perp\}$,*
- *the event space $\mathscr{F}$ is the powerset of the sample space, and*
- *the probability function is the unique reachability function*

$$\mathbb{P}(s) = \mathbb{P}^M(\diamond s).$$

The final state probability space justifies defining random variables on the final states. Theoretically, we could also define them on non-termination. Practically, this is not a good idea, as we will later see that defining it on the final states gives us nice equations. Instead our random variables will only be defined on the final states and how non-termination is included is decided by the flavor of the calculus. The uniqueness of this function follows from the disjointness of the corresponding cylinder sets and classical probability theory concepts.

As previously mentioned, Markov chains do not offer non-probabilistic non-determinism. To enrich a Markov model with non-determinism, we can, however, assign every node a set of transition probability functions depending on some

non-deterministic choice. This idea gives rise to the *Markov decision processes*. There we introduce a set A of actions, from which a scheduler can choose for each node. Each action is associated with a transition probability function, which we apply if the corresponding action was chosen. This adds another layer in the reasoning process. A Markov decision process does not have one probability space, but instead for every possible resolving — that is for every scheduler — we have a probability space.

Schedulers are sometimes also called policies.

Definition 4.1.8 (Markov Decision Process [48]) *A Markov decision process is a tuple* $(S, Act, \mathbf{P}, s_0)$ *where*

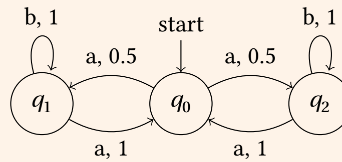
- S *is a countable nonempty set of states,*
- $Act\colon S \to Pow(A)$ *is a map from states to enabled actions,*
- $\mathbf{P}\colon S \times A \times S \to [0,1]$ *is the transition probability function such that for all states* s *and actions* $a \in Act(s)$*:*

$$\sum_{s' \in S} \mathbf{P}(s, a, s') = 1,$$

- $s_0 \in S$ *is an initial state.*

We call a function $\mathfrak{s}\colon S^* \to A$ *that maps a history of states to some next enabled action* a *(deterministic, history dependent) scheduler. It is partly-defined in case the set of enabled actions is empty for some state.*

Example 4.1.4 Consider the following Markov decision process where we annotated the action next to the probability:



There are infinitely many schedulers. Every time we arrive at q_1 or q_2 — which we will definitely do —, we can choose either action a or action b. If we choose the scheduler that always chooses a, we obtain paths that match exactly the ones from Example 4.1.2. In the next definition, we will formalize the relation between Markov decision processes and Markov chains.

We will consider here the *infinite-horizon* problem. This means that we consider paths in the Markov decision process not limited by any step number. Indeed, when using this for program verification, a limit on the number of steps would not allow us to reason about all programs, but only on programs with static runtime bounds. We consider here a deterministic and history dependent resolution of the non-determinism. Other schedulers may also allow randomization. It turns out, that the values we are interested in — that is the greatest lower bound on the expected values regarding the probability distribution of a probabilistic program — is the same for deterministic, history dependent schedulers and for randomized, history dependent schedulers.

Definition 4.1.9 (Induced Markov Chain [48]) *Let $(S, Act, \mathbf{P}, s_0)$ be a Markov decision process M. We define the Markov chain of M induced by scheduler $\mathfrak{s}$ as $(S', \mathbf{P}_\mathfrak{s}, s_0)$ with*

- *the states as paths on the Markov decision process $S' = \{\pi \in S^* \mid \pi = s_0, \dots\}$ and*
- *the transition probability function $\mathbf{P}_\mathfrak{s}$ as induced by the transition probability function of M:*

$$\mathbf{P}_\mathfrak{s}((s_0, \dots s_n), (s_0, \dots s_n, s_{n+1})) \quad = \quad \mathbf{P}(s_n, \mathfrak{s}(s_0, \dots s_n), s_{n+1}).$$

In case that $Act(s_n) = \emptyset$ we define:

$$\mathbf{P}_\mathfrak{s}((s_0, \dots s_n), (s_0, \dots s_n, s_{n+1})) \quad = \quad \begin{cases} 1 & \text{if } s_n = s_{n+1} \\ 0 & \text{else.} \end{cases}$$

Having an empty set of enabled actions is somewhat non-standard, but handy for defining final states. We resolve empty enabled actions here by self-loops.

Relating back to Definition 4.1.6, we also use the notation $\mathbb{P}_\mathfrak{s}(\diamond s)$ for $\mathbb{P}^M(\diamond s)$ where M is the by $\mathfrak{s}$ induced Markov chain of its Markov decision process.

Since we use Markov decision processes to represent operational semantics of probabilistic programs, we are interested in reachability probabilities for states that we define as final states. These final states encode a terminating program. In order to lift the Hoare triple reasoning from the last chapter, we assign values to those final states using random variables — usually called *rewards*. For this definition, we reuse final states from Definition 4.1.7 and lift it by using the induced Markov chains from Definition 4.1.9.

Definition 4.1.10 (Final Rewards and Minimal Expected Value) *Let $(S, Act, \mathbf{P}, s_0)$ be a Markov decision process M. Let $F \subseteq S$ be a set of final states for M where every enabled action set is empty: for all $s \in F$ we have $Act(s) = \emptyset$. Final rewards are random variables on final states. We define the minimal expected value of a random variable $X\colon F \to \mathbb{R}^\infty_{\geq 0}$ as*

$$\mathbb{E}(X) \quad = \quad \inf_\mathfrak{s} \sum_{s \in F} X(s) \cdot \mathbb{P}_\mathfrak{s}(\diamond s).$$

Computing the minimal expected value seems dawning. How should one proceed, even if the state space is finite? Indeed, there is a solution! The minimal expected value can be computed using fixed points on the Markov decision process. These fixed points are given by the *Bellman equation* and allow local reasoning for every state.

Definition 4.1.11 (Bellman Equation [48]) *Let $(S, Act, \mathbf{P}, s_0)$ be a Markov decision process M. Let $F \subseteq S$ be a set of final states for M and let $X\colon F \to \mathbb{R}^\infty_{\geq 0}$ be a random variable.*
The Bellman equation is defined as

$$\Phi_X(s) \quad = \quad \begin{cases} X(s) & \text{if } s \in F \\ \inf\limits_{a \in Act(s)} \sum\limits_{s' \in S} \Phi_X(s') \cdot \mathbb{P}(s, a, s') & \text{else.} \end{cases}$$

Usually the Bellman equation is stated using a random variable $X\colon S \to \mathbb{R}_{\geq 0}$ defined on the whole set of states and with $X(s)$ added in the recursive part of the equation. However, if we set $X(s) = 0$ for $s \notin F$ and add a transition from final states to a sink state, we get a similar result.

Crucially, we require here that $\mathrm{Act}(s)$ is not empty, as otherwise the Bellman equation assigns the state s the value 1, whereas the expected value assigns paths

with this state value 0. The Bellman equation now gives rise to an easier method to computing the expected value of a random variable. Instead of solving the infimum over an infinite sum, we only need to find the least solution to the Bellman equation. In a perfect world, this result would already be proven in its most generality. However, this is unfortunately not the case yet. The following thus remains a conjecture.

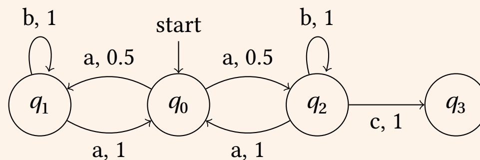
Conjecture 4.1.1 (Least Bellman Solution) *Let* $(S, Act, \mathbf{P}, s_0)$ *be a Markov decision process* M*. Let* $F \subseteq S$ *be a set of final states for* M *such that all non final states* $s \notin F$ *are non-blocking* $Act(s) \neq \emptyset$ *and let* $X \colon F \to \mathbb{R}^{\infty}_{\geq 0}$ *be a random variable over final states.*
The expected value $\mathbb{E}(X)$ *is the least solution to the Bellman equation.*

[30, 31] attempted to prove it. Unfortunately the proofs have some hard to spot mistakes. However, [49] provides us instead with the following theorem.

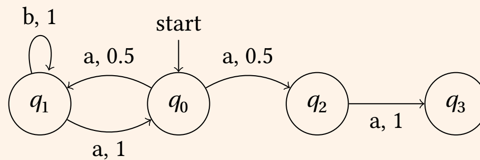
Theorem 4.1.2 (Least Bellman Solution of Random Variables for Finitly Branching MDPs [49]) *Let* $(S, Act, \mathbf{P}, s_0)$ *be a Markov decision process* M*. Let* $F \subseteq S$ *be a set of final states for* M *such that all non final states* $s \notin F$ *are non-blocking* $Act(s) \neq \emptyset$ *and such that* M *is finitely branching and let* $X \colon F \to \mathbb{R}^{\infty}_{\geq 0}$ *be a random variable.*
The expected value $\mathbb{E}(X)$ *is the least solution to the Bellman equation.*

This result is unfortunately not strong enough for the crucial non-determinism used in this chapter, as allocation is always infinitely branching non-deterministic if there are infinitely many unallocated locations in the heap.

Example 4.1.5 We consider the Markov decision process from Example 4.1.4. We added a new state q_3 that serve as final state. Moreover, we removed the transition from q_2 to q_0. Otherwise, the *minimal probability* to reach the final state q_3 would have been 0, as we can always choose to not go into the final state.



In the above Markov decision process we let the set of final states be $F = \{ q_3 \}$ and calculate the expected value of $X(q_3) = 1$, that is, the minimal probability to reach q_3. The scheduler achieving the minimal value is the scheduler that takes first a and then b. Indeed, in this case, the program can never reach q_3 and thus we have $\mathbb{E}(X) = 0$.
Consider instead the following Markov decision process:



Again, we let the set of final states be $F = \{q_3\}$ and calculate the expected value of $X(q_3) = 1$. The scheduler archiving the minimal value is the scheduler that chooses b in q_1 (and everywhere else has no choice). Now, the program will take action a in the first step. With probability 0.5 we definitely reach q_3. The rest of the probability mass ends up in the loop in q_1 which is evaluated to 0. Thus we have $\mathbb{E}(X) = 0.5$.
It is easy to verify that both examples also satisfy the Bellman-equation.

The least solution of the Bellman-equation guarantees that non-terminating paths are assigned the least element 0.

4.2. Operational Semantics for Probabilistic Programs

As detailed out in the previous section, we will model probabilistic programs as Markov decision processes and their functional correctness as expected values of random variables over final states. The Markov decision process has — similar to the transition systems in Chapter 3 — pairs of programs and stack heap pairs as the Markov decision process states.

To mimic the notation from Chapter 3, we will use a special notation for the transition probability function:

$$\mathbf{P}(s, a, s') = p \iff s \xrightarrow[\text{a}]{p} s'$$

This sometimes hides the behavior of the transition to be a function and not a relation. Again, to stress this, the notation $s \xrightarrow[\text{a}]{p} s'$ denotes a function that maps the parameters s, a and s' to the value p.

The programming language for probabilistic programs contains all statements of cPL except for concurrency, but with probabilistic branching.

Definition 4.2.1 (Probabilistic Programming Language) *We define the probabilistic programming language* pPL *as the language generated by this grammar:*

$\mathscr{C}$	::=	$\texttt{skip}$	*empty program*
	|	$x \texttt{ := } e$	*assignment*
	|	$\texttt{<}e\texttt{>} \texttt{ := } e'$	*heap mutation*
	|	$x \texttt{ := } \texttt{<}e\texttt{>}$	*heap lookup*
	|	$x \texttt{ := CAS}(e_1, e_2, e_3)$	*compare and set*
	|	$x \texttt{ := new}(e)$	*allocation*
	|	$\texttt{free}(e, e')$	*deallocation*
	|	$\texttt{if } (e_b)\ \{\mathscr{C}\}\ \texttt{else}\ \{\mathscr{C}\}$	*conditional branching*
	|	$\texttt{pif } \left(e_p\right)\ \{\mathscr{C}\}\ \texttt{else}\ \{\mathscr{C}\}$	*probabilistic branching*
	|	$\texttt{while}(e_b)\{\mathscr{C}\}$	*looping*
	|	$\mathscr{C}\,;\mathscr{C}$	*program composition*

where $e, e', e_1, e_2, e_3 \in \mathsf{ValExpr}$ *are value expressions,* $e_b \in \mathsf{BoolExpr}$ *are Boolean expressions and* $e_p : \mathsf{Stacks} \times \mathsf{LStacks} \to [0, 1]$ *are probabilistic expressions. We call the set of probabilistic expressions* ProbExpr. *We use* ↓ *to depict terminated programs and* ↯ *to depict erroneous programs.*

$$\frac{}{\texttt{skip}, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s,h)}\ \text{SKIP} \qquad \frac{}{x \mathrel{:=} e, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s\,[x \coloneqq e(s)]\,, h)}\ \text{ASSIGN}$$

$$\frac{e(s) \in \mathsf{dom}\,(h)}{x \mathrel{:=} \texttt{<}e\texttt{>}, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s\,[x \coloneqq h(e(s))]\,, h)}\ \text{LOOKUP} \qquad \frac{e(s) \notin \mathsf{dom}\,(h)}{x \mathrel{:=} \texttt{<}e\texttt{>}, (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{LOOKUP-ABT}$$

$$\frac{e(s) \in \mathsf{dom}\,(h)}{\texttt{<}e\texttt{>} \mathrel{:=} e', (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s, h\,[e(s) \coloneqq e'(s)])}\ \text{MUT} \qquad \frac{e(s) \notin \mathsf{dom}\,(h)}{\texttt{<}e\texttt{>} \mathrel{:=} e', (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{MUT-ABT}$$

$$\frac{e_1(s) \in \mathsf{dom}\,(h) \qquad h(e_1) \neq e_2(s)}{x \mathrel{:=} \text{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s\,[x \coloneqq 0]\,, h)}\ \text{CAS-FALSE}$$

$$\frac{e_1(s) \in \mathsf{dom}\,(h) \qquad h(e_1) = e_2(s)}{x \mathrel{:=} \text{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s\,[x \coloneqq 1]\,, h\,[e_1(s) \coloneqq e_3(s)])}\ \text{CAS-TRUE}$$

$$\frac{e_1(s) \notin \mathsf{dom}\,(h)}{x \mathrel{:=} \text{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{CAS-ABT}$$

$$\frac{e(s) = n \in \mathbb{N} \qquad \ell, \ldots, \ell + n - 1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}\,(h) \qquad h' = h \cup \{\ell \mapsto 0, \ldots, \ell + n - 1 \mapsto 0\}}{x \mathrel{:=} \mathsf{new}\,(e)\,, (s,h) \xrightarrow[\text{Alloc } \ell]{1} \downarrow, (s\,[x \coloneqq \ell]\,, h')}\ \text{ALLOC}$$

$$\frac{e(s) \notin \mathbb{N}}{x \mathrel{:=} \mathsf{new}\,(e)\,, (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{ALLOC-ABT}$$

$$\frac{e(s) + 0, \ldots\, e(s) + e'(s) - 1 \in \mathsf{dom}\,(h) \qquad h' = h\,[e(s) + 0 \coloneqq \mathrm{undef}] \ldots [e(s) + e'(s) - 1 \coloneqq \mathrm{undef}]}{\texttt{free}(e, e'), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s, h')}\ \text{FREE}$$

$$\frac{\exists i \in \{0, \ldots, e'(s) - 1\}\,.\, e(s) + i \notin \mathsf{dom}\,(h)}{\texttt{free}(e, e'), (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{FREE-ABT}$$

Figure 4.1.: Operational semantics for all statements in pPL. Heap manipulating statements may abort due to the accessed location not being allocated. Compare and set changes the value at a certain location only if the value is as expected.

Probabilistic expressions map states to values in the closed interval between 0 and 1. As in Chapter 3, we do not give a concrete syntax for these expressions for reasons of brevity.

Probabilistic branching allows us to pick one branch *randomly*. That is, the evaluated value of the probabilistic "guard" gives the probability to execute the left branch and the remaining probability mass is the probability to execute the right branch. This already is sufficient to implement a lot of distributions and probabilistic programs, especially all distributions that a probabilistic Turing machine can compute.

Example 4.2.1 We will use the following program in pPL as a running example for this chapter:

```
x = 1;
y = 0;
while(x = 1){
    pif (0.5) {
        x := 0
    }else{
        x := 1
    };
    y' := new(1);
    <y'> := y;
    y := y'
}
```

In this program, we loop until we (probabilistically) assign 0 to x. This happens with a probability of 0.5 in each iteration. We also create a list in this loop, whose size is incremented every iteration by one. Thus the interesting question in this example is the *expected size* of the list after termination.

The semantics of the language defined in Definition 6.1.1 can be seen in Figures 4.1 and 4.2. Whenever we do not assign any value to a transition, we assume it to be 0. We use the action "Det" for transitions that are either deterministic or probabilistic. We use the other (infinite set of) actions "Alloc ℓ" for the non-deterministic allocation statement. Actions are enabled, if we define some non-zero probability transition for them.

A relevant property of the operational semantics presented here is that they are actually a Markov decision process. For this, we need to check that for every enabled action, the probability adds up to one.

Theorem 4.2.1 *The semantics from Figures 4.1 and 4.2 yield a Markov decision processes for every initial state, that is, for all* pPL *programs C and $(s, h) \in$ States and for every enabled action $a \in Act(C, (s, h))$ we have:*

$$\sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p = 1.$$

Proof. By induction over C. □

Example 4.2.2 The program from Example 4.2.1 has unfortunately an infinite state space for the initial state where $s(z) = 0$ for all $z \in$ Vars and $h = \emptyset$ as we can reach each state with a list of arbitrary size. The following graph depicts the structure of an induced Markov chain, where q_n represents the state before the while loop and where the heap is of the form $h = \{j_n \mapsto j_{n-1}, \ldots, j_0 \mapsto 0\}$ for some heap locations $j_0, \ldots j_n$. Moreover, q'_n is the corresponding terminated

$$\frac{C_1' \neq \lightning \qquad C_1, (s,h) \xrightarrow[\text{a}]{p} C_1', (s',h')}{C_1 \,;\, C_2, (s,h) \xrightarrow[\text{a}]{p} C_1' \,;\, C_2, (s',h')}\ \text{SEQ} \qquad \frac{}{\downarrow \,;\, C_2, (s,h) \xrightarrow[\text{Det}]{1} C_2, (s,h)}\ \text{SEQ-END}$$

$$\frac{C_1, (s,h) \xrightarrow[\text{a}]{p} \lightning, (s,h)}{C_1 \,;\, C_2, (s,h) \xrightarrow[\text{a}]{p} \lightning, (s,h)}\ \text{SEQ-ABT} \qquad \frac{}{\lightning \,;\, C_2, (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)}\ \text{SEQ-ABT-2}$$

$$\frac{e_b(s) = \mathsf{true}}{\mathtt{if}\ (\, e_b \,)\ \{C_1\}\ \mathtt{else}\ \{C_2\}, (s,h) \xrightarrow[\text{Det}]{1} C_1, (s,h)}\ \text{IF-T} \qquad \frac{e_b(s) = \mathsf{false}}{\mathtt{if}\ (\, e_b \,)\ \{C_1\}\ \mathtt{else}\ \{C_2\}, (s,h) \xrightarrow[\text{Det}]{1} C_2, (s,h)}\ \text{IF-F}$$

$$\frac{e_p(s) = p}{\mathtt{pif}\ \left(e_p\right)\ \{C_1\}\ \mathtt{else}\ \{C_2\}, (s,h) \xrightarrow[\text{Det}]{p} C_1, (s,h)}\ \text{PIF-T}$$

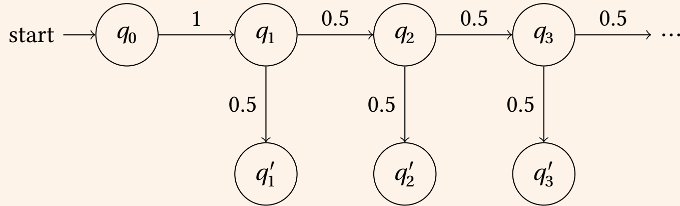
$$\frac{e_p(s) = p}{\mathtt{pif}\ \left(e_p\right)\ \{C_1\}\ \mathtt{else}\ \{C_2\}, (s,h) \xrightarrow[\text{Det}]{1-p} C_2, (s,h)}\ \text{PIF-F}$$

$$\frac{e_b(s) = \mathsf{true}}{\mathtt{while}(\, e_b \,)\{C_1\}, (s,h) \xrightarrow[\text{Det}]{1} C_1 \,;\, \mathtt{while}(\, e_b \,)\{C_1\}, (s,h)}\ \text{WHILE-T}$$

$$\frac{e_b(s) = \mathsf{false}}{\mathtt{while}(\, e_b \,)\{C_1\}, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s,h)}\ \text{WHILE-F}$$

Figure 4.2.: Operational Semantics for all flow related program constructs in pPL. Sequencing is defined inductively, branching and looping are defined transitionally. Probabilistic branching executes either path, but only with a certain probability and is the only statement having probabilistic behavior. In any case, we immediately transition to an abort statement when an underlying command fails.

state with the same heap.



In the Markov decision process, every allocation would yield infinitely many successor states, one for every free location. In every induced Markov chain, we have one infinite path. This infinite path, however, has probability 0. Moreover, the expected size of the list is

$$\sum_{i \in \mathbb{N}_{>0}} i \cdot 0.5^i \quad = \quad 2.$$

4.3. Quantitative Logic

Our goal in this section is to describe a logic similar to qualitative separation logic from Chapter 3 that enabled us to express both *random variables* and *expected values*. Indeed, it turns out that in programs, the expected value of a random variable is again a random variable. Why is that? A random variable is a map from stack heap pairs to non-negative reals or infinity and the expected value of

such a random variable depends on the initial state of the program, thus it is also a map from stack heap pairs to non-negative reals or infinity. Indeed, we can also break down a program into two sequentially composed parts and consider the expected value of the latter part to be the random variable for the computation of the first part. In this sense, every object we will argue with, is a random variable and some of them are also expected values. The literature usually uses the other choice of nomenclature and calls these objects *expectations* instead of random variables as all but one are expected values of a random variable. The logic we will use to express these random variables is called *quantitative separation logic* (QSL). The version we present here is based on [30].

Definition 4.3.1 (Random Variables in Quantitative Separation Logic) *A random variable in quantitative separation logic* $X\colon \mathsf{States} \times \mathsf{LStacks} \to \mathbb{R}^{\infty}_{\geq 0}$ *is a map from stack heap pairs and values of logical variables to extended non-negative reals. We denote the set of random variables in quantitative separation logic as* RV^{∞}.

To express these random variables, we use operations from fuzzy logics [50]. We can cast propositions into 0 and 1 — to 0 if the stack heap pair satisfies the proposition and to 1 if stack heap pair does not satisfy the proposition. We lift conjunction and disjunction to minimum and maximum respectively, we lift quantifiers to suprema and infima respectively and so on and so forth. However, since we want to express values outside of the interval between 0 and 1 in order to express sizes of data structures and expected values, our domain needs to be the non-negative extended reals, that is the non-negative reals with positive infinity. In this domain, some operations, unfortunately, do not exist. Especially, we do not have a meaningful operation for *negation*.

Multiplication by infinity is infininty except when multiplied by zero, then its zero.

Additionally to the lifting of Boolean operations, we require addition and multiplication to express weighted sums. Indeed, a weighted sum is essential for encoding probabilistic branching. We already observed in Chapter 3, how a logic can have multiple conjunctions. It turns out, that multiplication in our domain is also such an additional possible candidate for conjunction. Indeed, multiplication behaves conservative towards logical conjunction if we identify 0 with false and 1 with true. This observation gives rise to define a separating operation that uses multiplication instead of minimum. At this point, this choice seems arbitrary. However, multiplication is usually more useful to express probabilities and expectations in practice. We thus get this definition for the *separating multiplication* in quantitative separation logic:

We use different symbols for Fuzzy/Quantiative separation logic ($\star$) and qualitative separation logic ($*$) to make them sufficiently differentiable.

$$(X \star Y)(s,h,\eta) \quad = \quad \sup\{X(s,h_1,\eta)\cdot Y(s,h_2,\eta) \mid h_1 \perp h_2 \wedge h = h_1 \cup h_2\}.$$

Like in qualitative separation logic, the separating multiplication in QSL has its adjoint, which we confusingly also call the magic wand — or to differentiate them the *quantitative magic wand*:

$$(X \mathrel{-\!\!\star} Y)(s,h,\eta) \quad = \quad \inf\left\{ \frac{Y(s,h\cup h',\eta)}{X(s,h',\eta)} \,\middle|\, h \perp h' \right\}.$$

We define for all a that division by ∞ is zero, i.e. that $\frac{a}{\infty} = 0$ and division of infinity by some nonzero and non-infinite value is infinity, i.e. that $\frac{\infty}{a} = \infty$.

A list with connectives in qualitative separation logic and their semantics can be found in Table 4.1.

An entailment in this logic is the pointwise lifting of the less-than-or-equal

Table 4.1.: This is a list for semantics for various operations in qualitative separation logic. (s,h) is a stack heap pair. Logical variables η are now explicitly part of the parameters. Iverson brackets $[\Phi]$ allow the use of quantitative separation logic. $Z\colon \mathbb{N} \to \mathsf{RV}^\infty$ is a map from natural numbers to random variables.

X	$X(s,h,\eta)$
$[\Phi]$	1 if $(s,h) \models_\eta \Phi$ and 0 else
$\langle e \rangle$	$e(s,\eta)$
$Y[x := e]$	$Y(s[x := e(s)], h, \eta)$
$Y_1 \sqcap Y_2$	$\min\{Y_1(s,h,\eta),\ Y_2(s,h,\eta)\}$
$Y_1 \sqcup Y_2$	$\max\{Y_1(s,h,\eta),\ Y_2(s,h,\eta)\}$
$\mathsf{S}a.\, Y$	$\sup\{Y(s,h,\eta[a := v]) \mid v \in \mathbb{Q}\}$
$\mathsf{I}a.\, Y$	$\inf\{Y(s,h,\eta[a := v]) \mid v \in \mathbb{Q}\}$
$Y_1 + Y_2$	$Y_1(s,h,\eta) + Y_2(s,h,\eta)$
$Y_1 \cdot Y_2$	$Y_1(s,h,\eta) \cdot Y_2(s,h,\eta)$
$Y_1 \star Y_2$	$\sup\{Y_1(s,h_1,\eta) \cdot Y_2(s,h_2,\eta) \mid h_1 \perp h_2 \wedge h = h_1 \cup h_2\}$
$\bigstar_{i=0}^{n} Z$	$\begin{cases} [\mathsf{emp}] & \text{if } n = -1 \\ Z(n) \star \bigstar_{i=0}^{n-1} Z & \text{if } n > -1 \end{cases}$
$Y_1 \mathrel{-\!\!\star} Y_2$	$\inf\left\{ \frac{Y_2(s,h \cup h',\eta)}{Y_1(s,h',\eta)} \,\middle|\, h \perp h' \right\}$

relation on the extended reals.

Definition 4.3.2 (Entailments in QSL) *Let $X, Y \in \mathsf{RV}^\infty$ be random variables in QSL. The entailment in QSL, denoted as $X \models Y$, is:*

$$X \models Y \iff \text{for all } (s,h), \eta, \text{ we have that } X(s,h,\eta) \le Y(s,h,\eta).$$

Example 4.3.1 Consider the following entailment:

$$[1 \mapsto 2] \star [2 \mapsto 3] \models [\mathsf{true}] \star \mathsf{S}a.\ [1 \mapsto a]$$

We note, that we use here Iverson brackets to transform a qualitative separation logic proposition like $1 \mapsto 2$ into a QSL random variable $[1 \mapsto 2]$.
To prove that the entailment above holds, we apply the definition of entailments and have now to prove for arbitrary s, h and η that

$$([1 \mapsto 2] \star [2 \mapsto 3])(s,h,\eta) \le ([\mathsf{true}] \star \mathsf{S}a.\ [1 \mapsto a])(s,h,\eta).$$

we first eliminate the separating multiplication on the left side. We thus obtain that there are heaps h_1, h_2 with $h_1 \perp h_2$ and $h = h_1 \cup h_2$ and need to prove that

$$[1 \mapsto 2](s,h_1,\eta) \cdot [2 \mapsto 3](s,h_2,\eta) \le ([\mathsf{true}] \star \mathsf{S}a.\ [1 \mapsto a])(s,h,\eta).$$

We have $h_1 = \{1 \mapsto 2\}$ and $h_2 = \{2 \mapsto 3\}$ as otherwise the left side would be zero and the states satisfies the entailment. With these assignments, the left side is 1. We thus need to prove that the right side is also (at least) 1. Eliminating the right separating multiplication, we choose h_2 for the left part and h_1 for the right part and thus have left to prove that

$$1 \le [\mathsf{true}](s,h_2,\eta) \cdot (\mathsf{S}a.\ [1 \mapsto a])(s,h_1,\eta).$$

$[\mathrm{true}]\,(s, h_2, \eta)$ is vacuously one and thus we are left with

$$1 \leq (\mathfrak{S}a.\ [1 \mapsto a])(s, h_1, \eta).$$

Eliminating the supremum by choosing $a = 2$, i.e. using $\eta' = \eta\,[a \coloneqq 2]$, we finally obtain

$$1 \leq [1 \mapsto 2]\,(s, h_1, \eta') = 1,$$

which holds by reflexivity of the less-than-or-equal relation.

Example 4.3.2 Consider the following entailment:

$$\langle 0.4 \rangle \cdot [1 \mapsto 2] + \langle 0.6 \rangle \cdot [1 \mapsto 3] \models \mathfrak{S}a.\ [1 \mapsto a]$$

This entailment does not contain any separating connectives. It does however illustrate that the logic is quantitative. Again we eliminate our entailment relation and have to prove for arbitrary s, h and η that

$$(\langle 0.4 \rangle \cdot [1 \mapsto 2] + \langle 0.6 \rangle \cdot [1 \mapsto 3])(s, h, \eta) \leq (\mathfrak{S}a.\ [1 \mapsto a])(s, h, \eta).$$

We eliminate the piecewise operations on the left and obtain

$$0.4 \cdot [1 \mapsto 2]\,(s, h, \eta) + 0.6 \cdot [1 \mapsto 3]\,(s, h, \eta) \leq (\mathfrak{S}a.\ [1 \mapsto a])(s, h, \eta).$$

Since $[1 \mapsto 2]\,(s, h, \eta)$ is only one if $h = \{1 \mapsto 2\}$ and $[1 \mapsto 3]\,(s, h, \eta)$ is only one if $h = \{1 \mapsto 3\}$ and both cannot hold at the same time, we have that the left side is either 0, 0.4 or 0.6. If the left side is 0, the states vacuously satisfies the entailment. Thus we have two cases left:

It is common in these quantitative logics to make a case distinction when dealing with addition.

- We have to prove that with $h = \{1 \mapsto 2\}$ we have

  $$0.4 \leq (\mathfrak{S}a.\ [1 \mapsto a])(s, h, \eta).$$

  Eliminating the supremum on the right by choosing the value $a = 2$, i.e. using $\eta' = \eta\,[a \coloneqq 2]$, we have that

  $$0.4 \leq [1 \mapsto 2]\,(s, h, \eta') = 1,$$

  which holds trivially.
- We have to prove that with $h = \{1 \mapsto 3\}$ we have

  $$0.6 \leq (\mathfrak{S}a.\ [1 \mapsto a])(s, h, \eta).$$

  Eliminating the supremum on the right by choosing the value $a = 3$, i.e. using $\eta'' = \eta\,[a \coloneqq 3]$, we have that

  $$0.6 \leq [1 \mapsto 3]\,(s, h, \eta'') = 1,$$

  which holds trivially.

As with separating conjunction, we also have commutativity and associativity for the separating multiplication.

Theorem 4.3.1 (Commutativity and Associativity [30]) *Let $X_1, X_2, X_3 \in \mathsf{RV}^\infty$ be random variables in QSL. The following statements hold:*

$$X_1 \star X_2 \models X_2 \star X_1 \qquad \text{(Commutativity)}$$
$$X_1 \star (X_2 \star X_3) \models (X_1 \star X_2) \star X_3 \qquad \text{(Associativity)}$$

Proof. See [43] at `LeanFSL.SL.QuantitativeProofrules`. □

Quantitative separation logic shares a lot of properties with qualitative separation logic. This is not surprising, since qualitative separation logic behaves *conservative* towards qualitative separation logic. That means, if we do not use any real valued atoms and use the obvious analogous operations, an entailment in the one logic holds if and only if the entailment holds in the other. Formally we can phrase this like the following theorem.

Theorem 4.3.2 (Conservativity [30]) *Let $\Phi, \Psi \in \mathsf{SLProp}$ be SL propositions. We have:*

- $([\Phi] \sqcap [\Psi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \Phi \wedge \Psi$,
- $([\Phi] \sqcup [\Psi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \Phi \vee \Psi$,
- $(\mathsf{S}\, a.\ [\Phi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \exists a.\, \Phi$,
- $(\mathsf{I}\, a.\ [\Phi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \forall a.\, \Phi$,
- $([\Phi] + [\Psi])(s, h, \eta) \geq 1$ *if and only if* $(s, h) \models_\eta \Phi \vee \Psi$,
- $([\Phi] \cdot [\Psi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \Phi \wedge \Psi$,
- $([\Phi] \star [\Psi])(s, h, \eta) = 1$ *if and only if* $(s, h) \models_\eta \Phi * \Psi$, *and*
- $([\Phi] \mathbin{-\!\!\star} [\Psi])(s, h, \eta) \geq 1$ *if and only if* $(s, h) \models_\eta \Phi \mathbin{-\!\!*} \Psi$.

Infimum is not conservative towards all quantification in this sense if the type we quantifier over is empty as in that case the infimum would yield ∞ and not 1. The rationals are, however, nonempty.

Proof. See [43] at `LeanFSL.SL.Conservativity`. □

As for qualitative separation logic, we also have monotonicity, modus ponens and adjointness for quantitative separation logic. Monotonicity allows us to eliminate separating multiplication when they are on both sides of the entailment and have appropriate left- and right-hand sides. Adjointness allows us to eliminate quantitative magic wands on the right side of an entailment. This is fully analogous to the adjointness in qualitative separation logic and is possible as multiplication and division are (except for null values) adjoint. Modus Ponens allows us to eliminate quantitative magic wands on the left side of entailments. Contrary to qualitative separation logic, there are now actually *two reasons* that modus ponens is eliminating. That is, if we eliminate a magic wand on the left side, we also loose access to the premise we used to eliminate it. For once, we have that the operation is spatial: Removing a heap using the separating operation and recreating it using the magic wand is eliminating. Secondly, we have that multiplication with division is also eliminating, i.e. for $b \neq 0$ we have that $\frac{a}{b} \cdot b = a$ but not necessarily that $\frac{a}{b} \cdot b \leq a \cdot b$.

Since $\frac{0.5}{0.5} \cdot 0.5 = 0.5$ but $\frac{0.5}{0.5} \cdot 0.5 \nleq 0.5 \cdot 0.5$.

Theorem 4.3.3 (Monotonicity, Adjointness and Modus Ponens [30]) *Let $X_1, X_2, X_3, X_4 \in \mathsf{RV}^\infty$ be random variables in QSL. We have that:*

$$X_1 \models X_2 \quad \textit{and} \quad X_3 \models X_4 \qquad \textit{then} \qquad X_1 \star X_3 \models X_2 \star X_4 \quad \text{(Monotonicity)}$$
$$X_1 \star X_2 \models X_3 \qquad \textit{iff} \qquad X_1 \models X_2 \mathrel{-\!\!\star} X_3 \quad \text{(Adjointness)}$$
$$X_1 \star (X_1 \mathrel{-\!\!\star} X_2) \models X_2 \quad \text{(Modus Ponens)}$$

Proof. See [43] at `LeanFSL.SL.QuantitativeProofrules`. □

Example 4.3.3 Consider the following entailment:

$$[1 \mapsto 2] \models [2 \mapsto 3] \mathrel{-\!\!\star} [1 \mapsto 2] \star [2 \mapsto 3]$$

Using adjointness, we have that

$$[1 \mapsto 2] \star [2 \mapsto 3] \models [1 \mapsto 2] \star [2 \mapsto 3]\,,$$

which holds by reflexivity of the entailment.
Without using adjointness, we can first eliminate the entailment and have to prove for arbitrary s, h, η that

$$[1 \mapsto 2]\,(s, h, \eta) \leq ([2 \mapsto 3] \mathrel{-\!\!\star} ([1 \mapsto 2] \star [2 \mapsto 3]))(s, h, \eta).$$

Eliminating the magic wand we get that we need to prove for a heap h' with $h \perp h'$ and $[2 \mapsto 3]\,(s, h', \eta) \neq 0$ that

$$[1 \mapsto 2]\,(s, h, \eta) \leq \frac{([1 \mapsto 2] \star [2 \mapsto 3])(s, h \cup h', \eta)}{[2 \mapsto 3]\,(s, h', \eta)}.$$

The division is well-defined as the denominator is not zero.

Using now the adjointness of the division, we can transform this into

$$[1 \mapsto 2]\,(s, h, \eta) \cdot [2 \mapsto 3]\,(s, h', \eta) \leq ([1 \mapsto 2] \star [2 \mapsto 3])(s, h \cup h', \eta).$$

Lastly, we eliminate the separating multiplication by choosing h and h' respectively and obtain

$$[1 \mapsto 2]\,(s, h, \eta) \cdot [2 \mapsto 3]\,(s, h', \eta) \leq [1 \mapsto 2]\,(s, h, \eta) \cdot [2 \mapsto 3]\,(s, h', \eta),$$

which holds by reflexivity.

Example 4.3.4 Consider the following entailment:

$$[2 \mapsto 3] \mathrel{-\!\!\star} [1 \mapsto 2] \star [2 \mapsto 3] \models [2 \mapsto 3] \mathrel{-\!\!\star} [\mathsf{true}] \star [2 \mapsto 3]$$

Using first adjointness, we receive the proof obligation

$$([2 \mapsto 3] \mathrel{-\!\!\star} [1 \mapsto 2] \star [2 \mapsto 3]) \star [2 \mapsto 3] \models [\mathsf{true}] \star [2 \mapsto 3]\,.$$

Using transitivity of the entailment and modus ponens, our next proof obligation is

$$[1 \mapsto 2] \star [2 \mapsto 3] \models [\mathsf{true}] \star [2 \mapsto 3]\,.$$

Using monotonicity of the separating multiplication, reflexivity and that for

all $(s, h), \eta$ we have $[1 \mapsto 2]\,(s, h, \eta) \leq 1$, the entailment holds.
Without using adjointness or modus ponens, we first unfold the definition of entailments and have to prove for arbitrary s, h, η that

$$([2 \mapsto 3] \mathrel{-\!\!\star} [1 \mapsto 2] \star [2 \mapsto 3])(s, h, \eta) \\ \leq ([2 \mapsto 3] \mathrel{-\!\!\star} [\mathsf{true}] \star [2 \mapsto 3])(s, h, \eta).$$

We eliminate the magic wand on the right-hand side and need to prove that for an arbitrary h' with $h \perp h'$ and $[2 \mapsto 3]\,(s, h', \eta) \neq 0$ we have that

$$([2 \mapsto 3] \mathrel{-\!\!\star} [1 \mapsto 2] \star [2 \mapsto 3])(s, h, \eta) \leq \frac{([\mathsf{true}] \star [2 \mapsto 3])(s, h \cup h', \eta)}{[2 \mapsto 3]\,(s, h', \eta)}.$$

Since $[2 \mapsto 3]\,(s, h', \eta) \neq 0$, we have that $h' = \{2 \mapsto 3\}$. On the left-hand side, we also eliminate the magic wand by choosing h'. Thus we now need to prove that

$$\frac{([1 \mapsto 2] \star [2 \mapsto 3])(s, h \cup h', \eta)}{[2 \mapsto 3]\,(s, h', \eta)} \leq \frac{([\mathsf{true}] \star [2 \mapsto 3])(s, h \cup h', \eta)}{[2 \mapsto 3]\,(s, h', \eta)}.$$

We can now reduce the fraction as the fraction on both sides have the same denominator. We are left with

$$([1 \mapsto 2] \star [2 \mapsto 3])(s, h \cup h', \eta) \leq ([\mathsf{true}] \star [2 \mapsto 3])(s, h \cup h', \eta).$$

We will now shorten $h \cup h'$ as h''. Eliminating the separating multiplication on the left-hand side, we have some h_1, h_2 with $h_1 \perp h_2$ and $h'' = h_1 \cup h_2$ and need to prove that

$$[1 \mapsto 2]\,(s, h_1, \eta) \cdot [2 \mapsto 3]\,(s, h_2, \eta) \leq ([\mathsf{true}] \star [2 \mapsto 3])(s, h'', \eta).$$

We can eliminate the right separating multiplication by choosing h_1 and h_2 respectively and are left with

$$[1 \mapsto 2]\,(s, h_1, \eta) \cdot [2 \mapsto 3]\,(s, h_2, \eta) \leq [\mathsf{true}]\,(s, h_1, \eta) \star [2 \mapsto 3]\,(s, h_2, \eta).$$

We have that $h_1 = \{1 \mapsto 2\}$ and $h_2 = \{2 \mapsto 3\}$ as otherwise the left-hand side would be zero and the state satisfies the entailment vacuously. Then we also have that the right-hand side evaluates to 1 and thus by reflexivity, the entailment holds.

4.4. Axiomatic Semantics for Probabilistic Programs

In literature, this calculus is usually refered to as *weakest preexpectation.*

In a similar manner how we defined the weakest precondition semantics in Chapter 3, we will now define the *weakest expectation* of some random variable. In this calculus, the *expectation* refers to the expected value of the random variable and takes the role of the precondition. The random variable takes the role of the postcondition. The prefix “weakest” still reflects its origin from the weakest precondition calculi. It computes the greatest lower bound over all possible expected values for an initial value. We choose the lower bound as we assume the worst schedule for allocation. This is in contrast to concurrency, where we usually like to reason about bounds on all possible schedules. Since the weakest expectation is not liberal, we do not take non-terminating runs in consideration. More precisely, our random variables need to assign 0 to non-terminating runs.

On the other side, we need to make sure that the weakest expectation is memory safe. That is, non-memory safe runs should also be assigned the value 0. This is achieved by requiring that runs finishing in $\lightning$ are also assigned 0. Thus random variables may only assign values to runs finishing in $\downarrow$. We therefore give no option to assign values to different programs, but only to the stack heap pairs (s, h). Choosing some arbitrary assignment of values for logical variables, we naturally have quantitative separation logic as a foundation for specifying random variables. As the expected value of this random variable is parametrized by the initial state — which is again a stack heap pair (s, h) — we also have that quantitative separation logic is suited for specifying the expected value. This concludes that we can express the relation between the random variable, the program and their expected value relation similar to Chapter 3 as triples.

As we aim to give an axiomatic semantics for over-approximating the expected value, we require over-approximating rules for each of the statements. Unfortunately, the rule for allocation does not translate nicely with our current assumptions. The problem is that we never required finiteness of the heap until now. If the heap is total — i.e. every location is allocated — allocation will not be able to allocate anything. In the semantics, this is encoded as dead-locking. Dead-locking yields the top element in the weakest expectation as we take the greatest lower bound of all next actions. Since there are no further actions, the top element is the greatest lower bound. However, since this path has no final state and thus no reward, we will account for it with 0 as the expected value. Thus we do not yield equality between the weakest expectation operation and the actual expected value defined earlier. However, restricting the programs to finite heaps guarantees us this equality. To yield valid over-approximations, we will thus require now that heaps have a finitely defined domain. This ensures that we have never empty action sets for allocation statements.

Definition 4.4.1 (Weakest Expectation) *Let C be a* pPL *program and $X \in \mathrm{RV}^\infty$ be a random variable in QSL. We define the weakest expectation* **we** *as the least solution of the following equation over finite heaps:*

$$\mathsf{we}[\![C]\!]\,(X) \quad = \quad \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \mathsf{step}[\![C]\!]\,\big(\lambda C'.\,\mathsf{we}[\![C']\!]\,(X)\big) & \text{else} \end{cases}$$

where $\mathsf{step}[\![C]\!]\,(f)$ *for a* pPL *program C, $(s, h) \in \mathsf{States}$, where h is finite, and a function f mapping programs to quantitative separation logic is defined as*

$$\begin{aligned} & \mathsf{step}[\![C]\!]\,(f)\,(s, h, \eta) \\ = \quad & \inf\left\{ \sum_{(s,h)\xrightarrow[a]{p} C',(s',h') \text{ and } C' \neq \lightning} p \cdot f(C')(s, h, \eta) \;\middle|\; a \in \mathit{Act}(C, (s, h)) \right\}. \end{aligned}$$

Unsurprisingly, the equation on which **we** is defined, is a variant of the Bellman-equation we saw in Section 4.1. We do not include any resource invariant in this version, as we do not feature concurrency. As such, we also do not need the overhead with adding shared heap to local heap before each step and removing the shared heap again after each step.

Example 4.4.1 We compute the value of

$$\mathrm{we}[\![\texttt{pif}\ (0.5)\ \{x\ \texttt{:=}\ 1\}\ \texttt{else}\ \{x\ \texttt{:=}\ 2\}]\!]\,(\langle x\rangle),$$

that is, an upper bound to the expected value of x after executing the program. The program itself assigns randomly with probability 0.5 the values 1 or 2 to x. We thus expect the following equation to hold:

$$\mathrm{we}[\![\texttt{pif}\ (0.5)\ \{x\ \texttt{:=}\ 1\}\ \texttt{else}\ \{x\ \texttt{:=}\ 2\}]\!]\,(\langle x\rangle) = \langle 1.5\rangle$$

Unfolding the fixed point of **we** twice on some arbitrary (s, h) and η, we have

$$\begin{aligned}
&\mathrm{we}[\![\texttt{pif}\ (0.5)\ \{x\ \texttt{:=}\ 1\}\ \texttt{else}\ \{x\ \texttt{:=}\ 2\}]\!]\,(\langle x\rangle)\,(s, h, \eta) \\
&= 0.5 \cdot \mathrm{we}[\![\downarrow]\!]\,(\langle x\rangle)\,(s\,[x := 1]\,, h, \eta) + 0.5 \cdot \mathrm{we}[\![\downarrow]\!]\,(\langle x\rangle)\,(s\,[x := 2]\,, h, \eta) \\
&= 0.5 \cdot \langle x\rangle(s\,[x := 1]\,, h, \eta) + 0.5 \cdot \langle x\rangle(s\,[x := 2]\,, h, \eta) \\
&= 0.5 \cdot 1 + 0.5 \cdot 2 = 1.5.
\end{aligned}$$

Similar to Chapter 3, we define triples (and not quadruples due to the lack of the resource invariant) which encode that a value is an *upper bound* on the weakest expectation.

Definition 4.4.2 (Upper Bound Specification for pPL Programs) *Let C be a* pPL *program, $X \in \mathrm{RV}^{\infty}$ a parametrized bound on the expected value and $Y \in \mathrm{RV}^{\infty}$ a random variable. We define:*

$$\{X\}_{\geq}\ C\ \{Y\} \qquad \Longleftrightarrow \qquad \mathrm{we}[\![C]\!]\,(Y) \models X.$$

Example 4.4.2 Consider the specification

$$\{\langle 1.5\rangle\}_{\geq}\ \texttt{pif}\ (0.5)\ \{x\ \texttt{:=}\ 1\}\ \texttt{else}\ \{x\ \texttt{:=}\ 2\}\ \{\langle x\rangle\}.$$

This is equivalent to

$$\mathrm{we}[\![\texttt{pif}\ (0.5)\ \{x\ \texttt{:=}\ 1\}\ \texttt{else}\ \{x\ \texttt{:=}\ 2\}]\!]\,(x) \models \langle 1.5\rangle.$$

We already know the value of the left-hand side for an arbitrary stack heap pair and values for logical variables, so we can replace it to

$$\langle 1.5\rangle \models \langle 1.5\rangle,$$

which holds by reflexivity of the entailment relation.

We choose here to only consider over-approximations of the expected value as we consider a least fixed point. Indeed, since the fixed point is the least one, we can choose any pre-fixed point for looping programs to obtain an over-approximation. Under-approximations for least fixed points on the other hand, are challenging. We depict our proof rules for the specifications following Definition 4.4.2 in Figure 4.3. A lot of the proof rules in Figure 4.3 are very similar to the ones we introduced in Chapter 3 and indeed also hold for very similar reasons.

For once, we dropped rules for concurrency and resource invariants as those are not present or needed in this programming language. However, some notable differences we explain in the following are the different while loop, the lack of a

framing rule, the change from the consequence rule to the monotonicity rule and the missing of a maximum rule. Moreover, we have now a rule for eliminating addition called linearity.

The proof rule for while loops now include the dual version for least fixed points. Since our semantics are defined as a least fixed point and we require that our invariant is a pre-fixed point as all pre-fixed points are upper bounds to the least fixed point. This is exactly what our proof rule encodes. This is different to the weakest resource-safe liberal precondition, which is defined as a greatest fixed point and thus is lower bounded by all post-fixed points.

The lack of the framing rule is a bit more intricate. Indeed, **we** allows the framing rule

$$\mathrm{we}[\![C]\!]\,(X) \star Y \models \mathrm{we}[\![C]\!]\,(X \star Y)\,,$$

given that $\mathrm{Write}(C) \cap \mathrm{Vars}\,(Y) = \emptyset$, where $\mathrm{Write}(C)$ and $\mathrm{Vars}\,(Y)$ are defined analogous to the qualitative case. Unfortunately, the entailment is the wrong way around. Framing allows us to compute lower bounds and not upper bounds. Indeed, if we instead constructed axiomatic semantics for lower bounding, the framing rule could be included. However, reasoning about lower bounds for loops has increase difficulty instead. In case one requires lower bounds, there are also proof rules for loops that allow proving lower bounds for least fixed point semantics, making an axiomatic semantics for lower bound reasoning also feasible [51].

Linearity allows us to eliminate addition from our specification. However, we require that the program does not feature allocation. This is because addition does not nicely distribute over infimum for upper bounds — which we introduce when we have any non-probabilistic non-determinism. With allocation, we can only hope for lower bounds similar to the framing rule:

$$\langle a \rangle \cdot \mathrm{we}[\![C]\!]\,(Y_1) + \mathrm{we}[\![C]\!]\,(Y_2) \models \mathrm{we}[\![C]\!]\,(\langle a \rangle \cdot Y_1 + Y_2)$$

Since allocation is the only source for non-probabilistic non-determinism, having programs without allows us to use linearity of the expected value.

The consequence rule is now called *monotonicity*. This is because the relevant property for this is the monotonicity of the **we** underlying the triple. However, we may also notice that the entailments switched around. This is because we now compute upper bounds and thus it is sufficient to show the proof obligation for a greater random variable and for a lesser expected value. This results in the entailment sequence

$$\mathrm{we}[\![C]\!]\,(Y) \models \mathrm{we}[\![C]\!]\,(Y') \models X' \models X,$$

where the first entailment follows from monotonicity. The whole sequence of entailments justify the monotonicity proof rule.

The lack of a rule for eliminating maxima in the triples rely on the missing distributivity between addition and the maximum and minimum operation. Instead, we have sub- and super-distributivity respectively. That is, we have

$$\min\{a,\ b\} + \min\{c,\ d\} \leq \min\{a + c,\ b + d\}$$
$$\max\{a,\ b\} + \max\{c,\ d\} \geq \max\{a + c,\ b + d\}$$

This limits how we can use them with the weakest expectation. The maximum allows a proof rule for lower bounds and the minimum allows a proof rule for

$$\frac{}{\{X\}_{\geq}\ \texttt{skip}\ \{X\}}\ \text{skip} \qquad \frac{}{\{X[x := e]\}_{\geq}\ x\ \texttt{:=}\ e\ \{X\}}\ \text{assign}$$

$$\frac{}{\{\text{Ƨ}a.\ [e \mapsto a] \star ([e \mapsto a] \mathrel{-\!\!\star} X[x := a])\}_{\geq}\ x\ \texttt{:=}\ \texttt{<e>}\ \{X\}}\ \text{lookup}$$

$$\frac{}{\left\{(\text{Ƨ}a.\ [e \mapsto a]) \star \left([e \mapsto e'] \mathrel{-\!\!\star} X\right)\right\}_{\geq}\ \texttt{<e>}\ \texttt{:=}\ e'\ \{X\}}\ \text{mut}$$

$$\frac{}{\left\{[e \in \mathbb{N}] \cdot \mathrm{I}\,a.\ \mathop{\bigstar}_{i=0}^{e-1} [a+i \mapsto 0] \mathrel{-\!\!\star} X[x := a]\right\}_{\geq}\ x\ \texttt{:=}\ \texttt{new}(e)\ \{X\}}\ \text{alloc}$$

$$\frac{}{\left\{[e' \in \mathbb{N}] \cdot \left(\mathop{\bigstar}_{i=0}^{e'-1} \text{Ƨ}a.\ [e+i \mapsto a]\right) \star X\right\}_{\geq}\ \texttt{free}(e, e')\ \{X\}}\ \text{dealloc}$$

$$\frac{}{\{[e_l \mapsto e_c] \star ([e_l \mapsto e_s] \mathrel{-\!\!\star} X[x := 1])\}_{\geq}\ x\ \texttt{:=}\ \texttt{CAS}(e_l, e_c, e_s)\ \{X\}}\ \text{cas-true}$$

$$\frac{}{\{\text{Ƨ}a.\ ([e_l \mapsto a] \cdot [\neg a = e_c]) \star ([e_l \mapsto a] \mathrel{-\!\!\star} X[x := 0])\}_{\geq}\ x\ \texttt{:=}\ \texttt{CAS}(e_l, e_c, e_s)\ \{X\}}\ \text{cas-false}$$

$$\frac{\{X_1\}_{\geq}\ C_1\ \{Y\} \qquad \{X_2\}_{\geq}\ C_2\ \{Y\}}{\{[e_b] \cdot X_1 \sqcup [\neg e_b] \cdot X_2\}_{\geq}\ \texttt{if}\ (e_b)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{Y\}}\ \text{if}$$

$$\frac{\{X_1\}_{\geq}\ C_1\ \{Y\} \qquad \{X_2\}_{\geq}\ C_2\ \{Y\}}{\{\langle e\rangle \cdot X_1 + \langle 1-e\rangle \cdot X_2\}_{\geq}\ \texttt{pif}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{Y\}}\ \text{pif}$$

$$\frac{\{X\}_{\geq}\ C_1\ \{Y\} \qquad \{Y\}_{\geq}\ C_2\ \{Z\}}{\{X\}_{\geq}\ C_1 ; C_2\ \{Z\}}\ \text{seq} \qquad \frac{[e_b] \cdot X \sqcup [\neg e_b] \cdot Y \models I \qquad \{X\}_{\geq}\ C\ \{I\}}{\{I\}_{\geq}\ \texttt{while}(e_b)\{C\}\ \{Y\}}\ \text{while}$$

$$\frac{\{X\}_{\geq}\ C\ \{Y\} \qquad \{X'\}_{\geq}\ C\ \{Y'\}}{\{X \sqcap X'\}_{\geq}\ C\ \{Y \sqcap Y'\}}\ \text{min}$$

$$\frac{a \in \mathbb{R}_{\geq 0}^{\infty} \qquad C \text{ does not contain allocation} \qquad \{X_1\}_{\geq}\ C\ \{Y_1\} \qquad \{X_2\}_{\geq}\ C\ \{Y_2\}}{\{\langle a\rangle \cdot X_1 + X_1\}_{\geq}\ C\ \{\langle a\rangle \cdot Y_1 + Y_2\}}\ \text{linearity}$$

$$\frac{X' \models X \qquad \{X'\}_{\geq}\ C\ \{Y'\} \qquad Y \models Y'}{\{X\}_{\geq}\ C\ \{Y\}}\ \text{monotonicity}$$

Figure 4.3.: Axiomatic semantics using the notation defined in Definition 4.4.2. We use here semantics similar to denotational semantics, as they are usually depicted denotational and not axiomatic. The first proof rules deal with atomic statements, followed by flow related programs and lastly elimination proof rules.

upper bounds. Since we aim for upper bounds, and since the minimum also is distributive over infimum, we gain the minimum rule.

