Automated Deductive Verification of Probabilistic Programs

Φ

Kevin Batz

Automated Deductive Verification of Probabilistic Programs

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

vorgelegt von

Kevin Stefan Batz, M. Sc.
aus Grevenbroich

Berichter: Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen
Prof. Ichiro Hasuo, Ph.D.

Tag der mündlichen Prüfung: 20.12.2024

Abstract

We study both foundational and practical aspects of the automated deductive verification of discrete probabilistic programs. Our special emphasis is on the verification of possibly unbounded loops by quantitative loop invariants. We build upon Kozen's, McIver & Morgan's, and Kaminski's weakest preexpectation calculus for reasoning about expected outcomes such as the probability of terminating in some postcondition or the expected final value of a program variable.

The weakest preexpectation calculus replaces predicates from classical program verification by more general *expectations* — functions mapping program states to numbers instead of truth values. We study *a syntax for specifying expectations*, providing a foundation for automated deductive probabilistic program verifiers. We prove that our syntax is *expressive* and hence obtain a *relatively complete verification system* for reasoning about expected outcomes.

We then present three different approaches for *automating* the verification of bounds on expected outcomes of linear loops, all of which combine weakest preexpectation reasoning with Satisfiability Modulo Theories (SMT) techniques:

1. We revisit two well-established verification techniques for transition systems, k-induction and bounded model checking (BMC), in the more general setting of bounding least fixpoints of functions over complete lattices, yielding *latticed k-induction* and *latticed BMC*. Instantiating our latticed techniques with the weakest preexpectation calculus enables the fully automatic verification of linear loops taken from the literature.

2. We present a counterexample-guided inductive synthesis approach for the *synthesis of quantitative loop invariants*. This enables the fully automatic verification of bounds on both expected outcomes and expected runtimes. Our implementation synthesizes quantitative invariants of various linear loops taken from the literature, can beat state-of-the-art probabilistic model checkers on finite-state programs, and is competitive with modern tools dedicated to invariant synthesis or expected runtime reasoning.

3. We present PrIC3 — the first truly quantitative extension of the state-of-the-art qualitative symbolic model checking algorithm IC3 to symbolic model checking of Markov decision processes. By marrying PrIC3 with the weakest preexpectation calculus, we obtain a symbolic model checking algorithm for probabilistic programs. Alongside, we present an implementation of PrIC3 featuring IC3-specific techniques such as generalization.

Zusammenfassung

Wir studieren sowohl grundlegende als auch praktische Aspekte der automatisierten deduktiven Verifikation von diskreten probabilistischen Programmen. Unser besonderer Schwerpunkt liegt auf der Verifikation von möglicherweise unbeschränkten Schleifen mittels quantitativer Schleifeninvarianten. Wir bauen auf Kozens, McIver & Morgans und Kaminskis Kalkül der Schwächsten Vorerwartung auf, um erwartete Ergebnisse, wie z.B. die Wahrscheinlichkeit, in einer Nachbedingung zu terminieren oder den erwarteten finalen Wert einer Programmvariablen, zu ermitteln.

Der Kalkül der Schwächsten Vorerwartung ersetzt Prädikate aus der klassischen Programmverifikation durch allgemeinere Erwartungen — Funktionen, die Programmzustände auf Zahlen anstelle von Wahrheitswerten abbilden. Wir untersuchen eine Syntax zur Spezifikation von Erwartungen, die eine Grundlage für automatisierte deduktive probabilistische Programmverifizierer liefert. Wir beweisen, dass unsere Syntax aussagekräftig ist und erhalten so ein relativ vollständiges Verifikationssystem für erwartete Ergebnisse.

Anschließend stellen wir drei verschiedene Ansätze für die automatisierte Verifikation von Schranken auf erwartete Ergebnisse linearer Schleifen. Diese Ansätze kombinieren den Kalkül der Schwächsten Vorerwartung mit Techniken der Erfüllbarkeit Modulo Theorien:

1. Wir betrachten zwei etablierte Verifikationstechniken für Übergangssysteme, k-Induktion und begrenzte Modellüberprüfung (BMÜ), im allgemeineren Rahmen der Begrenzung kleinster Fixpunkte von Funktionen über vollständigen Verbänden. Dies führt zu *vergitterter k-Induktion* und *vergitterter BMÜ* führt. Die Instanziierung unserer vergitterten Techniken mit dem Kalkül der Schwächsten Vorerwartung ermöglicht die vollautomatische Verifikation von linearen Schleifen, die aus der Literatur stammen.

2. Wir präsentieren einen gegenbeispielgeleiteten induktiven Syntheseansatz für die Synthese quantitativer Schleifeninvarianten. Dies ermöglicht die vollautomatische Verifikation von Schranken auf erwartete Ergebnisse und erwartete Laufzeiten. Unsere Implementierung synthetisiert quantitative Invarianten verschiedener linearer Schleifen aus der Literatur, kann modernste probabilistische Modellprüfer für Programme mit endlichen Zustandsräumen schlagen und ist mit modernen Werkzeugen konkurrenzfähig, die sich der Invariantensynthese oder dem berechnen erwarteter Laufzeiten widmen.

3. Wir präsentieren PrIC3 — die erste wirklich quantitative Erweiterung des modernen qualitativen symbolischen Modellprüfungsalgorithmus IC3 zur symbolischen Modellprüfung von Markov-Entscheidungsprozessen. Durch das Verbinden von PrIC3 mit dem Kalkül der Schwächsten Vorerwartung erhalten wir einen symbolischen Modellprüfungsalgorithmus für probabilistische Programme. Wir stellen eine Implementierung von PrIC3 vor, die IC3-spezifische Techniken, wie z.B. Generalisierung, beinhaltet.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor Joost-Pieter Katoen. Joost-Pieter's guidance was invaluable. He introduced me to the extremely fascinating field of formal verification, he was there whenever I needed him, and created an amazing working environment. Joost-Pieter, I think you are an outstanding boss and supervisor because you truly care about your people. Pursuing a PhD with you teaches your students way more than writing papers. You invite numerous guests, actively support collaborations, and enable amazing business trips while sharing your enormous experience with us. Thank you for this extraordinarily enriching journey. I would also like to thank Ichiro Hasuo for reviewing my thesis and for providing me with valuable feedback. Ichiro also enabled one of the most awesome scientific and cultural experiences of my life by inviting me to his JST Aspire workshop in Japan.

I am grateful to Benjamin Kaminski, Christoph Matheja, and Sebastian Junges. Benjamin and Christoph supervised my bachelor's thesis back in 2017. Benjamin, Christoph, and Sebastian supervised my master's thesis in 2019. Since then, we have successfully continued to work together. Benni, Christoph, Sebastian, you have not only been extremely helpful and approachable but also showed me right from the beginning of my career that work can be both productive and fun at the same time. Thank you for sharing your knowledge with me and for all those exhausting-yet-satisfactory deadline sprints! I have always been (and still am) looking forward to every business trip where we meet again.

Lutz Klinkenberg, what we've been joking about during our first semester at RWTH Aachen actually came true: We became office mates! Thank you for all the laughs and "deep scientific chats" we shared. These chats have lightened up my everyday life at work significantly. Tobias "Tobais" Winkler, my former roommate, colleague, and friend, thank you for all the laughs we shared, for the fruitful discussions, and for our collaborations. Be it skiing in Kleinwalsertal or exploring the nightlife of Argentina or Boston, the time with you, Lutz and Tobi, was just amazing.

Tim Quatmann, thank you for guiding Lutz, Tobi, and myself as the world's best mentor for first-year students, for all the help you provided during our time as colleagues, and for all the (not always) scientific chats and refreshing meetings at the coffee machine.

It was my pleasure to supervise brilliant B.Sc. and M.Sc. students while they have been working on their thesis. Thank you, Florian Keßler, Adrian Gallus, Marvin Jansen, Philipp Schröer, Ben Sturgis, Tom Biskup, Nora Orhan, Daniel Zilken, Dinis Vitorino, and Daniel Basgöze for sharing your insights with me. I would also like to thank my former student assistants Philipp Schröer, Adrian Gallus, and Nora Orhan as well as our interns Tom Biskup and Zhiang Wu. Collaborating with you was a great experience.

I would like to thank my colleagues Alexander Bork, Bahare Salmani, Christina Gehnen, Christopher Brix, Daniel Cloerkes, Daniel Zilken, Darion Haase, Èlèanore Meyer, Hannah Mertens, Ira Fesefeldt, Jana Berger, Jan-Christoph Kassing, Jip Spel, Joshua Moerman, László Antal, Lena Verscht, Jasper Nalbach, József Kovács, Lina Gerlach, Matthias Volk, Marcel Hark, Mingshuai Chen, Mojgan Kamali, Nicolai Radke, Nils Lommen, Philipp Schroer, Raphael Berthon, Roy Hermanns, Stefan Dollase, Thomas Noll, and Valentin Promies for being so helpful and approachable. I will certainly miss all those spontaneous pleasant conversations and our awesome trips to Kleinwalsertal with the MOVES group!

Thank you, Marcel Moosbrugger, Stefano Maria Nicoletti, Felix Walter, and Bram Kohlen for the awesome trips we experienced together. It was a pleasure to meet you. Special thanks go to Pedro R. D'Argenio for hosting me at the National University of Córdoba in such a hospitable way.

I would like to thank my dear friends Annika Jüttner, Cathrin Simon, Daniela Arndt, Jasmin Dahmen, Joshua Stenz, Karen Pelzer, Lennart Friederichs, Leo Martin Moll, Lukas Fehst, Maximiliane Heyer, Norman Heinze, Pablo Guardia Martinez, Paul Stenz, Sarah Spicker, and Tom Stenz for attending my defense and/or the subsequent celebration. With you guys, it was a truly amazing day that still resonates with me.

Finally, it is hard to express my gratefulness to my beloved parents Monika Batz and Hartmut Batz as well as my beloved grandmother Brigitte Batz for their unconditional love and support throughout my life.

Feedback and LaTeX Sources. I am indebted to Joost-Pieter Katoen and Ichiro Hasuo for providing me with feedback on the entire thesis. Moreover, I am indebted to Benjamin Kaminski for providing me with feedback on the abstract, Chapter 1, Sections 3.1 and 3.2, and to Christoph Matheja for providing me with feedback on Section 1.1. I would like to thank Tobias Winkler for fruitful discussions on Markov decision processes. Finally, I am grateful to Sebastian Junges, Tim Quatmann, and Matthias Volk for providing me with the LaTeX sources of their thesis template.

Contents

1 Introduction

Soft- and hardware systems are ubiquitous. They steer trains, planes, cars, medical devices, and more. Ensuring their *correctness* is notoriously hard yet indispensable as our society is increasingly reliant on these systems, employing them in evermore safety-critical contexts. The aim of *formal methods* is to prove or disprove — *in a mathematically rigorous sense* — that these systems work correctly.

Many soft- and hardware systems are subject to uncertainty and randomness. Environments exhibit stochastic behavior, yielding sensor data to be noisy or communication channels to be lossy. Some systems explicitly exploit randomness, e.g., for increasing their performance or for breaking symmetries in anonymous networks. Formal methods must hence be able to quantify the impact of uncertainty and randomness on such systems. In this thesis, we contribute to the *automated deductive verification of probabilistic programs* — a branch of formal methods suitable for tackling the aforementioned tasks.

1.1 Classical Deductive Program Verification

Verifying classical non-probabilistic programs means answering questions like:

Does a given program terminate on all inputs?
Does the output of a given program always comply with the desired result?

Deductive program verification techniques establish such properties about the input-output behavior of programs by means of logical inference. This involves (i) formalizing the desired property about the program's input-output behavior as a mathematical statement and (ii) applying a deductive proof system to establish the validity of these statements. The most prevalent deductive program verification technique — nowadays often referred to as *Floyd-Hoare Logic* — originated from the ground-breaking works by Robert Floyd [Flo67] and Sir Antony Hoare [Hoa69]. In Floyd-Hoare Logic, the central objects are triples — often referred to as *Hoare triples* — of the form

$\langle P \rangle \; C \; \langle Q \rangle \quad$ where

1. P is a predicate over program states referred to as the *precondition*,
2. C is a program, and
3. Q is a predicate over program states referred to as the *postcondition*.

One distinguishes between *partial* and *total correctness*: The triple $\langle P\rangle\ C\ \langle Q\rangle$ is *valid for partial correctness*, if all terminating executions of C on initial states $\sigma \in P$ from the precondition yield final states $\tau \in Q$ from the postcondition. Partial correctness does not require C to terminate. If, additionally, C terminates on all initial states $\sigma \in P$, then the triple is *valid for total correctness*, i.e.,

$$\text{partial correctness} + \text{termination} \quad = \quad \text{total correctness}\,.$$

Proving a Hoare triple valid for partial or total correctness is the key task in deductive program verification. This can be challenging. Simply executing C on all states from the precondition P is not an option: P might — and usually *does* — contain *infinitely many* states. Even for a *single* initial state σ, determining whether executing C on σ results in a final state from the postcondition can be challenging and is generally undecidable — a consequence of Alan Turing's famous result on the undecidability of the halting problem [Tur37].

Hoare therefore proposed a deductive proof system — a collection of axioms and inference rules — to establish the validity of Hoare triples. These axioms and rules formalize intuitive proof strategies. Consider, for instance, a Hoare triple involving a conditional choice:

$$\langle P\rangle\ \texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_2\}\ \langle Q\rangle$$

The guard B determines which of the two branches C_1 and C_2 is going to be executed. It is therefore intuitive to split the task of proving the triple valid into two sub-tasks by distinguishing the cases where either of the branches is executed. Hoare's inference rule for conditional choices formalizes this strategy:

$$\frac{\overbrace{\langle P \wedge B\rangle\ C_1\ \langle Q\rangle \qquad \langle P \wedge \neg B\rangle\ C_2\ \langle Q\rangle}^{\text{premisses of the rule}}}{\underbrace{\langle P\rangle\ \texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_2\}\ \langle Q\rangle}_{\text{conclusion of the rule}}}$$

Repeatedly applying such rules of inference in a bottom-up manner gives rise to a proof tree, decomposing the original program into ever smaller sub-programs.

If all the leaves of such a tree are axioms, i.e., triples whose validity requires no further justification such as

$$\overline{\langle P \rangle \ \mathtt{skip} \ \langle P \rangle} \quad (\mathtt{skip} \text{ is the immediately terminating effectless program})$$

then the original Hoare triple is proven to be valid.

From Theory to Automation. Floyd-Hoare Logic provides a systematic and syntax-directed approach to program verification. This made it the formal basis of various modern automated program verifiers. These software tools are either *semi-automated*, meaning that the user has to provide additional hints on how to construct the aforementioned proof trees, or *fully automatic*. There is a trade-off between the degree of automation and the supported programming language features and specifications. This trade-off is due to the following two main challenges of automated deductive program verification:

Reasoning about loops is, unsurprisingly, one of the most intricate tasks. Proving partial correctness of loops requires coming up with suitable *loop invariants*. Additionally proving termination is tackled via so-called *loop variants*. Intuitively speaking, loop (in)variants are predicates over program states summarizing a loop's execution (termination) behavior.

Second, pre- and postconditions as well as loop (in)variants are typically represented as logical formulae over the program variables. For instance, validity for total correctness of $\langle \exists i \in \mathbb{Z} \colon 2 \cdot i = x \rangle \ C \ \langle x = 0 \rangle$ specifies that C terminates in a final state where x is equal to 0 whenever x initially stores an even number. Reasoning about pre- and postconditions or establishing the validity of loop (in)variants requires possibly complex arithmetical reasoning. This challenge is mostly tackled by offloading the arithmetical reasoning to *Satisfiability Modulo Theories (SMT) solvers*, which are software tools dedicated to this task. The rapidly increasing capabilities of SMT solvers such as Z3 [MB08], CVC5 [BBBK+22], or MathSat5 [CGSS13] play an important role in the success of automated deductive program verification. The required arithmetical theories are often undecidable. Depending on the expressive power of the employed theories, users must provide the SMT solver with additional hints.

Prime examples of modern verifiers supporting high-level programming language features such as generic classes or dynamic data structures include Dafny [Lei10] and Prusti [ABFG+22]. Dafny and Prusti target the verification of real-world programs written in high-level programming languages. They are semi-automated in the sense that the user must — on top of the desired specifications — provide additional verification hints such as loop invariants. Tools targeting *full* automation by, e.g., trying to *automatically synthesize* loop invariants include CPAchecker [BK11], SeaHorn [GKKN15], and Infer [CDDG+15]. Full automation typically comes at the cost of restrictions on the supported arithmetical theories, programming language features, and specifications.

1.2 Probabilistic Programs

Probabilistic programs are ordinary programs whose execution may depend on the outcomes of random experiments such as coin flips. Randomized computations have a long-standing history [LMSS56] and a plethora of applications.

Probabilistic programs describe *randomized algorithms*, which exploit randomness for the purpose of, e.g., solving computational problems more efficiently in expectation or for rendering certain computational problems at all feasible. Hoare's randomized variant of his Quicksort algorithm [Hoa62] is a prime example of the former. Examples of the latter include scenarios that require some kind of symmetry-breaking such as self-stabilizing protocols [Her90].

Probabilistic programs moreover provide a powerful means for modeling complex stochastic processes. This is extensively leveraged in the field of *probabilistic model checking* [BK08; Kat16; BHK19], where probabilistic programs — written, e.g., in the PRISM-language[1] — provide concise modular descriptions of Markov chains or more general Markov decision processes. Most modern probabilistic models checkers, including PRISM [CHLP10; HKNP06; KNP02] and Storm [HJKQ+22; DJKV17], support the PRISM-language.

Probabilistic programs are capable of modeling complex (conditional) probability distributions in a structured programmatic manner. This has led to the rapidly emerging field of *probabilistic programming*, where the goal is to "[...] enable probabilistic modeling and machine learning to be accessible to the working programmer" [GHNR14]. In probabilistic programming, the main task is *probabilistic inference*: Given a (conditional) probability distribution encoded as a probabilistic program, compute an explicit representation of that distribution or properties thereof such as its mode or the expected value of a given random variable. Applications of probabilistic programming and their associated inference methods are spread over various fields of research, including cognitive science [GTC16], biology [RKSB+21], and artificial intelligence [Gha15].

[1] https://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction

1.3 Deductive Verification of Probabilistic Programs

Introducing randomization into computations has severe consequences. Rather than producing a single (if any) final state, a probabilistic program produces a *probability (sub)distribution* of final states. Reasoning about probabilistic programs hence becomes inherently *quantitative*. Typical questions include:

What is the probability of terminating in a given final state?
What is the probability of terminating in some postcondition?
What is the expected final value of a program variable?

We refer to such quantities as *expected outcomes of probabilistic programs*. Reasoning about expected outcomes is hard. For instance, the question of whether a probabilistic program terminates with probability 1 *on a given initial state* is computationally as hard as the question of whether a non-probabilistic program terminates *on all states* [KK15; KKM19; Kam19].

As with classical deductive program verification, the deductive verification of probabilistic programs involves (i) formalizing the desired property of a probabilistic program as a mathematical statement and (ii) applying some kind of logical inference — in the form of program calculi and proof rules — for establishing the validity of these statements. Seminal works on this subject date back to the 1970s [Ram79] and 1980s [HSP82; Koz83; Koz85] and has been an active area of research since. Approaches targeting the deductive verification of probabilistic programs can roughly be classified into *distribution*- and *expectation*-based techniques. For distribution-based techniques, the central objects are (predicates over) probability distributions, enabling to explicitly reason about input-output distributions of probabilistic programs, see, e.g., [Har99; HV02; BEGG^{+}18; RZ15; LAH23; CCMS07]. For expectation-based techniques, the central objects are *random variables* (functions mapping program states to numbers), enabling to reason more directly about specific expected outcomes of probabilistic programs. Expectation-based reasoning as pioneered by Kozen [Koz83; Koz85] and McIver & Morgan [MM05; MMS96] is central to many results presented in this thesis and will be detailed in Section 2.4. We refer to [BKS20a] for a state-of-the-art overview of the theoretical underpinnings of probabilistic programs and their associated reasoning techniques.

1.4 Contributions and Synopsis

In this thesis, we contribute both foundational and practical aspects of the automated deductive verification of probabilistic programs. We build upon the *weakest preexpectation calculus* by Kozen [Koz83; Koz85], McIver & Morgan [MMS96; MM05], and Kaminski [Kam19] — a quantitative extension of Dijkstra's weakest precondition calculus [Dij75]. Our special emphasis is on reasoning about possibly unbounded loops by combining Satisfiability Modulo Theories techniques with quantitative loop invariants.

Chapter 2 equips us with the required foundations. Our main contributions are presented in Chapters 3 to 6. Each of these chapters contains a dedicated introduction, an outline, and a section on future and related work. In what follows, we briefly summarize the contents and key contributions of each chapter.

Chapter 2: Foundations. We recap the foundations we build upon in this thesis by discussing elementary results from fixpoint theory, Markov decision processes, probabilistic programs, and the weakest preexpectation calculus. We emphasize (i) the role of fixpoint theory in the latter fields and (ii) connections between notions from Markov decision processes and weakest preexpectations.

Chapter 3: Relatively Complete Verification [12]. The weakest preexpectation calculus replaces predicates from classical program verification by more general *expectations* — functions mapping program states to numbers instead of truth values. As with classical program verification, automating the verification of probabilistic programs requires a syntax for such "quantitative assertions". We present an *expressive formal language of expectations* for the weakest preexpectation calculus, obtaining a *relatively complete verification system* for reasoning about expected outcomes. Our language is a cornerstone of the modern semi-automated deductive probabilistic program verifier CAESAR[2] [18].

Chapter 4: Latticed *k*-Induction [11]. k-induction [SSS00] and bounded model checking (BMC) [CBRZ01] are two well-established verification techniques for transition systems integrated into various automated soft- and hardware verifiers. As pointed out by Krishnan et al. [KVGG19]:

"The simplicity of applying k-induction made it the go-to technique for SMT-based infinite-state model checking."

We revisit k-induction and BMC in the more general setting of bounding least fixpoints of monotone functions over complete lattices. Instantiating our latticed techniques with the weakest preexpectation calculus enables the fully

[2] CAESAR itself will not be presented in this thesis.

automatic SMT-based verification of piecewise linear bounds on expected outcomes of possibly unbounded linear[3] loops. Our implementation manages to automatically verify specifications of loops taken from the literature. Moreover, the generality of our latticed techniques has led to more applications discovered by other researchers: Latticed k-induction has been leveraged by Winkler and Katoen [WK23a] in the context of probabilistic pushdown automata and by Yang et al. [YFKZ$^+$24] for the synthesis of quantitative loop invariants.

Chapter 5: Automatic Loop Invariant Synthesis [16]. We present an SMT-based counterexample-guided inductive synthesis approach for quantitative loop invariants. This enables the fully automatic verification of piecewise linear bounds on both expected outcomes and expected runtimes. Our implementation finds invariants of various linear loops taken from the literature, can beat state-of-the-art probabilistic model checkers, and is competitive with modern tools dedicated to the synthesis of quantitative invariants or expected runtimes.

Chapter 6: Property Directed Reachability [6]. Aaron R. Bradley's algorithm IC3 [Bra11a] has been a leap forward in symbolic model checking of transition systems. We first recap the fundamental concepts underlying IC3. We then present PrIC3 — the first truly quantitative extension of IC3 to symbolic model checking of Markov decision processes. Moreover, by leveraging tight connections between Markov decision processes and probabilistic programs, we marry PrIC3 with the weakest preexpectation calculus, obtaining a symbolic SMT-based model checking algorithm for finite-state probabilistic programs. Alongside, we present an implementation of PrIC3 including key ingredients for IC3's scalability such as (inductive) generalization and propagation.

1.5 Origins

The results presented in Chapters 3 to 6 are based on prior publications I have co-authored, each of which is cited in the respective chapters. Below I give a list of all publications I have co-authored as a doctoral researcher at RWTH Aachen University, divided into the publications covered in this thesis and further publications. The university's regulations for doctoral studies moreover obliges me to discuss my contributions to these publications. This discussion can be found in Appendix 4.

[3] i.e., the loop's body is loop-free and all arithmetic is linear, see Definition 4.3 on page 146.

Publications Covered in this Thesis.

[16] K. Batz, M. Chen, S. Junges, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants.*" *TACAS (2)*. Volume 13994. Lecture Notes in Computer Science. Springer, 2023, pages 410–429

[12] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-based Reasoning.*" *Proc. ACM Program. Lang.* 5.POPL (2021), pages 1–30

[11] K. Batz, M. Chen, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. "*Latticed k-Induction with an Application to Probabilistic Programs.*" *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 524–549

[6] K. Batz, S. Junges, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. "*PrIC3: Property Directed Reachability for MDPs.*" *CAV (2)*. Volume 12225. Lecture Notes in Computer Science. Springer, 2020, pages 512–538

Further Publications.

[19] K. Batz, T. J. Biskup, J.-P. Katoen, and T. Winkler. "*Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs.*" *Proc. ACM Program. Lang.* 8.POPL (2024), pages 2792–2820

[18] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*A Deductive Verification Infrastructure for Probabilistic Programs.*" *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pages 2052–2082

[17] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and L. Verscht. "*A Calculus for Amortized Expected Runtimes.*" *Proc. ACM Program. Lang.* 7.POPL (2023), pages 1957–1986

[15] K. Batz, A. Gallus, B. L. Kaminski, J.-P. Katoen, and T. Winkler. "*Weighted Programming: A Programming Paradigm for Specifying Mathematical Models.*" *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pages 1–30

[14] K. Batz, I. Fesefeldt, M. Jansen, J.-P. Katoen, F. Keßler, C. Matheja, et al. "*Foundations for Entailment Checking in Quantitative Separation Logic.*" *ESOP*. Volume 13240. Lecture Notes in Computer Science. Springer, 2022, pages 57–84

[9] L. Klinkenberg, K. Batz, B. L. Kaminski, J.-P. Katoen, J. Moerman, and T. Winkler. "*Generating Functions for Probabilistic Programs.*" *LOPSTR*. Volume 12561. Lecture Notes in Computer Science. Springer, 2020, pages 231–248

[5] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and T. Noll. "*Quantitative Separation logic: A Logic for Reasoning about Probabilistic Pointer Programs.*" *Proc. ACM Program. Lang.* 3.POPL (2019), 34:1–34:29

[2] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times.*" *ESOP*. Volume 10801. Lecture Notes in Computer Science. Springer, 2018, pages 186–213

2 Foundations

This chapter treats the foundations we build upon in this thesis. In Section 2.1, we study the (order-theoretic) theory of fixpoints. In Section 2.2, we study Markov decision processes and expected rewards. Section 2.3 introduces the probabilistic programming language considered throughout this thesis. Finally, in Section 2.4, we introduce the deductive program verification techniques which we extend and automate in the subsequent chapters.

Basic Notation. We denote the *set of natural numbers including* 0 by

$$\mathbb{N} = \{0, 1, 2, \ldots\} .$$

The *set of rationals* is denoted by $\mathbb{Q}$. The *set of reals* is denoted by $\mathbb{R}$. The *set of non-negative rationals* and the *set of non-negative reals* are respectively defined as

$$\mathbb{Q}_{\geq 0} = \{q \in \mathbb{Q} \mid q \geq 0\} \qquad \text{and} \qquad \mathbb{R}_{\geq 0} = \{\alpha \in \mathbb{R} \mid \alpha \geq 0\} .$$

Moreover, the *set of non-negative extended reals* is defined as

$$\mathbb{R}_{\geq 0}^{\infty} = \{\alpha \in \mathbb{R} \mid \alpha \geq 0\} \cup \{\infty\} .$$

Finally, if both the domain and the codomain are clear from the context, we often use *lambda expressions* to denote functions, i.e, function $\lambda x. f$ applied to some argument a evaluates to f in which x is replaced by a.

2.1 Fixpoint Theory

In this section, we treat the concepts from (order-theoretic) fixpoint theory over complete lattices which provide us with the mathematical foundations for reasoning about transition systems, Markov decision processes, and probabilistic programs in a unified manner. We start with a motivating example borrowed from the propositional μ-calculus [Koz82] in Section 2.1.1, which will serve us as a running example. In Section 2.1.2, we consider *partial orders and complete lattices*. Finally, in Sections 2.1.3 to 2.1.5, we study *(least) fixpoints of endofunctions* over complete lattices. We follow the presentation from [AJ95].

2.1.1 Motivating Example: Reachability in Transition Systems

We motivate the concepts related to fixpoint theory employed in this thesis by means of reachability in transition systems. Transition systems[1] are a prominent state-based model to formally describe the behavior of, e.g., hardware- and software systems. We mainly follow the presentation from [BK08, Chapter 2].

Definition 2.1 (Transition Systems).
A *transition system* is a structure

$$\mathsf{TS} \;=\; (\mathcal{S}, \longrightarrow, \mathcal{S}_I)\,,$$

where:

1. $\mathcal{S}$ is the countable set of *states*,
2. $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the total *transition relation*, i.e.,
   $$\text{for all } s \in \mathcal{S}\colon \text{ exists } t \in \mathcal{S}\colon \quad (s,t) \in \longrightarrow\,,$$
3. $\mathcal{S}_I \subseteq \mathcal{S}$ is the set of *initial states*.

We often write $s \longrightarrow t$ instead of $(s,t) \in \longrightarrow$.

If $s \longrightarrow t$, then we say that t is a *(direct) successor* of s. If $\mathcal{S}$ is finite, then we say that TS is *finite-state*. Otherwise, we say that TS is *infinite-state*. If every $s \in \mathcal{S}$ has only finitely many successors, then we say that TS is *finitely branching*. Otherwise, we say that TS is *infinitely branching*.

Intuitively, a transition system TS models the possible executions of a system. The set $\mathcal{S}_I$ contains the initial states where execution may start and the transition relation $\longrightarrow$ describes possible state changes, i.e., how the system can evolve from one state to another.

Example 2.1.
Transition systems can be represented as graphs, where the nodes correspond to the states and the edges determine the transition relation. The graph shown in Figure 2.1 on page 13 represents the transition system

$$\mathsf{TS1} \;=\; (\mathcal{S}, \longrightarrow, \mathcal{S}_I)\,,$$

where

1. $\mathcal{S} = \{s_0, \ldots, s_6\}$,

[1] Transition systems are often referred to as Kripke structures [Kri63].

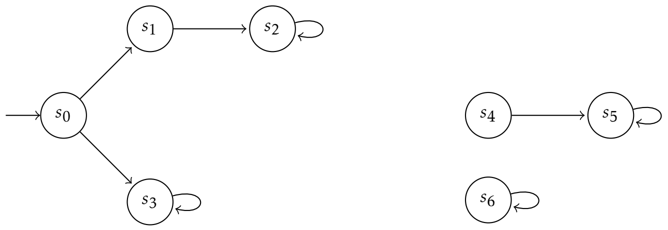


Figure 2.1: The finite-state and finitely branching transition system TS1.

2. $s \longrightarrow s'$ iff there is an edge from s to s', and
3. $\mathcal{S}_I = \{s_0\}$.

Notice that $\mathcal{S}$ is *nondeterministic*: There are two outgoing edges from s_0, meaning that system execution *either* proceeds in state s_1 *or* proceeds in s_3.

One of the most important notions for transition systems is *reachability*: Assuming that system execution starts in some initial state and evolves according to the transition relation, which states of the system can be reached by some execution? We are thus interested in the set of states reachable by some execution of the system. For the transition system from Example 2.1, this set is given by $\{s_0, s_1, s_2, s_3\}$. In what follows, we discuss two different, yet equivalent, approaches for defining this set. Section 2.1.1.1 provides an operational perspective: We formally define the possible executions described by a given transition system. In Section 2.1.1.2, we define the set of reachable states by means of a recursive equation from which we derive a characterization of the set of reachable states as the least (in a sense we will make precise) fixpoint of a certain function. This will give rise to the main concepts and questions for which fixpoint theory provides us with an answer.

2.1.1.1 Reachability via Execution Fragments

Fix a transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$. A *finite execution fragment π of* TS is a finite, non-empty sequence of states

$$\pi \;=\; s_0 \ldots s_n \qquad \text{such that} \qquad \text{for all } i \in \{0, \ldots, n-1\}\colon s_i \longrightarrow s_{i+1}\ .$$

Similarly, an *infinite execution fragment π of* TS is an *infinite* sequence of states

$$\pi \;=\; s_0 s_1 s_2 \ldots \qquad \text{such that} \qquad \text{for all } i \in \mathbb{N} \colon s_i \longrightarrow s_{i+1}~.$$

If for a (finite or infinite) execution fragment π we have $s_0 \in \mathcal{S}_I$, then π is called an *initial* (finite or infinite) execution fragment of TS. A state $s \in \mathcal{S}$ is called *reachable (in* TS*)* if there is an initial finite execution fragment $s_0 \ldots s_n$ with $s_n = s$. Otherwise, s is called *unreachable (in* TS*)*. Finally, we define the set Reach(TS) of *reachable states* as

$$\mathsf{Reach}(\mathsf{TS}) \;=\; \{s \in \mathcal{S} \mid s \text{ is reachable in TS}\}$$

and the set $\mathsf{Reach}^{\leq n}(\mathsf{TS})$ of *states reachable in at most $n \in \mathbb{N}$ steps* as

$$\begin{aligned} &\mathsf{Reach}^{\leq n}(\mathsf{TS}) \\ =\; &\{s \in \mathcal{S} \mid \text{there is } m \in \{0, \ldots, n\} \\ &\qquad\qquad \text{and initial finite execution fragment } s_0 \ldots s_m \colon s_m = s\}~. \end{aligned}$$

Notice that

$$\mathsf{Reach}(\mathsf{TS}) \;=\; \bigcup_{n \in \mathbb{N}} \mathsf{Reach}^{\leq n}(\mathsf{TS})~,$$

i.e., Reach(TS) contains all states reachable in an unbounded number of steps.

2.1.1.2 Reachability via Least Fixpoints

Fix a transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$. We start with an *informal* recursive characterization of what it means for a state s to be reachable. This informal characterization is given by the observation that

A state $s \in \mathcal{S}$ is *reachable*
iff
s is an initial state or s is a successor of some *reachable* state t.

The above characterization is recursive since the term “reachable” occurs on both sides of the equivalence. Put more formally, “s is reachable” is a *predicate* $X \subseteq \mathcal{S}$ over states, which satisfies the *recursive equation*

$$\underbrace{X}_{\text{the set of reachable states}} \;=\; \underbrace{\mathcal{S}_I}_{\text{contains all initial states}} \;\cup\; \underbrace{\{s \mid \text{exists } t \in X \colon t \longrightarrow s\}}_{\text{and is closed under successors}}~. \tag{2.1}$$

The *solution domain* [Hüt10] for the above equation is

$$\mathcal{P}(\mathcal{S}) \;=\; \{X \mid X \subseteq \mathcal{S}\}\;,$$

i.e., the *power set of* $\mathcal{S}$ and the set Reach(TS) of reachable states defined in Section 2.1.1.1 is indeed a solution of Equation (2.1). There is, however, generally *not a unique* solution of Equation (2.1). Only the *least* (in a sense we make precise below) coincides with Reach(TS).

In the framework of fixpoint theory, we formalize solutions of Equation (2.1) as fixpoints of a function over the solution domain $\mathcal{P}(\mathcal{S})$:

Definition 2.2 (Reachability Operators).
Let $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ be a transition system. We call

$$\Phi_{\mathsf{TS}} \colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}), \qquad \Phi_{\mathsf{TS}}(X) \;=\; \mathcal{S}_I \cup \{s \mid \text{exists } t \in X \colon t \longrightarrow s\}$$

the *reachability operator (of* TS*)*.

In words, applying Φ_{TS} to some set X of states gives the set of initial states plus all direct successors of states in X. A set of states X is a *fixpoint* of Φ_{TS}, if

$$\Phi_{\mathsf{TS}}(X) \;=\; X$$

and we say that X is a solution of Equation (2.1) if X is a fixpoint Φ_{TS}.

Example 2.2.
Reconsider the transition system TS1 from Figure 2.1. There are exactly 5 fixpoints of Φ_{TS1} and thus 5 solutions of Equation (2.1):

1. $X_1 = \mathcal{S}$,
2. $X_2 = \{s_0, s_1, s_2, s_3, s_4, s_5\}$,
3. $X_3 = \{s_0, s_1, s_2, s_3, s_5\}$,
4. $X_4 = \{s_0, s_1, s_2, s_3, s_6\}$, and
5. $X_5 = \{s_0, s_1, s_2, s_3\}$.

Only X_5 is equal to the set Reach(TS1) of reachable states as defined in Section 2.1.1.1. Notice that X_5 is the *least* fixpoint of Φ_{TS1} in the sense that for all fixpoints $X \in \{X_1, \ldots, X_5\}$, we have $X_5 \subseteq X$.

Function Φ_{TS} might have several fixpoints. Given the solution domain $\mathcal{P}(\mathcal{S})$ and the order $\subseteq$ on this domain, we are interested in the *least* fixpoint of Φ_{TS} w.r.t. $\subseteq$

and only this least fixpoint coincides with the set $\mathsf{Reach}(\mathsf{TS})$ of reachable states as defined in Section 2.1.1.1. The intuition is that

the set $\mathsf{Reach}(\mathsf{TS})$ of reachable states is the *smallest* set of states which contains all initial states and is closed under successors.

The above approach of defining the set of reachable states as the least fixpoint of a function over a given solution domain is in fact an instance of a general framework. We will see in Section 2.2 that tightly related ideas give rise to constructions of minimal and maximal expected rewards in Markov decision processes, and in Section 2.4.3 that similar ideas lead to constructions of minimal and maximal expected outcomes of probabilistic programs [MM05; Kam19]. These approaches always follow the same scheme:

1. Define the solution domain D.
2. Define a suitable order $\sqsubseteq$ on D.
3. Define a function $\Phi\colon D \to D$.
4. *Ensure* that Φ has a unique least fixpoint.

The latter aspect is one of the key applications of fixpoint theory as it is employed in this thesis. Fixpoint theory provides us with general and sufficient conditions on D, $\sqsubseteq$, and Φ which guarantee that the least fixpoint of Φ exists uniquely. Furthermore, fixpoint theory provides us with generic techniques for *constructing and approximating* least fixpoints in a systematic manner. These techniques apply, amongst others, to transition systems, Markov decision processes, and probabilistic programs and they underlie *all* approaches for automating the deductive verification of probabilistic systems presented in this thesis.

We proceed as follows: In Section 2.1.2, we introduce *partial orders* and *complete lattices* — notions providing us with said sufficient conditions on D and $\sqsubseteq$. In Section 2.1.3, we consider sufficient conditions for the existence of *least fixpoints* of functions $\Phi\colon D \to D$. Finally, Sections 2.1.4 and 2.1.5 treat the systematic construction and approximation of least fixpoints.

2.1.2 Partial Orders and Complete Lattices

We start by introducing *partial orders* — sets equipped with a binary relation satisfying certain well-behavedness conditions. These conditions enable us to sensibly speak about, e.g., least fixpoints of functions.