We do not go over all proof rules from Figure 4.3, but instead only explain the proof rule for mutation and for compare and set. The proofs and more details for the proof rules (sometimes stated slightly different) can be found in [30, 31].

Allocation x := new (e) generates a number of $e(s, \eta)$ allocations initialized with zero. If e is not a natural number, the program crashes which we assign with reward zero. The minimal expected value of X after executing x := new (e) is the minimal value of X where h has additionally e allocated with zero assigned locations and additionally the stack has x assigned the first location. We can encode this by taking the location, that minimizes the value where we add e many locations to the heap using the magic wand. Since the left side of that magic wand is quantitative it behaves similar to the qualitative magic wand. We

encode changing the value of x by a substitution on X.

Compare and set $x := \mathrm{CAS}(e_l, e_c, e_s)$ is a composure of mutation, lookups and assignments. When the location e_l has the value e_c, we need to lookup the value of e_l without changing it, but changing the value of x to 0. Thus we use a rule similar to the lookup rule together with the comparison $[\neg a \neq e_c]$, which compares the value at location e_l with the value e_c and sets the value of x to zero similarly to the assignment rule. If however the values are the same, we instead mutate the value at location e_l to e_s similarly to the mutation rule and set x to one similar to the assignment rule.

We will not look deeper into the soundness proofs (and will not prove any more general statement) for other proof rules and instead in later chapters focus on one key problem of this axiomatic semantics in Chapter 9: How to prove the entailments that result from the axiomatic semantics? We will see there one technique that enables creating proof systems for entailments involving inductively defined data structures that require some user input but are otherwise automatic.

Theorem 4.4.1 (Soundness [30]) *The proof rules from Figure 4.3 are sound.*

We will use annotated programs for the expectation specification triples similar to Chapter 3. Here we use the annotations with a greater-than-or-equal symbol to highlight that we prove upper bounds like

$$\begin{array}{l} \quad /\!/_{\geq}\ X \\ C \\ \quad /\!/_{\geq}\ Y \end{array}$$

to express that

$$\{X\}_{\geq}\ C\ \{Y\}.$$

Contrary to the annotated program annotation in Chapter 3, this notation does not lead to confusing or wrong interpretations. This is because we can not share resources anymore. We do, however, need to keep in mind that we are over-approximating expected values. This results in different semantics than in Chapter 3, as there we were under-approximating the precondition.

Example 4.4.3 We reconsider the program from Example 4.2.1:

```
x = 1; y = 0;
while(x = 1){
    pif (0.5) {x := 0} else {x := 1};
    y' := new(1);
    <y'> := y;
    y := y'
}
```

We program allocates a list of arbitrary size. One may think that this violates our assumption of finite heaps. However, our program only allocated arbitrary sized finite heaps, and never actually reaches the limit heap which would be

infinite. Thus we can continue reasoning about this program with finite heaps only.

For our specification, we introduce a predicate [ls] that holds if and only if the heap is a list. That is, a stack heap pair (s, h) with values for logical variables η satisfies the predicate $(s, h) \models_\eta$ [ls] (x) if $h = \{ s(x) \mapsto \ell_1, \dots \ell_{n-1} \mapsto \ell_n, \ell_n \mapsto 0 \}$ or $s(x) = 0$ and $h = \emptyset$. We will use the quantitative lifting of the list predicates using Iverson brackets: [ls] (x). Moreover, we introduce the predicate size, which represents exactly the size of the heap. That is, we define the predicate size as $\text{size}(s, h, \eta) = |h|$.

We now aim to prove the specification

$$\{ \langle 2 \rangle \}_{\geq} \; C \; \{ \text{size} \cdot [\text{ls}]\,(y) \},$$

that is, we create a list in the program, and its size is in expectation at most 2. The following annotated program shows the proof outline for the specification:

$$
\begin{array}{l}
\quad /\!\!/\!\!/_{\geq} \;\; \langle 2 \rangle \cdot [\text{emp}] \\
x = 1\,; \\
\quad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot [\text{emp}] \sqcup [x = 1] \cdot \langle 2 \rangle \cdot [\text{emp}] \\
\quad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot \text{size} \cdot [\text{ls}]\,(0) \sqcup [x = 1] \cdot (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(0) \\
y = 0\,; \\
\quad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot \text{size} \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(y) \\
\texttt{while}\,(\,x = 1\,)\,\{ \\
\qquad /\!\!/\!\!/_{\geq} \;\; (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(y) \\
\qquad /\!\!/\!\!/_{\geq} \;\; \langle 0.5 \rangle \cdot (\text{size} + \langle 1 \rangle) \cdot [\text{ls}]\,(y) + \langle 0.5 \rangle \cdot (\text{size} + \langle 3 \rangle) \cdot [\text{ls}]\,(y) \\
\qquad \texttt{pif}\;(\,0.5\,)\,\{\,x := 0\,\}\;\texttt{else}\;\{\,x := 1\,\}\,; \\
\qquad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot (\text{size} + \langle 1 \rangle) \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 3 \rangle) \cdot [\text{ls}]\,(y) \\
\qquad y' := \texttt{new}\,(1)\,; \\
\qquad /\!\!/\!\!/_{\geq} \;\; \begin{cases} \;\;\; [x \neq 1] \cdot ((\text{size} + \langle 1 \rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S}a.\ [y' \mapsto a]) \\ \sqcup\ [x = 1] \cdot ((\text{size} + \langle 3 \rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S}a.\ [y' \mapsto a]) \end{cases} \\
\qquad \texttt{<}y'\texttt{>} := y \\
\qquad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot \text{size} \cdot [\text{ls}]\,(y') \sqcup [x = 1] \cdot (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(y') \\
\qquad y := y' \\
\qquad /\!\!/\!\!/_{\geq} \;\; [x \neq 1] \cdot \text{size} \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(y) \\
\} \\
\quad /\!\!/\!\!/_{\geq} \;\; \text{size} \cdot [\text{ls}]\,(y)
\end{array}
$$

We will go through the proof from bottom to top (and leave out some details related to heap-reasoning for brevity). We start with the random variable for which we want to prove an upper bound on its expected value. The random variable asserts a list starting from location y and is evaluated with its size. We are thus asking for the *expected size* of the data structure. We need to find an invariant (sometimes called super-invariant as we aim for upper bounds). This invariant is

$$I \quad = \quad [x \neq 1] \cdot \text{size} \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 2 \rangle) \cdot [\text{ls}]\,(y).$$

Some authors prefer to use addition instead of maximimum here. Since the left and right sides are mutually exclusive, the addition and maximum behaves equivalently here.

It states intuitively that either if $x = 0$, then there is nothing that will be done and thus its expected value is just the random variable itself. However, if $x = 1$, we will execute the loop, which executes a geometric distribution to randomly select a list. This will then yield an addition of 2 list elements in expectation. Let us verify this. First we need to find a value X for $\{X\}_{\geq}\ C_{body}\ \{I\}$, where C_{body} is the loops body. The annotated program already suggests to pick $X = (\text{size} + \langle 2\rangle) \cdot [\text{ls}]\,(y)$. We will skip over most of the proof and instead focus on two parts and lastly the entailment that the loop rule requires.

- We investigate the specification from the following annotated program:

$$\begin{aligned}
&/\!\!/_{\geq}\ \ [x \neq 1] \cdot (\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y) \\
&y' := \text{new}\,(1)\ ; \\
&/\!\!/_{\geq}\ \begin{cases} \quad [x \neq 1] \cdot ((\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} a.\ [y' \mapsto a]) \\ \sqcup\ [x = 1] \cdot ((\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} a.\ [y' \mapsto a]) \end{cases}
\end{aligned}$$

For the validity of this, we need to reason that the following entailment holds:

$$\begin{aligned}
&([1 \in \mathbb{N}_{>0}]) \cdot \mathrm{I}\, a.\ [a \mapsto 0] \mathrel{-\!\!\star} \\
&\qquad [x \neq 1] \cdot ((\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} b.\ [a \mapsto b]) \\
&\quad \sqcup [x = 1] \cdot ((\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} b.\ [a \mapsto b]) \\
&\models\ [x \neq 1] \cdot (\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)
\end{aligned}$$

Firstly, we can drop $[1 \in \mathbb{N}_{>0}]$ as 1 is a natural number and thus the Iverson bracket evaluates to one. Next we eliminate the infimum and now need to provide an a such that

$$\begin{aligned}
&[a \mapsto 0] \mathrel{-\!\!\star} \\
&\qquad [x \neq 1] \cdot ((\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} b.\ [a \mapsto b]) \\
&\quad \sqcup [x = 1] \cdot ((\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)) \star (\mathfrak{S} b.\ [a \mapsto b]) \\
&\models\ [x \neq 1] \cdot (\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)
\end{aligned}$$

Indeed, it does not matter, which a we choose, as long as a is not allocated in the heap. Choosing such one, we first introduce it using the magic wand and later remove it again due to the subpart $\cdots \star (\mathfrak{S} b.\ [a \mapsto b])$. Indeed, the supremum there is only non-zero if $b = 0$, thus choosing this value. Together, we can eliminate both operations with each other and have left

$$\begin{aligned}
&\quad [x \neq 1] \cdot ((\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y)) \\
&\sqcup [x = 1] \cdot ((\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y)) \\
&\models\ [x \neq 1] \cdot (\text{size} + \langle 1\rangle) \cdot [\text{ls}]\,(y) \sqcup [x = 1] \cdot (\text{size} + \langle 3\rangle) \cdot [\text{ls}]\,(y),
\end{aligned}$$

which holds by reflexivity.

- We also investigate the specification from the following annotated program:

$$\begin{aligned}
&/\!\!/\!_{\geq}\ (\mathrm{size} + \langle 2\rangle) \cdot [\mathrm{ls}]\,(y) \\
&/\!\!/\!_{\geq}\ \langle 0.5\rangle \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) + \langle 0.5\rangle \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y) \\
&\texttt{pif}\ (0.5)\,\{x := 0\}\ \texttt{else}\ \{x := 1\}\,; \\
&/\!\!/\!_{\geq}\ [x \neq 1] \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) \sqcup [x = 1] \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y)
\end{aligned}$$

  The lower part boils down to the entailment

$$\begin{aligned}
&\langle 0.5\rangle \cdot [0 \neq 1] \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) \\
&\quad \sqcup [0 = 1] \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y) \\
&+ \langle 0.5\rangle \cdot [1 \neq 1] \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) \\
&\quad \sqcup [1 = 1] \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y) \\
\models\ &\langle 0.5\rangle \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) + \langle 0.5\rangle \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y),
\end{aligned}$$

  which holds by eliminating $[0 = 1]$ and $[1 \neq 1]$ which are always zero, eliminating $[0 \neq 1]$ and $[1 = 1]$, which are always one and propagating the zero to the maximum, at which point we can eliminate the maximum. The upper part requires us to prove the entailment

$$\begin{aligned}
&\langle 0.5\rangle \cdot (\mathrm{size} + \langle 1\rangle) \cdot [\mathrm{ls}]\,(y) + \langle 0.5\rangle \cdot (\mathrm{size} + \langle 3\rangle) \cdot [\mathrm{ls}]\,(y) \\
\models\ &(\mathrm{size} + \langle 2\rangle) \cdot [\mathrm{ls}]\,(y).
\end{aligned}$$

  We can use distributivity and have to proof at the end that $\langle 0.5\rangle \cdot \mathrm{size} + \langle 0.5\rangle \cdot \mathrm{size} \models \mathrm{size}$, which holds, and that $\langle 0.5\rangle \cdot \langle 1\rangle + \langle 0.5\rangle \cdot \langle 3\rangle \models \langle 2\rangle$, which also holds.
- Lastly we have to prove that the invariant is actually an invariant, thus we have to prove that

$$\begin{aligned}
&[x = 1] \cdot (\mathrm{size} + \langle 2\rangle) \cdot [\mathrm{ls}]\,(y) \sqcup [x \neq 1] \cdot \mathrm{size} \cdot [\mathrm{ls}]\,(y) \\
\models\ &[x \neq 1] \cdot \mathrm{size} \cdot [\mathrm{ls}]\,(y) \sqcup [x = 1] \cdot (\mathrm{size} + \langle 2\rangle) \cdot [\mathrm{ls}]\,(y),
\end{aligned}$$

  which holds by commutativity of maximum.

Thus we have proven that 2 is an upper bound of the expected value of the list's size.

5. Fuzzy Separation Logic

In the previous chapter, we investigated a logic mapping states to the extended non-negative reals. This enabled us to reason about both, probabilities and about expected values. It has, however, the downside that reasoning about loops is easier for *upper bounds*, while the *framing rule* for which separation logic is prominent, does not yield upper bounds. A solution to this is considering liberal semantics. For liberal semantics, we assign the top element of our domain to non-terminating runs. If we continue to use random variables mapping to non-negative reals, we immediately run into the problem, that programs which are not certainly terminating yield the expected value infinite. While this could itself make up calculi of interest, we take the typical route from the literature [52, 53] and restrict our domain instead to values in the unit interval, i.e. to values in the closed interval from 0 to 1. In this domain, we assign the value 1 to non-terminating runs. This enables us to reason with liberal semantics about probabilities of conditions, where we always add the probability of non-termination on top. That is, if the program is almost-surely terminating for at least one schedule, our liberal semantics yields the expected value of the given random variable. Moreover, these liberal semantics grant us simple proof rules for *lower bounds* and thus let us apply the frame rule. Finally, this setting also admits proof rules for concurrent programs, thus yielding a combination of the ideas from Chapters 3 and 4. We will deal with these considerations in more detail in Chapter 6 and instead focus on *representing random variables on the unit interval as a separation logic* in this chapter first.

Certain termination requires that every possible run is finite. Runs with probability zero also need to be finite.

In fuzzy logics, propositions map to values between zero and one. For our application, this is ideal to represent probabilities depending on some initial valuation of variables. By lifting operations from classical logic to real values, we obtain the maximum as a logical or, the minimum as a logical and, suprema for existence quantifiers and infima for universal quantifiers. Contrary to the last chapter, we now also have negation. Indeed, $1 - a$ is a useful interpretation for negation. However, for probability theory, weighted sums $a \cdot b + (1 - a) \cdot c$ are crucial as well. We now have two possibilities to deal with weighted sums:

1. We can introduce an operator with three parameters for weighted sums. The three parameters are the weight, the left side of the addition and the right side of the addition. This has the advantage that the operation is well defined. For every three parameters in the unit interval, the result of this operation is again a value in the unit interval. It has, however, also one big drawback. We now are always required to express random variables in this operation — which is cumbersome.
2. We can also introduce a *truncated addition.* This addition cuts off values to one which exceed one. Indeed, this operation still has a lot of nice properties which are worth to be examined. Moreover, expressing random variables with this operation is easier, but less intuitive. For example, we lose distributivity over multiplication.

We decided to choose the second option and express operations in our logic when necessary as truncated versions. Truncated operations gives rise to a truncated version of arithmetic.

As an example, let us investigate the problem with distributivity of the truncated addition. We would like to have that for $a, b, c \in [0, 1]$ we have

$$a \cdot (b + c) \stackrel{?}{=} a \cdot b + a \cdot c.$$

Now we set $b = c = 1$ and $a = 0.5$. Then we have that

$$b + c = 1,$$

as, remember, the real valued addition results in 2. But the truncated addition truncates the value to 1. On the other hand, we have

$$a \cdot b = 0.5 \qquad a \cdot c = 0.5.$$

We have on the left side of our false distributivity law that

$$a \cdot (b + c) = a \cdot 1 = 0.5$$

and on the right side of our false distributivity law that

$$a \cdot b + a \cdot c = 0.5 + 0.5 = 1.$$

As you may already see here, subdistributivity still holds.

Together we unfortunately have falsified the distributivity law.

5.1. Truncated Arithmetic on the Unit Interval

Fuzzy logics have the unit interval [0,1] as its co-domain. In this section, we will introduce all necessary operations and statements about the unit interval sufficient to introduce fuzzy separation logic.

The maximum and minimum operations serve as liftings for Boolean disjunctions and conjunctions, respectively. Similarly, the supremum and infimum operations serve as liftings for the existence and universal quantifiers. Indeed, we already observed in Theorem 4.3.2 that those operations behave conservatively towards qualitative reasoning. We can establish a similar relation to fuzzy reasoning. For now, it suffices that maxima, minima, suprema and infima exist, i.e. that the unit interval is a complete linear order.

Theorem 5.1.1 *The unit interval [0,1] is a complete linear order.*

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

We already saw previously, that addition, subtraction and multiplication will play a crucial role in order to reason about probabilities, namely to state weighted sums of the form $a \cdot b + (1 - a) \cdot c$. We will consider all of these in detail.

Multiplication is well defined in the unit interval. That is, for any value in the unit interval a, b, we have that $a \cdot b \in [0, 1]$. This means of course, that we also inherit all properties of the multiplication from the reals. Moreover, we have that multiplication is monotone in both parameters on the unit interval.

Lemma 5.1.2 *The following statements hold:*

$$\forall a, b \in [0,1].\ a \cdot b \in [0,1] \qquad \text{(Well-Defined)}$$
$$\forall a, b \in [0,1].\ a \cdot b = b \cdot a \qquad \text{(Commutativity)}$$
$$\forall a, b, c \in [0,1].\ a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad \text{(Associativity)}$$
$$\forall a \in [0,1].\ 0 \cdot a = 0 \qquad \forall a \in [0,1].\ 1 \cdot a = 1 \qquad \text{(Zero and Neutral)}$$
$$\forall a_1, a_2, b_1, b_2 \in [0,1].\ a_1 \leq a_2 \wedge b_1 \leq b_2 \Rightarrow a_1 \cdot b_1 \leq a_2 \cdot b_2 \quad \text{(Monotonicity)}$$

Proof. See [54] and [43] at `LeanFSL.Mathlib.Probabilities`. □

We only have a multiplicative inverse for exactly one value in the unit interval — the value 1. However, we can still define a division, that will become useful later. This division is our first *truncated operator*. Division between two values in the unit interval always yields a positive (or undefined) value. Our strategy here is to truncate the value to one if it exceeds one and define the value of division by zero as one. All of this guarantees adjointness with respect to multiplication. We call this operator the *truncated division*.

Consider for example the inverse of 0.5, which is 2 and thus not in the unit interval

Definition 5.1.1 (Truncated Division) *We define the truncated division $a \mathbin{\dot{/}} b$ as*

$$a \mathbin{\dot{/}} b \quad = \quad \begin{cases} a/b & \text{if } a < b \\ 1 & \text{else.} \end{cases}$$

The truncated division is well-defined. The main property we like to have from this division is being adjoint towards multiplication — which it is. Zero in the first parameter is yielding zero if the second one is not zero, and zero in the second parameter is yielding one. One is behaving neutral in the second parameter. We can observe regular division behavior whenever the first parameter is less than the second and truncated behavior otherwise. Moreover, as ordinary division, it is monotone in the first and antitone in the second argument.

Lemma 5.1.3 *The following statements hold:*

$$\forall a, b \in [0,1].\ a \mathbin{\dot{/}} b \in [0,1] \qquad \text{(Well-Defined)}$$
$$\forall a, b, c \in [0,1].\ a \leq b \mathbin{\dot{/}} c \iff a \cdot c \leq b \qquad \text{(Adjointness)}$$
$$\forall a \in [0,1].\ a \neq 0 \Rightarrow 0 \mathbin{\dot{/}} a = 0 \qquad \forall a \in [0,1].\ a \mathbin{\dot{/}} 0 = 1 \qquad \text{(Zero Element)}$$
$$\forall a \in [0,1].\ a \mathbin{\dot{/}} 1 = a \qquad \text{(Neutral Element)}$$
$$\forall a, b \in [0,1].\ b \neq 0 \Rightarrow a \cdot b \mathbin{\dot{/}} b = a \qquad \text{(Cancelation)}$$
$$\forall a_1, a_2, b_1, b_2 \in [0,1].\ a_1 \leq a_2 \wedge b_1 \geq b_2 \Rightarrow a_1 \mathbin{\dot{/}} b_1 \leq a_2 \mathbin{\dot{/}} b_2$$
$$\text{(Antitonicity \& Monotonicity)}$$

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

Next, we move forward to addition and subtraction. For addition, the regular addition is unfortunately not well-defined. We will instead use the *truncated addition*. Similar to division, the truncated addition truncates every value exceeding one to one.

Definition 5.1.2 (Truncated Addition) *Let $a, b \in [0,1]$. We define the truncated addition $a \dotplus b$ as*

$$a \dotplus b \quad = \quad \min\{1,\ a+b\}\,.$$

The truncated addition is well-defined. Since zero is neutral for addition and this value is preserved under the minimum, zero is also a neutral element for truncated addition. For truncated addition, one is absorbing. Adding one to anything will always yield one. Indeed, this behavior is one of the main disadvantages as it is unintuitive to an inexperienced user. Truncated addition is also commutative, associative and monotone in both arguments.

Lemma 5.1.4 *The following statements hold:*

$$\forall a, b \in [0,1].\ a \dotplus b \in [0,1] \qquad \text{(Well-Defined)}$$
$$\forall a \in [0,1].\ a \dotplus 0 = a \qquad \text{(Neutral Element)}$$
$$\forall a \in [0,1].\ a \dotplus 1 = 1 \qquad \text{(Absorbing Element)}$$
$$\forall a, b \in [0,1].\ a \dotplus b = b \dotplus a \qquad \text{(Commutativity)}$$
$$\forall a, b, c \in [0,1].\ a \dotplus (b \dotplus c) = (a \dotplus b) \dotplus c \qquad \text{(Associativity)}$$
$$\forall a_1, a_2, b_1, b_2 \in [0,1].\ a_1 \leq a_2 \wedge b_1 \leq b_2 \Rightarrow a_1 \dotplus b_1 \leq a_2 \dotplus b_2 \qquad \text{(Monotonicity)}$$

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

Lemma 5.1.4 proves that $([0,1], \dotplus)$ is an ordered commutative monoid [55]. This is important as it allows us moving forward to infinite sums. For now, we will proceed with distributivity and subtraction and revisit this topic later.

When the truncated operations behave regular, we regain regular distributivity of course

We have certain sub- and superdistributivity laws with multiplication and truncated division. We already saw in the introduction to this chapter, that distributivity unfortunately does not hold for multiplication — neither does it hold for truncated division.

Lemma 5.1.5 *The following statements hold:*

$$\forall a, b, c \in [0,1].\ a \cdot (b \dotplus c) \leq a \cdot b \dotplus a \cdot c \qquad \text{(Subdistributivity)}$$
$$\forall a, b, c \in [0,1].\ b \mathbin{\dot{/}} a \dotplus c \mathbin{\dot{/}} a \leq (b \dotplus c) \mathbin{\dot{/}} a \qquad \text{(Superdistributivity)}$$

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

There are two kinds of subtraction, we will consider. The first is a unary subtraction, which subtracts a value from one. This subtraction is well-defined in the unit interval and serves as our basis for negation. The second kind of subtraction is binary *truncated subtraction* and is playing only a minor role in this dissertation.

Lemma 5.1.6 (One Minus Symmetries) *The following statements hold:*

$$\begin{aligned}
\forall a, b \in [0,1].\, a \leq 1 - b &\iff b \leq 1 - a \\
\forall a, b \in [0,1].\, 1 - a \leq b &\iff 1 - b \leq a \\
\forall a, b \in [0,1].\, a < 1 - b &\iff b < 1 - a \\
\forall a, b \in [0,1].\, 1 - a < b &\iff 1 - b < a
\end{aligned}$$

Proof. See [54]. □

Truncated subtraction on the other hand enables us to prove that summation is *canonic*. We call our addition canonic if we can define the less-or-equal-than order over the addition operator as

$$a \leq b \iff \exists c \in [0,1].\, a \dot{+} c = b.$$

To prove this, truncated subtraction is helpful as this allows us to define c in terms of a and b. Truncated subtraction truncates every negative value to zero.

Definition 5.1.3 (Truncated Subtraction) *We define for $a, b \in [0,1]$ the truncated subtraction $a \dot{-} b$ as*

$$a \dot{-} b \quad = \quad \max\{0,\ a - b\}.$$

In literature, this operation is usually called *monus*. For consistency reasons, we stick to the "truncated" naming scheme.

Truncated subtraction is well-defined and most importantly, it allows us (conditionally) to cancel addition.

Lemma 5.1.7 *The following statements hold:*

$$\forall a, b \in [0,1].\, a \dot{-} b \in [0,1] \qquad \text{(Well-Defined)}$$

$$\forall a, b \in [0,1].\, a \leq b \Rightarrow a \dot{+} (b \dot{-} a) = b \qquad \text{(Cancelation)}$$

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

This enables us to prove that summation is canonic.

Lemma 5.1.8 *The following statement holds:*

$$\forall a, b \in [0,1].\, a \leq b \iff \exists c.\, a \dot{+} c = b \qquad \text{(Canonicily)}$$

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. □

Example 5.1.1 Let us consider the inequality

$$0.5 \leq 1 - (a \cdot 0.5 \dot{+} 0.1) \dot{/} 0.25$$

and solve for a. First, we use Lemma 5.1.6 and obtain the proof goal

$$(a \cdot 0.5 \dot{+} 0.1) \dot{/} 0.25 \leq 1 - 0.5 = 0.5.$$

The division is unfortunately on the wrong side of the inequality in order to use adjointness. Thus, we instead make a case distinction:

$a \cdot 0.5 \dotplus 0.1 < 0.25$: Then we can use the statement converting truncated division to regular division from Lemma 5.1.3 so that we have

$$(a \cdot 0.5 \dotplus 0.1)/0.25 \leq 0.5 \iff a \cdot 0.5 \dotplus 0.1 \leq 0.5 \cdot 0.25 = 0.125.$$

Next we can equate $0.125 = \min\{1,\ 0.025 + 0.1\} = 0.025 \dotplus 0.1$, thus using monotonicity from Lemma 5.1.4 yields

$$a \cdot 0.5 \dotplus 0.1 \leq 0.125 = 0.025 \dotplus 0.1 \Leftarrow a \cdot 0.5 \leq 0.025 \wedge 0.1 \leq 0.1.$$

$0.1 \leq 0.1$ holds by reflexivity, leaving us with

$$a \cdot 0.5 \leq 0.025,$$

which holds when $a \leq 0.05$. Thus for $0 \leq a \leq 0.05$, the equation holds.

$a \cdot 0.5 \dotplus 0.1 \geq 0.25$: Then we can use Lemma 5.1.3 so that we have

$$(a \cdot 0.5 \dotplus 0.1) \mathbin{\dot{/}} 0.25 = 1 \leq 0.5,$$

which cannot hold, therefore there is no a satisfying these conditions.

Together, the inequality holds for $0 \leq a \leq 0.05$.

Example 5.1.2 Now we want to prove that the inequality

$$a \dotplus b \leq (0.5 \cdot a \dotplus 0.5 \cdot b) \mathbin{\dot{/}} 0.5$$

holds for all a and b. First, we can apply adjointness from Lemma 5.1.3 and commutativity from Lemma 5.1.2 to obtain the new proof goal

$$0.5 \cdot (a \dotplus b) \leq 0.5 \cdot a \dotplus 0.5 \cdot b.$$

The resulting inequality already holds because of Lemma 5.1.5. For didactic reasons, we will investigate the proof of sub-distributivity for this example.

$a + b \leq 1$: Thus we have that $a \dotplus b = a + b$ and therefore

$$0.5 \cdot (a \dotplus b) = 0.5 \cdot (a + b) = 0.5 \cdot a + 0.5 \cdot b$$

and are left to prove that

$$0.5 \cdot a + 0.5 \cdot b \leq 0.5 \cdot a \dotplus 0.5 \cdot b = \min\{1,\ 0.5 \cdot a + 0.5 \cdot b\}.$$

We can eliminate the minimum on the right side by proving that the left side is smaller than both of them. Since we have $a + b \leq 1$, we also have $0.5 \cdot a + 0.5 \cdot b \leq 1$. Lastly, we have

$$0.5 \cdot a + 0.5 \cdot b \leq 0.5 \cdot a + 0.5 \cdot b$$

by reflexivity, proving the inequality.

$a + b > 1$: Thus we have that $a \dotplus b = 1$ and therefore

$$0.5 \cdot (a \dotplus b) = 0.5 \cdot 1 = 0.5.$$

We now need to prove that

$$0.5 \leq 0.5 \cdot a \dotplus 0.5 \cdot b = \min\{1,\ 0.5 \cdot a + 0.5 \cdot b\}.$$

Again, we prove that both parameters of the minimum of the right side is at least the left. For 1 this is trivial as $0.5 \leq 1$. For the other side we have

$$0.5 = 0.5 \cdot 1 < 0.5 \cdot (a + b) = 0.5 \cdot a + 0.5 \cdot b,$$

thus proving the claim.

Infinite truncated summation is the sum over an infinite set using a function mapping the elements of the set to some added value. Finite summation is the same for finite sets. As truncated addition is a commutative monoid, it is easy to see that finite truncated summation is well-defined. That infinite truncated summation is also well-defined is not that easy. To work towards this goal, we first formally introduce finite truncated summation.

Definition 5.1.4 (Truncated Summation over Finite Sets) *Let $A \subseteq B$ be a finite set and $f\colon B \to [0,1]$ be a function. We define the finite truncated summation as*

$$\sum_{a \in A}^{\text{finite}} f(a) \quad = \quad \begin{cases} 0 & \text{if } A = \emptyset \\ f(b) \dotplus \sum\limits_{a \in A \setminus \{b\}}^{\text{finite}} f(a) & \text{for some } b \in A \end{cases}$$

As we have

$$f(b) \dotplus f(c) \dotplus \sum_{a \in A \setminus \{b,c\}}^{\text{finite}} f(a) = f(c) \dotplus f(b) \dotplus \sum_{a \in A \setminus \{c,b\}}^{\text{finite}} f(a),$$

due to commutativity and associativity, Definition 5.1.4 is well-defined.

In the Lean formalization [43] we use the formalization of Mathlib [54], which uses a lifting of summation on lists instead.

Infinite summation is the limit process on summation over all finite subsets. We can understand this limit over finite subsets as the value that all limits over sequences on finite subsets converging to the set take. In this definition we have two kinds of limits. A limit over sequences of unit interval values and a limit over sequences of (finite) subsets. The limit over sequences of unit interval values is standard. We say that

In the Lean formalization, we use a different formalization using filters from topology used in Mathlib [54].

$$\lim_{i \in \mathbb{N}} a_i = b \iff \forall \epsilon \in [0,1].\, \exists j \in \mathbb{N}.\, \forall j \leq i \in \mathbb{N}.\, |b - a_i| \leq \epsilon.$$

The limits on sequences of (finite) subsets is similar. Here the difference is measured in an element of the set. That is, we say for finite $A_i \subseteq B$ that

$$\lim_{i \in \mathbb{N}} A_i = B \iff \forall b \in B.\, \exists j \in \mathbb{N}.\, \forall j \leq i \in \mathbb{N}.\, b \in A_i.$$

If the limits for all sequences mapping finite subsets of a set B to some value do not yield the same value, the infinite limit over sets is undefined, and otherwise the value is this one unique value. We say that this limit is defined as

$$\lim_{A \to B} f(A) = b \iff \forall A_1, A_2, \dots \subseteq B.\ \lim_{i \to \infty} A_i = B \Rightarrow \lim_{i \to \infty} f(A_i) = b,$$

where A and all A_i are finite.

Definition 5.1.5 (Truncated Summation over Infinite Sets) *Let B be a set and $f: B \to [0,1]$ be a function. We define the infinite truncated summation as*

$$\dot{\sum_{a \in B}} f(a) \quad = \quad \lim_{A \to B} \sum_{a \in A}^{\text{finite}} f(a).$$

For simplicity, we will refer to the infinite truncated summation as just truncated summation. Indeed, for finite sets, both operators agree on the value. Now we still have the problem, that we do not know if the truncated sum is defined. Luckily, it is well defined for all possible functions $f: B \to [0,1]$! For the proof, we need two more ingredients. First, the finite truncated summation is monotone:

$$\forall A \subseteq A', \sum_{a \in A}^{\text{finite}} f(a) \leq \sum_{a \in A'}^{\text{finite}} f(a).$$

This follows from the fact that truncated addition is canonical as we established in Lemma 5.1.8. Assume we have $A \subseteq A'$, then there is some A'' with $A' = A \cup A''$ and A and A'' are disjoint, so then we have

$$\sum_{a \in A}^{\text{finite}} f(a) + \sum_{a \in A''}^{\text{finite}} f(a) = \sum_{a \in A'}^{\text{finite}} f(a) \Rightarrow \sum_{a \in A}^{\text{finite}} f(a) \leq \sum_{a \in A'}^{\text{finite}} f(a).$$

Secondly, since the finite truncated sum, which the limit takes its values of, is monotone and the unit interval [0,1] is a complete lattice, we can use a version of the monotone convergence theorem guaranteeing the existence of the limit.

Theorem 5.1.9 (Monotone Convergence Theorem) *Let B be a set and $f: Pow(B) \to [0,1]$ be a monotone function. Then we have that*

$$\lim_{A \to B} f(A) \quad = \quad \sup_{A \subseteq B} f(A).$$

Thus the limit on the left side is always defined.

[54] proves something much more general.

Proof. See [54]. ☐

Together, we can finally conclude that truncated summation is always well-defined, even for infinite sets:

Theorem 5.1.10 (Truncated Summation is Well-Defined) *Let B a set and $f: B \to [0,1]$ be a function. We have that*

$$\dot{\sum_{a \in B}} f(a) \in [0,1]$$

and thus is always defined.

Proof. See [43] at `LeanFSL.Mathlib.Probabilities`. ☐

Example 5.1.3 Consider the set

$$B \quad = \quad \{ s \in \text{Stacks} \mid s(x) \in \mathbb{N}, s(y) \in \{2,3\}, \forall z \notin \{x,y\}. s(z) = 0 \}.$$

We sum over this set using the mapping

$$f\colon B \to [0,1], s \mapsto \text{if } (s(x) = 1) \text{ then } \frac{1}{s(y)} \text{ else } 0.$$

Thus, we now want to compute the value

$$\dot{\sum_{s \in B}} f(s) \quad = \quad \dot{\sum_{s \in B}} \text{if } (s(x) = 1) \text{ then } \frac{1}{s(y)} \text{ else } 0.$$

As all null values can be ignored, we can filter the set B according to the condition $s(x) = 1$ and have

$$\dot{\sum_{s \in B}} \text{if } (s(x) = 1) \text{ then } \frac{1}{s(y)} \text{ else } 0 \quad = \quad \dot{\sum_{s \in B \smallsetminus \{ s \mid s(x) \neq 1 \}}} \frac{1}{s(y)}.$$

Since $B \smallsetminus \{ s \mid s(x) \neq 1 \}$ is finite, we have that

$$\dot{\sum_{s \in B \smallsetminus \{ s \mid s(x) \neq 1 \}}} \frac{1}{s(y)} \quad = \quad \frac{1}{2} \dot{+} \frac{1}{3} \quad = \quad \frac{5}{6}.$$

Example 5.1.4 Consider the set

$$B \quad = \quad \{ s \in \text{Stacks} \mid s(x) \in \mathbb{N}, s(y) \in \{2,3\}, \forall z \notin \{x,y\}. s(z) = 0 \}.$$

We sum over this set using the mapping

$$f\colon B \to [0,1], s \mapsto \frac{1}{s(y)} \cdot \frac{1}{2^{s(x)+2}}.$$

Thus, we now want to compute the value

$$\dot{\sum_{s \in B}} f(s) \quad = \quad \dot{\sum_{s \in B}} \frac{1}{s(y)} \cdot \frac{1}{2^{s(x)+2}}.$$

Since there are finitely many choices for y, we can rewrite it as the two sums

$$\dot{\sum_{s \in B}} \frac{1}{s(y)} \cdot \frac{1}{2^{s(x)+2}} \quad = \quad \dot{\sum_{s \in B \smallsetminus \{ s \mid s(y) \neq 2 \}}} \frac{1}{2} \cdot \frac{1}{2^{s(x)+2}} \dot{+} \dot{\sum_{s \in B \smallsetminus \{ s \mid s(y) \neq 3 \}}} \frac{1}{3} \cdot \frac{1}{2^{s(x)+2}}.$$

We will not formally prove this. However, since for $0 < a$

$$0 \leq \sum_{s \in B \smallsetminus \{ s \mid s(y) \neq 3 \}} \frac{1}{a} \cdot \frac{1}{2^{s(x)+2}} \leq 1,$$

we have that

$$\dot{\sum_{s \in B \smallsetminus \{ s \mid s(y) \neq 3 \}}} \frac{1}{a} \cdot \frac{1}{2^{s(x)+2}}$$

behaves regular, i.e. the same as its non-truncated counterpart. Thus we have

that

$$\begin{aligned} &\dot{\sum_{s \in B \setminus \{ s \mid s(y) \neq 3 \}}} \frac{1}{a} \cdot \frac{1}{2^{s(x)+2}} \\ =& \sum_{s \in B \setminus \{ s \mid s(y) \neq 3 \}} \frac{1}{a} \cdot \frac{1}{2^{s(x)+2}} \\ =& \frac{1}{a} \cdot \sum_{s \in B \setminus \{ s \mid s(y) \neq 3 \}} \frac{1}{2^{s(x)+2}}. \end{aligned}$$

This infinite sum is now the famous geometric sum and thus we have that

$$\frac{1}{a} \cdot \sum_{s \in B \setminus \{ s \mid s(y) \neq 3 \}} \frac{1}{2^{s(x)+2}} \quad = \quad \frac{1}{a} \cdot \frac{1}{2}.$$

Plugging this back into our equation, we have that

$$\dot{\sum_{s \in B \setminus \{ s \mid s(y) \neq 2 \}}} \frac{1}{2} \cdot \frac{1}{2^{s(x)+2}} \dotplus \dot{\sum_{s \in B \setminus \{ s \mid s(y) \neq 3 \}}} \frac{1}{3} \cdot \frac{1}{2^{s(x)+2}} \quad = \quad \frac{1}{2} \cdot \frac{1}{2} \dotplus \frac{1}{3} \cdot \frac{1}{2}.$$

Wrapping everything up, we have that

$$\dot{\sum_{s \in B}} f(s) \quad = \quad \frac{1}{2} \cdot \frac{1}{2} \dotplus \frac{1}{3} \cdot \frac{1}{2} \quad = \quad \frac{5}{12}.$$

5.2. Fuzzy Separation Logic

We will now introduce a fuzzy variant of quantitative separation logic. This logic maps states to values in the unit interval. We coin this logic *fuzzy separation logic* (FSL). Fuzzy separation logic gives rise to random variables mapping only to the unit interval. For brevity, we will also call these random variables when it is clear from the context that they map to the unit interval.

FSL is often called QSL in literature (e.g. [31]) as well. We pick different names to differentiate them.

Definition 5.2.1 (Random Variables in Fuzzy Separation Logic) *A random variable in fuzzy separation logic* $X\colon \mathsf{States} \times \mathsf{LStacks} \to [0,1]$ *is a map from stack heap pairs and values of logical variables to the unit interval. We denote the set of random variables in fuzzy separation logic as* RV^1.

Entailments in this logic are defined similar to the entailments in quantitative separation logic.

Definition 5.2.2 (Entailments in Fuzzy Separation Logic) *Let* $X, Y \in \mathsf{RV}^1$ *be random variables in FSL. The entailment in FSL* $X \models Y$ *is defined as*

$$X \models Y \iff \forall s, h, \eta.\, X(s,h,\eta) \leq Y(s,h,\eta).$$

We define operations in fuzzy separation logic in Table 5.1. We will explain these operations in the following in more detail.

In fuzzy separation logic, we have the same atomic random variables as in quantitative separation logic. The only difference is that we cannot introduce reals beyond one. Since our logic only allows for values in the closed interval

X	$X(s, h, \eta)$
$[\Phi]$	1 if $(s, h) \models_\eta \Phi$ and 0 else
$\langle e \rangle$	$e(s, \eta)$
$Y[x := e]$	$Y(s[x := e(s)], \eta)$
$\sim Y$	$1 - Y(s, h, \eta)$
$Y_1 \sqcap Y_2$	$\min\{Y_1(s, h, \eta),\ Y_2(s, h, \eta)\}$
$Y_1 \sqcup Y_2$	$\max\{Y_1(s, h, \eta),\ Y_2(s, h, \eta)\}$
$Ƨ a.\, Y$	$\sup\{Y(s, h, \eta[a := v]) \mid v \in \mathbb{Q}\}$
$\mathrm{I}\, a.\, Y$	$\inf\{Y(s, h, \eta[a := v]) \mid v \in \mathbb{Q}\}$
$Y_1 \dot{+} Y_2$	$Y_1(s, h, \eta) \dot{+} Y_2(s, h, \eta)$
$Y_1 \cdot Y_2$	$Y_1(s, h, \eta) \cdot Y_2(s, h, \eta)$
$Y_1 \star Y_2$	$\sup\{Y_1(s, h_1, \eta) \cdot Y_2(s, h_2, \eta) \mid h_1 \perp h_2 \land h = h_1 \cup h_2\}$
$\bigstar_{i=0}^{n} Z$	$\begin{cases} [\mathsf{emp}] & \text{if } n = -1 \\ Z(n) \star \bigstar_{i=0}^{n-1} Z & \text{if } n > -1 \end{cases}$
$Y_1 \mathrel{-\!\!\circledast} Y_2$	$\inf\{Y_2(s, h \cup h', \eta) \mathbin{\dot{/}} Y_1(s, h', \eta) \mid h \perp h'\}$

Table 5.1.: This is a list for semantics for various operations in fuzzy separation logic. (s, h) is a stack heap pair. Logical variables η are explicitly part of the parameters. Iverson brackets $[\Phi]$ allow the use of qualitative separation logic. $Z \colon \mathbb{N} \to \mathsf{RV}^1$ is a map from natural numbers to random variables.

from zero to one, every construct in the logic needs to guarantee that its image is in this interval. The operations we introduced in Section 5.1 will help us here as we now already have the required low level operations on the unit interval for these.

Namely, Iverson brackets cast a proposition Φ into a random variable which maps a stack heap pair (s, h) with values for logical variables η to one if they satisfy the proposition Φ and zero otherwise. We allow casting expressions mapping stacks s and values for logical variables η to values in the unit interval into a random variable in fuzzy separation logic.

The fuzzy version of quantitative separation logic now also supports a candidate for negation. We called this operator in Section 5.1 the "one minus" operator. Indeed, it holds that "one minus" $[\mathsf{true}]$ is mapped to $[\mathsf{false}]$. There are, however, some caveats. One may be wondering if our negation allows for the excluded middle. That is, whether we have that $[\mathsf{true}] \models X \sqcup \sim X$. This is unfortunately not the case. For this, consider $X(s, h, \eta) = 0.5$. Then we have $\sim X = X, X \sqcup X = X$ and $[\mathsf{true}] \not\models X$, disproving excluded middle. This also disproves any kind of weak excluded middle, in which we "just need to negate often enough", as X is a fixed point on the negation.

We also have the pointwise liftings of the lattice operations on the unit interval, namely the minimum on random variables $X \sqcap Y$, the maximum on random variables $X \sqcup Y$, the supremum on a random variable parametrized by a logical variable $Ƨ a.\, X(a)$ and the infimum on a random variable parametrized by a logical variable $\mathrm{I}\, a.\, X(a)$.

In our lean formalization, we allow any type instead of rationals for the supremum and infimum.

Next we add the arithmetic operators $X \dot{+} Y$ and $X \cdot Y$ lifting the operators from the previous section to random variables.

Example 5.2.1 Let us prove that the entailment

$$[\mathsf{emp}] \sqcup ([1 = x] \cdot \langle 0.5 \rangle) \models [\mathsf{emp}] \dotplus \langle 0.5 \rangle$$

holds. We use the definition of the entailment on fuzzy separation logic and have thus to prove that for all stack heap pairs (s, h) and values for logical variables η we have

$$([\mathsf{emp}] \sqcup ([1 = x] \cdot \langle 0.5 \rangle))(s, h, \eta) \leq ([\mathsf{emp}] \dotplus \langle 0.5 \rangle)(s, h, \eta).$$

We eliminate the left maximum by considering either choice separately.

1. We have to prove that

   $$[\mathsf{emp}]\,(s, h, \eta) \leq ([\mathsf{emp}] \dotplus \langle 0.5 \rangle)(s, h, \eta) = [\mathsf{emp}]\,(s, h, \eta) \dotplus 0.5.$$

   Since 0 is neutral towards truncated addition, the previous proof obligation is equivalent to

   $$[\mathsf{emp}]\,(s, h, \eta) \dotplus 0 \leq [\mathsf{emp}]\,(s, h, \eta) \dotplus 0.5.$$

   Using monotonicity of truncated addition, we have to prove that

   $$[\mathsf{emp}]\,(s, h, \eta) \leq [\mathsf{emp}]\,(s, h, \eta) \qquad \text{and} \qquad 0 \leq 0.5,$$

   which holds.
2. We have to prove that

   $$([1 = x] \cdot \langle 0.5 \rangle)(s, h, \eta) \leq ([\mathsf{emp}] \dotplus \langle 0.5 \rangle)(s, h, \eta).$$

   After simplification, we are left to prove that

   $$[1 = x]\,(s, h, \eta) \cdot 0.5 \leq [\mathsf{emp}]\,(s, h, \eta) \dotplus 0.5.$$

   Again, we use that 0 is neutral towards truncated addition and have that

   $$0 \dotplus [1 = x]\,(s, h, \eta) \cdot 0.5 \leq [\mathsf{emp}]\,(s, h, \eta) \dotplus 0.5.$$

   Using monotonicity of truncated addition, we are left with

   $$0 \leq [\mathsf{emp}]\,(s, h, \eta) \qquad \text{and} \qquad [1 = x]\,(s, h, \eta) \cdot 0.5 \leq 0.5.$$

   The left part vacuously holds. For the right part, we use that 1 is neutral towards multiplication. We thus transform the right into

   $$[1 = x]\,(s, h, \eta) \cdot 0.5 \leq 1 \cdot 0.5.$$

   Using monotonicity of multiplication, we are then finally left with

   $$[1 = x]\,(s, h, \eta) \leq 1 \qquad \text{and} \qquad 0.5 \leq 0.5,$$

   which holds.

The separation operation is defined analogously compared to the one from quantitative separation logic, thus we call it also separating multiplication. For a stack heap pair (s, h) and values for logical variables η, we define the separating multiplication $X \star Y$ as a supremum over the multiplication $X(s, h_1, \eta) \cdot Y(s, h_2, \eta)$. Each side receives a heap, h_1 and h_2 respectively, which are disjoint $h_1 \perp h_2$ and

the union of which is the original heap $h = h_1 \cup h_2$. Since the multiplication of two values in the unit interval is again in the unit interval, the separating multiplication of two random variables on the unit interval is also a random variable on the unit interval. Together, we have

$$(X \star Y)(s,h,\eta) \quad = \quad \sup\{X(s,h_1,\eta)\cdot Y(s,h_2,\eta) \mid h_1 \perp h_2 \wedge h = h_1 \cup h_2\}.$$

For the separating multiplication in fuzzy separation logic, we also have commutativity and associativity.

Theorem 5.2.1 (Commutativity and Associativity) *Let $X_1, X_2, X_3 \in \mathrm{RV}^1$ be random variables in FSL. The following statements hold:*

$$X_1 \star X_2 \models X_2 \star X_1 \qquad \text{(Commutativity)}$$
$$X_1 \star (X_2 \star X_3) \models (X_1 \star X_2) \star X_3 \qquad \text{(Associativity)}$$

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

The magic wand in fuzzy separation logic is defined slightly different compared to the quantitative magic wand. This is not surprising as the quantitative magic wand would be ill defined in fuzzy separation logic, as it has values exceeding one. Instead we make use of the truncated division to obtain the *truncated fuzzy magic wand*. For a stack heap pair (s,h) and values for logical variables η, the truncated fuzzy magic wand $X \mathrel{-\!\!\circledast} Y$ takes a heap h' that is disjoint to the original one $h \perp h'$. Then we take the infimum for all these heaps over the truncated division $Y(s, h\cup h', \eta) \mathbin{\dot{/}} X(s,h',\eta)$, dividing the right side with the union of these heaps $h \cup h'$ by the left side with the new heap h'. Together we have

$$(X \mathrel{-\!\!\circledast} Y)(s,h,\eta) \quad = \quad \inf\left\{Y(s,h\cup h',\eta) \mathbin{\dot{/}} X(s,h',\eta) \,\middle|\, h \perp h'\right\}.$$

Example 5.2.2 Let us prove the entailment

$$(Y_1 \dotplus Y_2) \models X \mathrel{-\!\!\circledast} ((X \sqcup [\mathsf{emp}]) \star (Y_1 \dotplus Y_2))$$

in fuzzy separation logic. First, we apply the definition of the entailment and thus now have to prove for any stack heap pair (s,h) and values for logical variables η that

$$(Y_1 \dotplus Y_2)(s,h,\eta) \le (X \mathrel{-\!\!\circledast} ((X \sqcup [\mathsf{emp}]) \star (Y_1 \dotplus Y_2)))(s,h,\eta).$$

We first consider the right side. For this, we introduce some arbitrary h' such that $h \perp h'$ in order to eliminate the truncated fuzzy magic wand. Thus we need to prove that

$$(Y_1 \dotplus Y_2)(s,h,\eta) \le ((X \sqcup [\mathsf{emp}]) \star (Y_1 \dotplus Y_2))(s,h\cup h',\eta) \mathbin{\dot{/}} X(s,h',\eta).$$

To eliminate the separating multiplication, we pick the heaps h' and h for the left and right side respectively, obtaining

$$(Y_1 \dotplus Y_2)(s,h,\eta) \le ((X \sqcup [\mathsf{emp}])(s,h',\eta)\cdot (Y_1 \dotplus Y_2)(s,h,\eta)) \mathbin{\dot{/}} X(s,h',\eta).$$

Now, let us do a case distinction on the value of $[\mathsf{emp}]\,(s,h',\eta)$

1. If $[\mathsf{emp}]\,(s, h', \eta) = 0$, then we have
$$(X \sqcup [\mathsf{emp}])(s, h', \eta) = X(s, h', \eta)$$
and thus we need to prove that
$$(Y_1 \dotplus Y_2)(s, h, \eta) \leq (X(s, h', \eta) \cdot (Y_1 \dotplus Y_2)(s, h, \eta)) \mathbin{\dot{/}} X(s, h', \eta).$$
Using adjointness of the truncated division, we get
$$X(s, h', \eta) \cdot (Y_1 \dotplus Y_2)(s, h, \eta) \leq X(s, h', \eta) \cdot (Y_1 \dotplus Y_2)(s, h, \eta),$$
which holds by reflexivity.
2. If $[\mathsf{emp}]\,(s, h', \eta) = 1$, then we have
$$(X \sqcup [\mathsf{emp}])(s, h', \eta) = 1$$
and thus we need to prove that
$$(Y_1 \dotplus Y_2)(s, h, \eta) \leq (1 \cdot (Y_1 \dotplus Y_2)(s, h, \eta)) \mathbin{\dot{/}} X(s, h', \eta).$$
Simplification gives us
$$(Y_1 \dotplus Y_2)(s, h, \eta) \leq (Y_1 \dotplus Y_2)(s, h, \eta) \mathbin{\dot{/}} X(s, h', \eta),$$
and using adjointness of truncated division yields
$$X(s, h', \eta) \cdot (Y_1 \dotplus Y_2)(s, h, \eta) \leq (Y_1 \dotplus Y_2)(s, h, \eta).$$
Using 1 as a neutral element of multiplication, we obtain
$$X(s, h', \eta) \cdot (Y_1 \dotplus Y_2)(s, h, \eta) \leq 1 \cdot (Y_1 \dotplus Y_2)(s, h, \eta),$$
and using monotonicity of multiplication, we require to prove that
$$X(s, h', \eta) \leq 1 \quad \text{and} \quad (Y_1 \dotplus Y_2)(s, h, \eta) \leq (Y_1 \dotplus Y_2)(s, h, \eta),$$
both of which hold.

5.3. Proof Rules for Fuzzy Separation Logic

Whereas we skimmed over proof rules for qualitative and quantitative separation logic in Chapters 3 and 4, we will give more details about the fuzzy versions, as they play a more important role in this dissertation. To this end, we will begin with the classic proof rules for our separation operators. We have monotonicity for separating multiplication in both arguments. We have antitonicity in the first and monotonicity in the second argument for the fuzzy truncated magic wand. This allows eliminating these operators if they occur on both sides of the entailment and the sub-random variables are appropriate.

Theorem 5.3.1 (Monotonicity and Antitonicity of Separating Multiplication and Fuzzy Truncated Magic Wand) *Let* $X_1, X_2, Y_1, Y_2 \in \mathsf{RV}^1$ *be random variables in FSL. We have that:*

$$X_1 \models X_2 \land Y_1 \models Y_2 \Rightarrow X_1 \star Y_1 \models X_2 \star Y_2$$
$$X_2 \models X_1 \land Y_1 \models Y_2 \Rightarrow X_1 \mathbin{-\!\circledast} Y_1 \models X_2 \mathbin{-\!\circledast} Y_2$$

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

If the magic wand is on the right side of an entailment, adjointness allows us to transform it into a separating multiplication on the left side of the entailment. Modus Ponens allows us to eliminate magic wands on the left side, if an appropriate separating multiplication is available.

Theorem 5.3.2 (Adjointness and Modus Ponens) *Let* $X_1, X_2, X_3, X_4 \in \mathsf{RV}^1$ *be random variables in FSL. We have that:*

$$X_1 \star X_2 \models X_3 \quad \textit{iff} \quad X_1 \models X_2 \mathbin{-\!\circledast} X_3 \qquad \text{(Adjointness)}$$
$$X_1 \star (X_1 \mathbin{-\!\circledast} X_2) \models X_2 \qquad \text{(Modus Ponens)}$$

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

In Chapters 3 and 4 we only referred to the possibility of eliminating substitution and gave an example. In this section, we are more rigorous and give explicit proof rules eliminating them.

Lemma 5.3.3 (Substitution Proof Rules) *Let* $X, Y \in \mathsf{RV}^1$ *be random variables in FSL,* $e \in \mathsf{ValExpr}$ *a value expression and* $x \in \mathsf{Vars}$ *a variable. We have that:*

$$
\begin{aligned}
[\Phi]\,[x := e] &= [\Phi\,[x := e]] \\
\langle e' \rangle\,[x := e] &= \langle e'\,[x := e] \rangle \\
({\sim} X)\,[x := e] &= {\sim} X\,[x := e] \\
(X \sqcap Y)\,[x := e] &= X\,[x := e] \sqcap Y\,[x := e] \\
(X \sqcup Y)\,[x := e] &= X\,[x := e] \sqcup Y\,[x := e] \\
(X \dot{+} Y)\,[x := e] &= X\,[x := e] \dot{+} Y\,[x := e] \\
(X \cdot Y)\,[x := e] &= X\,[x := e] \cdot Y\,[x := e] \\
(\text{Ƨ}\, a.\, X)\,[x := e] &= \text{Ƨ}\, a.\, X\,[x := e] \\
(\text{I}\, a.\, X)\,[x := e] &= \text{I}\, a.\, X\,[x := e] \\
(X \star Y)\,[x := e] &= X\,[x := e] \star Y\,[x := e] \\
\left(\mathop{\bigstar}_{i=0}^{n} X\right)[x := e] &= \mathop{\bigstar}_{i=0}^{n} X\,[x := e] \\
(X \mathbin{-\!\circledast} Y)\,[x := e] &= X\,[x := e] \mathbin{-\!\circledast} Y\,[x := e]
\end{aligned}
$$

Proof. See [43] at `LeanFSL.SL.FuzzySubstSimp`. □

In this chapter, we will also take the time to explore the relationship of the atomic proposition [emp] with the separating operations. For the separating multiplication, [emp] acts as a neutral element on both sides, for the magic wand only on the left side. Moreover, we can distribute a multiplied [emp] over separating multiplication.

Lemma 5.3.4 (Empty Heap Predicate) *Let $X, Y \in \mathrm{RV}^1$ be random variables in FSL. We have that:*

$$[\mathrm{emp}] \star X = X$$
$$[\mathrm{emp}] \cdot (X \star Y) = ([\mathrm{emp}] \cdot X) \star ([\mathrm{emp}] \cdot Y)$$
$$[\mathrm{emp}] \mathbin{-\!\!\circledast} X = X$$

Proof. See [43] at `LeanFSL.SL.FuzzySubstSimp`. □

We remark here some differences to first order logic about the lattice theoretical operations in separation logic. For once, we have

$$[\mathrm{true}] \star X \neq [\mathrm{true}] \quad \text{and} \quad [\mathrm{true}] \mathbin{-\!\!\circledast} X \neq X$$

as [true] allows the heap to be arbitrarily expanded. Instead, we have

$$[\mathrm{true}] \star X \models [\mathrm{true}]\,,$$

as one expects from the top element of the lattice. On the other side, we have

$$[\mathrm{false}] \star X = [\mathrm{false}] \quad \text{and} \quad [\mathrm{false}] \mathbin{-\!\!\circledast} X = [\mathrm{true}]\,.$$

Example 5.3.1 We want to prove the entailment

$$([x = y] \sqcap [x \mapsto y]) \star [\mathrm{true}] \models [x \mapsto x] \star ([x \mapsto 0] \mathbin{-\!\!\circledast} [y \neq 0]\,[y := x]).$$

First, we apply monotonicity and obtain the two new proof goals

$$[x = y] \sqcap [x \mapsto y] \models [x \mapsto x] \text{ and } [\mathrm{true}] \models [x \mapsto 0] \mathbin{-\!\!\circledast} [y \neq 0]\,[y := x]\,.$$

We first consider the entailment

$$[x = y] \sqcap [x \mapsto y] \models [x \mapsto x]\,.$$

For any (s, h) and η, we do a case distinction. If $s(x) \neq s(y)$, then the left side is zero, making the entailment vacuously true. Thus we assume the equality. Then we can replace y in $[x \mapsto y]$ by x, making the entailment hold by reflexivity.
Secondly we prove the entailment

$$[\mathrm{true}] \models [x \mapsto 0] \mathbin{-\!\!\circledast} [y \neq 0]\,[y := x]\,.$$

We use adjointness to transform it into the entailment

$$[\mathrm{true}] \star [x \mapsto 0] \models [y \neq 0]\,[y := x]\,.$$

Next, we eliminate substitution and obtain

$$[\mathsf{true}] \star [x \mapsto 0] \models [x \neq 0]\,.$$

Finally, let us consider any (s, h) and η and do a case distinction. If $s(x) = 0$, then the left side is zero as $[x \mapsto 0]$ requires $s(x)$ to be positive. But when the left side is zero, the entailment vacuously holds. Thus we assume that $s(x) \neq 0$. But then the right side is one, making the entailment hold vacuously again.

Another property that we also discussed in quantitative separation logic is conservativity. That is, the possibility to transform propositions in qualitative separation logic into random variables in fuzzy separation logic.

Theorem 5.3.5 (Conservativity of Fuzzy Separation Logic) *Let $\Phi, \Psi \in \mathsf{SLProp}$ be propositions in SL and $\Theta \colon \mathbb{N} \to \mathsf{SLProp}$ a map from natural numbers to propositions in SL. The following statements hold:*

We will in Chapter 8 see a different approach to translate a subset of FSL random variables into SL propositions as well.

$$
\begin{aligned}
\sim [\Phi] &= [\neg\Phi] \\
[\Phi] \sqcap [\Psi] &= [\Phi \wedge \Psi] \\
[\Phi] \sqcup [\Psi] &= [\Phi \vee \Psi] \\
[\Phi] \dotplus [\Psi] &= [\Phi \vee \Psi] \\
[\Phi] \cdot [\Psi] &= [\Phi \wedge \Psi] \\
\mathsf{S}a.\ [\Phi] &= [\exists a.\, \Phi] \\
\mathsf{I}\,a.\ [\Phi] &= [\forall a.\, \Phi] \\
[\Phi] \star [\Psi] &= [\Phi * \Psi] \\
[\Phi] \mathbin{-\!\!\circledast} [\Psi] &= [\Phi \mathbin{-\!\!*} \Psi] \\
\bigstar_{i=0}^{n} [\Theta] &= \left[\mathop{\ast}_{i=0}^{n} \Theta \right]
\end{aligned}
$$

Proof. See [43] at `LeanFSL.SL.Conservativity`. □

We depict the conservativity here slightly different to Theorem 4.3.2. The reason is that in quantitative separation logic, we cannot relate addition to logical disjunction using an equation. We instead had to say that

$$([\Phi] + [\Psi])(s, h, \eta) \geq 1 \iff (s, h) \models_\eta \Phi \vee \Psi.$$

In the fuzzy variant, we have that $a \geq 1 \iff a = 1$ and thus we can make a direct connection using equalities. Further using that

$$[\Phi]\,(s, h, \eta) = 1 \iff (s, h) \models_\eta \Phi,$$

we obtain the results from Theorem 4.3.2. Theorem 5.3.5, however, allows us to directly transform random variables in-place instead of replacing an entailment, thus we prefer this version.

Example 5.3.2 We want to prove the entailment

$$\left(\bigstar_{i=0}^{n} \lambda i.\ [x + i \mapsto i]\right) \star [0 \leq k < n] \models [x + k \mapsto k] \star [\text{true}]\,.$$

There are multiple ways of approaching this entailment. Our solution here is to use conservativity as SL is often easier to reason about. Translating the random variable into SL yields

$$\left(\mathop{\ast}_{i=0}^{n} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n \models x + k \mapsto k \ast \text{true}.$$

Now, we prove the claim using induction on n.
For $n = 0$, We simplify the proof obligation to

$$[\text{emp}] \ast 0 \leq k < 0 \models x + k \mapsto k \ast \text{true},$$

where $0 \leq k < 0$ is unsatisfiable, thus the entailment vacuously holds.
For $n > 0$, we unfold the separating conjunction once to have

$$\left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast x + n - 1 \mapsto n - 1 \ast 0 \leq k < n \models x + k \mapsto k \ast \text{true}.$$

We now do a case distinction. Either we have $k = n - 1$ or $k < n - 1$, as we definitely have $k < n$ as otherwise the left side would not be satisfied, making the entailment hold vacuously.

$k = n - 1$: Then we can rearrange the entailment as

$$x + n - 1 \mapsto n - 1 \ast \left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n \models x + k \mapsto k \ast \text{true},$$

and using monotonicity of the separating conjunction, we have to prove

$$x + n - 1 \mapsto n - 1 \models x + k \mapsto k,$$

which holds as $k = n - 1$, and

$$\left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n \models \text{true},$$

which holds vacuously due to true on the right side.

$k < n - 1$: By rearranging the entailment, we obtain

$$\left(\left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n\right) \ast x + n - 1 \mapsto n - 1$$
$$\models (x + k \mapsto k \ast \text{true}) \ast \text{true}.$$

Again using monotonicity of separating conjunction, we are left with

$$\left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n \models x + k \mapsto k \ast \text{true},$$

which by $k < n - 1$ can be rewritten into

$$\left(\mathop{\ast}_{i=0}^{n-1} \lambda i.\, x + i \mapsto i\right) \ast 0 \leq k < n - 1 \models x + k \mapsto k \ast \text{true}.$$

This holds by the induction hypothesis. The other proof obligation

$$x + n - 1 \mapsto n - 1 \models \mathsf{true}$$

holds vacuously because of the true on the right side of the entailment.

Finally, we consider distributivity of the separating multiplication and the magic wand with maxima, minima, addition, multiplication and quantifiers. That is, we are investigating rules with some operation op of the form

$$X \star (Y \,\mathrm{op}\, Z) = (X \star Y) \,\mathrm{op}\, (X \star Z)$$

for distributivity of separating multiplication,

$$X \star (Y \,\mathrm{op}\, Z) \models (X \star Y) \,\mathrm{op}\, (X \star Z)$$

for sub-distributivity of separating multiplication,

$$(X \mathbin{-\!\!\circledast} Y) \,\mathrm{op}\, (X \mathbin{-\!\!\circledast} Z) = X \mathbin{-\!\!\circledast} (Y \,\mathrm{op}\, Z)$$

for distributivity of the magic wand, and

$$(X \mathbin{-\!\!\circledast} Y) \,\mathrm{op}\, (X \mathbin{-\!\!\circledast} Z) \models X \mathbin{-\!\!\circledast} (Y \,\mathrm{op}\, Z)$$

for super-distributivity of the magic wand. For quantifiers these look analogous.

First, we consider distributivity for separating multiplication. Over maximum we have distributivity. This is because separating multiplication is defined over a supremum and a multiplication. Both distribute over maxima, thus obtaining our result. This does not hold for minimum, as supremum and minimum do not distribute. More concrete, the inequality

$$\begin{aligned} &\sup\{\min\{X(s_1),\ Y(s_2)\} \mid (s_1, s_2) \in A\} \\ \leq\ &\min\big\{\sup\{X(s_1) \mid (s_1, s_2) \in A\},\ \sup\{Y(s_2) \mid (s_1, s_2) \in A\}\big\}, \end{aligned}$$

holds, but not the other direction

$$\begin{aligned} &\sup\{\min\{X(s_1),\ Y(s_2)\} \mid (s_1, s_2) \in A\} \\ \not\geq\ &\min\big\{\sup\{X(s_1) \mid (s_1, s_2) \in A\},\ \sup\{Y(s_2) \mid (s_1, s_2) \in A\}\big\}. \end{aligned}$$

For addition, we have a similar problem and thus only have sub-distributivity. Multiplication is in general not distributive. However, when restricting the minor side of the separating multiplication to qualitative values, multiplication degrades to minima, therefore the same reasoning as for minimum holds. Lastly for quantifiers, we have distributivity for suprema and sub-distributivity for infima. For quantifiers, we require that the minor side does not have the bound variable as a free logical variable.

We define free variables and free logical variables analogously to free variables in Chapter 3. We say that a variable x is free if there is a stack for which we can change the value of x in some way in order to yield a different value of the random variable. Statically checking which variables occur in any expression for random variables may only give an over-approximation of the free variables. For free logical variables we similarly say that a variable a is free if there is a mapping from logical variables to values for which we can change the value of a

in some way in order to obtain a different value of the random variable.

Definition 5.3.1 (Free Variables) *Let $X \in \mathsf{RV}^1$ be a random variable in FSL. The set $Vars(X)$ of free variables of X is such that $x \in Vars(X)$ if and only if there exists $(s, h) \in \mathsf{States}$, $\eta \in \mathsf{LStacks}$ and value $v \in \mathbb{Q}$ with*

$$X(s, h, \eta) \quad \neq \quad X(s[x := v], h, \eta).$$

The set $LVars(X)$ of free logical variables of X is such that $a \in LVars(X)$ if and only if there exists $(s, h) \in \mathsf{States}$, $\eta \in \mathsf{LStacks}$ and value $v \in \mathbb{Q}$ with

$$X(s, h, \eta) \quad \neq \quad X(s, h, \eta[a := v]).$$

Theorem 5.3.6 (Sub- and Distributivity of Separating Multiplication) *Let $X, Y, Z \in \mathsf{RV}^1$ be random variables in FSL and $\Phi \in \mathsf{SLProp}$ a separation logic proposition. The following holds:*

$$X \star (Y \sqcup Z) = (X \star Y) \sqcup (X \star Y)$$
$$X \star (Y \sqcap Z) \models (X \star Y) \sqcap (X \star Y)$$
$$X \star (Y \dot{+} Z) \models (X \star Y) \dot{+} (X \star Y)$$
$$[\Phi] \star (Y \cdot Z) \models ([\Phi] \star Y) \cdot ([\Phi] \star Z)$$
$$a \notin LVars(X) \Rightarrow X \star ⅁ a.\, Y = ⅁ a.\, X \star Y$$
$$a \notin LVars(X) \Rightarrow X \star \mathrm{I}\, a.\, Y \models \mathrm{I}\, a.\, X \star Y$$

In our Lean formalization, we do not need to check for free logical variables due to internal renaming by Lean.

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

Example 5.3.3 Let us prove the validity of the entailment

$$\begin{aligned} &([x \mapsto 5] \star ([y \mapsto 0] \sqcap [y \mapsto 1])) \sqcup ([x \mapsto 5] \star ([y \mapsto 1] \sqcap [y \mapsto 0])) \\ \models\ & [x \mapsto 5] \star [y \mapsto 0] \sqcap [x \mapsto 5] \star [y \mapsto 1]\,. \end{aligned}$$

First, we can use distributivity with maximum to simplify and obtain

$$\begin{aligned} &[x \mapsto 5] \star ([y \mapsto 0] \sqcap [y \mapsto 1]) \sqcup ([y \mapsto 1] \sqcap [y \mapsto 0]) \\ \models\ & [x \mapsto 5] \star [y \mapsto 0] \sqcap [x \mapsto 5] \star [y \mapsto 1]\,. \end{aligned}$$

Using commutativity of minimum and that the maximum of two equal elements is the element itself, we further simplify the left side to

$$\begin{aligned} &[x \mapsto 5] \star ([y \mapsto 0] \sqcap [y \mapsto 1]) \\ \models\ & [x \mapsto 5] \star [y \mapsto 0] \sqcap [x \mapsto 5] \star [y \mapsto 1]\,. \end{aligned}$$

The remaining proof obligation holds by sub-distributivity with minimum.

We also introduce super-distributivity and distributivity of the magic wand. The dual idea and problem arises when dealing with the magic wand. As the magic wand is defined as an infimum, we have no problem distributing it over a minimum, but only have super-distributivity for the maximum. We also only have super-distributivity for addition. For suprema and infima, we have a similar

situation as for maxima and minima. The magic wand admits super-distributivity for suprema and distributivity for infima.

Theorem 5.3.7 (Super- and Distributivity of Separating Multiplication) *Let $X, Y, Z \in \mathrm{RV}^1$ be random variables in FSL. The following holds:*

$$(X \multimap\!\!\circledast\, Y) \sqcup (X \multimap\!\!\circledast\, Z) \vDash X \multimap\!\!\circledast\, (Y \sqcup Z)$$
$$(X \multimap\!\!\circledast\, Y) \sqcap (X \multimap\!\!\circledast\, Z) = X \multimap\!\!\circledast\, (Y \sqcap Z)$$
$$(X \multimap\!\!\circledast\, Y) \dotplus (X \multimap\!\!\circledast\, Z) \vDash X \multimap\!\!\circledast\, (Y \dotplus Z)$$
$$a \notin LVars(X) \Rightarrow \mathrm{Ƨ} a.\, X \multimap\!\!\circledast\, Y \vDash X \multimap\!\!\circledast\, (\mathrm{Ƨ} a.\, Y)$$
$$a \notin LVars(X) \Rightarrow \mathrm{I} a.\, X \multimap\!\!\circledast\, Y = X \multimap\!\!\circledast\, (\mathrm{I} a.\, Y)$$

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

We can gain distributivity of separating multiplication with certain operations by requiring an additional condition called *preciseness*. This will help us later to gain access to a (weaker) linearity proof rule. Preciseness for fuzzy separation logic is similar to preciseness for qualitative separation logic. For a random variable to be precise, we require that *for every heap, there is at most one included or equal heap which makes the random variable non-zero.* Equivalently, for every heap, there exists one smaller or equal heap, such that all heaps which are different, yield zero for the random variable.

And indeed there is also a very similar analogous for Quantitative Separation Logic [31].

Definition 5.3.2 (Preciseness for Random Variables) *Let $X \in \mathrm{RV}^1$ be a random variable in FSL. X is* precise *if and only if for all stack heap pairs (s, h) and values of logical variables η we have*

$$\exists h' \subseteq h.\, \forall h'' \subseteq h.\, h' \neq h'' \Rightarrow X(s, h'', \eta) = 0.$$

Preciseness essentially eliminates all but one possibility to split up the heap in case of separating multiplication, if we do not want to have zero as result. This allows us to regain distributivity with truncated addition and with minima.

Theorem 5.3.8 (Distributivity of Separating Multiplication with Preciseness) *Let $X, Y, Z \in \mathrm{RV}^1$ be random variables in FSL and X precise. The following holds:*

$$X \star (Y \sqcap Z) = (X \star Y) \sqcap (X \star Y)$$
$$X \star (Y \dotplus Z) = (X \star Y) \dotplus (X \star Y)$$
$$a \notin LVars(X) \Rightarrow X \star \mathrm{I} a.\, Y = \mathrm{I} a.\, X \star Y$$

Proof. See [43] at `LeanFSL.SL.FuzzyProofrules`. □

A similar condition called strictly-exact also exists to allow distributivity laws for the magic wand. Since this condition does not play a role in this thesis, we instead refer to [31] for more details.

6. Concurrent Probabilistic Programs

In this chapter, we combine the insights from Chapters 3 to 5 to define operational semantics and axiomatic semantics for concurrent probabilistic programs. We use the idea of *resource invariants* from Chapter 3, the idea of using *random variables as the objects to reason about programs* as proposed in Chapter 4 and *the fuzzy logic to reason about random variables* presented in Chapter 5. The reason to use a fuzzy logic instead of a quantitative logic lies in the way non-termination is handled in this semantics. Since we will define a *liberal semantics*, non-terminating runs are assigned the top element. In the shown quantitative logic, the top element is infinity and thus the weakest expectation of a program on an initial state that has non-terminating runs — even if the non-terminating runs have probability mass zero — is infinite. To avoid this behavior, we use fuzzy logics that still enable us to reason about the probability of a certain end result for a given initial state.

We adopt the example program from Chapter 3 and extend it with a probabilistic behavior by using the *probabilistic branching operation* we introduced in Chapter 4:

```
        <r> := −1;
pif (0.5) {   ||
   <r> := 0   || x := <r>;
} else {      || while (x = −1) { x := <r> };
   <r> := 1   ||
}             ||
```

This time the program can realize the postcondition $x = 0$ only with probability 0.5 (or not terminate at all). We thus expect to be able to prove the specification that the probability that on termination $x = 0$ plus the probability to not terminate is lower bounded by 0.5. Indeed, if we restrict ourselves to fair schedulers, the program actually terminates and the probability that on termination $x = 0$ is indeed exactly 0.5 given that r is allocated.

We use resource invariants to capture the behavior of the environment during concurrent execution. Threads can rely on the environment remaining invariant, but whenever they change the environment, they need to guarantee that the environment still satisfies the invariant. Both the invariant and the environment are concepts that a programmer maybe have during programming, but usually does not encode syntactically in the program. Instead, when we prove the validity of a statement on that program, we use the concepts of invariance of the environment to help us with the proof. In this example, r is the environment and our invariant is that r points either to -1 or 0. That seems curious since a careful examination of the program reveals that the program also manipulates the value at r to 1 (with probability 0.5). So why does our resource invariant not include this possibility? Remember that we want to prove that $x = 0$, finally. However, when $r \mapsto 1$, then we also have that $x \neq 0$. In a way, we are not interested in this kind of behavior, and thus we exclude it from the invariant. In the presence of probabilistic (or non-deterministic) behavior, we thus only

Environment here refers to the memory shared between threads.

include those behaviors in the invariant that will yield the end result we aim for.

Large parts of this chapter have been formalized in Lean. We formalized most of the proofs, excluding the adequacy theorem. The adequacy states the axiomatic semantics in terms of probability theory terms similar to Conjecture 4.1.1 and Theorem 4.1.2. Adequacy was out of scope for this dissertation as it requires to formalize larger parts of infinite sums, as well as theory for Markov decision processes. We formalized the examples, although the current framework does not offer a great support for entailment checking. However, we discuss some work on entailment checking for fuzzy separation logic in Chapter 8.

6.1. Concurrent Probabilistic Programs

We combine the syntax from concurrent programs as defined in Chapter 3 and the syntax from probabilistic programs as defined in Chapter 4 to obtain the syntax for concurrent probabilistic programs. Such programs feature probabilistic behavior and non-deterministic behavior due to the allocation of memory and concurrent interleaving.

Definition 6.1.1 (Concurrent Probabilistic Programming Language) *We define the concurrent probabilistic programming language* cpPL *as the language generated by this grammar:*

$\mathscr{C}$	$::=$	$\texttt{skip}$	*empty program*
	$\mid$	$x \texttt{ := } e$	*assignment*
	$\mid$	$\texttt{<}e\texttt{>} \texttt{ := } e'$	*heap mutation*
	$\mid$	$x \texttt{ := } \texttt{<}e\texttt{>}$	*heap lookup*
	$\mid$	$x \texttt{ := } \mathrm{CAS}(e_1, e_2, e_3)$	*compare and set*
	$\mid$	$x \texttt{ := } \texttt{new}(e)$	*allocation*
	$\mid$	$\texttt{free}(e, e')$	*deallocation*
	$\mid$	$\texttt{if}\ (e_b)\ \{\mathscr{C}\}\ \texttt{else}\ \{\mathscr{C}\}$	*conditional branching*
	$\mid$	$\texttt{pif}\ \left(e_p\right)\ \{\mathscr{C}\}\ \texttt{else}\ \{\mathscr{C}\}$	*probabilistic branching*
	$\mid$	$\texttt{while}(e_b)\{\mathscr{C}\}$	*looping*
	$\mid$	$\mathscr{C}\,;\mathscr{C}$	*program composition*
	$\mid$	$\mathscr{C} \parallel \mathscr{C}$	*concurrent execution*

where $e, e', e_1, e_2, e_3 \in \mathsf{ValExpr}$ *are value expressions,* $e_b \in \mathsf{BoolExpr}$ *are Boolean expressions and* $e_p \in \mathsf{ProbExpr}$ *are probabilistic expressions. We use* $\downarrow$ *to depict terminated programs and* $\lightning$ *to depict erroneous programs.*

We define the semantics operationally as Markov decision processes. A program is transformed into a (possibly infinite but countable) Markov decision process. In case of unbounded loops, the Markov decision process is infinite. Concurrency is modelled as non-determinism in the Markov decision process, where we use a recursive structure to model arbitrary nesting of concurrency. The concurrency operator admits two threads, a left thread and a right thread.

The operational semantics largely match the semantics from Chapter 4. For completeness, we include all rules again. We thus have the same semantics

for statements in cpPL as shown in Figure 6.1. As in Chapter 4, we denote the probability transition function with

$$\sigma \xrightarrow[\text{a}]{p} \sigma' \Rightarrow \mathbf{P}(\sigma, a, \sigma') = p$$

and whenever a rule is missing, we have that $\mathbf{P}(\sigma, a, \sigma') = 0$. Actions are enabled if they occur in a rule. We formally define the enabled actions in Definition 6.1.2. We differ from the semantics of Chapter 4 by having concurrent executions. In Figure 6.2 we depict all rules for flow related operations, including concurrent execution.

Definition 6.1.2 (Enabled Actions) *Let C be a* cpPL *program, terminated program or an erroneous program, and (s, h) a stack heap pair. We define the set of* enabled actions *$Act(C, (s, h))$ in the MDP corresponding to C with transition probabilities defined in Figures 6.1 and 6.2 as*

$$\begin{aligned}
&Act(\downarrow, (s,h)) = \emptyset,\\
&Act(\lightning, (s,h)) = \emptyset,\\
&Act(\texttt{skip}, (s,h)) = \{Det\},\\
&Act(x \texttt{ := } e, (s,h)) = \{Det\},\\
&Act(\texttt{<}e\texttt{>} \texttt{ := } e', (s,h)) = \{Det\},\\
&Act(x \texttt{ := } \texttt{<}e\texttt{>}, (s,h)) = \{Det\},\\
&Act(x \texttt{ := } \texttt{CAS}(e_1, e_2, e_3), (s,h)) = \{Det\},\\
&Act(x \texttt{ := } \texttt{new}(e), (s,h)) = \{Alloc\ \ell \mid e(s) \in \mathbb{N}\\
&\qquad\qquad \wedge h(\ell + 0) = undef \wedge \ldots \wedge h(\ell + e(s) - 1) = undef\}\\
&\qquad\qquad \cup \{Det \mid e(s) \notin \mathbb{N}\},\\
&Act(\texttt{free}(e, e'), (s,h)) = \{Det\},\\
&Act(\texttt{if } (e_b)\ \{C_1\} \texttt{ else } \{C_2\}, (s,h)) = \{Det\},\\
&Act(\texttt{pif } \left(e_p\right)\ \{C_1\} \texttt{ else } \{C_2\}, (s,h)) = \{Det\},\\
&Act(\texttt{while}\,(e_b)\,\{C\}, (s,h)) = \{Det\},\\
&Act(C_1 ; C_2, (s,h)) = \{Det \mid C_1 = \downarrow \vee C_1 = \lightning\} \cup Act(C_1, (s,h)),\\
&Act(C_1 \parallel C_2, (s,h)) = \{Det \mid C_1 = \downarrow \wedge C_2 = \downarrow \vee C_1 = \lightning \vee C_2 = \lightning\}\\
&\qquad\qquad \cup \left\{Left\ a \;\middle|\; a \in Act(C_1, (s,h))\right\}\\
&\qquad\qquad \cup \left\{Right\ a \;\middle|\; a \in Act(C_2, (s,h))\right\}.
\end{aligned}$$

We depict here the enabled actions explicitly, contrary to Chapter 4. However, the set of enabled actions for non-concurrent probabilistic programs are the same.

Similar to Chapter 4, we use the action Det both for deterministic transitions and for probabilistic transitions. The action Alloc ℓ depicts the action for allocating memory starting from location ℓ, of which there are countably infinite. Concurrent execution either allows to execute the left or the right thread, given that its not yet terminated or that it will not abort. We use the recursive structure that allows either Left a or Right a for an action a. The left action yields an execution of the left thread and the right action yields an execution of the right thread. In case either thread aborts taking this action, the concurrent execution aborts. For technical reasons, we also include the case that we execute an abort statement. In

$$\frac{}{\texttt{skip}, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s,h)} \text{SKIP} \qquad \frac{}{x := e, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s[x := e(s)], h)} \text{ASSIGN}$$

$$\frac{e(s) \in \mathsf{dom}(h)}{x := \texttt{<}e\texttt{>}, (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s[x := h(e(s))], h)} \text{LOOKUP} \qquad \frac{e(s) \notin \mathsf{dom}(h)}{x := \texttt{<}e\texttt{>}, (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)} \text{LOOKUP-ABT}$$

$$\frac{e(s) \in \mathsf{dom}(h)}{\texttt{<}e\texttt{>} := e', (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s, h[e(s) := e'(s)])} \text{MUT} \qquad \frac{e(s) \notin \mathsf{dom}(h)}{\texttt{<}e\texttt{>} := e', (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)} \text{MUT-ABT}$$

$$\frac{e_1(s) \in \mathsf{dom}(h) \qquad h(e_1) \neq e_2(s)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s[x := 0], h)} \text{CAS-FALSE}$$

$$\frac{e_1(s) \in \mathsf{dom}(h) \qquad h(e_1) = e_2(s)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s[x := 1], h[e_1(s) := e_3(s)])} \text{CAS-TRUE}$$

$$\frac{e_1(s) \notin \mathsf{dom}(h)}{x := \texttt{CAS}(e_1, e_2, e_3), (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)} \text{CAS-ABT}$$

$$\frac{e(s) = n \in \mathbb{N} \qquad \ell, \dots, \ell + n - 1 \in \mathbb{N}_{>0} \setminus \mathsf{dom}(h) \qquad h' = h \cup \{\ell \mapsto 0, \dots, \ell + n - 1 \mapsto 0\}}{x := \mathsf{new}(e), (s,h) \xrightarrow[\text{Alloc } \ell]{1} \downarrow, (s[x := \ell], h')} \text{ALLOC}$$

$$\frac{e(s) \notin \mathbb{N}}{x := \mathsf{new}(e), (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)} \text{ALLOC-ABT}$$

$$\frac{e(s) + 0, \dots e(s) + e'(s) - 1 \in \mathsf{dom}(h) \qquad h' = h[e(s) + 0 := \mathsf{undef}] \dots [e(s) + e'(s) - 1 := \mathsf{undef}]}{\texttt{free}(e, e'), (s,h) \xrightarrow[\text{Det}]{1} \downarrow, (s, h')} \text{FREE}$$

$$\frac{\exists i \in \{0, \dots, e'(s) - 1\}.\, e(s) + i \notin \mathsf{dom}(h)}{\texttt{free}(e, e'), (s,h) \xrightarrow[\text{Det}]{1} \lightning, (s,h)} \text{FREE-ABT}$$

Figure 6.1.: Operational semantics for all statements in cpPL. Heap manipulating statements may abort due to the accessed location not being allocated. Compare and set changes the value at a certain location only if the value is as expected.

this case, the concurrent execution also aborts. If both threads are terminated, we terminate the concurrent execution. The difference to the operational semantics from Chapter 3 is that we have to introduce explicit actions for executing the left or the right thread due to our choice of a Markov decision process as our semantic model.

The semantics in Figures 6.1 and 6.2 are indeed a Markov decision process, that is for every state and action, the probabilities add up to 1.

Notice that we use the non-truncated sum here.

Theorem 6.1.1 *The semantics from Figures 6.1 and 6.2 yield Markov decision processes for every initial state, that is, for all* cpPL *programs* C *and* $(s,h) \in \mathsf{States}$ *and for every enabled action* $a \in Act(C, (s,h))$ *we have:*

$$\sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p = 1.$$

Proof. See [43] at `LeanFSL.Program.SumOne`. □

$$\dfrac{C_1' \neq \lightning \qquad C_1,(s,h) \xrightarrow[\text{a}]{p} C_1',(s',h')}{C_1\,;C_2,(s,h) \xrightarrow[\text{a}]{p} C_1'\,;C_2,(s',h')}\ \text{SEQ} \qquad \dfrac{}{\downarrow;C_2,(s,h) \xrightarrow[\text{Det}]{1} C_2,(s,h)}\ \text{SEQ-END}$$

$$\dfrac{C_1,(s,h) \xrightarrow[\text{a}]{p} \lightning,(s,h)}{C_1\,;C_2,(s,h) \xrightarrow[\text{a}]{p} \lightning,(s,h)}\ \text{SEQ-ABT} \qquad \dfrac{}{\lightning\,;C_2,(s,h) \xrightarrow[\text{Det}]{1} \lightning,(s,h)}\ \text{SEQ-ABT-2}$$

$$\dfrac{e_b(s) = \mathsf{true}}{\texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\},(s,h) \xrightarrow[\text{Det}]{1} C_1,(s,h)}\ \text{IF-T} \qquad \dfrac{e_b(s) = \mathsf{false}}{\texttt{if}\ (\,e_b\,)\ \{C_1\}\ \texttt{else}\ \{C_2\},(s,h) \xrightarrow[\text{Det}]{1} C_2,(s,h)}\ \text{IF-F}$$

$$\dfrac{e_p(s) = p}{\texttt{pif}\ \left(e_p\right)\ \{C_1\}\ \texttt{else}\ \{C_2\},(s,h) \xrightarrow[\text{Det}]{p} C_1,(s,h)}\ \text{PIF-T}$$

$$\dfrac{e_p(s) = p}{\texttt{pif}\ \left(e_p\right)\ \{C_1\}\ \texttt{else}\ \{C_2\},(s,h) \xrightarrow[\text{Det}]{1-p} C_2,(s,h)}\ \text{PIF-F}$$

$$\dfrac{e_b(s) = \mathsf{true}}{\texttt{while}(\,e_b\,)\{C_1\},(s,h) \xrightarrow[\text{Det}]{1} C_1\,;\texttt{while}(\,e_b\,)\{C_1\},(s,h)}\ \text{WHILE-T}$$

$$\dfrac{e_b(s) = \mathsf{false}}{\texttt{while}(\,e_b\,)\{C_1\},(s,h) \xrightarrow[\text{Det}]{1} \downarrow,(s,h)}\ \text{WHILE-F}$$

$$\dfrac{C_1' \neq \lightning \qquad C_1,(s,h) \xrightarrow[a]{p} C_1',(s',h')}{C_1 \parallel C_2\,,(s,h) \xrightarrow[\text{Left}\ a]{p} C_1' \parallel C_2\,,(s',h')}\ \text{CON-L} \qquad \dfrac{C_2' \neq \lightning \qquad C_2,(s,h) \xrightarrow[a]{p} C_2',(s',h')}{C_1 \parallel C_2\,,(s,h) \xrightarrow[\text{Right}\ a]{p} C_1 \parallel C_2'\,,(s',h')}\ \text{CON-R}$$

$$\dfrac{C_1,(s,h) \xrightarrow[a]{p} \lightning,(s,h)}{C_1 \parallel C_2\,,(s,h) \xrightarrow[\text{Left}\ a]{p} \lightning,(s,h)}\ \text{CON-L-ABT} \qquad \dfrac{C_2,(s,h) \xrightarrow[a]{p} \lightning,(s,h)}{C_1 \parallel C_2\,,(s,h) \xrightarrow[\text{Right}\ a]{p} \lightning,(s,h)}\ \text{CON-R-ABT}$$

$$\dfrac{}{\lightning \parallel C_2\,,(s,h) \xrightarrow[\text{Det}]{1} \lightning,(s,h)}\ \text{CON-L-ABT-2} \qquad \dfrac{}{C_1 \parallel \lightning\,,(s,h) \xrightarrow[\text{Det}]{1} \lightning,(s,h)}\ \text{CON-R-ABT-2}$$

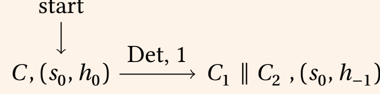
$$\dfrac{}{\downarrow \parallel\ \downarrow,(s,h) \xrightarrow[\text{Det}]{1} \downarrow,(s,h)}\ \text{CON-END}$$

Figure 6.2.: Operational Semantics for all flow related program constructs in cpPL. Sequencing is defined inductively, branching and looping are defined transitionally. Probabilistic branching executes either path, but only with a certain probability and is the only statement having probabilistic behavior. In any case, we immediately transition to an abort statement when an underlying command fails. Concurrency can either execute the left thread with an action Left a and the right thread with an action Right a, where a is the action of the thread. Concurrency aborts when either thread aborts or when an abort statement is to be executed. We terminate concurrent execution when both threads terminated.

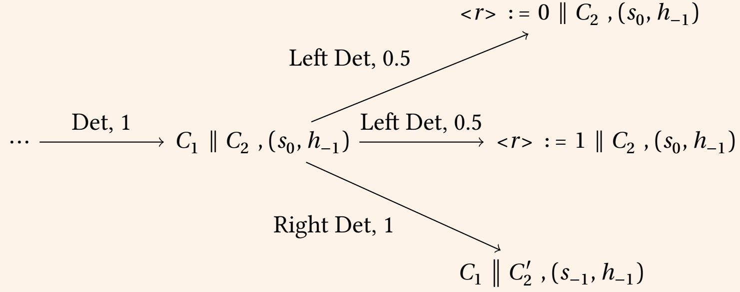
Example 6.1.1 We reconsider the example program C from the introduction:

```
          <r> := -1;
pif (0.5) {    ||
   <r> := 0    ||
} else {       ||  x := <r>;
   <r> := 1    ||  while(x = -1){x := <r>}
}              ||
```

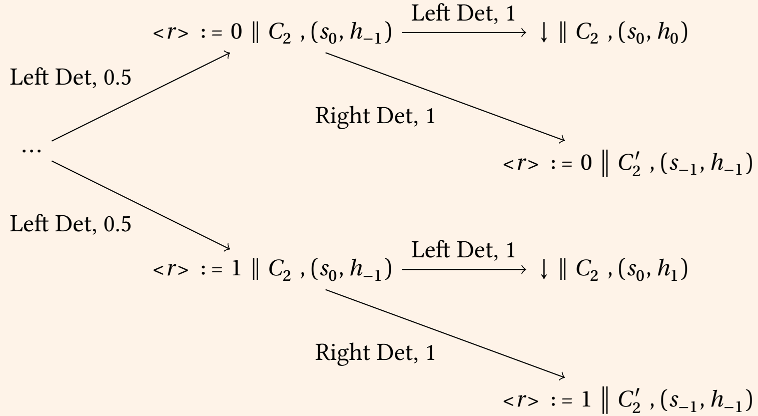
We construct the operational semantics of this program. Similarly to Example 3.2.2, we leave out the SEQ-END steps for brevity. Remark that the state space for a fixed initial state is finite. We thus fix the initial state (s, h) with $\forall x \neq r.\, s(x) = 0$, $s(r) = 1$ and $h = \{1 \mapsto 0\}$. We use the short-hand notation $s_a(y) =$ if $(y = x)$ then a else $s(y)$ and $h_a = \{1 \mapsto a\}$. We further notate the program with variables as $<r> := -1;\ (C_1 \parallel C_2)$. We first have the execution of $<r> := -1$, thus obtaining this part of the MDP:



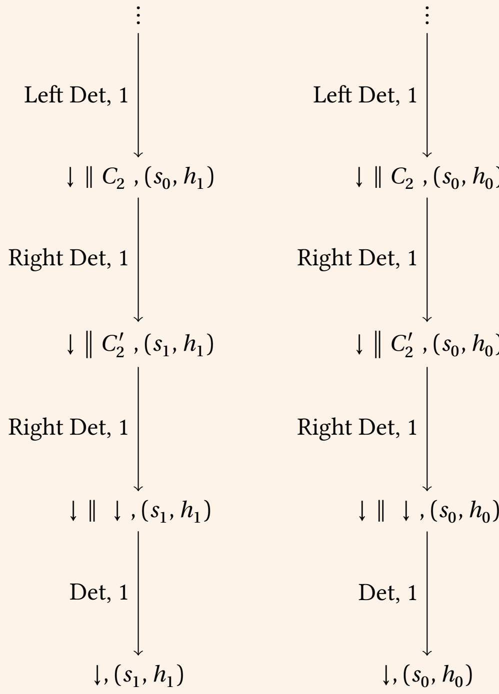
Next we have the choice to either execute the left thread or the right thread. Executing the left thread will resolve the probabilistic choice. Executing the right thread will change the stack. We thus obtain



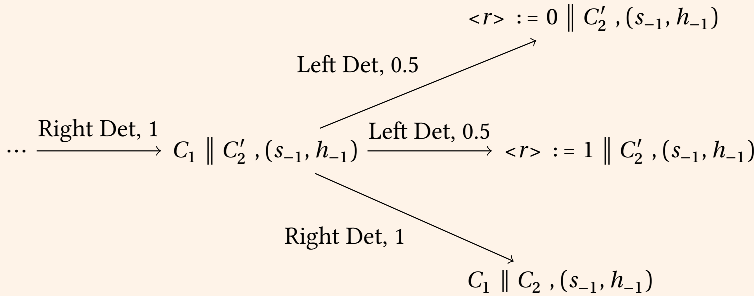
where C_2' is the loop of the right thread $\texttt{while}(x = -1)\{x := <r>\}$.
We first continue with the states obtained from executing the left thread as these will be easy. Executing the manipulation changes the heap once again and leaves the left threads as terminated:



For both left executing paths we now have the following straightforward paths and do not enter the loop:

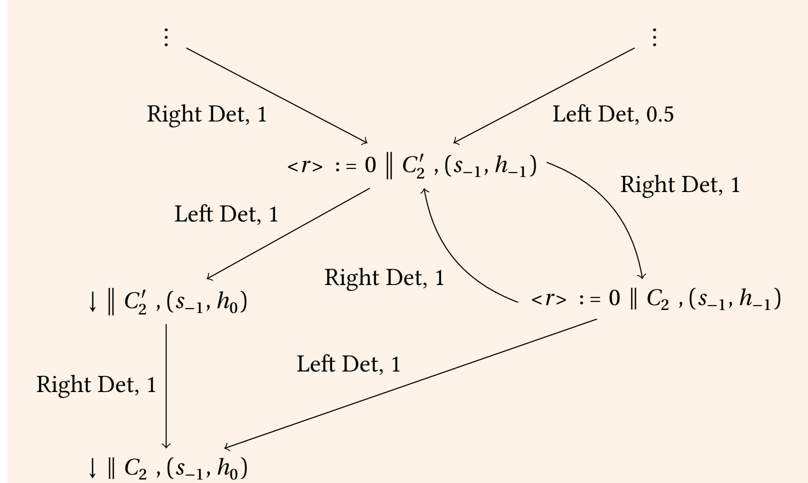


Now we further follow the other reachable states from state $C_1 \parallel C_2'$, (s_{-1}, h_{-1}). We can again either execute the left thread or the right thread. We notice however, that states reappear:



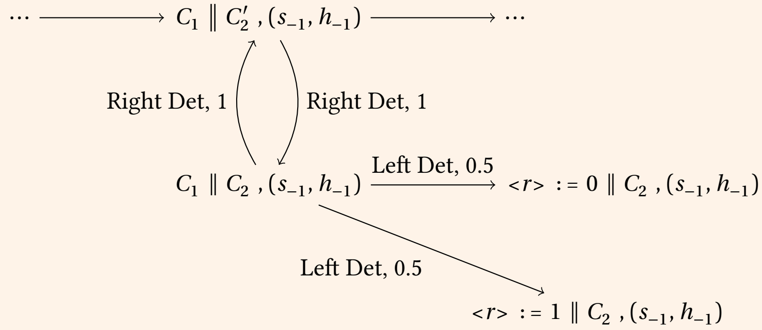
Since the loop condition is still valid in the state (s_{-1}, h_{-1}), executing the right branch transitions the state $C_1 \parallel C_2'$, (s_{-1}, h_{-1}) to the state $C_1 \parallel \left(x := \texttt{<}r\texttt{>} ; C_2'\right)$, (s_{-1}, h_{-1}). Since $x := \texttt{<}r\texttt{>} ; C_2'$ is the same as C_2, we denote this state instead as $C_1 \parallel C_2$, (s_{-1}, h_{-1}). Taking the transition for the left thread yields a state that already appeared. We continue with the execution from this state and obtain:

We obtained the current state twice, as such we depict two sources with dots.



We observe that the loop may not terminate in case we always pick the action Right Det. Indeed, to guarantee termination, we require fairness of the schedule here. We already investigated the transitions that are possible from state $\downarrow \parallel C_2$, (s_0, h_0), thus we do not repeat them. The transitions from state $<r> := 1 \parallel C_2'$, (s_{-1}, h_{-1}) are analogous. Thus we are only left with the state $C_1 \parallel C_2$, (s_{-1}, h_{-1}). For every possible transition from this state, we obtain a state that we already considered:

We depict the state $C_1 \parallel C_2'$, (s_{-1}, h_{-1}) here special to highlight, that we obtained the state $C_1 \parallel C_2$, (s_{-1}, h_{-1}) from there.



The two states $<r> := 0 \parallel C_2$, (s_{-1}, h_{-1}) and $<r> := 1 \parallel C_2$, (s_{-1}, h_{-1}) occurred in the previous representation. The state $C_1 \parallel C_2'$, (s_{-1}, h_{-1}) is where we come from. Since it forms a cycle with our investigated state, we depict it here again.

The Markov decision process is now fully constructed and careful observation reveals that the program does not terminate (due to non-fair schedule) or the program terminates with probability 0.5 in a state with $s_0(x) = 0$ and with probability 0.5 in a state with $s_1(x) = 1$.

6.2. Weakest Resource-Safe Liberal Expectation

In order to reason about the possibly infinite MDP semantics introduced in the previous section, we use axiomatic semantics similar to the ones used in

Chapters 3 and 4. We use weakest expectation semantics to justify the soundness of the axiomatic semantics. However, similar to Chapter 3, we require *liberal semantics* to enable a rule for concurrency and *resource invariants* to reason about shared memory.

Resource invariants are used to describe an *invariant behavior of shared memory*. Their use can be lifted to the probabilistic setting. However, for weakest expectations, we now also allow the use of *fuzzy invariants*. As such, an invariant can "save" and "transmit" collected probability mass between threads. We use a technique to introduce resource invariants similar to the one used in Definition 3.3.1: Before every execution step we add a heap "satisfying" the resource invariant to the state, execute the step, and then remove a heap "satisfying" the resource invariant again. This way, we always need to guarantee that the shared memory does indeed "satisfy" the resource invariant — otherwise there is nothing to remove. We put satisfaction into quotation marks here as fuzzy separation logic does not offer a satisfaction relation. A more precise description is that we add heaps with a weight and remove heaps with a weight. Unfortunately this is not as illustrative.

Remark that [33] only allows qualitative invariants.

Executing one step is performed similar to the way we defined executing one step in Definition 4.4.1: For one action, we take the weighted sum $\sum_\sigma p_\sigma \cdot X(\sigma)$ for every possible transition to state σ over the random variable $X(\sigma)$ to evaluate weighted by the probability of that transition. We exclude aborting transitions as we always weight erroneous executions with 0 and since some proofs are easier when we exclude aborting transitions. We then take the infimum of these sums over all enabled actions since we reason about lower bounds for any action.

In this sense, we combine the definition of weakest expectation (Definition 4.4.1 on page 67) with the definition of weakest resource-safe liberal precondition (Definition 3.3.1 on page 36) to obtain semantics for concurrent probabilistic programs. We will later in this section see that this also transfers to the axiomatic semantics and that we obtain axiomatic semantics reflecting a combination of those for concurrent programs and those for probabilistic programs.

Definition 6.2.1 (Weakest Resource-Safe Liberal Expectation) *Let C be a* cpPL *program, $X \in \mathsf{RV}^1$ a random variable in fuzzy separation logic and $R \in \mathsf{RV}^1$ a resource invariant. The* weakest resource-safe liberal expectation **wrle** *is the greatest solution of the equivalence*

$$\mathsf{wrle}[\![C]\!]\,(X \mid R) = \begin{cases} 0 & \textit{if } C = \lightning \\ X & \textit{if } C = \downarrow \\ R \mathbin{-\!\!\circledast} \mathsf{step}[\![C]\!]\left(\lambda C'.\, \mathsf{wrle}[\![C']\!]\,(X \mid R) \star R\right) & \textit{else} \end{cases}$$

where $\mathsf{step}[\![C]\!]\,(f)$ *for a* cpPL *program C, $(s,h) \in \mathsf{States}$, $\eta \in \mathsf{LStacks}$ and a function f mapping programs to* RV^1 *is defined as*

$$\begin{aligned} &\mathsf{step}[\![C]\!]\,(f)\,(s,h,\eta) \\ = &\inf_{a \in Act(C,(s,h))} \quad \dot{\sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h'),C' \neq \lightning}} p \cdot f(C')(s',h',\eta). \end{aligned}$$

We encourage comparing this definition with Definitions 3.3.1 and 4.4.1 to see both similarities and differences.

The weakest resource-safe liberal expectation is a liberal semantics, which means that it weights non-termination with the top value. One may wonder now if that

causes any problems in the presence of unfair schedulers. An unfair scheduler starves one thread in favor of a busy waiting thread, thus stopping any further progress. This yields spurious non-terminating behavior. Usually we assume that in a system allowing multiple threads, the scheduler never starves any thread. This is however not an actual problem. Consider an initial state (s, h) for which one schedule never terminates (e.g. an unfair schedule) and one schedule that terminates with a certain probability. Since we take the infimum over schedules and non-terminating schedules will yield the top value, the value of a terminating schedule will always be lower and **wrle** always prefers fair schedules over unfair schedules.

We can formalize the connection between the weakest resource-safe liberal expectation and the weakest resource-safe liberal precondition by means of conservativity. Indeed, the **wrle** yields exactly the same results as **wrlp** if applied to non-probabilistic programs and 1 interpreted as true and 0 interpreted as false.

Theorem 6.2.1 (Conservativity of **wrle**) *Let C be a* cPL *program,* $\Phi \in \mathsf{SLProp}$ *a postcondition and* $\xi \in \mathsf{SLProp}$ *a resource invariant in qualitative separation logic. We have that*

$$\left[\mathsf{wrlp}[\![C]\!]\,(\Phi \mid \xi)\right] \quad = \quad \mathsf{wrle}[\![C]\!]\,([\Phi] \mid [\xi])\,.$$

Proof. Follows from applying Theorem 2.3.3 and transfinite induction on the ordinals. Finally, using conservativity from Theorem 5.3.5, it only remains to prove that the step functions behave the same. Since no probability is present, the addition in the **wrle** step function degrades into $f(C')(s', h', \eta)$ if for all a, C', s', h' with $a \in \mathrm{Act}(C, (s, h))$ and $C, (s, h) \xrightarrow[a]{p} C', (s', h')$ we have $C' \neq \lightning$, and zero otherwise. Note that when $a \in \mathrm{Act}(C, (s, h))$ we have $C, (s, h) \to C', (s', h')$ for **wrlp** and vice versa. Thus we have the equality. □

We prove a similar adequacy to the one we hypothesize in Conjecture 4.1.1. If we take every terminating path, and sum the probability of that path multiplied with the random variable applied to final states with program $\downarrow$ of the path, we obtain the expected value regarding the random variable. If we now also add the probability of divergence, we obtain a notion of *liberal expected value*. We encode diverging runs here by not reaching the final states, which are states with program $\downarrow$ or $\lightning$. Runs resulting in $\lightning$ are thus always weighted with 0. The weakest resource-safe liberal expectation with resource invariant $[\mathsf{emp}]$ is indeed equal to the liberal expected value of one-bounded non-negative random variables.

Theorem 6.2.2 (Adequacy of Weakest Resource-Safe Liberal Expectation) *Let C be a* cpPL *program, $X \in \mathrm{RV}^1$ a random variable, $(s,h) \in \mathsf{States}$ a program state and $\eta \in \mathsf{LVars}$ assignments for logical variables. We denote with $\mathbb{P}_{\mathfrak{s}}^{C,(s,h)}$ the probability transition function for the induced Markov chain with scheduler $\mathfrak{s}$ from initial state $C,(s,h)$. We have the following equality between the weakest resource-safe liberal expectation and the liberal expected value*

$$\begin{aligned} &\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}])\,(s,h,\eta) \\ &= \inf_{\mathfrak{s} \in \mathbb{S}} \sum_{\downarrow,(s',h')} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\diamond \downarrow,(s',h')) \cdot X(s',h',\eta) \\ &\quad + 1 - \sum_{C,(s',h'),C \in \{\downarrow,\lightning\}} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\diamond C,(s',h')). \end{aligned}$$

That means that **wrle** with empty resource invariant and for the random variable X is equal to the value of $\inf_{\mathfrak{s}} \mathbb{E}(X) + \mathbb{P}(\neg\mathrm{term})$.

In order to prove this theorem, another theorem is helpful. We can transform the fixed point defined in Definition 6.2.1 into an infimum using Theorem 2.3.1. This brings us already one step closer to verifying the equality in Theorem 6.2.2. We could also use Theorem 2.3.3 to do this, however, unifying the resulting expressions seem difficult. To apply Theorem 2.3.1, we need to prove that **wrle** is co-ω-Scott-continues.

Theorem 6.2.3 *For a* cpPL *program C, the weakest resource-safe liberal expectation $\lambda X.\,\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}])$ is co-ω-Scott-continues.*

Proof. See [43] at `LeanFSL.CFSL.Cocontinous`. ☐

We now proceed to prove Theorem 6.2.2. This proof has not been formalized in Lean due to lack of formalization of the necessary theorems and definitons for the right side of the equation concerning MDP semantics, which were out of scope for this thesis.

Proof of Theorem 6.2.2. We will transform both sides of the equation into different forms that are easier to handle and which allow the use of natural induction to reason about.