Definition 2.3 (Partial Orders).
Let D be a set and let $\sqsubseteq \subseteq D \times D$. The structure

$$(D, \sqsubseteq)$$

is called a *partial order*, if the following conditions hold:

1. $\sqsubseteq$ is *reflexive*, i.e.,
$$\text{for all } d \in D\colon \quad d \sqsubseteq d\,,$$
2. $\sqsubseteq$ is *antisymmetric*, i.e.,
$$\text{for all } d, e \in D\colon \quad d \sqsubseteq e \text{ and } e \sqsubseteq d \quad \text{implies} \quad d = e\,,$$
3. $\sqsubseteq$ is *transitive*, i.e.,
$$\text{for all } d, e, e' \in D\colon \quad d \sqsubseteq e \text{ and } e \sqsubseteq e' \quad \text{implies} \quad d \sqsubseteq e'\,.$$

If $d \sqsubseteq e$, then we say that e *upper-bounds* d or that d *lower-bounds* e.

Example 2.3.
1. Both the non-negative rationals and the non-negative reals equipped with the usual “at most”-relation are partial orders, i.e., both
$$(\mathbb{Q}_{\geq 0}, \leq) \qquad \text{and} \qquad (\mathbb{R}_{\geq 0}, \leq)$$
are partial orders.
2. Partial orders can be constructed via power sets: For *any* set M, the structure $(\mathcal{P}(M), \subseteq)$ is a partial order, where $\mathcal{P}(M)$ is the power set of M and where $\subseteq$ is the subset-relation. In particular, every transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ induces the partial order $(\mathcal{P}(\mathcal{S}), \subseteq)$.

Now fix a partial order $(D, \sqsubseteq)$ and let $S \subseteq D$. An element $d \in D$ is called an *upper bound for S*, if d upper-bounds every element of S, i.e., if

$$\text{for all } e \in S\colon \quad e \sqsubseteq d\,.$$

If there is a *least* upper bound for S, i.e., if there is $d \in D$ satisfying

$$d \text{ is upper bound of } S \text{ and for all upper bounds } e \text{ of } S \text{ we have } d \sqsubseteq e\,,$$

then d is called the *supremum of S*. Antisymmetry of $\sqsubseteq$ implies that if the supremum of S exists, then it is unique and we denote it by $\bigsqcup S$. Dually, $d \in D$ is called a *lower bound for S*, if d lower-bounds every element of S, i.e., if

$$\text{for all } e \in S\colon \quad d \sqsubseteq e\,.$$

If there is a *greatest* lower bound for S, i.e., if there is $d \in D$ satisfying

$$d \text{ is lower bound of } S \text{ and for all lower bounds } e \text{ of } S \text{ we have } e \sqsubseteq d\,,$$

then d is called the *infimum of S*. As with suprema, if the infimum of S exists, then it is unique and we denote it by $\bigsqcap S$.

Example 2.4.

1. For the partial order $(\mathbb{Q}_{\geq 0}, \leq)$, the set
$$S \;=\; \{q \in \mathbb{Q}_{\geq 0} \mid q \cdot q < 2\}$$
has infinitely many upper bounds, namely all $r \in \mathbb{Q}_{\geq 0}$ greater than $\sqrt{2}$. There is, however, no *least* upper bound in $\mathbb{Q}_{\geq 0}$ since $\sqrt{2} \notin \mathbb{Q}_{\geq 0}$.

2. For the partial order $(\mathcal{P}(\mathcal{S}), \subseteq)$ induced by a transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$, suprema and infima of $S \subseteq \mathcal{P}(\mathcal{S})$ are given by set union and set intersection, respectively, i.e.,
$$\bigsqcup S \;=\; \bigcup_{X \in S} X \qquad \text{and} \qquad \bigsqcap S \;=\; \bigcap_{X \in S} X\,.$$

We usually restrict to partial orders where suprema and infima are guaranteed to exist. Such partial orders are called *complete lattices*:

Definition 2.4 (Complete Lattices).
A partial order $(D, \sqsubseteq)$ is called a *complete lattice*, if every $S \subseteq D$ has a supremum $\bigsqcup S \in D$ and an infimum $\bigsqcap S \in D$.

In particular, every complete lattice $(D, \sqsubseteq)$ has a *least element* $\bot$ given by $\bot = \bigsqcup \emptyset$, and a *greatest element* $\top$ given by $\top = \bigsqcap \emptyset$.

Example 2.5.

1. We have seen in Example 2.4 that $(\mathbb{Q}_{\geq 0}, \leq)$ is not a complete lattice. The partial order $(\mathbb{R}_{\geq 0}, \leq)$ is not a complete lattice either since it has no greatest element. We do, however, obtain a complete lattice by adding a greatest element, which we denote by ∞, thereby obtaining the *non-negative extended reals*
$$\mathbb{R}_{\geq 0}^{\infty} \;=\; \{\alpha \in \mathbb{R} \cup \{\infty\} \mid \alpha \geq 0\}\,.$$
The structure $(\mathbb{R}_{\geq 0}^{\infty}, \leq)$ is a complete lattice with $\bot = 0$ and $\top = \infty$.

2. Every partial order constructed via power sets (cf. Example 2.3) is a complete lattice. In particular, the partial order $(\mathcal{P}(\mathcal{S}), \subseteq)$ induced by

some transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ is a complete lattice with

$$\bot = \emptyset \qquad \text{and} \qquad \top = \mathcal{S}\,.$$

2.1.3 Existence of Least Fixpoints

We introduce the notion of (least) fixpoints of functions over complete lattices — a central notion for this thesis.

Definition 2.5 (Fixpoints).
Let $(D, \sqsubseteq)$ be a complete lattice and $\Phi\colon D \to D$.

1. An element $d \in D$ is called a *fixpoint of* Φ, if $\Phi(d) = d$.
2. A fixpoint $d \in D$ of Φ is called the *least fixpoint of* Φ, if

   $$\text{for all fixpoints } d' \text{ of } \Phi\colon \quad d \sqsubseteq d'\,.$$

   If it exists, we denote the least fixpoint of Φ by $\mathsf{lfp}\, \Phi$.

Antisymmetry of $\sqsubseteq$ implies that if the least fixpoint of Φ exists, then it is unique, which justifies the above notation.

Example 2.6.
Recall from Definition 2.2 that every transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ induces the reachability operator $\Phi_{\mathsf{TS}}\colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ over the complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$. For the transition system TS1 from Example 2.1, Φ_{TS1} has 5 fixpoints given by the sets $X_1, \ldots, X_5$ from Example 2.2 — all solutions of Equation (2.1). The least fixpoint $\mathsf{lfp}\, \Phi_{\mathsf{TS1}}$ is X_5 — the least solution.

The fact that the function Φ_{TS1} from Example 2.6 has a least fixpoint is not a coincidence but follows from the fact that it satisfies a property called *monotonicity* — a sufficient condition for the existence of the least fixpoint:

Definition 2.6 (Monotone Functions).
Let $(D, \sqsubseteq)$ be a complete lattice. A function

$$\Phi\colon D \to D$$

is called *monotone (w.r.t. $(D, \sqsubseteq)$)*, if

$$\text{for all } d, e \in D\colon \quad d \sqsubseteq e \quad \text{implies} \quad \Phi(d) \sqsubseteq \Phi(e)\,.$$

Alfred Tarski's fixpoint theorem states that every monotone function over a complete lattice has a least fixpoint.

Theorem 2.1 (Tarski's Fixpoint Theorem [Tar55]).
Let $(D, \sqsubseteq)$ be a complete lattice. Every monotone function $\Phi\colon D \to D$ has a least fixpoint given by

$$\mathsf{lfp}\,\Phi \;=\; \bigsqcap \{d \in D \mid \Phi(d) \sqsubseteq d\}\;.$$

Proof. Write $S = \{d \in D \mid \Phi(d) \sqsubseteq d\}$. First notice that $\bigsqcap S$ exists because $(D, \sqsubseteq)$ is a complete lattice. To prove $\mathsf{lfp}\,\Phi = \bigsqcap S$, one shows that $\bigsqcap S$ is a fixpoint of Φ. Since S contains all fixpoints of Φ by reflexivity of $\sqsubseteq$, this yields

$$\text{for all fixpoints } d' \text{ of } \Phi\colon \qquad \bigsqcap S \sqsubseteq d'$$

and thus $\mathsf{lfp}\,\Phi = \bigsqcap S$. Now, for proving that $\Phi(\bigsqcap S) = \Phi(\bigsqcap S)$ holds, it suffices to show that both $\Phi(\bigsqcap S) \sqsubseteq \bigsqcap S$ and $\bigsqcap S \sqsubseteq \Phi(\bigsqcap S)$ hold by antisymmetry of $\sqsubseteq$. To see that $\Phi(\bigsqcap S) \sqsubseteq \bigsqcap S$ holds, consider the following:

$$\begin{array}{lll}
 & \text{for all } d' \in S\colon \bigsqcap S \sqsubseteq d' & \text{(definition of infima)}\\
\text{implies} & \text{for all } d' \in S\colon \Phi\big(\bigsqcap S\big) \sqsubseteq \Phi(d') & \text{(monotonicity of } \Phi)\\
\text{implies} & \text{for all } d' \in S\colon \Phi\big(\bigsqcap S\big) \sqsubseteq \Phi(d') \sqsubseteq d' & \\
 & & \text{(definition of } S \text{ and transitivity of } \sqsubseteq)\\
\text{implies} & \Phi\big(\bigsqcap S\big) \text{ is lower bound for } S & \text{(definition of infima)}\\
\text{implies} & \Phi\big(\bigsqcap S\big) \sqsubseteq \bigsqcap S & (\bigsqcap S \text{ is greatest lower bound for } S)
\end{array}$$

To see that $\bigsqcap S \sqsubseteq \Phi(\bigsqcap S)$ holds, consider the following:

$$\begin{array}{lll}
 & \Phi\big(\bigsqcap S\big) \sqsubseteq \bigsqcap S & \text{(see above)}\\
\text{implies} & \Phi\Big(\Phi\big(\bigsqcap S\big)\Big) \sqsubseteq \Phi(\bigsqcap S) & \text{(monotonicity of } \Phi)\\
\text{implies} & \Phi\big(\bigsqcap S\big) \in S & \text{(definition of } S)\\
\text{implies} & \bigsqcap S \sqsubseteq \Phi\big(\bigsqcap S\big) & \text{(definition of infima)}
\end{array}$$

This completes the proof. ∎

If $\Phi(d) \sqsubseteq d$, then d is called a *prefixpoint of* Φ. Theorem 2.1 thus states that the least fixpoint of a monotone function Φ is the infimum of all prefixpoints.

Example 2.7.
Reconsider the reachability operator Φ_{TS} induced by some transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ (Definition 2.2). The reachability operator is monotone [Koz82] because for $X, X' \in \mathcal{P}(\mathcal{S})$ with $X \subseteq X'$, we have

$$\begin{aligned}
& \Phi_{\mathsf{TS}}(X) \\
={}& \mathcal{S}_I \cup \{s \mid \text{exists } t \in X \colon t \longrightarrow s\} && \text{(by definition)} \\
\subseteq{}& \mathcal{S}_I \cup \{s \mid \text{exists } t \in X' \colon t \longrightarrow s\} && (t \in X \text{ implies } t \in X') \\
={}& \Phi_{\mathsf{TS}}(X') \,. && \text{(by definition)}
\end{aligned}$$

Hence, by Theorem 2.1, Φ_{TS} has a least fixpoint.

2.1.4 Constructive Fixpoint Theorems

Even though Theorem 2.1 guarantees the existence of least fixpoints of monotone functions over complete lattices, it is not particularly constructive. In what follows, we discuss Kleene's fixpoint theorem (Theorem 2.2) and its generalization by Cousot & Cousot (Theorem 2.3), which provide us with more constructive characterizations of least fixpoints.

Kleene's fixpoint theorem requires the notion of *continuity*:

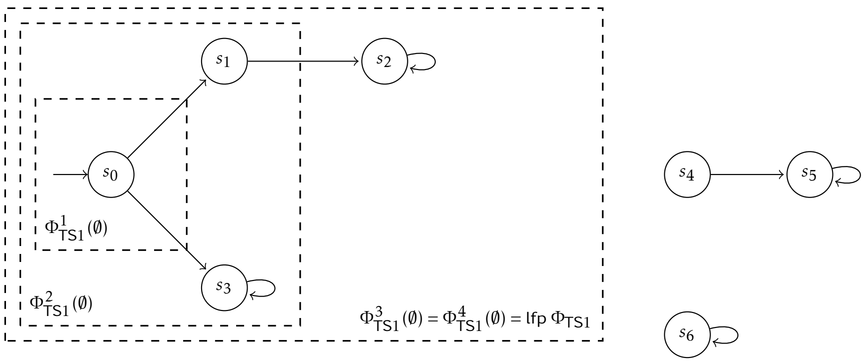
Definition 2.7 (Continuous Functions [Win93]).
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi \colon D \to D$. We say that Φ is *continuous (w.r.t. $(D, \sqsubseteq)$)*, if for all increasing ω-chains $S = \{d_0 \sqsubseteq d_1 \sqsubseteq \ldots\} \subseteq D$, we have

$$\Phi\left(\bigsqcup S\right) = \bigsqcup \left\{\Phi(d) \mid d \in S\right\}.$$

In words, continuity states that taking suprema of ω-chains S and applying Φ commute: Applying Φ to $\bigsqcup S$ yields the same result as taking the supremum of the set obtained from applying Φ to every element of S.

Now let $(D, \sqsubseteq)$ be a complete lattice, let $\Phi \colon D \to D$, and $d \in D$. Given a natural number n, we denote by $\Phi^n(d)$ the *(upper) n-fold iteration of Φ on d* defined as

$$\Phi^n(d) = \begin{cases} d & \text{if } n = 0 \\ \Phi\left(\Phi^{n-1}(d)\right) & \text{if } n > 0\,. \end{cases}$$



s_0
s_1
s_2
s_3
s_4
s_5
s_6
$\Phi^1_{\mathsf{TS1}}(\emptyset)$
$\Phi^2_{\mathsf{TS1}}(\emptyset)$
$\Phi^3_{\mathsf{TS1}}(\emptyset) = \Phi^4_{\mathsf{TS1}}(\emptyset) = \mathsf{lfp}\ \Phi_{\mathsf{TS1}}$

Figure 2.2: Fixpoint iterates for determining the set $\mathsf{Reach}(\mathsf{TS1}) = \mathsf{lfp}\ \Phi_{\mathsf{TS1}}$ of reachable states of the transition system TS1 from Example 2.1.

Theorem 2.2 (Kleene's Theorem [Win93]).
Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi\colon D \to D$. If Φ is continuous, then

$$\mathsf{lfp}\ \Phi \;=\; \bigsqcup \{\Phi^n(\bot) \mid n \in \mathbb{N}\}\ .$$

Theorem 2.2 states that the least fixpoint $\mathsf{lfp}\ \Phi$ of a continuous function Φ is given by the supremum of the set obtained from iterating Φ on the least element $\bot$. This iterative process, i.e., determining $\Phi^n(\bot)$ for increasing n is called a *fixpoint iteration* and we call $\Phi^n(\bot)$ the *n-th iterate of* Φ. It is worthwhile to note that continuity of Φ implies monotonicity of Φ, which implies that

$$\left\{\bot \sqsubseteq \Phi(\bot) \sqsubseteq \Phi^2(\bot) \sqsubseteq \ldots\right\}$$

is an increasing ω-chain. Hence, a fixpoint iteration yields increasingly precise lower bounds on $\mathsf{lfp}\ \Phi$, and iterating — so to speak — *ad infinitum* yields $\mathsf{lfp}\ \Phi$.

Example 2.8.
Reconsider the transition system $\mathsf{TS1} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ from Example 2.1 and the reachability operator $\Phi_{\mathsf{TS1}}\colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ from Definition 2.2. The reachability operator Φ_{TS1} can be shown to be continuous[a]. The least element of the complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$ is $\bot = \emptyset$ and suprema correspond

to set union. Hence, Theorem 2.2 yields

$$\mathsf{lfp}\,\Phi_{\mathsf{TS1}} \;=\; \bigcup\left\{\Phi^{n}_{\mathsf{TS1}}(\emptyset) \mid n \in \mathbb{N}\right\},$$

i.e., the set $\mathsf{Reach}(\mathsf{TS1})$ of reachable states is given by the above supremum obtained from fixpoint iteration. This fixpoint iteration is depicted in Figure 2.2. Notice that determining $\Phi^{n}_{\mathsf{TS1}}(\emptyset)$ for increasing n corresponds to performing a breadth-first search from the initial states in the graph representing the transition system. This fixpoint iteration *stabilizes* after finitely many steps, i.e., we have $\Phi^{3}_{\mathsf{TS1}}(\emptyset) = \Phi^{4}_{\mathsf{TS1}}(\emptyset)$, which implies

$$\begin{aligned}
&\text{for all } n \geq 3\colon\ \Phi^{n}_{\mathsf{TS1}}(\emptyset) &&=\; \Phi^{n+1}_{\mathsf{TS1}}(\emptyset)\\
\text{which implies}\quad &\bigcup\left\{\Phi^{n}_{\mathsf{TS1}}(\emptyset) \mid n \in \mathbb{N}\right\} &&=\; \Phi^{3}_{\mathsf{TS1}}(\emptyset)\\
\text{and thus}\quad &\mathsf{lfp}\,\Phi_{\mathsf{TS1}} \;=\; \mathsf{Reach}(\mathsf{TS1}) &&=\; \Phi^{3}_{\mathsf{TS1}}(\emptyset)\,.
\end{aligned}$$

For finite-state transition systems, stabilization after finitely many fixpoint iterations is guaranteed. For infinite-state transition systems this is, however, not the case. Consider the simple infinite-state transition system TS2 depicted in Figure 2.3 as an example: For this transition system, we have

$$\text{for all } n \geq 1\colon\quad \Phi^{n}_{\mathsf{TS2}}(\emptyset) \;=\; \{s_0, \ldots, s_{n-1}\}\,,$$

which implies that

$$\{\Phi^{0}_{\mathsf{TS2}}(\emptyset) \subset \Phi^{1}_{\mathsf{TS2}}(\emptyset) \subset \Phi^{2}_{\mathsf{TS2}}(\emptyset) \subset \ldots\}$$

is a *strictly* increasing ω-chain and the supremum

$$\bigcup\left\{\Phi^{n}_{\mathsf{TS2}}(\emptyset) \mid n \in \mathbb{N}\right\} \;=\; \{s_n \mid n \in \mathbb{N}\} \;=\; \mathsf{lfp}\,\Phi_{\mathsf{TS2}}$$

is attained only after infinitely many fixpoint iterations.

[a]This is, in fact, the case for *every* transition system, not just the concrete one considered in this example. See, e.g., [Koz82; Fon08].

Remark 2.1.
There is a general connection between the $(n+1)$-th fixpoint iterate $\Phi^{n+1}_{\mathsf{TS}}(\emptyset)$ and the set $\mathsf{Reach}^{\leq n}(\mathsf{TS})$ of states reachable in at most n steps, namely

$$\text{for all } n \in \mathbb{N}\colon\quad \Phi^{n+1}(\bot) \;=\; \mathsf{Reach}^{\leq n}(\mathsf{TS})\,.$$

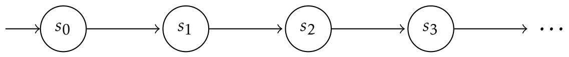


Figure 2.3: The simple infinite-state transition system TS2.

Theorem 2.2 thus enables us to see that $\mathsf{lfp}\ \Phi_{\mathsf{TS}}$ and $\mathsf{Reach}(\mathsf{TS})$ coincide since

$$\mathsf{lfp}\ \Phi \;=\; \bigcup\left\{\Phi^{n}_{\mathsf{TS}}(\emptyset) \mid n \in \mathbb{N}\right\} \;=\; \bigcup\left\{\mathsf{Reach}^{\leq n}(\mathsf{TS}) \mid n \in \mathbb{N}\right\} \;=\; \mathsf{Reach}(\mathsf{TS})~.$$

Hence, Theorem 2.2 provides a link between the operational perspective on reachability from Section 2.1.1.1 and the fixpoint perspective presented here. This link often helps to understand a least fixpoint construction and to prove its correctness w.r.t. some ground truth. In Section 2.2.4 we proceed analogously to understand least fixpoint constructions of minimal and maximal expected rewards in Markov decision processes.

Theorem 2.2 thus provides us with a constructive characterization of least fixpoints in case the function Φ under consideration is continuous. This raises the question: Is there a similarly constructive characterization of $\mathsf{lfp}\ \Phi$ in case Φ is monotone *but not necessarily continuous*? This situation occurs naturally:

Example 2.9.
Let $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ be a transition system and let $\mathcal{T} \subseteq \mathcal{S}$ be a set of *target states*. Adopting notation from linear temporal logic (LTL) [BK08, Chapter 5], define the set $\Diamond\mathcal{T} \subseteq \mathcal{S}$ of states that eventually (and definitely, i.e., regardless of which transitions are taken) reach a state in $\mathcal{T}$ as

$$\Diamond\mathcal{T} \;=\; \{s_0 \in \mathcal{S} \mid \text{for all infinite execution fragments } \pi = s_0 s_1 s_2 \ldots\colon \\ \text{exists } i \in \mathbb{N}\colon s_i \in \mathcal{T}\}~.$$

The set $\Diamond\mathcal{T}$ is the least fixpoint of $\Theta_{\mathsf{TS},\mathcal{T}}\colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ defined as[a]

$$\Theta_{\mathsf{TS},\mathcal{T}}(X) \;=\; \mathcal{T} \cup \{s \in \mathcal{S} \mid \text{for all direct successors } t \text{ of } s\colon t \in X\}$$

over the complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$ induced by TS. The intuition is that

$\Diamond\mathcal{T}$ is the *smallest* set of states which contains all target states from $\mathcal{T}$ and all states s for which *all* direct successors t are in $\Diamond\mathcal{T}$.

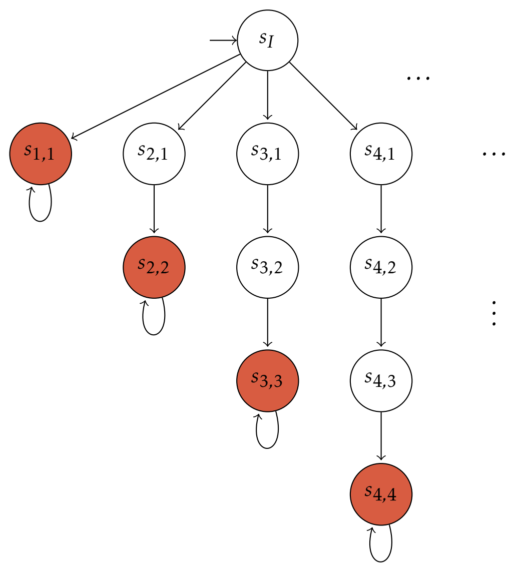


Figure 2.4: The infinite-state and infinitely branching transition system TS3 adapted from [Koz82].

Even though $\Theta_{\mathsf{TS},\mathcal{T}}$ is monotonic, continuity is not guaranteed [Dij76; Koz82]. Consider the infinite-state transition system $\mathsf{TS3} = (\mathcal{S}, \longrightarrow, \{s_I\})$ depicted in Figure 2.4. TS3 is *infinitely branching* since s_I has infinitely many successors. Each of these transitions leads to a subsystem of increasing length. Now fix the set of target states (colored in red)

$$\mathcal{T} \;=\; \{s_{i,i} \mid i \geq 1\} \qquad \text{for which we have} \qquad \Diamond\mathcal{T} \;=\; \mathcal{S}\,.$$

To see that $\Theta_{\mathsf{TS3},\mathcal{T}}$ is not continuous, define for each $n \in \mathbb{N}$ the set $\Diamond^{\leq n}\mathcal{T}$ of states which (definitely) reach a state in $\mathcal{T}$ in *at most* n steps as

$$\Diamond^{\leq n}\mathcal{T} \;=\; \{s_0 \in \mathcal{S} \mid \text{for all infinite execution fragments } \pi = s_0 s_1 s_2 \ldots\colon \\ \text{exists } i \leq n\colon s_i \in \mathcal{T}\}\,.$$

Clearly,

$$\left\{\Diamond^{\leq 0}\mathcal{T} \subseteq \Diamond^{\leq 1}\mathcal{T} \subseteq \Diamond^{\leq 2}\mathcal{T} \subseteq \ldots\right\} \;=\; \left\{\Diamond^{\leq n}\mathcal{T} \mid n \in \mathbb{N}\right\}$$

is an increasing ω-chain and we have

$$\bigcup\left\{\Diamond^{\leq n}\mathcal{T} \mid n \in \mathbb{N}\right\} \;=\; \mathcal{S} \setminus \{s_I\}\,,$$

i.e., for all states s *except* for the initial state s_I, there is some $n \in \mathbb{N}$ such that s (definitely) reaches a state in $\mathcal{T}$ in at most n steps. The above ω-chain yields a counterexample to continuity of $\Theta_{\mathsf{TS3},\mathcal{T}}$ since

$$\begin{aligned}
&\Theta_{\mathsf{TS3},\mathcal{T}}\Big(\bigcup\left\{\Diamond^{\leq n}\mathcal{T} \mid n \in \mathbb{N}\right\}\Big) \\
=\;& \Theta_{\mathsf{TS3},\mathcal{T}}(\mathcal{S} \setminus \{s_I\}) \\
=\;& \mathcal{S} && \text{(since } \textit{all} \text{ direct successors of } s_I \text{ are in } \mathcal{S} \setminus \{s_I\}) \\
\neq\;& \mathcal{S} \setminus \{s_I\} \\
=\;& \bigcup\left\{\Diamond^{\leq n+1}\mathcal{T} \mid n \in \mathbb{N}\right\} \\
=\;& \bigcup\left\{\Theta_{\mathsf{TS3},\mathcal{T}}\left(\Diamond^{\leq n}\mathcal{T}\right) \mid n \in \mathbb{N}\right\}.
\end{aligned}$$

[a]This follows from the fact that $\Diamond\mathcal{T}$ is the least solution of the *expansion law* of LTL [BK08, Lemma 5.18].

Cousot & Cousot [CC79] provide a generalization of Theorem 2.2 for possibly non-continuous but monotone functions. For that, we need to generalize the

n-fold iteration of Φ on some element $d \in D$ to *transfinite* ordinals: Given an ordinal $\mathfrak{n}$, we denote by $\Phi^{\mathfrak{n}}(d)$ the *(upper) $\mathfrak{n}$-fold iteration of Φ on d* defined by transfinite recursion as[2]

$$\Phi^{\mathfrak{n}}(d) = \begin{cases} d & \text{if } \mathfrak{n} = 0\,, \\ \Phi\left(\Phi^{\mathfrak{m}}(d)\right) & \text{if } \mathfrak{n} = \mathfrak{m}+1 \text{ is a successor ordinal}\,, \\ \bigsqcup\{\Phi^{\mathfrak{m}}(d) \mid \mathfrak{m} < \mathfrak{n}\} & \text{if } \mathfrak{n} \text{ is a limit ordinal}\,. \end{cases}$$

To gain some intuition on the above notion, consider the special case where $\mathfrak{n}$ is the first limit ordinal ω, which can be identified with the set $\mathbb{N}$ of natural numbers. The ω-fold iteration of Φ on d is given by

$$\Phi^{\omega}(d) = \bigsqcup\{\Phi^{n}(d) \mid n \in \mathbb{N}\}\,.$$

Hence, in case Φ is continuous, Theorem 2.2 can be stated more concisely as

$$\mathsf{lfp}\,\Phi = \Phi^{\omega}(\bot)\,.$$

Now consider the case where $\mathfrak{n}$ is the successor ordinal $\omega+1$. We have

$$\Phi^{\omega+1}(d) = \Phi\left(\Phi^{\omega}(d)\right) = \Phi\left(\bigsqcup\{\Phi^{n}(d) \mid n \in \mathbb{N}\}\right),$$

i.e., $\Phi^{\omega+1}(d)$ is obtained from first taking the supremum of all finite n-fold iterations of Φ on d (think: iterating Φ ω-times on d), and then applying Φ once more to the so-obtained element.

With this notion at hand, consider Cousot & Cousot's theorem:

Theorem 2.3 (Constructive Version of Tarski's Theorem [CC79]). Let $(D, \sqsubseteq)$ be a complete lattice and let $\Phi\colon D \to D$ be monotone. There

$$\text{exists ordinal } \mathfrak{n}\colon \quad \mathsf{lfp}\,\Phi = \Phi^{\mathfrak{n}}(\bot)\,.$$

Proof. We recommend [Ech05] for a concise proof. Let us recap the key ideas. One proves that there exists an ordinal $\mathfrak{n}$ whose cardinality exceeds the cardinality of D at which the transfinite iteration of Φ on $\bot$ stabilizes, i.e.,

$$\Phi(\Phi^{\mathfrak{n}}(\bot)) = \Phi^{\mathfrak{n}}(\bot)\,.$$

Hence, $\Phi^{\mathfrak{n}}(\bot)$ is a fixpoint of Φ. To see that $\Phi^{\mathfrak{n}}(\bot)$ is the least fixpoint of Φ,

[2] Following [CC79], we fix an ambient ordinal $\mathfrak{k}$, which is the smallest ordinal such that $|\mathfrak{k}| > |D|$, and tacitly assume $\mathfrak{n} < \mathfrak{k}$ for all ordinals $\mathfrak{n}$ considered throughout.

one shows by transfinite induction that for all fixpoints d' of Φ, we have

$$\text{for all ordinals } \mathfrak{m}\colon \quad \Phi^{\mathfrak{m}}(\bot) \sqsubseteq d' ,$$

which implies the claim by taking $\mathfrak{m} = \mathfrak{n}$. ■

Example 2.10.
Reconsider the transition system TS3 depicted in Figure 2.4 and the function $\Theta_{\mathsf{TS3},\mathcal{T}}$ from Example 2.9. We have seen that $\Theta_{\mathsf{TS3},\mathcal{T}}$ is monotone but *not* continuous and, indeed, Theorem 2.2 — Kleene's fixpoint theorem — does *not* apply: For each $n \in \mathbb{N}$, we have

$$\Theta^{n+1}_{\mathsf{TS3},\mathcal{T}}(\emptyset) = \Diamond^{\leq n}\mathcal{T},$$

i.e., the $(n+1)$-th fixpoint iterate is the set of states that definitely reach a state in $\mathcal{T}$ in at most n steps. Hence, we get

$$\Theta^{\omega}_{\mathsf{TS3},\mathcal{T}}(\emptyset) = \bigcup\left\{\Theta^{n}_{\mathsf{TS3},\mathcal{T}}(\emptyset) \mid n \in \mathbb{N}\right\} = \bigcup\left\{\Diamond^{\leq n}\mathcal{T} \mid n \in \mathbb{N}\right\} = \mathcal{S} \setminus \{s_I\}$$

The set $\mathcal{S} \setminus \{s_I\}$ is *not* the least fixpoint of Φ since we have

$$\Theta_{\mathsf{TS3},\mathcal{T}}(\mathcal{S} \setminus \{s_I\}) = \mathcal{S} .$$

The set $\mathcal{S}$, on the other hand, *is* the least fixpoint of $\Theta_{\mathsf{TS3},\mathcal{T}}$ (and indeed coincides with $\Diamond\mathcal{T}$). We thus need $\omega+1$ fixpoint iterations to obtain $\mathsf{lfp}\ \Phi$, i.e.,

$$\mathsf{lfp}\ \Theta_{\mathsf{TS3},\mathcal{T}} = \Theta^{\omega+1}_{\mathsf{TS3},\mathcal{T}}(\emptyset) = \Theta_{\mathsf{TS3},\mathcal{T}}(\Theta^{\omega}_{\mathsf{TS3},\mathcal{T}}(\emptyset)) .$$

Hence, Cousot & Cousot's Theorem 2.3 applies for $\mathfrak{n} = \omega + 1$.

2.1.5 Park Induction for Upper Bounds on Least Fixpoints

In the previous section, we have seen that least fixpoints are obtained via fixpoint iteration, which yields increasingly precise lower bounds on the least fixpoint of interest. In this section, we consider a proof rule — often referred to as *Park induction* — for establishing *upper bounds* on least fixpoints.

Lemma 2.4 (Park Induction [Par69]).
Let $(D, \sqsubseteq)$ be a complete lattice and $\Phi\colon D \to D$ be monotone. We have

$$\text{for all } d \in \Phi\colon \qquad \Phi(d) \sqsubseteq d \quad \text{implies} \quad \mathsf{lfp}\ \Phi \sqsubseteq d .$$

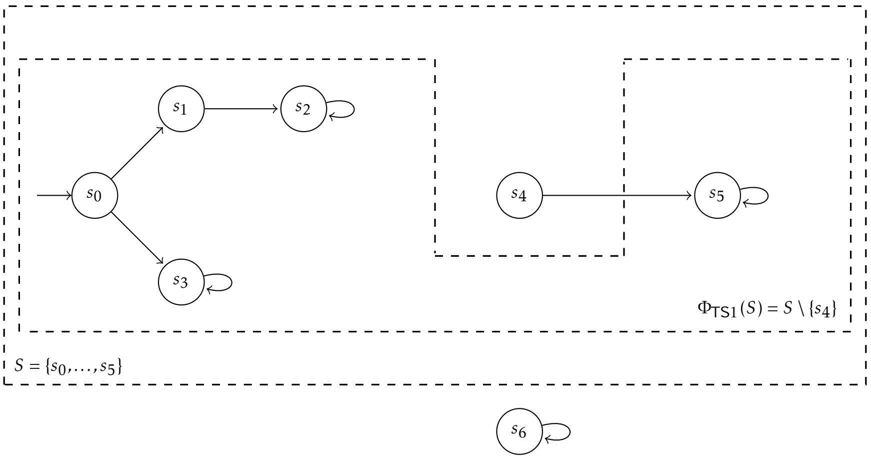

s_0
s_1
s_2
s_3
s_4
s_5
s_6
$\Phi_{\mathsf{TS1}}(S) = S \setminus \{s_4\}$
$S = \{s_0, \ldots, s_5\}$

Figure 2.5: Applying Lemma 2.4 to prove that the subset $S = \{s_0, \ldots, s_5\}$ of states contains all reachable states of the transition system TS1 from Example 2.1, i.e., that $\mathsf{Reach}(\mathsf{TS1}) \subseteq S$ holds.

Proof. This is an immediate consequence of Theorem 2.1: We have

$$d \in \{d \in D \mid \Phi(d) \sqsubseteq d\}$$

and thus

$$\mathsf{lfp}\ \Phi = \bigsqcap \{d \in D \mid \Phi(d) \sqsubseteq d\} \sqsubseteq d$$

by the definition of infima. ■

Even though $\mathsf{lfp}\ \Phi$ is a supremum of a possibly transfinite fixpoint iteration (cf. Theorem 2.3), proving that d upper-bounds $\mathsf{lfp}\ \Phi$ by Lemma 2.4 — if it applies — can be fairly easy: We only need to apply Φ *once* to d.

Example 2.11.
We apply Lemma 2.4 to prove that a given subset of states $S \subseteq \mathcal{S}$ of some transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ contains all reachable states: For the

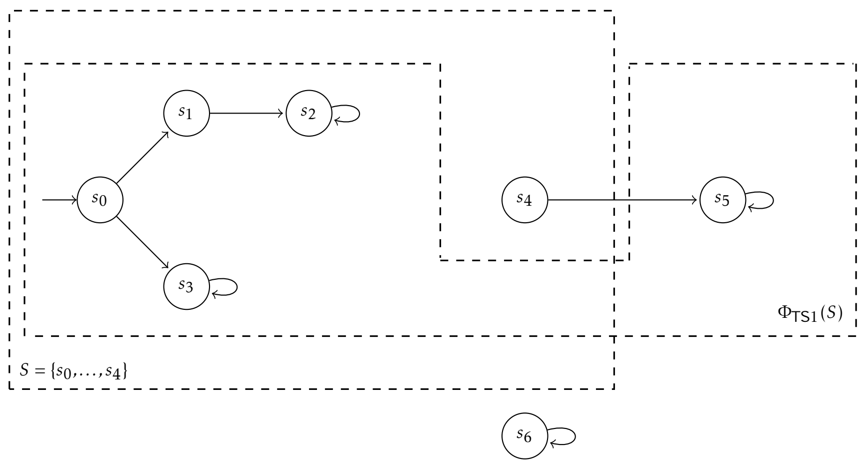


Figure 2.6: A counterexample to the converse direction of Lemma 2.4: We have $\mathsf{lfp}\ \Phi_{\mathsf{TS1}} \subseteq S$ but $\Phi_{\mathsf{TS1}}(S) \not\subseteq S$.

reachability operator $\Phi_{\mathsf{TS}} \colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ from Definition 2.2 we have

$$\mathsf{lfp}\ \Phi_{\mathsf{TS}} \;=\; \mathsf{Reach}(\mathsf{TS})\ .$$

Hence, by Lemma 2.4, if $\Phi_{\mathsf{TS}}(S) \subseteq S$, then $\mathsf{Reach}(\mathsf{TS}) \subseteq S$. Now consider the transition system TS1 depicted in Figure 2.5. We prove that $S = \{s_0, \ldots, s_5\}$ contains all reachable states of TS1. For that, we compute $\Phi_{\mathsf{TS1}}(S)$. This is easy since we simply need to collect all initial states and all successors of states in S. Since $\Phi_{\mathsf{TS1}}(S) \subseteq S$ holds, the claim follows by Lemma 2.4.

Importantly, Lemma 2.4 is *only a sufficient condition*, i.e.,

$$\lightning \quad \mathsf{lfp}\,\Phi \;\sqsubseteq\; d \qquad \text{implies} \qquad \Phi(d) \;\sqsubseteq\; d \quad \lightning$$

does not hold in general.

Example 2.12.
Consider the situation depicted in Figure 2.6, where $S = \{s_0, \ldots, s_4\}$. We

have $\mathsf{lfp}\ \Phi_{\mathsf{TS}} \subseteq S$ but $\Phi_{\mathsf{TS1}}(S) \not\subseteq S$.

Park induction can be seen as one of the fixpoint theoretic key principles underlying various state-of-the-art verification techniques such as, e.g., invariant-based reasoning for the verification of classical programs [Hoa69], probabilistic programs [MM05; Kam19], or hardware [MP95; Bra11a]. From a fixpoint theoretic perspective, unsoundness of the converse direction of Lemma 2.4 gives rise to one of the main challenges when employing these techniques. Regarding the deductive verification of probabilistic programs, the techniques presented in Chapters 4 to 6 tackle this challenge in distinct ways.

2.2 Markov Decision Processes

We consider countably infinite-state Markov decision processes (MDPs) and undiscounted expected rewards for $\mathbb{R}^{\infty}_{\geq 0}$-valued reward functions. MDPs are a suitable model for an operational semantics of probabilistic programs (cf. Section 2.3.2). The notion of expected rewards enables us to establish a connection between this operational semantics and the deductive verification techniques employed in this thesis (cf. Section 2.4.5).

We rely on least fixpoint characterizations of minimal and maximal expected rewards in MDPs, i.e., on the fact that expected rewards are given as the least solution of Bellman's optimality equations [Bel57]. These least fixpoint characterizations are well-understood in various settings (see, e.g., [Put94, Theorem 7.2.3a and Theorem 7.3.3a]). There is, however, a slight mismatch between the notion of expected rewards considered in the literature and the notion required in this thesis: For *maximal* expected rewards, our setting is closely related to what [Put94, Section 7.2] calls *positive bounded models*. [Put94, Section 7.2] assumes that both reward functions and corresponding maximal expected rewards are *pointwise smaller than* ∞ (cf. [Put94, Assumption 7.2.1]). We cannot rely on this assumption since ∞-valued maximal expected rewards occur naturally when reasoning about expected outcomes of probabilistic programs (cf. Remark 2.3). Similarly, for *minimal* expected rewards, our setting is closely related to what [Put94, Section 7.3] calls *negative models*. Even though ∞-valued minimal expected rewards are not excluded, reward functions are assumed to be pointwise smaller than ∞ — another assumption we cannot rely on.

For this reason, we employ the concepts from fixpoint theory presented in Section 2.1 to derive least fixpoint characterizations of possibly ∞-valued minimal and maximal expected rewards and prove these characterizations correct. We proceed as follows: In Section 2.2.1, we define MDPs alongside corresponding basic notions such as paths and schedulers. In Section 2.2.2, we introduce the notion of expected rewards in MDPs considered in this thesis and study their properties in Section 2.2.3. The least fixpoint characterizations of expected rewards are treated in Section 2.2.4. We then conclude with a note on the existence of *optimal* schedulers in Section 2.2.5.

2.2.1 Definition and Basic Notions

We consider finitely branching Markov decision processes (MDPs) with countable state spaces. MDPs can be thought of as a probabilistic extension of transition systems. We mainly follow the presentation from [BK08, Chapter 10].

Definition 2.8 (Markov Decision Processes).
A *Markov decision process (MDP)* is a structure

$$\mathcal{M} \;=\; (\mathcal{S}, \mathsf{Act}, P)\,,$$

where:

1. $\mathcal{S}$ is the countable *set of states*,
2. Act is the finite *set of actions*, and
3. $P\colon \mathcal{S} \times \mathsf{Act} \times \mathcal{S} \to [0,1]$ is the *transition probability function* satisfying
   a) for all $s \in \mathcal{S}$: for all $\mathfrak{a} \in \mathsf{Act}$:
   $$\{s' \in \mathcal{S} \mid P(s,\mathfrak{a},s') > 0\} \text{ is finite} \qquad \text{and} \qquad \sum_{s' \in \mathcal{S}} P(s,\mathfrak{a},s') \in \{0,1\}$$
   b) for all $s \in \mathcal{S}$: exists $\mathfrak{a} \in \mathsf{Act}$: $\sum_{s' \in \mathcal{S}} P(s,\mathfrak{a},s') = 1$.

We say that an action $\mathfrak{a}$ is *enabled* in state s if $\sum_{s' \in \mathcal{S}} P(s,\mathfrak{a},s') = 1$ and denote the finite *set of actions enabled in s* by $\mathsf{Act}(s)$. If $P(s,\mathfrak{a},s') > 0$, then s' is called an $\mathfrak{a}$-*successor of s* and we denote the finite *set of* $\mathfrak{a}$*-successors of s* by $\mathsf{Succs}_{\mathfrak{a}}(s)$. If $|\mathsf{Act}(s)| = 1$ for all $s \in \mathcal{S}$, then $\mathcal{M}$ is called a *Markov chain (MC)*. If $\mathcal{S}$ is finite, then we say that $\mathcal{M}$ is *finite-state*. Otherwise, we say that $\mathcal{M}$ is *infinite-state*.

Intuitively, an MDP $\mathcal{M}$ models an agent operating in a stochastic environment. Starting in some state s, the agent first nondeterministically chooses some enabled action $\mathfrak{a} \in \mathsf{Act}(s)$. The agent's next state is then determined probabilistically according to the probability distribution $P(s,\mathfrak{a},\cdot)$ over $\mathfrak{a}$-successors of s, i.e., the probability of moving to state s' is given by $P(s,\mathfrak{a},s')$. From there, the agent chooses the next enabled action, and so on. If $\mathcal{M}$ is a Markov chain, then

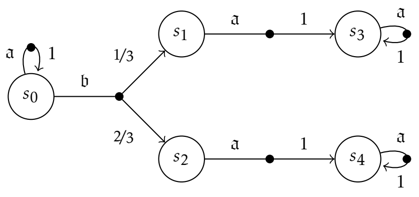


Figure 2.7: The finite-state MDP $\mathcal{M}1$.

this process is *fully probabilistic* in the sense that the agent never has a choice.

Example 2.13.
Similarly to transition systems, MDPs can be depicted as graphs. Consider the finite-state MDP $\mathcal{M}1 = (\mathcal{S}, \mathsf{Act}, P)$ depicted in Figure 2.7. We have $\mathcal{S} = \{s_0, \ldots, s_4\}$ and $\mathsf{Act} = \{\mathfrak{a}, \mathfrak{b}\}$. State s_0 has two enabled actions, i.e., $\mathsf{Act}(s_0) = \{\mathfrak{a}, \mathfrak{b}\}$. The transition probability function P is given by the labeled edges. For instance, when taking action $\mathfrak{b}$ in state s_0, the probability of transitioning to state s_1 is $1/3$, i.e., $P(s_0, \mathfrak{b}, s_1) = 1/3$.

Toward defining expected rewards in MDPs, we introduce the notion of paths, schedulers, and induced path probabilities.

Definition 2.9 (Paths).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathcal{T} \subseteq \mathcal{S}$.

1. A (finite) *path* in $\mathcal{M}$ is a finite, non-empty sequence of states
$$\pi \;=\; s_0 \ldots s_n$$
such that for all $i \in \{0, \ldots, n-1\}$ there is $\mathfrak{a} \in \mathsf{Act}(s_i)$ with $s_{i+1} \in \mathsf{Succs}_{\mathfrak{a}}(s_i)$.

2. We define the *set of paths starting in $s \in \mathcal{S}$ of length at most $n \in \mathbb{N}$* as
$$\mathsf{Paths}^{\leq n}(s) \;=\; \{s_0 \ldots s_m \text{ is a path in } \mathcal{M} \mid s_0 = s \text{ and } m \leq n\} \ .$$

3. We define the set of *set of all paths starting in state $s \in \mathcal{S}$* as
$$\mathsf{Paths}(s) \;=\; \bigcup_{n \in \mathbb{N}} \mathsf{Paths}^{\leq n}(s) \ .$$

4. We define the *set of paths starting in state $s \in \mathcal{S}$ and eventually reaching a state in $\mathcal{T}$ of length at most n* as
$$\begin{aligned}&\mathsf{Paths}^{\leq n}(s, \mathcal{T})\\ = &\left\{s_0 \ldots s_m \in \mathsf{Paths}^{\leq n}(s) \mid s_m \in \mathcal{T} \text{ and for all } i \in \{0, \ldots, m-1\}\colon s_i \notin \mathcal{T}\right\}.\end{aligned}$$

5. We define the *set of all paths starting in state $s \in \mathcal{S}$ and eventually reaching a state in $\mathcal{T}$* as
$$\mathsf{Paths}(s, \mathcal{T}) = \bigcup_{n \in \mathbb{N}} \mathsf{Paths}^{\leq n}(s, \mathcal{T}).$$

All of the above sets are countable. Furthermore, every path $\pi \in \mathsf{Paths}(s, \mathcal{T})$ contains *exactly one* state from $\mathcal{T}$, namely the last state of π.

Next, we consider *schedulers* for resolving the nondeterminism in an MDP.

Definition 2.10 (Schedulers).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP. A *scheduler* for $\mathcal{M}$ is a function
$$\mathfrak{S}\colon \mathcal{S}^+ \to \mathsf{Act}$$
satisfying
$$\text{for all } s_0 \ldots s_n \in \mathcal{S}^+\colon \quad \mathfrak{S}(s_0 \ldots s_n) \in \mathsf{Act}(s_n).$$
We denote the *set of all schedulers for $\mathcal{M}$* by Scheds. Moreover, a scheduler $\mathfrak{S}$ is called *memoryless*, if
$$\begin{aligned}&\text{for all } s_0 \ldots s_n \in \mathcal{S}^+\colon \text{for all } t_0 \ldots t_m \in \mathcal{S}^+\colon\\ &\qquad s_n = t_m \quad \text{implies} \quad \mathfrak{S}(s_0 \ldots s_n) = \mathfrak{S}(t_0 \ldots t_m).\end{aligned}$$
Otherwise, $\mathfrak{S}$ is called *history-dependent*. We often identify memoryless schedulers $\mathfrak{S}$ with functions of type $\mathcal{S} \to \mathsf{Act}$. We denote the *set of all memoryless schedulers for $\mathcal{M}$* by $\mathsf{MLScheds}$.

A scheduler $\mathfrak{S}$ resolves the nondeterminism in an MDP $\mathcal{M}$ as follows: Starting in some state s_0, the first action that is to be chosen is $\mathfrak{S}(s_0)$. If the probabilistically chosen $\mathfrak{S}(s_0)$-successor of s_0 is s_1, then the next action that is to be chosen is $\mathfrak{S}(s_0 s_1)$, and so on. For the special case of memoryless schedulers, the action that is to be chosen only depends on the current state.

Since a scheduler $\mathfrak{S}$ resolves the nondeterminism in $\mathcal{M}$, it makes sense to speak about the *probability of a path in $\mathcal{M}$ under $\mathfrak{S}$*.

Definition 2.11 (Induced Path Probabilities).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP, let $\mathfrak{S} \in \mathsf{Scheds}$, and let $s_0 \ldots s_n \in \mathcal{S}^+$. The *probability of $s_0 \ldots s_n$ in $\mathcal{M}$ under $\mathfrak{S}$* is defined as

$$\mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_n) \;=\; \prod_{i=0}^{n-1} P(s_i, \mathfrak{S}(s_0 \ldots s_i), s_{i+1})\,,$$

where the empty product equals 1.

Example 2.14.
Reconsider the MDP $\mathcal{M}1$ from Figure 2.7 and let $\mathfrak{S}$ be the memoryless scheduler defined as

$$\mathfrak{S}(s) \;=\; \begin{cases} \mathfrak{b} & \text{if } s = s_0 \\ \mathfrak{a} & \text{otherwise}\,. \end{cases}$$

We have $\mathsf{Prob}_{\mathfrak{S}}(s_0 s_0) = 0$ and $\mathsf{Prob}_{\mathfrak{S}}(s_0 s_2 s_4) = 2/3$.

2.2.2 Expected Reachability-Rewards

We consider expected reachability-rewards in MDPs — a generalization of reachability probabilities (cf. [BK08, Chapter 10]). We start with a formalization of summation over countable index sets of non-negative extended reals.

Throughout this thesis, we let $0 \cdot \infty = \infty \cdot 0 = 0$ and $\alpha + \infty = \infty + \alpha$ for all $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$. Given a countable set A and $g\colon A \to \mathbb{R}_{\geq 0}^{\infty}$, we define[3]

$$\sum_{a \in A} g(a) \;=\; \sup \left\{ \sum_{a \in F} g(a) \;\middle|\; F \text{ finite and } F \subseteq A \right\},$$

which is a well-defined quantity in $\mathbb{R}_{\geq 0}^{\infty}$. An important characteristic of such a sum is that even for countably *infinite* A, the order of summation is irrelevant: For every bijective function $\mathsf{enum}\colon \mathbb{N} \to A$, we have [Wil07, Proposition 6.1]

$$\sum_{a \in A} g(a) \;=\; \sum_{i=0}^{\infty} g(\mathsf{enum}(i)) \;=\; \sup_{n \in \mathbb{N}} \sum_{i=0}^{n} g(\mathsf{enum}(i))\,.$$

With this notion at hand, we define reachability-reward functions and corresponding expected reachability-rewards in MDPs.

[3] following [Wil07, Section 6.2]

Definition 2.12 (Reachability-Reward Functions).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathcal{T} \subseteq \mathcal{S}$ be a set of states. A function

$$\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$$

is called *(reachability-)reward function (for $\mathcal{M}$)*.

Intuitively, $\mathcal{T}$ is a set of *target states* and $\mathsf{rew}(t)$ is the *reward* collected upon reaching target state $t \in \mathcal{T}$. We call $\mathsf{rew}(t)$ the *reward of state t*.

Definition 2.13 (Expected Reachability-Rewards).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP, let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$ be a reward function, and let $\mathfrak{S} \in \mathsf{Scheds}$. We define the following quantities in $\mathbb{R}_{\geq 0}^{\infty}$:

1. The *expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ in at most $n \in \mathbb{N}$ steps under $\mathfrak{S}$* is defined as
$$\mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right) \;=\; \sum_{s_0 \ldots s_m \in \mathsf{Paths}^{\leq n}(s, \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m) \cdot \mathsf{rew}(s_m)\,.$$
Notice that the number of summands is finite.

2. The *expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ under $\mathfrak{S}$ in an arbitrary number of steps* is defined as
$$\mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond\mathsf{rew}\right) \;=\; \sum_{s_0 \ldots s_m \in \mathsf{Paths}(s, \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m) \cdot \mathsf{rew}(s_m)\,.$$
Notice that the number of summands can be infinite.

3. The *minimal expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ in at most $n \in \mathbb{N}$ steps* is defined as
$$\mathsf{MinER}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right) \;=\; \inf_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right)\,.$$

4. The *minimal expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ in an arbitrary number of steps* is defined as
$$\mathsf{MinER}\left(\mathcal{M}, s \models \Diamond\mathsf{rew}\right) \;=\; \inf_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond\mathsf{rew}\right)\,.$$

5. The *maximal expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ in at most $n \in \mathbb{N}$ steps* is defined as
$$\mathsf{MaxER}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right) \;=\; \sup_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right)\,.$$

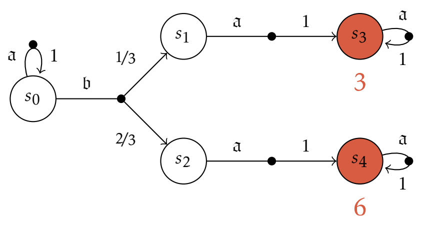


Figure 2.8: MDP $\mathcal{M}1$ from Figure 2.7 annotated with a reward function $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$. States in $\mathcal{T}$ and their rewards are colored in red.

6. The *maximal expected (reachability-)reward to eventually reach a state in $\mathcal{T}$ from state $s \in \mathcal{S}$ in an arbitrary number of steps* is defined as

$$\mathsf{MaxER}(\mathcal{M}, s \models \lozenge\mathsf{rew}) = \sup_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}(\mathcal{M}, s \models \lozenge\mathsf{rew}) .$$

If clear from the context, we often omit $\mathcal{M}$ in the above-defined notions. If $\mathcal{M}$ is a Markov chain, then $\mathsf{MinER}(s \models \lozenge\mathsf{rew})$ and $\mathsf{MaxER}(s \models \lozenge\mathsf{rew})$ coincide, in which case we often write $\mathsf{ER}(s \models \lozenge\mathsf{rew})$.

Let us gain some intuition on the above notions. $\mathsf{ER}_{\mathfrak{S}}(s \models \lozenge\mathsf{rew})$ is the expected (or *average*) reward collected if the agent modeled by $\mathcal{M}$ behaves as follows: The agent starts in state s and chooses actions according to $\mathfrak{S}$. Upon reaching a target state $s' \in \mathcal{T}$, the agent collects a reward of $\mathsf{rew}(s')$, and then stops, i.e., collects no further reward. Similarly, $\mathsf{ER}_{\mathfrak{S}}\left(s \models \lozenge^{\leq n}\mathsf{rew}\right)$ is the expected reward collected if the agent is restricted to at most n steps. $\mathsf{MinER}(s \models \lozenge\mathsf{rew})$ is the minimal — under all possible resolutions of the nondeterminism, i.e., all choices the agent can make — expected reward when starting in state s. Analogously, $\mathsf{MaxER}(s \models \lozenge\mathsf{rew})$ is the maximal[4] expected reward. Finally, $\mathsf{MinER}\left(s \models \lozenge^{\leq n}\mathsf{rew}\right)$ and $\mathsf{MaxER}\left(s \models \lozenge^{\leq n}\mathsf{rew}\right)$ are the corresponding step-bounded variants.

Example 2.15.
Figure 2.8 depicts the MDP $\mathcal{M}1$ annotated with the reward function

[4]This value is not necessarily attained, i.e., the supremum from Definition 2.13.6 is not necessarily a maximum (cf. Section 2.2.5). We will nonetheless speak of maximal expected rewards for the sake of convenience.

$\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$ with $\mathcal{T} = \{s_3, s_4\}$ and

$$\mathsf{rew}(s_3) = 3 \qquad \text{and} \qquad \mathsf{rew}(s_4) = 6\,.$$

We have

$$\mathsf{MinER}\,(\mathcal{M}, s_0 \models \Diamond \mathsf{rew}) = 0$$

since for every scheduler $\mathfrak{S}$ with $\mathfrak{S}(s_0) = \mathfrak{a}$, every state in $\mathcal{T}$ is unreachable, i.e., there is no path $\pi \in \mathsf{Paths}(s_0, \mathcal{T})$ with $\mathsf{Prob}_{\mathfrak{S}}(\pi) > 0$. Moreover, we have

$$\mathsf{MaxER}\,(\mathcal{M}, s_0 \models \Diamond \mathsf{rew}) = 1/3 \cdot 3 + 2/3 \cdot 6 = 5\,,$$

which is realized by every scheduler $\mathfrak{S}$ with $\mathfrak{S}(s_0) = \mathfrak{b}$.

2.2.3 Properties of Expected Reachability-Rewards

Expected reachability-rewards generalize the notion of reachability probabilities, which are defined as follows (cf. [BK08, Chapter 10]): For scheduler $\mathfrak{S}$, the *probability to reach a state in $\mathcal{T}$ under $\mathfrak{S}$ from state s* is defined as

$$\mathsf{Pr}_{\mathfrak{S}}\,(\mathcal{M}, s \models \Diamond \mathcal{T}) = \sum_{s_0 \ldots s_m \in \mathsf{Paths}(s, \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m)\,.$$

The *minimal probability to reach a state in $\mathcal{T}$ from state s* is defined as

$$\mathsf{MinPr}\,(\mathcal{M}, s \models \Diamond \mathcal{T}) = \inf_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{Pr}_{\mathfrak{S}}\,(\mathcal{M}, s \models \Diamond \mathcal{T})\,.$$

Analogously, the *maximal probability to reach a state in $\mathcal{T}$ from state s* is defined as

$$\mathsf{MaxPr}\,(\mathcal{M}, s \models \Diamond \mathcal{T}) = \sup_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{Pr}_{\mathfrak{S}}\,(\mathcal{M}, s \models \Diamond \mathcal{T})\,.$$

If clear from the context, we often omit $\mathcal{M}$. If $\mathcal{M}$ is a Markov chain, then $\mathsf{MinPr}\,(s \models \Diamond \mathcal{T})$ and $\mathsf{MaxPr}\,(s \models \Diamond \mathcal{T})$ coincide, which justifies writing $\mathsf{Pr}\,(s \models \Diamond \mathcal{T})$.

Corollary 2.5 (Reachability Probabilities via Expected Rewards).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathcal{T} \subseteq \mathcal{S}$ be a set of target states. Moreover, let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$ be the reward function defined as $\mathsf{rew}(t) = 1$ for all $t \in \mathcal{T}$. We have for every $s \in \mathcal{S}$:

1. $$\mathsf{MinER}\,(s \models \Diamond \mathsf{rew}) = \mathsf{MinPr}\,(s \models \Diamond \mathcal{T})$$
2. $$\mathsf{MaxER}\,(s \models \Diamond \mathsf{rew}) = \mathsf{MaxPr}\,(s \models \Diamond \mathcal{T})$$

We say that a state $s \in \mathcal{S}$ is a *sink* if all outgoing transitions of s lead to s, i.e., if for all $\mathfrak{a} \in \mathsf{Act}(s)$ we have $\mathsf{Succs}_{\mathfrak{a}} = \{s\}$. If $\mathcal{T}$ contains only sink states, then we can express expected rewards in terms of reachability probabilities:

Theorem 2.6 (Expected Rewards via Reachability Probabilities).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$ be a reward function. If every $t \in \mathcal{T}$ is a sink, then:

1. $$\mathsf{MinER}(s \models \Diamond \mathsf{rew}) \;=\; \inf_{\mathfrak{S} \in \mathsf{Scheds}} \sum_{t \in \mathcal{T}} \mathsf{Pr}_{\mathfrak{S}}(s \models \Diamond\{t\}) \cdot \mathsf{rew}(t)$$

2. $$\mathsf{MaxER}(s \models \Diamond \mathsf{rew}) \;=\; \sup_{\mathfrak{S} \in \mathsf{Scheds}} \sum_{t \in \mathcal{T}} \mathsf{Pr}_{\mathfrak{S}}(s \models \Diamond\{t\}) \cdot \mathsf{rew}(t)$$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. $$\mathsf{ER}(s \models \Diamond \mathsf{rew}) \;=\; \sum_{t \in \mathcal{T}} \mathsf{Pr}(s \models \Diamond\{t\}) \cdot \mathsf{rew}(t)\,.$$

Proof. This is an immediate consequence of the fact that

$$\mathsf{Paths}(s, \mathcal{T}) \;=\; \dot{\bigcup_{t \in \mathcal{T}}} \mathsf{Paths}(s, \{t\})\,,$$

i.e., the sets $\mathsf{Paths}(s, \{t\})$ for $t \in \mathcal{T}$ partition the set $\mathsf{Paths}(s, \mathcal{T})$. ∎

2.2.4 Expected Reachability-Rewards via Least Fixpoints

Our goal is to obtain least fixpoint characterizations of minimal and maximal expected reachability-rewards and to prove these characterizations correct. For that, we employ the framework of fixpoint theory presented in Section 2.1.

Said least fixpoint constructions are obtained from a variant of Bellman's optimality equations [Bel57]. Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$ a reward function. We have for every $s \in \mathcal{S}$

$$\begin{aligned} &\mathsf{MinER}(s \models \Diamond \mathsf{rew}) \\ &= \begin{cases} \mathsf{rew}(s) & \text{if } s \in \mathcal{T} \\ \min\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot \mathsf{MinER}(s' \models \Diamond \mathsf{rew}) & \text{otherwise} \end{cases} \end{aligned}$$

and

$$\begin{aligned}&\mathsf{MaxER}\,(s \models \Diamond \mathsf{rew})\\ =& \begin{cases}\mathsf{rew}(s) & \text{if } s \in \mathcal{T}\\ \max\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot \mathsf{MaxER}\,(s' \models \Diamond \mathsf{rew}) & \text{otherwise}\,.\end{cases}\end{aligned}$$

In words, if $s \in \mathcal{T}$, then the minimal (resp. maximal) expected reward of state s is $\mathsf{rew}(s)$. Otherwise, the minimal (resp. maximal) expected reward of s is the minimum (resp. maximum) of the sums over the minimal (resp. maximal) expected reward of $\mathfrak{a}$-successors s' of s, where each summand is weighted according to the probability of moving from s to s' when taking action $\mathfrak{a}$.

In fixpoint-theoretic terms, we define a function $X\colon \mathcal{S} \to \mathbb{R}_{\geq 0}^{\infty}$, which maps each state s to its (minimal or maximal) expected reward as the least — in a sense we make precise below — solution of the recursive equation

$$X \;=\; \lambda s. \begin{cases}\mathsf{rew}(s) & \text{if } s \in \mathcal{T}\\ \mathop{\mathsf{op}}\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot X(s') & \text{otherwise}\,,\end{cases} \tag{2.2}$$

where the choice of $\mathsf{op} \in \{\min, \max\}$ depends on whether we are considering minimal or maximal expected rewards. In what follows, we prove that the least solution of Equation (2.2) exists uniquely (Theorem 2.7) and indeed yields the sought-after expected rewards (Theorem 2.9). The solution domain of the above equation is the complete lattice of *value functions*:

Definition 2.14 (Value functions).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP. The complete lattice of *value functions (for $\mathcal{M}$)* is defined as

$$(\mathbb{V}, \sqsubseteq)\,,$$

where:

1. $\mathbb{V} = \mathcal{S} \to \mathbb{R}_{\geq 0}^{\infty}$ is the *set of value functions*, and
2. $\sqsubseteq$ is the pointwise lifted order on $\mathbb{R}_{\geq 0}^{\infty}$, i.e.,

$$\text{for all } v, v' \in \mathbb{V}\colon \quad v \sqsubseteq v' \text{ iff for all } s \in \mathcal{S}\colon v(s) \leq v'(s)\,.$$

The least element of $(\mathbb{V}, \sqsubseteq)$ is the constant-zero-function $\lambda s.0$, which we —

slightly abusing notation — denote by 0. Moreover, suprema and infima in $(\mathbb{V}, \sqsubseteq)$ are given as the pointwise liftings of suprema and infima in $(\mathbb{R}^{\infty}_{\geq 0}, \leq)$, i.e.,

$$\text{for all } V \subseteq \mathbb{V}\colon \quad \bigsqcup V = \lambda s.\, \sup\{v(s) \mid v \in V\} \text{ and } \bigsqcap V = \lambda s.\, \inf\{v(s) \mid v \in V\}\ .$$

(Least) solutions of Equation (2.2) are formalized as (least) fixpoints of functions of type $\mathbb{V} \to \mathbb{V}$, which are called *Bellman operators*.

Definition 2.15 (Bellman operators).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$.

1. The function ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\colon \mathbb{V} \to \mathbb{V}$ defined as

$$ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}(v) \;=\; \lambda s. \begin{cases} \mathsf{rew}(s) & \text{if } s \in \mathcal{T} \\ \min\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot v(s') & \text{otherwise}\ . \end{cases} $$

   is called the *min-Bellman operator of $\mathcal{M}$ w.r.t.* rew.

2. Analogously, the function ${}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}\colon \mathbb{V} \to \mathbb{V}$ defined as

$$ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}(v) \;=\; \lambda s. \begin{cases} \mathsf{rew}(s) & \text{if } s \in \mathcal{T} \\ \max\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot v(s') & \text{otherwise}\ . \end{cases} $$

   is called the *max-Bellman operator of $\mathcal{M}$ w.r.t.* rew.

If $\mathcal{M}$ is a Markov chain, then ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ and ${}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ coincide, in which case we often write ${}_{\mathcal{M}}\Phi_{\mathsf{rew}}$. Least fixpoints of the Bellman operators exist uniquely:

Theorem 2.7 (Continuity of Bellman Operators).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$.

1. Both the min-Bellman operator ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ and the max-Bellman operator ${}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ of $\mathcal{M}$ w.r.t. rew are continuous w.r.t. $(\mathbb{V}, \sqsubseteq)$.

2. Both ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ and ${}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ have a least fixpoint given by

$$ \begin{aligned} & \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; {}^{\min}_{\mathcal{M}}\Phi^{\omega}_{\mathsf{rew}}(0) \\ \text{and} \qquad & \mathsf{lfp}\ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; {}^{\max}_{\mathcal{M}}\Phi^{\omega}_{\mathsf{rew}}(0)\ . \end{aligned} $$

Proof. Theorem 2.7.1 is a consequence of the fact that addition, multiplication, minima, and maxima are continuous w.r.t. $(\mathbb{R}^{\infty}_{\geq 0}, \leq)$ [DKV09, Chapter 1]. Theorem 2.7.2 is then an instance of Kleene's Theorem 2.2. ∎

Our goal is now to prove that the least fixpoints of the Bellman operators indeed evaluate to the sought-after expected rewards, i.e.,

$$\begin{aligned} & \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\, \mathsf{MinER}(s \models \Diamond \mathsf{rew}) \\ \text{and} \qquad & \mathsf{lfp}\ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\, \mathsf{MaxER}(s \models \Diamond \mathsf{rew})\ . \end{aligned}$$

Analogously to Remark 2.1, we first establish a link between fixpoint iterates and step-bounded expected rewards.

Lemma 2.8 (Step-Bounded Expected Rewards via Fixpoint Iteration). Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$. We have:

1. for all $n \in \mathbb{N}$: $\quad {}^{\min}_{\mathcal{M}}\Phi^{n+1}_{\mathsf{rew}}(0) \;=\; \lambda s.\, \mathsf{MinER}\big(s \models \Diamond^{\leq n} \mathsf{rew}\big)$
2. for all $n \in \mathbb{N}$: $\quad {}^{\max}_{\mathcal{M}}\Phi^{n+1}_{\mathsf{rew}}(0) \;=\; \lambda s.\, \mathsf{MaxER}\big(s \models \Diamond^{\leq n} \mathsf{rew}\big)$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. for all $n \in \mathbb{N}$: $\quad {}_{\mathcal{M}}\Phi^{n+1}_{\mathsf{rew}}(0) \;=\; \lambda s.\, \mathsf{ER}\big(s \models \Diamond^{\leq n} \mathsf{rew}\big)\ .$

Proof. By induction on n. See Appendix 1.1.1 for details. ■

Using some additional auxiliary results given in Appendix 1.1, this enables us to prove our least fixpoint construction correct:

Theorem 2.9 (Expected Rewards via Least Fixpoints). Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$. We have:

1. $\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\, \mathsf{MinER}(\mathcal{M}, s \models \Diamond \mathsf{rew})$
2. $\mathsf{lfp}\ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\, \mathsf{MaxER}(\mathcal{M}, s \models \Diamond \mathsf{rew})$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. $\mathsf{lfp}\ {}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\, \mathsf{ER}(\mathcal{M}, s \models \Diamond \mathsf{rew})\ .$

Proof. For maximal expected rewards, the claim follows from Theorem 2.2 and Lemma 2.8.2. The proof for minimal expected rewards is more involved and relies on the fact $\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ gives rise to an optimal memoryless scheduler (cf. Lemma A.1). The full proof is given in Appendix 1.1.2. ■

Example 2.16.
Reconsider the MDP $\mathcal{M}1$ and the reward function rew from Figure 2.8. By Lemma 2.8, performing a fixpoint iteration on ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ (resp. ${}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}$) yields increasingly precise lower bounds on the minimal (resp. maximal) expected reward of each state, which — in the limit — yields the precise minimal (resp. maximal) expected rewards. In the literature, this process is often called *value iteration* [BK08, Chapter 10]. We exemplify this for minimal expected rewards. For that, we denote value functions by column vectors, i.e., $v \in \mathbb{V}$ is denoted by

$$\begin{pmatrix} v(s_0) \\ v(s_1) \\ v(s_2) \\ v(s_3) \\ v(s_4) \end{pmatrix} .$$

Let us now perform a fixpoint iteration on ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$:

$${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 3 \\ 6 \end{pmatrix}$$

$${}^{\min}_{\mathcal{M}}\Phi^2_{\mathsf{rew}}\left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 3 \\ 6 \\ 3 \\ 6 \end{pmatrix}$$

$${}^{\min}_{\mathcal{M}}\Phi^3_{\mathsf{rew}}\left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ 3 \\ 6 \\ 3 \\ 6 \end{pmatrix}$$

Since ${}^{\min}_{\mathcal{M}}\Phi^2_{\mathsf{rew}}(0)$ and ${}^{\min}_{\mathcal{M}}\Phi^3_{\mathsf{rew}}(0)$ coincide, we obtain $\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ after finitely many fixpoint iterations. We remark that, even for finite-state MDPs, this is not guaranteed in general.

2.2.5 Existence of Optimal Schedulers

We briefly comment on the existence of *optimal* schedulers for expected reachability-rewards. Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let rew be a reward function. We say that a scheduler $\mathfrak{S}$ is *min-optimal for $\mathcal{M}$ w.r.t.* rew *at state s*, if

$$\mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;=\; \mathsf{MinER}(\mathcal{M}, s \models \Diamond \mathsf{rew}) \;.$$

Moreover, we say that $\mathfrak{S}$ is *uniformly min-optimal for $\mathcal{M}$ w.r.t.* rew, if $\mathfrak{S}$ is min-optimal at all states $s \in \mathcal{S}$, i.e., if

$$\text{for all } s \in \mathcal{S}\colon \quad \mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;=\; \mathsf{MinER}(\mathcal{M}, s \models \Diamond \mathsf{rew}) \;.$$

For maximal expected rewards, both notions are defined analogously.

We obtain as a by-catch of Lemma A.1 and Theorem 2.9.1 that there is always a uniformly *min*-optimal scheduler $\mathfrak{S}$, which is moreover memoryless.[5]

Theorem 2.10 (Optimal Schedulers for Minimal Expected Rewards). Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let rew be a reward function. There is a memoryless scheduler $\mathfrak{S} \in \mathsf{MLScheds}$ with

$$\text{for all } s \in \mathcal{S}\colon \quad \mathsf{MinER}(s \models \Diamond \mathsf{rew}) \;=\; \mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;.$$

In particular, the infimum from Definition 2.13.4 is always attained by a *memoryless* scheduler, i.e., we have

$$\mathsf{MinER}(s \models \Diamond \mathsf{rew}) \;=\; \min_{\mathfrak{S} \in \mathsf{MLScheds}} \mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;. \tag{2.3}$$

Regarding *maximal* expected rewards, it is known that (both non-uniform and uniform) max-optimal schedulers do *not* necessarily exist (see, e.g., [Orn69]). In particular, the supremum from Definition 2.13.6 is *not necessarily attained*. Analogously to Equation (2.3), one could, however, ask whether the supremum from Definition 2.13.6 over *all* schedulers can be replaced by a supremum over *memoryless* schedulers, i.e., whether for all states s,

$$\mathsf{MaxER}(s \models \Diamond \mathsf{rew}) \;\overset{?}{=}\; \sup_{\mathfrak{S} \in \mathsf{MLScheds}} \mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;,$$

which is the case if for all states s and all $\epsilon > 0$ there is an ϵ-max-optimal memoryless scheduler $\mathfrak{S}$, i.e., a memoryless scheduler $\mathfrak{S}$ with

$$\mathsf{MaxER}(s \models \Diamond \mathsf{rew}) - \epsilon \;\leq\; \mathsf{ER}_{\mathfrak{S}}(s \models \Diamond \mathsf{rew}) \;.$$

[5] A similar result is known for *expected total rewards* in case the reward function is pointwise smaller than ∞ [Put94, Theorem 7.3.6a].

The answer to this question is, to the best of our knowledge, open. A more in-depth treatment of the existence of schedulers is outside the scope of this thesis.

2.3 Probabilistic Programs

In this thesis, we study the deductive verification of *discrete probabilistic programs featuring nondeterministic choices*. The probabilistic programming language considered throughout is McIver & Morgan's *probabilistic guarded command language* (pGCL) [MM99; MM05] — an extension of Dijkstra's guarded command language [Dij76]. The syntax of pGCL is provided in Section 2.3.1. In Section 2.3.2, we provide a formal operational semantics based on Markov decision processes.

2.3.1 The Probabilistic Programming Language pGCL

Let $\mathsf{Vars} = \{x, y, z, \ldots\}$ be a countably infinite *set of program variables* and let Vals be a countable *set of values*. Unless explicitly stated otherwise, we assume that $\mathsf{Vals} = \mathbb{Q}_{\geq 0}$. A *(program) state* is a function of type $\mathsf{Vars} \to \mathsf{Vals}$. Program states are denoted by σ, τ, and variations thereof. We restrict to *non-negative* rationals for the sake of convenience, which is discussed in Section 2.4.2. To ensure that the *set of program states* is countable, we define it as

$$\mathsf{States} \;=\; \{\sigma\colon \mathsf{Vars} \to \mathbb{Q}_{\geq 0} \mid \text{the set } \{x \in \mathsf{Vars} \mid \sigma(x) > 0\} \text{ is finite}\}$$

and tacitly assume that all program states are taken from States. A *predicate (over* States*)* is a set $P \subseteq \mathsf{States}$ and the *set of predicates* is $\mathcal{P}(\mathsf{States})$. We usually write $\sigma \models P$ instead of $\sigma \in P$ and say that σ *satisfies* P. Moreover, we employ the usual logical connectives between predicates. For instance, we write

$$P \wedge Q \qquad \text{instead of} \qquad P \cap Q\,.$$

With these notions at hand, we define pGCL.

Definition 2.16 (The Probabilistic Guarded Command Language). Programs C in the *probabilistic guarded command language*, which we denote by pGCL, adhere to the grammar

$$\begin{array}{lcll}
C & \longrightarrow & \mathtt{skip} & \text{(effectless program)} \\
 & | & x := A & \text{(assignment)} \\
 & | & C\,;C & \text{(sequential composition)} \\
 & | & \{C\}\,[p]\,\{C\} & \text{(probabilistic choice)}
\end{array}$$

$$
\begin{aligned}
&\mid \{C\} \mathbin{\square} \{C\} && \text{(nondeterministic choice)}\\
&\mid \texttt{if}\,(B)\,\{C\}\,\texttt{else}\,\{C\} && \text{(conditional choice)}\\
&\mid \texttt{while}(B)\{C\}\,, && \text{(while loop)}
\end{aligned}
$$

where:

1. A: States $\rightarrow$ Vals is an *arithmetic expression*,
2. $p \in [0,1] \cap \mathbb{Q}$ is a *rational probability*, and
3. $B \in \mathcal{P}$(States) is a predicate also referred to as a *guard*.

If a program C does not contain nondeterministic choice, then we call C *fully probabilistic*. Otherwise, we call C *nondeterministic*. If C does not contain probabilistic choices, then we call C *non-probabilistic*. If C does not contain while loops, then we call C *loop-free*. Otherwise, we call C *loopy*. We do not (yet) give a concrete syntax for arithmetic expressions or predicates since such a syntax is not relevant at this point. In order to ensure that the set pGCL of programs is countable[6], we assume that both arithmetic expressions and predicates are taken from some computable set.

Let us gain some intuition on what it means to execute each of the individual constructs on some initial program state σ. A formal treatment of pGCL's operational semantics is given in Section 2.3.2.

Effectless Program. `skip` immediately terminates in σ.

Assignment. $x := A$ evaluates A in the current state σ and assigns the result to x. Hence, the assignment terminates in the state $\sigma[x \mapsto A(\sigma)]$ obtained from updating the value of x to $A(\sigma)$. More generally, we define for every $q \in \mathbb{Q}_{\geq 0}$

$$\sigma[x \mapsto q] \;=\; \lambda y.\begin{cases} q & \text{if } y = x \\ \sigma(y) & \text{otherwise}\,. \end{cases}$$

Sequential Composition. $C_1\,; C_2$ first executes C_1. If C_1 terminates in some state σ', then C_2 is subsequently executed on this new state σ'.

Probabilistic Choice. Executing $\{C_1\}\;[p]\;\{C_2\}$ corresponds to a random experiment, where we flip a coin with bias p and base our decision of which

[6]This will ensure that pGCL's operational MDP is countable, see Definition 2.17 on page 50.

branch is to be executed on the outcome of this coin flip: The branch C_1 is executed with probability p and the branch C_2 is executed with probability $1-p$.