First, we can transform the equation defining $\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}])$ using that the resource invariant is emp into the equation

$$\begin{aligned} &\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}])\,(s,h,\eta) \\ &= \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \inf\limits_{a \in \mathrm{Act}(C,(s,h))} \dot{\sum\limits_{C,(s,h) \xrightarrow[a]{p} C',(s',h'),C' \neq \lightning}} p \cdot \mathsf{wrle}[\![C']\!]\,(X \mid [\mathsf{emp}])\,(s',h',\eta) & \text{else} \end{cases} \end{aligned}$$

We can extract the characteristic function from this equation defining a fixed

point and obtain that for the characteristic function

$$\mathsf{wrlestep}[\![C]\!]\,(X \mid [\mathsf{emp}])\,(Y)(s,h,\eta) = \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \inf\limits_{a \in \mathrm{Act}(C,(s,h))} \dot{\sum\limits_{C,(s,h) \xrightarrow[a]{p} C',(s',h'),C' \neq \lightning}} p \cdot Y(C')(s',h',\eta) & \text{else,} \end{cases}$$

we have that

$$\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}]) = \mathsf{gfp}\ \mathsf{wrlestep}[\![C]\!]\,(X \mid [\mathsf{emp}])\,.$$

We can furthermore prove (see [43] at `LeanFSL.CFSL.Bellman`) that the characteristic function is equivalent to the one of the Bellman equation, thus obtaining the equality

$$\mathsf{wrlestep}[\![C]\!]\,(X \mid [\mathsf{emp}])\,(Y)(s,h,\eta) = \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \inf\limits_{a \in \mathrm{Act}(C,(s,h))} \dot{\sum\limits_{C,(s,h) \xrightarrow[a]{p} C',(s',h')}} p \cdot Y(C')(s',h',\eta) & \text{else.} \end{cases}$$

Moreover, since in each step we only have convex sums (of even only two elements), we also have that the sum behaves regular and thus the following equality also holds

$$\mathsf{wrlestep}[\![C]\!]\,(X \mid [\mathsf{emp}])\,(Y)(s,h,\eta) = \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \inf\limits_{a \in \mathrm{Act}(C,(s,h))} \sum\limits_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p \cdot Y(C')(s',h',\eta) & \text{else.} \end{cases}$$

Finally we can apply Theorem 2.3.1 because of Theorem 6.2.3 and obtain that the weakest resource-safe liberal expectation $\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}])$ with empty resource invariant is equivalent to the infimum of the natural fixed point iteration

$$\mathsf{wrle}[\![C]\!]\,(X \mid [\mathsf{emp}]) \quad = \quad \inf_{n \in \mathbb{N}} \mathsf{wlestep}^n[\![C]\!]\,(X)$$

where we define $\mathsf{wlestep}^n[\![C]\!]\,(X)$ as

$$\begin{aligned} &\mathsf{wlestep}^0[\![C]\!]\,(X)\,(s,h,\eta) = 1 \\ &\mathsf{wlestep}^{n+1}[\![C]\!]\,(X)\,(s,h,\eta) \\ &= \begin{cases} 0 & \text{if } C = \lightning \\ X & \text{if } C = \downarrow \\ \inf\limits_{a \in \mathrm{Act}(C,(s,h))} \sum\limits_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p \cdot \mathsf{wlestep}^n[\![C']\!]\,(X)\,(s',h',\eta) & \text{else} \end{cases} \end{aligned}$$

Secondly, we transform the right side of the equation in Theorem 6.2.2 using monotone convergence and the definition of reachability probabilities $\mathbb{P}_{\mathfrak{s}}^{C,(s,h)}$

with initial state $C, (s, h)$ into

$$\inf_{n \in \mathbb{N}} \inf_{\mathfrak{s} \in \mathbb{S}} \sum_{|\pi| \leq n, \pi_1 = C,(s,h), \pi_k = \downarrow,(s_k,h_k)} \left(\prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \right) \cdot X(s_k, h_k, \eta) + 1 - \sum_{|\pi| \leq n, \pi_1 = C,(s,h), \pi_k = C_k,(s_k,h_k), C_k \in \{\downarrow, \lightning\}} \prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}),$$

where π is a path in the MDP, $|\pi|$ the length of path, π_i the ith element of the path and $\overline{\pi_i}$ are the first i elements of the path. If the value for the above expression is equal to $\text{wlestep}^n[\![C]\!](X)(s, h, \eta)$ for every n, the infimum of these is also equal. Thus we proceed by using natural induction to prove the equivalence.

We generalize over the programs C and states (s, h) for this induction.

For $n = 0$ we have that $\text{wlestep}^0[\![C]\!](X)(s, h, \eta) = 1$ and since $|\pi| = 0$ means that π is empty, we have that the other expression is

$$\begin{aligned}
&\inf_{\mathfrak{s} \in \mathbb{S}} \sum_{|\pi| \leq 0, \pi_1 = C,(s,h), \pi_k = \downarrow,(s_k,h_k)} \left(\prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \right) \cdot X(s_k, h_k, \eta) \\
&\quad + 1 - \sum_{|\pi| \leq 0, \pi_1 = C,(s,h), \pi_k = C_k,(s_k,h_k), C_k \in \{\downarrow, \lightning\}} \prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \\
= &\inf_{\mathfrak{s} \in \mathbb{S}} 0 + 1 - 0 = 1.
\end{aligned}$$

We obtain as induction hypothesis that for a fixed but arbitrary n and for all C and (s, h) we have

$$\begin{aligned}
&\text{wlestep}^n[\![C]\!](X)(s, h, \eta) \\
= &\inf_{\mathfrak{s} \in \mathbb{S}} \sum_{|\pi| \leq n, \pi_1 = C,(s,h), \pi_k = \downarrow,(s_k,h_k)} \left(\prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \right) \cdot X(s_k, h_k, \eta) \\
&\quad + 1 - \sum_{|\pi| \leq n, \pi_1 = C,(s,h), \pi_k = C_k,(s_k,h_k), C_k \in \{\downarrow, \lightning\}} \prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}).
\end{aligned}$$

For the induction step, we need to prove that

$$\begin{aligned}
&\text{wlestep}^{n+1}[\![C]\!](X)(s, h, \eta) \\
= &\inf_{\mathfrak{s} \in \mathbb{S}} \sum_{|\pi| \leq n+1, \pi_1 = C,(s,h), \pi_k = \downarrow,(s_k,h_k)} \left(\prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \right) \cdot X(s_k, h_k, \eta) \\
&\quad + 1 - \sum_{|\pi| \leq n+1, \pi_1 = C,(s,h), \pi_k = C_k,(s_k,h_k), C_k \in \{\downarrow, \lightning\}} \prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}).
\end{aligned}$$

In case that $C = \lightning$ we immediately have that both sides are 0 since $\lightning$ is a final state and the addition does not add values ending in $\lightning$. In case that $C = \downarrow$ we immediately have that both sides are X since $\downarrow$ is a final state. Thus we assume that $C \notin \{\downarrow, \lightning\}$.

By extracting the first element of the big product we obtain:

$$
\begin{aligned}
&\inf_{\mathfrak{s}\in\mathbb{S}} \sum_{|\pi|\le n+1, \pi_1=C,(s,h), \pi_k=\downarrow,(s_k,h_k)} \left(\prod_{1\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}})\right) \cdot X(s_k, h_k, \eta) \\
&\quad + 1 - \sum_{|\pi|\le n+1, \pi_1=C,(s,h), \pi_k=C_k,(s_k,h_k), C_k\in\{\downarrow,\lightning\}} \prod_{1\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}) \\
=\ &\inf_{\mathfrak{s}\in\mathbb{S}} \sum_{|\pi|\le n+1, \pi_1=C,(s,h), \pi_k=\downarrow,(s_k,h_k)} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\pi_1, \pi_1\pi_2) \\
&\quad \cdot \left(\prod_{2\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}})\right) \cdot X(s_k, h_k, \eta) \\
&\quad + 1 - \sum_{|\pi|\le n+1, \pi_1=C,(s,h), \pi_k=C_k,(s_k,h_k), C_k\in\{\downarrow,\lightning\}} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\pi_1, \pi_1\pi_2) \\
&\quad \cdot \prod_{2\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}).
\end{aligned}
$$

Now we can split the sum by first summing over all possible transitions in the first step and then the rest of the path:

$$
\begin{aligned}
=\ &\inf_{\mathfrak{s}\in\mathbb{S}} \sum_{C,(s,h) \xrightarrow[\mathfrak{s}(C,(s,h))]{p} C',(s',h')} \ \sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=\downarrow,(s_k,h_k)} \\
&\qquad \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\pi_1, \pi_1\pi_2) \cdot \left(\prod_{2\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}})\right) \cdot X(s_k, h_k, \eta) \\
&+ 1 - \sum_{C,(s,h) \xrightarrow[\mathfrak{s}(C,(s,h))]{p} C',(s',h')} \ \sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=C_k,(s_k,h_k), C_k\in\{\downarrow,\lightning\}} \\
&\qquad \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\pi_1, \pi_1\pi_2) \cdot \prod_{2\le i<k} \mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\overline{\pi_i}, \overline{\pi_{i+1}}).
\end{aligned}
$$

Using distributivity we can extract the probability of the first transition and replace $\mathbb{P}_{\mathfrak{s}}^{C,(s,h)}(\pi_1, \pi_1\pi_2) = p$ to obtain:

$$
\begin{aligned}
=\ &\inf_{\mathfrak{s}\in\mathbb{S}} \sum_{C,(s,h) \xrightarrow[\mathfrak{s}(C,(s,h))]{p} C',(s',h')} p \cdot \sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=\downarrow,(s_k,h_k)} \text{—"—} \\
&\quad + 1 - \sum_{C,(s,h) \xrightarrow[\mathfrak{s}(C,(s,h))]{p} C',(s',h')} p \\
&\quad \cdot \sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=C_k,(s_k,h_k), C_k\in\{\downarrow,\lightning\}} \text{—"—}.
\end{aligned}
$$

Further using distributivity and associativity/commutativity of addition, we obtain

$$
\begin{aligned}
=\ &\inf_{\mathfrak{s}\in\mathbb{S}} \sum_{C,(s,h) \xrightarrow[\mathfrak{s}(C,(s,h))]{p} C',(s',h')} p \cdot \Bigg(\sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=\downarrow,(s_k,h_k)} \text{—"—} \\
&\qquad + 1 - \sum_{|\pi|\le n, \pi_1=C,(s,h), \pi_2=C',(s',h'), \pi_k=C_k,(s_k,h_k), C_k\in\{\downarrow,\lightning\}} \text{—"—} \Bigg).
\end{aligned}
$$

We can split the infimum $\inf_{\mathfrak{s}\in\mathbb{S}} \ldots$ into $\inf_{a\in\mathrm{Act}(C,(s,h))} \inf_{\mathfrak{s}\in\mathbb{S}} \ldots$ since from the scheduler $\mathfrak{s}$ we can construct the action $\mathfrak{s}(\sigma) \in \mathrm{Act}(\sigma)$ and the scheduler $\mathfrak{s}'(\pi) =$

$\mathfrak{s}(\sigma\pi)$ to achieve the same value and vice verse we can construct from an action $a \in \mathrm{Act}(\sigma)$ and a scheduler $\mathfrak{s}$ the new scheduler defined as $\mathfrak{s}'(\sigma) = a$ and $\mathfrak{s}'(\sigma_1 \ldots \sigma_k) =$ if $(\sigma_1 = \sigma)$ then $\mathfrak{s}(\sigma_2 \ldots \sigma_k)$ else $\mathfrak{s}(\sigma_1 \ldots \sigma_k)$. Thus we obtain

$$= \inf_{a \in \mathrm{Act}(C,(s,h))} \inf_{\mathfrak{s} \in \mathbb{S}} \sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p \cdot \text{—"—}$$

$$= \inf_{a \in \mathrm{Act}(C,(s,h))} \sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p$$

$$\cdot \inf_{\mathfrak{s} \in \mathbb{S}} \Bigg(\sum_{|\pi| \leq n, \pi_1 = C',(s',h'), \pi_k = \downarrow,(s_k,h_k)} \cdot \Bigg(\prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C',(s',h')}(\pi_i, \pi_{i+1}) \Bigg) \cdot X(s_k, h_k, \eta)$$

$$+ 1 - \sum_{|\pi| \leq n, \pi_1 = C',(s',h') \pi_k = C_k,(s_k,h_k), C_k \in \{\downarrow, \lightning\}} \prod_{1 \leq i < k} \mathbb{P}_{\mathfrak{s}}^{C',(s',h')}(\pi_i, \pi_{i+1}) \Bigg).$$

By applying now the induction hypothesis, we obtain

$$= \inf_{a \in \mathrm{Act}(C,(s,h))} \sum_{C,(s,h) \xrightarrow[a]{p} C',(s',h')} p \cdot \mathsf{wlestep}^n [\![C']\!] (X) (s', h', \eta).$$

□

Example 6.2.1 Revisiting Example 6.1.1, we compute the weakest resource-safe liberal expectation with respect to the random variable $[x = 0]$ ad-hoc for the MDP presented there. That is, for the program C given as

```
          <r> := −1;
pif (0.5) {      ‖
   <r> := 0      ‖  x := <r>;
}else{           ‖  while(x = −1){x := <r>}
   <r> := 1      ‖
}                ‖
```

we compute $\mathsf{wrle}[\![C]\!]([x = 0] \mid [\mathsf{emp}])(s, h)$ where the initial program state s, h is defined as $\forall x \neq r.\, s(x) = 0, s(r) = 1$ and $h = \{1 \mapsto 0\}$. We apply Theorem 6.2.2 in order to avoid computing the largest fixed point. It is now sufficient to compute the value for the actions realizing the smallest value. This is the case for any fair schedule. Indeed (but without proof), any fair schedule will realize the same value in this example. We thus consider the schedule that first always picks the thread on the left-hand side and then the thread on the right-hand side. Furthermore, since any fair schedule will yield a finite induced Markov chain, it is sufficient to consider a sufficiently big n for transition steps. Following the reasoning of Example 6.1.1, we can conclude that we reach a state $\downarrow, (s, h)$ with $s(x) = 1$ with probability 0.5 and furthermore we have that $[x = 0](s, h) = 1$, thus we obtain $\mathsf{wrle}[\![C]\!]([x = 0] \mid [\mathsf{emp}])(s, h) = 0.5$.

This demonstrates that we can indeed compute probabilities for realizing a condition after termination.

Example 6.2.2 In this example we will consider an alternative view on resource invariants. For this, we explore an alternative (less elegant) way to handle concurrency. Instead of defining rules to handle concurrency locally, one may also transform a concurrent program into a sequential program with

non-determinism and verify the sequential program instead. Consider a program of the form $C_1 \parallel C_2$ and let x, y_1, y_2 be fresh variables. Then we can write a program that non-deterministically either assigns $x = 1$ or $x = 2$. If $x = 1$, we execute a step in program C_1 at step y_1 and if $x = 2$, we execute a step in program C_1 at step y_2. Let us be more concrete and say we transform the program

```
<z> := 1 ∥ <z> := 2
```

into the program

```
y₁ := 0;
y₂ := 0;
while(y₁ ≠ −1 ∨ y₂ ≠ −1){
  if (nondet) {x := 1} else {x := 2};
  if (x = 1 ∨ y₂ = −1) {
    <z> := 1;
    y₁ := −1
  }
  if (x = 2 ∨ y₁ = −1) {
    <z> := 2;
    y₂ := −1
  }
}
```

to reason about this instead. A resource invariant for the original program, for example $[z \mapsto 1] \vee [z \mapsto 2]$ is now *also a loop invariant* for the transformed program. That is, a resource invariant can be used with possibly minor adaptions as a loop invariant in the sequential program simulating the execution flow of the concurrent program. Remark that the other way does not work. Loop invariants may have access to variables y_1, y_2, x, which the resource invariant has no access to.

6.3. Axiomatic Semantics for Lower Bounds on Concurrent Probabilistic Programs

Computing the Markov decision process semantics and its induced Markov chain is not always possible as the (1) the Markov chain is usually infinite, (2) the Markov decision process may be infinite, and (3) we would like to reason about the Markov decision process for each initial state and not only for one. While problem (1) can be solved by computing the value on the MDP immediately instead, the other two problems require different semantics. We propose here to use *axiomatic semantics* in order to deal with concurrency, which is otherwise difficult to handle. The semantics we offer allow proving a lower bound of wrle and thus can especially be used to compute lower bounds of probabilities for a certain condition to hold. In principle, we can also compute the lower bounds of *expected values* for random variables. However, it is rare that we are interested in the expected value of a random variable that is bound in the unit interval $[0, 1]$.

We notate lower bound specifications in a tuple containing the lower bound, the program, the random variable and the resource invariant. This is analogous to the specifications we introduced in Chapter 3.

Definition 6.3.1 (Lower Bound Specification for cpPL Programs) *We let* $X, Y, R \in \mathsf{RV}^1$ *be random variables and* C *a* cpPL *program. We define the lower bound specification* $R \vdash \{X\}_{\leq}\ C\ \{Y\}$ *as*

$$R \vdash \{X\}_{\leq}\ C\ \{Y\} \qquad \Longleftrightarrow \qquad X \models \mathsf{wrle}[\![C]\!]\,(Y \mid R)\,.$$

From Theorem 6.2.1, we can already and immediately derive the next conservativity theorem concerning lower bound specifications.

Corollary 6.3.1 *Let* C *be a* cPL *program,* $\Phi \in \mathsf{SLProp}$ *a precondition,* $\Psi \in \mathsf{SLProp}$ *a postcondition and* $\xi \in \mathsf{SLProp}$ *a resource invariant in qualitative separation logic. We have that*

$$[\xi] \vdash \{[\Phi]\}_{\leq}\ C\ \{[\Psi]\} \qquad \Longleftrightarrow \qquad \xi \vdash \{\Phi\}\ C\ \{\Psi\},$$

where the right-hand side is as defined in Definition 3.3.2.

Using the specifications from Definition 6.3.1, we define proof rules shown in Figure 6.3 allowing the verification of such specifications. These proof rules constitute the axiomatic semantics for concurrent probabilistic programs. The proof rules look very similar to the proof rules from Figures 3.3 and 4.3. We have proof rules for every program construct. A key difference between the proof rules presented here and the proof rules from Chapter 4 is that we swapped the order of the entailment relation again (in the rules for loops and monotonicity), as we now argue about *lower bounds* instead of upper bounds. This also enables us to eliminate maxima instead of minima. We do not have a rule for minima at all, not even if the resource invariant is precise. The reason is that the weakest liberal expectation does not nicely behave for minima.

We could also add a proof rule combining cas-true and cas-false. Our Lean implementation also features this additional proof rule.

This is different in the qualitative setting where we have a conjunction elimination rule.

Example 6.3.1 Consider the program

$$C = \texttt{pif}\ (0.5)\ \{x := 1\}\ \texttt{else}\ \{x := 2\}.$$

Then we have that $\mathsf{wrle}[\![C]\!]\,([x = 1] \sqcap [x = 2] \mid [\mathsf{emp}]) = [\mathsf{false}]$ as we have $[x = 1] \sqcap [x = 2] = [\mathsf{false}]$ and as **wrle** is monotone. Moreover, remark that $[\mathsf{emp}]$ is precise. However, if we would eliminate the minima using an unsound minima elimination rule, we have $\mathsf{wrle}[\![C]\!]\,([x = 1] \mid [\mathsf{emp}]) = \langle 0.5 \rangle$ and $\mathsf{wrle}[\![C]\!]\,([x = 2] \mid [\mathsf{emp}]) = \langle 0.5 \rangle$, and it should hold that

$$\langle 0.5 \rangle \sqcap \langle 0.5 \rangle \models \mathsf{wrle}[\![C]\!]\,([x = 1] \sqcap [y = 1] \mid [\mathsf{emp}])\,.$$

However, we have that

$$\langle 0.5 \rangle \sqcap \langle 0.5 \rangle = \langle 0.5 \rangle \not\models [\mathsf{false}] = \mathsf{wrle}[\![C]\!]\,([\,] \mid x = 1) \sqcap [y = 1]\,[\mathsf{emp}].$$

Therefore the rule is wrong.

Since in FSL we have introduced the quantitative negation $\sim X$, we also use the quantitative negation and replace the qualitative negation inside of Iverson

$$\frac{}{R \vdash \{X\}_{\leq}\ \texttt{skip}\ \{X\}}\ \text{skip} \qquad \frac{x \notin \mathrm{Vars}(R)}{R \vdash \{X[x := e]\}_{\leq}\ x\ \texttt{:=}\ e\ \{X\}}\ \text{assign}$$

$$\frac{x \notin \mathrm{Vars}(R)}{R \vdash \{\text{Ƨ}a.\ [e \mapsto a] \star ([e \mapsto a] \mathbin{-\!\!\circledast} X[x := a])\}_{\leq}\ x\ \texttt{:=}\ \texttt{<}e\texttt{>}\ \{X\}}\ \text{lookup}$$

$$\frac{}{R \vdash \left\{(\text{Ƨ}a.\ [e \mapsto a]) \star \left([e \mapsto e'] \mathbin{-\!\!\circledast} X\right)\right\}_{\leq}\ \texttt{<}e\texttt{>}\ \texttt{:=}\ e'\ \{X\}}\ \text{mut}$$

$$\frac{x \notin \mathrm{Vars}(R)}{R \vdash \left\{[e \in \mathbb{N}] \cdot \mathrm{I}\,a.\ \mathop{\bigstar}_{i=0}^{e-1} [a + i \mapsto 0] \mathbin{-\!\!\circledast} X[x := a]\right\}_{\leq}\ x\ \texttt{:=}\ \texttt{new}(e)\ \{X\}}\ \text{alloc}$$

$$\frac{}{R \vdash \left\{[e' \in \mathbb{N}] \cdot \left(\mathop{\bigstar}_{i=0}^{e'-1} \text{Ƨ}a.\ [e + i \mapsto a]\right) \star X\right\}_{\leq}\ \texttt{free}(e, e')\ \{X\}}\ \text{dealloc}$$

$$\frac{x \notin \mathrm{Vars}(R)}{R \vdash \{[e_l \mapsto e_c] \star ([e_l \mapsto e_s] \mathbin{-\!\!\circledast} X[x := 1])\}_{\leq}\ x\ \texttt{:=}\ \texttt{CAS}(e_l, e_c, e_s)\ \{X\}}\ \text{cas-true}$$

$$\frac{x \notin \mathrm{Vars}(R)}{R \vdash \{\text{Ƨ}a.([e_l \mapsto a] \cdot \sim[a = e_c]) \star ([e_l \mapsto a] \mathbin{-\!\!\circledast} X[x := 0])\}_{\leq}\ x\ \texttt{:=}\ \texttt{CAS}(e_l, e_c, e_s)\ \{X\}}\ \text{cas-false}$$

$$\frac{R \vdash \{X_1\}_{\leq}\ C_1\ \{Y\} \qquad R \vdash \{X_2\}_{\leq}\ C_2\ \{Y\}}{R \vdash \{[e_b] \cdot X_1 \sqcup \sim[e_b] \cdot X_2\}_{\leq}\ \texttt{if}\ (e_b)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{Y\}}\ \text{if}$$

$$\frac{R \vdash \{X_1\}_{\leq}\ C_1\ \{Y\} \qquad R \vdash \{X_2\}_{\leq}\ C_2\ \{Y\}}{R \vdash \{\langle e\rangle \cdot X_1 + \sim\langle e\rangle \cdot X_2\}_{\leq}\ \texttt{pif}\ (e)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{Y\}}\ \text{pif}$$

$$\frac{R \vdash \{X\}_{\leq}\ C_1\ \{Y\} \qquad R \vdash \{Y\}_{\leq}\ C_2\ \{Z\}}{R \vdash \{X\}_{\leq}\ C_1 ; C_2\ \{Z\}}\ \text{seq}$$

$$\frac{I \vDash [e_b] \cdot X \sqcup \sim[e_b] \cdot Y \qquad R \vdash \{X\}_{\leq}\ C\ \{I\}}{R \vdash \{I\}_{\leq}\ \texttt{while}(e_b)\{C\}\ \{Y\}}\ \text{while}$$

$$\frac{R \vdash \{X_1\}_{\leq}\ C_1\ \{Y_1\} \qquad R \vdash \{X_2\}_{\leq}\ C_2\ \{Y_2\} \qquad \forall i \in \{1,2\}.\ \mathrm{Write}(C_i) \cap \mathrm{Vars}(C_{3-i}, Y_{3-i}, R) = \emptyset}{R \vdash \{X_1 \star X_2\}_{\leq}\ C_1 \parallel C_2\ \{Y_1 \star Y_2\}}\ \text{concur}$$

$$\frac{R \star Q \vdash \{X\}_{\leq}\ C\ \{Y\}}{R \vdash \{X \star Q\}_{\leq}\ C\ \{Y \star Q\}}\ \text{share} \qquad \frac{[\mathrm{emp}] \vdash \{X \star R\}_{\leq}\ C\ \{Y \star R\} \qquad C \text{ is a terminating atom}}{R \vdash \{X\}_{\leq}\ C\ \{Y\}}\ \text{atom}$$

$$\frac{R \vdash \{X\}_{\leq}\ C\ \{Y\} \qquad R \vdash \{X'\}_{\leq}\ C\ \{Y'\}}{R \vdash \{X \sqcup X'\}_{\leq}\ C\ \{Y \sqcup Y'\}}\ \text{max}$$

$$\frac{R \vdash \{X\}_{\leq}\ C\ \{X'\} \qquad \mathrm{Write}(C) \cap \mathrm{Vars}(Y) = \emptyset}{R \vdash \{X \star Y\}_{\leq}\ C\ \{X' \star Y\}}\ \text{frame}$$

$$\frac{R \vdash \{X\}_{\leq}\ C\ \{Y\} \qquad R \vdash \{X'\}_{\leq}\ C\ \{Y'\} \qquad R \text{ is precise} \qquad \mathrm{Write}(C) \cap \mathrm{Vars}(e) = \emptyset}{R \vdash \{\langle e\rangle \cdot X \dotplus \sim\langle e\rangle \cdot X'\}_{\leq}\ C\ \{Y \sqcup Y'\}}\ \text{weight}$$

$$\frac{X \vDash X' \qquad R \vdash \{X'\}_{\leq}\ C\ \{Y'\} \qquad Y' \vDash Y}{R \vdash \{X\}_{\leq}\ C\ \{Y\}}\ \text{monotonicity}$$

Figure 6.3.: Axiomatic semantics using the notation defined in Definition 6.3.1. We use here semantics similar to denotational semantics, as they are usually depicted denotational and not axiomatic. The first proof rules deal with atomic statements, followed by flow related programs and lastly elimination proof rules.

brackets. We use the quantitative negation to express the convex sum in the pif rule and the weight rule.

For the frame rule, the weight rule and the concurrency rule, we have extra conditions on the variables that are written to in the program. We define the variables that are written to as the variables that appear left of an assignment or a lookup. The atom rule is only applicable for terminating atoms. A terminating atom is a program that terminates after exactly one step. This is similar to how we defined these in Chapter 3.

Definition 6.3.2 *Let C be a* cpPL *Program.*

- *We define $Write(C)$ as the set of variables which occur on the left-hand side of an assignment or lookup in C.*
- *We define $Vars(C)$ as the set of variables occurring in C and we use the shorthand notation $Vars(C, X, Y) = Vars(C) \cup Vars(X) \cup Vars(Y)$.*
- *We say C is a terminating atom if for $C, (s, h) \xrightarrow[a]{p} C', (s, h')$ with $p > 0$ we always have $C' \in \{ \downarrow, \lightning \}$.*

The soundness of the proof rules is proven using induction on the fixed point defined in Definition 6.2.1. Remark that we have not proved that **wrle** is co-ω-Scott-continuous. We only proved this for empty resource invariants. We thus perform transfinite induction on ordinals using Theorem 2.3.3 to prove the soundness.

Theorem 6.3.2 *The proof rules in Figure 6.3 are sound.*

Proof. See [43] at `LeanFSL.CFSL.SafeTuple`. ☐

We will also use an annotated program style of representing proofs for specification. Similarly to Chapter 3, we have that

$$\begin{array}{l} \quad /\!\!/_{\leq} \; X \\ C \\ \quad /\!\!/_{\leq} \; Y \end{array}$$

means $R \vdash \{X\}_{\leq} \; C \; \{Y\}$ for some R that is clear from the context. If we want to denote the resource invariant explicitly, we also write

$$\begin{array}{l} \quad /\!\!/_{\leq} \; X \mid R \\ C \\ \quad /\!\!/_{\leq} \; Y \mid R \end{array}$$

for $R \vdash \{X\}_{\leq} \; C \; \{Y\}$. Remark, that the resource invariant needs to be equal. When using the rules share and atom, this notation may result in confusing notations, similar to the notation used in Chapter 3.

Example 6.3.2 We will now finally prove the specification of the running example from Examples 6.1.1 and 6.2.1. That is, for the program C given as

```
          <r> := −1;
pif (0.5) {           ‖
   <r> := 0           ‖ x := <r>;
} else {              ‖ while (x = −1) { x := <r> }
   <r> := 1           ‖
}                     ‖
```

we aim to verify the specification

$$[\mathsf{emp}] \vdash \{\langle 0.5\rangle \star \text{Ꝛ}a.\ [r \mapsto a]\}_{\leq}\ C \left\{ \begin{array}{c} [\mathsf{true}] \star [x = 0] \\ \star\ ([r \mapsto -1] \sqcup [r \mapsto 1]) \end{array} \right\}.$$

We will now use $R = [r \mapsto -1] \sqcup [r \mapsto 0]$ to shorten what will become the resource invariant in our proof. It is easy to spot that the location of r is a shared memory, thus it is also naturally to include it in the resource invariant. We include the possible values -1 and 0 since we know that if the heap location ever stores 1, the random variable $[x = 0]$ will yield zero, thus not contributing to the probability mass. An annotated version of the program is the following

```
//≤ ⟨0.5⟩ ⋆ Ꝛa. [r ↦ a] | [emp]
       <r> := −1;
//≤ ⟨0.5⟩ ⋆ [true] ⋆ ([r ↦ −1] ⊔ [r ↦ 1]) | [emp]
//≤ ⟨0.5⟩ ⋆ [true] | R
   //≤ ⟨0.5⟩ | R           ‖
  pif (0.5) {              ‖
   //≤ [true] | R          ‖  //≤ [true] | R
     <r> := 0              ‖  x := <r>;
   //≤ [true] | R          ‖  //≤ [x = 0] ⊓ [x = 1] | R
  } else {                 ‖  while (x = −1) { x := <r> }
   //≤ [false] | R         ‖  //≤ [x = 0] | R
     <r> := 1              ‖
   //≤ [false] | R         ‖
  }                        ‖
//≤ [true] ⋆ [x = 0] | R
//≤ [true] ⋆ [x = 0] ⋆ ([r ↦ −1] ⊔ [r ↦ 1]) | [emp]
```

to help follow our proof.
For the proof, we now apply the seq proof rule to obtain the two proof obligations

$$[\mathsf{emp}] \vdash \{\langle 0.5\rangle \star \text{Ꝛ}a.\ [r \mapsto a]\}_{\leq}\ \texttt{<r> := -1}\ \{\langle 0.5\rangle \star [r \mapsto -1]\},$$
$$[\mathsf{emp}] \vdash \{\langle 0.5\rangle \star [r \mapsto -1]\}_{\leq}\ C_1 \parallel C_2\ \{[\mathsf{true}] \star [x = 0] \star R\},$$

where C_1 and C_2 are the two threads.

Initialization. The first proof obligation can be proven by first applying the frame rule to frame away $\langle 0.5 \rangle$, then applying monotonicity and proving the entailment

$$\text{Ƨ}a.\ [r \mapsto a] \models (\text{Ƨ}a.\ [r \mapsto a]) \star ([r \mapsto -1] \mathbin{-\!\!\circledast} [r \mapsto -1]),$$

followed by applying the mut proof rule. The entailment holds by extending it with [emp], using monotonicity of the separating multiplication to get rid of $\text{Ƨ}a.\ [r \mapsto a]$, followed by using adjointness of the magic wand.
Concurrency. We prove the proof obligation

$$[\mathrm{emp}] \ \vdash\ \{\langle 0.5 \rangle \star [r \mapsto -1]\}_{\leq}\ C_1 \parallel C_2\ \{[\mathrm{true}] \star [x = 0] \star R\}$$

by first applying monotonicity and prove the entailment $\langle 0.5 \rangle \star [r \mapsto -1] \models \langle 0.5 \rangle \star R$ to obtain

$$[\mathrm{emp}] \ \vdash\ \{\langle 0.5 \rangle \star R\}_{\leq}\ C_1 \parallel C_2\ \{[\mathrm{true}] \star [x = 0] \star R\}.$$

Applying the share proof rule, we obtain the proof obligation

$$[\mathrm{emp}] \star R \ \vdash\ \{\langle 0.5 \rangle\}_{\leq}\ C_1 \parallel C_2\ \{[\mathrm{true}] \star [x = 0]\}.$$

We have that $[\mathrm{emp}] \star R$ is equal to R. We will not mention this equality further and use it whenever convenient. Now we can rewrite $\langle 0.5 \rangle$ to $\langle 0.5 \rangle \star [\mathrm{true}]$ to obtain

$$R \ \vdash\ \{\langle 0.5 \rangle \star [\mathrm{true}]\}_{\leq}\ C_1 \parallel C_2\ \{[\mathrm{true}] \star [x = 0]\}$$

and apply the concur proof rule. With this we obtain the four new proof obligations

A specification with [true] on the right-hand side and a resource invariant R is not trivial, as we require that R is always reestablished.

$$\begin{gathered} R \ \vdash\ \{\langle 0.5 \rangle\}_{\leq}\ C_1\ \{[\mathrm{true}]\}, \\ R \ \vdash\ \{[\mathrm{true}]\}_{\leq}\ C_2\ \{[x = 0]\}, \\ \mathrm{Write}(C_1) \cap \mathrm{Vars}\,(C_2, [x = 0], R) = \emptyset, \\ \mathrm{Write}(C_2) \cap \mathrm{Vars}\,(C_1, [\mathrm{true}], R) = \emptyset. \end{gathered}$$

The last two proof obligations are simple: $\mathrm{Write}(C_1) = \emptyset$ since there is no assign or lookup, thus the obligation holds immediately, and $\mathrm{Write}(C_2) = \{x\}$, but $x \notin \mathrm{Vars}\,(C_1, [\mathrm{true}], R)$.
First Thread. For the proof obligation

$$R \ \vdash\ \{\langle 0.5 \rangle\}_{\leq}\ C_1\ \{[\mathrm{true}]\}$$

we apply monotonicity and by proving the entailment

$$\langle 0.5 \rangle \models \langle 0.5 \rangle \cdot [\mathrm{true}] \dotplus {\sim}\langle 0.5 \rangle \cdot [\mathrm{false}]$$

(which is simple), we obtain the proof obligation

$$R \ \vdash\ \{\langle 0.5 \rangle \cdot [\mathrm{true}] \dotplus {\sim}\langle 0.5 \rangle \cdot [\mathrm{false}]\}_{\leq}\ C_1\ \{[\mathrm{true}]\}.$$

We apply the pif proof rule and obtain the two new proof obligations

We remark that every specification with [false] on the left-hand side holds as it simplifies to an inequality of the form $0 \leq \ldots$ and 0 is the bottom element.

$$\begin{gathered} R \ \vdash\ \{[\mathrm{true}]\}_{\leq}\ \texttt{<r>} := 0\ \{[\mathrm{true}]\} \\ R \ \vdash\ \{[\mathrm{false}]\}_{\leq}\ \texttt{<r>} := 1\ \{[\mathrm{true}]\}. \end{gathered}$$

We apply the atom proof rule on the first to obtain the proof obligations

$$[\mathsf{emp}] \vdash \{[\mathsf{true}] \star R\}_{\leq} \ \texttt{<}r\texttt{>} := 0 \ \{[\mathsf{true}] \star R\},$$
$$\texttt{<}r\texttt{>} := 0 \text{ is a terminating atom.}$$

The second proof obligation is trivial to prove. For the first two, we apply monotonicity to obtain

$$[\mathsf{emp}] \vdash \{(\text{Ꙅ}a.\ [r \mapsto a]) \star ([r \mapsto 1] \multimap\!\!\circledast X)\}_{\leq} \ \texttt{<}r\texttt{>} := 0 \ \{X\},$$

where $X = [\mathsf{true}] \star R$ by proving the entailment

$$[\mathsf{true}] \star R \models (\text{Ꙅ}a.\ [r \mapsto a]) \star ([r \mapsto 0] \multimap\!\!\circledast [\mathsf{true}] \star R).$$

We apply commutativity and monotonicity of the separating multiplication and it remains to prove that

$$[\mathsf{true}] \models [r \mapsto 0] \multimap\!\!\circledast [\mathsf{true}] \star R,$$
$$R \models \text{Ꙅ}a.\ [r \mapsto a].$$

The first entailment holds using adjointness of the magic wand, getting rid of [true] using monotonicity of the separating multiplication, and choosing the right-hand side of the maximum in R to match against. The second entailment holds by applying a case distinction on the maximum in R and choosing a adequately. The proof obligation

$$[\mathsf{emp}] \vdash \{(\text{Ꙅ}a.\ [r \mapsto a]) \star ([r \mapsto 1] \multimap\!\!\circledast X)\}_{\leq} \ \texttt{<}r\texttt{>} := 0 \ \{X\}$$

is proven using the mut proof rule.
For the proof obligation

$$R \vdash \{[\mathsf{false}]\}_{\leq} \ \texttt{<}r\texttt{>} := 1 \ \{[\mathsf{true}]\},$$

we do not need to use the atom proof rule. Notice that nothing stops us from using the mut proof rule without first gaining access to the resource invariant, even if we change shared memory. We may, however, obtain trivial results. This is also here the case, where we prove the trivial lower bound [false]. Thus by using monotonicity we obtain

$$R \vdash \{(\text{Ꙅ}a.\ [r \mapsto a]) \star ([r \mapsto 1] \multimap\!\!\circledast [\mathsf{true}])\}_{\leq} \ \texttt{<}r\texttt{>} := 1 \ \{[\mathsf{true}]\},$$

which we prove using the mut proof rule.
Second Thread. For the proof obligation

$$R \vdash \{[\mathsf{true}]\}_{\leq} \ C_2 \ \{[x = 0]\},$$

we first apply the seq rule to eliminate sequential composition. We obtain the proof obligations

$$R \vdash \{[\mathsf{true}]\}_{\leq} \ x := \texttt{<}r\texttt{>} \ \{[x = -1] \sqcup [x = 0]\},$$
$$R \vdash \{[x = -1] \sqcup [x = 0]\}_{\leq} \ \texttt{while}(x = -1)\{x := \texttt{<}r\texttt{>}\} \ \{[x = 0]\}.$$

For the first proof obligation, we apply the atom proof rule to obtain

$$[\mathsf{emp}] \vdash \{[\mathsf{true}] \star R\}_{\leq} \ x := \texttt{<}r\texttt{>} \ \{X \star R\},$$

where we shorten $X = [x = -1] \sqcup [x = 0]$. Furthermore, we apply monotonicity by proving the entailment

$$[\mathsf{true}] \star R \models \text{Ƨ}a.\ [r \mapsto a] \star ([r \mapsto a] \mathrel{-\!\!\circledast} (X \star R)[x := a])$$

to obtain a specification that we prove by using the lookup proof rule. The entailment is proven by first applying distributivity of separating multiplication with maxima and then eliminating the maximum on the left to obtain the entailments

$$\begin{aligned}
[\mathsf{true}] \star [r \mapsto -1] &\models \text{Ƨ}a.\ [r \mapsto a] \star ([r \mapsto a] \mathrel{-\!\!\circledast} (X \star R)[x := a]),\\
[\mathsf{true}] \star [r \mapsto 0] &\models \text{Ƨ}a.\ [r \mapsto a] \star ([r \mapsto a] \mathrel{-\!\!\circledast} (X \star R)[x := a]).
\end{aligned}$$

We prove both analogously: We first pick a as -1 and 0 respectively, use commutativity and monotonicity of the separating multiplication to remove $[r \mapsto -1]$ and $[r \mapsto 0]$ respectively, apply adjointness of the magic wand, eliminate the substitution and apply monotonicity to obtain the proof obligations

$$\begin{aligned}
[\mathsf{true}] &\models [-1 = -1] \sqcup [0 = -1]\,,\\
[r \mapsto -1] &\models [r \mapsto -1] \sqcup [r \mapsto 0]\,,\\
[\mathsf{true}] &\models [0 = -1] \sqcup [-1 = -1]\,,\\
[r \mapsto 0] &\models [r \mapsto -1] \sqcup [r \mapsto 0]\,,
\end{aligned}$$

which we can all prove by choosing the adequate side of the maximum and simplifying equations.
For the second proof obligation, we need to find a loop invariant. We choose the value $X = [x = -1] \sqcup [x = 0]$ and apply the while proof rule to obtain the proof obligations

$$\begin{gathered}
[x = -1] \sqcup [x = 0] \models [x = -1] \cdot X \sqcup \sim [x = -1] \cdot [x = 0]\\
R \vdash \{X\}_{\leq}\ \texttt{x := <r>}\ \{[x = -1] \sqcup [x = 0]\}
\end{gathered}$$

Since $X = [x = -1] \sqcup [x = 0]$ the entailment is straightforward to prove. The second proof obligation is proven analogously to the previous specification for a lookup.
This concludes the proof for the given specification. This proof has also been formalized in [43] at `LeanFSL.Examples.SendData`.

6.4. Lossy Producer Consumer Example

In this section, we will prove a specification on a program that is not easily also proven using model checking techniques by deriving a Markov decision process from the program semantics. In this example, the program provided in Figure 6.4 consists of three threads: (1) a producer who generates some data and saves it in a shared array at location z_1, (2) a lossy channel that takes the values from the shared array at location z_1, may discards the value and replaces it by an invalid data with a certain probability p and saves it in another array at location z_2 shared with the last thread, and finally (3) a consumer that reads the value from the shared array at location z_2 and counts how many data items of the $y + 1$ transmitted were valid.

```
l := 0;
y1 := y; y2 := y; y3 := y;

while(0 ≤ y1){               ||  while(0 ≤ y2){                ||  while(0 ≤ y3){
  pif (0.5) {                ||    x2 := <z1 + y2>;            ||    x3 := <z2 + y3>;
    x1 := 1                  ||    if (x2 ≠ 0) {               ||    if (x3 ≠ 0) {
  }else{                     ||      pif (p) {                 ||      if (x3 ≠ −1) {l := l + 1};
    x1 := 2                  ||        <z2 + y2> := x2         ||      y3 := y3 − 1
  };                         ||      }else{                    ||    }
  <z1 + y1> := x1;           ||        <z2 + y2> := −1         ||  }
  y1 := y1 − 1               ||      }                         ||
}                            ||      y2 := y2 − 1              ||
                             ||    }                           ||
                             ||  }                             ||
```

Figure 6.4.: A program modeling a producer consumer protocol that is extended by a third thread modeling a lossy channel between both other threads. The producer thread (left) generates data and saves it in an array that the lossy channel has access to. The lossy channel (middle) discards the data with a certain probability and saves the result in a second array that the consumer has access to. The consumer (right) counts the amount of valid data.

We generally encode with 1 and 2 generated values, with 0 an unused location and with -1 an invalid (corrupted) value. Since for each array only one thread only writes and only one thread only reads, we do not need any mechanisms to guarantee exclusive access. At several places we use conditional branching without else branch. In these cases, the else branch consists of a skip and is left out for brevity reasons. We also leave out allocation and freeing of the shared memory for brevity reasons. For an easier proof, we assume $y \in \mathbb{N}$ and $p \in [0, 1]$ to be constants and not program variables. However, the proof can also be done with both of these as program variables.

The specification we are interested in is the probability that all data is send successfully depending on the probability p. Lower bounding this probability is useful in order to find values for p which guarantee a certain probability on the successful transmission of the generated data. It turns out that this probability is actually p^{y+1}. We also have to guarantee the existence of the arrays, which we can do using the random variable R

R will become the resource invariant later.

$$\begin{aligned} R \quad = \quad & \bigstar_{i=0}^{y} ([z_1 + i \mapsto 0] \sqcup [z_1 + i \mapsto 1] \sqcup [z_1 + i \mapsto 2]) \\ & \star \bigstar_{i=0}^{y} ([z_2 + i \mapsto 0] \sqcup [z_2 + i \mapsto 1] \sqcup [z_2 + i \mapsto 2]) \ . \end{aligned}$$

For C as in Figure 6.4, we thus verify the specification

$$[\mathrm{emp}] \ \vdash \ \left\{ \langle p^{y+1} \rangle \star R \right\}_{\leq} \ C \ \{ [l = y + 1] \star R \} .$$

Intuitively the specification holds as we require that for every iteration of the loop in the second thread, the probabilistic branching needs to pick the first branch. This happens only with probability p^{y+1}. We also require that the initial values for y_1, y_2 and y_3 are valid, i.e. at least 0 to avoid trivial specifications where no

array exists in the heap. The following proof for this specification has also been formalized in [43] at `LeanFSL.CFSL.Examples.ProducerConsumer.lean`.

To prove that the specification holds, we additionally need that y_1, y_2, y_3 are always natural numbers until the loop condition does not hold anymore, at which point it is -1. Otherwise we would later not be able to extract the one index from the big separating multiplication in R. Thus we apply monotonicity to obtain the specification

$$[\mathsf{emp}] \vdash \left\{ \langle p^{y+1} \rangle \star R \right\}_{\leq} C \left\{ [l = y + 1] \star [y_i \in \{-1, \dots, y\}] \star R \right\},$$

where we shorten $[y_i \in \mathbb{Z}] \cdot [-1 \leq y_i \leq y]$ with the expression $[y_i \in \{-1, \dots, y\}]$ and the expression $[y_1 \in \{-1, \dots, y\}] \cdot [y_2 \in \{-1, \dots, y\}] \cdot [y_3 \in \{-1, \dots, y\}]$ as the expression $[y_i \in \{-1, \dots, y\}]$.

In order to prove this specification, we first split the program into the initial part and the concurrent part. For the first thread, we prove that the thread adheres to the resource invariant (which will be R) and that y_1 remains well-formed, thus we use $X_1 = [y_1 \in \{-1, \dots, y\}]$ as random variable and lower bound. For the second thread, we want to prove that the thread adheres to the resource invariant with probability p^{y+1} and that y_2 remains well-formed. Thus we use the lower bound $X_2 = [y_2 \in \{0, \dots, y\}] \cdot \langle p^{y_2+1} \rangle \sqcup [y_2 < 0] \cdot [y_2 \in \{-1, \dots, y\}]$ and the random variable $[y_2 \in \{-1, \dots, y\}]$. Remark that we require the extra condition for $y_2 = -1$ here to be inductive. Since $y_2 = y \in \mathbb{N}$, this case is superfluous, however. Finally, the last thread is supposed to count all successful transmits, thus we use the random variable $[y_3 = -1] \cdot [l = y - y_3]$ and the lower bound $X_3 = [y_3 \in \{-1, \dots, y\}] \cdot [l = y - y_3]$. We require $y_3 = -1$ in the random variable to guarantee that after the loop we have $l = y + 1$. We will adjust the random variables to these in a later step. After applying the seq proof rule, we obtain the two new proof obligations:

$$[\mathsf{emp}] \vdash \left\{ \langle p^{y+1} \rangle \star R \right\}_{\leq} C_{init} \left\{ X_1 \star X_2 \star X_3 \star R \right\}$$

$$[\mathsf{emp}] \vdash \left\{ X_1 \star X_2 \star X_3 \star R \right\}_{\leq} C_1 \parallel C_2 \parallel C_3 \left\{ \begin{array}{c} [l = y + 1] \\ \star\ [y_i \in \{-1, \dots, y\}] \star R \end{array} \right\}$$

Initialization We can prove the first obligation straightforwardly by applying the monotonicity proof rule, the seq proof rule and the assign proof rule adequately and finally have the entailment. Due to the assignments of y_1, y_2 and y_3 to y, we can replace them by y and thus the cases $[y_2 < 0]$ and $[y_3 < 0]$ can be canceled and the other parts can be simplified. The last assignment then finally allows canceling any remaining condition containing l. The following is an annotated program depicting these steps:

$$\begin{array}{l} \quad /\!\!/\!_{\leq}\ \langle p^{y+1} \rangle \star R \\ \quad /\!\!/\!_{\leq}\ \langle p^{y+1} \rangle \star [\mathsf{true}] \star R \\ l := 0\,; \\ \quad /\!\!/\!_{\leq}\ \langle p^{y+1} \rangle \star [l = 0] \star R \\ \quad /\!\!/\!_{\leq}\ [\mathsf{true}] \star \langle p^{y+1} \rangle \star [l = 0] \star R \\ y_1 := y\,;\ y_2 := y\,;\ y_3 := y\,; \\ \quad /\!\!/\!_{\leq}\ X_1 \star X_2 \star X_3 \star R \end{array}$$

Concurrency For the second proof obligation, we apply the share proof rule to R and the monotonicity proof rule to match the right-hand side for an application of the concur proof rule as discussed previously. Thus we obtain the proof obligation:

$$R \vdash \{X_1 \star X_2 \star X_3\}_{\leq} \ C_1 \parallel C_2 \parallel C_3 \ \left\{ \begin{array}{c} [y_1 \in \{-1, \dots, y\}] \\ \star \ [y_2 \in \{-1, \dots, y\}] \\ \star \ ([y_3 = -1] \cdot [l = y - y_3]) \end{array} \right\}$$

Applying the concur proof rule, we obtain the three proof obligations

$$R \vdash \{X_1\}_{\leq} \ C_1 \ \{[y_1 \in \{-1, \dots, y\}]\},$$
$$R \vdash \{X_2\}_{\leq} \ C_2 \ \{[y_2 \in \{-1, \dots, y\}]\},$$
$$R \vdash \{X_3\}_{\leq} \ C_3 \ \{[y_3 = -1] \cdot [l = y - y_3 + 1]\}$$

and additionally the four proof obligations

$$\begin{gathered} \text{Write}(C_1) \cap (\text{Vars}\,(C_2 \parallel C_3) \\ \cup \text{Vars}\,([y_2 \in \{-1, \dots, y\}] \star ([y_3 = -1] \cdot [l = y - y_3]), R)) = \emptyset, \\ \text{Write}(C_2 \parallel C_3) \cap \text{Vars}\,(C_1, [y_1 \in \{-1, \dots, y\}], R) = \emptyset, \\ \text{Write}(C_2) \cap \text{Vars}\,(C_3, [[y_3 = -1] \cdot [l = y - y_3]], R) = \emptyset, \\ \text{Write}(C_3) \cap \text{Vars}\,(C_2, [y_2 \in \{-1, \dots, y\}], R) = \emptyset. \end{gathered}$$

The last four proof obligations are straightforward to verify using various trivial approximations for the free variables in the random variables. The first three proof obligations we will further investigate in the following subsections.

The first proof obligation will require us to prove that any change to the shared memory performed by the producer adheres to the resource invariant. For the second proof obligation, we have to prove that the resource invariant is only maintained in the channel with a probability of $\langle p^{y_2+1} \rangle$. Finally, for the third proof obligation, we will prove that if the resource invariant is adhered, for l to be y after the consumer terminates, l needs to be $y - y_3$ before the consumer runs.

Producer

We restate program C_1, the producer

```
while(y₁ ≥ 0){
   pif (0.5) {x₁ := 1} else {x₁ := 2};
   <z₁ + y₁> := x₁;
   y₁ := y₁ − 1
}
```

and the specification to prove

$$R \vdash \{[y_1 \in \{-1, \dots, y\}]\}_{\leq} \ C_1 \ \{[y_1 \in \{-1, \dots, y\}]\}.$$

For this we will prove both that the resource invariant is always adhered to and that y_1 remains in the interval $\{-1, \dots, y\}$. The following annotated program

illustrates the idea behind the following proof:

$$
\begin{array}{l}
/\!\!/\!_{\le}\ [y_1 \in \{-1, \dots, y\}] \\
\texttt{while}(0 \le y_1)\{ \\
\quad /\!\!/\!_{\le}\ [y_1 \in \{0, \dots, y\}] \\
\quad \texttt{pif}\ (0.5)\ \{x_1 := 1\}\ \texttt{else}\ \{x_1 := 2\}; \\
\quad /\!\!/\!_{\le}\ ([x_1 = 1] \sqcup [x_1 = 2]) \cdot [y_1 \in \{0, \dots, y\}] \\
\quad <z_1 + y_1> := x_1; \\
\quad /\!\!/\!_{\le}\ [y_1 \in \{0, \dots, y\}] \\
\quad y_1 := y_1 - 1 \\
\quad /\!\!/\!_{\le}\ [y_1 \in \{-1, \dots, y\}] \\
\} \\
/\!\!/\!_{\le}\ [y_1 \in \{-1, \dots, y\}]
\end{array}
$$

Loop Applying the while proof rule, we are required to provide the random variable $X = [y_1 \in \{0, \dots, y\}]$ and need to prove that

Remark that we assume here that y_1 is always natural at the start of the loop.

$$
[y_1 \in \{-1, \dots, y\}] \models [y_1 \geq 0] \cdot X \sqcup \sim [y_1 \geq 0] \cdot [y_1 \in \{-1, \dots, y\}],
$$
$$
R \vdash \{X\}_{\le}\ C_1^{body}\ \{[y_1 \in \{-1, \dots, y\}]\}
$$

where C_1^{body} is the loop body of C_1. The first obligation holds trivially.

For the remaining obligation, we can apply the seq proof rule to gain the new proof obligations

$$
R \vdash \{X\}_{\le}\ \begin{array}{c}\texttt{pif}\ (0.5)\ \{x_1 := 1\} \\ \texttt{else}\{x_1 := 2\}\end{array}\ \{([x_1 = 1] \sqcup [x_1 = 2]) \cdot X\}
$$
$$
R \vdash \{([x_1 = 1] \sqcup [x_1 = 2]) \cdot X\}_{\le}\ <z_1 + y_1> := x_1\ \{X\}
$$
$$
R \vdash \{X\}_{\le}\ y_1 := y_1 - 1\ \{[y_1 \in \{-1, \dots, y\}]\}
$$

Random Choice For the first proof obligation, we apply monotonicity (and proving the trivial entailment $X \models \langle 0.5 \rangle \cdot X + \sim\langle 0.5 \rangle \cdot X$) to obtain the proof obligations

$$
R \vdash \left\{ \begin{array}{c} \langle 0.5 \rangle \cdot X \\ + \sim\langle 0.5 \rangle \cdot X \end{array} \right\}_{\le}\ \begin{array}{c}\texttt{pif}\ (0.5)\ \{x_1 := 1\} \\ \texttt{else}\{x_1 := 2\}\end{array}\ \{([x_1 = 1] \sqcup [x_1 = 2]) \cdot X\}.
$$

Now we apply the proof rule pif to obtain the two proof obligations

$$
R \vdash \{X\}_{\le}\ x_1 := 1\ \{([x_1 = 1] \sqcup [x_1 = 2]) \star X\},
$$
$$
R \vdash \{X\}_{\le}\ x_1 := 2\ \{[x_1 = 1] \sqcup [x_1 = 2] \star X\},
$$

which we handle by applying monotonicity and proving the simple entailments $X \models (([x_1 = 1] \sqcup [x_1 = 2]) \star X)[x_1 := a]$ with $a \in \{1, 2\}$, and finish the proof obligations with the assign proof rule.