Nondeterministic Choice. Rather than having information on the likelihood of C_1 or C_2 being executed, all we know is that for both C_1 and C_2 there is the *possibility* of being executed. In this sense, the nondeterministic choice is actually not executable unless we explicitly *resolve* the nondeterminism by deciding which branch to take. As with resolving nondeterminism in MDPs, this decision may be based on the current state σ or on the complete computation history. See Section 2.3.2 for details.

Conditional Choice. $\texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_1\}$ executes C_1 if the current state σ satisfies the guard B. Otherwise, C_2 is executed.

While Loop. $\texttt{while}\,(B)\{C\}$ checks whether the current state σ satisfies the guard B. If not, the loop terminates. Otherwise, the loop executes C on σ — the *loop body*. If the loop body C terminates in a new state σ', then $\texttt{while}\,(B)\{C\}$ repeats this procedure i.e., checks whether σ' satisfies B, and so on. Notice that only loops can cause nonterminating (or *diverging*) behavior.

Example 2.17.
Consider the fully probabilistic loop C given by

$$\texttt{while}\,(y=1)\{\{y := 0\}\,[1/2]\,\{x := x+1\}\}\,.$$

C can be understood as a sampler for the geometric distribution with parameter $1/2$: Given an initial state σ with $\sigma \models y = 1$ and $i \in \mathbb{N}$, the probability that C terminates in a final state τ with $\tau(x) = \sigma(x) + i$ is $1/2^{i+1}$.

2.3.2 Operational MDP Semantics of pGCL

Towards defining an operational MDP semantics of pGCL, we define a small-step execution relation à la Plotkin [Plo04]. We mainly follow the presentation from [5] with adaptions from [19]. We define the *set of configurations* as

$$\mathsf{Conf} \;=\; (\mathsf{pGCL} \cup \{\Downarrow\}) \times \mathsf{States}\,.$$

Configurations are denoted by $\mathfrak{c}$ and variations thereof. A configuration of the form $(\Downarrow, \tau)$ is called *final* and indicates termination in the final state τ. A

1. Final configurations:

$$\frac{}{(\Downarrow, \sigma) \xrightarrow{N,1} (\Downarrow, \sigma)}$$

2. Effectless Program:

$$\frac{}{(\mathtt{skip}, \sigma) \xrightarrow{N,1} (\Downarrow, \sigma)}$$

3. Assignment:

$$\frac{}{(x := A, \sigma) \xrightarrow{N,1} (\Downarrow, \sigma[x \mapsto A(\sigma)])}$$

4. Sequential Composition:

$$\frac{(C_1, \sigma) \xrightarrow{\mathfrak{a},p} (\Downarrow, \sigma')}{(C_1 ; C_2, \sigma) \xrightarrow{\mathfrak{a},p} (C_2, \sigma')} \qquad \frac{(C_1, \sigma) \xrightarrow{\mathfrak{a},p} (C_1', \sigma')}{(C_1 ; C_2, \sigma) \xrightarrow{\mathfrak{a},p} (C_1' ; C_2, \sigma')}$$

Figure 2.9: Rules defining the small-step execution relation $\rightarrow$ (first part).

5. Probabilistic Choice:

$$\frac{C_1 \neq C_2}{(\{C_1\}\,[p]\,\{C_2\},\sigma) \xrightarrow{N,p} (C_1,\sigma)} \qquad \frac{C_1 \neq C_2}{(\{C_1\}\,[p]\,\{C_2\},\sigma) \xrightarrow{N,1-p} (C_2,\sigma)}$$

$$\frac{}{(\{C\}\,[p]\,\{C\},\sigma) \xrightarrow{N,1} (C,\sigma)}$$

6. Nondeterministic Choice:

$$\frac{}{(\{C_1\}\,\square\,\{C_2\},\sigma) \xrightarrow{L,1} (C_1,\sigma)} \qquad \frac{}{(\{C_1\}\,\square\,\{C_2\},\sigma) \xrightarrow{R,1} (C_2,\sigma)}$$

7. Conditional Choice:

$$\frac{\sigma \models B}{(\texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_2\},\sigma) \xrightarrow{N,1} (C_1,\sigma)}$$

$$\frac{\sigma \models \neg B}{(\texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_2\},\sigma) \xrightarrow{N,1} (C_2,\sigma)}$$

8. While Loop:

$$\frac{\sigma \models B}{(\texttt{while}(B)\{C\},\sigma) \xrightarrow{N,1} (C\,;\texttt{while}(B)\{C\},\sigma)}$$

$$\frac{\sigma \models \neg B}{(\texttt{while}(B)\{C\},\sigma) \xrightarrow{N,1} (\Downarrow,\sigma)}$$

Figure 2.10: Rules defining the small-step execution relation $\rightarrow$ (second part).

configuration of the form (C, σ) indicates that program C is to be executed on initial state σ. We then define the *small-step execution relation*

$$\to \; \subseteq \; \mathsf{Conf} \times \{N, L, R\} \times ([0,1] \cap \mathbb{Q}) \times \mathsf{Conf}$$

as the smallest relation satisfying the rules given in Figures 2.9 and 2.10, where we write $\mathfrak{c} \xrightarrow{\mathfrak{a}, p} \mathfrak{c}'$ instead of $(\mathfrak{c}, \mathfrak{a}, p, \mathfrak{c}') \in \to$. This yields pGCL's *operational MDP*:

Definition 2.17 (Operational MDP).
The *operational MDP of* pGCL is

$$\mathcal{O} \;=\; (\mathsf{Conf}, \mathsf{Act}, P)\,,$$

where:

1. $\mathsf{Act} = \{N, L, R\}$ is the set of actions,
2. $P\colon \mathsf{Conf} \times \mathsf{Act} \times \mathsf{Conf} \to [0,1]$ is the transition probability function with

$$P(\mathfrak{c}, \mathfrak{a}, \mathfrak{c}') \;=\; \begin{cases} p & \text{if } \mathfrak{c} \xrightarrow{\mathfrak{a}, p} \mathfrak{c}' \\ 0 & \text{otherwise}\,. \end{cases}$$

The transition probability function P of the operational MDP $\mathcal{O}$ formalizes the informal explanation of pGCL's execution behavior from Section 2.3.1: If

$$P((C, \sigma), \mathfrak{a}, \mathfrak{c}') \;=\; p$$

then executing C for *one* step on initial state σ when taking action $\mathfrak{a}$ yields the new configuration $\mathfrak{c}'$ with probability p. We have $p \in (0,1)$ only if executing C for one step involves a probabilistic choice. For all other statements, we have $p \in \{0,1\}$. Moreover, we have $|\mathsf{Act}((C,\sigma))| = \{L, R\}$ iff executing C for one step involves a nondeterministic choice, where L is the action for the left branch and R is the action for the right branch. If executing C for one step involves one of the remaining statements, then $\mathsf{Act}((C,\sigma)) = \{N\}$, indicating that no nondeterministic choice is possible.

A scheduler $\mathfrak{S}$ for $\mathcal{O}$ resolves the nondeterminism encountered when executing a given program C on some initial state σ. If the nondeterminism is resolved by a scheduler, then it makes sense to speak about the subdistribution of final states obtained from executing C on initial state σ. We adopt the following definition from [Kam19, Proposition 3.13].

Definition 2.18 (Subdistribution of Final States).
Let $C \in \mathsf{pGCL}$, let $\sigma \in \mathsf{States}$, and let $\mathfrak{S}$ be a scheduler for $\mathcal{O}$. The *subdistribu-*

tion of final states obtained from executing C on σ under $\mathfrak{S}$ is defined as

$$[\![C]\!]^{\sigma}_{\mathfrak{S}} \colon \mathsf{States} \to [0,1]\,, \qquad [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau) \;=\; \Pr_{\mathfrak{S}}\left((C,\sigma) \models \Diamond(\Downarrow,\tau)\right)\,.$$

$[\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)$ is the probability that C terminates in state τ when executed on initial state σ and resolving the nondeterminism according to $\mathfrak{S}$. If C is fully probabilistic, then $\mathfrak{S}$ is irrelevant and we simply write $[\![C]\!]^{\sigma}$. We speak of a *sub*distribution since the probabilities to reach some final state do not necessarily add up to 1, i.e., we might have

$$\underbrace{\sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)}_{\text{probability that } C \text{ terminates on } \sigma \text{ under } \mathfrak{S}} \qquad \overset{\text{possibly}}{<} \qquad 1\,,$$

and the "missing" probability mass is the probability that C diverges on σ under scheduler $\mathfrak{S}$. For instance, for the fully probabilistic program

$$C \;=\; \mathtt{while}\,(\mathsf{true})\,\{\mathtt{skip}\}\,,$$

we have for all initial states σ that

$$\sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}(\tau) \;=\; 0\,,$$

since C diverges with probability 1 on every initial state, and this fact does not depend on a scheduler since C is fully probabilistic.

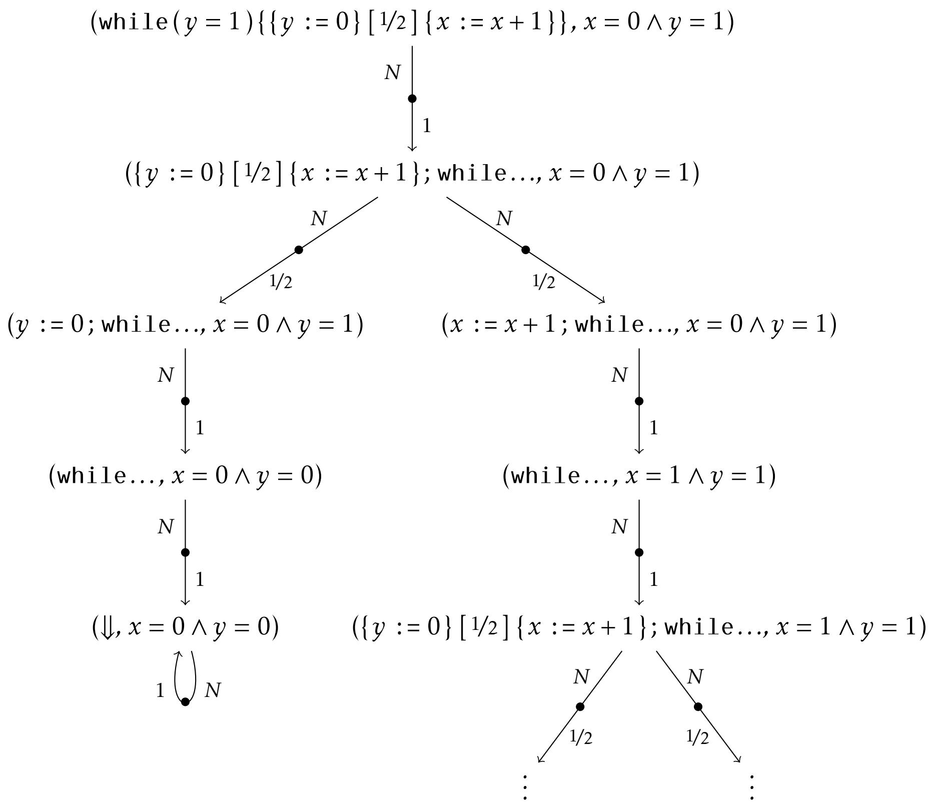
Example 2.18.
Reconsider the fully probabilistic loop C from Example 2.17 given by

$$\mathtt{while}\,(y=1)\,\{\{y := 0\}\,[\,1/2\,]\,\{x := x+1\}\}\,.$$

Figure 2.11 on page 52 depicts a reachable fragment of pGCL's operational MDP $\mathcal{O}$ from the initial configuration (C,σ) with $\sigma(x) = 0$ and $\sigma(y) = 1$. Since C is fully probabilistic, so is the depicted reachable fragment in the sense that for all reachable configurations $\mathfrak{c}$, we have $\mathsf{Act}(\mathfrak{c}) = \{N\}$. Notice that — even for a fixed initial configuration — the number of reachable configurations is *infinite* which is due to the fact that C is an unbounded loop.

2.4 Deductive Verification of Probabilistic Programs

We introduce the deductive program verification techniques we build upon in this thesis — the weakest preexpectation calculus for reasoning about expected



(while(y = 1){{y := 0} [1/2] {x := x + 1}}, x = 0 ∧ y = 1)
N
1
({y := 0} [1/2] {x := x + 1}; while..., x = 0 ∧ y = 1)
N
1/2
N
1/2
(y := 0; while..., x = 0 ∧ y = 1)
(x := x + 1; while..., x = 0 ∧ y = 1)
N
1
N
1
(while..., x = 0 ∧ y = 0)
(while..., x = 1 ∧ y = 1)
N
1
N
1
(⇓, x = 0 ∧ y = 0)
1
N
({y := 0} [1/2] {x := x + 1}; while..., x = 1 ∧ y = 1)
N
1/2
N
1/2

Figure 2.11: A reachable fragment of pGCL's operational MDP $\mathcal{O}$ for the loop from Example 2.17. Program states are denoted by conjunctions of equalities, i.e., $x = n \wedge y = m$ indicates that the current state σ satisfies $\sigma(x) = n$ and $\sigma(y) = m$.

outcomes of pGCL programs. Weakest preexpectation reasoning has been pioneered by Kozen [Koz83; Koz85] for fully probabilistic programs. McIver & Morgan [MM05; MMS96] extended weakest preexpectation reasoning to probabilistic programs featuring nondeterministic choices. McIver & Morgan's calculus is a quantitative generalization of Dijkstra's weakest precondition calculus [Dij75; Dij76] and enables reasoning about expected values of *bounded* random variables on a program's state space. Kaminski [Kam19] extended this calculus further to enable reasoning about *unbounded* and ∞*-valued* random variables, which allows to leverage the concepts from fixpoint theory presented in Section 2.1. We mainly follow the presentation from [Kam19].

2.4.1 Motivation: Reasoning about Expected Outcomes

The weakest preexpectation calculus enables us to reason about *expected outcomes* of probabilistic programs in a *syntax-based* manner, i.e., by reasoning on the source code of a given program. Before we deal with the fully-fledged formalization of this calculus, let us gain some intuition on expected outcomes. For that, let C — for the sake of simplicity — be a fully probabilistic program and let σ be a state. By "expected outcomes" we mean quantities like:

1. *What is the $\underline{\text{probability}}$ that C terminates on initial state σ?*
2. *Given a predicate $P \in \mathcal{P}(\mathsf{States})$, what is the $\underline{\text{probability}}$ that C terminates in a final state satisfying P on initial state σ?*
3. *Given a program variable x, what is the $\underline{\text{expected}}$ (or $\underline{\text{average}}$) final value of x on termination of C on initial state σ?*

Example 2.19.
Consider the simple pGCL program C given by

$$\{\mathtt{skip}\}\,[\nicefrac{1}{3}]\,\{x := x+3\}\ .$$

Program C flips a biased coin and either does nothing (left branch) or increments x by 3 (right branch). Let us consider each of the quantities from above for this concrete program C.

1. Program C terminates with probability 1 on every initial state.
2. Let P be the predicate $x = 3$. Given an initial program state σ, the probability that C terminates in a final state satisfying $x = 3$ depends on the initial value $\sigma(x)$ of x:

a) If $\sigma(x) = 3$, this probability is $1/3$ (due to the left branch).

b) If $\sigma(x) = 0$, this probability is $2/3$ (due to the right branch).

c) If neither $\sigma(x) = 3$ nor $\sigma(x) = 0$, this probability is 0.

3. Given an initial program state σ, the expected (or average) final value of x depends on the initial value $\sigma(x)$ of x: With probability $1/3$, the final of x is $\sigma(x)$. With probability $2/3$, the final of x is $\sigma(x) + 3$. The expected final value of x is thus given by the weighted sum

$$1/3 \cdot \sigma(x) + 2/3 \cdot (\sigma(x) + 3) \;=\; \sigma(x) + 2\,.$$

Hence, on average, C increments x by 2.

We now make the crucial observation that *all* of the above quantities can be expressed as *the expected value of a random variable* $X\colon \mathsf{States} \to \mathbb{R}_{\geq 0}^{\infty}$ *w.r.t. the distribution of final states obtained from executing* C *on* σ, which we define as

$$\sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau) \quad \in \mathbb{R}_{\geq 0}^{\infty}\,,$$

i.e., we sum over $X(\tau)$ for each state $\tau \in \mathsf{States}$ and weigh each summand by the probability $[\![C]\!]^{\sigma}(\tau)$ of terminating in τ. In particular:

1. The probability that C terminates on initial state σ is obtained by choosing $X = \lambda\tau.1$ — the random variable which maps every state to 1, i.e., C's termination probability on initial state σ is

$$\sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau) \;=\; \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau)\,.$$

2. The probability that C terminates in a final state satisfying a given predicate P on initial state σ is obtained by choosing $X = [P]$ — the *Iverson bracket* (or *indicator function*) of P defined as

$$[P] \;=\; \lambda\tau.\begin{cases} 1 & \text{if } \tau \models P \\ 0 & \text{if } \tau \not\models P\,, \end{cases}$$

because this yields

$$\sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau) \;=\; \sum_{\substack{\tau \in \mathsf{States} \\ \tau \models P}} [\![C]\!]^{\sigma}(\tau)\,.$$

Notice that termination probabilities, i.e., the probability of terminating in an *arbitrary* final state, is a special case of the above for $P = [\mathrm{true}]$.

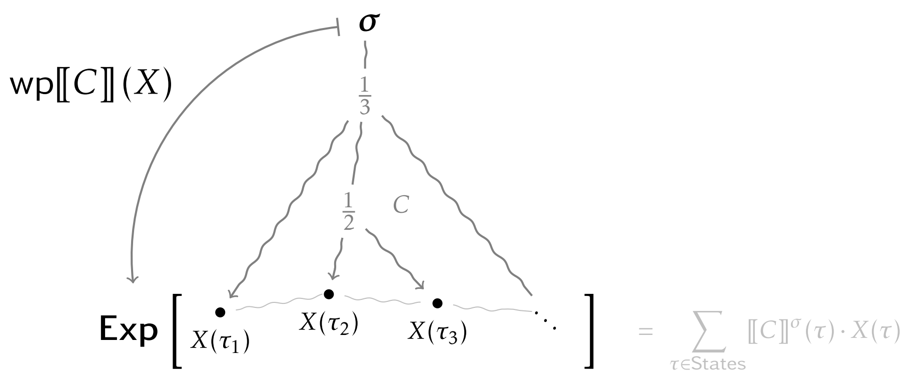

σ
$\mathsf{wp}[\![C]\!](X)$
$\frac{1}{3}$
$\frac{1}{2}$
C
Exp
$X(\tau_1)$
$X(\tau_2)$
$X(\tau_3)$
$= \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau)$

Figure 2.12: For fully probabilistic C, the weakest preexpectation $\mathsf{wp}[\![C]\!](X)$ of C w.r.t. postexpectation X maps each initial state σ to the expected value of X w.r.t. the distribution $[\![C]\!]^{\sigma}$ of final states obtained from executing C on σ (adapted from [15]).

3. The expected final value of variable x on termination of C on initial state σ is obtained by choosing $X = \lambda\tau.\,\tau(x)$ — the random variable which maps every state τ to the value of x under τ, because this yields
$$\sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau) \;=\; \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot \tau(x)\,.$$

In the jargon of the weakest preexpectation calculus, random variables of type $\mathsf{States} \to \mathbb{R}^{\infty}_{\geq 0}$ are called *expectations*[7]. The expectations X from above are referred to as *postexpectations*: They specify the expected outcomes we are interested in and are evaluated in the *final states*. The function which maps every *initial* state σ to the expected value of X w.r.t. the distribution of final states is again of type $\mathsf{States} \to \mathbb{R}^{\infty}_{\geq 0}$ — thus an expectation — called the *weakest preexpectation of C w.r.t. (postexpectation) X*, denoted by
$$\mathsf{wp}[\![C]\!](X) \;=\; \lambda\sigma.\, \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau)\,.$$

[7]The terminology is standard yet slightly misleading: By an "expectation" we mean functions of type $\mathsf{States} \to \mathbb{R}^{\infty}_{\geq 0}$, *not* an expected value.

Figure 2.12 depicts the function $\mathsf{wp}[\![C]\!](X)$ graphically.

For a nondeterministic program C, the distribution of final states is generally not unique but depends on the resolution of the nondeterminism, i.e., on a scheduler $\mathfrak{S}$ for pGCL's operational MDP $\mathcal{O}$. Hence, expected outcomes like the probability of termination or the expected final value of a program variable are generally not unique, as well. Analogously to minimal and maximal expected rewards in MDPs, we are interested in minimal and maximal — under all possible resolutions of the nondeterminism — expected outcomes. In the jargon of weakest preexpectations, we speak of the *demonic and angelic weakest preexpectation of C w.r.t. postexpectation X*, which are respectively denoted by

$$\mathsf{dwp}[\![C]\!](X) \;=\; \lambda\sigma.\; \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)\cdot X(\tau)\,,\text{and}$$

$$\mathsf{awp}[\![C]\!](X) \;=\; \lambda\sigma.\; \sup_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)\cdot X(\tau)\,.$$

If C is fully probabilistic, then $\mathsf{dwp}[\![C]\!](X)$ and $\mathsf{awp}[\![C]\!](X)$ coincide, i.e.,

$$\mathsf{wp}[\![C]\!](X) \;=\; \mathsf{dwp}[\![C]\!](X) \;=\; \mathsf{awp}[\![C]\!](X)\;.$$

In this sense, dwp and awp are extensions of wp for nondeterministic programs.

One of our prime objectives is to employ the weakest preexpectation calculus to automate the verification of *bounds on expected outcomes*: Given a program $C \in \mathsf{pGCL}$, expectations X, Y, and $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$, automatically check whether

$$\begin{array}{ll} & \mathcal{T}[\![C]\!](X) \;\sqsubseteq\; Y \\ \text{which is the case iff} \qquad & \text{for all } \sigma \in \mathsf{States}\colon \mathcal{T}[\![C]\!](X)(\sigma) \;\leq\; Y(\sigma)\,, \end{array}$$

i.e., automatically verify whether, for every initial state σ, the quantitiy $Y(\sigma)$ upper-bounds the expected outcome $\mathcal{T}[\![C]\!](X)(\sigma)$.

Example 2.20.
The verification of bounds on expected outcomes has natural applications when, e.g., reasoning about *probabilistic safety* of stochastic processes modeled by pGCL programs. Let us consider a variant of the *bounded retransmission protocol* (*BRP*) as an example [HSV93; DJJL01]. The goal of the BRP is to send a file consisting of N packets via a lossy channel. Each transmission of a packet can fail with probability 0.01. Whenever this happens, the packet has to be retransmitted. Crucially, the BRP allows for at most M

retransmissions per packet. If this number is exceeded for some packet, we say that the protocol *fails*, i.e., did not manage to transmit the file. This stochastic process is naturally modeled by the following pGCL program C:

```
$sent := 0; failed := 0;$
while($sent < N \wedge failed \leq M$){
    {$failed := failed + 1$} [0.01] {$failed := 0; sent := sent + 1$}
}
```

Variable *sent* keeps track of the number of successfully transmitted packages and *failed* keeps track of the number of retransmissions for the current packet. The loop terminates either when $sent = N$, i.e., when all packets have been successfully transmitted, or when $failed > M$, i.e., when the protocol fails. Notice that N and M are program variables, i.e., the number of packets and maximal retransmissions are part of the input to C. Now, a crucial property for the BRP is that *the probability that the protocol fails is sufficiently small*. The BRP fails whenever it terminates in a final state τ with $\tau \models failed > M$. Hence, the weakest preexpectation

$$\mathsf{wp}[\![C]\!]([failed > M])$$

maps every initial program state σ, i.e., all values for N and M, to the probability that the protocol fails for the given number of packets and maximal retransmissions. Suppose we know that whenever we run the protocol, there are at most $8 \cdot 10^6$ packets to send; and that we may allow for at least 5 retransmission tries. Let Y be the expectation defined as

$$Y = \lambda\sigma. \begin{cases} 0.001 & \text{if } \sigma(N) \leq 8 \cdot 10^6 \wedge \sigma(M) \geq 5 \\ \infty & \text{otherwise .} \end{cases}$$

By verifying that

$$\mathsf{wp}[\![C]\!]([failed > M]) \sqsubseteq Y$$

holds, we verify *for all relevant inputs to the protocol* that the probability of failing is at most 0.001.

It is one of Kozen's and McIver & Morgan's key insight that weakest preexpectations can be defined in a *syntax-based* manner, i.e., by reasoning on the *source code* of the given program C. In fact, we will see in Section 2.4.3 that both $\mathsf{dwp}[\![C]\!](X)$ and $\mathsf{awp}[\![C]\!](X)$ can be defined *by induction on the structure of* C. Hence, we can reason on the *finite* source code rather than the infinite-state operational MDP $\mathcal{O}$ and the infinite sets of paths involved in the above sums.

2.4.2 Expectations

In this section, we formalize expectations, treat their algebraic properties, and define the arithmetic operations needed for the weakest preexpectation calculus.

Definition 2.19 (Expectations).
The complete lattice of *expectations* is defined as

$$(\mathbb{E}, \sqsubseteq)\,,$$

where:

1. $\mathbb{E} = \mathsf{States} \to \mathbb{R}_{\geq 0}^{\infty}$ is the *set of expectations*,
2. $\sqsubseteq$ is the pointwise lifted order on $\mathbb{R}_{\geq 0}^{\infty}$, i.e.,
$$\text{for all } X, Y \in \mathbb{E}\colon \quad X \sqsubseteq Y \text{ iff for all } \sigma \in \mathsf{States}\colon X(\sigma) \leq Y(\sigma)\,.$$

The least element of $(\mathbb{E}, \sqsubseteq)$ is the constant-zero-expectation $\lambda\sigma.\,0$. The greatest element is $\lambda\sigma.\,\infty$. Moreover, suprema and infima are given as the pointwise liftings of suprema and infima in $(\mathbb{R}_{\geq 0}^{\infty}, \leq)$, i.e., for $E \subseteq \mathbb{E}$, we have

$$\bigsqcup E \;=\; \lambda\sigma.\,\sup\{X(\sigma) \mid X \in E\} \qquad \text{and} \qquad \bigsqcap E \;=\; \lambda\sigma.\,\inf\{X(\sigma) \mid X \in E\}\,.$$

Given two expectations $X, Y \in \mathbb{E}$, we often write

$$X \sqcup Y \text{ instead of } \bigsqcup\{X, Y\} \qquad \text{and similarly} \qquad X \sqcap Y \text{ instead of } \bigsqcap\{X, Y\}\,.$$

We define standard arithmetic operations on expectations as the pointwise liftings of these operations on the non-negative extended reals, i.e.,

$$X \cdot Y \;=\; \lambda\sigma.\,X(\sigma) \cdot Y(\sigma) \qquad \text{and} \qquad X + Y \;=\; \lambda\sigma.\,X(\sigma) + Y(\sigma)\,,$$

where we recall that we let $0 \cdot \infty = \infty \cdot 0 = 0$. We agree on the usual order of precedence for these operations, i.e., $\cdot$ binds stronger than $+$, and we use parentheses to resolve ambiguities. Moreover, we often identify constants in $\mathbb{R}_{\geq 0}^{\infty}$ with expectations, i.e., we identify $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$ with the constant expectation

$$\lambda\sigma.\,\alpha\,.$$

Similarly, we often identify a program variable x with the expectation

$$\lambda\sigma . \sigma(x) ,$$

which, on state σ, evaluates to the value of x under σ. The *Iverson bracket* $[P]$ of a predicate $P \in \mathcal{P}(\mathsf{States})$ casts P to an expectation, defined as

$$[P] \;=\; \lambda\sigma . \begin{cases} 1 & \text{if } \sigma \models P \\ 0 & \text{if } \sigma \not\models P . \end{cases}$$

Example 2.21.
The above conventions enable us to denote expectations conveniently, e.g.,

$$[z \leq 2] \cdot (x+2) + [z > 2] \cdot (2 \cdot y) \;=\; \lambda\sigma . \begin{cases} \sigma(x) + 2 & \text{if } \sigma(z) \leq 2 \\ 2 \cdot \sigma(y) & \text{if } \sigma(z) > 2 \\ 0 & \text{otherwise} . \end{cases}$$

Moreover, we can conveniently specify various weakest preexpectations:

1. Given a predicate P, $\mathsf{wp}[\![C]\!]([P])$ is the expectation which maps every initial state σ to the probability that C terminates in a state satisfying P. In particular, for the special case where $P = \mathsf{true}$, $\mathsf{wp}[\![C]\!]([\mathsf{true}])(\sigma)$ is the probability that C terminates on σ.

2. $\mathsf{wp}[\![C]\!](x)$ is the expectation which maps every initial state σ to the expected final value of variable x on termination of C.

We do, however, not give a concrete syntax for expectations here since we present the weakest preexpectation calculus in a fully extensional setting. A syntax for expectations is presented in Chapter 3.

Finally, given $X \in \mathbb{E}$, a variable x, and an arithmetic expression $A \colon \mathsf{States} \to \mathsf{Vals}$, we define the *expectation $X[x/A]$ obtained from replacing x in X with A* as

$$X[x/A] \;=\; \lambda\sigma . X(\sigma[x \mapsto A(\sigma)]) .$$

On the Domain of Program Variables. We do not let the program variables range over a domain containing *negative* numbers since, otherwise, the expectation $\lambda\sigma . \sigma(x)$ identified with variable x would generally not be well-defined (recall that expectations map program states to *non-negative*[8] extended reals). Even

[8]Weakest preexpectation calculi for possibly *negative* expectations exist [KK17] but are less handy and outside the scope of this thesis.

C	$\mathsf{dwp}[\![C]\!](X)$
`skip`	X
$x := A$	$X[x/A]$
$C_1\,;C_2$	$\mathsf{dwp}[\![C_1]\!](\mathsf{dwp}[\![C_2]\!](X))$
$\{C_1\}\,\Box\,\{C_2\}$	$\mathsf{dwp}[\![C_1]\!](X) \sqcap \mathsf{dwp}[\![C_2]\!](X)$
$\{C_1\}\,[p]\,\{C_2\}$	$p \cdot \mathsf{dwp}[\![C_1]\!](X) + (1-p) \cdot \mathsf{dwp}[\![C_2]\!](X)$
$\mathtt{if}\,(B)\,\{C_1\}\,\mathtt{else}\,\{C_2\}$	$[B] \cdot \mathsf{dwp}[\![C_1]\!](X) + [\neg B] \cdot \mathsf{dwp}[\![C_2]\!](X)$
$\mathtt{while}(B)\{C'\}$	$\mathsf{lfp}\, Y.\, [B] \cdot \mathsf{dwp}[\![C']\!](Y) + [\neg B] \cdot X$

Table 2.1: Inductive definition of $\mathsf{dwp}[\![C]\!]$ adapted from [Kam19].

though we could circumvent this by, e.g., identifying x with the expectations

$$\lambda\sigma.\, \max\{0, \sigma(x)\} \qquad \text{or} \qquad \lambda\sigma. |\sigma(x)|\,,$$

this renders — in the author's opinion — the weakest preexpectation calculus and its applications harder to understand. We remark that signed variables can be encoded by introducing auxiliary variables (see, e.g., [12, Section 11.2]).

2.4.3 The Weakest Preexpectation Calculus

In this section, we consider the weakest preexpectation calculus in more detail. In particular, we study how weakest preexpectations can be defined by induction on a program's structure, and how least fixpoint constructions yield weakest preexpectations of loops. For that, consider the following definition:

Definition 2.20 (Weakest Preexpectations).
Let $C \in \mathsf{pGCL}$ and $X \in \mathbb{E}$.

1. The *demonic weakest preexpectation transformer of C*

$$\mathsf{dwp}[\![C]\!] \colon \mathbb{E} \to \mathbb{E}$$

is defined by induction on C in Table 2.1. We call

$$\mathsf{dwp}[\![C]\!](X) \;\in\; \mathbb{E}$$

the *demonic weakest preexpectation of C w.r.t. postexpectation X*.

2. The *angelic weakest preexpectation transformer of C*

   $$\mathsf{awp}[\![C]\!]\colon \mathbb{E} \to \mathbb{E}$$

   is defined by induction on C in Table 2.1, where every occurrence of dwp is replaced by awp and $\sqcap$ is replaced by $\sqcup$. We call

   $$\mathsf{awp}[\![C]\!](X) \in \mathbb{E}$$

   the *angelic weakest preexpectation of C w.r.t. postexpectation X*.

The two transformers $\mathsf{dwp}[\![C]\!]$ and $\mathsf{awp}[\![C]\!]$ differ in the way they treat nondeterminism: *d*emonically (minimizing) versus *a*ngelically (maximizing). If C is fully probabilistic, then $\mathsf{dwp}[\![C]\!]$ and $\mathsf{awp}[\![C]\!]$ coincide, i.e.,

$$\text{for all } X \in \mathbb{E}\colon \quad \mathsf{dwp}[\![C]\!](X) \;=\; \mathsf{awp}[\![C]\!](X)~.$$

For fully probabilistic programs C, we often write $\mathsf{wp}[\![C]\!](X)$ called the *weakest preexpectation of C w.r.t postexpectation X*. Let us now go over each of the individual rules for $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$.

Effectless Program. Since `skip` does not modify the program state and terminates immediately, $\mathcal{T}[\![\mathtt{skip}]\!](X)$ is just X.

Assignment. $\mathcal{T}[\![x := A]\!](X)$ is obtained from substituting x in X by A.

Sequential Composition. The rule for a sequential composition $C_1\,;C_2$ suggests that weakest preexpectations are obtained in a *backward-moving, continuation-passing fashion*: We first determine $\mathcal{T}[\![C_2]\!](X)$. This intermediate preexpectation is then plugged into $\mathcal{T}[\![C_1]\!]$, which yields $\mathcal{T}[\![C_1\,;C_2]\!](X)$.

Probabilistic Choice. The weakest preexpectation of a probabilistic choice is the sum of the preexpectations of the two branches, where the summands are weighted by the probability of the respective branch being executed.

Nondeterministic Choice. We have

$$\begin{aligned}\mathsf{dwp}[\![\{C_1\}\,\square\,\{C_2\}]\!](X) &= \lambda\sigma.\,\min\{\mathsf{dwp}[\![C_1]\!](X)(\sigma), \mathsf{dwp}[\![C_2]\!](X)(\sigma)\}~, \text{ and}\\ \mathsf{awp}[\![\{C_1\}\,\square\,\{C_2\}]\!](X) &= \lambda\sigma.\,\max\{\mathsf{awp}[\![C_1]\!](X)(\sigma), \mathsf{awp}[\![C_2]\!](X)(\sigma)\}~,\end{aligned}$$

i.e., the demonic (resp. angelic) weakest preexpectation of $\{C_1\}\,\square\,\{C_2\}$ is given by the pointwise minimum (resp. maximum) of the demonic (resp. angelic) weakest preexpectations of the two branches.

Conditional Choice. We have

$$\mathcal{T}[\![\mathtt{if}\,(B)\,\{C_1\}\,\mathtt{else}\,\{C_2\}]\!](X) \;=\; \lambda\sigma.\begin{cases}\mathcal{T}[\![C_1]\!](X)(\sigma) & \text{if } \sigma \models B\\ \mathcal{T}[\![C_2]\!](X)(\sigma) & \text{if } \sigma \not\models B\,,\end{cases}$$

i.e., depending on whether the program state σ satisfies the guard B or not, the weakest preexpectation of the branch that is to be executed is chosen.

Example 2.22.
For loop-free C, determining $\mathcal{T}[\![C]\!](X)$ consists mostly of syntactic reasoning[a]. Reconsider the program C from Example 2.19 given by

$$\{\mathtt{skip}\}\,[1/3]\,\{x := x+3\}\;.$$

1. We calculate $\mathsf{wp}[\![C]\!]([x=3])$ — the expectation which maps every initial state σ to the probability that C terminates in a final state satisfying $x=3$ on initial state σ:

$$\begin{aligned}&\mathsf{wp}[\![\{\mathtt{skip}\}\,[1/3]\,\{x := x+3\}]\!]([x=3])\\ =\;& 1/3\cdot\mathsf{wp}[\![\mathtt{skip}]\!]([x=3]) + 2/3\cdot\mathsf{wp}[\![x := x+3]\!]([x=3])\\ =\;& 1/3\cdot[x=3] + 2/3\cdot[x+3=3]\\ =\;& 1/3\cdot[x=3] + 2/3\cdot[x=0]\\ =\;& \lambda\sigma.\begin{cases}1/3 & \text{if } \sigma\models x=3\\ 2/3 & \text{if } \sigma\models x=0\\ 0 & \text{otherwise}\,.\end{cases}\end{aligned}$$

2. We calculate $\mathsf{wp}[\![C]\!](x)$ — the expectation which maps every initial state σ to the expected final value of x obtained from executing C:

$$\begin{aligned}&\mathsf{wp}[\![\{\mathtt{skip}\}\,[1/3]\,\{x := x+3\}]\!](x)\\ =\;& 1/3\cdot\mathsf{wp}[\![\mathtt{skip}]\!](x) + 2/3\cdot\mathsf{wp}[\![x := x+3]\!](x)\\ =\;& 1/3\cdot x + 2/3\cdot(x+3)\\ =\;& x+2\,,\end{aligned}$$

i.e., on average, C increments x by 2.

[a]It is not entirely justified to speak of "syntactic reasoning" here since expectations are mere

mathematical functions rather than syntactic objects. We refer to Chapter 3 for a formal language of expectations.

While Loop. Weakest preexpectations of loops are defined by means of a least fixpoint construction. To get an intuition on this construction, we first observe that the weakest preexpectations of the two programs

$$\texttt{while}\,(B)\,\{C'\} \quad \text{and} \quad \texttt{if}\,(B)\,\{C';\texttt{while}\,(B)\,\{C'\}\}\,\texttt{else}\,\{\texttt{skip}\}$$

must coincide, since these two programs are semantically equivalent. Hence, by the rules for `skip`, sequential composition, and conditional choice from Table 2.1, $\mathcal{T}[\![\texttt{while}\,(B)\,\{C'\}]\!](X)$ must satisfy the recursive equation

$$\begin{aligned} &\mathcal{T}[\![\texttt{while}\,(B)\,\{C'\}]\!](X) \\ =\ & [B]\cdot\mathcal{T}[\![C']\!](\mathcal{T}[\![\texttt{while}\,(B)\,\{C'\}]\!](X)) + [\neg B]\cdot X\,, \end{aligned} \tag{2.4}$$

and the *least* solution of this equation in the complete lattice $(\mathbb{E}, \sqsubseteq)$ precisely captures (minimal or maximal) expected outcomes of loops. To formalize this in fixpoint-theoretic terms, we associate with each loop and postexpectation respective functions of type $\mathbb{E}\to\mathbb{E}$:

Definition 2.21 (Characteristic Functions of Loops).
Let $C \in \mathsf{pGCL}$ be the loop $\texttt{while}\,(B)\,\{C'\}$, and let $X \in \mathbb{E}$.

1. The dwp-*characteristic function of C w.r.t. X* is defined as
$${}^{\mathsf{dwp}}_{C}\Phi_X \colon \mathbb{E}\to\mathbb{E}\,, \qquad {}^{\mathsf{dwp}}_{C}\Phi_X(Y) \;=\; [B]\cdot\mathsf{dwp}[\![C']\!](Y) + [\neg B]\cdot X\,.$$
2. The awp-*characteristic function of C w.r.t. X* is defined as
$${}^{\mathsf{awp}}_{C}\Phi_X \colon \mathbb{E}\to\mathbb{E}\,, \qquad {}^{\mathsf{awp}}_{C}\Phi_X(Y) \;=\; [B]\cdot\mathsf{awp}[\![C']\!](Y) + [\neg B]\cdot X\,.$$

If C is fully probabilistic, we often use

3. the wp-*characteristic function of C w.r.t. X* defined as
$${}^{\mathsf{wp}}_{C}\Phi_X \colon \mathbb{E}\to\mathbb{E}\,, \qquad {}^{\mathsf{wp}}_{C}\Phi_X(Y) \;=\; [B]\cdot\mathsf{wp}[\![C']\!](Y) + [\neg B]\cdot X\,.$$

Hence, the least solution of Equation (2.4) is given by the least fixpoint of the respective characteristic function, i.e.,

$$\mathsf{dwp}[\![C]\!](X) \;=\; \mathsf{lfp}\ {}^{\mathsf{dwp}}_{C}\Phi_X \qquad \text{and} \qquad \mathsf{awp}[\![C]\!](X) \;=\; \mathsf{lfp}\ {}^{\mathsf{awp}}_{C}\Phi_X\,.$$

These least fixpoints are guaranteed to exist uniquely:

Theorem 2.11 (Healthiness Conditions [Kam19]).
Let $C \in \mathsf{pGCL}$ and $X \in \mathbb{E}$.

1. Both $\mathsf{dwp}[\![C]\!]$ and $\mathsf{awp}[\![C]\!]$ are monotonic w.r.t. $(\mathbb{E}, \sqsubseteq)$.
2. Both $\mathsf{dwp}[\![C]\!]$ and $\mathsf{awp}[\![C]\!]$ are continuous w.r.t. $(\mathbb{E}, \sqsubseteq)$.
3. For loop $C = \mathtt{while}(B)\{C'\}$, both ${}^{\mathsf{dwp}}_{\;C}\Phi_X$ and ${}^{\mathsf{awp}}_{\;C}\Phi_X$ are continuous w.r.t. $(\mathbb{E}, \sqsubseteq)$. Hence, by Kleene's Theorem 2.2, we have
$$\mathsf{lfp}\ {}^{\mathsf{dwp}}_{\;C}\Phi_X ~=~ \bigsqcup_{n \in \mathbb{N}} {}^{\mathsf{dwp}}_{\;C}\Phi_X^n(0) \qquad \text{and} \qquad \mathsf{lfp}\ {}^{\mathsf{awp}}_{\;C}\Phi_X ~=~ \bigsqcup_{n \in \mathbb{N}} {}^{\mathsf{awp}}_{\;C}\Phi_X^n(0)\,.$$

As with reachability in transition systems (Remark 2.1) and expected rewards in MDPs (Lemma 2.8), Kleene's Theorem 2.2 helps us to understand why this least fixpoint construction makes sense and indeed gives the expected outcomes we are interested in. Define an auxiliary program statement abort satisfying[9]

$$\text{for all } X \in \mathbb{E}: \quad \mathsf{dwp}[\![\mathtt{abort}]\!](X) ~=~ \mathsf{awp}[\![\mathtt{abort}]\!](X) ~=~ 0\,,$$

i.e., the expected final value of X on termination of abort is 0. Now define for each loop $C = \mathtt{while}(B)\{C'\}$ and each $n \in \mathbb{N}$ the *n-th unfolding of C* as

$$\begin{aligned} &\mathtt{while}^{\leq n}(B)\{C'\} \\ =\; &\begin{cases} \mathtt{abort} & \text{if } n = 0 \\ \mathtt{if}\,(B)\left\{C';\mathtt{while}^{\leq n-1}(B)\{C'\}\right\}\mathtt{else}\,\{\mathtt{skip}\} & \text{if } n > 0\,. \end{cases} \end{aligned}$$

Intuitively, $\mathtt{while}^{\leq n}(B)\{C'\}$ behaves likes $\mathtt{while}(B)\{C'\}$ but aborts after n iterations. We have for $\mathcal{T} \in \{\mathsf{awp}, \mathsf{dwp}\}$ and all $n \in \mathbb{N}$,

$${}^{\mathcal{T}}_{C}\Phi_X^n(0) ~=~ \mathcal{T}[\![\mathtt{while}^{\leq n}(B)\{C'\}]\!](X)\,.$$

Hence, ${}^{\mathcal{T}}_{C}\Phi_X^n(0)(\sigma)$ is the (minimal or maximal) expected final value of X on termination of C on initial state σ when restricting C to at most n loop iterations. Allowing for an *arbitrary* number of loop iterations by taking the supremum over n thus yields by Theorem 2.11.3

$$\bigsqcup_{n \in \mathbb{N}} \mathcal{T}[\![\mathtt{while}^{\leq n}(B)\{C'\}]\!](X) ~=~ \bigsqcup_{n \in \mathbb{N}} {}^{\mathcal{T}}_{C}\Phi_X^n(0) ~=~ \mathcal{T}[\![C]\!](X)\,.$$

[9] abort is syntactic sugar for while(true){skip} but using abort is more intuitive in this context.

Example 2.23.
Reconsider loop C from Example 2.17 given by

$$\texttt{while}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x + 1\}\}$$

and recall that for initial state σ satisfying $y = 1$, the probability that C terminates in a final state τ with $\tau(x) = \sigma(x) + i$ is

$$[\![C]\!]^{\sigma}(\tau) \;=\; \frac{1}{2^{i+1}}\;.$$

Hence, the expected final value of x is

$$\sum_{i=0}^{\infty} \frac{1}{2^{i+1}} \cdot (\sigma(x) + i) \;=\; \sigma(x) + 1\;,$$

i.e., if executed on an initial state satisfying its loop guard, C increments x by 1 in expectation. Let us verify this by means of the weakest preexpectation calculus. For that, we calculate $\mathsf{wp}[\![C]\!]\,(x)$ using Theorem 2.11.3. It can be shown by induction on $n \geq 1$ that

$$\begin{aligned} &\mathsf{wp}[\![\texttt{while}^{\leq n}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x + 1\}\}]\!]\,(x) \\ =\;& {}^{\mathsf{wp}}_{C}\Phi^{n}_{X}(0) \;=\; [y = 1] \cdot \left(\sum_{i=0}^{n-2} \frac{x+i}{2^{i+1}}\right) + [y \neq 1] \cdot x \end{aligned}$$

Hence, by Theorem 2.11.3, we get

$$\begin{aligned} \mathsf{wp}[\![C]\!]\,(x) \;&=\; \bigsqcup_{n \in \mathbb{N}} [y = 1] \cdot \left(\sum_{i=0}^{n-2} \frac{x+i}{2^{i+1}}\right) + [y \neq 1] \cdot x \\ &=\; [y = 1] \cdot (x + 1) + [y \neq 1] \cdot x\;, \end{aligned}$$

i.e., on an initial state satisfying the loop guard, C increments x by 1 in expectation (first summand). Otherwise, x is not modified at all, which yields x's expected final value to be its initial value (second summand).

2.4.4 Quantitative Loop Invariants

Reasoning about weakest preexpectations of loops is difficult and one of the most challenging tasks in probabilistic program verification. Weakest preexpectations of loops are defined as higher-order least fixpoints of the respective characteristic functions, which are uncomputable in general [KKM19]. *Invariant-based* reasoning for loops simplifies the verification of bounds on expected outcomes of loops significantly. Reasoning with quantitative loop invariants (or *superinvariants*) is one of the key techniques employed in this thesis.

Definition 2.22 (Superinvariants).
Let $C = \texttt{while}(B)\{C'\}$ and let $X, I \in \mathbb{E}$.

1. We call I a dwp-*superinvariant of* C *w.r.t.* X, if
$$^{\mathsf{dwp}}_{\ \ C}\Phi_X(I) \ \sqsubseteq\ I\ .$$
2. We call I an awp-*superinvariant of* C *w.r.t.* X, if
$$^{\mathsf{awp}}_{\ \ C}\Phi_X(I) \ \sqsubseteq\ I\ .$$
3. If C is fully probabilistic, we call I a wp-*superinvariant of* C *w.r.t.* X, if
$$^{\mathsf{wp}}_{\ C}\Phi_X(I) \ \sqsubseteq\ I\ .$$

Since weakest preexpectations of loops are defined as least fixpoints monotone functions over the complete lattice of expectations, it follows by Park induction (Lemma 2.4) that superinvariants yield upper bounds on expected outcomes:

Theorem 2.12 (Park Induction for Loops [Par69; Kam19]).
Let $C = \texttt{while}(B)\{C'\}$ be a loop and let $X, I \in \mathbb{E}$. We have:

1. If I is a dwp-superinvariant of C w.r.t. X, then
$$\mathsf{dwp}[\![C]\!](X) \ \sqsubseteq\ I\ .$$
2. If I is a awp-superinvariant of C w.r.t. X, then
$$\mathsf{awp}[\![C]\!](X) \ \sqsubseteq\ I\ .$$
3. If C is fully probabilistic and I is a wp-superinvariant of C w.r.t. X, then
$$\mathsf{wp}[\![C]\!](X) \ \sqsubseteq\ I\ .$$

Example 2.24.
Reconsider the loop C from Example 2.23 given by
$$\texttt{while}(y=1)\{\{y := 0\}\,[1/2]\,\{x := x+1\}\}\ .$$
Determining that $\mathsf{wp}[\![C]\!](x) = [y=1]\cdot(x+1) + [y \neq 1]\cdot x$ was quite involved since we needed to come up with a closed-form expression for the n-th

fixpoint iterate of ${}^{\mathsf{wp}}_{C}\Phi_X$, and to calculate the corresponding supremum over n. On the other hand, proving that

$$I \;=\; [y=1]\cdot(x+1)+[y\neq 1]\cdot x$$

upper-bounds $\mathsf{wp}[\![C]\!]\,(x)$ by means of Theorem 2.12 is way easier: We have to verify that I is a wp-superinvariant of C w.r.t. x. For that, we calculate:

$$\begin{aligned}
&{}^{\mathsf{wp}}_{C}\Phi_x(I)\\
={}& [y=1]\cdot \mathsf{wp}[\![\{y := 0\}\,[1/2]\,\{x := x+1\}]\!]\,(I)+[y\neq 1]\cdot x\\
={}& [y=1]\cdot(1/2\cdot I\,[y/0]+1/2\cdot I\,[x/x+1])+[y\neq 1]\cdot x\\
={}& [y=1]\cdot(1/2\cdot x+1/2\cdot([y=1]\cdot(x+2)+[y\neq 1]\cdot(x+1))+[y\neq 1])\cdot x\\
={}& [y=1]\cdot(1/2\cdot x+1/2\cdot(x+2))+[y\neq 1])\cdot x\\
={}& [y=1]\cdot(x+1)+[y\neq 1]\cdot x\\
={}& I \sqsubseteq I\,.
\end{aligned}$$

Hence, Theorem 2.12 yields

$$\mathsf{wp}[\![C]\!]\,(x) \;\sqsubseteq\; I \;=\; [y=1]\cdot(x+1)+[y\neq 1]\cdot x\,.$$

Notice that checking the superinvariant condition only requires us to compute a single weakest preexpectation of the *loop body*. This is particularly easy if the loop body itself is *loop-free* since the wp computation consists mostly of syntactic reasoning. It is the power of invariant-based reasoning that it suffices to reason about the loop body in order to obtain upper bounds on the entire loop. In Chapters 4 to 6 we exploit this fact to automate the verification of upper bounds on weakest preexpectations of loops.

Understanding Park Induction For Loops. Let $C = \mathtt{while}\,(B)\{C'\}$ be a loop and $X \in \mathbb{E}$. Recall from Theorem 2.11.3 that for $\mathcal{T} \in \{\mathsf{awp}, \mathsf{dwp}\}$,

$$\mathcal{T}[\![C]\!]\,(X) \;=\; \bigsqcup_{n\in\mathbb{N}} {}^{\mathcal{T}}_{C}\Phi^n_X(0) \;=\; \bigsqcup_{n\in\mathbb{N}} \mathcal{T}[\![\mathtt{while}^{\leq n}\,(B)\{C'\}]\!]\,(X)\,.$$

Now let $I \in \mathbb{E}$ and suppose we want to prove that $\mathcal{T}[\![C]\!]\,(X) \sqsubseteq I$ holds. Using the above identity and the definition of suprema, this is the case if and only if

$$\text{for all } n\in\mathbb{N}\colon\quad \mathcal{T}[\![\mathtt{while}^{\leq n}\,(B)\{C'\}]\!]\,(X) \;\sqsubseteq\; I\,.$$

We may thus attempt to prove $\mathcal{T}[\![C]\!](X) \sqsubseteq I$ by induction on n, which, intuitively, corresponds to a proof by induction on the number of iterations the loop C performs. The base case $n = 0$ is trivial since $\mathcal{T}[\![\texttt{while}^{\leq 0}(B)\{C'\}]\!](X) = 0$. For the induction step, consider the following:

$$\begin{aligned}
& \mathcal{T}[\![\texttt{while}^{\leq n+1}(B)\{C'\}]\!](X) \\
={}& \mathcal{T}[\![\texttt{if}\,(B)\left\{C';\texttt{while}^{\leq n}(B)\{C'\}\right\}\texttt{else}\,\{\texttt{skip}\}]\!](X) \\
={}& [B]\cdot\mathcal{T}[\![C']\!]\big(\mathcal{T}[\![\texttt{while}^{\leq n}(B)\{C'\}]\!](X)\big) + [\neg B]\cdot X && \text{(Table 2.1)} \\
\sqsubseteq{}& [B]\cdot\mathcal{T}[\![C']\!](I) + [\neg B]\cdot X && \text{(I.H. and monotonicity of } \mathcal{T}) \\
={}& {}^{\mathcal{T}}_{C}\Phi_X(I) && \text{(Definition 2.21)}
\end{aligned}$$

Now, if we know that I is a $\mathcal{T}$-superinvariant of C w.r.t. X, then we may complete the above reasoning for the induction step by

$$\sqsubseteq\ I\,, \qquad\qquad (I \text{ is } \mathcal{T}\text{-superinvariant of } C \text{ w.r.t. } X)$$

which is what we have to show. Park induction for loops can thus be understood as a proof by induction on the number of iterations the given loop performs, where we prove that I is a $\mathcal{T}$-superinvariant of C in order to have a sufficiently strong induction hypothesis at hand.

Lower Bounds on Weakest Preexpectations of Loops. Invariant-based proof rules for *lower-bounding* weakest preexpectation of loops, i.e., for proving

$$I \sqsubseteq \mathcal{T}[\![\texttt{while}(B)\{C'\}]\!](X)$$

exist but are more involved. In particular, the analogue of Theorem 2.12 for lower bounds, i.e., that for $C = \texttt{while}(B)\{C'\}$ and $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$ it holds that

$$ϟ \qquad I \sqsubseteq {}^{\mathcal{T}}_{C}\Phi_X(I) \qquad \text{implies} \qquad I \sqsubseteq \mathcal{T}[\![C]\!](X) \qquad ϟ$$

is *unsound in general* (cf. [Kam19, Counterexample 5.8]). We refer to [HKGK20] and [Kam19, Chapter 5] for an overview of invariant-based proof rules for lower-bounding weakest preexpectations of loops. In Section 5.7.2, we present a restricted variant of the proof rule from [HKGK20] for lower-bounding possibly unbounded expected outcomes of loops.

2.4.5 Soundness of the Weakest Preexpectation Calculus

In this section, we prove that the inductive definitions of dwp and awp (Definition 2.20) indeed coincide with their characterizations from Section 2.4.1 which refer to the distribution of final states defined via pGCL's operational MDP $\mathcal{O}$, i.e., that for all $C \in \mathsf{pGCL}$ and $X \in \mathbb{E}$, we have

$$\begin{aligned}\mathsf{dwp}[\![C]\!](X) &= \lambda\sigma.\ \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)\cdot X(\tau)\,, \text{ and}\\ \mathsf{awp}[\![C]\!](X) &= \lambda\sigma.\ \sup_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{\tau\in\mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau)\cdot X(\tau)\,.\end{aligned}$$

We call this *soundness* of the weakest preexpectation calculus because it is in this sense that dwp and awp indeed soundly determine minimal and maximal expected outcomes of probabilistic programs. An analogous result has been proven in [GKM14] for *bounded* expectations[10]. Our proofs very closely follow the lines of [5; 3, Theorem 4.5] (for dwp) and [17, Theorem 4.6] (for awp), unifying these results with minor simplifications.

We proceed by proving that $\mathsf{dwp}[\![C]\!](X)$ and $\mathsf{awp}[\![C]\!](X)$ evaluate to appropriate minimal and maximal expected rewards in the operational MDP $\mathcal{O}$. For that, we associate with each $X \in \mathbb{E}$ a reward function rew_X for $\mathcal{O}$ defined as

$$\mathsf{rew}_X\colon \{(\Downarrow,\tau) \mid \tau \in \mathsf{States}\} \to \mathbb{R}^{\infty}_{\geq 0}\,, \qquad \mathsf{rew}_X((\Downarrow,\tau)) = X(\tau)\,.$$

Definition 2.23 (Operational Weakest Preexpectations).
Let $C \in \mathsf{pGCL}$ and $X \in \mathbb{E}$.

1. We define the *demonic operational weakest preexpectation of C w.r.t. X* as
$$\mathsf{dop}[\![C]\!](X) = \lambda\sigma.\mathsf{MinER}(\mathcal{O},(C,\sigma) \models \Diamond\mathsf{rew}_X)\,.$$
2. We define the *angelic operational weakest preexpectation of C w.r.t. X* as
$$\mathsf{aop}[\![C]\!](X) = \lambda\sigma.\mathsf{MaxER}(\mathcal{O},(C,\sigma) \models \Diamond\mathsf{rew}_X)\,.$$

Our goal is now to show that for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$, we have

$$\mathsf{dwp}[\![C]\!](X) = \mathsf{dop}[\![C]\!](X) \quad \text{and} \quad \mathsf{awp}[\![C]\!](X) = \mathsf{aop}[\![C]\!](X)\,.$$

We proceed by proving two inequalities for each of the respective transformers; and then exploit antisymmetry of $\sqsubseteq$. We start with the $\sqsubseteq$-direction.

[10] An expectation X is *bounded*, if there is a constant $\alpha \in \mathbb{R}$ such that $X(\sigma) \leq \alpha$ for all $\sigma \in \mathsf{States}$.

Lemma 2.13.
We have for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$:

1. $\mathsf{dop}[\![C]\!]\,(X) \sqsubseteq \mathsf{dwp}[\![C]\!]\,(X)$
2. $\mathsf{aop}[\![C]\!]\,(X) \sqsubseteq \mathsf{awp}[\![C]\!]\,(X)$

Proof. We prove the claim for dwp. The proof for awp is completely analogous. The key idea is to apply Park induction (Lemma 2.4) to the min-Bellman operator of pGCL's operational MDP $\mathcal{O}$ w.r.t. rew_X (Definition 2.15.1). For that, we first observe that for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$, we have

$$\begin{aligned} & \mathsf{dwp}[\![C]\!]\,(X) \\ =\ & \lambda\sigma.\ \min_{\mathfrak{a}\in\mathsf{Act}(C,\sigma)} \sum_{(C,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} p\cdot\begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C']\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}' = (C',\sigma')\,, \end{cases} \end{aligned} \tag{2.5}$$

which follows by induction on the structure of C (cf. Lemma A.3). Now define the value function v for pGCL's operational MDP $\mathcal{O}$ as

$$v \ =\ \lambda\mathfrak{c}.\begin{cases} X(\tau) & \text{if } \mathfrak{c} = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C]\!]\,(X)\,(\sigma) & \text{if } \mathfrak{c} = (C,\sigma)\,. \end{cases}$$

It is an immediate consequence of Equation (2.5) that

$$^{\min}_{\mathcal{O}}\Phi_{\mathsf{rew}_X}(v) \ =\ v \ \sqsubseteq\ v\,,$$

which, by Park induction (Lemma 2.4), yields

$$\mathsf{lfp}\ ^{\min}_{\mathcal{O}}\Phi_{\mathsf{rew}_X} \ \sqsubseteq\ v\,.$$

By Theorem 2.9.1, we thus get

$$\mathsf{lfp}\ ^{\min}_{\mathcal{O}}\Phi_{\mathsf{rew}_X} \ =\ \lambda\mathfrak{c}.\,\mathsf{MinER}(\mathcal{O},\mathfrak{c}\models\Diamond\mathsf{rew}_X) \ \sqsubseteq\ v$$

so, in particular, we have for every $C \in \mathsf{pGCL}, X \in \mathbb{E}$, and $\sigma \in \mathsf{States}$,

$$\begin{aligned} & \mathsf{dop}[\![C]\!]\,(X)\,(\sigma) \\ =\ & \mathsf{MinER}(\mathcal{O},(C,\sigma)\models\Diamond\mathsf{rew}_X) \\ \leq\ & v(C,\sigma) \\ =\ & \mathsf{dwp}[\![C]\!]\,(X)\,(\sigma)\,, \end{aligned}$$

which is what we have to show. ∎

Next, we prove the converse direction.

Lemma 2.14.
We have for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$:

1. $\mathsf{dwp}[\![C]\!](X) \ \sqsubseteq\ \mathsf{dop}[\![C]\!](X)$
2. $\mathsf{awp}[\![C]\!](X) \ \sqsubseteq\ \mathsf{aop}[\![C]\!](X)$.

Proof. We prove the claim for dwp. The proof for awp is completely analogous. The key observation is that dop satisfies the following big-step decomposition w.r.t. sequential composition: For all $C_1, C_2 \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$,

$$\mathsf{dop}[\![C_1]\!](\mathsf{dop}[\![C_2]\!](X)) \ \sqsubseteq\ \mathsf{dop}[\![C_1\,;\,C_2]\!](X)\ .$$

(Lemma A.4 on page 273)

The claim then follows by induction on C. The most interesting case is $C = \texttt{while}(B)\{C'\}$: We show that $\mathsf{dop}[\![C]\!](X)$ is a dwp-superinvariant of C w.r.t. X (Definition 2.22.1). For that, consider the following:

$$\begin{aligned}
&{}^{\mathsf{dwp}}_{C}\Phi_X(\mathsf{dop}[\![C]\!](X)) \\
=\ & [B]\cdot \mathsf{dwp}[\![C']\!](\mathsf{dop}[\![C]\!](X)) + [\neg B]\cdot X && \text{(Definition 2.22.1)}\\
\sqsubseteq\ & [B]\cdot \mathsf{dop}[\![C']\!](\mathsf{dop}[\![C]\!](X)) + [\neg B]\cdot X && \text{(I.H.)}\\
\sqsubseteq\ & [B]\cdot \mathsf{dop}[\![C'\,;\,C]\!](X) + [\neg B]\cdot X && \text{(Lemma A.4)}\\
\sqsubseteq\ & \lambda\sigma.\begin{cases}\mathsf{MinER}(\mathcal{O}, (C';C,\sigma)\models \Diamond \mathsf{rew}_X) & \text{if } \sigma\models B\\ X(\sigma) & \text{if } \sigma\not\models B\end{cases}\\
\sqsubseteq\ & \lambda\sigma.\,\mathsf{MinER}(\mathcal{O},(C,\sigma)\models\Diamond\mathsf{rew}_X) && \text{(Definition 2.13.3 and Figure 2.10)}\\
=\ & \mathsf{dop}[\![C]\!](X)\ . && \text{(Definition 2.23)}
\end{aligned}$$

Hence, $\mathsf{dwp}[\![C]\!](X) \sqsubseteq \mathsf{dop}[\![C]\!](X)$ follows by Theorem 2.12.1. ■

Lemmas 2.13 and 2.14, and antisymmetry of $\sqsubseteq$ now yield the desired claim.

Theorem 2.15 (Soundness of the Weakest Preexpectation Calculus).
We have for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$:

1. $\mathsf{dwp}[\![C]\!](X) \ =\ \mathsf{dop}[\![C]\!](X)$
2. $\mathsf{awp}[\![C]\!](X) \ =\ \mathsf{aop}[\![C]\!](X)$

With our observations from Sections 2.2.3 and 2.2.5, we finally obtain the following characterizations of demonic and angelic weakest preexpectations.

Corollary 2.16.
We have for all $C \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$:

1. $\mathsf{dwp}[\![C]\!](X) \;=\; \lambda\sigma.\; \min_{\mathfrak{S} \in \mathsf{MLScheds}} \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau) \cdot X(\tau)$
2. $\mathsf{awp}[\![C]\!](X) \;=\; \lambda\sigma.\; \sup_{\mathfrak{S} \in \mathsf{Scheds}} \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}_{\mathfrak{S}}(\tau) \cdot X(\tau)$

Moreover, if C is fully probabilistic, then

3. $\mathsf{wp}[\![C]\!](X) \;=\; \lambda\sigma.\; \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma}(\tau) \cdot X(\tau)\,.$

Proof. These claims follow from Theorem 2.15 and Theorem 2.6 since every configuration of the form $(\Downarrow, \tau)$ is a sink in MDP $\mathcal{O}$. The fact that in Corollary 2.16.1 it suffices to take the *minimum* over all *memoryless* schedulers follows from Theorem 2.10. Finally, we may omit schedulers in Corollary 2.16.3 because there is no nondeterministic choice reachable from (C, σ). ∎

Remark 2.2 (On the Kozen Duality for Fully Probabilistic Programs).
Corollary 2.16.3 for fully probabilistic programs is an instance of the *Kozen duality* [Koz83; Koz85], which is a more general duality between expectation transformers and measure transformers.

Remark 2.3 (On the Relevance of Unbounded Expected Rewards).
We are now in a position to see the relevance of *unbounded* reward functions and expected rewards. Consider, e.g., the nondeterministic loop C given by

$$\texttt{while}(y = 1)\{\{y := 0\}\,[\nicefrac{1}{2}]\,\{\{x := x+1\} \,\square\, \{x := 2\cdot x\}\}\}\,.$$

On every iteration, C flips a fair coin and either terminates with probability $\nicefrac{1}{2}$ (left branch) or nondeterministically (right branch) increments x by 1 or doubles the value of x, also with probability $\nicefrac{1}{2}$. Now suppose we want to reason about the maximal expected final value of x. We have

$$\mathsf{awp}[\![C]\!](x) \;=\; [y = 1]\cdot\infty + [y \neq 1]\cdot x\,,$$

i.e., the maximal expected final value of x is unboundedly large for all initial states satisfying the loop guard. The intuition is that the exponential decay of the probability to keep iterating is compensated by the exponential growth of x. Now, establishing the link

$$\mathsf{awp}[\![C]\!](x) \;=\; \lambda\sigma.\,\mathsf{MaxER}(\mathcal{O},(C,\sigma) \models \Diamond\mathsf{rew}_x)$$

from Theorem 2.15 requires the right-hand side to possibly be ∞-valued.

Remark 2.4 (On Existing Variants of Theorem 2.15).
To the best of our knowledge, there exist three variants of Theorem 2.15 in the literature, which we briefly discuss next.

1. Gretz, Katoen, and McIver [GKM14] prove a variant of Theorem 2.15 but restrict to *bounded* expectations.

2. Batz et. al [5] propose a variant of Theorem 2.15 for an extension of dwp to heap-manipulating probabilistic programs involving *unbounded* nondeterminism. Their proof relies on an auxiliary result [3, Lemma B.7] stating — phrased in our terminology — that

   $$\mathsf{dop}[\![C]\!](X) \;=\; \lambda\sigma.\,\mathsf{MinER}(\mathcal{O},(C,\sigma) \models \Diamond\mathsf{rew}_X)$$

   is indeed obtained from the least fixpoint of ${}^{\min}_{\mathcal{O}}\Phi_{\mathsf{rew}_X}$. The induction proof of [3, Lemma B.7] is, however, flawed (the considered relation is generally not well-founded so that well-founded induction does not apply). It is for this reason that we have (i) developed least fixpoint characterizations of possibly unbounded or ∞-valued minimal expected rewards in MDPs in Section 2.2.3 and (ii) recapped the proof of Theorem 2.15 in this thesis. While our results from Section 2.2.3 do not apply to programs involving unbounded nondeterminsim, they are a first step towards fixing this issue.

3. Batz et. al [17, Theorem 4.6] propose a variant of Theorem 2.15 for an extension of awp to heap-manipulating probabilistic programs involving *unbounded* nondeterminism and `tick` statements for modelling expected runtimes. Their proof invokes [Bla67, Theorem 2] for concluding — phrased in our terminology — that

   $$\mathsf{aop}[\![C]\!](X) \;=\; \lambda\sigma.\,\mathsf{MaxER}(\mathcal{O},(C,\sigma) \models \Diamond\mathsf{rew}_X)$$

   is indeed obtained from the least fixpoint of ${}^{\max}_{\mathcal{O}}\Phi_{\mathsf{rew}_X}$. [Bla67, Theorem 2] restricts, however, to *bounded* rewards functions and does, therefore, not apply. It is for this reason that we have (i) developed least fixpoint characterizations of possibly unbounded or ∞-valued minimal expected rewards in MDPs in Section 2.2.3 and (ii) recapped the proof of Theorem 2.15 in this thesis. While our results from Section 2.2.3 do not apply to programs involving unbounded nondeterminsim, they are a first step towards fixing this issue.

3 Relatively Complete Verification

This chapter is based on our prior publications [12; 8].

Assumptions. In this chapter, we assume that all programs $C \in \mathsf{pGCL}$ considered throughout are *fully probabilistic*, i.e., C does not contain nondeterministic choices. Extending the presented results to nondeterministic programs is left as a promising direction for future work. See Section 3.6 for details.

3.1 Motivation and Problem Statement

This chapter treats a fundamental aspect regarding the automated deductive verification of probabilistic programs based on weakest preexpectation reasoning:

> We present an *expressive formal language* Exp *of expectations*, enabling the *relatively complete* verification of probabilistic programs and providing a foundation for the development of *automated deductive verifiers*.

To motivate and understand this aspect, let us consider what is meant by *extensional vs. intensional* approaches of deductive program verification [NN07].

Extensional Probabilistic Program Verification. In Section 2.4, we have presented the weakest preexpectation calculus in an *extensional* setting. That is, we treat expectations as *purely mathematical entities*: We admit *arbitrary* expectations X of type $\mathbb{E}$ and do not care about, e.g., finite representability of X. From a mathematical perspective, there are good reasons to work in the extensional setting when developing program calculi such as the weakest preexpectations calculus. Desirable properties such as well-definedness of $\mathsf{wp}[\![C]\!] \colon \mathbb{E} \to \mathbb{E}$ as defined in Table 2.1 on page 60 follow from concise fixpoint-theoretic arguments. For instance, the fact that for *every, arbitrarily complex* loop $C = \mathtt{while}(B)\{C'\}$ and *all* postexpectations $X \in \mathbb{E}$, the weakest preexpectation

$$\mathsf{wp}[\![C]\!](X) \;=\; \mathsf{lfp}\; {}^{\mathsf{wp}}_{C}\Phi_X$$

denotes a well-defined expectation is a consequence of Tarski's Fixpoint Theorem 2.1 on page 20. As another example, the soundness of invariant-based reasoning for upper bounds on weakest preexpectations of loops (cf. Theorem 2.12 on page 66) is an instance of Park induction (cf. Lemma 2.4 on page 28).

Intensional Probabilistic Program Verification. Automated deductive verifiers based on the weakest preexpectation calculus ultimately rely on computing or approximating $\mathsf{wp}[\![C]\!](X)$ for $X \in \mathbb{E}$ of interest. This raises the question:

How can one syntactically represent X and $\mathsf{wp}[\![C]\!](X)$?

Recall that $\mathsf{wp}[\![C]\!](X)$ is defined by induction on the structure of C, i.e., in a compositional and program-syntax based manner. We argued in Example 2.22 on page 62 that this yields determining $\mathsf{wp}[\![C]\!](X)$ for loop-free C to consist *mostly* of syntactic reasoning. Quantitative loop invariants, in turn, can reduce reasoning about loopy programs to reasoning about *loop-free* programs (cf. Example 2.24 on page 66). As long as we treat the postexpectation X as a purely mathematical entity as is done in the extensional setting, we can, however, not speak of *entirely* syntactic reasoning. For that, we need to have some syntactic representation of the postexpectation X at hand. With the notational conventions from Section 2.4.2, we ensured this to be the case for the examples we have considered so far. By, e.g., identifying the variable x with the expectation $\lambda\sigma.\,\sigma(x)$, determining

$$\mathsf{wp}[\![x := y+z]\!](x) \quad = \quad y+z$$

can indeed be considered a syntactic operation. Since the set $\mathbb{E}$ of expectations is *uncountably* infinite, there is no hope to finitely represent *all* expectations.

Therefore, in the *intensional setting*, we do not admit arbitrary expectations $X \in \mathbb{E}$. Rather, the goal is to provide a *computable — and thus countable* — formal language Exp of *syntactic expectations* together with a semantic function

$$[\![\cdot]\!] : \mathsf{Exp} \to \mathbb{E}\,,$$

which associates to each syntactic expectation $f \in \mathsf{Exp}$ an expectation $[\![f]\!] \in \mathbb{E}$. Let us say that an expectation X is *expressible* in Exp, if there is some $f \in \mathsf{Exp}$ with $[\![f]\!] = X$. We immediately encounter the question: Which expectations should be expressible in Exp? We identify the following requirements:

1. We can express enough expectations in Exp to specify interesting expected outcomes (cf. Section 2.4.1). In particular:

a) The constant-1-expectation $\lambda\sigma.1$ should be expressible in Exp to reason about termination probabilities, given by $\mathrm{wp}[\![C]\!](\lambda\sigma.1)$.

b) Indicator functions $[P]$ for a sufficiently large class of predicates[1] P should be expressible in Exp to reason about the probabilities to reach a final state satisfying P, given by $\mathrm{wp}[\![C]\!]([P])$.

c) For every $x \in \mathrm{Vars}$, the expectation $\lambda\sigma.\sigma(x)$ should be expressible in Exp to reason about expected final values of program variables, given by $\mathrm{wp}[\![C]\!](\lambda\sigma.\sigma(x))$.

2. For every program C, expectations expressible in Exp should be closed under $\mathrm{wp}[\![C]\!]$. This property is called *expressiveness*. Formally:

Definition 3.1 (Expressiveness [Coo78; Win93]).
Let Exp be a formal language of syntactic expectations equipped with a semantic function $[\![\cdot]\!] : \mathrm{Exp} \to \mathbb{E}$. We say that Exp is *expressive*, if

$$\text{for all } C \in \mathrm{pGCL}, f \in \mathrm{Exp}\colon \text{exists } g \in \mathrm{Exp}\colon \quad \mathrm{wp}[\![C]\!]([\![f]\!]) \;=\; [\![g]\!]\ .$$

Challenges. If expressiveness was our only requirement, there would be a simple solution to our problem: Let $\mathrm{Exp}_0 = \{0\}$ be the singleton with $[\![0]\!] = \lambda\sigma.0$, i.e., Exp_0 contains a single syntactic expectation expressing the constant-0-expectation. Then Exp_0 is expressive by *strictness* of wp, i.e.,

$$\text{for all } C \in \mathrm{pGCL}\colon \quad \mathrm{wp}[\![C]\!]([\![0]\!]) \;=\; [\![0]\!]\ . \qquad \text{([Kam19, Theorem 4.14.A])}$$

Unfortunately, Exp_0 is useless for specifying non-trivial expected outcomes.

If we require *both* that Exp is expressive *and* that Exp enables the specification of interesting expected outcomes, the situation becomes much more challenging. To get a first intuition, consider the following program C given by

```
$x := 1$;
while($x > 0$){
    {$x := x - 1$} [$1/2$] {$x := x + 2$}
}.
```

[1]The language presented in the subsequent sections subsumes first-order arithmetic $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ over $\mathbb{Q}_{\geq 0}$ (cf. Section 3.4.2.1), thus admitting any predicate P definable in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$.

```
$x_1$ := geo(1/4); $x_2$ := geo(1/4);
{$x$ := $x_1 + x_2$} [5/9] {$x$ := $x_1 + x_2 + 1$};
$t$ := 1;
repeat 3 times{
    $z$ := 1; $y_1$ := 0; $y_2$ := 0;
    while($z \leq 2 \cdot x$){
        {$y_1$ := $y_1 + 1$} [1/2] {$y_2$ := $y_2 + 1$};
        $z$ := $z + 1$
    }
    if ($y_1 \neq y_2$) {$t$ := 0} else {skip}
}
```

Figure 3.1: A pGCL program adapted from [FPS11]. Here x_i := geo(1/4) is a shorthand for x_i := 0; z := 1; while($z = 1$){{x_i := $x_i + 1$} [1/4] {z := 0}}.

For this program C, we have [OKKM16]

$$\mathsf{wp}[\![C]\!](\lambda\sigma.1) \;=\; \lambda\sigma.\frac{\sqrt{5}-1}{2}\,,$$

i.e., if the constant-1-expectation is expressible in Exp, it already follows that *irrational*, yet algebraic, numbers such as $\sqrt{5}-1/2$ must also be expressible in Exp.

So do algebraic reals suffice for an expressive language of expectations? The answer is no. Consider the program C shown in Figure 3.1. We have

$$\mathsf{wp}[\![C]\!]([t=1]) \;=\; \lambda\sigma.\frac{1}{\pi}\,,$$

i.e., if $[t=1]$ is expressible in Exp, which is a reasonable requirement, then we need to be able to express *transcendental* reals in Exp *even though all ingredients of C are rational-valued*. We conclude that admitting even the most simple postexpectations in Exp requires the ability to express non-trivial quantities in Exp. Yet, we will see in the subsequent sections that there is a (syntactically) simple formal language Exp meeting our requirements. The proof that this Exp is indeed expressive is, however, rather involved. We will outline that parts of our proof strategy are inspired by existing expressiveness proofs for Dijkstra's weakest pre*conditions* for *non-probabilistic* programs [Win93; LSS84]. One of the key differences between this and our setting is that we have to reason about possibly transcendental reals. This fact requires us to introduce additional technical machinery, see Sections 3.4.3 and 3.4.4 for details.

Application: A Foundation for Automated Deductive Verifiers. We comment on an application of the results presented in this chapter. In [18], we have presented the automated deductive verifier Caesar based on the weakest preexpectation calculus. Since the results from [18] will not be included in this thesis, suffice it to say here that one of Caesar's main applications is to compute or approximate weakest preexpectations of possibly loopy probabilistic programs in a semi-automatic fashion, thereby enabling the semi-automatic verification of bounds on expected outcomes. When developing Caesar, we thus naturally encountered the question of how to syntactically represent a sufficiently large class of expectations. The language Exp which we present in the subsequent section provided us with a solid foundation: Caesar's quantitative assertion language HeyLo subsumes Exp, thus taking benefit from its expressive power and the fact that the results presented here enable the *relatively complete verification* of pGCL programs. Section 3.5 includes a detailed discussion of these aspects.