Mutation We prove the second proof obligation

$$R \vdash \{([x_1 = 1] \sqcup [x_1 = 2]) \star X\}_{\leq} \texttt{<}z_1 + y_1\texttt{>} := x_1 \{X\}.$$

For this we require access to the resource invariant. Since the mutation is atomic, we can apply the atom proof rule to gain access to the resource invariant and thus have to prove that

$$[\mathsf{emp}] \vdash \{([x_1 = 1] \sqcup [x_1 = 2]) \star X \star R\}_{\leq} \texttt{<}z_1 + y_1\texttt{>} := x_1 \{X \star R\}.$$

To prove this, we apply monotonicity and the mut proof rule and are thus left to prove the entailment

$$\begin{aligned} &([x_1 = 1] \sqcup [x_1 = 2]) \star X \star R \\ \models\ &(\mathsf{S}a.\ [z_1 + y_1 \mapsto a]) \star ([z_1 + y_1 \mapsto x_1] \mathrel{-\!\!\circledast} X \star R) \end{aligned}$$

Since we have $y_1 \in \{0, \ldots, y\}$, we can extract the $y_1{}^{\text{th}}$ element from the first array in R to obtain

$$\begin{aligned} &([x_1 = 1] \sqcup [x_1 = 2]) \star X \\ &\star ([z_1 + y_1 \mapsto 0] \sqcup [z_1 + y_1 \mapsto 1] \sqcup [z_1 + y_1 \mapsto 2]) \\ &\star \mathop{\bigstar}_{i=0}^{y} ([i \neq y_1] \sqcap ([z_1 + i \mapsto 0] \sqcup [z_1 + i \mapsto 1] \sqcup [z_1 + i \mapsto 2]) \\ &\qquad \sqcup [i = y_1] \sqcap [\mathsf{emp}]) \\ &\star \mathop{\bigstar}_{i=0}^{y} ([z_2 + i \mapsto 0] \sqcup [z_2 + i \mapsto 1] \sqcup [z_2 + i \mapsto 2]) \\ \models\ &(\mathsf{S}a.\ [z_1 + y_1 \mapsto a]) \star ([z_1 + y_1 \mapsto x_1] \mathrel{-\!\!\circledast} X \star R). \end{aligned}$$

We use monotonicity of separating multiplication to split away the points to at location $z_1 + y_1$ to obtain the entailments

$$[z_1 + y_1 \mapsto 0] \sqcup [z_1 + y_1 \mapsto 1] \sqcup [z_1 + y_1 \mapsto 2] \models \mathsf{S}a.\ [z_1 + y_1 \mapsto a],$$

$$\begin{aligned} &([x_1 = 1] \sqcup [x_1 = 2]) \star X \\ &\star \mathop{\bigstar}_{i=0}^{y} ([i \neq y_1] \sqcap ([z_1 + i \mapsto 0] \sqcup [z_1 + i \mapsto 1] \sqcup [z_1 + i \mapsto 2]) \\ &\qquad \sqcup [i = y_1] \sqcap [\mathsf{emp}]) \\ &\star \mathop{\bigstar}_{i=0}^{y} ([z_2 + i \mapsto 0] \sqcup [z_2 + i \mapsto 1] \sqcup [z_2 + i \mapsto 2]) \\ \models\ &[z_1 + y_1 \mapsto x_1] \mathrel{-\!\!\circledast} X \star R. \end{aligned}$$

The first entailment is proven using a case distinction on the maximum on the left and matching the correct a for the supremum on the right. For the second entailment, we apply adjointness of separating multiplication and the truncated magic wand to obtain

$$-\text{"}- \star [z_1 + y_1 \mapsto x_1] \models X \star R.$$

We can upper bound the expression $([x_1 = 1] \sqcup [x_1 = 2]) \star [z_1 + y_1 \mapsto x_1]$ by the expression $[z_1 + y_1 \mapsto 0] \sqcup [z_1 + y_1 \mapsto 1] \sqcup [z_1 + y_1 \mapsto 2]$, thus obtaining

$$\begin{aligned} & X \\ & \star ([z_1 + y_1 \mapsto 0] \sqcup [z_1 + y_1 \mapsto 1] \sqcup [z_1 + y_1 \mapsto 2]) \\ & \star \mathop{\bigstar}_{i=0}^{y} ([i \neq y_1] \sqcap ([z_1 + i \mapsto 0] \sqcup [z_1 + i \mapsto 1] \sqcup [z_1 + i \mapsto 2]) \\ & \qquad \sqcup [i = y_1] \sqcap [\mathsf{emp}]) \\ & \star \mathop{\bigstar}_{i=0}^{y} ([z_2 + i \mapsto 0] \sqcup [z_2 + i \mapsto 1] \sqcup [z_2 + i \mapsto 2]) \\ \models {} & X \star R. \end{aligned}$$

Finally, recombining the big separating multiplication we obtain

$$\begin{aligned} & X \\ & \star \mathop{\bigstar}_{i=0}^{y} ([z_1 + i \mapsto 0] \sqcup [z_1 + i \mapsto 1] \sqcup [z_1 + i \mapsto 2]) \\ & \star \mathop{\bigstar}_{i=0}^{y} ([z_2 + i \mapsto 0] \sqcup [z_2 + i \mapsto 1] \sqcup [z_2 + i \mapsto 2]) \\ \models {} & X \star R, \end{aligned}$$

which holds by reflexivity.

Decrementation of Index We prove the third proof obligation

$$R \vdash \{X\}_{\leq}\ y_1 := y_1 - 1\ \{[y_1 \in \{-1, \ldots, y\}]\}$$

by applying monotonicity, proving the easy to verify entailment

$$[y_1 \in \{0, \ldots, y\}] \models [y_1 \in \{-1, \ldots, y\}]\,[y_1 := y_1 - 1]$$

and applying the assign proof rule.

Lossy Channel

We restate program C_2, the lossy channel

```
while(0 ≤ y2){
  x2 := <z1 + y2>;
  if (x2 ≠ 0) {
    pif (p) {<z2 + y2> := x2} else {<z2 + y2> := −1}
    y2 := y2 − 1
  }
}
```

and the specification to prove

$$R \vdash \left\{ \begin{array}{c} [y_2 \in \{0, \ldots, y\}] \cdot \langle p^{y_2+1} \rangle \\ \sqcup [y_2 < 0] \cdot [y_2 \in \{-1, \ldots, y\}] \end{array} \right\}_{\leq} C_2\ \{[y_2 \in \{-1, \ldots, y\}]\}.$$

The following proof is illustrated as this annotated program:

```
⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩ ⊔ [y₂ < 0] · [y₂ ∈ {−1,…,y}]
while(0 ≤ y₂){
  ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩
  x₂ := <z₁ + y₂>;
  ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩ · [x₂ ∈ {0,1,2}]
  if (x₂ ≠ 0) {
    ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩ · [x₂ ∈ {1,2}]
    pif (p) {<z₂ + y₂> := x₂} else {<z₂ + y₂> := −1}
    ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂)⟩
    y₂ := y₂ − 1
    ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩ ⊔ [y₂ < 0] · [y₂ ∈ {−1,…,y}]
  }
  ⫽≤ [y₂ ∈ {0,…,y}] · ⟨p^(y₂+1)⟩ ⊔ [y₂ < 0] · [y₂ ∈ {−1,…,y}]
}
⫽≤ [y₂ ∈ {−1,…,y}]
```

Loop First we apply the while proof rule and thus have to provide $X = [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1} \rangle$. We obtain the proof obligations

$$\begin{array}{c} [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \\ \sqcup [y_2 < 0] \cdot [y_2 \in \{-1,\dots,y\}] \end{array} \models \begin{array}{c} [0 \le y_2] \cdot [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \\ \sqcup \sim [0 \le y_2] \cdot [y_2 \in \{-1,\dots,y\}] \end{array}$$

$$R \vdash \left\{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle\right\}_{\le}\ C_2^{body} \left\{ \begin{array}{c} [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \\ \sqcup [y_2 < 0] \cdot [y_2 \in \{-1,\dots,y\}] \end{array} \right\}.$$

The entailment is straightforward to prove. For the second one, we apply the seq proof rule and obtain the new proof obligations

$$R \vdash \{-\text{"}-\}_{\le}\ x_2 := \texttt{<}z_1 + y_2\texttt{>} \left\{ \begin{array}{c} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{0,1,2\}] \end{array} \right\}$$

$$R \vdash \left\{ \begin{array}{c} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{0,1,2\}] \end{array} \right\}_{\le} \texttt{if}\ (x_2 \ne 0)\ \{\dots\}\ \{-\text{"}-\}.$$

Lookup For the first specification, we apply the atom proof rule, monotonicity proof rule and the lookup proof rule and need to verify the entailment

$$\begin{aligned} &([y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle) \star R \\ \models\ & \mathfrak{S}a.\ [z_1 + y_2 \mapsto a] \star \big([z_1 + y_2 \mapsto a] \\ & \quad -\!\!\circledast \big(([y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \cdot [x_2 \in \{0,1,2\}]) \star R\big)[x_2 := a] \big). \end{aligned}$$

We prove this by first extracting $[z_1 + y_2 \mapsto 0] \sqcup [z_1 + y_2 \mapsto 1] \sqcup [z_1 + y_2 \mapsto 2]$ from R on the left-hand side, which we can do as $y_2 \in \{0,\dots,y\}$. Next we distinguish between each case of the value at the location $z_1 + y_2$. For each case, we eliminate the supremum by matching the case. Then we apply monotonicity of separating multiplication and match $[z_1 + y_2 \mapsto a]$. Lastly we apply adjointness to regain

the location $z_1 + y_2$ and re-add it to R. Finally, $[x_2 \in \{0,1,2\}]\,[x_2 := a]$ simplifies and both sides are equal.

Conditional Branching For the conditional branching, we apply monotonicity and the if proof rule (and the skip proof rule for the empty else branch) to obtain the to be proven entailment

$$\begin{aligned} & [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \cdot [x_2 \in \{0,1,2\}] \\ \models \quad & [x_2 \neq 0] \cdot [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \cdot [x_2 \in \{1,2\}] \\ & \sqcup \sim [x_2 \neq 0] \cdot \big([y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \sqcup [y_2 < 0] \cdot [y_2 \in \{-1,\dots,y\}]\big) \end{aligned}$$

and the to be proven specification

$$R \vdash \left\{ \begin{matrix} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{1,2\}] \end{matrix} \right\}_{\leq} \texttt{pif}\ (p)\ \{\dots\};\ \dots\ \{-"-\}.$$

The entailment is straightforward to prove by distinction on the value of x_2. For the specification, we apply the seq proof rule and need to prove that

$$R \vdash \left\{ \begin{matrix} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{1,2\}] \end{matrix} \right\}_{\leq} \texttt{pif}\ (p)\ \{\dots\}\ \{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle\}$$

$$R \vdash \{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle\}_{\leq}\ y_2 := y_2 - 1 \left\{ \begin{matrix} [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2+1}\rangle \\ \sqcup [y_2 < 0] \cdot [y_2 \in \{-1,\dots,y\}] \end{matrix} \right\}.$$

Probabilistic Branching We prove the specification

$$R \vdash \left\{ \begin{matrix} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{1,2\}] \end{matrix} \right\}_{\leq} \texttt{pif}\ (p)\ \{\dots\}\ \{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle\}$$

by applying monotonicity and the pif proof rule. For this we prove the entailment

$$\begin{matrix} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2+1}\rangle \cdot [x_2 \in \{1,2\}] \end{matrix} \models \begin{matrix} \langle p\rangle \cdot [y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle \cdot [x_2 \in \{1,2\}] \\ + \sim\langle p\rangle \cdot [\mathsf{false}] \end{matrix},$$

which is straightforward to prove and thus are left with

$$R \vdash \left\{ \begin{matrix} [y_2 \in \{0,\dots,y\}] \\ \cdot\langle p^{y_2}\rangle \cdot [x_2 \in \{1,2\}] \end{matrix} \right\}_{\leq} \texttt{<}z_2 + y_2\texttt{>} := x_2\ \{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle\}$$

$$R \vdash \{[\mathsf{false}]\}_{\leq}\ \texttt{<}z_2 + y_2\texttt{>} := -1\ \{[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle\}.$$

The first specification is analogous to the mutation in the producer thread and the second specification is trivial as there is [false] on the left.

Decrementation of Index For the decrementation of the index, we apply monotonicity and the assign proof rule and thus need to prove the entailment

$$[y_2 \in \{0,\dots,y\}] \cdot \langle p^{y_2}\rangle \models \begin{matrix} [y_2 - 1 \in \{0,\dots,y\}] \cdot \langle p^{y_2-1+1}\rangle \\ \sqcup [y_2 - 1 < 0] \cdot [y_2 - 1 \in \{-1,\dots,y\}] \end{matrix}$$

By simplifying, we obtain

$$[y_2 \in \{0, \dots, y\}] \cdot \langle p^{y_2} \rangle \models \begin{array}{c} [y_2 \in \{1, \dots, y+1\}] \cdot \langle p^{y_2} \rangle \\ \sqcup [y_2 < 1] \cdot [y_2 \in \{0, \dots, y+1\}] \end{array}.$$

When $y_2 = 0$, we have that $p^{y_2} = p^0 = 1$, thus the right-hand side of the maximum matches. Otherwise the left-hand side of the maximum matches.

Consumer

We restate program C_3, the consumer

```
while(0 ≤ y3){
  x3 := <z2 + y3>;
  if (x3 ≠ 0) {
    if (x3 ≠ −1) {l := l + 1};
    y3 := y3 − 1
  }
}
```

and the specification to prove

$$R \vdash \{[y_3 \in \{-1, \dots, y\}] \cdot [l = y - y_3]\}_{\leq}\ C_3 \left\{ \begin{array}{c} [y_3 = -1] \\ \cdot [l = y - y_3] \end{array} \right\}.$$

This specification requires us to prove that the program always adheres to the resource invariant and that when the resource invariant is adhered to, that the variable l has value $y - y_3$ before the loop is executed. This annotated program illustrates the following proof:

```
//≤ [y3 ∈ {−1, … , y}] · [l = y − y3]
while(0 ≤ y3){
  //≤ [y3 ∈ {0, … , y}] · [l = y − y3]
  x3 := <z2 + y3>;
    //≤ [y3 ∈ {0, … , y}] · [l = y − y3] · [x3 ∈ {0, 1, 2}]
  if (x3 ≠ 0) {
    //≤ [y3 ∈ {0, … , y}] · [l = y − y3] · [x3 ∈ {1, 2}]
    if (x3 ≠ −1) {l := l + 1};
    //≤ [y3 ∈ {0, … , y}] · [l = y − y3 + 1]
    y3 := y3 − 1
    //≤ [y3 ∈ {−1, … , y}] · [l = y − y3]
  }
  //≤ [y3 ∈ {−1, … , y}] · [l = y − y3]
}
//≤ [y3 = −1] · [l = y − y3]
```

Loop We prove the the specification by first applying the while proof rule and thus have to prove that

$$\begin{aligned} & [y_3 \in \{-1, \dots, y\}] \cdot [l = y - y_3] \\ \vDash \quad & [0 \le y_3] \cdot [y_3 \in \{0, \dots, y\}] \cdot [l = y - y_3] \\ & \sqcup \sim [0 \le y_3] \cdot [y_3 = -1] \cdot [l = y - y_3], \end{aligned}$$

which follows immediately, and that

$$R \vdash \{[y_3 \in \{0, \dots, y\}] \cdot [l = y - y_3]\}_{\le} \; C_3^{body} \; \{[y_3 \in \{-1, \dots, y\}] \cdot [l = y - y_3]\}.$$

We apply the seq proof rule to split the loop body into the two proof obligations

$$R \vdash \{-"-\}_{\le} \; \texttt{x}_3 \; \texttt{:=} \; \texttt{<}z_2 + y_3\texttt{>} \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \cdot [l = y - y_3] \\ \cdot [x_3 \in \{0, 1, 2\}] \end{array} \right\}$$

$$R \vdash \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l = y - y_3] \cdot [x_3 \in \{0, 1, 2\}] \end{array} \right\}_{\le} \; \texttt{if (} x_3 \ne 0 \texttt{) \{...\}} \; \{-"-\}.$$

The first specification is proven similarly to the lookup for the channel thread.

Conditional Branching For the other specification, we apply monotonicity to get the left-hand side into the right form and apply the if proof rule to obtain a new proof obligation. Thus together, we obtain the to be proven entailment

$$\begin{aligned} & [y_3 \in \{0, \dots, y\}] \cdot [l = y_3 - y] \cdot [x_3 \in \{0, 1, 2\}] \\ \vDash \quad & [x_3 \ne 0] \cdot [y_3 \in \{0, \dots, y\}] \cdot [l = y - y_3] \cdot [x_3 \in \{1, 2\}] \\ & \sqcup \sim [x_3 \ne 0] \cdot [l = y - y_3] \cdot [y_3 \in \{-1, \dots, y\}] \end{aligned}$$

and the to be proven specification

$$R \vdash \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l = y - y_3] \cdot [x_3 \in \{1, 2\}] \end{array} \right\}_{\le} \; \texttt{if (} x_3 \ne -1 \texttt{) \{...\}; ...} \; \{-"-\}.$$

Note that the else branch where $y_3 = -1$ can not be reached under our resource invariant. Thus we can associate this branch with [false] and thus have to prove

We also already adjust the information about l for the next statement.

$$\begin{aligned} & [y_3 \in \{0, \dots, y\}] \cdot [l = y - y_3] \cdot [x_3 \in \{1, 2\}] \\ \vDash \quad & [y_3 \ne -1] \cdot [y_3 \in \{0, \dots, y\}] \cdot [l + 1 = y - y_3 + 1] \\ & \sqcup \sim [y_3 \ne -1] \cdot [\mathsf{false}], \end{aligned}$$

$$R \vdash \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l + 1 = y - y_3 + 1] \end{array} \right\}_{\le} \; \texttt{l := l + 1} \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l = y - y_3 + 1] \end{array} \right\},$$

$$R \vdash \{[\mathsf{false}]\}_{\le} \; \texttt{skip} \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l = y - y_3 + 1] \end{array} \right\},$$

$$R \vdash \left\{ \begin{array}{c} [y_3 \in \{0, \dots, y\}] \\ \cdot [l = y - y_3 + 1] \end{array} \right\}_{\le} \; \texttt{y}_3 \; \texttt{:=} \; \texttt{y}_3 - 1 \left\{ \begin{array}{c} [y_3 \in \{-1, \dots, y\}] \\ \cdot [l = y - y_3] \end{array} \right\},$$

where all four obligations are now proven similarly to previous proofs.

7. First Epilogue

Separation Logics for Concurrent Programs The work presented in Chapter 3 is based on [25, 26], which presented the first concurrent separation logics able to use resource invariants in order to reason about interaction between concurrent threads. They use separation logic [23, 24] to describe separation of heap in the presence of multiple threads that may use local memory or shared memory. The semantics presented in Chapter 3 differ from [26] by not embedding framing in the **wrlp** function. Instead we require that the operational semantics of the program enables framing.

The logic and tool VerCors [56] allows annotating programs with preconditions, postconditions and resource invariants to allow automatic verification of program specifications in a concurrent separation logic. It does so by generating verification conditions in the sequential intermediate language of the separation logic verifier Viper [57].

A major step towards more expressive logics was the introduction of abstract predicates [58], which allowed embedding resource invariants in the pre- and postconditions instead of carrying them separately in the specification. Opening a resource invariant is allowed by using a *repartitioning operator*. This work ultimately sparked the creation of new logics for program verification. To allow for a wider range of atomicity, TaDa [59] introduces a notion of non-atomic programs for which we may open the resource invariant as well. FCSL [60] builds upon the idea of having abstract predicates to reason about resource invariants, but extends predicates to *state transition systems* to model the possible changes in a concurrent system. Iris [41] lifts the idea of abstract predicates to allow arbitrary resources, whose definition can again include Iris propositions to allow for resource invariants as resources. However, this means that the definition of the Iris proposition type is recursive. For this to be well-defined, Iris defines the type of its propositions as a fixed point, which is guaranteed to exist. Iris and its extension have been implemented in the Rocq theorem prover [61].

Opening a resource invariant correspond to applying the atom proof rule.

The Rocq theorem prover was previously known as Coq.

Separation Logics for Probabilistic Programs In order to support probabilistic programs, various attempts using separation logic are found in literature. [62, 63] use separation to encode *independence* of random variables. Two random variables are independent if their joint cumulative distribution functions is the product of the individual cumulative distribution functions. This allows — similar to heap and resource based separation logic — a prover to apply a frame rule on random variables.

The expressiveness of the Iris logic allows the development of various logics on top of the base logic to reason about probabilistic programs. Polaris [64] allows a prover to prove *relations between probabilistic concurrent programs* by using (synchronous) coupling between samplings in the program. This way, we can express or approximate the expected value for some random variable with respect to a complex probabilistic program in terms of the expected value of the same random variable with respect to a simple probabilistic program. Computing the expected value of the random variable in the simple program is then assumed to be easy. The logic Clutch [65] extends this idea to allow *asynchronous* couplings between related probabilistic non-concurrent programs.

To achieve this, they use *presampling tapes* which allows sampling a value at an earlier position in the program in order to synchronise the sampling between the programs. Error credits in the Iris-based logic Eris [66] were introduced to reason about the *probability to abort* in probabilistic non-concurrent programs. This can be employed to also reason about the probability to fail a certain specification and thus also to reason about lower bounds to satisfy a certain specification. Tachis [67] further extends the logic Eris and allows for more general notions of credits. Separately from that, ExpIris [68] was developed to reason about expected credits on probabilistic concurrent programs.

Outcome logic [69] is a logic that allows the prover to reason about outcomes and to split outcomes by the *outcome conjunction* (similar to the separating conjunction in separation logic). This allows the prover to reason about reachability of incorrect states, only reachability of correct states and also to reason about probabilistic programs. Outcome separation logic [70] extends this by the separating conjunction to also reason about separation of heaps. This combination then also allowed the development of a concurrent outcome logic [71].

Other Semantics for Probabilistic Concurrent Programs There have also been axiomatic and denotational semantics for probabilistic and concurrent programs that are not based on separation logic. The rely-guarantee calculus has been extended to probabilistic programs, which allows a prover to use rely-guarantee reasoning for probabilistic concurrent programs [72]. A semantics for probabilistic and concurrent programs [73] was developed using an equivalence class of directed acyclic graphs that model the execution of the program, which they call pomsets. By enriching the pomsets with formulae, the authors can differentiate between conditional branching and probabilistic branching. Concurrent execution can be resolved using a linearization, which then allows us to gain a monadic view on their semantics.

Weakest Expectation The function we call *weakest expectation* in Chapters 4 and 6 is usually called in literature *weakest pre-expectation* [29], which was first introduced in [27, 28]. This was later extended by using the quantitative separation logic as introduced in Chapter 4 by [30]. Quantitative separation logic extends classical separation logic by mapping states not to Boolean values but to non-negative reals. By further introducing operations for multiplication, addition and especially separating multiplication, the authors are able to construct denotational semantics for probabilistic and heap-manipulating programs. A different separation logic aimed to reason about *expected (amortized) runtimes* (instead of general expected values) is [74]. There the authors use a *separating addition*, which allows the definition of a runtime calculus for heap-manipulating programs and for amortized runtimes. The axiomatic semantics as presented in Chapter 6 allow a prover to reason about lower bounds on the probability to reach a certain set of final states in probabilistic and concurrent programs by using resource invariants to reason about shared memory [33].

Concurrent Fuzzy Separation Logic

We combine ideas from concurrent separation logic [25, 26] and quantitative separation logic [30] to develop concurrent fuzzy separation logic [33]. We use the idea of introducing resource invariants that are invariant over the span of the

program and combine it with the idea to generalize Boolean separation logic to a logic mapping to non-negative reals. In order to avoid the problem of assigning infinity to non-terminating runs, we use a variant which we coin fuzzy separation logic that maps states to reals in the interval between zero and one. This allows us to reason about lower bounds of probabilities to reach a certain set of final states or to not terminate. We develop semantics in terms of fuzzy separation logic called the weakest resource-safe liberal expectation and provide proof rules to reason about specifications in these semantics.

By staying close to the inspiration of [25, 26], we gain conservativity towards concurrent separation logic. This means that any specification stated in concurrent separation logic can also be stated in concurrent fuzzy separation logic. Unfortunately we lost access to the minimum rule, which corresponds to the conjunction rule in concurrent separation logic. This rule allows eliminating minima in the specification. Since **wrle** is defined using a sum and sums behave ill with minima, we do not have a rule for minima.

We state the adequacy of our weakest liberal expectation semantics in terms of path semantics. That is, we sum the random variable assigned to the final state over all paths weighted by the probability of that path and add additionally the probability to not terminate. An equality between weakest liberal expectation and this notion is established.

Future Work Weakest liberal resource-safe expectations are linear in case the resource invariant is empty and the program is almost surely terminating [33]. This can be easily derived from our adequacy theorem. However, we conjecture that **wrle** is also linear when the resource invariant is *strictly-exact*. A random variable is strictly-exact if for any stack there is at most one heap for which the random variable is not zero. When the left-hand side of a quantitative or fuzzy magic wand is strictly-exact, the addition on the right-hand side of the magic wand distributes. This may allow developing a similar adequacy theorem for strictly-exact resource invariants and allow proving linearity for almost surely terminating programs.

We have hinted towards a liberal expectation semantics on the non-negative reals. This would mean in consequence that runs that are not terminating are weighted with infinity. Loops that allow infinite runs — even if the runs have probability zero — may then evaluate to infinity for any random variable. However in liberal semantics, it is easier to reason about about lower bounds of the expected value of random variables. Whenever the program is terminating (for at least one scheduler), the value of a liberal expectation semantics corresponds to the value of a non-liberal expectation semantics. Moreover, liberal semantics over non-negative reals may give access to a concurrency rule for non-negative random variables. Some ground work towards weakest liberal expectations on the non-negative reals for probabilistic programs have been laid out in [75].

The probabilistic program

$$\texttt{pif}\;(0.5)\;\{\texttt{<}r\texttt{>} := x\}\;\texttt{else}\;\{\texttt{<}r\texttt{>} := y\}$$

is not atomic. Nevertheless, its effect is atomic in nature. There is no difference between another thread interfering with this thread before the probabilistic branching and after (and thus before the mutation). As such, we would assume

that it is also sound to open the resource invariant for such a program. Formulating a notion of atomicity for non-atomic programs may help to improve modularity of the **wrle** semantics.

It is known that the current information present in the state may not be enough to reason about the possible final states. For example, it is difficult to prove that for the program C as

$$\begin{array}{c} <r> := 0; \\ \begin{array}{r|l} x := <r>; & x := <r>; \\ <r> := x + 2 & <r> := x + 2 \end{array} \end{array}$$

the final state $[r \mapsto 0]$ is not reachable. That is, the specification

$$[\mathsf{emp}] \vdash \{\mathfrak{S} a.\ [r \mapsto a]\}_{\leq}\ C\ \{[r \mapsto 2] \sqcup [r \mapsto 4]\}$$

is difficult to prove. In qualitative separation logic, ghost states [76] or ghost code [77] help to gain the necessary additional knowledge to verify such specifications. Developing ghost states or ghost code for weakest expectation reasoning styles is therefore desirable.

Proof Assistant Support

We formalized parts of Chapter 6 in the proof assistant Lean [6]. This especially includes a definition of the weakest resource-safe liberal expectation, which is defined as a fixed point on the characteristic function transitioning one step in the underlying Markov decision process while guaranteeing the validity of the resource invariant. We furthermore formalized the presented proof rules and their soundness proofs to allow reusing the proof rules when using Lean to verify probabilistic programs in our given programming language.

Similar verified frameworks have been developed prior. A verified framework in Isabelle HOL [78] integrates probabilistic computation into the underlying functional programming language of Isabelle such that reasoning about distributions computed by a probabilistic program can be done directly in Isabelle HOL [79]. An approach in the Rocq theorem prover to verify probabilistic programs using an interpretation of probabilistic programs as monadic transformers using maps from states to the unit interval has been presented in [80]. Further verified proof system in the Rocq theorem prover for probabilistic programs include the previous mentioned extensions of Iris [64–68].

There has also been prior work done to provide a verified proof system for expectation based approaches. For probabilistic programs that are neither concurrent nor heap manipulating, [75, 81] provide formalizations of weakest expectation semantics in Isabelle HOL. A formalization of quantitative separation logic and its weakest expectation as presented in Chapter 4 has been formalized in [82]. All of these, including this dissertation, lack a formalization of the adequacy theorem. That is, none of these prove an equivalence between the expected reward on a Markov model and the weakest expectation.

Future Work Formalizing Theorem 6.2.2, which guarantees us the adequacy of **wrle**, has multiple advantages. A formalization gives more trust into the proof, which is prominently error prone [49]. Additionally, we can use it to formalize

a proof for the linearity of **wrle** in case of an empty resource invariant and an almost surely terminating program. It further allows to derive new conclusions from specification for a user of the framework.

In the current formalization, no tactics are provided to ease the use of the proof system. That makes using it a hurdle. Tactics that apply various trivial theorems at once or prove easy entailments automatically may help to ease the use of the proof system.

Entailments in Fuzzy and Quantitative Separation Logic

8. Transforming Fuzzy Separation Logic into Qualitative Separation Logic

Assertions about quantitative properties introduce new complexities in the reasoning process. For that reason, it is beneficial to split quantitative and qualitative reasoning in two separate proof obligations instead of reasoning about them simultaneously. That is, *our goal is to decompose entailments in fuzzy separation logic into entailments in real arithmetic and into entailments in qualitative separation logic.*

Let us for example consider the entailment

$$\langle 0.3\rangle \star [x \mapsto y] \dotplus \langle 0.4\rangle \star [y \mapsto x] \models \langle 1\rangle \star [x \mapsto y] \sqcup \langle 1\rangle \star [y \mapsto x]\,.$$

Ad hoc we observe that the left-hand side has the three possible evaluations $0, 0.3, 0.4$ and 0.7. The right-hand side has the evaluations 0 and 1. For the entailment to hold, we need to guarantee that the evaluation of a left-hand side is never greater than the evaluation of the right-hand side. Instead of checking that for these evaluations the entailment does not hold, we can also check that certain entailments need to hold in qualitative separation logic. For this, we transform a random variable X into the proposition $a \leq X(s, h, \eta)$, where a is a possible evaluation. The entailments thus become qualitative by this transformation. It turns out, that we can transform the expression $a \leq X(s, h, \eta)$ for certain fuzzy separation logic random variables X into equivalent qualitative separation logic propositions, such that we can leverage techniques to solve qualitative separation logic entailments so as to solve fuzzy separation logic entailments. The idea of this transformation is to take those parts of the formula that translate into evaluations greater than the target value and connect them in a disjunction.

An evaluation here is an element of the image.

We give more details about the soundness of this approach in the next section.

Let us consider this more concretely for our example. We consider all four evaluations.

- For the evaluation 0, we can form the qualitative separation logic proposition
  $$0 \leq ([x \mapsto y] \sqcup [y \mapsto x])(s, h, \eta) \iff (s, h) \models_\eta \mathsf{true}.$$
  Since the right-hand side of the qualitative entailment is true, the resulting entailment vacuously holds.
- For the evaluation 0.3, we need to transform both sides into formulae which hold when their evaluation is greater than 0.3. For the left, we have that this is the case when
  $$\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x] \vee \mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]),$$
  as these yield either $0.3, 0.4$ or 0.7. For the right, we have that the only evaluation that is at least 0.3 is 1, thus the formula
  $$\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x]$$
  is equivalent to the right term. It is now easy to verify that the qualitative entailment
  $$\begin{aligned} &\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x] \vee \mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]) \\ \models\ &\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x] \end{aligned}$$

actually holds.

- For the evaluation 0.4, the left-hand side transforms similarly to the proposition

  $$\mathsf{true} * [y \mapsto x] \vee \mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]).$$

  Not every state satisfying $\mathsf{true} * [x \mapsto y]$ would yield a weight at least 0.4, thus it is left out of the outermost disjunction. The right-hand side again transforms into

  $$\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x]\,.$$

  The resulting qualitative entailment is

  $$\mathsf{true} * [y \mapsto x] \vee \mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]) \models \mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x]\,,$$

  which also holds.

- The evaluation 0.7 is only reached when both sides of the addition are satisfied. Thus we have for the left-hand side the transformed proposition

  $$\mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]).$$

  The right-hand side again transforms into

  $$\mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x]\,,$$

  and we obtain the qualitative entailment

  $$\mathsf{true} * ([x \mapsto y] \wedge [y \mapsto x]) \models \mathsf{true} * [x \mapsto y] \vee \mathsf{true} * [y \mapsto x]\,.$$

  This entailment also holds.

Since we checked all entailments for each possible evaluations of the left-hand side, we know that the fuzzy entailment indeed holds.

8.1. The At-Least Predicate

Given a fuzzy entailment $X \models Y$. To transform this into a qualitative entailment $\Phi \models \Psi$, we first need to transform X and Y into qualitative statements. The idea to do this is by considering all possible values of the domain of X. Indeed, we have that $a \leq b$ if and only if $\forall c.\ c \leq a \Rightarrow c \leq b$. We will exploit this equivalence to construct a transformation from fuzzy into qualitative entailments.

Let us investigate this equivalence in more detail. First, we assume that $a \leq b$ and that $c \leq a$. Then by transitivity $c \leq b$. For the other direction, we assume that for all c with $c \leq a$ we have that $c \leq b$. Then we choose c with $c = a$, and so we have that $a \leq b$.

Definition 8.1.1 (At-least Predicate) *Let $X \in \mathsf{RV}^1$ be a random variable in fuzzy separation logic and $a \in [0, 1]$ a real number in the unit interval. We define the* at-least predicate *in qualitative separation logic as*

$$(s, h) \models_\eta a \leq X \iff a \leq X(s, h, \eta).$$

We can already define the transformation from fuzzy separation logic into qualitative separation logic with the previous equivalence. However, there is still one more goal to reach. Say, the image of the random variables is finite — we thus have only finitely many evaluations. Do we then obtain a finite number of entailments? With the previous explained approach, this is not true. However, we can find a slightly different formulation of this theorem to also obtain that a finite domain yields finite entailments.

Definition 8.1.2 (Image of a Function) *Let $X\colon A \to B$ be a function. The image of X is*

$$\mathit{image}(X) = \{ b \in B \mid \exists a \in A.\, X(a) = b \}.$$

Theorem 8.1.1 *Let $X, Y \in \mathsf{RV}^1$ be random variables in fuzzy separation logic and let $\mathit{image}(X) \subseteq A$. We have that*

$$X \models Y \iff \forall a \in A.\, a \leq X \models a \leq Y.$$

Notice that we quantify over a possible *over-approximation* of the image.

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

The proof is very similar to the one not restricted to an over-approximation of the image. The only difference is that we are required to prove that $a \in A$ where $\text{image}(X) \subseteq A$ for the second direction now. However, in our case we have that $a = X(s, h, \eta)$ for some s, h and η, making this obligation trivially true.

Example 8.1.1 We like to verify the fuzzy entailment

$$\langle 0.3 \rangle \star [x \mapsto y] \dotplus \langle 0.4 \rangle \star [y \mapsto x] \models \langle 0.7 \rangle \star [x \mapsto y] \dotplus \langle 0.7 \rangle \star [y \mapsto x],$$

which is a slightly adapted version of the introductory example in this chapter. In order to translate this entailment into a qualitative entailment using Theorem 8.1.1, we first need to compute (an over-approximation of) the image of the left hand-side. We can over-approximate it by considering all points-to random variables as either 0 or 1, obtaining the possible values $0, 0.3, 0.4$ and 0.7. This then yields the entailments

$$\begin{aligned} & 0 \leq \langle 0.3 \rangle \star [x \mapsto y] \dotplus \langle 0.4 \rangle \star [y \mapsto x] \\ \models\ & 0 \leq \langle 0.7 \rangle \star [x \mapsto y] \dotplus \langle 0.7 \rangle \star [y \mapsto x], \end{aligned}$$

$$\begin{aligned} & 0.3 \leq \langle 0.3 \rangle \star [x \mapsto y] \dotplus \langle 0.4 \rangle \star [y \mapsto x] \\ \models\ & 0.3 \leq \langle 0.7 \rangle \star [x \mapsto y] \dotplus \langle 0.7 \rangle \star [y \mapsto x], \end{aligned}$$

$$\begin{aligned} & 0.4 \leq \langle 0.3 \rangle \star [x \mapsto y] \dotplus \langle 0.4 \rangle \star [y \mapsto x] \\ \models\ & 0.4 \leq \langle 0.7 \rangle \star [x \mapsto y] \dotplus \langle 0.7 \rangle \star [y \mapsto x], \end{aligned}$$

$$\begin{aligned} & 0.7 \leq \langle 0.3 \rangle \star [x \mapsto y] \dotplus \langle 0.4 \rangle \star [y \mapsto x] \\ \models\ & 0.7 \leq \langle 0.7 \rangle \star [x \mapsto y] \dotplus \langle 0.7 \rangle \star [y \mapsto x], \end{aligned}$$

which we will have to translate into a form using standard operations from qualitative separation logic.

We can also give explicit rules to compute the over-approximations for a variety of fuzzy separation logic operations. For random variables with infinite image, the image may not include limit points. Suprema and infima will, however, include these limit points. Thus we need to make sure that limit points are included in the over-approximation. This is achieved by the *closure* of that set. The closure includes all these limit points. Whenever a set of reals is finite, the closure of the set is equal to the set itself. For this reason, we do not need the closure of that set whenever the image is finite. Similarly we require the closure for separating multiplication and the truncated magic wand due the underlying supremum and infimum respectively in the definition. If we assume that heaps only have finite allocated locations, we can drop the closure for the separating multiplication as well as then the separating multiplication takes the supremum over finitely many values. This is not the case for the truncated magic wand, as for heap h there may exist infinitely many heaps h' even if we require both to have only finitely many locations.

Topologically speaking, the closure is the smallest superset that is closed.

Definition 8.1.3 *Let $A \subseteq [0,1]$ be a set of reals in the unit interval. The closure of A, denoted as $closure(A)$, is the smallest superset such that*

$$\forall a_0, a_1, \dots \in closure(A).\ \lim_{i \in \mathbb{N}} a_i \in closure(A).$$

Theorem 8.1.2 *Let $X, Y, X', Y' \in \mathsf{RV}^1$ be random variables in fuzzy separation logic, where X' and Y' have finite image. We have that:*

Remark that the image of $[\Phi]$ can be $\{0\}$, $\{1\}$ or $\{0,1\}$, but not $\emptyset$.

$$\begin{aligned}
image([\mathsf{true}]) &= \{1\} \\
image([\mathsf{false}]) &= \{0\} \\
image([\Phi]) &\subseteq \{0,1\} \\
image(\langle e \rangle) &= image(e) \\
image(X[x := e]) &\subseteq image(X) \\
image({\sim} X) &= \{1 - a \mid a \in image(X)\} \\
image(X \sqcap Y) &\subseteq \{\min\{a,\ b\} \mid a \in image(X) \land b \in image(Y)\} \\
image(X \sqcup Y) &\subseteq \{\max\{a,\ b\} \mid a \in image(X) \land b \in image(Y)\} \\
image(X \dotplus Y) &\subseteq \{a \dotplus b \mid a \in image(X) \land b \in image(Y)\} \\
image(X \cdot Y) &\subseteq \{a \cdot b \mid a \in image(X) \land b \in image(Y)\} \\
image(\mathsf{S}\, a.\, X) &\subseteq closure(image(X)) \\
image(\mathsf{S}\, a.\, X') &\subseteq image(X') \\
image(\mathsf{I}\, a.\, X) &\subseteq closure(image(X)) \\
image(\mathsf{I}\, a.\, X') &\subseteq image(X') \\
image(X \star Y) &\subseteq closure(\{a \cdot b \mid a \in image(X) \land b \in image(Y)\}) \\
image(X' \star Y') &\subseteq \{a \cdot b \mid a \in image(X) \land b \in image(Y)\} \\
image(X \multimap\!\!\circledast\, Y) &\subseteq closure(\{b \,\dot{/}\, a \mid a \in image(X) \land b \in image(Y)\}) \\
image(X' \multimap\!\!\circledast\, Y') &\subseteq \{b \,\dot{/}\, a \mid a \in image(X) \land b \in image(Y)\}
\end{aligned}$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

Theorems 8.1.1 and 8.1.2 allow us to generate (possibly infinite) many qualitative

entailments for the given operations. If the image of the random variables are, however, finite, we can fully automate the process as both theorems admit full automatization in that case. It is now left to also transform the at-least predicate into classical operations in qualitative separation logic.

8.2. Proof Rules for the At-Least Predicate

Reasoning about the predicate $a \le X$ is still hard, so it would be nice if we could translate this predicate into the logic we are already custom to.

Whenever we have $a = 0$, we have that the left-hand side of the inequality is zero and thus the proposition trivially holds. Therefore we have $0 \le X = \text{true}$. Because of this, we may assume that $0 < a$.

Lemma 8.2.1 *For a random variable $X \in FSL$ in fuzzy separation logic, we have that*

$$0 \le X \iff \text{true}.$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

For the random variables [true] and [false], the transformation is trivial. We have $a \le [\text{true}]$ is equivalent to true and if $0 < a$ then $a \le [\text{false}]$ is equivalent to false and otherwise $a \le [\text{false}]$ is equivalent to true.

Lemma 8.2.2 *For a positive real in the unit interval $a \in (0, 1]$ we have that*

$$\begin{aligned} a \le [\text{true}] &\iff \text{true}, \text{ and} \\ a \le [\text{false}] &\iff \text{false}. \end{aligned}$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

For a proposition Φ, $[\Phi]$ evaluates to 0 or 1. However, 1 values are trivial anyway as $a \le 1$ holds always. So we only need to care for the 0 values and we have for $0 < la$ that $a \le [\Phi]$ is equivalent to Φ.

For an expression e, we have that $a \le \langle e \rangle$ is equivalent to $a \le e$. We leave out here how to reasoning about the expression e. However, if e is a constant, we can check the inequality statically.

Lemma 8.2.3 *For a positive real in the unit interval $a \in (0, 1]$, a separation logic proposition $\Phi \in SL$, and a probabilistic expression e we have that*

$$\begin{aligned} a \le [\Phi] &\iff \Phi, \text{ and} \\ a \le \langle e \rangle &\iff \lambda(s, h, \eta).\, a \le e(s, \eta). \end{aligned}$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

Substitution $X[x := e]$ distributes nicely over the at-least predicate, so we obtain that $a \leq X[x := e]$ is equivalent to $(a \leq X)[x := e]$.

The quantitative negation $\sim X$ is a bit more involved. We transform $a \leq \sim X$ into the separation logic proposition

$$\neg(\inf\{ b \in A \mid 1 - a < b \} \leq X)$$

with $\text{image}(X) \subseteq A$. One direction holds without any further assumptions. We will first concentrate on this direction. If the above proposition holds, then so does $a \leq \sim X$. For the proof, we transform $(s,h) \vDash_\eta a \leq \sim X$ into $a \leq 1 - X(s,h,\eta)$ for arbitrary (s,h) and η. Due to symmetries of "$1-$" in the unit interval (compare Lemma 5.1.6), this is equivalent to $\neg(1 - a < X(s,h,\eta))$. Now we can apply contraposition and instead show that when $1 - a < X(s,h,\eta)$, we have that $(s,h) \vDash_\eta \inf\{ b \in \text{image}(X) \mid 1 - a < b \} \leq X$. We can simplify and eliminate the left infimum by proving that $X(s,h,\eta) \in \text{image}(X) \wedge 1 - a < X(s,h,\eta)$. The first part vacuously holds, the second part holds by $1 - a < X(s,h,\eta)$. When proving the validity of an entailment, this direction is the one that interests us, since when we prove an entailment with these propositions, it also tells us that the quantitative entailment holds.

Nevertheless, we can also establish the other direction if we assume that

$$1 - a < \inf\{ b \in A \mid 1 - a < b \}.$$

This condition can be guaranteed if A is finite and $0 < a$. When $a = 0$, we have again that $a \leq \sim X = [\text{true}]$. If $0 < a$, we have that $1 - a < 1$. Now we consider two cases. Either $\{ b \in A \mid 1 - a < b \}$ is empty or not. If it is empty, then $\inf\{ b \in A \mid 1 - a < b \} = 1$ and since $1 - a < 1 = \inf\{ b \in A \mid 1 - a < b \}$, the statement is proven. Otherwise if it is nonempty, we have that $1 - a < \inf\{ b \in A \mid 1 - a < b \}$ as the infimum takes a value of its set on finite sets.

Remark that since the unit interval is a complete lattice, we have $\inf \emptyset = 1$.

The other direction holds, since if we assume that $a \leq 1 - X(s,h,\eta)$ for some (s,h) and η, using symmetries from Lemma 5.1.6, we have that $X(s,h,\eta) \leq 1 - a$. And since $1 - a < \inf\{ b \in A \mid 1 - a < b \}$, it is $X(s,h,\eta) < \inf\{ b \in A \mid 1 - a < b \}$, which is equivalent to what we needed to show.

Lemma 8.2.4 *For a real $a \in [0,1]$ in the unit interval, a random variable $X \in FSL$ in fuzzy separation logic, a variable $x \in \text{Vars}$, a value expression e and an over-approximation $\text{image}(X) \subseteq A$ of the image of X we have that*

$$a \leq X[x := e] \iff (a \leq X)[x := e], \text{ and}$$
$$a \leq \sim X \Leftarrow \neg(\inf\{ b \in A \mid 1 - a < b \} \leq X).$$

If additionally we have that $1 - a < \inf\{ b \in A \mid 1 - a < b \}$ then

$$a \leq \sim X \iff \neg(\inf\{ b \in A \mid 1 - a < b \} \leq X).$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

The predicate $a \leq X \sqcap Y$ is satisfied by (s,h) and η when both $a \leq X$ and $a \leq Y$ are satisfied by (s,h) and η. Thus the predicate is equivalent to $a \leq X \wedge a \leq Y$.

Similarly, the predicate $a \leq X \sqcup Y$ is satisfied by (s,h) and η when $a \leq X$ or $a \leq Y$ is satisfied by (s,h) and η. Thus the predicate is equivalent to $a \leq X \vee a \leq Y$.

Lemma 8.2.5 *For a real $a \in [0,1]$ in the unit interval and random variables $X, Y \in FSL$ in fuzzy separation logic we have that*

$$a \leq X \sqcup Y \iff a \leq X \vee a \leq Y, \text{ and}$$
$$a \leq X \sqcap Y \iff a \leq X \wedge a \leq Y.$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

The at-least predicate with addition $a \leq X \dotplus Y$ is satisfied by (s,h) and η exactly when for some $a_1 \in \text{image}(X), a_2 \in \text{image}(Y)$ with $a \leq a_1 \dotplus a_2$ we have that both $a_1 \leq X$ and $a_2 \leq Y$ are satisfied by (s,h) and η. For the first direction, we assume that $(s,h) \models_\eta a \leq X \dotplus Y$. Then we have that $a \leq X(s,h,\eta) \dotplus Y(s,h,\eta)$. Thus with $a_1 = X(s,h,\eta)$ and $a_2 = Y(s,h,\eta)$, we have that $a_1 \leq X(s,h,\eta)$ as well as $a_2 \leq Y(s,h,\eta)$. For the other direction, we have that $a \leq a_1 \dotplus a_2 \leq X(s,h,\eta) \dotplus Y(s,h,\eta)$, which is what was to show.

Multiplication works very similar and thus we have that $a \leq X \cdot Y$ is satisfied by (s,h) and η exactly when there are some $a_1 \in \text{image}(X), a_2 \in \text{image}(Y)$ with $a \leq a_1 \cdot a_2$ such that we have that both predicates $a_1 \leq X$ and $a_2 \leq Y$ are satisfied by (s,h) and η.

Similarly to Lemma 8.2.4, we allow over-approximations of the images to ease automatic reasoning.

Lemma 8.2.6 *For a real $a \in [0,1]$ in the unit interval, the random variables $X, Y \in FSL$ in fuzzy separation logic and the over-approximations $\text{image}(X) \subseteq B, \text{image}(Y) \subseteq C$ of the image of X and Y we have that*

$$a \leq X \dotplus Y \iff \exists b \in B, c \in C.\, a \leq b \dotplus c \wedge b \leq X \wedge c \leq Y, \text{ and}$$
$$a \leq X \cdot Y \iff \exists b \in B, c \in C.\, a \leq b \cdot c \wedge b \leq X \wedge c \leq Y.$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

The infimum $a \leq \text{I}\, b.\, X$ can be translated into an all-quantification. The at-least predicate with infimum is satisfied by (s,h) and η exactly when there is an a such that $a \leq X$. This follows from the definition of the infimum.

The reasoning does not work for the supremum. The predicate $a \leq \text{Ƨ}\, b.\, X$ is only equivalent to $\exists b.\, a \leq X$ if

$$\sup \{ a \mid \exists v.\, X(s,h,\eta[b := v]) = a \} \in \{ a \mid \exists v.\, X(s,h,\eta[b := v]) = a \},$$

that is, if the respective supremum is actually achieved. This of course holds if the image of X is finite (and non-empty). The first direction holds without assuming this extra condition. That is, if $(s,h) \models_\eta \exists b.\, a \leq X$ then also $(s,h) \models_\eta a \leq \text{Ƨ}\, b.\, X$. For this, it is sufficient to show that a is smaller than any upper bound c of $X(s,h,\eta[b := v])$. Then we have that $a \leq X(s,h,\eta[b := v]) \leq c$ due to our assumption and since c is an upper bound. For the other direction, we assume that $(s,h) \models_\eta a \leq \text{Ƨ}\, b.\, X$. By our extra condition, we have that $X(s,h,\eta[b := v]) = \sup \{ a \mid \exists v.\, X(s,h,\eta[b := v]) = a \}$ for some v. Thus using v in the existential quantification, we have to prove that $a \leq \sup \{ a \mid \exists v.\, X(s,h,\eta[b := v]) = a \}$, which is the assumption.

Lemma 8.2.7 *For a real $a \in [0, 1]$ in the unit interval, a random variable $X \in FSL$ in fuzzy separation logic and an over-approximation $image(X) \subseteq B$ of the image of X we have that*

$$a \leq \text{I} b.\, X \iff \forall b.\, a \leq X, \text{ and}$$
$$a \leq \text{Ɂ} b.\, X \Leftarrow \exists b.\, a \leq X.$$

If additionally we have that for all stack heap pairs (s, h) and values η for logical variables we have that

$$\sup \{ c \mid \exists v.\, X(s, h, \eta[b := v]) = c \} \in \{ c \mid \exists v.\, X(s, h, \eta[b := v]) = c \},$$

then

$$a \leq \text{Ɂ} b.\, X \iff \exists b.\, a \leq X.$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

For the separating multiplication, two previous ideas come together. We have both the problem that the maximum may not be achieved since we did not assume that heaps are finite and we have an arithmetic operation which needs a similar treatment as for multiplication or addition. We have that if there are $a_1 \in \text{image}(X)$, $a_2 \in \text{image}(Y)$ with $a \leq a_1 \cdot a_2$ such that $(s, h) \vDash_\eta a_1 \leq X * a_2 \leq Y$, then also $(s, h) \vDash_\eta a \leq X \star Y$. This direction holds since if we can split h in h_1 and h_2 such that $(s, h_1) \vDash_\eta a_1 \leq X$ and $(s, h_2) \vDash_\eta a_2 \leq Y$, then we also have that $a \leq a_1 \cdot a_2 \leq X(s, h_1, \eta) \cdot Y(s, h_2, \eta) \leq (X \star Y)(s, h, \eta)$. For the other direction, we require that for all (s, h) and η we have

$$(X \star Y)(s, h, \eta) \in \{ X(s, h_1, \eta) \cdot Y(s, h_2, \eta) \mid h_1 \perp h_2 \wedge h = h_1 \cup h_2 \},$$

as then we can use the achieved maximum heap split as the split for the existential quantification. To guarantee the extra condition, we can either require that the images of X and Y are finite or that heaps are finite.

Remark that we now use the *separating* conjunction instead of the regular conjunction to connect the at-least predicates.

Lemma 8.2.8 *For a real $a \in [0, 1]$ in the unit interval, the random variables $X, Y \in FSL$ in fuzzy separation logic and the over-approximations $image(X) \subseteq B$, $image(Y) \subseteq C$ of the image of X and Y we have that*

$$a \leq X \star Y \Leftarrow \exists b \in B, c \in C.\, a \leq b \cdot c \wedge (b \leq X) * (c \leq Y).$$

If additionally we have that for all stack heap pairs (s, h) and values for logical variables η we have that

$$(X \star Y)(s, h, \eta) \in \{ X(s, h_1, \eta) \cdot Y(s, h_2, \eta) \mid h_1 \perp h_2 \wedge h = h_1 \cup h_2 \},$$

then

$$a \leq X \star Y \iff \exists b \in B, c \in C.\, a \leq b \cdot c \wedge (b \leq X) * (c \leq Y).$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

Lastly, we investigate truncated fuzzy magic wand. For division, the field looks much more complicated. A naïve approach may assume, similar to how we did

this before, that if for all stack heap pairs (s, h) and values for logical variables η we have

$$(X \mathrel{-\!\!\circledast} Y)(s, h, \eta) \in \{ Y(s, h \cup h', \eta) \mathbin{\dot{/}} X(s, h', \eta) \mid h' \cup h \},$$

then we also have that

$$a \leq X \mathrel{-\!\!\circledast} Y \iff \exists b \in \text{image}(X), c \in \text{image}(Y).\, a \leq c \mathbin{\dot{/}} b \wedge b \leq X \mathrel{-\!\!*} c \leq Y.$$

Unfortunately, this is unsound. As a counterexample, we consider $s(x) = 0$ $h = \emptyset$, $\eta(a) = 0$, and X and Y where for all $(s', h'), \eta'$ we have $X(s', h', \eta') = 0$ and

$$Y(s', h', \eta') = \begin{cases} h'(1) & h'(1) \neq \text{undef} \wedge 0 < h'(1) \wedge h'(1) \leq 1 \\ 1 & \text{else.} \end{cases}$$

Now we have that $(X \mathrel{-\!\!\circledast} Y)(s', h', \eta') = 1$ as $X(s', h', \eta') = 0$ and we also have that $1 \leq (X \mathrel{-\!\!\circledast} Y)(s', h', \eta')$ vacuously. However, we do not have that

$$\exists b \in B, c \in C.\, 1 \leq c \mathbin{\dot{/}} b \wedge b \leq X \mathrel{-\!\!*} c \leq Y$$

for (s, h) and η. Let b and c be such values, h' a heap with $h \cup h'$ and $(s, h') \models_\eta b \leq X$. Since we have that $c \in \text{image}(Y)$, we have that $0 < c$. However, for any fixed chosen c, we can always find h' such that $h'(1) = \frac{c}{2}$, thus we have $c > Y(s, h \cup h', \eta)$, which shows that the naïve approach does not work. A formalization of this argument can be found in [43] at `LeanFSL.Entailments.FSL2SLCounter`.

It is still unclear which variants will make this work. One very restrictive approach is to limit the left-hand side of the magic wand to qualitative random variables, i.e. random variables that map only to 0 and 1. Then we have that

$$a \leq X \mathrel{-\!\!\circledast} Y \iff 1 \leq X \mathrel{-\!\!*} a \leq Y.$$

Again we consider both directions separately. For the first direction, we assume for arbitrarily (s, h) and η that $(s, h) \models_\eta a \leq X \mathrel{-\!\!\circledast} Y$. If the left-hand side of the fuzzy magic wand is qualitative, we know that $Y(s, h \cup h', \eta) \mathbin{\dot{/}} X(s, h', \eta)$ is either equal to 1 or to $Y(s, h \cup h', \eta)$. Assuming that we have $1 \leq X(s, h', \eta)$ allows us then to reason that $a \leq Y(s, h \cup h', \eta)$.