Chapter Outline. The remainder of this chapter is structured as follows: In Section 3.2 we discuss the syntax and semantics of our language Exp. In Section 3.3, we prove expressiveness of Exp for *loop-free* programs. Proving expressiveness for *all*, *possibly loopy* programs is significantly more involved and treated in Section 3.4, where we provide a proof outline in Section 3.4.1. In Section 3.5, we then discuss further consequences of the results presented in this chapter. Finally, in Section 3.6, we discuss future and related work.

3.2 The Language Exp of Syntactic Expectations

We describe the syntax and semantics of a language Exp of syntactic expectations, which we prove to be expressive in the subsequent sections. Recall that in the definition of pGCL (Definition 2.16 on page 45), we did not fix a syntax for arithmetic or Boolean expressions. To obtain a computable set of pGCL programs and, in particular, an expressive and computable language of syntactic expectations, we now *do* fix a natural syntax for both arithmetic and Boolean expressions occurring in pGCL programs. These expressions will moreover be the basic building blocks (or *atoms*) of our language of syntactic expectations.

3.2.1 Syntax of Arithmetic Expressions

We first describe a *syntax of arithmetic expressions*, which form *precisely the right-hand-sides of* *assignments* that we allow in pGCL programs.

Definition 3.2 (Arithmetic Expressions).
Arithmetic expressions, which we denote by a, b, c, and variations thereof, in the set AExpr adhere to the grammar

$$
\begin{aligned}
a \quad \longrightarrow \quad & q \in \mathbb{Q}_{\geq 0} && \text{(non-negative rationals)} \\
\mid\ & x \in \mathsf{Vars} && \text{($\mathbb{Q}_{\geq 0}$-valued variables)} \\
\mid\ & a + a && \text{(addition)} \\
\mid\ & a \cdot a\,, && \text{(multiplication)} \\
\mid\ & a \dot{-} a\,. && \text{(subtraction truncated at 0 (“monus”))}
\end{aligned}
$$

We agree on the usual order of precedence for arithmetic operations and use parentheses to resolve ambiguities.

3.2.2 Syntax of Boolean Expressions

We next describe a *syntax for Boolean expressions* over AExpr, which form *precisely the guards* for conditional choices and while loops in pGCL.

Definition 3.3 (Boolean Expressions).
Boolean expressions, which we denote by φ, ψ, ξ, and variations thereof, in the set BExpr adhere to the grammar

$$\begin{array}{lll} \varphi & \longrightarrow & a < a & \text{(strict inequality of arithmetic expressions)} \\ & & \mid\ \varphi \wedge \varphi & \text{(conjunction)} \\ & & \mid\ \neg\varphi\,. & \text{(negation)} \end{array}$$

As usual, the following Boolean expressions are syntactic sugar:

$$\mathsf{false}\,, \qquad \mathsf{true}\,, \qquad \varphi \vee \psi\,, \qquad \varphi \implies \psi\,, \qquad a = b\,, \qquad a \leq b$$

We agree on the usual order of precedence for the Boolean connectives, i.e.,

$$\neg \text{ binds stronger than } \wedge \text{ binds stronger than } \vee \text{ binds stronger than } \implies\,,$$

and we use parenthesis to resolve ambiguities.

3.2.3 Syntax of Expectations

We now describe the syntax of our formal language Exp of *syntactic expectations*.

Definition 3.4 (Syntactic Expectations).
Syntactic expectations, which we denote by f, g, h, and variations thereof, in the set Exp adhere to the grammar

$$\begin{array}{lll} f & \longrightarrow & a & \text{(arithmetic expressions)} \\ & & \mid\ [\varphi] & \text{(Boolean expressions)} \\ & & \mid\ f + f & \text{(addition)} \\ & & \mid\ f \cdot f & \text{(multiplication)} \\ & & \mid\ \reflectbox{\textsf{S}}\, v\colon f & \text{(supremum over } v) \\ & & \mid\ \reflectbox{\textsf{J}}\, v\colon f\,, & \text{(infimum over } v) \end{array}$$

where $a \in \mathsf{AExpr}$, $\varphi \in \mathsf{BExpr}$, and $v \in \mathsf{Vars}$.

$\reflectbox{\textsf{S}}$ is the *supremum quantifier* and $\reflectbox{\textsf{J}}$ is the *infimum quantifier*. They are the quantitative analogs of the $\exists$ and $\forall$ quantifiers of first-order predicate logic. Analogously to predicate logic, variable v is *bound* in $\reflectbox{\textsf{S}} v\colon f$ or $\reflectbox{\textsf{J}} v\colon f$. Bound variables are typically denoted by v, u, w, and variations thereof. To reduce the number of parenthesis resolving ambiguities, we assume that

$$\cdot \text{ binds stronger than } +;\ + \text{ binds stronger than } \reflectbox{\textsf{S}}, \reflectbox{\textsf{J}}\ .$$

Except for the quantifiers, the above constructs are largely self-explanatory since they are chosen reminiscent of the operations for (semantic) expectations from Section 2.4.2. Let us therefore get a first intuition on our choice of Exp.

Arithmetic Expressions. These form the base case and it is immediate that they are needed for an expressive language. Assume, for instance, that we want to know the "expected" (in fact: certain) value of variable x — itself an arithmetic expression by definition — after executing $x := a$. Then this is given by[2] $\mathsf{wp}[\![x := a]\!](x) = a$ — again an arithmetic expression. As a could have been *any* arithmetic expression, we at least need all arithmetic expressions in an expressive language of expectations.

Boolean Expressions, Addition, and Multiplication. These constructs are used for expressing the weakest preexpectations of conditional choices, probabilistic choices, and loops. As we have, for instance,

$$\mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C_1\}\,\mathtt{else}\,\{C_2\}]\!](f) \;=\; [\varphi]\cdot\mathsf{wp}[\![C_1]\!](f) + [\neg\varphi]\cdot\mathsf{wp}[\![C_2]\!](f)\ ,$$

and

$$\mathsf{wp}[\![\{C_1\}\,[p]\,\{C_2\}]\!](f) \;=\; p\cdot\mathsf{wp}[\![C_1]\!](f) + (1-p)\cdot\mathsf{wp}[\![C_2]\!](f)\ ,$$

these constructs are convenient for being expressive.

Suprema and Infima. The supremum and infimum quantification $\reflectbox{\textsf{S}} v\colon f$ and $\reflectbox{\textsf{J}} v\colon f$ take over the role of the $\exists v\colon \mathcal{P}$ and $\forall v\colon \mathcal{P}$ quantification of first-order predicate logic. $\reflectbox{\textsf{S}}$ and $\reflectbox{\textsf{J}}$ are necessary for Exp to be expressive in the same way as the $\exists$ and $\forall$ quantifiers are necessary for first-order arithmetic to be expressive for weakest preconditions of non-probabilistic programs [Win93].

[2]Strictly speaking, we must not write $\mathsf{wp}[\![C]\!](f)$ since f is a syntactic object rather than an expectation of type $\mathbb{E}$. In the following, we do so nonetheless to gain some intuition on why include the different constructs in the language Exp. The formal semantics of syntactic expectations is given in Section 3.2.4.

f	$\mathbf{FV}(f)$
a	$\mathsf{FV}(a)$
$[\varphi]$	$\mathsf{FV}(\varphi)$
$g+h$	$\mathsf{FV}(g) \cup \mathsf{FV}(h)$
$g \cdot h$	$\mathsf{FV}(g) \cup \mathsf{FV}(h)$
$\reflectbox{\textsf{S}} v \colon g,\ \reflectbox{\textsf{J}} v \colon g$	$\mathsf{FV}(g) \setminus \{v\}$

Table 3.1: Inductive definition of $\mathsf{FV}(f) \subset \mathsf{Vars}$.

Finally, we introduce the notion of free variables of syntactic expectations. Given $a \in \mathsf{AExpr}$, we denote by $\mathsf{FV}(a) \subset \mathsf{Vars}$ the finite set of (necessarily free) variables in a. For $\varphi \in \mathsf{BExpr}$, we define $\mathsf{FV}(\varphi)$ analogously.

Definition 3.5 (Free Variables of Syntactic Expectations).
Let $f \in \mathsf{Exp}$. The finite *set of free variables in f*

$$\mathsf{FV}(f) \ \subset \ \mathsf{Vars}$$

is defined by induction on f in Table 3.1.

Thus, $\mathsf{FV}(f)$ contains all variables occurring at least once in f without being in the scope of a $\reflectbox{\textsf{S}}$ or $\reflectbox{\textsf{J}}$ quantifier. When defining specific syntactic expectations, we will often write $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are pairwise distinct variables, to indicate that *at most* the variables $x_1, \ldots, x_n$ occur freely in f, i.e.,

$$\mathsf{FV}(f) \ \subseteq \ \{x_1, \ldots, x_n\} \,.$$

Remark 3.1 (Comparing Exp to the Language Presented in [12]).
The language Exp presented here differs from the language presented in [12]. The language from [12] does *not allow* for products $f \cdot g$ of two *arbitrary* syntactic expectations f and g. This was because our proof strategy[a] for expressiveness required that for every $f \in \mathsf{Exp}$, we can compute an *equivalent* syntactic expectation f' in *prenex normalform*. This turned out to be challenging when allowing for such general products (cf. [12, Section 4.6]). In the course of writing this thesis, it turned out that prenex normal forms of

a	$[\![a]\!](\sigma)$	φ	$\sigma \in [\![\varphi]\!]$ iff
$q \in \mathbb{Q}_{\geq 0}$	q	$b < c$	$[\![b]\!](\sigma) < [\![c]\!](\sigma)$
$x \in \mathsf{Vars}$	$\sigma(x)$	$\psi \wedge \xi$	$\sigma \in [\![\psi]\!]$ and $\sigma \in [\![\xi]\!]$
$b + c$	$[\![b]\!](\sigma) + [\![c]\!](\sigma)$	$\neg\psi$	$\sigma \notin [\![\psi]\!]$
$b \cdot c$	$[\![b]\!](\sigma) \cdot [\![c]\!](\sigma)$		
$b \dot{-} c$	$\max([\![b]\!](\sigma) - [\![c]\!](\sigma), 0)$		

Table 3.2: Inductive definitions of the semantics of $a \in \mathsf{AExpr}$ and $\varphi \in \mathsf{BExpr}$.

syntactic expectations can actually be avoided, leading to a slightly simpler and more natural language of syntactic expectations which *does* allow for arbitrary products. The expressive power of the language Exp presented here and our original language from [12] does, however, coincide since general products are expressible in our original language (cf. [12, Theorem 9.4]).

[a]More precisely, this was required for a construction similar to Theorem 3.10 on page 105.

3.2.4 Semantics of Expressions and Expectations

The semantics $[\![f]\!]$ of a syntactic expectation is of type $\mathbb{E}$, i.e., $[\![f]\!]$ is an expectation. We first define the semantics of arithmetic and Boolean expressions.

Definition 3.6 (Semantics of Arithmetic and Boolean Expressions).
Let $a \in \mathsf{AExpr}$ and $\varphi \in \mathsf{BExpr}$.

1. The *semantics of a* is the function

$$[\![a]\!] \colon \mathsf{States} \to \mathbb{Q}_{\geq 0}$$

defined by induction on the structure of a in Table 3.2 (left).

2. The *semantics of φ* is the predicate

$$[\![\varphi]\!] \in \mathcal{P}(\mathsf{States})$$

defined by induction on the structure of φ in Table 3.2 (right). We usually write $\sigma \models \varphi$ instead of $\sigma \in [\![\varphi]\!]$.

f	$\llbracket f \rrbracket(\sigma)$
a	$\llbracket a \rrbracket(\sigma)$
$[\varphi]$	$\begin{cases} 1 & \text{if } \sigma \models \varphi \\ 0 & \text{if } \sigma \not\models \varphi \end{cases}$
$g + h$	$\llbracket g \rrbracket(\sigma) + \llbracket h \rrbracket(\sigma)$
$g \cdot h$	$\llbracket g \rrbracket(\sigma) \cdot \llbracket h \rrbracket(\sigma)$
$\reflectbox{\textsf{S}} v \colon g$	$\sup \{ \llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0} \}$
$\rotatebox{180}{\textsf{L}} v \colon g$	$\inf \{ \llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0} \}$

Table 3.3: Inductive definition of the semantics of $f \in \text{Exp}$. Suprema and infima are taken w.r.t. the complete lattice $(\mathbb{R}^{\infty}_{\geq 0}, \leq)$.

Definition 3.7 (Semantics of Syntactic Expectations).
Let $f \in \text{Exp}$. The *semantics of f* is the expectation

$$\llbracket f \rrbracket \in \mathbb{E}$$

defined by induction on the structure of f in Table 3.3.

Recall that we let $0 \cdot \infty = \infty \cdot 0 = 0$. Given syntactic expectations f and g, we write $f = g$ if they *coincide syntactically*. We say that f and g are *(semantically) equivalent*, denoted $f \equiv g$, if they denote the same expectation, i.e.,

$$f \equiv g \qquad \text{iff} \qquad \llbracket f \rrbracket = \llbracket g \rrbracket .$$

Most of the rules in Table 3.3 are self-explanatory. The most involved rules are the ones for quantifiers. $\reflectbox{\textsf{S}}$ and $\rotatebox{180}{\textsf{L}}$ are the quantitative analogues of the $\exists$ and $\forall$ quantifiers from first-order predicate logic [EFT94]. The $\reflectbox{\textsf{S}}$ quantifier maximizes a quantity just like the $\exists$ quantifier maximizes a truth value: The interpretation of the $\reflectbox{\textsf{S}} v \colon f$ quantification interprets f under all possible values of the variable v and then returns the supremum of all these values. Dually, $\rotatebox{180}{\textsf{L}}$ minimizes f under all possible values of v. Notice that $\llbracket f \rrbracket(\sigma)$ is a well-defined quantity in $\mathbb{R}^{\infty}_{\geq 0}$ for each $\sigma \in \text{States}$ since $(\mathbb{R}^{\infty}_{\geq 0}, \leq)$ is a complete lattice.

Even though all atomic syntactic expectations of the form $f = a$ or $f = [\varphi]$ evaluate to *rationals*, a syntactic expectation f involving $\reflectbox{S}$ or $\reflectbox{J}$ possibly evaluates to an *irrational* number. Let us consider some introductory examples. A more extensive discussion on expressible expectations is given in Section 3.5.

Example 3.1.

1. For
$$f \;=\; \reflectbox{S} v\colon\; [v \cdot v < y] \cdot v\;,$$
we have for every $\sigma \in \mathsf{States}$,
$$\begin{aligned} & [\![f]\!](\sigma) \\ =\;& \sup\{[\![[v \cdot v < y] \cdot v]\!](\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} \\ =\;& \sup\{q \mid q \in \mathbb{Q}_{\geq 0} \text{ and } q \cdot q < \sigma(y)\} \\ =\;& \sqrt{\sigma(y)}\;. \end{aligned}$$
Hence, if, e.g., $\sigma(y) = 2$, we have $[\![f]\!](\sigma) = \sqrt{2}$ — an irrational number.

2. For
$$f \;=\; \reflectbox{S} v\colon\; v\;,$$
we have $[\![f]\!](\sigma) = \infty$ for all $\sigma \in \mathsf{States}$.

3. Even though fractions like $1/x$ are not allowed by our syntax, we can express them using quantifiers. We have, e.g.,
$$\begin{aligned} & [\![\reflectbox{S} v\colon\; [v \cdot x = 1] \cdot v]\!](\sigma) \\ =\;& \{q \mid q \in \mathbb{Q}_{\geq 0} \text{ and } q \cdot \sigma(x) = 1\} \\ =\;& \begin{cases} 1/\sigma(x) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0\;. \end{cases} \end{aligned}$$
The role of the $\reflectbox{S} v$ quantifier is — so to speak — to select the rational number we are interested in. When constructing fractions this way, we never run into well-definedness issues arising from, e.g., dividing by 0. The above construction automatically defaults to 0 in case we "divide by 0". Any other default value $r \in \mathbb{Q}_{\geq 0}$ can be constructed via
$$[x = 0] \cdot r + [x > 0] \cdot (\reflectbox{S} v\colon\; [v \cdot x = 1] \cdot v)\;.$$

4. In Example 3.5 on page 110, we will define a syntactic expectation which evaluates to $\pi^2/6$ — a *transcendental* number.

3.2.5 Capture-Avoiding Substitutions of Variables

Given $f \in \mathsf{Exp}$, $a \in \mathsf{AExpr}$, and $x \in \mathsf{Vars}$, our goal is to define $f[x/a] \in \mathsf{Exp}$ — the syntactic expectation obtained from substituting x in f by a. A natural and, in fact, necessary requirement for such a substitution is that

$$\text{for all } \sigma \in \mathsf{States}\colon \quad [\![f[x/a]]\!](\sigma) \;=\; [\![f]\!](\sigma[x \mapsto [\![a]\!](\sigma)])\,,$$

i.e., syntactically substituting x in f by a and then evaluating the so-obtained syntactic expectation in some state σ yields the same quantity as evaluating f in the state obtained from σ by updating the value of x by $[\![a]\!](\sigma)$.

An <u>Unsound</u> Approach. If we were to naively define

ϟ "obtain $f[x/a]$ by simply replacing every free occurrence of x by a" ϟ

then the above requirement is generally not met. The problem arises if a contains variables that are substituted into the scope of a quantifier[3]. Consider, for instance, the syntactic expectation $f = \mathsf{S}y\colon x$ and the arithmetic expression $a = y$. With the above naive definition of syntactic substitution, we get $f[x/y] = \mathsf{S}y\colon y$. However, we have for every $\sigma \in \mathsf{States}$,

$$\begin{aligned}
& [\![f[x/y]]\!](\sigma) \\
=\;& [\![\mathsf{S}y\colon y]\!](\sigma) \\
=\;& \infty \\
\neq\;& \sigma(y) \\
=\;& \sup\{\sigma(y) \mid q \in \mathbb{Q}_{\geq 0}\} \\
=\;& \sup\{[\![x]\!](\sigma[x \mapsto [\![y]\!](\sigma)][y \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} \\
=\;& [\![\mathsf{S}y\colon x]\!](\sigma[x \mapsto [\![y]\!](\sigma)])\,.
\end{aligned}$$

We would expect that $[\![f[x/y]]\!](\sigma) = \sigma(y)$ but since variable y gets bound by the S quantifier when substituting x by y, the value of y under σ becomes irrelevant when evaluating $\mathsf{S}y\colon y$ so that $[\![\mathsf{S}y\colon y]\!]$ is the constant expectation ∞.

[3] In the field of lambda calculi, this situation is called *variable capture* [Lea00].

A Sound Approach. We can fix the above issue by renaming bound variables by fresh variables. It is obvious that renaming a bound variable in some $f \in \mathsf{Exp}$ by some fresh variable not occurring in f never changes the semantics[4] of f. For instance, reconsidering the situation from above, if we first rename the bound variable y in f by the fresh variable y' (which does not occur in f or a), we get $f' = \mathcal{S} y' \colon x$ satisfying $f' \equiv f$. If we now substitute every free occurrence of x in f' by $a = y$, we obtain the desired result since we have for every $\sigma \in \mathsf{States}$ that

$$\begin{aligned} & [\![f'[x/y]]\!](\sigma) \\ = \; & [\![\mathcal{S} y' \colon y]\!](\sigma) \\ = \; & \sigma(y) \\ = \; & [\![f]\!](\sigma[x \mapsto [\![y]\!](\sigma)]) \,. \end{aligned}$$

Renaming bound variables to avoid variable capture when substituting variables by expressions is called *capture-avoiding substitution*, which we formalize in the following. Given two arithmetic expressions a, b and $x \in \mathsf{Vars}$, denote by $a[x/b]$ the arithmetic expression obtained from substituting every occurrence of x in a by b, which is defined by induction on a in the obvious way. For Boolean expressions φ, we analogously define $\varphi[x/a]$ as the Boolean expression obtained from substituting every occurrence of x in φ by a. Neither arithmetic nor Boolean expressions contain quantifiers, so variable capture is not an issue here.

Definition 3.8 (Capture-Avoiding Substitutions of Variables).
Let $f \in \mathsf{Exp}$, $a \in \mathsf{AExpr}$, and let $x \in \mathsf{Vars}$. The syntactic expectation

$$f[x/a] \in \mathsf{Exp}$$

obtained from substituting every free occurrence of x in f by a in a capture-avoiding manner is defined by induction on f in Table 3.4.

Capture-avoiding substitutions indeed meet our soundness requirement.

Lemma 3.1 (Coincidence of Syntactic and Semantic Substitutions).
Let $f \in \mathsf{Exp}$, $a \in \mathsf{AExpr}$, and let $x \in \mathsf{Vars}$. We have

$$\text{for all } \sigma \in \mathsf{States}\colon \quad [\![f[x/a]]\!](\sigma) \;=\; [\![f]\!](\sigma[x \mapsto [\![a]\!](\sigma)]) \,.$$

Finally, we generalize the above concept to the *simultaneous* substitution of *multiple free variables*. Recall that we often define specific syntactic expectations f by writing $f(x_1, \ldots, x_n)$ to indicate that at most the pairwise distinct variables

[4]this is also known as *α-conversion*.

f	$f\,[x/a]$
b	$b\,[x/a]$
$[\varphi]$	$[\varphi\,[x/a]]$
$g + h$	$g\,[x/a] + h\,[x/a]$
$g \cdot h$	$g\,[x/a] \cdot h\,[x/a]$
$\reflectbox{$\mathsf{S}$}\, v\colon g$	$\reflectbox{$\mathsf{S}$}\, v'\colon\ (g\,[v/v'])\,[x/a]$
$\rotatebox{-30}{\reflectbox{J}}\, v\colon g$	$\rotatebox{-30}{\reflectbox{J}}\, v'\colon\ (g\,[v/v'])\,[x/a]$

Table 3.4: Inductive definition of capture-avoiding substitutions. $v' \in \mathsf{Vars}$ is fresh, i.e, distinct from both x and v and we have $v' \not\in \mathsf{FV}(g) \cup \mathsf{FV}(a)$.

$x_1,\ldots,x_n$ occur freely in f. Given $a_1,\ldots,a_n \in \mathsf{AExpr}$, we then write $f(a_1,\ldots,a_n)$ to denote the syntactic expectation obtained from f by *simultaneously* substituting $x_1,\ldots,x_n$ in f by $a_1,\ldots,a_n$ in a capture-avoiding manner so that we have

$$\text{for all } \sigma \in \mathsf{States}\colon\ [\![f(a_1,\ldots,a_n)]\!](\sigma) = [\![f]\!](\sigma[x_1 \mapsto [\![a_1]\!](\sigma),\ldots,x_n \mapsto [\![a_n]\!](\sigma)])\,,$$

where

$$\sigma[x_1 \mapsto [\![a_1]\!](\sigma),\ldots,x_n \mapsto [\![a_n]\!](\sigma)] \;=\; \lambda x.\begin{cases} [\![a_1]\!](\sigma) & \text{if } x = x_1 \\ \quad\vdots & \\ [\![a_n]\!](\sigma) & \text{if } x = x_n \\ \sigma(x) & \text{otherwise}\,. \end{cases}$$

Example 3.2.
Let us define the syntactic expectation $\mathsf{fraction}(x_1, x_2)$ by

$$\mathsf{fraction}(x_1, x_2) \;=\; \reflectbox{S}\, v\colon\ [v \cdot x_2 = x_1] \cdot v\,.$$

We then denote by, e.g., $\mathsf{fraction}(z, z+2)$ the syntactic expectation

$$\mathsf{fraction}(z, z+2) \;=\; \reflectbox{S}\, v'\colon\ [v' \cdot (z+2) = z] \cdot v'\,.$$

3.3 Expressiveness for Loop-Free Programs

Before we deal with loops, we now show that our language Exp of syntactic expectations is *expressive for all* <u>*loop-free*</u> pGCL *programs*. Proving expressiveness for loops is *way more involved* and will be addressed in the subsequent sections.

Theorem 3.2 (Expressiveness of Exp for Loop-Free Programs).
Exp is expressive (cf. Definition 3.1) for the loop-free fragment of pGCL, i.e.,

$$\text{for all loop-free } C \in \mathsf{pGCL}, f \in \mathsf{Exp}\colon \text{ exists } g \in \mathsf{Exp}\colon \mathsf{wp}[\![C]\!]([\![f]\!]) = [\![g]\!] ~.$$

Moreover, g is effectively constructible.

Proof. By induction on the structure of C. Let $f \in \mathsf{Exp}$.

Effectless Program $C = \mathtt{skip}$. We have

$$\mathsf{wp}[\![\mathtt{skip}]\!]([\![f]\!]) ~=~ [\![f]\!] \qquad \text{(Table 2.1)}$$

and $f \in \mathsf{Exp}$ by assumption.

Assignment $C = x := a$. We have

$$\begin{aligned} &\mathsf{wp}[\![x := a]\!]([\![f]\!]) \\ &= [\![f]\!][x/a] && \text{(Table 2.1)} \\ &= [\![f[x/a]]\!] && \text{(Lemma 3.1)} \end{aligned}$$

and $f[x/a] \in \mathsf{Exp}$ by assumption and definition of Exp.

Induction Hypothesis. For arbitrary, but fixed, loop-free $C_1, C_2 \in \mathsf{pGCL}$ and all $f_1, f_2 \in \mathsf{Exp}$ there are effectively constructible $g_1, g_2 \in \mathsf{Exp}$ such that

$$\mathsf{wp}[\![C_1]\!]([\![f_1]\!]) ~=~ [\![g_1]\!] \quad \text{and} \quad \mathsf{wp}[\![C_2]\!]([\![f_2]\!]) ~=~ [\![g_2]\!] ~.$$

Sequential Composition $C = C_1 \,;\, C_2$. We have

$$\begin{aligned} &\mathsf{wp}[\![C_1 \,;\, C_2]\!]([\![f]\!]) \\ =~ &\mathsf{wp}[\![C_1]\!](\mathsf{wp}[\![C_2]\!]([\![f]\!])) && \text{(Table 2.1)} \\ =~ &\mathsf{wp}[\![C_1]\!]([\![g_2]\!]) && (g_2 \in \mathsf{Exp} \text{ with } [\![g_2]\!] = \mathsf{wp}[\![C_2]\!]([\![f]\!]) \text{ exists by I.H.}) \end{aligned}$$

$$= \llbracket g_1 \rrbracket \,. \qquad (g_1 \in \mathsf{Exp} \text{ with } \llbracket g_1 \rrbracket = \mathsf{wp}\llbracket C_1 \rrbracket (\llbracket g_2 \rrbracket) \text{ exists by I.H.})$$

Probabilistic Choice $\{C_1\}\,[p]\,\{C_2\}$. We have

$$\begin{aligned}
& \mathsf{wp}\llbracket \{C_1\}\,[p]\,\{C_2\} \rrbracket (\llbracket f \rrbracket) \\
={}& p \cdot \mathsf{wp}\llbracket C_1 \rrbracket (\llbracket f \rrbracket) + (1-p) \cdot \mathsf{wp}\llbracket C_2 \rrbracket (\llbracket f \rrbracket) && \text{(Table 2.1)} \\
={}& p \cdot \llbracket g_1 \rrbracket + (1-p) \cdot \llbracket g_2 \rrbracket && \text{(suitable } g_1, g_2 \in \mathsf{Exp} \text{ exist by I.H.)} \\
={}& \llbracket p \cdot g_1 + (1-p) \cdot g_2 \rrbracket \,, && \text{(Table 3.3)}
\end{aligned}$$

and $p \cdot g_1 + (1-p) \cdot g_2 \in \mathsf{Exp}$, where we treat $(1-p)$ as an atomic expression since p is a constant rational probability by Definition 2.16 on page 45.

Conditional Choice $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$. We have

$$\begin{aligned}
& \mathsf{wp}\llbracket \texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\} \rrbracket (\llbracket f \rrbracket) \\
={}& [\llbracket \varphi \rrbracket] \cdot \mathsf{wp}\llbracket C_1 \rrbracket (\llbracket f \rrbracket) + [\llbracket \neg\varphi \rrbracket] \cdot \mathsf{wp}\llbracket C_2 \rrbracket (\llbracket f \rrbracket) && \text{(Table 2.1)} \\
={}& [\llbracket \varphi \rrbracket] \cdot \llbracket g_1 \rrbracket + [\llbracket \neg\varphi \rrbracket] \cdot \llbracket g_2 \rrbracket && \text{(suitable } g_1, g_2 \in \mathsf{Exp} \text{ exist by I.H.)} \\
={}& \llbracket [\varphi] \cdot g_1 + [\neg\varphi] \cdot g_2 \rrbracket && \text{(Table 3.3)}
\end{aligned}$$

and $[\varphi] \cdot g_1 + [\neg\varphi] \cdot g_2 \in \mathsf{Exp}$. ■

3.4 Expressiveness for Loopy Programs

3.4.1 Overview

Before we get to the technical details, we outline the main challenges — and the steps we took to address them — of proving expressiveness of our language Exp of syntactic expectations for pGCL programs including loops; the technical details of the involved encodings and auxiliary results are considered throughout Sections 3.4.2 – 3.4.5. This section is intended to support navigation through the individual components of the expressiveness proof; as such, we provide various references to follow-up sections.

3.4.1.1 Setup

As in the loop-free case considered in Section 3.3, we prove expressiveness of Exp for all pGCL programs (including loopy programs) by induction on the program structure; all cases except loops are completely analogous to the proof of Theorem 3.2 on page 90. Our remaining proof obligation thus boils down to proving that for every loop $C = \mathtt{while}(\varphi)\{C'\}$, we have

$$\text{for all } f \in \mathsf{Exp}\text{: exists } g \in \mathsf{Exp}\text{:} \quad \mathsf{wp}[\![\mathtt{while}(\varphi)\{C'\}]\!]([\![f]\!]) \;=\; [\![g]\!]\;, \qquad (\dagger)$$

where we already know by the I.H. that for the loop body C', we have

$$\text{for all } f' \in \mathsf{Exp}\text{: exists } g' \in \mathsf{Exp}\text{:} \quad \mathsf{wp}[\![C']\!]([\![f']\!]) \;=\; [\![g']\!]\;. \qquad (3.1)$$

A Simplification for this Overview. Just for this overview section, we assume that the set Vars of all variables is *finite* instead of countably infinite. This is a convenient simplification to avoid a few purely technical details such that we can focus on the actual ideas of the proof. We do *not* make this assumption in follow-up sections. Rather, our construction will ensure that only the finite set of "relevant" variables — those that appear in the program or the postexpectation under consideration — are taken into account.

3.4.1.2 Basic Idea: Exploiting the Kozen Duality

We first move to an alternative characterization of the weakest preexpectations of loops whose components are simpler to capture with syntactic expectations. In particular, we will be able to apply our induction hypothesis (3.1) to some of these components. Recall the Kozen duality from Corollary 2.16.3 on page 72:

$$\mathsf{wp}[\![C]\!](X) \;=\; \lambda\sigma_0.\; \sum_{\tau \in \mathsf{States}} [\![C]\!]^{\sigma_0}(\tau) \cdot X(\tau)\,,$$

where $[\![C]\!]^{\sigma_0}$ is the subdistribution of final states obtained from executing C on initial state σ_0. Adapting the above equality to our concrete case in which C is a loop and $X = [\![f]\!]$, we obtain

$$\begin{aligned} &\mathsf{wp}[\![\mathtt{while}(\varphi)\{C'\}]\!]([\![f]\!]) \\ =\;& \lambda\sigma_0.\; \sum_{\tau \in \mathsf{States}} [\![[\neg\varphi]\cdot f]\!](\tau) \cdot [\![\mathtt{while}(\varphi)\{C'\}]\!]^{\sigma_0}(\tau)\,, \end{aligned}$$

where we strengthened the postexpectation X to $[\neg\varphi]\cdot X$ to account for the fact that the loop guard φ is violated in every final state, see [Kam19, Corollary 4.6]. The main idea is — instead of viewing the whole distribution $[\![\mathtt{while}(\varphi)\{C'\}]\!]^{\sigma_0}$ in a single "big step" — to take an operational "small-step" perspective: we consider the intermediate states reached after each guarded loop iteration, which corresponds to executing the program

$$C_{\text{iter}} \;=\; \texttt{if}\,(\varphi)\,\{C'\}\,\texttt{else}\,\{\texttt{skip}\}\;.$$

We then sum over all terminating *execution paths* — finite sequences of states $\sigma_0, \ldots \sigma_k$ with initial state σ_0 and final state $\sigma_k = \tau$ — instead of a single final state τ. The probability of an execution path is then given by the product of the probability $[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1})$ of each intermediate step, i.e., the probability of reaching the state σ_{i+1} from the previous state σ_i by one guarded loop iteration:

$$\begin{aligned}&\mathsf{wp}[\![\texttt{while}\,(\varphi)\,\{C'\}]\!]\,([\![f]\!])\\ &=\; \lambda\sigma_0.\,\sup_{k\in\mathbb{N}} \sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}} [\![[\neg\varphi]\cdot f]\!](\sigma_k)\cdot\prod_{i=0}^{k-1}[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1})\,. \qquad (3.2)\end{aligned}$$

Notice that the above sum (without the sup) considers all execution paths of a fixed length k; we take the supremum over all natural numbers k to account for all terminating execution paths.

Next, we aim to apply the induction hypothesis (3.1) to the probability $[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1})$ of each step such that we can write it as a syntactic expectation. To this end, we need to characterize $[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1})$ in terms of weakest preexpectations. We employ a syntactic expectation $[\sigma]$ — called the *characteristic expectation* (inspired by the *characteristic assertions* from [Win93]) of state σ — capturing the values assigned to variables by σ:[5]

$$[\sigma] \;=\; \left[\bigwedge_{x\in\mathsf{Vars}} x = \sigma(x)\right] \in \mathsf{Exp}\,.$$

By the Kozen duality (Corollary 2.16.3 on page 72), the probability of reaching state σ_{i+1} from σ_i by *one* guarded loop iteration C_{iter} is then given by

$$[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1}) \;=\; \mathsf{wp}[\![C_{\text{iter}}]\!]\,([\![[\sigma_{i+1}]]\!])(\sigma_i)\,.$$

Using the induction hypothesis (3.1), we construct a *single* syntactic expectation $g_{C_{\text{iter}}}^{\sigma_{i+1}} \in \mathsf{Exp}$, where we introduce auxiliary variables to realize the parameterization in σ_{i+1}, such that *for all* $\sigma_i, \sigma_{i+1} \in \mathsf{States}$, we have

$$[\![C_{\text{iter}}]\!]^{\sigma_i}(\sigma_{i+1}) \;=\; \mathsf{wp}[\![C_{\text{iter}}]\!]\,([\![[\sigma_{i+1}]]\!])(\sigma_i) \;=\; \left[\!\!\left[g_{C_{\text{iter}}}^{\sigma_{i+1}}\right]\!\!\right](\sigma_i)\,.$$

[5]Recall that we assume for simplicity that Vars is finite.

Plugging the above equality into our "small-step" characterization of loops (3.2) then yields the following characterization of $[\![g]\!]$ in (†):

$$\begin{aligned} & \mathsf{wp}[\![\mathtt{while}(\varphi)\{C'\}]\!]([\![f]\!]) \\ = {} & \lambda\sigma_0.\ \underbrace{\sup_{k\in\mathbb{N}} \underbrace{\sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}} \underbrace{\left[\!\!\left[\underbrace{[\neg\varphi]\cdot f}_{\in\mathsf{Exp}}\right]\!\!\right](\sigma_k)\cdot\underbrace{\prod_{i=0}^{k-1}\left[\!\!\left[\underbrace{g^{\sigma_{i+1}}_{C_{\mathrm{iter}}}}_{\in\mathsf{Exp}}\right]\!\!\right](\sigma_i)}_{\textit{product of variable length expressible in } \mathsf{Exp}?}}_{\text{products of constant length 2 expressible in } \mathsf{Exp}}}_{\textit{sum over variable-length state sequences expressible in } \mathsf{Exp}?}}_{\text{suprema supported in } \mathsf{Exp} \text{ via Ƨ}} \end{aligned} \tag{3.3}$$

3.4.1.3 Encoding Loops as Syntactic Expectations

Let us now revisit the individual components of the expectation (3.3) from above and discuss how to encode them as syntactic expectations in Exp, moving through the braces from bottom to top:

The Supremum $\sup_{k\in\mathbb{N}}$. The supremum ensures that terminating execution paths *of arbitrary length* are accounted for; it is supported in Exp by the Ƨ quantifier. If we already know a syntactic expectation $g \in \mathsf{Exp}$ for the entire sum that follows, we hence obtain an encoding of the whole expectation, namely

$$\text{Ƨ} k\colon g \;\in\; \mathsf{Exp}\,,$$

where we ensure that g evaluates to 0 if k evaluates to a non-integral number.

The Sum $\sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}}$. This sum *cannot* directly be written as a syntactic expectation: First, it sums over *sequences of program states* of length k. Second, its *number of summands depends on the length* k whereas Exp (syntactically) only supports sums with a constant number of summands. To deal with the first issue, there is a standard solution in proofs of expressiveness for classical non-probabilistic programs (cf. [LS87; Win93; TCA19; TCA09]): We employ *Gödelization* to encode both program states and finite sequences of program states as natural numbers in syntactic expectations. The details are found in Section 3.4.2. In particular:

- We show that Exp subsumes both first-order arithmetic $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ over $\mathbb{Q}_{\geq 0}$ and first-order arithmetic $\mathbf{A}_{\mathbb{N}}$ over $\mathbb{N}$ in Section 3.4.2.1.
- This enables us to utilize Gödelization [Göd31] in Sections 3.4.2.2 and 3.4.2.3, i.e., to encode finite sequences over both $\mathbb{N}$ and $\mathbb{Q}_{\geq 0}$ as Gödel numbers in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ and thereby in Exp.
- In Section 3.4.2.4, we then encode program states and sequences of program states in Exp as sequences over $\mathbb{Q}_{\geq 0}$ and $\mathbb{N}$, respectively.

To deal with the second issue (the sum having a variable number of summands), we also rely on the ability to encode sequences as Gödel numbers in Exp. Roughly speaking, we encode the sum as follows:

- In Section 3.4.3, we construct for each syntactic expectation h its *Dedekind-characteristic formula* to represent h in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$, exploiting that the *reals* α that h evaluates to can be represented by their *Dedekind cuts* [Ber49] — the set of *rationals* strictly smaller than α.
- In Section 3.4.4, we then combine Dedekind-characteristic formulae with Gödelization to construct for every $h \in \mathsf{Exp}$ another *syntactic* expectation

$$\overset{y}{\underset{x=0}{\mathsf{SUM}}}\colon h \quad \text{with} \quad \left[\!\!\left[\overset{y}{\underset{x=0}{\mathsf{SUM}}}\colon h \right]\!\!\right](\sigma) \;=\; \begin{cases} \sum_{i=0}^{\sigma(y)} [\![h]\!](\sigma[x \mapsto i]) & \text{if } \sigma(y) \in \mathbb{N} \\ 0 & \text{otherwise}\,, \end{cases}$$

  evaluating to the sought-after sums with a variable number of summands.