For the other direction, we assume that whenever $1 \leq X(s, h', \eta)$ we also have that $a \leq Y(s, h \cup h', \eta)$. When $X(s, h \cup h', \eta)$ is not 1 (and thus 0), we know that $a \leq 1 = Y(s, h \cup h', \eta) \mathbin{\dot{/}} X(s, h', \eta)$. Thus we assume that $X(s, h \cup h', \eta) = 1$. From there we can derive that $a \leq Y(s, h \cup h', \eta) = Y(s, h \cup h', \eta) \mathbin{\dot{/}} X(s, h', \eta)$. In our naïve approach, this was not possible as we have allowed the left-hand side of the fuzzy magic wand to require a lower bound of 0.

Lemma 8.2.9 *For a real $a \in [0, 1]$ in the unit interval and random variables $X, Y \in FSL$ in fuzzy separation logic such that image$(X) \in \{0, 1\}$ we have that*

$$a \leq X \mathrel{-\!\!\circledast} Y \iff 1 \leq X \mathrel{-\!\!*} a \leq Y.$$

Proof. See [43] at `LeanFSL.Entailments.FSL2SL`. □

8.3. Complexity of a Syntactic Transformation

In this section, we will discuss the complexity of the transformations presented in the previous section on a certain syntax of which the semantics allow automatic reasoning of the fuzzy aspects of the entailments. Since up to this point, we did not formally introduce a syntax, arguing about runtime complexity is hard. As such, we will define formally a syntax for fuzzy and qualitative separation logic in this section, define a size measure on both of them and give a proof how the size measure changes after using the transformation rules from the previous chapter. We do not give semantics of these formulae formally, however, by mapping them to the already introduced operations, we obtain the appropriate semantics.

The results from this section were not formalized in Lean since FSL and SL are defined extensional in Lean and thus do not offer a concise syntax. We define it like this as we do not currently know guarantees that the semantics from Chapters 4 and 6 are expressible in a certain FSL and SL syntax respectively. For the results in this chapter, however, we do require a certain syntax.

Definition 8.3.1 (A Syntax for Qualitative Separation Logic) *Qualitative separation logic formulae are generated by the following grammar*

φ	$::=$	Φ	*predicates*
	$\mid$	$\varphi\,[x := e]$	*substitution*
	$\mid$	$\neg\varphi$	*negation*
	$\mid$	$\varphi \wedge \varphi$	*conjunction*
	$\mid$	$\varphi \vee \varphi$	*disjunction*
	$\mid$	$\exists a.\, \varphi$	*existential quantifier*
	$\mid$	$\forall a.\, \varphi$	*universal quantifier*
	$\mid$	$\varphi \ast \varphi$	*separating conjunction*
	$\mid$	$\varphi \mathrel{-\!\!\ast} \varphi$	*qualitative magic wand*

where Φ is a predicate symbol from a predefined set of predicates with given semantics and $x \in \mathsf{Vars} \cup \mathsf{LVars}$ is a variable and $e \colon \mathsf{Stacks} \times \mathsf{LStacks} \to \mathbb{Q}$ are value expressions.

We assume our fragment of qualitative separation logic to be instantiated by some set of predicates. We do not concern ourselves with decidability of entailments in the qualitative separation logic formulae, thus leave finding a suitable fragment in qualitative separation logic with suitable predicates to the reader.

[83–97] may provide the reader with literature about suitable fragments in qualitative separation logic.

We will limit the possible syntax for our fuzzy separation logic fragment to allow a transformation in which fuzzy aspects can be checked automatically. This especially involves requiring *constant* quantitative expressions, such that these can be handled during the computation of qualitative separation logic formulae. Moreover, due to the problems about magic wands we discussed in the previous section, we require magic wands to be guarded by a qualitative formula.

Definition 8.3.2 (A Syntax for Fuzzy Separation Logic) *Fuzzy separation logic formulae are generated by the following grammar*

$$
\begin{array}{rcll}
f & ::= & [\varphi] & \textit{iverson brackets} \\
& | & \langle b \rangle & \textit{rational number} \\
& | & f\,[x := e] & \textit{substitution} \\
& | & {\sim} f & \textit{fuzzy negation} \\
& | & f \sqcup f & \textit{maximum} \\
& | & f \sqcap f & \textit{minimum} \\
& | & f \dotplus f & \textit{addition} \\
& | & f \cdot f & \textit{multipication} \\
& | & \mathsf{S}\, a.\, f & \textit{supremum quantifier} \\
& | & \mathsf{I}\, a.\, f & \textit{infimum quantifier} \\
& | & f \star f & \textit{separating multiplication} \\
& | & [\varphi] \multimap f & \textit{guarded magic wand}
\end{array}
$$

where φ is a formula in the syntax for qualitative separation logic from Definition 8.3.1, $b \in [0,1] \cap \mathbb{Q}$ is a rational number in the unit interval, x, e as in Definition 8.3.1.

Since the correctness of the translations follows from Lemmas 8.2.1 to 8.2.9, we concentrate on the *complexity* of our transformation and do not introduce a formal semantics for the formulae. However, their semantics is obtained by replacing the respective symbols with the operations for which we used the same symbols in fuzzy and qualitative separation logic. The transformation $\mathrm{fsl2sl}(a, f)$ can be used to transform the predicate $a \leq f$ into a formula using the syntax from Definition 8.3.1. For some of these lemmas, we also allowed over-approximations of the images. Computing the exact image is infeasible, thus we introduce an over-approximation $\mathrm{apImg}(f)$ of the image using Theorem 8.1.2.

Definition 8.3.3 (Over-approximation of Fuzzy Separation Logic Images) *We define the over-approximation $apImg(f)$ for f as in Definition 8.3.2 as*

$$
\begin{array}{rcl}
apImg([\varphi]) & = & \{0, 1\} \\
apImg(\langle b \rangle) & = & \{0, b, 1\} \\
apImg(f\,[x := e]) & = & apImg(f) \\
apImg({\sim} f) & = & \{1 - a \mid a \in apImg(f)\} \\
apImg(f \sqcup g) & = & apImg(f) \cup apImg(g) \\
apImg(f \sqcap g) & = & apImg(f) \cup apImg(g) \\
apImg(f \dotplus g) & = & \{a \dotplus b \mid a \in apImg(f) \land b \in apImg(g)\} \\
apImg(f \cdot g) & = & \{a \cdot b \mid a \in apImg(f) \land b \in apImg(g)\} \\
apImg(\mathsf{S}\, a.\, f) & = & apImg(f) \\
apImg(\mathsf{I}\, a.\, f) & = & apImg(f) \\
apImg(f \star g) & = & \{a \cdot b \mid a \in apImg(f) \land b \in apImg(g)\} \\
apImg([\varphi] \multimap f) & = & apImg(f)
\end{array}
$$

where φ, b, x, e are as in Definition 8.3.2.

We only include 0 and 1 in the over-approximation of $\mathrm{apImg}(\langle b \rangle)$ to ease the complexity reasoning later.

We later prove that $\{0, 1\} \subseteq \mathrm{apImg}(f)$, thus we do not explicitly add these values for the image of the magic wand.

The approximation does not require computing any closures as the set is always finite. This is guaranteed as the syntax we introduced only allows constants in quantitative expressions. Finiteness of the images also guarantee us the side conditions from Lemmas 8.2.4, 8.2.7 and 8.2.8. Due to this and Theorem 8.1.2, $\mathrm{apImg}(f)$ does indeed yield an over-approximation of the image of the semantics of f.

Definition 8.3.4 (Transformation) *We define the transformation* $fsl2sl(a, f)$ *for* $a \in [0,1] \cap \mathbb{Q}$ *and* f *as in Definition 8.3.2 as*

$$
\begin{aligned}
fsl2sl(a, [\varphi]) &= \begin{cases} \mathsf{true} & \text{if } a = 0 \\ \varphi & \text{else} \end{cases} \\
fsl2sl(a, \langle b \rangle) &= \begin{cases} \mathsf{true} & \text{if } a \leq b \\ \mathsf{false} & \text{else} \end{cases} \\
fsl2sl(a, f\,[x := e]) &= fsl2sl(a, f)\,[x := e] \\
fsl2sl(a, \sim f) &= \begin{cases} \mathsf{true} & \text{if } a = 0 \\ \neg fsl2sl(\inf\{\, b \in A \mid 1 - a < b\,\}, f) & \text{else} \end{cases} \\
fsl2sl(a, f \sqcup g) &= fsl2sl(a, f) \vee fsl2sl(a, g) \\
fsl2sl(a, f \sqcap g) &= fsl2sl(a, f) \wedge fsl2sl(a, g) \\
fsl2sl(a, f \dotplus g) &= \bigvee_{\substack{b \in apImg(f), c \in apImg(g) \\ a \leq b \dotplus c}} fsl2sl(b, f) \wedge fsl2sl(c, g) \\
fsl2sl(a, f \cdot g) &= \bigvee_{\substack{b \in apImg(f), c \in apImg(g) \\ a \leq b \cdot c}} fsl2sl(b, f) \wedge fsl2sl(c, g) \\
fsl2sl(a, \mathcal{S}\, a.\, f) &= \exists a.\, fsl2sl(a, f) \\
fsl2sl(a, \mathrm{I}\, a.\, f) &= \forall a.\, fsl2sl(a, f) \\
fsl2sl(a, f \star g) &= \bigvee_{\substack{b \in apImg(f), c \in apImg(g) \\ a \leq b \cdot c}} fsl2sl(b, f) * fsl2sl(c, g) \\
fsl2sl(a, [\varphi] \multimap\!\!\circledast\, f) &= \varphi \mathrel{-\!\!*} fsl2sl(a, f)
\end{aligned}
$$

where φ, b, x, e *are as in Definition 8.3.2.*

Definition 8.3.4 applies Lemmas 8.2.1 to 8.2.9 to formulae in the syntax from Definition 8.3.2 and thus yields sound equivalences for $a \leq X$. The existential quantification for values in the images of f and g transform to disjunctions as the images are finite. Side conditions are automatically satisfied due to the finiteness of the images. Indeed, we see that the soundness of the here presented transformation relies on the finiteness of the images.

Next, we define size measures for the formulae to define the increase of size by applying $\mathrm{fsl2sl}(a, f)$. We define these simply by assuming that every operation application increases the size by one.

Definition 8.3.5 (Size of Qualitative Separation Logic Formulae) *We define the size of a formula in the syntax from Definition 8.3.1 as*

$$\begin{aligned}
|\varphi| &= 1 \\
|\varphi\,[x := e]| &= |\varphi| + 1 \\
|\neg\varphi| &= |\varphi| + 1 \\
|\varphi \wedge \psi| &= |\varphi| + |\psi| + 1 \\
|\varphi \vee \psi| &= |\varphi| + |\psi| + 1 \\
|\exists a.\, \varphi| &= |\varphi| + 1 \\
|\forall a.\, \varphi| &= |\varphi| + 1 \\
|\varphi * \psi| &= |\varphi| + |\psi| + 1 \\
|\varphi \mathrel{-\!\!*} \psi| &= |\varphi| + |\psi| + 1
\end{aligned}$$

where φ, x, e are as in Definition 8.3.2.

Definition 8.3.6 (Size of Fuzzy Separation Logic Formulae) *We define the size of a formula in the syntax from Definition 8.3.2 as*

$$\begin{aligned}
|[\varphi]| &= |\varphi| \\
|\langle b \rangle| &= 1 \\
|f\,[x := e]| &= |f| + 1 \\
|{\sim} f| &= |f| + 1 \\
|f \sqcup g| &= |f| + |g| + 1 \\
|f \sqcap g| &= |f| + |g| + 1 \\
|f \dotplus g| &= |f| + |g| + 1 \\
|f \cdot g| &= |f| + |g| + 1 \\
|\mathsf{S}a.\, f| &= |f| + 1 \\
|\mathsf{I}\, a.\, f| &= |f| + 1 \\
|f \star g| &= |f| + |g| + 1 \\
|[\varphi] \mathrel{-\!\!\circledast} f| &= |\varphi| + |f| + 1
\end{aligned}$$

where φ, x, e are as in Definition 8.3.2.
We define a different size *of a formula in the syntax from Definition 8.3.2 as*

$|f|_p$ counts the operations contributing to exponential complexity in the transformation. The p here stands for product, as the operations we count induce set products in the given over-approximations.

$$\begin{aligned}
|[\varphi]|_p &= 0 \\
|\langle b \rangle|_p &= 1 \\
|f\,[x := e]|_p &= |f|_p \\
|{\sim} f|_p &= |f|_p \\
|f \sqcup g|_p &= |f|_p + |g|_p \\
|f \sqcap g|_p &= |f|_p + |g|_p \\
|f \dotplus g|_p &= |f|_p + |g|_p + 1 \\
|f \cdot g|_p &= |f|_p + |g|_p + 1 \\
|\mathsf{S}a.\, f|_p &= |f|_p
\end{aligned}$$

$$\begin{aligned} |\mathrm{I}\, a.\, f|_p &= |f|_p \\ |f \star g|_p &= |f|_p + |g|_p + 1 \\ |[\varphi] \multimap\!\!\circledast f|_p &= |f|_p \end{aligned}$$

where φ, x, e are as in Definition 8.3.2.

We will now progress to the complexity proofs aimed for in this section. We aim for three results. First, we like to estimate the complexity of the over-approximations $\mathrm{apImg}(f)$. Indeed, the size of this set also influences the size of the formula generated by $\mathrm{fsl2sl}(a, f)$. The second result thus considers the size of the generated formula. Lastly, we combine the two results to obtain an asymptotic complexity on the number of formulae and their sizes. This gives us an estimate how much work this approach costs us when we want to check an FSL formula in the syntax from Definition 8.3.2 using SL entailment solvers.

The size of the set $\mathrm{apImg}(f)$ is quadratically increased every time we encounter an operation that may generate new values, e.g. for addition we have for some g_1 and g_2 that

$$\mathrm{apImg}(g_1 \dotplus g_2) \approx |\mathrm{apImg}(g_1)| \cdot |\mathrm{apImg}(g_2)|.$$

Maxima and minima on the other side only generate a linear complexity increase, i.e. we have that

$$\mathrm{apImg}(g_1 \sqcup g_2) \leq 2 \cdot \max \left\{ |\mathrm{apImg}(g_1)|,\ |\mathrm{apImg}(g_2)| \right\}.$$

The number of quadratic increases depend on the size of the formula. Thus we obtain an exponential complexity in the size of the formula f.

Theorem 8.3.1 *Let f be a formula as in Definition 8.3.2. We have that*

$$|\mathit{apImg}(f)| \leq 2^{|f|_p + 1}.$$

Proof. We prove this by induction on the grammar of the formula f. For the induction base we have

$$\begin{aligned} |\mathrm{apImg}([\varphi])| &= |\{0, 1\}| = 2^1 = 2^{|[\varphi]|_p + 1}, \text{and} \\ |\mathrm{apImg}(\langle b \rangle)| &= |\{0, b, 1\}| = 3 \leq 2^2 = 2^{|\langle b \rangle|_p + 1}. \end{aligned}$$

Thus we now assume as induction hypothesis that $|\mathrm{apImg}(g_1)| \leq 2^{|g_1|_p + 1}$ and $|\mathrm{apImg}(g_2)| \leq 2^{|g_2|_p + 1}$ holds. We prove four cases. All other cases are analogous.

1. For $g_1\, [x := e]$ we have that

$$|\mathrm{apImg}(g_1\, [x := e])| = |\mathrm{apImg}(g_1)| \overset{IH}{\leq} 2^{|\mathrm{apImg}(g_1)| + 1} = 2^{|\mathrm{apImg}(g_1\, [x := e])| + 1}.$$

2. For $\sim g_1$ we have since $1 - a$ is a bijection that

$$\begin{aligned} |\mathrm{apImg}(\sim g_1)| &= \left| \left\{ 1 - a \,\middle|\, a \in \mathrm{apImg}(g_1) \right\} \right| = |\mathrm{apImg}(g_1)| \\ &\overset{IH}{\leq} 2^{|\mathrm{apImg}(g_1)| + 1} = 2^{|\mathrm{apImg}(\sim g_1)| + 1}. \end{aligned}$$

3. For $g_1 \sqcup g_2$ we require a little lemma:

Lemma 8.3.2 *For an arbitrary formula g defined as in Definition 8.3.2, we have that $\{0, 1\} \subseteq apImg(g)$.*

This lemma would not hold if we instead had $\mathrm{apImg}(\langle b \rangle) = \{b\}$.

Proof. By induction over g. □

We now differentiate three cases.
For $|g_1|_p = 0$ we have by the induction hypothesis

$$|\mathrm{apImg}(g_1)| \leq 2$$

and therefore by the previous lemma that

$$\mathrm{apImg}(g_1) = \{0, 1\} \text{ and } \{0, 1\} \subseteq \mathrm{apImg}(g_2).$$

Then we also have that

$$\begin{aligned} |\mathrm{apImg}(g_1 \sqcup g_2)| &= |\mathrm{apImg}(g_1) \cup \mathrm{apImg}(g_2)| = |\mathrm{apImg}(g_2)| \\ &\leq |\mathrm{apImg}(g_2)| \overset{IH}{\leq} 2^{|g_2|_p+1} = 2^{|g_1 \sqcup g_2|_p+1} \end{aligned}$$

The proof for $|g_2|_p = 0$ is similar to the proof for $|g_1|_p = 0$.
Lastly, for $0 < |g_1|_p$ and $0 < |g_2|_p$, we have that

$$\begin{aligned} |\mathrm{apImg}(g_1 \sqcup g_2)| &= |\mathrm{apImg}(g_1) \cup \mathrm{apImg}(g_2)| \\ &\leq |\mathrm{apImg}(g_1)| + |\mathrm{apImg}(g_2)| \overset{IH}{\leq} 2^{|g_1|_p+1} + 2^{|g_2|_p+1} \end{aligned}$$

Since we have that $0 < |g_1|_p$ and $0 < |g_2|_p$, we also have the following inequality

$$2^{|g_1|_p+1} + 2^{|g_2|_p+1} \leq 2^{|g_1|_p+|g_2|_p+1}$$

and thus

$$|\mathrm{apImg}(g_1 \sqcup g_2)| \leq 2^{|g_1|_p+|g_2|_p+1} = 2^{|g_1 \sqcup g_2|_p+1}.$$

4. For $g_1 \dot{+} g_2$ we have that

$$\begin{aligned} |\mathrm{apImg}(g_1 \dot{+} g_2)| &= \left| \left\{ a \dot{+} b \,\middle|\, a \in \mathrm{apImg}(g_1) \wedge b \in \mathrm{apImg}(g_2) \right\} \right| \\ &\leq |\mathrm{apImg}(g_1)| \cdot |\mathrm{apImg}(g_2)| \overset{IH}{\leq} 2^{|g_1|_p+1} \cdot 2^{|g_2|_p+1} \\ &= 2^{|g_1|_p+|g_2|_p+2} = 2^{|g_1 \dot{+} g_2|_p+1} \end{aligned}$$

□

The previous theorem yielded an upper bound on the size of $\mathrm{apImg}(f)$. To have a proper complexity estimate for the entailments we obtain by the transformation $\mathrm{fsl2sl}(a, f)$, we also require the size of the resulting formula of $\mathrm{fsl2sl}(a, f)$. It may not be surprising that the formula is also exponential in the size of f, because the size of the set $\mathrm{apImg}(f)$ is exponential in the size of f.

Theorem 8.3.3 *Let $a \in [0, 1] \cap \mathbb{Q}$ be a rational number in the unit interval and f be a formula as in Definition 8.3.2. We have that*

$$|fsl2sl(a, f)| \leq 2 \cdot |f| \cdot 2^{(|f|_p+1)^2}.$$

Proof. We prove this by induction on the grammar of the formula f for all rationals $a \in [0, 1] \cap \mathbb{Q}$ in the unit interval.

For the base cases $[\varphi]$ we have if $a = 0$ then

$$|\mathrm{fsl2sl}(a, [\varphi])| = |\mathtt{true}| \leq 1 \leq 2 \cdot |[\varphi]| \cdot 2^{(|[\varphi]|_p+1)^2}$$

and if $0 < a$ then

$$|\mathrm{fsl2sl}(a, [\varphi])| = |\varphi| \leq 2 \cdot |[\varphi]| \cdot 2^{(|[\varphi]|_p+1)^2}.$$

For the base case $\langle b \rangle$ we have if $a \leq b$ then

$$|\mathrm{fsl2sl}(a, \langle b \rangle)| = |\mathtt{true}| = 1 \leq 2 \cdot |\langle b \rangle| \cdot 2^{(|\langle b \rangle|_p+1)^2}$$

and similarly for the other case.

As induction hypothesis we now assume that

$$|\mathrm{fsl2sl}(a, g_1)| \leq 2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2},$$
$$|\mathrm{fsl2sl}(a, g_2)| \leq 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2}.$$

For $g_1\,[x := e]$ we have that

$$\begin{aligned} &|\mathrm{fsl2sl}(a, g_1\,[x := e])| = |\mathrm{fsl2sl}(a, g_1)\,[x := e]| \\ &= |\mathrm{fsl2sl}(a, g_1)| + 1 \overset{IH}{\leq} 2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 1 \\ &\leq 2 \cdot |g_1\,[x := e]| \cdot 2^{(|g_1[x:=e]|_p+1)^2}. \end{aligned}$$

For $\sim g_1$ we distinguish between the case $a = 0$, where

$$|\mathrm{fsl2sl}(a, \sim g_1)| = |\mathtt{true}| \leq 2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2}$$

and where $0 < a$, for which the proof is very similar to the case $g_1\,[x := e]$.

For $g_1 \sqcup g_2$ we have that

$$\begin{aligned} &|\mathrm{fsl2sl}(a, g_1 \sqcup g_2)| \\ =\ &|\mathrm{fsl2sl}(a, g_1) \vee \mathrm{fsl2sl}(a, g_2)| \\ =\ &|\mathrm{fsl2sl}(a, g_1)| + |\mathrm{fsl2sl}(a, g_2)| + 1 \\ \overset{IH}{\leq}\ &2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} + 1 \\ \leq\ &2 \cdot |g_1| \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} + 2 \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \\ =\ &2 \cdot (|g_1| + |g_2| + 1) \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \\ =\ &2 \cdot |g_1 \sqcup g_2| \cdot 2^{(|g_1 \sqcup g_2|_p+1)^2}. \end{aligned}$$

The proof for $g_1 \sqcap g_2$ is analogous to $g_1 \sqcup g_2$.

For $g_1 \dotplus g_2$ we have that

$$\begin{aligned} &|\mathrm{fsl2sl}(a, g_1 \dotplus g_2)| \\ =\ &\left| \bigvee_{b \in \mathrm{apImg}(g_1), c \in \mathrm{apImg}(g_2), a \leq b \dotplus c} \mathrm{fsl2sl}(b, g_1) \wedge \mathrm{fsl2sl}(c, g_2) \right| \end{aligned}$$

Now for b and c maximizing $|\mathrm{fsl2sl}(b, g_1)|$ and $|\mathrm{fsl2sl}(c, g_2)|$, we have that

$$\begin{aligned} & |\mathrm{fsl2sl}(a, g_1 \dot{+} g_2)| \\ \leq{} & |\mathrm{apImg}(g_1)| \cdot |\mathrm{apImg}(g_2)| \cdot (|\mathrm{fsl2sl}(b, g_1)| + |\mathrm{fsl2sl}(c, g_2)| + 2). \end{aligned}$$

Furthermore using Theorem 8.3.1 and the induction hypothesis, we obtain

$$\begin{aligned} & |\mathrm{fsl2sl}(a, g_1 \dot{+} g_2)| \\ \leq{} & |\mathrm{apImg}(g_1)| \cdot |\mathrm{apImg}(g_2)| \cdot (|\mathrm{fsl2sl}(b, g_1)| + |\mathrm{fsl2sl}(c, g_2)| + 2) \\ \leq{} & 2^{|g_1|_p+1} \cdot 2^{|g_2|_p+1} \cdot (2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} + 2) \qquad (\dagger) \end{aligned}$$

Remark that by assumption we have $0 < |g_1 \dot{+} g_2|_p$. We now distinguish two cases. Either $|g_1 \dot{+} g_2|_p = 1$, then we have $|g_1|_p = |g_2|_p = 0$ and

$$\begin{aligned} & |\mathrm{fsl2sl}(a, g_1 \dot{+} g_2)| \\ \leq{} & 2^{|g_1|_p+1} \cdot 2^{|g_2|_p+1} \cdot (2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} + 2) && (\text{see } \dagger) \\ ={} & 2^1 \cdot 2^1 \cdot (2 \cdot |g_1| \cdot 2^1 + 2 \cdot |g_2| \cdot 2^1 + 2) && (|g_1|_p = |g_2|_p = 0) \\ ={} & 2^4 \cdot (|g_1| + |g_2| + 1/2) \\ \leq{} & 2^4 \cdot |g_1 \dot{+} g_2| \\ \leq{} & 2 \cdot |g_1 \dot{+} g_2| \cdot 2^{(|g_1 \dot{+} g_2|_p+1)^2}. && (|g_1 \dot{+} g_2|_p = 1) \end{aligned}$$

If we have $1 < |g_1 \dot{+} g_2|_p$ then we have

$$\begin{aligned} & |\mathrm{fsl2sl}(a, g_1 \dot{+} g_2)| \\ \leq{} & 2^{|g_1|_p+1} \cdot 2^{|g_2|_p+1} \cdot (2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} + 2) \qquad (\text{see } \dagger) \\ ={} & 2^{|g_1 \dot{+} g_2|_p+1} \cdot (2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} + 2) \\ ={} & \quad 2 \cdot |g_1| \cdot 2^{(|g_1|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ & + 2 \cdot |g_2| \cdot 2^{(|g_2|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ & + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ \leq{} & \quad 2 \cdot (|g_1| + |g_2|) \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ & + 2 \cdot (|g_1| + |g_2|) \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ & + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ ={} & 2 \cdot 2 \cdot (|g_1| + |g_2|) \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ \leq{} & 2 \cdot 2 \cdot |g_1 \dot{+} g_2| \cdot 2^{(|g_1|_p+|g_2|_p+1)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ ={} & 2 \cdot 2 \cdot |g_1 \dot{+} g_2| \cdot 2^{(|g_1 \dot{+} g_2|_p)^2} \cdot 2^{|g_1 \dot{+} g_2|_p+1} + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ ={} & 2 \cdot |g_1 \dot{+} g_2| \cdot 2^{(|g_1 \dot{+} g_2|_p)^2+|g_1 \dot{+} g_2|_p+2} + 2 \cdot 2^{|g_1 \dot{+} g_2|_p+1} \\ \leq{} & 2 \cdot |g_1 \dot{+} g_2| \cdot (2^{(|g_1 \dot{+} g_2|_p)^2+|g_1 \dot{+} g_2|_p+2} + 2^{|g_1 \dot{+} g_2|_p+1}) \end{aligned}$$

For the last step we need the following side lemma:

Lemma 8.3.4 *For all natural numbers* $1 < n$ *we have* $2^{n^2+n+2} + 2^{n+1} \leq 2^{(n+1)^2}$.

Proof. By induction over n.

For the base case $n = 2$ we have $2^{2^2+2+2} + 2^{2+1} = 2^8 + 2^3 < 2^8 + 2^8 = 2^{(2+1)^2}$.

As induction hypothesis we assume that $2^{n^2+n+2} + 2^{n+1} \leq 2^{(n+1)^2}$.

We show that

$$\begin{aligned}
& 2^{(n+1)^2+n+1+2} + 2^{n+1+1} \\
=\ & 2^{n^2+2n+1+n+1+2} + 2^{n+1+1} \\
=\ & 2^{2n+2} \cdot 2^{n^2+n+2} + 2 \cdot 2^{n+1} \\
\leq\ & 2^{2n+2} \cdot 2^{n^2+n+2} + 2^{2n+2} \cdot 2^{n+1} \\
\leq\ & 2^{2n+2} \cdot (2^{n^2+n+2} + 2^{n+1}) \\
\leq\ & 2^{2n+2} \cdot (2^{(n+1)^2}) && \text{(IH)} \\
=\ & 2^{n^2+4n+3} \\
\leq\ & 2^{n^2+4n+4} \\
=\ & 2^{(n+1+1)^2}.
\end{aligned}$$

□

Using the previous lemma, we finally have that

$$\begin{aligned}
& |\mathrm{fsl2sl}(a, g_1 \dotplus g_2)| \\
\leq\ & 2 \cdot |g_1 \dotplus g_2| \cdot (2^{(|g_1 \dotplus g_2|_p)^2 + |g_1 \dotplus g_2|_p + 2} + 2^{|g_1 \dotplus g_2|_p + 1}) \\
\leq\ & 2 \cdot |g_1 \dotplus g_2| \cdot 2^{(|g_1 \dotplus g_2|_p + 1)^2}
\end{aligned}$$

The proof for $g_1 \cdot g_2$ is analogous to $g_1 \dotplus g_2$.

The proof for $\mathcal{E} a.\, g_1$ is analogous to $g_1\, [x := e]$.

The proof for $\mathrm{I}\, a.\, g_1$ is analogous to $g_1\, [x := e]$.

The proof for $g_1 \star g_2$ is analogous to $g_1 \dotplus g_2$.

For $[\varphi] \multimap\!\circledast\ g_1$ we have that

$$\begin{aligned}
& |\mathrm{fsl2sl}(a, [\varphi] \multimap\!\circledast\ g_1)| \\
=\ & |\varphi \mathrel{-\!\!*} \mathrm{fsl2sl}(a, g_1)| \\
=\ & |\varphi| + |\mathrm{fsl2sl}(a, g_1)| + 1 \\
\leq\ & |\varphi| + 2 \cdot |g_1| \cdot 2^{(|g_1|_p + 1)^2} + 1 && \text{(IH)} \\
\leq\ & 2 \cdot |g_1| \cdot 2^{(|[\varphi] \multimap\circledast g_1|_p + 1)^2} + (|\varphi| + 1) \cdot 2^{(|[\varphi] \multimap\circledast g_1|_p + 1)^2} \\
=\ & 2 \cdot (|g_1| + (|\varphi|/2 + 1/2)) \cdot 2^{(|[\varphi] \multimap\circledast g_1|_p + 1)^2} \\
\leq\ & 2 \cdot |[\varphi] \multimap\!\circledast\ g_1| \cdot 2^{(|[\varphi] \multimap\circledast g_1|_p + 1)^2}.
\end{aligned}$$

□

The previous two theorems give us all ingredients to obtain the overall complexity for generating SL entailments from an FSL entailment in the syntax given from Definition 8.3.2. Indeed, when we have an entailment of the form $f \vDash g$, we can transform this entailment into the entailments

$$\forall a \in \mathrm{apImg}(f).\, \mathrm{fsl2sl}(a, f) \vDash \mathrm{fsl2sl}(a, g).$$

We know that this will be at most $2^{|f|_p+1}$ many entailments due to Theorem 8.3.1 and each entailment will have size at most $2 \cdot |f| \cdot 2^{(|f|_p+1)^2} + 2 \cdot |g| \cdot 2^{(|g|_p+1)^2}$ due to Theorem 8.3.3. Therefore, we obtain an exponential overhead in the sizes $|f|$ and $|g|$ by this procedure.

Discharging the obtained entailments to a qualitative separation logic entailment checker may increase the complexity further. We offer an overview over some common entailment checkers in Table 8.1. Deciding which fragment is necessary depends on the obtained operations from the transformation. We mark in this table an operation as “pure” if it is only allowed to be used with pure predicates, i.e. for which changing the heap does not change the validity. The symbol + means that this operation is (with potentially minor restrictions) supported. The symbol − means that this fragment does not support this symbol. The mark “flat” means that this quantifier is only allowed on the outermost part of the formula. $*$ means that there are stronger restrictions on the use of these operations, but they are generally supported. For inductive predicates, some supports Lists marginnote[96] also supports lists of lists, i.e. 2 dimensional lists.. User defined predicates are predicates for which the user has to give a definition in some kind of framework. Often this is done by means of recursive definitions (compare also Chapter 9). We provide the respective complexity bounds if known in the table as well to estimate the obtained complexity for the technique presented in this section if we were to discharge the resulting entailments with this algorithm.

In order to verify which fragment is required by using the transformation from Definition 8.3.4, we provide a table that gives a quick hint. Whenever an FSL entailment includes an operation which we list on the left-hand side, the corresponding operations on the right-hand side are required to be supported by the algorithm to check the SL entailment. Some of the fragments in Table 8.1 have special requirements. How these requirements translate into the FSL fragment is currently not clear and requires further research in order to apply them. The substitution operation is usually not supported by entailment checkers. However, usually we are able to eliminate substitution syntactically.

Table 8.1.: We offer in this table various fragments in *SL* for which the entailment problem is decidable. We also provide hints on their support for various operations, the supported predicates for data structures and the complexity, if it is known. + means it is supported, $*$ means it is supported under restrictions, “pure” means it is supported for non-heap related predicates, “flat” means the quantifier is supported on the highest level of the formula and – means that it is not supported.

Paper	$\neg$	$\wedge$	$\vee$	$-\!\!*$	$\exists$	$\forall$	Ind. predicates	Complexity
[83]	pure	pure	pure	–	flat	–	user defined	ExpTime-hard
[84] [85]	–	pure	–	–	–	–	Lists	Polynomial
[86]	–	–	–	–	+	–	user defined	2-ExpTime-complete
[87]	–	–	+	–	+	–	user defined	2-ExpTime-complete
[88]	–	+	–	–	flat	–	user defined	?
[89]	–	pure	–	–	flat	–	user defined	ExpTime-complete
[90]	–	–	–	–	+	–	user defined	2-ExpTime
[91]	$*$	+	+	$*$	–	–	user defined	2-ExpTime
[92]	+	+	+	+	–	–	–	?
[95]	pure	+	+	–	+	–	–	Polynomial
[93]	+	+	+	–	–	–	Lists	PSpace-complete
[97]	+	+	+	+	–	–	Lists	PSpace-complete
[96]	+	+	+	–	–	–	Lists (of Lists)	PSpace-complete
[94]	+	+	+	$*$	$*$	$*$	–	PSpace-complete

Table 8.2.: SL operations required to discharge the entailments obtained by using the transformation from FSL to SL.

FSL fragment contains	SL contains/is closed under
$[\Psi]$	Ψ, true
$\langle a \rangle$	true, false
$X[x := e]$	$[x := e]$
$\sim X$	$\neg$, true
$X \sqcup Y$	$\vee$
$X \sqcap Y$	$\wedge$
$X \dotplus Y$	$\wedge$, $\vee$
$X \cdot Y$	$\wedge$, $\vee$
$\mathsf{S} a.\, X$	$\exists$
$\mathsf{I} a.\, X$	$\forall$
$X \star Y$	$*$, $\vee$
$[\Psi] \mathrel{-\!\!\circledast} X$	$\Psi \mathrel{-\!\!*} \cdot$

9. Cyclic Proofs for Quantitative Separation Logic

Separation logic is aiming to reason about heaps. Heaps are often used to implement various data structures such as singly linked lists, doubly linked lists, trees and many more. There is no direct way to express such data structures in separation logic natively. In this chapter, we use recursively-defined random variables. These are random variables in quantitative separation logic which are defined over equations. The random variable maps stack heap pairs to a value that is the (least) solution of that equation. Using this equation system, we can define predicates that are one whenever the heap consists of a certain data structure, thus obtaining predicates describing these data structures.

For example, we can define a random variable mapping stack heap pairs to one if and only if it is a singly linked list by the equation

$$[\mathrm{ls}]\,(a, c) \;=\; ([\mathsf{emp}] \cdot [a = c]) \sqcup (\mathfrak{S} b.\; [a \mapsto b] \star [\mathrm{ls}]\,(b, c)).$$

Note that this equation system now is over *parameterized random variables*. That is, the free variable in this equation is $[\mathrm{ls}]\,(a, c)$, which represents a function $\mathbb{Z} \times \mathbb{Z} \to \mathsf{RV}^{\infty}$. We can also define the size of a list by the equation

$$\text{ls-size}(a, c) \;=\; \mathfrak{S} b.\; [a \mapsto b] \star \text{ls-size}(b, c) + [a \mapsto b] \star [\mathrm{ls}]\,(b, c).$$

Remark that we need to use [ls] on the right-hand side of the addition to guarantee that the heap has the same form in both sides of the addition.

Defining these predicates by an equation is very elegant, but often requires some form of induction to reason about entailments containing them. And, even worse, since the predicate may not necessarily be ω-Scott-continuous, we cannot use natural induction by using Theorem 2.3.1, but instead need to use trans-finite induction by using Theorem 2.3.3. Both of these induction principles require that a user needs to guess the induction hypothesis. Automatizing guessing an induction hypothesis may be possible in certain cases, but is hard (and even impossible) in general. We choose to use a different principle, the principle of *infinite descent*. For infinite descent, we create proof trees that may have *backlinks*, i.e. edges that connect a node back to a previous node and thus create a cyclic graph. We call such a cyclic proof graph a pre-proof. In this pre-proof, we need to find certain values that decreases infinitely often in cycles present in the pre-proof. We then additionally require that the values are over a domain that does not contain any infinitely descending sequences. By performing a proof by contradiction, we can then obtain a contradiction to the previous found infinitely descending chain.

For an example, let us consider the entailment

$$\text{ls-size}(x, y) \star [\mathrm{ls}]\,(y, z) \models \text{ls-size}(x, z).$$

It expresses that the size of a sub-list is smaller than the size of the whole list. But how can we prove this? Since we have equations for the predicates $\text{ls-size}(x, y)$ and $[\mathrm{ls}]\,(y, z)$, we could replace them with the other side of the equation. This is called *unfolding*. However, unfolding predicates only introduces new predicates, which we may need to unfold again. Constructing a proof tree to illustrate this problem looks like the proof in Figure 9.1. Indeed, eventually we see the same entailment reappear. It opens the question, whether we could somehow *reuse* the existing proof to prove this later statement. In a sense, could we create a

$$
\dfrac{
\text{ls-size}(a,y) \star [\text{ls}](y,z) \models \text{ls-size}(a,z)
\qquad
\dfrac{\vdots}{[\text{ls}](a,y) \star [\text{ls}](y,z) \models [\text{ls}](a,z)}
}{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\begin{smallmatrix}\text{ls-size}(a,y)\star[\text{ls}](y,z)\\ +[\text{ls}](a,y)\star[\text{ls}](y,z)\end{smallmatrix} \models \begin{smallmatrix}\text{ls-size}(a,z)\\ +[\text{ls}](a,z)\end{smallmatrix}
}{
[x \mapsto a] \star \begin{smallmatrix}\text{ls-size}(a,y)\star[\text{ls}](y,z)\\ +[\text{ls}](a,y)\star[\text{ls}](y,z)\end{smallmatrix} \models [x \mapsto a] \star \begin{smallmatrix}\text{ls-size}(a,z)\\ +[\text{ls}](a,z)\end{smallmatrix}
}
}{
\begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,y)\star[\text{ls}](y,z)\\ +[x\mapsto a]\star[\text{ls}](a,y)\star[\text{ls}](y,z)\end{smallmatrix} \models \begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,z)\\ +[x\mapsto a]\star[\text{ls}](a,z)\end{smallmatrix}
}
}{
\begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,y)\star[\text{ls}](y,z)\\ +[x\mapsto a]\star[\text{ls}](a,y)\star[\text{ls}](y,z)\end{smallmatrix} \models \reflectbox{S} a. \begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,z)\\ +[x\mapsto a]\star[\text{ls}](a,z)\end{smallmatrix}
}
}{
\begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,y)\star[\text{ls}](y,z)\\ +[x\mapsto a]\star[\text{ls}](a,y)\star[\text{ls}](y,z)\end{smallmatrix} \models \text{ls-size}(x,z)
}
}{
\left(\reflectbox{S} a. \begin{smallmatrix}[x\mapsto a]\star\text{ls-size}(a,y)\\ +[x\mapsto a]\star[\text{ls}](a,y)\end{smallmatrix}\right) \star [\text{ls}](y,z) \models \text{ls-size}(x,z)
}
}{
\text{ls-size}(x,y) \star [\text{ls}](y,z) \models \text{ls-size}(x,z)
}
}
$$

Figure 9.1.: A proof tree demonstrating reasoning about entailments in QSL which eventually lead to proof obligations matching a previous statement. These matching statements marked in blue can be backlinked to create a pre-proof. Another check is required to prove infinite descent.

link from this leaf to the root? Not in general. Such proofs are cyclic and cyclic proofs are unsound in general. However, if we can prove that for any concrete stack heap pair and values for logical variables, this cyclic *pre-proof* is finite, we prove the entailment. In some sense, these *cyclic proofs* are templates which we can use to instantiate any model to obtain a proper *finite proof tree*.

Checking whether a pre-proof is a cyclic proof requires us to check that every cycle in the proof is *infinitely descending*. Different proof systems may use different metrics to measure the infinite descent. For qualitative separation logic, unfoldings on the left side of the entailment yield descents [98]. For quantitative separation logic, it seems difficult to prove or disprove the soundness of this method. Instead we use heap sizes as the metric for descent, which for quantitative separation logic is usually sufficient.

9.1. Recursively-Defined Quantitative Separation Logic Predicates

Similarly how we defined a syntax for FSL in Section 8.3, we will also define a syntax for QSL in order to allow syntactic reasoning about sequents of formulae in a proof tree. The syntax for this chapter is more limited compared to the syntax in Section 8.3 in order to guarantee monotonicity of the characteristic functions that are induced by the equation defining a predicate. Moreover, since our main focus in this chapter will be the soundness criterion for infinite descent in order to check whether a pre-proof is a cyclic proof, we will limit ourselves to a simple syntax.

A predicate P has a fixed number of parameters. We denote this fixed number as $|P|$. For now, we assume predicates are not recursively defined. Instead the semantics of predicates is assumed to be given. Later, we will introduce recursive definitions and define their semantics.

Definition 9.1.1 (Syntax for Quantitative Separation Logic) *For a set of predicate symbols $\mathcal{P}$, quantitative separation logic formulae are generated by the following grammar:*

$$
\begin{array}{rcll}
f & ::= & [k = o] & \textit{equality} \\
 & | & [k \neq o] & \textit{inequality} \\
 & | & [k \mapsto o] & \textit{points to} \\
 & | & [\mathsf{emp}] & \textit{empty heap} \\
 & | & \langle b \rangle & \textit{rational number} \\
 & | & f \sqcup f & \textit{maximum} \\
 & | & f + f & \textit{addition} \\
 & | & f \cdot f & \textit{multiplication} \\
 & | & \mathfrak{S} a.\, f & \textit{supremum quantifier} \\
 & | & f \star f & \textit{separating multiplication} \\
 & | & P(k_1, \ldots, k_{|P|}) & \textit{predicate symbol}
\end{array}
$$

for variables $k, o, k_1, \ldots, k_n \in \mathsf{Vars} \cup \mathsf{LVars}$, $a \in \mathsf{LVars}$, non-negative rational numbers $b \in \mathbb{Q}_{\geq 0}$ and predicate symbols $P \in \mathcal{P}$.

We define the semantics for this syntax as the straightforward mapping to the operations introduced in Chapter 4. We indeed require to do this here in order to argue both for the necessity of our different soundness criterion, but also to prove the correctness of our soundness criterion.

We not only fix the allowed operations, but also the expressions. Real expressions are only allowed to be constant rationals and value expressions are only allowed to be either logical or program variables. We also use the notation

$$
k(s, \eta) \quad = \quad \begin{cases} s(k) & \text{if } k \in \mathsf{Vars} \\ \eta(k) & \text{if } k \in \mathsf{LVars} \end{cases}
$$

for $k \in \mathsf{Vars} \cup \mathsf{LVars}$ to avoid writing conditionals on the type of k every time we evaluate k.

We will use in this chapter substitutions on formulae $f\,[k := o]$. We mean here a syntactic substitution, that is changing every free occurrence of k in f by o.

The semantics of predicates are parameterized random variables. A parameterized random variable with n parameters maps n values to a random variable.

Definition 9.1.2 (Parameterized Random Variable) *Let $n \in \mathbb{N}$ be a natural number. A parameterized random variable with n parameters is a map $X\colon \mathbb{Z}^n \to \mathsf{RV}^\infty$ from n values to a random variable.*

For now, we define the semantics of (recursively-defined) predicates as given. We represent the given semantics as a finite indexed sequence. An indexed sequence $(X_P)_{P \in \mathcal{P}}$ maps a predicate symbol P to a parameterized random variable $X_P\colon \mathbb{Z}^{|P|} \to \mathsf{RV}^\infty$ with correct parameter count.

$$
\begin{aligned}
[\![[k = o]]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \text{if } k(s,\eta) = o(s,\eta) \text{ then } 1 \text{ else } 0 \\
[\![[k \neq o]]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \text{if } k(s,\eta) \neq o(s,\eta) \text{ then } 1 \text{ else } 0 \\
[\![[k \mapsto o]]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \text{if } h = \{\, k(s,\eta) \mapsto o(s,\eta) \,\} \text{ then } 1 \text{ else } 0 \\
[\![[\mathsf{emp}]]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \text{if } h = \emptyset \text{ then } 1 \text{ else } 0 \\
[\![\langle b \rangle]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= b \\
[\![g_1 \sqcup g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \max\left\{ [\![g_1]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta),\ [\![g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) \right\} \\
[\![g_1 + g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= [\![g_1]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) + [\![g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) \\
[\![g_1 \cdot g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= [\![g_1]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) \cdot [\![g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) \\
[\![\text{\textsf{S}} a.\, g]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \sup\left\{ [\![g]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta\,[a := v]) \,\middle|\, v \in \mathbb{Z} \right\} \\
[\![g_1 \star g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= \sup\left\{ [\![g_1]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h_1, \eta) \cdot [\![g_2]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h_2, \eta) \,\middle|\, \begin{array}{c} h_1 \perp h_2, \\ h = h_1 \cup h_2 \end{array} \right\} \\
[\![Q(k_1, \ldots, k_{|Q|})]\!]_{(X_P)_{P \in \mathscr{P}}}(s, h, \eta) &= X_Q(k_1(s,\eta), \ldots, k_{|Q|}(s,\eta))(s, h, \eta).
\end{aligned}
$$

Figure 9.2.: Semantics for the syntax defined in Definition 9.1.1 without recursive definitions. Instead the semantics of undefined predicate are given when applying the semantics. Otherwise the semantics matches the operations from Table 4.1.

[99] chooses a slightly different semantics to ease automatization.

Definition 9.1.3 (Semantics of Quantitative Separation Logic) *For an indexed sequence of predicate semantics $(X_P)_{P \in \mathscr{P}}$, we define the semantics $[\![f]\!]_{(X_P)_{P \in \mathscr{P}}}$ of a formula f from Definition 9.1.1 as in Figure 9.2.*

The semantics from Definition 9.1.3 match exactly how we defined the corresponding operations in Table 4.1 to avoid any confusing. Only predicates symbols are newly introduced. We represent the semantics here again as we now formally introduce a syntax.

Recursive definitions are equations where the left side is a predicate symbol P and the right side is a formula f from the syntax of Definition 9.1.1:

$$P(a_1, \ldots, a_{|P|}) \quad = \quad f.$$

Since the left side of a recursive definition is always a predicate symbol, these definitions give rise to characteristic functions. These functions happen to be monotone for the allowed operations (as we show in Theorem 9.1.1) and thus guarantee the existence of a fixed point. We will use the least fixed point of this characteristic function to define the semantics of recursively-defined predicates.

Literature usually allows multiple definitions for one predicate. However, this is here syntactic sugar.

Definition 9.1.4 (Recursive Definitions) *For formulae f_P from the syntax of Definition 9.1.1, we call the indexed tuple*

$$(P(a_1, \ldots, a_{|P|}) \;=\; f_P)_{P \in \mathscr{P}}$$

recursive definitions *for the predicates in $\mathscr{P}$.*

We repeat here the example from the introduction to give more details.

Example 9.1.1 A list may be empty. In that case head and tail correspond to the same (possibly null) value. If a list is not empty, we can follow the pointer from the head and have a tail list remaining. By the following recursive equation, we define a predicate $[\text{ls}]\,(a, c)$ such that it is one for a heap if and only if the heap consists of a list from head a to tail c:

$$[\text{ls}]\,(a,c) \;=\; [a = c] \sqcup \text{Ƨ}b.\; [a \mapsto b] \star [\text{ls}]\,(b,c).$$

Whenever the predicate occurs in a formula, we can *unfold* it. Unfolding means here that we replace an occurrence of the left side of the equation by the right side. The other direction is called a *folding*.
The predicate above is qualitative. It maps only to zero and one. We can also have quantitative predicates. These usually map to some size information, for example to the size of the heap. By the following recursive equation, we define a predicate $\text{ls-size}(a, c)$ such that it is the size of the list (i.e. the number of pointers) if the heap is indeed a list:

$$\text{ls-size}(a,c) \;=\; \text{Ƨ}b.\; [a \mapsto b] \star \text{ls-size}(b,c) + [a \mapsto b] \star [\text{ls}]\,(b,c).$$

Here we do not have two “options” but instead use that an empty list has size 0. Thus we only consider the case of non-empty lists. The size of a non-empty list is the size of the tail incremented by one if the heap is actually a list. Since we need to guarantee that the heap is actually a list, we increment by $[a \mapsto b] \star [\text{ls}]\,(b, c)$. This way we only increment if the heap is a list.

The recursive definitions give rise to the characteristic function which takes semantics for predicates $(X_P)_{P\in\mathscr{P}}$ and transforms it into new semantics for predicates $(Y_P)_{P\in\mathscr{P}}$. The new semantics are obtained by defining each predicate Q by its formula $[\![f_Q]\!]_{(X_P)_{P\in\mathscr{P}}}$, where we use $(X_P)_{P\in\mathscr{P}}$ as the semantics for predicate symbols in the formula. The variable parameters are overwritten with the given values.

Definition 9.1.5 (Characteristic Function of Recursive Definitions) *Let $\mathscr{D} = (P(a_1, \dots, a_{|P|}) = f_P)_{P\in\mathscr{P}}$ be recursive definitions for the predicates in $\mathscr{P}$. We define the characteristic function $\Gamma_{\mathscr{D}}$ of $\mathscr{D}$ as the function mapping semantics for predicates $(X_P)_{P\in\mathscr{P}}$, a predicate symbol $Q \in \mathscr{P}$ and values $v_1, \dots, v_{|Q|} \in \mathbb{Z}$ to the random variable*

$$\begin{aligned}&\Gamma_{\mathscr{D}}((X_P)_{P\in\mathscr{P}})_Q(v_1, \dots, v_{|Q|})\\ &= \lambda s, h, \eta.\, [\![f_Q]\!]_{(X_P)_{P\in\mathscr{P}}}(s, h, \eta\,[a_1 := v_1]\dots[a_{|Q|} := v_{|Q|}]).\end{aligned}$$

Monotonicity of the characteristic function guarantees the existence of fixed points. We thus proceed proving that the characteristic function is always monotone.

Theorem 9.1.1 *Let $\mathscr{D} = (P(a_1, \dots, a_{|P|}) = f_P)_{P\in\mathscr{P}}$ be recursive definitions for the predicates in $\mathscr{P}$. The characteristic function $\Gamma_{\mathscr{D}}$ is monotone.*

Proof. See [43] at `LeanFSL.Entailments.QSLSystem`. □

Example 9.1.2 An application of the characteristic function to semantics for predicate will "include one more unfolding" in the semantics. A fixed point of the characteristic function thus "includes all possible unfoldings". Assume we have the definitions from Example 9.1.1 and call them $\mathscr{D}$. We can demonstrate this by applying the characteristic functions to the semantics

$$X_{[\mathrm{ls}]}(v_1, v_2)(s,h,\eta) = 0 \text{ and } X_{\text{ls-size}}(v_1, v_2)(s,h,\eta) = 0.$$

We now have for the list predicate that

$$\begin{aligned} &\Gamma_{\mathscr{D}}((X_P)_{P\in\mathscr{P}})_{[\mathrm{ls}]}(v_0, v_1)(s,h,\eta) \\ &= [\![f_{[\mathrm{ls}]}]\!]_{(X_P)_{P\in\mathscr{P}}}(s,h,\eta\,[a_0 := v_0]\,[a_1 := v_1]) \\ &\quad\vdots \\ &= \begin{cases} 1 & \text{if } v_0 = v_1 \wedge h = \emptyset \\ 0 & \text{else} \end{cases} \end{aligned}$$

and for the list size predicate that

$$\begin{aligned} &\Gamma_{\mathscr{D}}((X_P)_{P\in\mathscr{P}})_{\text{ls-size}}(v_0, v_1)(s,h,\eta) \\ &= [\![f_{\text{ls-size}}]\!]_{(X_P)_{P\in\mathscr{P}}}(s,h,\eta\,[a_0 := v_0]\,[a_1 := v_1]) \\ &\quad\vdots \\ &= 0 \end{aligned}$$

Here we assume that the empty list is of size zero, therefore we do not add anything in the first step.

since for the first unfolding we do not add anything.
Applying the function to this result again, we obtain one more unfolding and thus have for the list predicate that

$$\begin{aligned} &\Gamma_{\mathscr{D}}(\Gamma_{\mathscr{D}}((X_P)_{P\in\mathscr{P}}))_{[\mathrm{ls}]}(v_0, v_1)(s,h,\eta) \\ &= [\![f_{[\mathrm{ls}]}]\!]_{(X_P)_{P\in\mathscr{P}}}(s,h,\eta\,[a_0 := v_0]\,[a_1 := v_1]) \\ &\quad\vdots \\ &= \begin{cases} 1 & \text{if } (v_1 = v_2 \wedge h = \emptyset) \vee h = \{\, v_0 \mapsto v_1 \,\} \\ 0 & \text{else} \end{cases} \end{aligned}$$

and this time the list size changes due to the new value of $X_{[\mathrm{ls}]}$ after the first iteration to

$$\begin{aligned} &\Gamma_{\mathscr{D}}(\Gamma_{\mathscr{D}}((X_{\text{ls-size}})))(v_0, v_1)(s,h,\eta) \\ &= [\![f_{\text{ls-size}}]\!]_{(X_P)_{P\in\mathscr{P}}}(s,h,\eta\,[a_0 := v_0]\,[a_1 := v_1]) \\ &\quad\vdots \\ &= \begin{cases} 1 & \text{if } h = \{\, v_0 \mapsto v_1 \,\} \\ 0 & \text{else} \end{cases} \end{aligned}$$

to reflect that the list containing one pointer has length 1.
Further applying the function to the initial value will increase the value of the semantics. A fixed point satisfies now the equation system of $\mathscr{D}$ and allows arbitrary many unfoldings.

Since the characteristic function $\Gamma_{\mathscr{D}}$ is monotone, it also has a least fixed point according to Theorem 2.2.1. We use the least fixed point to define the semantics

of recursively-defined predicates.

Definition 9.1.6 (Semantics of Quantitative Separation Logic with Recursive Definitions) *Let* $\mathscr{D} = (P(a_0, \ldots, a_{|P|}) = f_P)_{P \in \mathscr{P}}$ *be recursive definitions for the predicates in* $\mathscr{P}$ *and g a formula obeying the syntax of Definition 9.1.1. We define the semantics with recursive definitions of g as*

$$[\![g]\!] = [\![g]\!]_{\mathsf{lfp}(\Gamma_{\mathscr{D}})}.$$

We use lfp to refer to the least fixed point of some function.

9.2. Cyclic Proofs

Reasoning about recursively-defined predicates requires the reasoner to use techniques to argue about structures of *arbitrary size.* This is not surprising as we want to use recursively-defined predicates to model data structures of arbitrary size in the first place! The classical approach is using complete induction. When we apply induction, we have to guess a potentially weaker induction hypothesis from which we can show the entailment at hand. Then we show that the induction hypothesis holds for base cases and progressing cases. In progressing cases, we usually have the induction hypothesis given for certain sub-structures. Guessing an induction hypothesis is, however, hard.

[98] proposes a different solution. Instead of guessing an induction hypothesis, we allow proofs to be *cyclic.* A cyclic proof can reason about structures of arbitrary size by picking any previous node in the proof tree as a proof. In some sense, we can *choose* an hypothesis from all *previous proof states.* Cyclic proofs are in general *unsound* — we could just start with a statement A and argue that A holds because A holds. Thus cyclic proofs require a special soundness criterion that makes a pre-proof an actual proof.

Proofs itself are composed of *inference rules* which transform one proof obligation into possibly more (easier) proof obligations or into no proof obligation. The latter are axioms and thus finish a proof. With inference rules we can construct a derivation tree. By allowing *backlinks* that connect a leave of this tree to other nodes, the proof "tree" is instead a "graph" in case there are backlinks inducing cycles. We call these graphs *pre-proofs.* Proof obligations are called sequents, which may be arbitrary chosen for this framework. Usually the set of sequents are chosen to fit the purpose of the proof system. Classic choices for sequents are logical entailments.

Definition 9.2.1 (Inference Rules and Pre-Proofs) *Let* $\mathscr{S}$ *be a set of sequents. An* inference rule $R\colon \mathscr{S} \times \mathscr{S}^*$ *transforms a sequent into a (possibly empty) sequence of other sequents. We call the first sequent the conclusion and the other sequents premises. We denote the set of inference rules as* $\mathscr{R}$*. We denote an inference rule* $R = (S, S_1, \ldots, S_n)$ *using inference notation as*

$$\frac{S_1, \ldots, S_n}{S}\ R$$

for a natural number $n \in \mathbb{N}$*.*
A pre-proof *is a directed tree* $(V, E, Rule, Label, Link)$ *for a finite set of vertices* V *equipped for some* $V' \subseteq V$ *with edges* $E\colon V \setminus V' \to V^*$*, a map for the used rules* $Rule\colon V \setminus V' \to \mathscr{R}$*, a labeling function* $Label\colon V \to \mathscr{S}$ *and a backlinking*

We use here a non-standard definition for edges to make it easier to reason about inference rules used in the pre-proof

function Link: $V' \to V$ *such that*

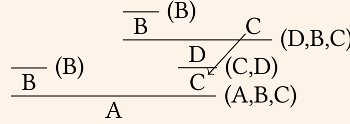
- *for* $v \in V \setminus V'$ *we have* $Rule(v) = (S, S_1, \dots, S_n)$ *and* $E(v) = (v_1, \dots, v_n)$ *such that* $Label(v) = S$, $Label(v_1) = S_1, \dots, Label(v_n) = S_n$, *and*
- *for* $v \in V'$ *we have* $Label(v) = Label(Link(v))$.

Example 9.2.1 Let the sequents be $\mathscr{S} = \{A, B, C, D, E\}$ and the inference rules be $\mathscr{R} = \{(A, B, C), (B), (C, D), (D, B, C), (E, A, E)\}$. We remark that the inference rule (E, A, E) looks nonsensical. It requires us to prove E to prove E. However, if the inference rule allows *progress*, it still can be useful in a cyclic proof! In a later example, E will however be invalid to avoid making this example trivial. The pre-proof $(V, E, Rule, Label, Link)$ is defined as

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$, $V' = \{v_5\}$,
- $E(v_0) = (v_1, v_2)$, $E(v_1) = ()$, $E(v_2) = (v_3)$, $E(v_3) = (v_4, v_5)$, $E(v_4) = ()$,
- $Rule(v_0) = (A, B, C)$, $Rule(v_1) = (B)$, $Rule(v_2) = (C, D)$, $Rule(v_3) = (D, B, C)$, $Rule(v_4) = (B)$,
- $Label(v_0) = A$, $Label(v_1) = B$, $Label(v_2) = C$, $Label(v_3) = D$, $Label(v_4) = B$, $Label(v_5) = C$, and
- $Link(v_5) = v_2$,

We will later use names instead of tuples to denote rules.

and is visually represented as



In the visual representation, we leave out names for the vertices and instead only use labels to denote them. Edges in the tree are represented as inference lines. The rule of a vertex is denoted next to inference line above it. Backlinks (i.e. the one between the two C labeled vertices) are represented with an arrow.

A pre-proof does not necessarily prove the sequent in the root, as pre-proofs allow unsound cyclic reasoning. To guarantee soundness, we have to impose a soundness criterion. This soundness criterion requires that every infinite path has a tail with an infinite descending (or *progressing*) trace. We map an edge in the tree to a set of pairs of trace values.

Classically, trace values are used to keep track of different progress conditions concurrently. When progress is achieved by unfoldings, we need to keep track which predicate was unfolded. Every trace value is identifying one predicate and the progress of this predicate in this setting. Since we will not use predicate unfoldings, we will only have a single trace value.

A trace is a (possibly infinite) path $(t_i)_{i \in \mathbb{N}}$ through a pre-proof such that the trace values t_i of every node and the trace value t_{i+1} of a successor of that node is included in the set of trace value pairs that we associate to that edge. These trace values are then further associated with a value of a *well order*. A well order guarantees that infinite descending chains do not exist. Thus we obtain a sequence of values of the well order, which we require to be a co-chain. If an associated value of the well order decreases strictly, the corresponding trace value pair is called decreasing or *progressing*. If there are infinitely many such progressing points on the trace, the trace is called *progressing*.

Requiring that a pre-proof has only infinite paths with infinitely many progressing points, corresponds to the idea that if we instantiate the sequent in the root with a fixed model, we could construct a proof tree from the pre-proof, that would be finite. In this sense, a cyclic proof is a template of proofs for each model. A model here is some object used to interpret truth values for sequents. In this dissertation, models are stack heap pairs with values for logical variables.

Definition 9.2.2 (Trace Values, Traces and Cyclic Proofs) *Let $(V, E, Rule, Label, Link)$ be a pre-proof and $\mathscr{T}$ set of trace values.*

- *We call $\delta\colon \mathscr{S}\times\mathscr{R}\times\mathscr{S} \to Pow(\mathscr{T}\times\mathscr{T}\times\{\text{true}, \text{false}\})$ a* trace pair function *where $\delta(S, R, S')$ is finite and computable.*
- *For an infinite path $(v_i)_{i\in\mathbb{N}}$ in the pre-proof, we call $(t_i)_{i\geq n}$ with $t_i \in \mathscr{T}$ for some $n \in \mathbb{N}$* a trace *following the path tail $(v_i)_{i\geq n}$ if for all $n \leq i$ we have either for some b that $(t_i, t_{i+1}, b) \in \delta(Label(v_i), Rule(v_i), Label(v_{i+1}))$ or $Link(v_i) = v_{i+1}$ and $t_i = t_{i+1}$. For $(t, t', b) \in \delta(Label(v_i), Rule(v_i), Label(v_{i+1}))$, we call (t, t') a trace pair and it is* progressing *if $b = \text{true}$.*
- *A* cyclic proof *is the tuple $(V, E, Rule, Label, Link, \delta)$ where for every infinite path $(v_i)_{i\in\mathbb{N}}$ in the pre-proof, there is a tail of the path $(v_i)_{i\geq n}$ such that there is a trace $(t_i)_{i\geq n}$ following the tail and for infinitely many distinct i we have that the trace pair (t_i, t_{i+1}) is progressing.*

Trace values are merely markers. They can be used for example to mark different predicates. Trace values are also not ordered.

A classical choice for trace values are labels for recursively-defined predicates. In the case where we model the semantics of predicates as sets, this works. Since we can approximate the least fixed point of the characteristic function $\Gamma_{\mathscr{D}}$ of the recursively-defined predicate as an ordinal-indexed sequence $\uparrow\Gamma_{\mathscr{D}}{}^{a}$ of semantics (compare Theorem 2.3.3), we can also ask for a certain state σ which ordinal a is the least such that $\sigma \in\uparrow\Gamma_{\mathscr{D}}{}^{a}$. In this setting, it turns out that unfolding the predicate once and applying changes to the state due to variable assignments, *decreases* the ordinal a' for which with the resulting state σ' we have $\sigma' \in\uparrow\Gamma_{\mathscr{D}}{}^{a'}$ and $a' < a$. The later result justifies calling the associated trace pair progressing.

To formulate soundness in the context of trace values, we define a criterion that the components of the system, i.e. the sequents, the inference rules and the trace pair function, together with an interpretation of the sequent and a mapping from trace values to ordinals need to satisfy. If they indeed satisfy this *global soundness criterion*, we can use the following theorem to guarantee that we only obtain valid sequents, i.e. sequents with valid interpretation, from the proof system. The proof of this theorem boils down to constructing infinite paths for a certain state in the cyclic proof. Since these infinite paths are infinite decreasing in a well order, we obtain a contradiction.

Definition 9.2.3 (Global Soundness Criterion) *Let $\mathscr{S}$ be a set of sequents, $\mathscr{R}$ be a set of inference rules and δ be a trace pair function for a set of trace values $\mathscr{T}$.*

- *The interpretation function $I\colon \text{States} \to Pow(\mathscr{S})$ maps states to a set of sequents.*
- *The ordinal trace function $\rho\colon \text{States}\times\mathscr{T} \to \text{Ordinal}$ maps states and trace values to an ordinal.*
- *The global soundness criterion is satisfied for $\mathscr{S}$, $\mathscr{R}$, δ, I and ρ if for any*

Using a well order or ordinals is equivalent as every well order is contained in the ordinals.

state $\sigma \in \text{States}$, *sequent* $S \in \mathscr{S}$ *and inference rule* $(S, S_1, \dots S_n) = R \in \mathscr{R}$ *if* $S \notin I(\sigma)$ *then there exists* $1 \leq i \leq n$ *and* $\sigma' \in \text{States}$ *such that*

- $S_i \notin I(\sigma')$,
- *if* $(t, t', \text{true}) \in \delta(S, R, S_i)$ *then* $\rho(\sigma', t') < \rho(\sigma, t)$, *and*
- *if* $(t, t', \text{false}) \in \delta(S, R, S_i)$ *then* $\rho(\sigma', t') \leq \rho(\sigma, t)$.

Theorem 9.2.1 (Correctness of the Global Soundness Criterion) *Let* $\mathscr{S}$ *be a set of sequents,* $\mathscr{R}$ *be a set of inference rules,* δ *be a trace pair function,* I *be an interpretation function and* ρ *be an ordinal trace function such that they satisfy the global soundness criterion. Further let* $(V, E, \mathit{Rule}, \mathit{Label}, \mathit{Link}, \delta)$ *be a cyclic proof for* $\mathscr{S}$ *and* $\mathscr{R}$*. Then we have for the root of the cyclic proof* v *that for all* $\sigma \in \text{States}$ *it is* $\mathit{Label}(v) \in I(\sigma)$.

A formalization of this proof as well as the related framework in Lean was unfortunately out of scope for this dissertation.

Proof. For a contradiction we assume that for the root v we have that $\mathit{Label}(v) \notin I(\sigma)$. Due to the global soundness criterion, we can construct a path $(v_i)_{i \in \mathbb{N}}$ where v_0 is the root and states $(\sigma_i)_{i \in \mathbb{N}}$ such that $\mathit{Label}(v_i) \notin I(\sigma_i)$ for all i. Since in a cyclic proof every infinite path $(v_i)_{i \in \mathbb{N}}$ has a tail $(v_i)_{i \geq n}$ with a trace $(t_i)_{i \geq n}$ that is infinitely progressing, we can apply again the global soundness criterion to obtain an ordinal trace $(\rho(\sigma_i, t_i))_{i \geq n}$ such that $\rho(\sigma_{i+1}, t_{i+1}) \leq \rho(\sigma_i, t_i)$ for all $i \geq n$ and for infinitely many i we also have $\rho(\sigma_{i+1}, t_{i+1}) < \rho(\sigma_i, t_i)$. This however contradicts that the ordinals are a well order. □

Example 9.2.2 We reconsider Example 9.2.1 and use the pre-proof

$$\dfrac{\dfrac{}{B}\,(B) \qquad \dfrac{\dfrac{\dfrac{}{B}\,(B) \quad C}{D}\,(D,B,C)}{C}\,(C,D)}{A}\,(A,B,C)$$

For sake of our example, we choose the interpretation $I(x) = \{A, B, C, D\}$ for the set of states $\text{States} = \mathbb{N}$. For this interpretation, E is unsatisfiable and all other sequents are valid. Knowing this interpretation, proving that A is valid, is of course trivial. However, usually I is not computable.
We use as trace values the singleton set $\mathscr{T} = \{t\}$ and the trace pair function δ defined as

$$\delta(S, R, S') = \begin{cases} (t, t, \text{true}) & \text{if } R = (D, B, C) \land S' = C \\ (t, t, \text{false}) & \text{if } R = (C, D) \\ \emptyset & \text{else.} \end{cases}$$

Since the only cycle lays on C, D, C, we only need trace pairs on these edges. We do not need trace pairs for backlinks as a valid cyclic proofs guarantees the equality of trace values. The respective ordinal function also must map to equal values as we adopt the state and have guaranteed equal trace values. We now observe that the pre-proof is indeed a cyclic proof as the only infinite path $A, C, D, C, \dots$ has the tail $C, D, C, \dots$ with a trace which is progressing from D to C.
Next we need an ordinal trace function ρ that we define as $\rho(n, t) = n$ with n then interpreted as the nth ordinal.
Checking the global soundness criterion is now trivial. We only have $S \notin I(\sigma)$ for $S = F$, thus with $R = (F, A, F)$, there exists state σ and the sequent F

such that $F \notin I(\sigma)$ and $\delta(F, (F, A, F), F) = \emptyset$, successfully proving the global soundness criterion.
The proof of the global soundness criterion may seem too easy. This is merely because the sequents are finite and the interpretation is easy. We will later prove this criterion for the previously introduced fragment of QSL, which follows a proof that is a bit more difficult.

9.3. Heap Size as Global Soundness Criterion

Classically, i.e. in Boolean logics, we define the trace pair function δ such that every instance of a predicate symbol on the left of an entailment has a unique trace value. If a symbol occurs multiple times, every occurrence receives a unique trace value. The trace pair is progressing when this predicate is unfolded. If the predicate is not unfolded but is also not removed from the sequent, we have a trace value pair that is not progressing for this predicate symbol. If the predicate symbol disappears (for whichever reason), the corresponding trace value also disappears. The soundness of this approach is proven using the ordinal approximations of least fixed points. More precisely, we say ρ maps a state σ and a trace value $t_{P(a_1,\dots a_{|P|})}$ to the smallest ordinal a with $\sigma \models \uparrow\Gamma_{\mathscr{D}}{}^{a}_{P}(a_1, \dots a_{|P|})$. For Boolean logics, we can interpret $\models$ as a set membership and obtain

This proof sketch is highly simplified. A complete and technical proof is given in [98].

$$\begin{aligned} & \sigma \in \uparrow\Gamma_{\mathscr{D}}{}^{a}_{P}(a_1, \dots a_{|P|}) \\ \iff & \sigma \in \bigcup_{b<a} \Gamma_{\mathscr{D}}(\uparrow\Gamma_{\mathscr{D}}{}^{b})_P(a_1, \dots a_{|P|}) \\ \iff & \exists b < a.\, \sigma \in \Gamma_{\mathscr{D}}(\uparrow\Gamma_{\mathscr{D}}{}^{b})_P(a_1, \dots a_{|P|}), \end{aligned}$$

where b is the smaller ordinal that the ordinal trace function maps to for the state and trace value for $P(\dots)$. We will not go into the details of this proof (and indeed, it is very technical). Instead we observe from the equivalences above a couple of complications:

1. What is the quantitative analogue of set membership? One may be tempted to say that the random variable has to evaluate to a certain value — be it one or infinity. However, there are ample predicates that may not evaluate to either? Answering this question remains open.
2. Secondly, even if we have a quantitative analogous operation for set membership, we also face the problem that

$$\begin{aligned} & \sigma \models \sup_{b<a} \Gamma_{\mathscr{D}}(\uparrow\Gamma_{\mathscr{D}}{}^{b})_P(a_1, \dots a_{|P|}) \\ \iff & \exists b < a.\, \sigma \models \Gamma_{\mathscr{D}}(\uparrow\Gamma_{\mathscr{D}}{}^{b})_P(a_1, \dots a_{|P|}), \end{aligned}$$

   may not hold (depending on the exact meaning of $\models$) in the reals as the supremum of the characteristic function can be a value not realized by any previous iteration (i.e. being a limit of a sequence of reals).

There is another way to see the problems of using the global soundness criterion based on unfolding predicates. That is, we can construct a proof for a valid entailment in QSL that would also be checked as sound by the proof system — however, it is not clear that the underlying ordinal trace function is actually progressing. For this we introduce the predicate ∞ that maps each state to

infinity. We define this predicate recursively as

$$\infty() = \infty() + \langle 1 \rangle.$$

Now we construct the pre-proof using the defined proof rules from Figures 9.3 and 9.4

$$\dfrac{\dfrac{\dfrac{\infty() \vdash \infty() + 1 \qquad \dfrac{}{1 \vdash 1}\text{ id}}{\infty() + 1 \vdash \infty() + 1 + 1}\text{ addition}}{\infty() + 1 \vdash \infty() + 1}\text{ unfold-r}}{\infty() \vdash \infty() + 1}\text{ unfold-l}$$

which is also a cyclic proof according to the trace pair function that progresses whenever an unfolding happens on a left-hand side of the entailment for the respective trace value's predicate. Now we go through the soundness proof of Theorem 9.2.1 to investigate the problem occurring. Considering an arbitrary state σ (for which we assume that $[\![\infty()]\!](\sigma) > [\![\infty()+1]\!](\sigma)$), we have $\rho(t_\infty, \sigma) = a$ for some ordinal a. However, neither t_∞ nor σ is "manipulated" by any of the inference rules. Instead, the inference rules only "manipulate" the *evaluation* of the state. As such, in a naïve approach, we assign the same ordinal a to any vertex on the cycle. But then we do not obtain an infinitely progressing chain.

We quote "manipulated" as inference rules formally do not manipulate the state. In the soundness proof of the rule, we manipulate states. As such we need to prove that there is no soundness proof in which the state is manipulated.

This is unfortunately not a formal counter-example, as we did not prove that there is no choice of σ' for which we have infinitely decreasing ordinals. Proving that there is indeed no such state σ' is an open problem.

We will not further investigate the problems of the classic global soundness criterion and instead propose a new one that is especially designed to work well for separation logics. This new global soundness criterion uses the size of the heap as ordinals. Frame rules allow decreasing the size of a heap and thus yield progressing trace pairs.

We will not aim for a cyclic proof system that is especially powerful or especially performant. Instead we will consider a toy proof system powerful enough to prove the examples we give and concentrate on the proposed global soundness criterion as the main contribution. Sequents in this proof system are syntactic entailments. To differentiate them from semantic entailments, we write $f \vdash g$ for syntactic entailments between the formulae f and g. We will depict inference rules as families of inference rules as shown in Figures 9.3 and 9.4. We highlight that these are indeed families as formulae in sequents are not allowed to have placeholders for formulae or non-formulae side-conditions. Checking the side-conditions is thus a task for the automated theorem prover generating a valid proof. The interpretation of syntactic entailments $I(s, h, \eta) = \{ f \vdash g \mid [\![f]\!](s, h, \eta) \leq [\![g]\!](s, h, \eta) \}$ is the entailment between the random variables obtained from applying the semantics.

We refer to [100] for a proof system with this aim.

For the soundness theorem of the inference rules of Figures 9.3 and 9.4, we are still missing a trace pair function, an ordinal trace function and the proof connecting these. As mentioned earlier, we will use decreasing heap sizes as our definition of progress. For this we need to assume that heaps are always finite. This matches nicely our assumption in Chapter 4, where we also have to require finite heaps for sound axiomatic program semantics.

$$\frac{\mathsf{S}a.\, f_P\,[a_1 := k_1]\ldots[a_{|P|} := k_{|P|}] \star f + f' \vdash g}{\mathsf{S}a.\, P(k_1, \ldots k_{|P|}) \star f + f' \vdash g}\ \text{unfold-l}$$

$$\frac{f \vdash \mathsf{S}a.\, f_P\,[a_1 := k_1]\ldots[a_{|P|} := k_{|P|}] \star g + g'}{f \vdash \mathsf{S}a.\, P(k_1, \ldots k_{|P|}) \star g + g'}\ \text{unfold-r}$$

$$\frac{f\,[a := k] \vdash g \qquad k \text{ not free in } f \text{ and } g}{\mathsf{S}a.\, f \vdash g}\ \text{sup-l} \qquad \frac{f \vdash g\,[a := k]}{f \vdash \mathsf{S}a.\, g}\ \text{sup-r}$$

$$\frac{f\,[k := o] \vdash g\,[k := o]}{([\mathsf{emp}] \cdot [k = o]) \star f \vdash g}\ \text{equal-sub} \qquad \frac{f \vdash g}{f \vdash ([\mathsf{emp}] \cdot [o = o]) \star g}\ \text{equal-triv-r}$$

$$\frac{\mathsf{S}a.\ [k \mapsto o] \star (f_1 + f_2) \vdash g}{\mathsf{S}a.\ [k \mapsto o] \star f_1 + [k \mapsto o] \star f_1 \vdash g}\ \text{pt-dist-l} \qquad \frac{f_3 \star f_1 + f_3 \star f_1 \vdash g}{f_3 \star (f_1 + f_2) \vdash g}\ \text{subdist-l}$$

$$\frac{f \vdash \mathsf{S}a.\ [k \mapsto o] \star (g_1 + g_2)}{f \vdash \mathsf{S}a.\ [k \mapsto o] \star g_1 + [k \mapsto o] \star g_2}\ \text{pt-dist-r} \qquad \frac{f \vdash ([\mathsf{emp}] \cdot [o = u]) \star g}{[k \mapsto o] \star f \vdash [k \mapsto u] \star g}\ \text{frame}$$

$$\frac{f\,[k := o] \vdash g\,[k := o] \qquad o \text{ not in } f \text{ and } g}{f \vdash g}\ \text{rename}$$

Figure 9.3.: First part of the proof system used to prove entailments in quantitative separation logic. We use this proof system to show how to use heap sizes as a global soundness criterion. Unfold rules replace an occurrence of a predicate symbol by its definition and substitute the parameter variables in the definition by the actual parameter values. Variables bound by a supremum on the left can be eliminated by using an unused variable and replacing the bound variable by this new variable. Supremum on the right can be eliminated by replacing the bound variable by any value. Equalities on the left allow substituting all occurrences of one part of the equation by the other. Since the points-to predicate is precise, we have distributivity for it, allowing the pt-dist-l and pt-dist-r rules. We can also use sub-distributivity on the left. Framing allows removing outer points-to predicates with same locations and yields a proof obligation stating that the values pointed to are equal. Renaming allows replacing one variable by a new fresh one. This is usually used for backlinking.

$$\frac{f_1 \vdash g_1 \qquad f_2 \vdash g_2}{f_1 + f_2 \vdash g_1 + g_2}\ \text{add} \qquad \frac{f_1 \vdash g_1 \qquad f_2 \vdash g_2}{f_1 \cdot f_2 \vdash g_1 \cdot g_2}\ \text{mul}$$

$$\frac{f \vdash g_1 \star g_3}{f \vdash (g_1 \sqcup g_2) \star g_3}\ \text{max-r1} \qquad \frac{f \vdash g_2 \star g_3}{f \vdash (g_1 \sqcup g_2) \star g_3}\ \text{max-r2}$$

$$\frac{f_1 \star f_3 \vdash g \qquad f_2 \star f_3 \vdash g}{(f_1 \sqcup f_2) \star f_3 \vdash g}\ \text{max-l} \qquad \frac{}{f \vdash f + g}\ \text{sub-id} \qquad \frac{}{f \vdash f}\ \text{id}$$

Figure 9.4.: Second part of the proof system used to prove entailments in quantitative separation logic. For addition and multiplication we allow using monotonicity. Maximum on the left requires us to prove the entailment for both options, on the right it allows us to choose one. The sub-identity axiom requires the right side to have more summands. In case the left and right are identical, we can use the identity axiom.

Definition 9.3.1 (Heap Size) *The size of a heap* $h \in \mathsf{Heaps}$ *is*

$$|h| \;=\; |\,\{\,\ell \mid h(\ell) \neq \mathit{undef}\,\}\,|.$$

We assume $|h|$ *to be finite.*

Since we are generally only interested in the state that is applied to the entailment, we do not need to differentiate between trace values. We thus use as trace value a constant t. The ordinal trace function maps this constant trace value and a state s, h, η to the ordinal corresponding to the size of the heap $|h|$. Our ordinal trace function thus only maps to natural numbers instead of the whole collection of ordinals. In our inference rules from Figures 9.3 and 9.4 there is only one inference rule that decreases the heap size (namely the frame rule) and all other rules preserve the heap size. We thus obtain that $\delta(S, R, S') = \{\,(t, t, \mathsf{true})\,\}$ if R has S as the conclusion, S' is a premise and R is (an instance of) the framing rule and that $\delta(S, R, S') = \{\,(t, t, \mathsf{false})\,\}$ if R has S as the conclusion, S' as a premise and R is not (an instance of) the framing rule.

Definition 9.3.2 (Cyclic Proof System for Quantitative Separation Logic) *We define* the cyclic proof system for quantitative separation logic *as*

- $\mathscr{S} = \{\, f \vdash g \mid f, g$ *formulae following the syntax of Definition 9.1.1*$\}$,
- $\mathscr{R}$ = *see Figures 9.3 and 9.4, where non-sequent premises are side-conditions,*
- $\mathscr{T} = \{\, t \,\}$ *for some constant* t,
- $\delta(S, R, S') = \begin{cases} \{\,(t, t, \mathsf{true})\,\} & R = (S, \dots, S', \dots) \wedge R \textit{ is framing,} \\ \{\,(t, t, \mathsf{false})\,\} & R = (S, \dots, S', \dots) \wedge R \textit{ is not framing,} \\ \emptyset & \textit{else,} \end{cases}$
- $I(s, h, \eta) = \{\, f \vdash g \mid [\![f]\!](s, h, \eta) \leq [\![g]\!](s, h, \eta) \,\}$, *and*
- $\rho((s, h, \eta), t) = |h|$.

We guard some of the inference rules by suprema, additions and separating multiplications as it may not always be possible to eliminate all guarding operations before applying the inference rule. Unfold-rules are such an example. We can unfold a predicate symbol by replacing it with its definition together with substitutions. Since this may reveal equations or points-to predicate, which may be necessary for other inference rules, we may not be able to eliminate other operations before an unfold. Since an unfold is rewriting the formula by an equation, we can generally apply unfolding anywhere. The inference rule depicted here, however, is limited to a certain kind of form that is common.

Instantiation rules for suprema differ on the left and on the right. Suprema on the left can be instantiated by replacing the bound variable with an unused fresh variable. On the right, we can eliminate the supremum by replacing it with any variable — unused or not.

Equalities on the left allow substitution of one side of the equality by the other. We require the equality to be guarded by an empty heap predicate to allow eliminating the predicate. The equal-sub inference rule allows us this reasoning step. For trivial equalities we allow elimination also on the right side of the sequent.

Since points-to is a precise predicate, we have distributivity with it for separating multiplication and addition (c.f. [31, Theorem 6.25]). This is reflected by the

inference rules pt-dist-l and pt-dist-r that allow applying distributivity in the given context. Again, we can apply distributivity anywhere, but the form in this inference rule is common. We remark here that we allow a, k and o to be identical. Moreover, we allow applying distributivity on the left for arbitrary formulae due to the subdistributivity of separating multiplication and addition (c.f. [31, Theorem 6.16]).

The inference rule for framing is a special case of the general framing rule. Here we only allow framing using points-to predicates. Indeed, for increased expressivity, more formulae should be allowed to frame. For our examples, framing points-to predicates is sufficient. When we frame a points-to predicate, we also know that the right part of the separating multiplication is receiving a smaller heap as the points-to predicate is only one if it exactly matches a heap part. Note that the word "framing" may be confusing as it differs from the framing rules used in Chapters 3 and 6. Instead, this framing argument uses monotonicity of the separating multiplication. Using our heap-size argument, we now can reason that the heap size is indeed decreasing applying the framing inference rule and thus we obtain progress.

In order to ease creating backlinks, we allow renaming variables to match exact sequents in different parts of the proof tree.

For addition and multiplication, we use their monotonicity.

Maxima behave different on the left and on the right of an entailment. On the left, they yield two new proof goals. On the right, we can pick which proof goal we want to prove. This is analogous to the behavior of disjunction in Boolean settings.

The sub-identity inference rule allows a bit more than identities. We ask whether the summands on the left are contained in the summands on the right, thus making the right a greater value — which is fine for our definition of entailment as a pointwise less-than-or-equal relation. The identity inference rule then also allows both sides to be equal.

Generally we assume ths proof system to be implemented such that associativity and commutativity is always allowed to be used. That is, when necessary summands or multiplicands are reordered to match the proof rules.

Showing that the system established in Definition 9.3.2 only allows cyclic proofs for valid sequents, requires us now to prove the global soundness criterion from Definition 9.2.3. Then we can use Theorem 9.2.1 to obtain that if $f \vdash g$ is provable in the cyclic proof system, then the entailment $[\![f]\!] \models [\![g]\!]$ holds.

Example 9.3.1 We reconsider the proof tree shown in Figure 9.1, but this time use rules to rigorously prove the claim. We will also consider all possible cycles to examine the progress condition further. We split the proof in three parts for better readability.
The first two proofs establish the sequent $[\mathrm{ls}]\,(a, y) \star [\mathrm{ls}]\,(y, z) \vdash [\mathrm{ls}]\,(a, z)$ which was left open with three dots in Figure 9.1. The first of the two proofs establishes a lemma used for the base case (i.e. an empty list) of the second proof. In the second, we unfold on both sides. For the unfold on the left, we obtain the base case † for which we use the first proof tree. The second case boils down to preparing the formula to apply framing, cleaning up and obtaining the same sequent we started with. Thus we obtain a pre-proof. The

$$\dfrac{\dfrac{}{[\mathrm{ls}]\,(a,z) \vdash [\mathrm{ls}]\,(a,z)}\ \text{id}}{[\mathrm{emp}] \cdot [a = y] \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z)}\ \text{equal-sub}$$

$$\dagger$$

$$\begin{array}{ll}
[\mathrm{ls}]\,(a,y) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z) & \text{(back-link to }\dagger\dagger\text{)} \\ \hline
[\mathrm{ls}]\,(b,y) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(b,z) & \text{rename} \\ \hline
[\mathrm{ls}]\,(b,y) \star [\mathrm{ls}]\,(y,z) \vdash ([\mathrm{emp}] \cdot [b = b]) \star [\mathrm{ls}]\,(b,z) & \text{equal-triv-r} \\ \hline
\text{—"—} \vdash [a \mapsto b] \star [a \mapsto b] \star [\mathrm{ls}]\,(b,z) & \text{frame} \\ \hline
\text{—"—} \vdash \mathfrak{S} b.\ [a \mapsto b] \star [a \mapsto b] \star [\mathrm{ls}]\,(b,z) & \text{sup-r} \\ \hline
\text{—"—} \vdash [\mathrm{emp}] \cdot [a = y] \sqcup \mathfrak{S} b.\ [a \mapsto b] \star [\mathrm{ls}]\,(b,z) & \text{max-r2} \\ \hline
[a \mapsto b] \star [\mathrm{ls}]\,(b,y) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z) & \text{unfold-r} \\ \hline
\dagger \qquad \mathfrak{S} b.\ [a \mapsto b] \star [\mathrm{ls}]\,(b,y) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z) & \text{sup-l} \\ \hline
([\mathrm{emp}] \cdot [a = y] \sqcup \mathfrak{S} b.\ [a \mapsto b] \star [\mathrm{ls}]\,(b,y)) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z) & \text{max-l} \\ \hline
[\mathrm{ls}]\,(a,y) \star [\mathrm{ls}]\,(y,z) \vdash [\mathrm{ls}]\,(a,z) & \text{unfold-l} \\ \hline
\dagger\dagger &
\end{array}$$

Figure 9.5.: Proof for the (qualitative) entailment that two disjoint lists sharing and end and a start node, respectively, form again a list. We use the proof system from Definition 9.3.2. This is used as a sub-proof for proof tree in Figure 9.6.

pre-proof is also a cyclic proof since on this cycle we apply the frame rule once.
The third proof is a proper version of the proof shown in Figure 9.1. Unfolding yields only one case. Since we define the size of the empty list as 0, we do not need to have an explicit base case. Similar to the previous proof, this proof also boils down to preparing both sides for the frame rule and cleaning up to obtain the same sequent which we started with. Again, the obtained cycle in the proof tree employs the frame rule once, as such we obtain a progressing path.

Theorem 9.3.1 (Global Soundness Criterion for QSL) *The cyclic proof system for quantitative separation logic from Definition 9.3.2 satisfies the global soundness criterion from Definition 9.2.3.*

Corollary 9.3.2 (Soundness of the Cyclic Proof System for QSL) *Whenever* $f \vdash g$ *is provable in the cyclic proof system for quantitative separation logic from Definition 9.3.2, we have that* $[\![f]\!] \vDash [\![g]\!]$.

Proof of Theorem 9.3.1. We have to prove the soundness criterion for every state s, h, η, sequent S and inference rule R. We now consider every possible inference rule individually.

Proof for unfold-l: Assuming that $\mathfrak{S} a.\, P(k_1, \ldots k_{|P|}) \star f + f' \vdash g \notin I(s,h,\eta)$, we obtain

$$(\mathfrak{S} a.\, [\![P(k_1, \ldots k_{|P|})]\!] \star [\![f]\!] + [\![f']\!])(s,h,\eta) \not\leq [\![g]\!]((s,h,\eta)).$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{ls-size}(x,y) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(x,z)}{\text{ls-size}(a,y) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(a,z)}\,\text{rename} \qquad \dagger\dagger}{\text{ls-size}(a,y) \star [\text{ls}]\,(y,z) + [\text{ls}]\,(a,y) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(a,z) + [\text{ls}]\,(a,z)}\,\text{add}}{[x \mapsto a] \star -"- \vdash [x \mapsto a] \star (\text{ls-size}(a,z) + [\text{ls}]\,(a,z))}\,\text{frame}}{[x \mapsto a] \star -"- \vdash [x \mapsto a] \star \text{ls-size}(a,z) + [x \mapsto a] \star [\text{ls}]\,(a,z)}\,\text{pt-dist-r}}{[x \mapsto a] \star -"- \vdash \text{Ƨ}a.\ [x \mapsto a] \star \text{ls-size}(a,z) + [x \mapsto a] \star [\text{ls}]\,(a,z)}\,\text{sup-r}}{[x \mapsto a] \star (\text{ls-size}(a,y) \star [\text{ls}]\,(y,z) + [\text{ls}]\,(a,y) \star [\text{ls}]\,(y,z)) \vdash \text{ls-size}(x,z)}\,\text{unfold-r}}{[x \mapsto a] \star \text{ls-size}(a,y) \star [\text{ls}]\,(y,z) + [x \mapsto a] \star [\text{ls}]\,(a,y) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(x,z)}\,\text{pt-dis-l}}{(\text{Ƨ}a.\ [x \mapsto a] \star \text{ls-size}(a,y) + [x \mapsto a] \star [\text{ls}]\,(a,y)) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(x,z)}\,\text{sup-l}}{\text{ls-size}(x,y) \star [\text{ls}]\,(y,z) \vdash \text{ls-size}(x,z)}\,\text{unfold-l}}{}$$

(An arrow connects the top sequent back to the bottom sequent.)

Figure 9.6.: Proof for the quantitative entailment that two lists, where they share a start and an end respectively, form again a list and the first list is smaller than both together. We use the proof system from Definition 9.3.2. At the ††, we place the proof-tree from Figure 9.5 as a sub-proof.

Using that $[\![P(k_1, \ldots k_{|P|})]\!]$ is defined as $\mathsf{lfp}(\Gamma_{\mathscr{D}})_P(k_1, \ldots k_{|P|})$, we can replace it with the fixed point and obtain

$$[\![P(k_1, \ldots k_{|P|})]\!](s,h,\eta) = \mathsf{lfp}(\Gamma_{\mathscr{D}})_P(s,h,\eta\,[a_0 := v_0] \ldots [a_{|P|} := v_{|P|}])$$

with $v_i = k_i(s,\eta)$ and further obtain

$$\begin{aligned} &\mathsf{lfp}(\Gamma_{\mathscr{D}})_P(s,h,\eta\,[a_0 := v_0] \ldots [x_{|P|} := v_{|P|}]) \\ =\ &[\![f_P]\!](s,h,\eta\,[a_0 := v_0] \ldots [a_{|P|} := v_{|P|}]). \end{aligned}$$

Finally we can rewrite this in the original entailment and have

$$(\text{Ƨ}a.\ [\![f_P]\!]\,[a_1 := k_1] \ldots [a_{|P|} := k_{|P|}] \star [\![f]\!] + [\![f']\!])(s,h,\eta) \not\preceq [\![g]\!](s,h,\eta),$$

which is equivalent to

$$[\![\text{Ƨ}a.\ f_P\,[a_1 := k_1] \ldots [a_{|P|} := k_{|P|}] \star f + f']\!](s,h,\eta) \not\preceq [\![g]\!](s,h,\eta).$$

Finally we obtain that

$$\text{Ƨ}a.\ f_P\,[a_1 := k_1] \ldots [a_{|P|} := k_{|P|}] \star f + f' \vdash g \notin I(s,h,\eta).$$

Moreover, we see that the heap is unchanged, therefore ρ is also unchanged.

Proof for unfold-l: Analogous to the proof for unfold-r.

Proof for sup-l: We assume that $\text{Ƨ}a.\ f \vdash g \notin I(s,h,\eta)$, thus we have that $[\![\text{Ƨ}a.\ f]\!](s,h,\eta) \not\preceq [\![g]\!](s,h,\eta)$. Further we consider the case that k is a logical variable. The case where k is a program variable is analogous. Then there exists a value v such that $[\![g]\!](s,h,\eta) < [\![f]\!](s,h,\eta\,[a := v])$, and expressed equivalently $[\![g]\!](s,h,\eta) < [\![f]\!](s,h,\eta\,[a := k(s,\eta)]\,[k := v])$ in case k is not a free variable in f and g. Thus finally we have $[\![g]\!](s,h,\eta') < [\![f\,[a := k]]\!](s,h,\eta')$ for some η' and thus also $f\,[a := k] \vdash g \notin I(s,h,\eta')$. Note that h is unchanged, thus there is no progress.

Proof for sup-r: We assume that $f \vdash \Sigma a.\, g \notin I(s,h,\eta)$, therefore $[\![f]\!](s,h,\eta) \nleq [\![\Sigma a.\, g]\!](s,h,\eta)$, or equivalently $[\![\Sigma a.\, g]\!](s,h,\eta) < [\![f]\!](s,h,\eta)$. Therefore we also have that $[\![g\,[a := k]]\!](s,h,\eta) = [\![g]\!](s,h,\eta\,[a := k(s,\eta)]) < [\![f]\!](s,h,\eta)$, as this makes the left side smaller. Finally we then have $f \vdash g\,[a := k] \notin I(s,h,\eta)$. As h is unchanged, there is no progress.

Proof for equal-sub: We assume that $([\mathsf{emp}] \cdot [k = o]) \star f \vdash g \notin I(s,h,\eta)$, thus we have $[\![([\mathsf{emp}] \cdot [k = o]) \star f]\!](s,h,\eta) \nleq [\![g]\!](s,h,\eta)$. If the left hand-side was zero, the inequality would hold, therefore we assume that the left hand-side is not zero. Then we have that $k(s,\eta)$ and $o(s,\eta)$ have the same value. Therefore we can substitute every occurrence of k by o in f and g and obtain $[\![([\mathsf{emp}] \cdot [k = o]) \star f\,[k := o]]\!](s,h,\eta) \nleq [\![g\,[k := o]]\!](s,h,\eta)$. Since the left hand-side of the separating multiplication is only one for the empty heap, the heap for the right hand-side is unchanged and thus we also have that $[\![f\,[k := o]]\!](s,h,\eta) \nleq [\![g\,[k := o]]\!](s,h,\eta)$ and therefore $f\,[k := o] \vdash g\,[k := o] \notin I(s,h,\eta)$. Since h is unchanged, there is no progress.

Proof for equal-triv-r: We assume that $f \vdash ([\mathsf{emp}] \cdot [o = o]) \star g \notin I(s,h,\eta)$, and thus have $[\![f]\!](s,h,\eta) \nleq [\![([\mathsf{emp}] \cdot [o = o]) \star g]\!](s,h,\eta)$. Since we have $[o = o]\,(s,h',\eta) = 1$ and $[\![[\mathsf{emp}] \cdot \langle 1 \rangle \star g]\!](s,h,\eta) = [\![g]\!](s,h,\eta)$, we also have that $[\![f]\!](s,h,\eta) \nleq [\![g]\!](s,h,\eta)$ and thus $f \vdash g \notin I(s,h,\eta)$. Since h remains unchanged, there is no progress.

Proof for pt-dist-l: We assume that

$$\Sigma a.\ [k \mapsto o] \star f_1 + [k \mapsto o] \star f_2 \vdash g \notin I(s,h,\eta).$$

Therefore we have that

$$[\![\Sigma a.\ [k \mapsto o] \star f_1 + [k \mapsto o] \star f_2]\!](s,h,\eta) \nleq [\![g]\!](s,h,\eta).$$

Using distributivity on the precise predicate $[k \mapsto o]$ (see [31, Theorem 6.25]) we have that $[\![\Sigma a.\ [k \mapsto o] \star (f_1 + f_2)]\!](s,h,\eta) \nleq [\![g]\!](s,h,\eta)$ and therefore $\Sigma a.\ [k \mapsto o] \star (f_1 + f_2) \vdash g \notin I(s,h,\eta)$. Since h is unchanged, there is no progress.

Proof for subdist-l: We assume that $f_3 \star (f_1 + f_2) \vdash g \notin I(s,h,\eta)$ and therefore we have that $[\![f_3 \star (f_1 + f_2)]\!](s,h,\eta) \nleq [\![g]\!](s,h,\eta)$. Equivalently we also have that $[\![g]\!](s,h,\eta) < [\![f_3 \star (f_1 + f_2)]\!](s,h,\eta)$. By sub-distributivity (see [31, Theorem 6.16]), we have that $[\![g]\!](s,h,\eta) < [\![f_3 \star (f_1 + f_2)]\!](s,h,\eta) \leq [\![f_3 \star f_1 + f_3 \star f_2]\!](s,h,\eta)$. Finally we have that $f_3 \star f_1 + f_3 \star f_1 \vdash g \notin I(s,h,\eta)$. Since h remains unchanged, there is no progress.

Proof for pt-dist-r: Analogous to the proof for pt-dist-l.

Proof for frame: We assume that $[k \mapsto o] \star f \vdash [k \mapsto u] \star g \notin I(s, h, \eta)$, thus we have that $[\![[k \mapsto o] \star f]\!](s, h, \eta) \not\leq [\![[k \mapsto u] \star g]\!](s, h, \eta)$. Equivalently, we have that $[\![[k \mapsto u] \star g]\!](s, h, \eta) < [\![[k \mapsto o] \star f]\!](s, h, \eta)$. If we do not have $h = h' \cup \{ k(s, \eta) \mapsto o(s, \eta) \}$, we would have that $[\![[k \mapsto o] \star f]\!](s, h, \eta)$ is zero. Then the strict inequality cannot hold. Therefore we assume this form for h. We also assume without loss of generality that $h \perp h'$. Furthermore, there is only one splitting of h that does not necessarily make either side zero, thus the splitting of the heap h into h' and $\{ k(s, \eta) \mapsto o(s, \eta) \}$ is taken in both separating multiplication. We obtain the strict inequality $1 \cdot [\![g]\!](s, h', \eta) < 1 \cdot [\![f]\!](s, h', \eta)$ if $u(s, \eta) = o(s, \eta)$ and $0 < 1 \cdot [\![f]\!](s, h', \eta)$ otherwise. We can combine the cases (and remove multiplication by the identity) again as the strict inequality $([\mathsf{emp}] \cdot [o = u]) \star [\![g]\!](s, h', \eta) < [\![f]\!](s, h', \eta)$. Finally we obtain $f \vdash ([\mathsf{emp}] \cdot [o = u]) \star g \notin I(s, h', \eta)$. Since $|h'| < |h|$, we have progress.

Proof for rename: We assume that $f \vdash g \notin I(s, h, \eta)$ and thus we also have that $[\![f]\!](s, h, \eta) \not\leq [\![g]\!](s, h, \eta)$. We assume that both k and o are program variables. For other combinations of program variables and logical variables, the program is analogous. Since o does not appear in either f or g, we also have that $[\![f]\!](s\,[k := s(o)]\,[o := s(k)]\,, h, \eta) \not\leq [\![g]\!](s\,[k := s(o), \eta]\,[o := s(k)]\,, h, \eta)$, which is equivalent to the statement that

$$[\![f\,[k := o]]\!](s\,[o := s(k)]\,, h, \eta) \not\leq [\![g\,[k := o]]\!](s\,[o := s(k)]\,, h, \eta)$$

and we further have $f\,[k := o] \vdash g\,[k := o] \notin I(s\,[o := s(k)]\,, h, \eta)$. Since h is unchanged, there is no progress.

Proof for add: We assume that $f_1 + f_2 \vdash g_1 + g_2 \notin I(s, h, \eta)$. Therefore we have that $[\![f_1 + f_2]\!](s, h, \eta) \not\leq [\![g_1 + g_2]\!](s, h, \eta)$. Using contraposition of monotonicity we have that either $[\![f_1]\!](s, h, \eta) \not\leq [\![g_1]\!](s, h, \eta)$ or $[\![f_2]\!](s, h, \eta) \not\leq [\![g_2]\!](s, h, \eta)$ and thus either $f_1 \vdash g_1 \notin I(s, h, \eta)$ or $f_2 \vdash g_2 \notin I(s, h, \eta)$. Since h is unchanged in both cases, there is no progress.

Proof for mul: Analogous to the proof for addition.

Proof for max-r1: We assume that $f \vdash (g_1 \sqcup g_2) \star g_3 \notin I(s, h, \eta)$ and thus we have that $[\![f]\!](s, h, \eta) \not\leq [\![(g_1 \sqcup g_2) \star g_3]\!](s, h, \eta)$. This means equivalently that $[\![(g_1 \sqcup g_2) \star g_3]\!](s, h, \eta) < [\![f]\!](s, h, \eta)$. Now we assume an arbitrary decomposition $h_1 \cup h_2 = h$ with $h_1 \perp h_2$ and obtain that $[\![g_1]\!](s, h_1, \eta) \cdot [\![g_3]\!](s, h_2, \eta) \leq [\![(g_1 \sqcup g_2)]\!](s, h_1, \eta) \cdot [\![g_3]\!](s, h_2, \eta) < [\![f]\!](s, h, \eta)$. Since we choose the splitting arbitrary, this holds for any splitting and thus we also have that $[\![g_1 \star g_2]\!](s, h, \eta) < [\![f]\!](s, h, \eta)$ and finally that $f \vdash g_1 \star g_3 \notin I(s, h, \eta)$. Since h is unchanged, there is no progress.

Proof for max-r2: Analogous to the proof for max-r1.

Proof for max-l: We assume that $(f_1 \sqcup f_2) \star f_3 \vdash g \notin I(s,h,\eta)$ and thus we have that $[\![(f_1 \sqcup f_2) \star f_3]\!](s,h,\eta) \not\leq [\![g]\!](s,h,\eta)$. Equivalently, we have that $[\![g]\!](s,h,\eta) < [\![(f_1 \sqcup f_2) \star f_3]\!](s,h,\eta)$. Thus we have for some $h_1 \cup h_2 = h$ that $[\![g]\!](s,h,\eta) < [\![(f_1 \sqcup f_2)]\!](s,h_1,\eta) \cdot [\![f_3]\!](s,h_2,\eta)$. Then we have that either $[\![g]\!](s,h,\eta) < [\![f_1]\!](s,h_1,\eta) \cdot [\![f_3]\!](s,h_2,\eta) \leq [\![f_1 \star f_3]\!](s,h,\eta)$ or $[\![g]\!](s,h,\eta) < [\![f_2]\!](s,h_1,\eta) \cdot [\![f_3]\!](s,h_2,\eta) \leq [\![f_2 \star f_3]\!](s,h,\eta)$. Finally we have then either $f_1 \star f_3 \vdash g \notin I(s,h,\eta)$ or $f_2 \star f_3 \vdash g \notin I(s,h,\eta)$. In either case h remains unchanged, thus there is no progress.

Proof for sub-id: We assume that $f \vdash f + g \notin I(s,h,\eta)$, and thus we have that $[\![f]\!](s,h,\eta) \not\leq [\![f+g]\!](s,h,\eta)$. However, we obviously have that $[\![f]\!](s,h,\eta) \leq [\![f]\!](s,h,\eta) + [\![g]\!](s,h,\eta)$. Since we have a contradiction, the assumption was wrong.

Proof for id: Analogous to the proof for sub-identity. □

9.4. Automatization

The proof system presented and proven soundness for in Definition 9.3.2 is not ideal for automatization. Many of the presented rules allow ambiguities, which introduce complexity in the automatization process. For example, on the left, we can always distribute in both directions when we use a points-to predicate (see pt-dist-l and subdist-l). Even worse, we require the program to guess a variable when we use a supremum on the right (see sup-r). A tool for automatization thus needs to use proof rules that guarantee an amount of choices that do not blow up. As such, in the master's thesis of Patrick Nossol [100], we investigate a proof system more adequate for automatization that applies these rules only when we can limit the possible choices.

Normal Form We introduce a normal form of random variables to allow for an efficient representation. This normal form always has the form $f = \mathsf{S} a_1 \ldots a_n.\ \sum_i \bigstar_j f_{i,j}$, that is, formulae consist of suprema over sums of separating multiplications on atoms $f_{i,j}$ that are either recursively-defined predicates, points-to predicates or (non-)equality predicates. We can introduce a limited number of other operations by certain elimination rules. For example, maxima can be eliminated using the max-r1, max-r2 and max-l rules. Multiplication of constants with the empty heap predicate can be eliminated in three steps:

Normalization does *not* guarantee that the values of the functions are natural as recursively-defined predicates can still have non-natural values.

1. we multiply every summand by a constant such that all appearing constants are natural,
2. we rewrite any $(\langle n+1 \rangle \cdot [\mathsf{emp}]) \star f$ as $(\langle n \rangle \cdot [\mathsf{emp}]) \star f + f$ (repeatedly), and
3. we rewrite any $(\langle 1 \rangle \cdot [\mathsf{emp}]) \star f$ as f.

This normalization also motivates the use of a different semantics for $\langle a \rangle$ than the one introduced in this dissertation. Instead we would define $\langle a \rangle$ as a for the empty heap and otherwise zero. We will stick with the semantics used in this dissertation for consistency reasons.

Table 9.1.: We depict runtimes, depth and number of backlinks of proofs for entailments in qualitative separation logic and compare the results of our proof system for quantitative separation logic with the results for the proof system for qualitative separation logic [101] in the theorem prover cyclist [102].

	Quantitative Cyclist			Classical Cyclist		
Entailment	Time (sec)	Depth	Backlinks	Time (sec)	Depth	Backlinks
List-List-le-List	0.01	2	1	0.02	3	1
List-RList-le-List	0.03	3	3	0.03	4	3
DLL-DLL-le-DLL	0.2	2	1	0.04	3	1
BTS-BTS-le-BTS	0.03	2	2	0.07	3	2
BTS-BT-le-BT	0.02	2	3	0.06	3	2

We will not go into further detail how the proof system in [100] avoids runtime blowups and instead focus on the hurdles that arise when developing such an automatization in a quantitative setting by means of experiments done on this implementation.

Experimental Setup We show the strengths and weaknesses of this implementation by various experiments we conducted. These experiments were run in parallel on a Debian Linux machine with two Intel Xeon Platinum 8160, 48 threads each of which has 2.1 GHz processor speed, and 384 GB RAM. The experiments timed out (marked with TO) when they ran for longer than 24 hours.

Qualitative Entailments As a first sanity check, we verify that we can still solve entailments in qualitative separation logic. For qualitative entailments, we do not require the sum. In these cases, the formulae degrade into the form $f = \mathcal{Z} a_1 \ldots a_n.\ \star_j f_j$. We depict five entailments involving a variety of recursively-defined predicates. The *List* predicate is defined analogous to [ls] from earlier. The *RList* predicate is defined from tail to head instead. The *DLL* predicate is a recursively-defined predicates for doubly linked lists. The predicates *BinTreeSeg* (abbreviated BTS) and *BinTree* (abbreviated BT) are both predicates defining some part of a binary tree. $\mathit{BinTreeSeg}(a, b)$ models a binary tree with root a, for which every leaf is null except for one which is b. $\mathit{BinTree}(a)$ instead is a binary tree with root a for which all leafs are null.

The entailments we use for the evaluation are:

$$\mathit{List}(x, y) \star \mathit{List}(y, z) \vdash \mathit{List}(x, z) \tag{List-List-le-List}$$

$$\mathit{List}(x, y) \star \mathit{RList}(y, z) \vdash \mathit{List}(x, z) \tag{List-RList-le-List}$$

$$\mathit{DLL}(x, y, z, w) \star \mathit{DLL}(a, x, w, b) \vdash \mathit{DLL}(a, y, z, b) \tag{DLL-DLL-le-DLL}$$

$$\mathit{BinTreeSeg}(x, y) \star \mathit{BinTreeSeg}(y, z) \vdash \mathit{BinTreeSeg}(x, z) \tag{BTSeg-BTSeg-le-BTSeg}$$

$$\mathit{BinTreeSeg}(x, y) \star \mathit{BinTree}(y) \vdash \mathit{BinTree}(x) \tag{BTSeg-BT-le-BT}$$

We observe in Table 9.1 that the proofs have similar computation time, depth and number of backlinks compared to the reference implementation for qualitative separation logic [101]. Thus we argue that the proofs by our proof system are similar.