The Product $[\![[\neg\varphi]\cdot f]\!]\cdot\ldots$. This product is expressible in Exp.

The Product $\prod_{i=0}^{k-1} \left[\!\!\left[g_{C_{\mathsf{iter}}}^{\sigma_{i+1}} \right]\!\!\right](\sigma_i)$. Products with a variable number of factors require a similar approach as for sums. See Section 3.4.4 for details.

The Expectations $[\![[\neg\varphi]\cdot f]\!]$ and $\left[\!\!\left[g_{C_{\mathsf{iter}}}^{\sigma_{i+1}} \right]\!\!\right]$. The former is a syntactic expectation by construction whereas the latter is obtained from the induction hypothesis, where we apply Gödelization to evaluate syntactic expectations in states given as Gödel numbers. See Section 3.4.5 for details.

The Expressiveness Proof. We glue together the constructions for the individual components of the expectation (3.3), which characterizes the weakest preexpectation of loops. We present the full construction, a proof of its correctness, and an example of the resulting syntactic expectation in Section 3.4.5.

$\mathcal{P} \in \mathbf{A}_{\mathbb{N}}$	$\mathcal{P}_{\mathbb{Q}_{\geq 0}} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$	$\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$	$[\mathcal{P}] \in \mathsf{Exp}$
φ	$\varphi \wedge N(x_1) \wedge \ldots \wedge N(x_n)$	φ	$[\varphi]$
$\exists v \colon \mathcal{P}'$	$\exists v \colon \mathcal{P}'_{\mathbb{Q}_{\geq 0}}$	$\exists v \colon \mathcal{P}'$	$\reflectbox{$\mathsf{S}$} v \colon [\mathcal{P}']$
$\forall v \colon \mathcal{P}'$	$\forall v \colon N(v) \longrightarrow \mathcal{P}'_{\mathbb{Q}_{\geq 0}}$	$\forall v \colon \mathcal{P}'$	$\reflectbox{$\mathsf{J}$} v \colon [\mathcal{P}']$

Table 3.5: Embedding of $\mathcal{P} \in \mathbf{A}_{\mathbb{N}}$ in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ (left, where $\mathsf{FV}(\varphi) = \{x_1, \ldots, x_n\}$) and of $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ in Exp (right).

3.4.2 Gödelization for Syntactic Expectations

We show that Exp subsumes the standard model of first-order arithmetic over both $\mathbb{Q}_{\geq 0}$ (Theorem 3.5) and $\mathbb{N}$ (Theorem 3.6). Consequently, Exp conservatively extends the standard assertion language of Floyd-Hoare logic for classical programs (cf. [Win93; LSS84; Coo78]). Moreover, this enables us to encode finite sequences of both natural numbers and non-negative rationals in Exp utilizing Gödelization [Göd31] — a central ingredient of our expressiveness proof.

3.4.2.1 Embedding First-Order Arithmetic in Exp

Formulas $\mathcal{P}$ in first-order arithmetic over $\mathbb{Q}_{\geq 0}$ are obtained by extending Boolean expressions φ by existential $\exists$ and universal $\forall$ quantifiers. For simplicity, we assume without loss of generality that all formulas $\mathcal{P}$ are in *prenex normalform*.

Definition 3.9 (First-Order Arithmetic over $\mathbb{Q}_{\geq 0}$).
First-order arithmetic formulae over $\mathbb{Q}_{\geq 0}$, which we denote by $\mathcal{P}, \mathcal{Q}$, and variations thereof, in the set $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ adhere to the grammar

$$\begin{aligned} \mathcal{P} \quad \longrightarrow \quad & \varphi \in \mathsf{BExpr} && \text{(Boolean expressions (see Definition 3.3))} \\ & \mid \exists v \colon \mathcal{P} && \text{(existential quantification)} \\ & \mid \forall v \colon \mathcal{P}\,. && \text{(universal quantification)} \end{aligned}$$

The semantics of formulae $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ is standard, e.g.,

$$\sigma \models \forall v \colon \mathcal{P} \qquad \text{iff} \qquad \text{for all } q \in \mathbb{Q}_{\geq 0} \colon \sigma[v \mapsto q] \models \mathcal{P}\,.$$

We adopt the same notation as for syntactic expectations, i.e., $\mathsf{FV}(\mathcal{P})$ is the finite set of free variables in $\mathcal{P}$ and $\mathcal{P}[x/a]$ is the formula obtained from substituting every occurrence of x in $\mathcal{P}$ by a in a capture-avoiding manner (cf. Definition 3.8).

The set $\mathbf{A}_{\mathbb{N}}$ of formulas $\mathcal{P}$ in first-order arithmetic over $\mathbb{N}$ is defined analogously by restricting to Boolean expressions where all constants are in $\mathbb{N}$. We denote the set of all such Boolean expressions by $\mathsf{BExpr}_{\mathbb{N}}$.

Definition 3.10 (First-Order Arithmetic over $\mathbb{N}$).
First-order arithmetic formulae over $\mathbb{N}$, which we also denote by $\mathcal{P}, \mathcal{Q}$, and variations thereof, in the set $\mathbf{A}_{\mathbb{N}}$ adhere to the grammar

$$\begin{aligned} \mathcal{P} \quad \longrightarrow \quad & \varphi \in \mathsf{BExpr}_{\mathbb{N}} && \text{(Boolean expressions with constants in } \mathbb{N}) \\ & \mid \exists v \colon \mathcal{P} && \text{(existential quantification)} \\ & \mid \forall v \colon \mathcal{P}\,. && \text{(universal quantification)} \end{aligned}$$

Since our program states are of type $\mathsf{Vars} \to \mathbb{Q}_{\geq 0}$ rather than $\mathsf{Vars} \to \mathbb{N}$, we define an auxiliary satisfaction relation $\models_{\mathbb{N}} \subseteq \mathsf{States} \times \mathbf{A}_{\mathbb{N}}$ which ensures (1) that all free variables occurring in a formula $\mathcal{P} \in \mathbf{A}_{\mathbb{N}}$ evaluate to values in $\mathbb{N}$ under the current state and (2) that the quantifiers range over $\mathbb{N}$:

$$\begin{aligned} \sigma \models_{\mathbb{N}} \varphi \quad & \text{iff} \quad \sigma \models \varphi \text{ and for all } x \in \mathsf{FV}(\varphi) \colon \sigma(x) \in \mathbb{N} \\ \sigma \models_{\mathbb{N}} \exists v \colon \mathcal{P}' \quad & \text{iff} \quad \text{exists } i \in \mathbb{N} \colon \sigma[v \mapsto i] \models_{\mathbb{N}} \mathcal{P}' \\ \sigma \models_{\mathbb{N}} \forall v \colon \mathcal{P}' \quad & \text{iff} \quad \text{for all } i \in \mathbb{N} \colon \sigma[v \mapsto i] \models_{\mathbb{N}} \mathcal{P}' \end{aligned}$$

Next, we adopt a result by Julia Robinson to show that $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ subsumes $\mathbf{A}_{\mathbb{N}}$.

Lemma 3.3 (Definability of $\mathbb{N}$ in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ (adapted from [Rob49])).
The set $\mathbb{N}$ of natural numbers is definable in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$, i.e., there is a formula $N(x) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ with free variable x such that for all $\sigma \in \mathsf{States}$, we have

$$\sigma \models [\![N(x)]\!] \quad \text{iff} \quad \sigma(x) \in \mathbb{N}\,.$$

Hence, $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ subsumes $\mathbf{A}_{\mathbb{N}}$ in the following sense:

Theorem 3.4 (Embedding of $\mathbf{A}_{\mathbb{N}}$ in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$).
Let $\mathcal{P} \in \mathbf{A}_{\mathbb{N}}$ and let $\mathcal{P}_{\mathbb{Q}_{\geq 0}} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ be the formula obtained from $\mathcal{P}$ by applying the rules shown in Table 3.5 (left). For all $\sigma \in \mathsf{States}$, we have

$$\sigma \models \mathcal{P}_{\mathbb{Q}_{\geq 0}} \quad \text{iff} \quad \sigma \models_{\mathbb{N}} \mathcal{P}\,.$$

Proof. By induction on $\mathcal{P}$. ■

3

Next, we show that Exp subsumes $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$. Given $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$, we define an effectively constructible syntactic expectation $[\mathcal{P}]$ obtained from $\mathcal{P}$ by (1) taking Iverson brackets for Boolean expressions, i.e., quantifier-free formulae, and (2) substituting the quantifiers $\exists/\forall$ by their quantitative analogs $\reflectbox{$\mathsf{S}$}/\reflectbox{$\mathsf{J}$}$.

Theorem 3.5 (Embedding of $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ in Exp).
Let $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ and let $[\mathcal{P}] \in \mathsf{Exp}$ be the syntactic expectation obtained from $\mathcal{P}$ by applying the rules shown in Table 3.5 (right). For all $\sigma \in \mathsf{States}$, we have

$$[\![[\mathcal{P}]]\!](\sigma) \;=\; \begin{cases} 1 & \text{if } \sigma \models \mathcal{P} \\ 0 & \text{if } \sigma \not\models \mathcal{P}\,. \end{cases}$$

Proof. By induction on $\mathcal{P}$. ∎

Finally, by Theorems 3.4 and 3.5, Exp also subsumes $\mathbf{A}_{\mathbb{N}}$.

Theorem 3.6 (Embedding of $\mathbf{A}_{\mathbb{N}}$ in Exp).
For every $\mathcal{P} \in \mathbf{A}_{\mathbb{N}}$, there is an effectively constructible syntactic expectation $[\mathcal{P}] \in \mathsf{Exp}$ such that for all $\sigma \in \mathsf{States}$, we have

$$[\![[\mathcal{P}]]\!](\sigma) \;=\; \begin{cases} 1 & \text{if } \sigma \models_{\mathbb{N}} \mathcal{P} \\ 0 & \text{if } \sigma \not\models_{\mathbb{N}} \mathcal{P}\,. \end{cases}$$

Notice that $[\mathcal{P}]$ possibly contains quantitative quantifiers $\reflectbox{$\mathsf{S}$}$ or $\reflectbox{$\mathsf{J}$}$.

3.4.2.2 Encoding Sequences of Natural Numbers

We employ a result by Gödel [Göd31] for encoding finite sequences of natural numbers in a *single* number — a process often referred to as *Gödelization*.

Lemma 3.7 (Gödelization [Göd31] of Finite Sequences over $\mathbb{N}$).
There is a formula $\mathsf{Elem}(x_1, x_2, x_3) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ such that for all $\sigma \in \mathsf{States}$:

1. If $\sigma(x_i) \notin \mathbb{N}$ for some $i \in \{1,2,3\}$, then $\sigma \not\models \mathsf{Elem}(x_1, x_2, x_3)$.
2. For all $m, m' \in \mathbb{N}$, we have
   $$\sigma \models \mathsf{Elem}(x_1, x_2, m) \text{ and } \sigma \models \mathsf{Elem}(x_1, x_2, m') \qquad \text{implies} \qquad m = m'\,,$$
   and if $\sigma(x_1), \sigma(x_2) \in \mathbb{N}$, then there is $m \in \mathbb{N}$ with $\sigma \models \mathsf{Elem}(x_1, x_2, m)$.
3. For all $k \geq 0$ and all (possibly empty) finite sequences $n_0, \ldots, n_{k-1}$ of

natural numbers, there is a Gödel number $num \in \mathbb{N}$ such that

$$\text{for all } i \in \{0, \ldots, k-1\}: \quad \sigma \models \mathsf{Elem}(num, i, x_3) \text{ iff } \sigma(x_3) = n_i \,.$$

Proof. By using the fact that Gödel's β-function is definable in $\mathbf{A}_{\mathbb{N}}$ and using the fact that $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ subsumes $\mathbf{A}_{\mathbb{N}}$ (Theorem 3.4). ■

Gödelization gives rise to a powerful concept: Quantifying over finite sequences of arbitrary length is not *syntactically* supported in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$. However, with Elem, such a quantification can nonetheless be expressed by quantifying over a *single* variable. We can thereby define various predicates in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ whose definability is — at first sight — far from obvious.

3

Example 3.3.
We use Elem to define a formula $\mathsf{Fac}(x,y) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ which expresses that y is the factorial of x, i.e.,

$$\begin{aligned}&\text{for all } \sigma \in \mathsf{States}:\\ &\qquad \sigma \models \mathsf{Fac}(x,y) \quad \text{iff} \quad \sigma(x), \sigma(y) \in \mathbb{N} \text{ and } \sigma(y) = \sigma(x)! \,.\end{aligned}$$

where $n!$ denotes the factorial of $n \in \mathbb{N}$ which we define recursively as

$$n! \;=\; \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise}\,. \end{cases}$$

The idea is to use the above recursive definition of $n!$ to express the fact that for $n, m \in \mathbb{N}$ we have $m = n!$ equivalently as the existence of a sequence $m_0, \ldots, m_n$ of natural numbers such that

$$m_0 = 1 \text{ and } m_n = m \text{ and for all } i \in \{1, \ldots, n\}\colon\; m_i = i \cdot m_{i-1} \,.$$

Using Elem, we can express this characterization in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ as follows:

$$\begin{aligned}&\mathsf{Fac}(x,y)\\ =\; & \underbrace{\exists v\colon}_{\text{there is a sequence } m_0,\ldots \text{ such that}} \quad \underbrace{\mathsf{Elem}(v,0,1)}_{m_0=1} \wedge \underbrace{\mathsf{Elem}(v,x,y)}_{m_x=y}\\ &\quad \wedge \underbrace{\forall 1 \leq u \leq x\colon \forall w\colon\; (\mathsf{Elem}(v, u \dot{-} 1, w) \implies \mathsf{Elem}(v,u,u\cdot w))}_{\text{for all } u\in\{1,\ldots,x\}\colon\; m_u = u\cdot m_{u-1}}\,.\end{aligned}$$

Notice that if $\sigma(x) \notin \mathbb{N}$ or $\sigma(y) \notin \mathbb{N}$, then $\sigma \not\models \mathsf{Fac}(x,y)$ by Lemma 3.7.1.

To assign a *unique* Gödel number *num* to a (possibly empty) sequence $n_0, \ldots, n_{k-1}$ of length k we employ *minimization*, i.e., we choose the least suitable Gödel number. Formally, we define the $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ formula

$$
\begin{aligned}
& \mathsf{Sequence}(x, y) \\
= \; & N(x) \wedge N(y) \\
& \wedge \forall v \colon \underbrace{\Big(\forall w \colon (N(w) \wedge w < y) \implies \exists u \colon \mathsf{Elem}(x, w, u) \wedge \mathsf{Elem}(v, w, u)\Big)}_{x \text{ and } v \text{ encode sequences coinciding on first } y \text{ elements}} \\
& \qquad \implies v \geq x \, ,
\end{aligned}
$$

i.e., $\sigma \models \mathsf{Sequence}(x, y)$ iff $\sigma(x), \sigma(y) \in \mathbb{N}$ and $num = \sigma(x)$ is the smallest Gödel number of the uniquely determined sequence $n_0, \ldots, n_{\sigma(y)-1} \in \mathbb{N}$ satisfying

$$
\text{for all } i \in \{0, \ldots, \sigma(y) - 1\} \colon \quad \sigma \models \mathsf{Elem}(num, i, n_i) \, .
$$

For every $k \in \mathbb{N}$ and every sequence $n_0, \ldots, n_{k-1} \in \mathbb{N}$ of length k, we then define *the unique* Gödel number of the sequence $n_0, \ldots, n_{k-1}$ as the natural number $\langle n_0, \ldots, n_{k-1} \rangle$ satisfying

$$
\mathsf{Sequence}(\langle n_0, \ldots, n_{k-1} \rangle, k) \;\wedge\; \bigwedge_{i=0}^{k-1} \mathsf{Elem}(\langle n_0, \ldots, n_{k-1} \rangle, i, n_i) \, ,
$$

where the empty conjunction is equivalent to true.

3.4.2.3 Encoding Sequences of Non-Negative Rationals

We employ the formula Elem from Lemma 3.7 to uniquely encode finite sequences of *non-negative rationals* in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$. The main idea is to exploit that every non-negative rational q can be uniquely written as a fraction $q = {}^{n}\!/_{m}$ in lowest terms, i.e., such that $n, m \in \mathbb{N}$ and n and m are relatively prime. Hence, every finite sequence of non-negative rationals can be identified with two finite sequences of natural numbers, which can be encoded via Elem.

Lemma 3.8 (Gödelization [Göd31] of Finite Sequences over $\mathbb{Q}_{\geq 0}$). There is a formula $\mathsf{RElem}(x_1, x_2, x_3) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ such that for all $\sigma \in \mathsf{States}$:

1. If $\sigma(x_i) \notin \mathbb{N}$ for some $i \in \{1, 2\}$, then $\sigma \not\models \mathsf{RElem}(x_1, x_2, x_3)$.
2. For all $q, q' \in \mathbb{Q}_{\geq 0}$,

$$
\sigma \models \mathsf{RElem}(x_1, x_2, q) \text{ and } \sigma \models \mathsf{RElem}(x_1, x_2, q') \qquad \text{implies} \qquad q = q' \, ,
$$

and if $\sigma(x_1), \sigma(x_2) \in \mathbb{N}$, then there is $q \in \mathbb{Q}_{\geq 0}$ with $\sigma \models \mathsf{RElem}(x_1, x_2, q)$.

3. For all $k \geq 0$ and all (possibly empty) finite sequences $q_0, \ldots, q_{k-1}$ of non-negative rationals, there is a Gödel number $num \in \mathbb{N}$ such that

$$\text{for all } i \in \{0, \ldots, k-1\}\colon \quad \sigma \models \mathsf{RElem}(num, i, x_3) \text{ iff } \sigma(x_3) = q_i\ .$$

Proof. See [8, Appendix B.4] for details. ■

Analogously to the previous section, we define a formula $\mathsf{RSequence}(x, y) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ that uses minimization to define a *unique* Gödel number num for every finite sequence $q_0, \ldots, q_{k-1}$ of non-negative rationals. The only difference between $\mathsf{RSequence}(x, y)$ and $\mathsf{Sequence}(x, y)$ is that every occurrence of Elem is replaced by RElem. Similarly, for every k and every sequence $q_0, \ldots, q_{k-1}$, we define *the unique* Gödel number encoding that sequence as the unique natural number $\langle q_0, \ldots, q_{k-1}\rangle \in \mathbb{N}$ satisfying the $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ formula

$$\mathsf{RSequence}(\langle q_0, \ldots, q_{k-1}\rangle, k) \ \wedge \ \bigwedge_{i=0}^{k-1} \mathsf{RElem}(\langle q_0, \ldots, q_{k-1}\rangle, i, q_i)\ .$$

Example 3.4.
Analogously to Example 3.3, we use RElem to define a formula $\mathsf{Harmonic}(x, y) \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ which expresses that y is the x-th harmonic number, i.e., such that for all $\sigma \in \mathsf{States}$, we have

$$\sigma \models \mathsf{Harmonic}(x, y) \quad \text{iff} \quad \sigma(x) \in \mathbb{N} \text{ and } \sigma(y) = \sum_{i=1}^{\sigma(x)} \frac{1}{i}\ .$$

We construct $\mathsf{Harmonic}(x, y)$ as follows:

$$\begin{aligned} &\mathsf{Harmonic}(x, y) \\ = \quad & \underbrace{\exists v\colon}_{\text{there is a sequence } q_0, \ldots \text{ such that}} \quad \underbrace{\mathsf{RElem}(v, 0, 0)}_{q_0 = 0} \wedge \underbrace{\mathsf{RElem}(v, x, y)}_{q_x = y} \\ & \underbrace{\begin{aligned} \wedge\ \forall 1 \leq u \leq x\colon \forall w\colon\ &(\mathsf{RElem}(v, u \dot{-} 1, w) \\ &\implies (\exists w'\colon w' \cdot u = 1 \wedge \mathsf{RElem}(v, u, w' + w)))\ . \end{aligned}}_{\text{for all } u \in \{1, \ldots, x\}\colon\ q_u = \frac{1}{u} + q_{u-1}} \end{aligned}$$

Using the fact that Exp subsumes $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ (Theorem 3.5), we also obtain a syntactic expectation $\mathsf{Harmonic}(x) \in \mathsf{Exp}$ which evaluates to the x-th harmonic number, which is defined as

$$\mathsf{Harmonic}(x) = \mathfrak{S} v' \colon [\mathsf{Harmonic}(x, v')] \cdot v' \text{ ,such that we have}$$

$$[\![\mathsf{Harmonic}(x)]\!](\sigma) = \begin{cases} \sum_{i=1}^{\sigma(x)} \frac{1}{i} & \text{if } \sigma(x) \in \mathbb{N} \\ 0 & \text{otherwise .} \end{cases}$$

Intuitively, the $\mathfrak{S}$ quantifier selects the rational for v' which makes $[\mathsf{Harmonic}(x, v')]$ evaluate to 1 (if such a rational exists). Multiplying with v' then yields the desired result by construction.

3.4.2.4 Encoding Sequences of Program States

We use RElem (Lemma 3.8) and Elem (Lemma 3.7) to encode both program states and finite sequences of program states in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$. For that, we first fix an ordered (possibly empty) k-tuple $\mathbf{x} = (x_0, \ldots, x_{k-1})$ of pairwise distinct “relevant” variables. Intuitively, $\mathbf{x}$ consists of all variables that appear in a given program or a postexpectation. Inspired by [Win93], the idea is then to encode (the relevant part of) a state σ as the sequence $\sigma(x_0), \ldots, \sigma(x_{k-1})$ of non-negative rationals.

We define an equivalence relation $\sim_{\mathbf{x}} \subseteq \mathsf{States} \times \mathsf{States}$ between states as

$$\sigma_1 \sim_{\mathbf{x}} \sigma_2 \qquad \text{iff} \qquad \text{for all } i \in \{0, \ldots, k-1\} \colon \sigma_1(x_i) = \sigma_2(x_i) \,.$$

Every num satisfying $\mathsf{RSequence}(num, k)$ encodes *exactly one* sequence of non-negative rationals of length k and thus also *exactly one* state σ modulo $\sim_{\mathbf{x}}$, i.e., one equivalence class of $\sim_{\mathbf{x}}$. We define the Gödel number $\langle \sigma \rangle_{\mathbf{x}} \in \mathbb{N}$ encoding a given state σ (w.r.t. $\mathbf{x}$) as the unique number satisfying the $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ formula

$$\mathsf{RSequence}(\langle \sigma \rangle_{\mathbf{x}}, k) \ \wedge \ \bigwedge_{i=0}^{k-1} \mathsf{RElem}(\langle \sigma \rangle_{\mathbf{x}}, i, \sigma(x_i)) \,.$$

Moreover, we define the $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ formula

$$\mathsf{EncodesState}_{\mathbf{x}}(x) = \mathsf{RSequence}(x, k) \ \wedge \ \bigwedge_{i=0}^{k-1} \mathsf{RElem}(x, i, x_i) \,,$$

which is satisfied by σ iff $\sigma(x)$ is the Gödel number of a state σ' with $\sigma \sim_{\mathbf{x}} \sigma'$.

Now, since program states are encoded as natural numbers, sequences of program states can be encoded as sequences of natural numbers. Let

f	$\mathsf{Dedekind}[f,x]$
a	$x < a \vee x = 0$
$[\varphi]$	$(\varphi \wedge x < 1) \vee x = 0$
$g + h$	$\exists v_1, v_2\colon \mathsf{Dedekind}[g, v_1] \wedge \mathsf{Dedekind}[h, v_2] \wedge x = v_1 + v_2$
$g \cdot h$	$\exists v_1, v_2\colon \mathsf{Dedekind}[g, v_1] \wedge \mathsf{Dedekind}[h, v_2] \wedge x = v_1 \cdot v_2$
$\mathsf{S} v\colon g$	$\exists v\colon \mathsf{Dedekind}[g, x]$
$\mathsf{L} v\colon g$	$x = 0 \vee (\exists v'\colon x < v' \wedge \forall v\colon \mathsf{Dedekind}[g, v'])$

Table 3.6: Inductive definition of $\mathsf{Dedekind}[f,x] \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$. Here v', v_1, and v_2 are fresh variables. Moreover, we assume w.l.o.g. that x is distinct from v.

$\sigma_0, \ldots, \sigma_{n-1}$ be such a sequence of length n. We define the *unique*[6] Gödel number $\langle\langle \sigma_0, \ldots, \sigma_{n-1} \rangle\rangle_{\mathbf{x}}$ encoding $\sigma_0, \ldots, \sigma_{n-1}$ as the unique number satisfying

$$\mathsf{Sequence}(\langle\langle \sigma_0, \ldots, \sigma_{n-1} \rangle\rangle_{\mathbf{x}}, n) \;\wedge\; \bigwedge_{i=0}^{n-1} \mathsf{Elem}(\langle\langle \sigma_0, \ldots, \sigma_{n-1} \rangle\rangle_{\mathbf{x}}, i, \langle \sigma_i \rangle_{\mathbf{x}})\,.$$

Moreover, we define the formula

$$\begin{aligned} & \mathsf{StateSequence}_{\mathbf{x}}(x, y) \\ = \; & \mathsf{Sequence}(x, y) \wedge \big(\exists y'\colon \mathsf{Elem}(x, 0, y') \wedge \mathsf{EncodesState}_{\mathbf{x}}(y')\big) \\ & \quad \wedge \forall u\colon \forall y'\colon \big((u < y \wedge \mathsf{Elem}(x, u, y')) \implies \mathsf{RSequence}(y', k)\big)\,, \end{aligned}$$

which is satisfied by state σ iff (1) $num = \sigma(x)$ is the unique Gödel number of some sequence $\sigma_0, \ldots, \sigma_{\sigma(y)-1}$ of states of length $\sigma(y) \in \mathbb{N}$ and (2) $\sigma \sim_{\mathbf{x}} \sigma_0$.

3.4.3 Dedekind-Characteristic Formulae

Recall from Example 3.1.1 that a syntactic expectation f possibly evaluates to an *irrational* number. Since variables evaluate to non-negative *rationals*, we can

[6]modulo $\sim_{\mathbf{x}}$

thus not directly bind a fresh variable x to the reals f evaluates to. Put more formally, there are $f \in \mathsf{Exp}$ for which there is no formula $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ such that

$$\text{for all } \sigma \in \mathsf{States}\colon \quad \sigma \models \mathcal{P} \quad \text{iff} \quad \underbrace{\sigma(x)}_{\in \mathbb{Q}_{\geq 0}} = \underbrace{[\![f]\!](\sigma)}_{\text{possibly } \notin \mathbb{Q}_{\geq 0}} .$$

To circumvent this issue, we exploit that every non-negative extended real $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$ can be identified with the set of rationals strictly smaller[7] than α:

Definition 3.11 (Dedekind Cuts adapted from [Ber49]).
Let $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$. The *Dedekind cut of α* is defined as

$$\mathsf{Cut}(\alpha) = \{q \in \mathbb{Q}_{\geq 0} \mid q < \alpha\} \cup \{0\} .$$

Every $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$ is equal to the supremum of its Dedekind cut:

Lemma 3.9 (Reals from Dedekind Cuts).
For every $\alpha \in \mathbb{R}_{\geq 0}^{\infty}$, we have $\alpha = \sup \mathsf{Cut}(\alpha)$.

Dedekind cuts thus reduce[8] reasoning about reals to reasoning about *sets of rationals*. Crucially, we have for all $\alpha, \beta \in \mathbb{R}_{\geq 0}^{\infty}$ and $\circ \in \{+, \cdot\}$,

$$\mathsf{Cut}(\alpha \circ \beta) = \{q \circ r \mid q \in \mathsf{Cut}(\alpha) \text{ and } r \in \mathsf{Cut}(\beta)\}$$

and thus, by Lemma 3.9,

$$\alpha \circ \beta = \sup \{q \circ r \mid q \in \mathsf{Cut}(\alpha) \text{ and } r \in \mathsf{Cut}(\beta)\} .$$

Now, even though we cannot directly bind a variable x to the reals f evaluates to, we show that the Dedekind cuts of these reals *can* be expressed in an effectively constructible manner. Towards this end, consider the following:

Definition 3.12 (Dedekind-Characteristic Formulae).
Let $f \in \mathsf{Exp}$ and let $x \in \mathsf{Vars}$. The *Dedekind-characteristic formula of f (w.r.t. x)*

$$\mathsf{Dedekind}[f, x] \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$$

is defined by induction on f in Table 3.6.

[7]We always include 0 for technical reasons since we operate in the *non-negative* extended reals rather than the whole extended real number line.

[8]In fact, Dedekind cuts are a means to construct the (non-negative extended) reals and their arithmetic operations from the rationals.

Theorem 3.10 (Dedekind Cuts of Syntactic Expectations in $\mathsf{A}_{\mathbb{Q}_{\geq 0}}$). Let $f \in \mathsf{Exp}$. We have for all $\sigma \in \mathsf{States}$:

$$\sigma \models \mathsf{Dedekind}[f, x] \quad \text{iff} \quad \sigma(x) \in \mathsf{Cut}(⟦f⟧(\sigma))$$

Proof. By induction on f. The base cases are straightforward.

The cases $f = g \circ h$ for $\circ \in \{+, \cdot\}$. Let $\sigma \in \mathsf{States}$. We have

$$\begin{array}{ll}
 & \sigma \models \mathsf{Dedekind}[f, x] \\
\text{iff} & \sigma \models \exists v_1, v_2 \colon \mathsf{Dedekind}[g, v_1] \wedge \mathsf{Dedekind}[h, v_2] \wedge x = v_1 \circ v_2 \\
 & \hfill \text{(Table 3.6)} \\
\text{iff} & \text{exists } q_1, q_2 \in \mathbb{Q}_{\geq 0} \colon q_1 \in \mathsf{Cut}(⟦g⟧(\sigma)) \text{ and } q_2 \in \mathsf{Cut}(⟦h⟧(\sigma)) \\
 & \qquad \text{and } \sigma(x) = q_1 \circ q_2 \qquad \text{(I.H., } v_1, v_2 \text{ do not occur in } g \text{ or } h\text{)} \\
\text{iff} & \sigma(x) \in \{q_1 \circ q_2 \mid q_1 \in \mathsf{Cut}(⟦g⟧(\sigma)) \text{ and } q_2 \in \mathsf{Cut}(⟦h⟧(\sigma))\} \\
\text{iff} & \sigma(x) \in \mathsf{Cut}(⟦g⟧(\sigma) \circ ⟦h⟧(\sigma)) \\
\text{iff} & \sigma(x) \in \mathsf{Cut}(⟦f⟧(\sigma)) ~.
\end{array}$$

The case $f = \reflectbox{$\mathsf{S}$} v \colon g$. Let $\sigma \in \mathsf{States}$. We have

$$\begin{array}{lll}
 & \sigma \models \mathsf{Dedekind}[f, x] & \\
\text{iff} & \sigma \models \exists v \colon \mathsf{Dedekind}[g, x] & \text{(Table 3.6)} \\
\text{iff} & \text{exists } q \in \mathbb{Q}_{\geq 0} \colon \sigma[v \mapsto q] \models \mathsf{Dedekind}[g, x] & \\
\text{iff} & \text{exists } q \in \mathbb{Q}_{\geq 0} \colon \sigma(x) \in \mathsf{Cut}(⟦g⟧(\sigma[v \mapsto q])) & \\
 & & \text{(I.H., } x \text{ is distinct from } v \text{ by assumption)} \\
\text{iff} & \text{exists } q \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < ⟦g⟧(\sigma[v \mapsto q]) \text{ or } \sigma(x) = 0 & \text{(Definition 3.11)} \\
\text{iff} & \sigma(x) = 0 \text{ or } \sigma(x) < \sup\{⟦g⟧(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} & \\
 & & \text{(standard property of suprema)} \\
\text{iff} & \sigma(x) = 0 \text{ or } \sigma(x) < ⟦\reflectbox{S} v \colon g⟧(\sigma) & \text{(Table 3.3 on page 85)} \\
\text{iff} & \sigma(x) \in \mathsf{Cut}(⟦f⟧(\sigma)) ~. & \text{(Definition 3.11)}
\end{array}$$

The case $f = \reflectbox{$\mathsf{J}$} v \colon g$. Let $\sigma \in \mathsf{States}$. We have

$$\begin{array}{lll}
 & \sigma \models \mathsf{Dedekind}[f, x] & \\
\text{iff} & \sigma \models x = 0 \vee (\exists v' \colon x < v' \wedge \forall v \colon \mathsf{Dedekind}[g, v']) & \text{(Table 3.6)}
\end{array}$$

$$\begin{aligned}
\text{iff}\quad & \sigma(x) = 0 \text{ or exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon \sigma[v \mapsto q][v' \mapsto q'] \models \mathsf{Dedekind}[g, v'] \\
& \qquad\qquad (v' \text{ is fresh and } x \text{ is distinct from } v \text{ by assumption}) \\
\text{iff}\quad & \sigma(x) = 0 \text{ or exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' && (\text{I.H., } v' \text{ is fresh}) \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon q' \in \mathsf{Cut}(\llbracket g \rrbracket(\sigma[v \mapsto q])) \\
\text{iff}\quad & \sigma(x) = 0 \text{ or exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' && (\text{Definition 3.11}) \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon q' < \llbracket g \rrbracket(\sigma[v \mapsto q]) \vee q' = 0 \\
\text{iff}\quad & \sigma(x) = 0 \text{ or exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' && (q' = 0 \text{ not possible}) \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon q' < \llbracket g \rrbracket(\sigma[v \mapsto q]) \\
\text{iff}\quad & \sigma(x) = 0 \text{ or } \sigma(x) < \inf\{\llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} && (\text{see below } (\dagger)) \\
\text{iff}\quad & \sigma(x) = 0 \text{ or } \sigma(x) < \llbracket \curlywedge v \colon g \rrbracket(\sigma) && (\text{Table 3.3 on page 85}) \\
\text{iff}\quad & \sigma(x) \in \mathsf{Cut}(\llbracket f \rrbracket(\sigma))\ . && (\text{Definition 3.11})
\end{aligned}$$

To see that the equivalence (†) holds, consider the following: If $\sigma(x) = 0$, the claim holds trivially. Now let $\sigma(x) \neq 0$. For the "if"-direction, we have:

$$\begin{aligned}
& \sigma(x) < \inf\{\llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} \\
\text{implies}\quad & \text{exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' && (\mathbb{Q}_{\geq 0} \text{ is dense}) \\
& \qquad \text{and } q' < \inf\{\llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} \\
\text{implies}\quad & \text{exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' && (\text{standard property of infima}) \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon q' < \llbracket g \rrbracket(\sigma[v \mapsto q])\ .
\end{aligned}$$

For the "only if"-direction, we have:

$$\begin{aligned}
& \text{exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' \\
& \qquad \text{and for all } q \in \mathbb{Q}_{\geq 0} \colon q' < \llbracket g \rrbracket(\sigma[v \mapsto q]) \\
\text{implies}\quad & \text{exists } q' \in \mathbb{Q}_{\geq 0} \colon \sigma(x) < q' \\
& \qquad \text{and } q' \leq \inf\{\llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\} \\
& \qquad\qquad (\text{standard property of infima}) \\
\text{implies}\quad & \sigma(x) < \inf\{\llbracket g \rrbracket(\sigma[v \mapsto q]) \mid q \in \mathbb{Q}_{\geq 0}\}\ .
\end{aligned}$$

This completes the proof. ■

Hence, we can recover f from $\mathsf{Dedekind}[f, x]$ in the following sense:

Lemma 3.11 (Syntactic Expectations from Dedekind Formulae).
Let $\mathsf{Dedekind}[f, x]$ be the Dedekind formula of $f \in \mathsf{Exp}$. We have

$$f \;\equiv\; \reflectbox{S} x\colon\, [\mathsf{Dedekind}[f, x]] \cdot x\,.$$

Proof. This follows from Lemma 3.9 and Theorem 3.10. ■

3.4.4 Sums and Products of Syntactic Expectations

This section deals with the syntactic Sum and Product expectations as described in Section 3.4.1.3. The crucial idea is to reduce reasoning about sums (resp. products) of non-negative extended reals to reasoning about sums (resp. products) of rationals as demonstrated in the previous Section 3.4.3. Formally:

Lemma 3.12 (Sums and Products of Reals via Sums of Rationals).
Let $n \in \mathbb{N}$ and let $\alpha_0, \ldots, \alpha_n \in \mathbb{R}^{\infty}_{\geq 0}$. We have:

1. $$q \in \mathsf{Cut}\left(\sum_{j=0}^{n} \alpha_j\right)$$
   iff there are $q_0 \in \mathsf{Cut}(\alpha_0), \ldots, q_n \in \mathsf{Cut}(\alpha_n)\colon\; q = \displaystyle\sum_{j=0}^{n} q_i$

2. $$q \in \mathsf{Cut}\left(\prod_{j=0}^{n} \alpha_j\right)$$
   iff there are $q_0 \in \mathsf{Cut}(\alpha_0), \ldots, q_n \in \mathsf{Cut}(\alpha_n)\colon\; q = \displaystyle\prod_{j=0}^{n} q_i$

We thus get by Lemma 3.9 that for $f \in \mathsf{Exp}$, $n \in \mathbb{N}$, $\sigma \in \mathsf{States}$, and $x \in \mathsf{Vars}$,

$$\sum_{i=0}^{n} [\![f]\!](\sigma[x \mapsto i]) \;=\; \sup\left\{ \sum_{i=0}^{n} q_i \;\middle|\; q_i \in \mathsf{Cut}([\![f]\!](\sigma[x \mapsto i])) \right\}.$$

We construct the above sums $\sum_{i=0}^{n} q_i$ of rationals in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$ using (1) Gödelization (Lemma 3.8 on page 100) and (2) Dedekind-characteristic formulae (Theorem 3.10 on page 105) to assert that $q_i \in \mathsf{Cut}([\![f]\!](\sigma[x \mapsto i]))$.