The depth is always off by one due to some proof steps being combined.

Table 9.2.: We depict runtimes and depth of proofs for entailments in quantitative separation logic without recursively-defined predicates and compare the results of our proof system for quantitative separation logic with the decision procedure envisioned in [32, Section 3.4].

	Quantitative Cyclist		Decision Procedure from [32]
Entailment	Time (sec)	Depth	Time (sec)
add-conflict-le [31]	≤ 0.01	1	0.07
le-add-conflict [31]	≤ 0.01	1	0.06
le-add-zero [32]	0.02	1	0.07

Addition When allowing addition in formulae, we face two challenges:

1. The general addition rule in Figure 9.4 does not support any suprema in front of the addition to be split. And indeed, if there is a supremum on the right-hand side, this rule also only holds if the summands on the right do not share variables bound by the supremum. We thus only obtain the rule:

$$\frac{\mathfrak{S} a.\, f_1 \vdash \mathfrak{S} b.\, g_1 \qquad \mathfrak{S} a.\, f_2 \vdash \mathfrak{S} c.\, g_2 \qquad c \text{ not in } g_1 \qquad b \text{ not in } g_2}{\mathfrak{S} a.\, (f_1 + f_2) \vdash \mathfrak{S} b.\, \mathfrak{S} c.\, (g_1 + g_2)}$$

2. The second challenge arises with regard to runtime complexity. Since summation is associative, commutative and neutral towards $\langle 0 \rangle$, there are many ways to match summands on the left to summands on the right. Indeed, any map from summands of the left to summands on the right may be possible. In case the map is not an injection, we match the remaining summands on the right with $\langle 0 \rangle$ and have a trivial correct sequent. However, assuming there are a many summands on the left and b many summands on the right, we have b^a many choices to map summands to each other.

To handle both problems simultaneously, we choose to only split summands as late as possible. This has unfortunately also the problem that we lose access to easier proofs that use addition in an early step to simplify the proof. Especially when addition is used to connect different data structures, we may fail to apply the frame proof rule and thus cannot obtain a proof at all. Finding heuristics to make summation feasible in earlier steps in the proof is still an open problem.

We will later see this problems arising in the experiments.

Quantitative Entailments without Recursion Now equipped with addition, we apply another sanity check and see how we can handle quantitative entailments that do not contain recursively-defined predicates. We compare this with a prototype implementation of the decision procedure envisioned in [32, Section 3.4]. This decision procedure essentially uses the transformation from Chapter 8 and hands it over to CVC5, which is able to decide entailments in a fragment of qualitative separation logic [92].

The entailmens we use to compare our implementation with the decision proce-

dure from [32] are:

$$\langle 0.333\rangle \cdot [\mathrm{emp}] + (\langle 0.666\rangle \cdot [\mathrm{emp}]) \star [x \mapsto y] \star [x \mapsto z] \vdash \langle 0.333\rangle \cdot [\mathrm{emp}] \tag{add-conflict-le}$$

$$\langle 0.333\rangle \cdot [\mathrm{emp}] \vdash \langle 0.333\rangle \cdot [\mathrm{emp}] + (\langle 0.666\rangle \cdot [\mathrm{emp}]) \star [x \mapsto y] \star [x \mapsto z] \tag{le-add-conflict}$$

$$\begin{gathered}(\langle 0.98\rangle \cdot [\mathrm{emp}]) \star [x \mapsto z2] \star [y \mapsto z1] \\ \vdash (\langle 0.999\rangle \cdot [\mathrm{emp}]) \star [x \mapsto z2] \star [y \mapsto z1] + (\langle 0.001\rangle \cdot [\mathrm{emp}]) \star \langle 0\rangle\end{gathered} \tag{le-add-zero}$$

The first two are the same entailment with swapped sides to prove the equivalence. The proof boils down to the idea that $[x \mapsto y] \star [x \mapsto z]$ simplifies to zero as we cannot have two points-to assertions sharing the same location. The last one boils down to verifying that 0.98 is smaller than 0.999 and that we can remove the summand with 0. In Table 9.2 we depict runtimes for both methods and also depict the depth of the cyclic proof. Since we do not have any recursively-defined predicate, we have no cycles. The depth is not surprisingly always 1 as we immediately apply simple matching rules. Both methods handle the entailments with ease. We remark however that the size of the normal form for the entailment le-add-zero blows up as we require to expand the formulae to almost 1000 summands due to the weights. Furthermore, the implementation of the decision procedure from [32] writes to the file system to interact with CVC5 and thus has higher runtimes due to that.

Quantitative Entailments with Recursion Lastly we look at entailments that are both quantitative and involve recursively-defined predicates. The predicates used here evaluate to the sizes of data structures. *ListLen* and *RListLen* evaluate to the length of a List, where the second one is defined from tail to head. The list predicates do not evaluate to one for the empty heap. To also allow for empty lists, *ListEmpty* and *ListEmptyLen* provide definitions, where the second considers an empty heap as a list of size one. *ListOLen* and *ListELen* evaluate to the length of an odd and even list, respectively. We define these two predicates mutually recursive with each other. *DLLLen* evaluates to the length of a doubly linked list. *BinTreeSegHeight* and *BinTreeHeight* evaluates to the height of the tree (segment) and *BinTreeSize* to the number of pointer in the tree, where a tree consisting of a root with two null children has height one. We also have the predicates *spTrue*, *spTrueSize* and *spTrue2Size*, which evaluate always to one and to the size of the heap, respectively. The 2 in *spTrue2Size* refers to the fact that there we require heap values to consist of two fields instead, which is useful for data structures such as trees.

These predicates evaluate to zero if the data structure is not the correct data structure.

We go through the entailments in Table 9.3 step by step. First we have entailments involving various list data structures:

$$\mathit{ListLen}(a, b) \star [b \mapsto c] \vdash \mathit{ListLen}(a, c) \tag{ListLen-le-ListLen}$$

$$\mathit{ListLen}(a, b) \star [b \mapsto c] \vdash \mathit{RListLen}(a, c) \tag{ListLen-le-RListLen}$$

$$\mathit{ListLen}(a, b) \vdash \mathit{spTrueSize}(a, b) \tag{ListLen-le-size}$$

$$[a \mapsto b] \star RListLen(b, c) \vdash ListLen(a, c) \tag{RListLen-le-ListLen}$$

$$[a \mapsto b] \star \mathit{RListLen}(b, c) \vdash \mathit{RListLen}(a, c) \tag{RListLen-le-RListLen}$$

$$\mathit{ListLen}(a, b) \star \mathit{List}(b, c) \vdash \mathit{ListLen}(a, c) \tag{ListLen-List-le-ListLen}$$

$$RListLen(a,b) \star List(b,c) \vdash ListLen(a,c) \qquad \text{(RListLen-List-le-ListLen)}$$
$$ListLen(a,b) \star RList(b,c) \vdash ListLen(a,c) \qquad \text{(ListLen-RList-le-ListLen)}$$
$$RListLen(a,b) \star RList(b,c) \vdash RListLen(a,c) \qquad \text{(RListLen-RList-le-RListLen)}$$
$$ListLen(a,b) \star List(b,c) \vdash RListLen(a,c) \qquad \text{(ListLen-List-RListLen)}$$
$$ListOLen(a,b) \star [b \mapsto c] \vdash ListLen(a,c) \qquad \text{(ListOLen-le-ListLen)}$$
$$ListLen(a,b) \star [b \mapsto c] + List(a,c) \vdash ListLen(a,c) \qquad \text{(ListLen-add-List-le-ListLen)}$$
$$ListLen(a,b) \star [b \mapsto c] + List(a,c) \vdash spTrue() + ListLen(a,c) \qquad \text{(ListLen-add-List-le-true-add-List)}$$
$$ListLen(a,b) \star List(b,c) + ListLen(b,c) \star List(a,b) \vdash ListLen(a,c) \qquad \text{(ListLen-add-LenList)}$$

We are able to solve many of them, however are unable to solve those involving addition. This not because we are not able to deal with addition in principle. Unrolling the length predicates will always introduce addition operations as well and we are clearly able to deal with those. However, we are not able to deal with the addition from ListLen-add-List-le-ListLen, ListLen-add-List-le-true-add-List and ListLen-add-LenList, as we require eliminating addition at an early position. Since the implementation only eliminates addition during identity axioms and backlinks, it fails to prove these entailments. It is also obversable that entailments with high runtime usually have many backlinks or a great depth. This thus suggests that a runtime can be decreased by further speeding up the process of checking the soundness criterion, for which certain methods exists [103, 104].

We also evaluate our tool on some entailments occurring in literature [31]:

$$\begin{aligned} &([c = o] \cdot \langle 0.5 \rangle \cdot [\mathsf{emp}]) \star ListEmptyLen(hd, nil) \\ &+ ([c = o] \cdot \langle 0.5 \rangle \cdot [\mathsf{emp}]) \star ListEmptyLen(hd, nil) \\ &+ ([c = o] \cdot [\mathsf{emp}]) \star spTrue() \\ &+ ([c \neq o] \cdot [\mathsf{emp}]) \star ListEmptyLen(hd, nil) \\ \vdash\ & ListEmptyLen(hd, nil) + ([c = o] \cdot [\mathsf{emp}]) \star spTrue() \end{aligned} \qquad \text{(constant-adds-ListEmptyLen)}$$

$$\begin{aligned} &([hd = nil] \cdot [\mathsf{emp}]) \star ListEmptyLen(rv, nil) \\ &+ (\langle 1 \rangle \cdot [hd \neq nil] \cdot [\mathsf{emp}]) \star ListEmptyLen(rv, nil) \star ListEmpty(hd, nil) \\ &+ (\langle 0.5 \rangle \cdot [hd \neq nil] \cdot [\mathsf{emp}]) \star ListEmpty(rv, nil) \star ListEmptyLen(hd, nil) \\ \vdash\ & ListEmptyLen(rv, nil) \star ListEmpty(hd, nil) \\ &+ (\langle 0.5 \rangle \cdot [hd \neq nil] \cdot [\mathsf{emp}]) \star ListEmptyLen(hd, nil) \star ListEmpty(rv, nil) \end{aligned} \qquad \text{(head-adds-ListEmptyLen)}$$

There are two problems occurring here:

1. On the one hand, we again have additions in the entailment, which may require us to eliminate addition at an earlier stage than the implementation currently applies.
2. On the other hand, we require to introduce some kind of case distinction to introduce cases for $[c = o]$ and $[c \neq o]$, and $[hd = nil]$ and $[hd \neq nil]$, respectively.

Both of these are difficult to implement efficiently, but may help to solve these

kind of entailments.

For doubly linked list and binary trees we can observe a similar pattern that we already observed for the singly linked list. We used the following entailments:

Points-to predicates may have two fields here.

$$DLLLen(r_1, l_1, l_2, r_2) \star [l_1 \mapsto x, l_2] \vdash DLLLen(r_1, x, l_1, r_2) \quad \text{(DLLLen-le-DLLLen)}$$

$$\begin{matrix} DLLLen(m_1, l_1, l_2, m_2) \star DLL(r_1, m_1, m_2, r_2) \\ +DLL(m_1, l_1, l_2, m_2) \star DLLLen(r_1, m_1, m_2, r_2) \end{matrix} \vdash DLLLen(r_1, l_1, l_2, r_2) \quad \text{(DLLLen-add-DLL-le-DLLLen)}$$

$$BinTreeSeg(a, b) \star BinTreeSegHeight(b, c) \vdash BinTreeHeight(a) \quad \text{(BTSeg-BTSegHeight-le-BTHeight)}$$

$$BinTreeSegHeight(a, b) \star BinTreeSeg(b, c) \vdash BinTreeSize(a) \quad \text{(BTSegHeight-BTSeg-le-BTSize)}$$

$$BinTreeSegHeight(a, b) \vdash BinTreeSize(a) \quad \text{(BTSegHeight-le-BTSize)}$$

We observe here again, that whenever an early elimination of addition is required, the implementation fails to prove the entailment.

We also experimented with predicates that model the size of the whole heap. These should always be greater than some size predicate:

$$BinTreeSize(a) \vdash spTrue2Size() \quad \text{(BTSize-le-2size)}$$

$$BinTreeHeight(a) \vdash spTrue2Size() \quad \text{(BTHeight-le-2size)}$$

$$BinTreeSegHeight(a) \vdash spTrue2Size() \quad \text{(BTSegHeight-le-2size)}$$

Due to the large amount of options that these size predicates on the right-hand side of these entailments allow, we either fail to find a proof or need an extensive amount of time to find them.

In Chapter 4, we introduced preciseness of random variables. Separating multiplication with precise predicates is distributivity with respect to addition. Thus we may gain access to further proof rules, which may also help to find proofs for such precise predicates easier. This hypothesis was not validated in our experiments. We use precise versions of the list predicates called *ListLenPrec* and *RListLenPrec* for the following entailments:

$$ListLenPrec(a, b) \star [b \mapsto c] \star ([b \neq c] \cdot [\mathsf{emp}]) \vdash ListLenPrec(a, c) \quad \text{(ListLenPrec-le-ListLenPrec)}$$

$$ListLenPrec(a, b) \vdash spTrueSize(a, c) \quad \text{(ListLenPrec-le-size)}$$

$$([a \neq b] \cdot [\mathsf{emp}]) \star [a \mapsto b] \star RListLenPrec(b, c) \vdash ListLenPrec(a, c) \quad \text{(RListLenPrec-le-ListLenPrec)}$$

As we observe, we can only validate the entailment ListLenPrec-le-size. This is because preciseness introduces negated equalities, which in some cases are harder to handle.

Table 9.3.: We depict runtimes, depth and number of backlinks of proofs for entailments in quantitative separation logic with recursively-defined predicates. As there is currently no method for such entailments, we do not give any comparison.

Entailment	Time (sec)	Depth	Backlinks
ListLen-le-ListLen	0.01	2	1
ListLen-le-RListLen	1.39	6	2
ListLen-le-size	0.01	2	1
RListLen-le-ListLen	6.35	6	2
RListLen-le-RListLen	0.01	2	1
ListLen-List-le-ListLen	0.01	2	1
RListLen-List-le-ListLen	10.06	6	4
ListLen-RList-le-ListLen	0.05	3	3
RListLen-RList-le-RListLen	0.01	2	1
ListLen-List-RListLen	3.58	6	4
ListOLen-le-ListLen	0.08	4	1
ListLen-add-List-le-ListLen	TO		
ListLen-add-List-le-true-add-List	TO		
ListLen-add-LenList	TO		
constant-adds-ListEmptyLen [31]	TO		
head-adds-ListEmptyLen [31]	TO		
DLLLen-le-DLLLen	0.03	3	1
DLLLen-add-DLL-le-DLLLen	TO		
BTSeg-BTSegHeight-le-BTHeight	0.19	3	4
BTSegHeight-BTSeg-le-BTSize	0.31	3	4
BTSegHeight-le-BTSize	0.03	2	2
BTSize-le-2size	TO		
BTHeight-le-2size	947.57	10	13
BTSegHeight-le-2size	3889.44	11	5
ListLenPrec-le-ListLenPrec	TO		
ListLenPrec-le-size	0.01	2	1
RListLenPrec-le-ListLenPrec	TO		

10. Second Epilogue

Checking entailments is a crucial part for verifying properties of programs using denotational or axiomatic semantics. In semantics such as those studied in Chapters 4 and 6, weakening or strengthening random variables is crucial to apply certain proof rules and especially to gain inductivity of random variables to reason about loops. However, verifying that the weakening or strengthening of a random variable to a different random variable is valid, requires checking an entailment between these. In these scenarios, it is usually also more important to gain confidence that the entailment holds and not that the entailment does not hold. As such, sound but possibly non-terminating approaches for verifying entailments also gain attraction.

To reason about fuzzy and quantitative separation logic, techniques for handling both quantities and separation logic are required. Both features pose challenges in their own regard and are well studied. The combination is a less well studied field on the other hand, but necessary for semantics such as the one presented in Chapters 4 and 6.

Checking entailments in separation logic has a long history, starting with simple techniques supporting only a limited number of predicates for data structures [84, 85, 93, 96]. This has soon been extended to user definable predicates [83, 86–91], similarly to how users can define predicates in the logic defined in Chapter 9. Additional effort went into supporting magic wands [91, 92, 94, 97], which are well-known to be particularly hard to deal with. While most of these techniques have non-polynomial runtime complexity, some effort was also put into offering polynomial runtime [84, 85, 95]. This rich history motivated our work towards a method to facilitate these techniques for entailment checking in fuzzy separation logic presented in Chapter 8 and [32].

A source of verifying separation logic entailments that is especially relevant for our work are cyclic proofs. Cyclic proof systems have been developed that support verification of entailments between separation logic formulae [101, 105–107] and program verification with separation logic [108]. Some of these techniques such as [105] then even gave rise to decidability results in their fragment.

A less automatic approach to verifying separation logic entailments facilitates the use of theorem proof assistants. One prominent such framework, which allows reasoning about separation logic entailments, is MoSeL [109] implemented in the Rocq prover language [61]. MoSeL implements a proof assistant designed for the use of separating conjunction and magic wands and is thus a powerful tool to reason about separation logic entailments.

Rocq prover was previously known as Coq.

Quantitative reasoning has a long history, dating back to one of the first decision procedures for quantitative "entailments" involving first-order formulae over linear inequalities in the natural numbers by Mojżesz Presburger [110] and recent extensions to it [111]. The famous simplex algorithm [112] gave rise to checking and even optimizing a solution for linear inequalities in the real numbers, better known as linear programming in modern times. Further development also yielded (possibly incomplete) techniques to verify or reject formulae in non-linear arithmetic over reals [113–116]. These advancements have been exploited by

a transformation that convert the quantitative logic HeyLO into a satisfiability modulo theory problem for which the previous mentioned techniques can yield verification results. This transformation then gave rise to a verification infrastructure for probabilistic programs. [117]

Decision Procedures for Fuzzy Separation Logic

In Chapter 8, we investigated a procedure to transform random variables in fuzzy separation logic into qualitative separation logic predicates. For this, we converted the entailment between random variables into a qualitative entailment between at-least predicates. At-least predicates can then be turned into expressions using operations from qualitative separation logic. The transformation introduces various predicates over real arithmetic, that most entailment checkers for separation logic are not able to handle. While a tool like MoSeL may be applicable, automated reasoning is thus not possible yet. However, by limiting the allowed random variables to the syntax we introduced in Section 8.3, the conditions on real arithmetic become decidable and we thus obtain qualitative separation logic formulae that can be discharged to standard separation logic entailment checkers. The complexity of this transformation is exponential in the length of the formula.

This approach yields various decidable fragments in fuzzy separation logic. Procedures that are tailored to solve the entailment on the quantitative level directly — instead of transforming it into a qualitative separation logic problem — may be beneficial, however. Such procedures may be able achieve better runtime complexity. Moreover, procedures tailored to these problems may also yield better coverage, especially they may be able to check entailments between random variables with infinite support.

Cyclic Proofs for Quantitative Separation Logic

Cyclic proofs have been used for more than a decade to verify entailments in qualitative logics, especially in first order logic [118, 119] and in qualitative separation logic [101, 105–108]. Cyclic proofs are useful when reasoning about predicates defined as fixed points over some function. Reasoning about fixed points usually requires induction. Cyclic reasoning introduces the dual of induction called infinite descent. As a progress measure for the infinite descent, unfolding of recursively-defined predicates is used. Quantitative recursively-defined predicates are useful to define sizes of data structures. As such, it is useful to lift cyclic reasoning to quantitative separation logic. It turns out that lifting the classic progress measure in the quantitative setting is non-trivial. We found counter examples for a naïve lifting of the proof, which we presented in Section 9.3. Instead of using the classic unfolding progress measure, we presented an approach using heap sizes as a progress measure.

It is imaginable that a proof system may also use both the classic unfolding progress measure and the heap size progress measure. Indeed, in a qualitative setting this is feasible. In quantitative settings however, this requires to verify when it is valid to apply the unfolding progress measure. Deciding if a predicate is qualitative (i.e. only maps to 0 or 1), is difficult and may even be undecidable in certain fragments of the logic.

In principle, a cyclic proof system for fuzzy separation logic is also feasible. The problem here is that it seems there is no use-case for fuzzy recursively-defined predicates. Indeed, we either are interested in defining data structures or their sizes — the latter is non-fuzzy in nature.

Cyclic proofs have also been used to reason about programs and to verify properties such as termination [108] of a program. Adapting these techniques to the quantitative setting using the heap size as progress measure may yield a cyclic proof system that allows reasoning about probabilistic heap-manipulating programs, especially also about such programs which are looping and where the loop is iterating over some form of data structure. In such loops, it is imaginable that frame rules can be applied to obtain progress.

The validity of the unfolding progress measure is still unclear. There are arguments that the current framework may not be sufficient to express the validity of the unfolding progress measure. We believe that either an extension of the current soundness framework from Definition 9.2.3 is necessary, or that there is a proper counter example for that soundness criterion, i.e. a proof using unfoldings on the left of the entailment verifying an invalid sequent.

Cyclic proofs have rich tool support [102, 120, 121]. The approach presented here was implemented in the Cyclist [102] tool with a different set of inference rules compared to the rules presented in this dissertation. The implementation aims to be efficient and loses expressivity instead [100]. Since this implementation does not require changes to the framework of the tool, we believe that adapting this implementation to other tools for cyclic proofs is not difficult.

Proof Assistant Support for Fuzzy and Quantitative Logics

Implementing program semantics in a proof assistant also requires us to prove entailments in the proof assistant's language. Since we implemented the semantics from Chapter 6 in the proof assistant Lean, a natural question that arises is concerning support to reason about entailments in the proof assistant Lean. Unfortunately, there is currently only limited support for this.

The logic Iris [41] has an infrastructure for reasoning about Iris separation logic formulae called MoSeL [109]. Naturally, one may ask two questions:

1. Can we adapt MoSeL for quantitative logics?
2. Is it possible to build an analogous of MoSeL for quantitative logics?

We do not answer any of these questions in this dissertation, but can give some insights.

The MoSeL framework is targeted at logics that are based on bunched implications (BI) [122]. The version of BI used by MoSeL uses various operations, including classic conjunction $\wedge$ and disjunction $\vee$, separating conjunction $*$, and a persistent modality $\boxdot$ and axioms on these. The persistent modality is used to make a predicate in that logic "persistent". The exact meaning of persistent differs from logic to logic, but an important property of persistence is that it is duplicatable. For an entailment relation $\vDash$, important axioms for our considerations are

Duplicatable here means that both $\boxdot P \vDash (\boxdot P) * (\boxdot P)$, $(\boxdot P) * (\boxdot P) \vDash \boxdot P$ holds.

1. and-elim: For $P \vDash Q$ and $P \vDash R$ we have $P \vDash Q \wedge R$,
2. or-elim: For $P \vDash R$ and $Q \vDash R$ we have $P \vee Q \vDash R$, and
3. pers-conj-elim: $(\boxdot P) \wedge Q \vDash P * Q$.

The first two axioms are standard. The third tells us that a proposition P that is persistent "should not add any heap" to Q. A classic way of defining the semantics for the persistent modality is to require that the proposition it is guarding holds for the empty heap.

Some authors [31, pp. 170-171] may reasonably suggest using multiplication to model conjunction of Boolean predicates. A fair idea is thus to also try it the other way around. Unfortunately, multiplication fails the and-elim axiom both in fuzzy and quantitative logic: $0.5 \leq 0.55$ and $0.5 \leq 0.55$ but $0.5 \nleq 0.55 \cdot 0.55 = 0.3025$. The same also holds for using addition as disjunction, which may be tempting to try considering the conservativity properties we found in Theorems 4.3.2 and 5.3.5. We unfortunately have for or-elim: $0.5 \leq 0.6$ and $0.4 \leq 0.6$ but $0.5 + 0.4 = 0.9 \nleq 0.6$. Thus it seems necessary to use infima and suprema for conjunction and disjunction respectively. However, a classic definition of the persistence modality requires that the conjunction and the separating conjunction use the same underlying operation in order for the pers-conj-elim operation to be valid. If we choose infima for conjunction, the separating operation should also be defined by an infimum instead of the product. But then we loose quantitative reasoning completely. It thus seems that adapting MoSeL for reasoning in fuzzy or quantitative separation logic is difficult. Runtime separation logic [123] may be another contender to be instantiated in MoSeL. There we use addition as conjunction but associate true with zero and false with infinity. Whether runtime separation logic can be reasonably instantiated into MoSeL is part of future work.

The second question goes in a direction that may be more fruitful in the long run. We have many separation logics (fuzzy, quantitative and runtime) to reason about quantities. A unifying framework to reason about all of them would allow to use one tool for each of them. This tool would need to be able to be general enough for every one of them be instantiable, but concrete enough to allow reasoning of statements that only hold in one but not the other logic. Finding such a unifying framework seems challenging.

While Iris is popular for its easy-to-use proof assistant support due to MoSeL, it also features easily defined recursive predicates that are not limited to monotonic functions like the ones in Chapter 9. This is realized by defining Iris as a step-indexing logic [41] that allows defining predicates recursively with the syntactic restriction that every recursive occurrence of a predicate must be guarded by a special operator called the later modality. This modality allows the step-counter to decrease by one. The fixed point of the predicate is then defined by a sequence of increasingly stronger predicates. Induction allows reasoning about these sequences. Adapting these for fuzzy, quantitative or runtime separation logic may yield a powerful tool to define and reason about recursively-defined predicates.

Note that the step-counter is part of the logic and not of any program semantics.

Bibliography

[1] Jan Eloff and Madeleine Bihina Bella. "Software Failures: An Overview." In: *Software Failure Investigation: A Near-Miss Analysis Approach*. Cham: Springer International Publishing, 2018, pp. 7–24. DOI: 10.1007/978-3-319-61334-5_2 (cited on page 1).

[2] Herb Krasner. *The Cost of Poor Software Quality in the US: A 2020 Report*. Tech. rep. Consortium for Information & Software Quality, 2021 (cited on page 1).

[3] H. B. Curry. "Functionality in Combinatory Logic*." In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. DOI: 10.1073/pnas.20.11.584 (cited on page 1).

[4] Haskell Brooks Curry and Robert M. Feys. *Combinatory Logic Vol. 1*. Amsterdam, Netherlands: North-Holland Publishing Company, 1958 (cited on page 1).

[5] W. A. Howard. "The Formulæ-as-Types Notion of Construction." In: *The Curry-Howard isomorphism*. Ed. by Philippe De Groote. Academia, 1995 (cited on page 1).

[6] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language." In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635 (cited on pages 1, 134).

[7] Thierry Coquand and Gérard Huet. "The calculus of constructions." In: *Information and Computation* 76.2 (1988), pp. 95–120. DOI: https://doi.org/10.1016/0890-5401(88)90005-3 (cited on page 1).

[8] Caroline Graf et al. "Animal, dog, or dalmatian? Level of abstraction in nominal referring expressions." In: *Cognitive Science* (2016) (cited on page 1).

[9] Robert D. Hawkins et al. "Why do you ask? Good questions provoke informative answers." In: *Cognitive Science* (2015) (cited on page 1).

[10] Daniel Ritchie et al. "Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo." In: *ACM Trans. Graph.* 34.4 (July 2015). DOI: 10.1145/2766895 (cited on page 1).

[11] Owain Evans et al. *Modeling Agents with Probabilistic Programs*. http://agentmodels.org. Accessed: 2025-4-24. 2017 (cited on page 1).

[12] C. A. R. Hoare. "Quicksort." In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. DOI: 10.1093/comjnl/5.1.10 (cited on page 2).

[13] Seth Pettie and Vijaya Ramachandran. "An optimal minimum spanning tree algorithm." In: *J. ACM* 49.1 (Jan. 2002), pp. 16–34. DOI: 10.1145/505241.505243 (cited on page 2).

[14] David R. Karger, Philip N. Klein, and Robert E. Tarjan. "A randomized linear-time algorithm to find minimum spanning trees." In: *J. ACM* 42.2 (Mar. 1995), pp. 321–328. DOI: 10.1145/201019.201022 (cited on page 2).

[15] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors." In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692 (cited on page 2).

[16] Heinz-Gerd Hegering and Alfred Lapple. *Ethernet; Building a Communications Infrastructure*. USA: Addison-Wesley Longman Publishing Co., Inc., 1993 (cited on page 2).

[17] Hagit Attiya and Jennifer Welch. "Leader Election in Rings." In: *Distributed Computing*. John Wiley & Sons, Ltd, 2004. Chap. 3, pp. 31–58. DOI: https://doi.org/10.1002/0471478210.ch3 (cited on page 2).

[18] Murali Krishna Ramanathan et al. "Randomized leader election." In: *Distributed Computing* 19.5 (Apr. 2007), pp. 403–418. DOI: 10.1007/s00446-007-0022-4 (cited on page 2).

[19] Philip N. Klein and Sairam Subramanian. “A Randomized Parallel Algorithm for Single-Source Shortest Paths.” In: *Journal of Algorithms* 25.2 (1997), pp. 205–220. DOI: https://doi.org/10.1006/jagm.1997.0888 (cited on page 2).

[20] C. A. R. Hoare. “An axiomatic basis for computer programming.” In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on page 2).

[21] Robert W. Floyd. “Assigning Meanings to Programs.” In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32 (cited on page 2).

[22] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs.” In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975 (cited on page 3).

[23] Peter O’Hearn, John Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures.” In: *Computer Science Logic*. Ed. by Laurent Fribourg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1 (cited on pages 3, 131).

[24] J.C. Reynolds. “Separation logic: a logic for shared mutable data structures.” In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 3, 131).

[25] Peter W. O’Hearn. “Resources, Concurrency and Local Reasoning.” In: *CONCUR 2004 - Concurrency Theory*. Ed. by Philippa Gardner and Nobuko Yoshida. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 49–67. DOI: 10.1007/978-3-540-28644-8_4 (cited on pages 3, 25, 131–133).

[26] Viktor Vafeiadis. “Concurrent Separation Logic and Operational Semantics.” In: *Electronic Notes in Theoretical Computer Science* 276 (2011). Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), pp. 335–351. DOI: 10.1016/j.entcs.2011.09.029 (cited on pages 3, 25, 33, 37, 39, 43, 44, 131–133).

[27] Dexter Kozen. “Semantics of probabilistic programs.” In: *Journal of Computer and System Sciences* 22.3 (1981), pp. 328–350. DOI: https://doi.org/10.1016/0022-0000(81)90036-2 (cited on pages 4, 132).

[28] Dexter Kozen. “A probabilistic PDL.” In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC ’83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 291–297. DOI: 10.1145/800061.808758 (cited on pages 4, 132).

[29] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. New York: Springer, 2005 (cited on pages 4, 132).

[30] Kevin Batz et al. “Quantitative separation logic: a logic for reasoning about probabilistic pointer programs.” In: 3.POPL (2019). DOI: 10.1145/3290347 (cited on pages 4, 56, 61, 64, 65, 70, 71, 132).

[31] Christoph Matheja. “Automated reasoning and randomization in separation logic.” Dissertation. Aachen: RWTH Aachen University, 2020. DOI: 10.18154/RWTH-2020-00940 (cited on pages 4, 56, 70, 84, 95, 172, 173, 176, 180, 182, 184, 188).

[32] Kevin Batz et al. “Foundations for Entailment Checking in Quantitative Separation Logic.” In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Cham: Springer International Publishing, 2022, pp. 57–84. DOI: 10.1007/978-3-030-99336-8_3 (cited on pages 4, 7, 180, 181, 185).

[33] Ira Fesefeldt, Joost-Pieter Katoen, and Thomas Noll. “Towards Concurrent Quantitative Separation Logic.” In: *33rd International Conference on Concurrency Theory (CONCUR 2022)*. Ed. by Bartek Klin, Sławomir Lasota, and Anca Muscholl. Vol. 243. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 25:1–25:24. DOI: 10.4230/LIPIcs.CONCUR.2022.25 (cited on pages 4, 7, 33, 105, 132, 133).

[34] Ira Fesefeldt et al. "Automated Checking and Completion of Backward Confluence for Hyperedge Replacement Grammars." In: *Graph Transformation*. Ed. by Fabio Gadducci and Timo Kehrer. Cham: Springer International Publishing, 2021, pp. 283–293 (cited on page 7).

[35] Bronisław Knaster. "Un theoreme sur les functions d'ensembles." In: *Ann. Soc. Polon. Math.* 6 (1928), pp. 133–134 (cited on pages 15, 16).

[36] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309 (cited on pages 15, 16).

[37] David Park. "Fixpoint induction and proofs of program properties." In: *Machine Intelligence* 5 (1969) (cited on page 16).

[38] J.-L. Lassez, V.L. Nguyen, and E.A. Sonenberg. "Fixed point theorems and semantics: a folk tale." In: *Information Processing Letters* 14.3 (1982), pp. 112–116. DOI: https://doi.org/10.1016/0020-0190(82)90065-5 (cited on page 17).

[39] A. G. Hamilton. "Ordinal and cardinal numbers." In: *Numbers, Sets and Axioms: The Apparatus of Mathematics*. Cambridge University Press, 1983 (cited on page 19).

[40] Patrick Cousot and Radhia Cousot. "Constructive versions of Tarski's fixed point theorems." In: *Pacific Journal of Mathematics* 82.1 (1979), pp. 43–57 (cited on page 20).

[41] Ralf Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on pages 25, 26, 37, 131, 187, 188).

[42] Afshin Amighi, Stefan Blom, and Marieke Huisman. "VerCors: A Layered Approach to Practical Verification of Concurrent Software." In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2016, pp. 495–503. DOI: 10.1109/PDP.2016.107 (cited on page 26).

[43] Ira Fesefeldt. *Lean Formalization for "Deductive Reasoning about Concurrent Probabilistic Programs"*. May 2025. DOI: 10.5281/zenodo.15358457 (cited on pages 28, 31, 64, 65, 76–79, 81, 82, 87, 89–91, 94, 95, 100, 107, 108, 115, 119, 121, 141–147, 163).

[44] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. "Separation and information hiding." In: *ACM Trans. Program. Lang. Syst.* 31.3 (2009). DOI: 10.1145/1498926.1498929 (cited on pages 42, 43).

[45] Kenneth E. Iverson. *A programming language*. USA: John Wiley & Sons, Inc., 1962 (cited on page 49).

[46] Gordan Žitković. *Theory of Probability: Measure theory, classical probability and stochastic analysis*. USA: Department of Mathamtics, University of Texas, 2010 (cited on pages 50, 51).

[47] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cited on pages 51–53).

[48] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. USA: John Wiley & Sons, Inc., 1994 (cited on pages 54, 55).

[49] Kevin Batz. "Automated deductive verification of probabilistic programs." Dissertation. RWTH Aachen University, 2024. DOI: 10.18154/RWTH-2025-00473 (cited on pages 56, 134).

[50] Vilém Novák, Irina Perfilieva, and Jiří Močkoř. *Mathematical Principles of Fuzzy Logic*. 1st ed. New York, NY: Springer US, 1999 (cited on page 61).

[51] Marcel Hark et al. "Aiming low is harder: induction for lower bounds in probabilistic program verification." In: *Proc. ACM Program. Lang.* 4.POPL (2019). DOI: 10.1145/3371105 (cited on page 69).

[52] A.K. McIver and Carroll Morgan. "Partial correctness for probabilistic demonic programs." In: *Theoretical Computer Science* 266.1 (2001), pp. 513–541. DOI: 10.1016/S0304-3975(00)00208-5 (cited on page 75).

[53] Nils Jansen et al. "Conditioning in Probabilistic Programming." In: *Electronic Notes in Theoretical Computer Science* 319 (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 199–216. DOI: https://doi.org/10.1016/j.entcs.2015.12.013 (cited on page 75).

[54] *Mathlib4*. 2024. URL: https://github.com/leanprover-community/mathlib4 (cited on pages 77, 79, 81, 82).

[55] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms*. first. Operations Research/Computer Science Interfaces Series. New York, NY: Springer, 2008 (cited on page 78).

[56] Stefan Blom et al. "The VerCors Tool Set: Verification of Parallel and Concurrent Software." In: *IFM*. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 102–110 (cited on page 131).

[57] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62 (cited on page 131).

[58] Thomas Dinsdale-Young et al. "Concurrent Abstract Predicates." In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by Theo D'Hondt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 504–528 (cited on page 131).

[59] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction." In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 207–231 (cited on page 131).

[60] Aleksandar Nanevski et al. "Communicating State Transition Systems for Fine-Grained Concurrent Resources." In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 290–310 (cited on page 131).

[61] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer Berlin, Heidelberg, 2004 (cited on pages 131, 185).

[62] Gilles Barthe, Justin Hsu, and Kevin Liao. "A probabilistic separation logic." In: 4.POPL (Dec. 2019). DOI: 10.1145/3371123 (cited on page 131).

[63] John M. Li, Amal Ahmed, and Steven Holtzen. "Lilac: A Modal Separation Logic for Conditional Probability." In: 7.PLDI (June 2023). DOI: 10.1145/3591226 (cited on page 131).

[64] Joseph Tassarotti and Robert Harper. "A separation logic for concurrent randomized programs." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290377 (cited on pages 131, 134).

[65] Simon Oddershede Gregersen et al. "Asynchronous Probabilistic Couplings in Higher-Order Separation Logic." In: 8.POPL (Jan. 2024). DOI: 10.1145/3632868 (cited on pages 131, 134).

[66] Alejandro Aguirre et al. "Error Credits: Resourceful Reasoning about Error Bounds for Higher-Order Probabilistic Programs." In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). DOI: 10.1145/3674635 (cited on pages 132, 134).

[67] Philipp G. Haselwarter et al. "Tachis: Higher-Order Separation Logic with Credits for Expected Costs." In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689753 (cited on pages 132, 134).

[68] Janine Lohse and Deepak Garg. *An Iris for Expected Cost Analysis*. 2024. URL: https://arxiv.org/abs/2406.00884 (cited on pages 132, 134).

[69] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. "Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning." In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023). DOI: 10.1145/3586045 (cited on page 132).

[70] Noam Zilberstein, Angelina Saliling, and Alexandra Silva. “Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects.” In: 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649821 (cited on page 132).

[71] Noam Zilberstein, Alexandra Silva, and Joseph Tassarotti. *Probabilistic Concurrent Reasoning in Outcome Logic: Independence, Conditioning, and Invariants*. 2024. URL: https://arxiv.org/abs/2411.11662 (cited on page 132).

[72] Annabelle McIver, Tahiry Rabehaja, and Georg Struth. “Probabilistic rely-guarantee calculus.” In: *Theoretical Computer Science* 655 (2016). Quantitative Aspects of Programming Languages and Systems (2013-14), pp. 120–134. DOI: https://doi.org/10.1016/j.tcs.2016.01.016 (cited on page 132).

[73] Noam Zilberstein, Daniele Gorla, and Alexandra Silva. *Denotational Semantics for Probabilistic and Concurrent Programs*. 2025. URL: https://arxiv.org/abs/2503.02768 (cited on page 132).

[74] Kevin Batz et al. “A Calculus for Amortized Expected Runtimes.” In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571260 (cited on page 132).

[75] Joe Hurd, Annabelle McIver, and Carroll Morgan. “Probabilistic guarded commands mechanized in HOL.” In: *Theoretical Computer Science* 346.1 (2005). Quantitative Aspects of Programming Languages (QAPL 2004), pp. 96–112. DOI: https://doi.org/10.1016/j.tcs.2005.08.005 (cited on pages 133, 134).

[76] Ralf Jung et al. “Higher-order ghost state.” In: *SIGPLAN Not.* 51.9 (Sept. 2016), pp. 256–269. DOI: 10.1145/3022670.2951943 (cited on page 134).

[77] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. “The spirit of ghost code.” In: *Formal Methods in System Design*. Vol. 48. 2016, pp. 152–174 (cited on page 134).

[78] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002 (cited on page 134).

[79] Andreas Lochbihler. “Probabilistic Functions and Cryptographic Oracles in Higher Order Logic.” In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 503–531 (cited on page 134).

[80] Philippe Audebaud and Christine Paulin-Mohring. “Proofs of Randomized Algorithms in Coq.” In: *Mathematics of Program Construction*. Ed. by Tarmo Uustalu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 49–68 (cited on page 134).

[81] David Cock. “Verifying Probabilistic Correctness in Isabelle with pGCL.” In: *Electronic Proceedings in Theoretical Computer Science* 102 (Nov. 2012), pp. 167–178. DOI: 10.4204/eptcs.102.15 (cited on page 134).

[82] Maximilian Paul Louis Haslbeck. “Verified Quantitative Analysis of Imperative Algorithms.” en. PhD thesis. Technische Universität München, 2021, p. 229 (cited on page 134).

[83] Timos Antonopoulos et al. “Foundations for Decision Problems in Separation Logic with General Inductive Predicates.” In: *Foundations of Software Science and Computation Structures*. Ed. by Anca Muscholl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 411–425. DOI: 10.1007/978-3-642-54830-7_27 (cited on pages 148, 158, 185).

[84] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. “A Decidable Fragment of Separation Logic.” In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*. Ed. by Kamal Lodaya and Meena Mahajan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–109. DOI: 10.1007/978-3-540-30538-5_9 (cited on pages 148, 158, 185).

[85] Byron Cook et al. "Tractable Reasoning in a Fragment of Separation Logic." In: *CONCUR 2011 – Concurrency Theory*. Ed. by Joost-Pieter Katoen and Barbara König. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 235–249. DOI: 10.1007/978-3-642-23217-6_16 (cited on pages 148, 158, 185).

[86] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. "Decidable Entailments in Separation Logic with Inductive Definitions: Beyond Establishment." In: *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*. Ed. by Christel Baier and Jean Goubault-Larrecq. Vol. 183. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 20:1–20:18. DOI: 10.4230/LIPIcs.CSL.2021.20 (cited on pages 148, 158, 185).

[87] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. "Unifying Decidable Entailments in Separation Logic with Inductive Definitions." In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 183–199. DOI: 10.1007/978-3-030-79876-5_11 (cited on pages 148, 158, 185).

[88] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. "The Tree Width of Separation Logic with Recursive Definitions." In: *Automated Deduction – CADE-24*. Ed. by Maria Paola Bonacina. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–38. DOI: 10.1007/978-3-642-38574-2_2 (cited on pages 148, 158, 185).

[89] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. "Deciding Entailments in Inductive Separation Logic with Tree Automata." In: *Automated Technology for Verification and Analysis*. Ed. by Franck Cassez and Jean-François Raskin. Cham: Springer International Publishing, 2014, pp. 201–218. DOI: 10.1007/978-3-319-11936-6_15 (cited on pages 148, 158, 185).

[90] 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*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 319–336. DOI: 10.1007/978-3-030-17465-1_18 (cited on pages 148, 158, 185).

[91] Christoph Matheja, Jens Pagel, and Florian Zuleger. "A Decision Procedure for Guarded Separation Logic Complete Entailment Checking for Separation Logic with Inductive Definitions." In: *ACM Trans. Comput. Logic* 24.1 (Jan. 2023). DOI: 10.1145/3534927 (cited on pages 148, 158, 185).

[92] Andrew Reynolds et al. "A Decision Procedure for Separation Logic in SMT." In: *Automated Technology for Verification and Analysis*. Ed. by Cyrille Artho, Axel Legay, and Doron Peled. Cham: Springer International Publishing, 2016, pp. 244–261. DOI: 10.1007/978-3-319-46520-3_16 (cited on pages 148, 158, 180, 185).

[93] Stéphane Demri, Etienne Lozes, and Alessio Mansutti. "The Effects of Adding Reachability Predicates in Quantifier-Free Separation Logic." In: *ACM Trans. Comput. Logic* 22.2 (June 2021). DOI: 10.1145/3448269 (cited on pages 148, 158, 185).

[94] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. "The Bernays-Schönfinkel-Ramsey Class of Separation Logic with Uninterpreted Predicates." In: 21.3 (Mar. 2020). DOI: 10.1145/3380809 (cited on pages 148, 158, 185).

[95] Quang Loc Le and Xuan-Bach D. Le. "An Efficient Cyclic Entailment Procedure in a Fragment of Separation Logic." In: *Foundations of Software Science and Computation Structures*. Ed. by Orna Kupferman and Pawel Sobocinski. Cham: Springer Nature Switzerland, 2023, pp. 477–497 (cited on pages 148, 158, 185).

[96] Tomáš Dacík et al. "Deciding Boolean Separation Logic via Small Models." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Cham: Springer Nature Switzerland, 2024, pp. 188–206. DOI: 10.1007/978-3-031-57246-3_11 (cited on pages 148, 157, 158, 185).

[97] Jens Pagel and Florian Zuleger. "Strong-separation Logic." In: 44.3 (July 2022). DOI: 10.1145/3498847 (cited on pages 148, 158, 185).

[98] James Brotherston. "Sequent calculus proof systems for inductive definitions." PhD thesis. Edinburgh: University of Edinburgh, 2006 (cited on pages 160, 165, 169).

[99] Patrick Nossol. "Entailments in quantitative separation logic with recursive definitions: constructing cyclyc proofs." Masterarbeit. Aachen: RWTH Aachen University, 2024. DOI: 10.18154/RWTH-2024-09539 (cited on page 162).

[100] Patrick Nossol. *Entailments in quantitative separation logic with recursive definitions: constructing cyclyc proofs*. Master Thesis. 2024. DOI: 10.18154/RWTH-2024-09539 (cited on pages 170, 178, 179, 187).

[101] James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. "Automated Cyclic Entailment Proofs in Separation Logic." In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 131–146. DOI: 10.1007/978-3-642-22438-6_12 (cited on pages 179, 185, 186).

[102] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. "A Generic Cyclic Theorem Prover." In: *Programming Languages and Systems*. Ed. by Ranjit Jhala and Atsushi Igarashi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 350–367. DOI: 10.1007/978-3-642-35182-2_25 (cited on pages 179, 187).

[103] Liron Cohen et al. "The Complex(ity) Landscape of Checking Infinite Descent." In: 8.POPL (Jan. 2024). DOI: 10.1145/3632888 (cited on page 182).

[104] Liron Cohen, Reuben N. S. Rowe, and Matan Shaked. "Cyclone: A Heterogeneous Tool for Verifying Infinite Descent." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Arie Gurfinkel and Marijn Heule. Cham: Springer Nature Switzerland, 2025, pp. 336–354 (cited on page 182).

[105] Makoto Tatsuta, Koji Nakazawa, and Daisuke Kimura. "Completeness of Cyclic Proofs for Symbolic Heaps with Inductive Definitions." In: *Programming Languages and Systems*. Springer, 2019, pp. 367–387. DOI: 10.1007/978-3-030-34175-6_19 (cited on pages 185, 186).

[106] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. "Automatic induction proofs of data-structures in imperative programs." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015, pp. 457–466. DOI: 10.1145/2737924.2737984 (cited on pages 185, 186).

[107] Quang-Trung Ta et al. "Automated lemma synthesis in symbolic-heap separation logic." In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158097 (cited on pages 185, 186).

[108] James Brotherston, Richard Bornat, and Cristiano Calcagno. "Cyclic proofs of program termination in separation logic." In: *SIGPLAN Not.* 43.1 (2008), pp. 101–112. DOI: 10.1145/1328897.1328453 (cited on pages 185–187).

[109] Robbert Krebbers et al. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic." In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236772 (cited on pages 185, 187).

[110] Mojżesz Presburger. "Uber die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt." In: *Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs*. 1929 (cited on page 185).

[111] Toghrul Karimov et al. "On the Decidability of Presburger Arithmetic Expanded with Powers." In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2755–2778. DOI: 10.1137/1.9781611978322.89 (cited on page 185).

[112] George Dantzig. *Reminiscences about the Origins of Linear Programming*. Tech. rep. Stanford University, 1981 (cited on page 185).

[113] George E. Collins. "Quantifier elimination for real closed fields by cylindrical algebraic decompostion." In: *Automata Theory and Formal Languages*. Ed. by H. Brakhage. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 134–183. DOI: 10.1007/3-540-07407-4_17 (cited on page 185).

[114] Volker Weispfenning. "The complexity of linear problems in fields." In: *J. Symb. Comput.* 5.1–2 (Feb. 1988), pp. 3–27. DOI: 10.1016/S0747-7171(88)80003-8 (cited on page 185).

[115] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Springer New York, NY, 1993 (cited on page 185).

[116] Pascal Van Hentenryck, David McAllester, and Deepak Kapur. "Solving Polynomial Systems Using a Branch and Prune Approach." In: *SIAM Journal on Numerical Analysis* 34.2 (1997), pp. 797–827. DOI: 10.1137/S0036142995281504 (cited on page 185).

[117] Philipp Schröer et al. "A Deductive Verification Infrastructure for Probabilistic Programs." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622870 (cited on page 186).

[118] James Brotherston. "Cyclic Proofs for First-Order Logic with Inductive Definitions." In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2005, pp. 78–92. DOI: 10.1007/11554554_8 (cited on page 186).

[119] James Brotherston and Alex Simpson. "Sequent calculi for induction and infinite descent." In: *Journal of Logic and Computation* 21.6 (2011), pp. 1177–1216. DOI: 10.1093/logcom/exq052 (cited on page 186).

[120] Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. "CycleQ: an efficient basis for cyclic equational reasoning." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2022, pp. 395–409. DOI: 10.1145/3519939.3523731 (cited on page 187).

[121] Cristina Serban and Radu Iosif. "An Entailment Checker for Separation Logic with Inductive Definitions." In: *Electronic Communications of the EASST* 76 (2019). DOI: 10.14279/tuj.eceasst.76.1073 (cited on page 187).

[122] Peter W. O'Hearn and David J. Pym. "The Logic of Bunched Implications." In: *Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090 (cited on page 187).

[123] Kevin Batz et al. "A Calculus for Amortized Expected Runtimes." In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571260 (cited on page 188).

Notation

General Math

Symbol	Type	Meaning
a, b, c		logical variables
n, i, j	$\mathbb{N}$	Natural numbers
p, q	$[0, 1]$	Probabilities

Programs

Symbol	Type	Meaning
$x, y, z, r, \ldots$	Vars	Program variables
k, o, u		either logical or program variables
$e, e', e_1, \ldots$	Expressions	Expressions mapping the program state to values in $\mathbb{Q}$, Booleans or values between 0 and 1

General Separation Logic

Symbol	Type	Meaning
$s, s', s_1, s_2, \ldots$	Stacks	Mappings from variables to rationals
$h, h', h_1, h_2, \ldots$	Heaps	Mappings from locations (natural numbers) to rationals
$\ell, \ell', \ell_1, \ell_2, \ldots$	$\mathbb{N}$	Locations in an heap
η	LVars $\rightarrow \mathbb{Q}$	Mappings from logical variables to values

Qualitative Separation Logic

Symbol	Type	Meaning
φ, ψ		Formulae in qualitative separation logic
Φ, Ψ, Θ	SL	Separation logic propositions
ξ, π	SL	Resource invariants in separation logic
emp	SL	Empty heap predicate proposition
$\Phi * \Psi$	SL	Separating conjunction connecting the propositions Φ and Ψ
$\Phi \mathrel{-\!\!*} \Psi$	SL	Magic wand connecting the propositions Φ and Ψ

Quantitative and Fuzzy Separation Logic

Symbol	Type	Meaning
f, g		Formulae in quantitative or fuzzy separation logic
X, Y, Z	QSL, FSL	Quantitative separation logic random variables
R, Q	QSL, FSL	Resource invariants in quantitative separtion logic
$[\mathsf{emp}]$	QSL, FSL	Empty heap predicate random variable
$\sim X$	FSL	The negation of a random variable X, i.e. $1 - X$
$X \sqcup Y$	QSL, FSL	The maximum of the random variables X and Y
$X \sqcap Y$	QSL, FSL	The minimum of the random variables X and Y
$X + Y$	QSL	The addition of the random variables X and Y
$X \dot{+} Y$	FSL	The truncated addition of the random variables X and Y
$\text{Ƨ} x.\, X$	QSL, FSL	Supremum with bounded variable x in the random variable X
$\text{I} x.\, X$	QSL, FSL	Supremum with bounded variable x in the random variable X
$X \star Y$	QSL, FSL	Separating conjunction connecting the random variables X and Y
$X \mathrel{-\!\!\star} Y$	QSL	Magic wand connecting the random variables X and Y
$X \mathrel{-\!\!\circledast} Y$	FSL	Truncated magic wand connecting the random variables X and Y

Alphabetical Index

at-least predicate, 140

bound
- greatest lower, 12
- least upper, 12

cardinal, 19
cardinality, 19
chain, 16
- ω-, 17
- ordinal-, 17

classical separation logic, *see also* separation logic
closure of a set, 142
cyclic proof, 167
- global soundness criterion, 167

finite truncated summation, 81
fixed point, 14
- greatest, 15
- least, 15
- post-, 14
- pre-, 14

folding, 163
free variable, 42, 115
function
- ω-Scott-continuous, 17
- antitone, 11
- co-ω-Scott-continuous, 17
- image, 141
- monotone, 11

fuzzy separation logic, 84, 149
- entailment, 84
- free variable, 94
- precise, 95
- random variable, 84
- separating multiplication, 87
- specification, 113
- truncated fuzzy magic wand, 87

heap, 27
- disjoint, 28
- partial order, 28
- subset, 28
- union, 28

heap size, 172

inference rule, 165
infimum, 13
infinite truncated summation, 82

logical variable, 27

Markov chain, 51
- cylinder set, 52
- final state, 53
- probability space, 52
- reachability, 53

Markov decision process, 54
- Bellman equation, 55
- enabled action, 54
- final states, 55
- induced Markov chain, 55
- reward, 55
- scheduler, 54

one minus operation, 78
order
- complete lattice, 13
- complete linear, 13
- linear, 12
- partial, 10

ordinal, 19

partial correctness, 36
postcondition, 36
pre-proof, 165
precondition, 36
probability space, 50
- distribution, 50
- event space, 50
- expected value, 51
- probability function, 50
- random variable, 51
- sample space, 50

program
- cPL, 32
- cpPL, 98
- pPL, 57

program state, 27

qualitative entailment, 30
qualitative separation logic, *see also* separation logic
quantitative separation logic, 61, 161
- entailment, 62

parameterized random variable, 161
quantitative magic wand, 61
random variable, 61
separating multiplication, 61
specification, 68

separation logic, 29, 148
entailment, 30
free variable, 42
precise, 42
specification, 39
sigma algebra, 50
size
fuzzy separation logic, 151
separation logic, 151
stack, 27
stack heap pair, *see also* program state
substitution
for values of logical variables, 28
in a heap, 28
in a stack, 28
on a formula, 161
supremum, 13

terminating atomic, 42, 115
trace, 167
trace value, 167
truncated addition, 78
truncated division, 77
truncated subtraction, 79
truncated summation, *see also* infinite truncated summation

unfolding, 163

we, *see also* weakest expectation
weakest expectation, 67
weakest resource-safe liberal expectation, 105
weakest resource-safe liberal precondition, 36
written variable, 42, 115
wrlp, *see also* weakest resource-safe liberal precondition