Theorem 3.13 (Sums and Products of Syntactic Expectations).
Let $f \in \mathsf{Exp}$ and let $y \in \mathsf{Vars}$ be a variable not occurring in f. We have:

1. There is an effectively constructible syntactic expectation
$$\underset{x=0}{\overset{y}{\mathsf{SUM}}}\colon f \ \in \ \mathsf{Exp}$$
such that for all $\sigma \in \mathsf{States}$, we have
$$\left[\!\!\left[\underset{x=0}{\overset{y}{\mathsf{SUM}}}\colon f \right]\!\!\right](\sigma) \ = \ \begin{cases} \sum_{i=0}^{\sigma(y)} [\![f]\!](\sigma[x \mapsto i]) & \text{if } \sigma(y) \in \mathbb{N} \\ 0 & \text{otherwise}\,. \end{cases}$$
2. There is an effectively constructible syntactic expectation
$$\underset{x=0}{\overset{y}{\mathsf{PROD}}}\colon f \ \in \ \mathsf{Exp}$$
such that for all $\sigma \in \mathsf{States}$, we have
$$\left[\!\!\left[\underset{x=0}{\overset{y}{\mathsf{PROD}}}\colon f \right]\!\!\right](\sigma) \ = \ \begin{cases} \prod_{i=0}^{\sigma(y)} [\![f]\!](\sigma[x \mapsto i]) & \text{if } \sigma(y) \in \mathbb{N} \\ 0 & \text{otherwise}\,. \end{cases}$$

Proof. We provide the construction for SUM. The construction for PROD is completely analogous. We proceed by constructing a formula $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$ with free variable z not occurring in f such that for all $\sigma \in \mathsf{States}$, we have

$$\sigma \models \mathcal{P} \quad \text{iff} \quad \sigma(y) \in \mathbb{N} \text{ and } \sigma(z) \in \mathsf{Cut}\left(\sum_{i=0}^{\sigma(y)} [\![f]\!](\sigma[x \mapsto i]) \right).$$

We then define

$$\underset{x=0}{\overset{y}{\mathsf{SUM}}}\colon f \ = \ \mathfrak{S} z\colon [\mathcal{P}] \cdot z\,,$$

which yields the desired syntactic expectation by Lemma 3.9 on page 104. It remains to construct $\mathcal{P}$. The crucial idea is to employ Lemma 3.12 and to use (1) the Dedekind-characteristic formula of f (Theorem 3.10 on page 105) and (2) the formula RElem (Lemma 3.8 on page 105) to encode the corresponding sequences of rationals in $\mathbf{A}_{\mathbb{Q}_{\geq 0}}$. Now let σ be an arbitrary state and assume in the following that all introduced bound variables do not occur in f. We have

$$\sigma(y) \in \mathbb{N} \text{ and } \sigma(z) \in \mathsf{Cut}\left(\sum_{i=0}^{\sigma(y)} [\![f]\!](\sigma[x \mapsto i]) \right)$$
$$\text{iff } \sigma(y) \in \mathbb{N} \qquad \text{(Lemma 3.12)}$$
$$\text{and exist } q_0, \ldots, q_{\sigma(y)} \in \mathbb{Q}_{\geq 0}\colon \sigma(z) = q_0 + \ldots + q_{\sigma(y)}$$

$$
\begin{aligned}
&\qquad\qquad \text{and for all } i \in \{0,\ldots,\sigma(y)\}\colon\ q_i \in \mathsf{Cut}(\llbracket f \rrbracket(\sigma[x \mapsto i])) \\
&\text{iff } \sigma(y) \in \mathbb{N} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(rewrite sum)} \\
&\quad \text{and exist } q_0,\ldots,q_{\sigma(y)} \in \mathbb{Q}_{\geq 0}\colon\ q_0 \in \mathsf{Cut}(\llbracket f \rrbracket(\sigma[x \mapsto 0])) \text{ and } \sigma(z) = q_{\sigma(y)} \\
&\qquad\qquad \text{and for all } 0 \leq i < \sigma(y)\colon \\
&\qquad\qquad\qquad \text{exists } q \in \mathsf{Cut}(\llbracket f \rrbracket(\sigma[x \mapsto i+1]))\colon\ q_{i+1} = q_i + q \\
&\text{iff } \sigma \models \underbrace{N(y)}_{\sigma(y)\in\mathbb{N}} \qquad\qquad\qquad\qquad\qquad\qquad \text{(Lemma 3.8 and Theorem 3.10)} \\
&\quad \underbrace{\wedge\, \exists v\colon \big(\exists u\colon \mathsf{Dedekind}[f,u][x/0] \wedge \mathsf{RElem}(v,0,u) \wedge \mathsf{RElem}(v,y,z)\big)}_{\text{there is a sequence } q_0,\ldots \text{ such that } q_0 \in \mathsf{Cut}_0(\llbracket f \rrbracket(\sigma[x \mapsto 0])) \text{ and } \sigma(z) = q_{\sigma(y)}} \\
&\qquad \underbrace{\wedge \forall w\colon \big(w < y \wedge N(w)\big)}_{\text{and for all } 0 \leq i < \sigma(y)\colon} \implies \underbrace{\big(\exists u,u'\colon \mathsf{Dedekind}[f,u][x/w+1]}_{\text{exists } q\in\mathsf{Cut}_0(\llbracket f \rrbracket(\sigma[x\mapsto i+1]))\colon} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\, \mathsf{RElem}(v,w,u') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \underbrace{\wedge\, \mathsf{RElem}(v,w+1,u+u')\big)}_{q_{i+1}=q_i+q}
\end{aligned}
$$

where the latter is the desired formula $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$. ■

Notice that

$$
g \;=\; \underset{x=0}{\overset{y}{\mathsf{SUM}}}\colon f \qquad \text{and} \qquad g \;=\; \underset{x=0}{\overset{y}{\mathsf{PROD}}}\colon f
$$

can be understood as a quantitative quantifier in the sense that we might have $x \in \mathsf{FV}(f)$ but $x \notin \mathsf{FV}(g)$, i.e., variable x gets bound by SUM or PROD.

Since infinite series in $\mathbb{R}_{\geq 0}^{\infty}$ evaluate to the supremum of their partial sums, we immediately obtain a construction for infinite series of syntactic expectations:

Corollary 3.14 (Infinite Series of Syntactic Expectations).
Let $f \in \mathsf{Exp}$. For every $\sigma \in \mathsf{States}$, we have

$$
\left\llbracket \mathsf{S}y\colon \underset{x=0}{\overset{y}{\mathsf{SUM}}}\colon f \right\rrbracket(\sigma) \;=\; \sum_{i=0}^{\infty} \llbracket f \rrbracket(\sigma[x \mapsto i]) \;=\; \sup_{n\in\mathbb{N}} \sum_{i=0}^{n} \llbracket f \rrbracket(\sigma[x \mapsto i])\,.
$$

Example 3.5.

1. SUM provides us with a much more convenient way to define the expectation $\mathsf{Harmonic}(x)$ from Example 3.4 on page 101:
$$\mathsf{Harmonic}(x) \;=\; \underset{v=0}{\overset{x}{\mathsf{SUM}}}\colon \mathcal{S}w\colon\; [v \geq 1 \wedge w \cdot v = 1] \cdot w\;.$$
2. Similarly, we can define the sum over squares of harmonic numbers:
$$\underset{v=0}{\overset{x}{\mathsf{SUM}}}\colon \mathcal{S}w\colon\; [v \geq 1 \wedge w \cdot v = 1] \cdot w \cdot w\;.$$
Since the corresponding infinite series evaluates to $\pi^2/6$, we obtain a syntactic expectation evaluating to a *transcendental number*:
$$\underset{v=0}{\overset{\infty}{\mathsf{SUM}}}\colon \mathcal{S}w\colon\; [v \geq 1 \wedge w \cdot v = 1] \cdot w \cdot w \quad \text{evaluates to} \quad \frac{\pi^2}{6}\;.$$

3.4.5 Expressiveness of Syntactic Expectations

Gluing the results from the previous sections together, we give a constructive expressiveness proof for our language Exp. Given arbitrary, but fixed, $C \in \mathsf{pGCL}$ and $f \in \mathsf{Exp}$, our goal is to effectively construct $g \in \mathsf{Exp}$ such that

$$[\![g]\!] \;=\; \mathsf{wp}[\![C]\!]\,([\![f]\!])\;.$$

Given $n \in \mathbb{N}$, we write $\mathbf{x} = (x_0, \ldots, x_{n-1})$ to indicate that $\mathbf{x}$ is a (possibly empty) n-tuple of pairwise distinct variables. Moreover, we denote by $\mathsf{States}_{\mathbf{x}} \subseteq \mathsf{States}$ an arbitrary, but fixed, set of states that contains *exactly one* state from each equivalence class of $\sim_{\mathbf{x}}$ (cf. Section 3.4.2.4). Moreover, given a state $\sigma \in \mathsf{States}$, we define the *characteristic expectation of σ (w.r.t. $\mathbf{x}$)* as

$$[\sigma]_{\mathbf{x}} \;=\; \left[\bigwedge_{i=0}^{n-1} x_i = \sigma(x_i)\right]. \qquad \text{(adapted from [Win93])}$$

Expectation $[\sigma]_{\mathbf{x}}$ evaluates to 1 on state σ' if $\sigma \sim_{\mathbf{x}} \sigma'$, and to 0 otherwise.

Denote by $\mathsf{Vars}(C) \subset \mathsf{Vars}$ the finite set of variables occurring in C. We formalize the characterization of weakest preexpectations of loops from Section 3.4.1.3.

Theorem 3.15 (Small-Step Characterization of Loops).
Let $C = \mathtt{while}(\varphi)\{C'\}$ be a loop and let $f \in \mathsf{Exp}$. Moreover, let $n \geq 1$ and

$\mathbf{x} = (x_0, \ldots, x_{n-1})$ with $\{x_0, \ldots, x_{n-1}\} = \mathsf{Vars}(C) \cup \mathsf{FV}(f)$. We have

$$\begin{aligned}
&\mathsf{wp}[\![\texttt{while}(\varphi)\{C'\}]\!]([\![f]\!])\\
=\ &\lambda\sigma.\ \sup_{k\in\mathbb{N}} \sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}_{\mathbf{x}}} [\sigma_0]_{\mathbf{x}}(\sigma)\cdot[\![[\neg\varphi]\cdot f]\!](\sigma_k)\\
&\qquad\cdot\prod_{i=0}^{k-1}\mathsf{wp}[\![\texttt{if}\ (\varphi)\ \{C'\}\ \texttt{else}\ \{\texttt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)\ .
\end{aligned}$$

Proof. See [8, Appendix D]. ■

We encode $\sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}_{\mathbf{x}}}$ in Exp by summing over all Gödel numbers of state sequences of length k (cf. Section 3.4.2.4) using SUM (Theorem 3.13 on page 107). It remains to show how to mimic substitutions and evaluations of syntactic expectations in program states given as Gödel numbers:

Lemma 3.16 (Gödelized Substitutions of Variables).
Let $f \in \mathsf{Exp}$, $x \in \mathsf{Vars}$, $n \in \mathbb{N}$, $\mathbf{x} = (x_0, \ldots, x_{n-1})$, and let $\mathbf{y} = (y_0, \ldots, y_{n-1})$. There is an effectively constructible syntactic expectation

$$\mathsf{Subst}_{\mathbf{x}}[f, x, y_0, \ldots, y_{n-1}] \in \mathsf{Exp}$$

such that for all $\sigma, \sigma' \in \mathsf{States}$ with $\sigma(x) = \langle\sigma'\rangle_{\mathbf{x}}$, we have

$$[\![\mathsf{Subst}_{\mathbf{x}}[f, x, y_0, \ldots, y_{n-1}]]\!](\sigma) \;=\; [\![f]\!](\sigma[y_0 \mapsto \sigma'(x_1), \ldots, y_{n-1} \mapsto \sigma'(x_{n-1})])\ .$$

Proof. Let $v_0, \ldots, v_{n-1}$ be pairwise distinct fresh variables. We construct

$$\begin{aligned}
&\mathsf{Subst}_{\mathbf{x}}[f, x, y_0, \ldots, y_{n-1}]\\
=\ &\mathcal{S} v_0\colon \ldots \mathcal{S} v_{n-1}\colon [\mathsf{RElem}(x, 0, v_0) \wedge \ldots \wedge \mathsf{RElem}(x, n-1, v_{n-1})]\\
&\quad\cdot(f[y_0/v_0]\ldots[y_{n-1}/v_{n-1}])\ .
\end{aligned}$$

■

Evaluating some f in some state given as a Gödel number then simply corresponds to substituting all free variables of f by their respective values.

Corollary 3.17 (Gödelized Evaluation of Syntactic Expectations).
Let $f \in \mathsf{Exp}$, $x \in \mathsf{Vars}$, $n \in \mathbb{N}$, and let $\mathbf{x} = (x_0, \ldots, x_{n-1})$ with $\{x_0, \ldots, x_{n-1}\} \supseteq \mathsf{FV}(f)$. There is an effectively constructible syntactic expectation

$$\mathsf{Apply}_{\mathbf{x}}[f, x] \in \mathsf{Exp}$$

such that for all $\sigma, \sigma' \in \mathsf{States}$ with $\sigma(x) = \langle \sigma' \rangle_{\mathbf{x}}$, we have

$$[\![\mathsf{Apply}_{\mathbf{x}}[f, x]]\!](\sigma) \;=\; [\![f]\!](\sigma') .$$

Proof. We construct

$$\mathsf{Apply}_{\mathbf{x}}[f, x] \;=\; \mathsf{Subst}_{\mathbf{x}}[f, x, x_0, \ldots, x_{n-1}] . \qquad \blacksquare$$

Finally, we require the following auxiliary result to express the probability

$$\mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)$$

of reaching state σ_{i+1} from σ_i by one guarded loop iteration.

Lemma 3.18.
Let $C \in \mathsf{pGCL}$, $n \in \mathbb{N}$, $\mathbf{x} = (x_0, \ldots, x_{n-1})$, and $\mathbf{y} = (y_0, \ldots, y_{n-1})$ with $\{x_0, \ldots, x_{n-1}\} \supseteq \mathsf{Vars}(C)$ and $\mathbf{x}, \mathbf{y}$ disjoint. For all $\sigma, \sigma' \in \mathsf{States}$, we have

$$\begin{aligned} &\mathsf{wp}[\![C]\!]([\sigma']_{\mathbf{x}})(\sigma) \\ =\; &\mathsf{wp}[\![C]\!]([x_0 = y_0 \wedge \ldots \wedge x_{n-1} = y_{n-1}]) \\ &\qquad\qquad (\sigma([y_0 \mapsto \sigma'(x_0), \ldots, y_{n-1} \mapsto \sigma'(x_{n-1})])) . \end{aligned}$$

Proof. By induction on C. $\blacksquare$

We are now in a position to prove expressiveness of Exp.

Theorem 3.19 (Expressiveness of Exp).
The language Exp of syntactic expectations is expressive, i.e.,

$$\text{for all } C \in \mathsf{pGCL}, f \in \mathsf{Exp}\colon \text{exists } g \in \mathsf{Exp}\colon \quad \mathsf{wp}[\![C]\!]([\![f]\!]) \;=\; [\![g]\!] .$$

Moreover, g is effectively constructible.

Proof. By induction on the structure of C. All cases except loops are completely analogous to the proof of Theorem 3.2 on page 90.

The case $C = \mathtt{while}(\varphi)\{C'\}$. Let $\mathbf{x} = (x_0, \ldots, x_{n-1})$ with $\{x_0, \ldots, x_{n-1}\} = \mathsf{Vars}(C) \cup \mathsf{FV}(f)$. Moreover, let $\mathbf{y} = (y_0, \ldots, y_{n-1})$ with $\mathbf{x}, \mathbf{y}$ disjoint. By the I.H., there is an effectively constructible syntactic expectation $h \in \mathsf{Exp}$ with

$$[\![h]\!] \;=\; \mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([x_0 = y_0 \wedge \ldots \wedge x_{n-1} = y_{n-1}]) ,$$

Now assume in the following that all introduced bound variables are fresh.

We effectively construct the desired syntactic expectation g by encoding the small-step characterization of $\mathsf{wp}[\![C]\!]([\![f]\!])$ from Theorem 3.15 on page 110:

$$\underbrace{\reflectbox{S}\, v\colon}_{=\lambda\sigma.\,\sup_{k\in\mathbb{N}}} \quad \underbrace{\overset{\infty}{\underset{z=0}{\mathsf{SUM}}}\colon\ [\mathsf{StateSequence}_{\mathbf{x}}(z,v+1)]}_{\sum_{\sigma_0,\ldots,\sigma_k\in\mathsf{States}_{\mathbf{x}}}[\sigma_0]_{\mathbf{x}}(\sigma)}$$

$$\underbrace{\cdot\,(\reflectbox{S}\, w\colon\ [\mathsf{Elem}(z,v,w)]\cdot\mathsf{Apply}_{\mathbf{x}}[[\neg\varphi]\cdot f, w])}_{\cdot[\![[\neg\varphi]\cdot f]\!](\sigma_k)}$$

$$\underbrace{\begin{array}{l}\cdot\Big([v=0]\cdot 1 \\ \quad +[v>0]\cdot\overset{v\dot{-}1}{\underset{z'=0}{\mathsf{PROD}}}\colon\ \reflectbox{S}\, w_1,w_2\colon\ [\mathsf{Elem}(z,z',w_1)\wedge\mathsf{Elem}(z,z'+1,w_2)] \\ \qquad\qquad\qquad\qquad \underbrace{\cdot\,\mathsf{Apply}_{\mathbf{x}}[\mathsf{Subst}_{\mathbf{x}}[h,w_2,y_0,\ldots,y_{n-1}],w_1]\Big)}_{\mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)}\end{array}}_{\cdot\prod_{i=0}^{k-1}\mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)}$$

Now let $\sigma_0\in\mathsf{States}$ be some initial program state. To see that the above construction is sound, recall from Section 3.4.2.4 that

$$\begin{array}{ll} & [\![[\mathsf{StateSequence}_{\mathbf{x}}(z,v+1)]]\!](\sigma_0) \;=\; 1 \\ \text{iff} & \sigma_0(v)+1=k \text{ for some } k\in\mathbb{N} \\ & \text{and } \sigma_0(z) \text{ is the unique (modulo } \sim_{\mathbf{x}}\text{) Gödel number of some} \\ & \quad \text{sequence } \sigma_0,\ldots,\sigma_k \text{ of states of length } k\,. \end{array}$$

Now assume that $[\![[\mathsf{StateSequence}_{\mathbf{x}}(z,v+1)]]\!](\sigma_0)=1$ and let $\sigma_0,\ldots,\sigma_k$ be the corresponding sequence of states. Then, by Corollary 3.17, we have

$$[\![\reflectbox{S}\, w\colon\ [\mathsf{Elem}(z,v,w)]\cdot\mathsf{Apply}_{\mathbf{x}}[[\neg\varphi]\cdot f,w]]\!](\sigma_0) \;=\; [\![[\neg\varphi]\cdot f]\!](\sigma_k)$$

since the $\reflectbox{S}\, w$ quantifier selects the Gödel number of σ_k. Moreover, if $i=\sigma_0(z')\in\{0,\ldots,k-1\}$, then

$$\begin{array}{l}[\![\reflectbox{S}\, w_1,w_2\colon\ [\mathsf{Elem}(z,z',w_1)\wedge\mathsf{Elem}(z,z'+1,w_2)] \\ \qquad\qquad \cdot\mathsf{Apply}_{\mathbf{x}}[\mathsf{Subst}_{\mathbf{x}}[h,w_2,y_0,\ldots,y_{n-1}],w_1]]\!](\sigma_0)\end{array}$$

$$
\begin{aligned}
&= [\![\mathsf{Apply}_{\mathbf{x}}[\mathsf{Subst}_{\mathbf{x}}[h, num_2, y_0, \ldots, y_{n-1}], num_1]]\!](\sigma_0) \\
&\text{(where } num_1 \text{ is Gödel number of } \sigma_i \text{ and } num_2 \text{ is Gödel number of } \sigma_{i+1}) \\
&= [\![\mathsf{Subst}_{\mathbf{x}}[h, num_2, y_0, \ldots, y_{n-1}]]\!](\sigma_i) && \text{(Corollary 3.17)} \\
&= [\![h]\!](\sigma_i[y_0 \mapsto \sigma_{i+1}(x_1), \ldots, y_{n-1} \mapsto \sigma_{i+1}(x_{n-1})]) && \text{(Lemma 3.16)} \\
&= \mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([x_0 = y_0 \land \ldots \land x_{n-1} = y_{n-1}]) \\
&\qquad (\sigma_i[y_0 \mapsto \sigma_{i+1}(x_1), \ldots, y_{n-1} \mapsto \sigma_{i+1}(x_{n-1})]) && \text{(I.H.)} \\
&= \mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)\,. && \text{(Lemma 3.18)}
\end{aligned}
$$

Finally, since for each $k \in \mathbb{N}$ and all sequences $\sigma_0, \ldots, \sigma_k$ of states in $\mathsf{States}_{\mathbf{x}}$, there is *exactly one* Gödel number num such that

$$
[\![[\mathsf{StateSequence}_{\mathbf{x}}(num, k+1)]]\!](\sigma_0) = 1\,,
$$

we get by Theorem 3.13 on page 102 (soundness of Sum and Product) that

$$
\begin{aligned}
[\![g]\!] = \lambda\sigma.\, \sup_{k\in\mathbb{N}} \sum_{\sigma_0,\ldots,\sigma_k \in \mathsf{States}_{\mathbf{x}}} & [\sigma_0]_{\mathbf{x}}(\sigma) \cdot [\![[\neg\varphi] \cdot f]\!](\sigma_k) \\
& \cdot \prod_{i=0}^{k-1} \mathsf{wp}[\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]([\sigma_{i+1}]_{\mathbf{x}})(\sigma_i)\,.
\end{aligned}
$$

and hence

$$
[\![g]\!] = \mathsf{wp}[\![C]\!]([\![f]\!])
$$

by Theorem 3.15. ■

3.5 Discussion

In what follows, we discuss several consequences of Theorem 3.19.

3.5.1 Relative Completeness of Probabilistic Program Verification

An immediate consequence of Theorem 3.19 is that for all pGCL programs C and all syntactic expectations $f, g \in \mathsf{Exp}$, verifying the bounds

$$
[\![g]\!] \sqsubseteq \mathsf{wp}[\![C]\!]([\![f]\!]) \qquad \text{or} \qquad \mathsf{wp}[\![C]\!]([\![f]\!]) \sqsubseteq [\![g]\!]
$$

reduces to *checking a single inequality* between two syntactic expectations in Exp, namely the given g and the *effectively constructible syntactic expectation* for $\mathsf{wp}[\![C]\!]([\![f]\!])$. In that sense, the wp calculus together with Exp form a *relatively complete* system in the sense of Cook [Coo78] for probabilistic program verification: Given an oracle for discharging inequalities between syntactic expectations, every correct inequality of the above form can be derived.

It is, however, important to note that Theorem 3.19 *does not imply decidability* of the above bounds. To see this, we first observe that *quantitative entailments* between syntactic expectations are undecidable.

Theorem 3.20 (Undecidability of Quantitative Entailments in Exp). Quantitative entailments in Exp are undecidable, i.e.,

Given $h_1, h_2 \in \mathsf{Exp}$, does $[\![h_1]\!] \sqsubseteq [\![h_2]\!]$ hold?

is undecidable.

Proof. Assume the contrary. Now consider the decision problem UAST:

Given fully probabilistic $C \in \mathsf{pGCL}$,
does C terminate almost-surely on each initial state?

It follows that UAST is decidable because we have

$$\begin{aligned} & C \text{ terminates almost-surely on each initial state} \\ \text{iff}\quad & [\![1]\!] \sqsubseteq \mathsf{wp}[\![C]\!]([\![1]\!]) \\ \text{iff}\quad & [\![1]\!] \sqsubseteq [\![h]\!]~, \end{aligned}$$

where $h \in \mathsf{Exp}$ is the effectively constructible syntactic expectation with

$$[\![h]\!] ~=~ \mathsf{wp}[\![C]\!]([\![1]\!])~.$$

This contradicts the undecidability of UAST [KKM19; KK15]. ■

Hence, we immediately get the following undecidability result:

Corollary 3.21 (Undecidability of Bounds on Expected Outcomes). Both of the decision problems

Given $C \in \mathsf{pGCL}$ and $f, g \in \mathsf{Exp}$, does $[\![g]\!] \sqsubseteq \mathsf{wp}[\![C]\!]([\![f]\!])$ hold?

and

Given $C \in \mathsf{pGCL}$ and $f, g \in \mathsf{Exp}$, does $\mathsf{wp}[\![C]\!]([\![f]\!]) \sqsubseteq [\![g]\!]$ hold?

are undecidable.

Proof. Assume for a contradiction that the first problem is decidable. Then quantitative entailments in Exp are decidable because for $h_1, h_2 \in \mathsf{Exp}$,

$$\begin{aligned} & [\![h_1]\!] \sqsubseteq [\![h_2]\!] \\ \text{iff} \quad & [\![h_1]\!] \sqsubseteq \mathsf{wp}[\![\texttt{skip}]\!]([\![h_2]\!]) \, , && \text{(Table 2.1)} \end{aligned}$$

which contradicts Theorem 3.20. The reasoning for the second problem is completely analogous. ■

Relative Completeness of Invariant-Based Reasoning for Loops. Recall from Section 2.4.4 that upper bounds on weakest preexpectations of loops can be verified in an invariant-based manner, i.e, for $C = \texttt{while}(\varphi)\{C\}$ and $X, I, \in \mathbb{E}$,

$$\underbrace{I \text{ is wp-superinvariant of } C \text{ w.r.t. } X}_{{}^{\mathsf{wp}}_{C}\Phi_X(I) \sqsubseteq I \text{ (cf. Definition 2.21 on page 63)}} \qquad \text{implies} \qquad \mathsf{wp}[\![C]\!](X) \;\sqsubseteq\; I \, .$$

From an extensional perspective, it is rather obvious that invariant-based reasoning is *complete* in the sense that for *every* $Y \in \mathbb{E}$ with

$$\mathsf{wp}[\![C]\!](X) \;\sqsubseteq\; Y \, ,$$

there is a wp-superinvariant witnessing this fact, namely $I = \mathsf{wp}[\![C]\!](X)$, i.e.,

$$\text{exists wp-superinvariant } I \in \mathbb{E} \text{ of } C \text{ w.r.t. } X\colon \quad \mathsf{wp}[\![C]\!](X) \;\sqsubseteq\; I \;\sqsubseteq\; Y \, .$$

By Theorem 3.19, this completeness result carries over to the *intensional* approach of probabilistic program verification. For *every* $f, g \in \mathsf{Exp}$ with

$$\mathsf{wp}[\![C]\!]([\![f]\!]) \;\sqsubseteq\; [\![g]\!] \, ,$$

there is a wp-superinvariant of C w.r.t. $[\![f]\!]$ expressible in Exp witnessing this fact, namely the effectively constructible $\mathsf{I} \in \mathsf{Exp}$ with $[\![\mathsf{I}]\!] = [\![\mathsf{wp}[\![C]\!]([\![f]\!])]\!]$.

3.5.2 Termination Probabilities

For each pGCL program C, the weakest preexpectation

$$\mathsf{wp}[\![C]\!]([\![1]\!])$$

maps each initial state σ to the probability that C terminates on σ. Hence, for *every* pGCL program C, there is an effectively constructible syntactic expectation *evaluating to C's termination probability for each initial state σ.*

Our syntax is thus capable of expressing functions from states to numbers that carry a *high degree of internal complexity* [KK15; KKM19]. More concretely, given $C \in \mathsf{pGCL}$, $\sigma \in \mathsf{States}$, and $p \in [0,1] \cap \mathbb{Q}$, deciding whether C terminates *at least* with probability p on initial state σ is Σ_1^0-complete in the arithmetical hierarchy. Deciding whether C terminates *at most* with probability p on initial σ is even Π_2^0–complete, thus strictly harder than, e.g., the universal termination problem for non-probabilistic programs.

3.5.3 Reachability Probabilities

For a probabilistic program C and a predicate expressed as a formula $\mathcal{P} \in \mathbf{A}_{\mathbb{Q}_{\geq 0}}$,

$$\mathsf{wp}[\![C]\!]([\![[\mathcal{P}]]\!])$$

maps each initial state σ to the probability that C terminates in some final state $\tau \models \mathcal{P}$. Since $[\mathcal{P}]$ is expressible in Exp, we have that $\mathsf{wp}[\![C]\!]([\![[\mathcal{P}]]\!])$ is also expressible in Exp by Theorem 3.19. A further consequence is that we *embed and generalize Dijkstra's weakest preconditions [Dij75; Dij76]* since for non-probabilistic[9] C, the weakest pre*condition* of C w.r.t. postcondition $\mathcal{P}$ is

$$\{\sigma \in \mathsf{States} \mid \mathsf{wp}[\![C]\!]([\![[\mathcal{P}]]\!])(\sigma) = 1\} .$$

3.5.4 Distribution of Final States

Let C be a pGCL program in which at most the variables $x_1, \ldots, x_k$ occur and recall that $[\![C]\!]^\sigma$ is the subdistribution of final states obtained from executing C on initial state σ (cf. Definition 2.18 on page 50). Then, by the Kozen duality (Corollary 2.16.3 on page 72), the probability $[\![C]\!]^\sigma(\tau)$ of C terminating in final state τ on initial state σ, where $\tau(x_i) = \sigma(x_i')$, is given by

$$[\![C]\!]^\sigma(\tau) \;=\; \mathsf{wp}[\![C]\!]\Big(\Big[x_1 = x_1' \wedge \cdots \wedge x_k = x_k'\Big]\Big)(\sigma) ,$$

i.e., $\sigma(x_1), \ldots, \sigma(x_k)$ are the initial values of $x_1, \ldots, x_k$ and $\sigma(x_1'), \ldots, \sigma(x_k')$ are the final values. Hence, the function

$$\lambda(\sigma, \tau).\, [\![C]\!]^\sigma(\tau) ,$$

[9] i.e., C does not contain a probabilistic choice

mapping each pair (σ, τ) of states (w.r.t. $x_1, \ldots, x_k$) to the probability $[\![C]\!]^{\sigma}(\tau)$ that C terminates in τ when executed on σ is expressible in Exp.

3.5.5 Ranking Functions and Supermartingales

There is a plethora of methods for proving termination of probabilistic programs based on ranking supermartingales [CS13; FH15; CFNH16; CNZ17; HFC18; FC19; HFCG19; AGR21; TOUH21; TOUH18]. Ranking supermartingales are similar to ranking functions, but one requires that the value decreases *in expectation*. Weakest preexpectations are a natural formalism to reason about this.

For algorithmic solutions, ranking supermartingales are often assumed to be, for instance, linear [CFNH18] or polynomial [CFG16; SO19; NCH18]. This also often applies to the allowed shape of templates for loop invariants in works on the automated synthesis of probabilistic loop invariants (cf. [KMMM10; FZJZ+17; BEFH16; CS13; BTPH+22]). *Functions linear or polynomial in the program variables are subsumed by our syntax.*

3.6 Future and Related Work

Future Work. We identify several directions for future work. First, we restricted to *fully probabilistic* programs not containing nondeterministic choices. We conjecture that Exp is also expressive for demonic and angelic weakest preexpectations, i.e., that for $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$, we have

$$\begin{aligned}&\text{for all possibly nondeterministic } C \in \mathsf{pGCL} \text{ and } f \in \mathsf{Exp}\colon\\ &\qquad \text{exists } g \in \mathsf{Exp}\colon \quad [\![g]\!] \;=\; \mathcal{T}[\![C]\!]([\![f]\!])~.\end{aligned}$$

The proof presented here does not straightforwardly generalize to nondeterministic programs since Theorem 3.15 on page 110 does generally not hold in the presence of nondeterminism. Moreover, we also conjecture that Exp is expressive for the ert-calculus [KKMO16; KKMO18] — a weakest preexpectation-style calculus for reasoning about expected runtimes.

A related direction for future work is to develop an expressive language for *quantitative separation logic (QSL)* [5] — a conservative extension of Reynold's separation logic [Rey02] for reasoning about probabilistic *pointer* programs. QSL requires reasoning about both nondeterminism and dynamic memory. We might benefit from existing works on expressive assertion languages for Reynold's separation logic (cf. paragraph on related work below).

Related Work. Relative completeness of Hoare logic was shown by Cook [Coo78]. Cook has proven expressiveness of first-order arithmetic for strongest postconditions. Clarke [Cla76] and Olderog [Old80] have proven that expressiveness first-order arithmetic for strongest postconditions is equivalent to expressiveness of first-order arithmetic for Dijkstra's weakest preconditions. Many techniques for proving expressiveness of Exp presented here are inspired by the direct expressiveness proofs for weakest preconditions given in [Win93] and [LSS84]. We recommend the works by Apt [AO19; Apt81] for excellent surveys on Hoare logic and related aspects of the logic's completeness.

For separation logic [Rey02] — a prominent logic for compositional reasoning about *pointer programs* — expressiveness was shown in [TCA09; TCA19], almost a decade later than the logic was originally developed and started to be used.

Perhaps most directly related to this chapter is the work by [HV02] on a Hoare-like logic for verifying probabilistic programs. They prove relative completeness (also in the sense of [Coo78]) of their logic for *loop-free* probabilistic programs and *restricted postconditions*; they leave expressiveness for loops as an open problem: "It is not clear whether the probabilistic predicates are sufficiently expressive [. . .] for a given while loop."

4 Latticed k-Induction

This chapter is based on our prior publications [10; 11].

In this chapter, we revisit two well-established verification techniques for transition systems, *k-induction* and *bounded model checking (BMC)*, in the more general setting of fixpoint theory over complete lattices (cf. Section 2.1). We present *latticed k-induction*, which (i) generalizes classical k-induction for verifying invariant properties of transition systems, (ii) generalizes Park induction for bounding fixpoints of monotone functions on complete lattices, and (iii) extends from naturals k to transfinite ordinals κ, thus yielding *κ-induction*.

4.1 Preview: Automatic Verification of Loops

The aforementioned fixpoint-theoretic understanding of k-induction and BMC enables us to apply both techniques to the *fully automatic verification of bounds on expected outcomes of possibly unbounded probabilistic loops*. Our implementation manages to automatically verify specifications of programs taken from the literature without requiring user-provided verification hints such as invariant annotations. Let us consider an introductory example.

k-Induction. Recall the program C modeling a variant of the bounded retransmission protocol [HSV93; DJJL01] from Example 2.20 on page 56:

```
while($sent < N \wedge failed \leq M$){
    {
        $failed := failed + 1 ; totalFailed := totalFailed + 1$
    } [0.1] {
        $failed := 0 ; sent := sent + 1$
    }
}
```

We have introduced an additional variable *totalFailed* to keep track of the total number of retransmissions until the protocol succeeds (i.e., terminates with $sent = N$) or fails (i.e., terminates with $failed = M$). We now aim to verify the following specification of C:

If there are at most 4 packets to send, i.e., if $N \leq 4$,
then totalFailed increases by at most $1/2$ on average.

In terms of weakest preexpectations, we thus aim to verify that

$$\mathsf{wp}[\![C]\!]\,(totalFailed) \;\sqsubseteq\; [N \leq 4] \cdot (totalFailed + 1/2) + [N > 4] \cdot \infty\,. \tag{4.1}$$

By Theorem 2.12 on page 66, we can try to verify this bound by checking whether it is a wp-superinvariant of C w.r.t. *totalFailed*, i.e., whether for

$$I \;=\; [N \leq 4] \cdot (totalFailed + 1/2) + [N > 4] \cdot \infty$$

we have

$$\underbrace{{}^{\mathsf{wp}}_{\;C}\Phi_{totalFailed}(I) \sqsubseteq I}_{\text{does not hold}} \quad \text{which would imply} \quad \underbrace{\mathsf{wp}[\![C]\!]\,(totalFailed) \sqsubseteq I}_{\text{holds}}\,.$$

However, even though (4.1) holds, I is *not* a wp-superinvariant of C w.r.t. *totalFailed*, i.e., Park induction does not apply to I: As it is the case for transition systems (cf. Example 2.12 on page 30), the converse direction of Theorem 2.12 does generally not hold. k-induction for probabilistic programs can mitigate this problem: We obtain a class of increasingly powerful proof rules — namely 1-induction, 2-induction, 3-induction, and so on — where 1-induction corresponds to Park induction (Theorem 2.12), and where for $k \geq 2$, k-induction is *strictly stronger* than Park induction. Moreover, k-induction is amenable to automation: By leveraging satisfiability modulo theory (SMT) techniques, our tool kipro2 fully automatically verifies that (4.1) holds by means of 5-induction.

Notice that, even though we bound the number of packets to send, we do *not* bound the number M of maximal retransmissions. Hence, kipro2 automatically solves an *infinite-state verification problem*: We have to reason about infinitely many initial program states in order to verify the given specification. In terms of pGCL's operational MDP $\mathcal{O}$ (cf. Definition 2.17 on page 50), the reachable fragment of configurations we have to reason about is infinite. Hence, finite-state probabilistic model checkers such as Storm [HJKQ+22; DJKV17] are not applicable to such verification problems.

Bounded Model Checking. BMC for probabilistic programs complements k-induction. With BMC, we can automatically *refute* upper bounds on expected outcomes of loops. For the program C from above, we might conjecture:

No matter how many packets there are to be sent,
totalFailed increases by at most 1/2 on average.

That is, we drop the assumption that there are at most 4 packets to send. In terms of weakest preexpectations, this specification reads

$$\mathsf{wp}[\![C]\!](\mathit{totalFailed}) \sqsubseteq \mathit{totalFailed} + 1/2 \;. \tag{4.2}$$

With BMC, our tool kipro2 automatically refutes this property and moreover provides us with a concrete initial program state σ — corresponding to an initial configuration of the bounded retransmission protocol — such that

$$\mathsf{wp}[\![C]\!](\mathit{totalFailed})(\sigma) > \sigma(\mathit{totalFailed}) + 1/2 \qquad \text{and} \qquad \sigma(N) = 8, \sigma(M) = 1 \;.$$

Hence, if there are 8 packets to send while allowing for at most 1 retransmission per packet, then *totalFailed* is expected to increase by more than 1/2.

Notice again that we tackle here an *infinite-state refutation problem*: Under *all, infinitely many* potential initial program states — each corresponding to an initial configuration of the bounded retransmission protocol — that might violate (4.2), our tool automatically finds one for which this is indeed the case.

Latticed k-Induction and BMC. k-induction and BMC for probabilistic programs are, in fact, instances of our more general latticed techniques. Latticed k-induction and BMC apply to *all* verification problems that can be phrased as upper-bounding the least fixpoint of a monotone function over a complete lattice, including, e.g., upper-bounding expected rewards in MDPs (cf. Section 2.2). Our latticed techniques have moreover been leveraged by Winkler and Katoen [WK23a] in the context of probabilistic pushdown automata and by Yang et al. [YFKZ+24] for the synthesis of quantitative loop invariants.

Chapter Outline. This chapter is structured as follows: In Section 4.2, we recap k-Induction and BMC for transition systems and exemplify how these techniques are employed in the field of software model checking. In Section 4.3, we present our theory of latticed k-induction. Section 4.4 treats the relation between classical k-induction for transition systems and latticed k-induction. We present latticed bounded model checking in Section 4.2.1. In Section 4.6, we instantiate latticed k-induction and BMC for the verification of probabilistic programs, demonstrate how the so-obtained theory yields algorithms for the automatic verification of probabilistic loops, and discuss empirical results of our implementation. Finally, in Section 4.7, we discuss future and related work.

4.2 Reasoning about Safety of Transition Systems

Reasoning about safety of hardware and software systems [MP95] often boils down to verifying or refuting so called *invariant properties* [BK08, Chapter 3] of possibly infinite-state transition systems (cf. Section 2.1.1):

> Given a transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ and a set $P \subseteq \mathcal{S}$ of states, *verify* or *refute* that $\mathsf{Reach}(\mathsf{TS}) \subseteq P$ holds.

Adopting LTL-style terminology [BK08, Chapter 3], we refer to P as the invariant property and say that P *holds for* TS, if P covers all reachable states of TS. Prime examples include, e.g., verifying or refuting whether a given program can crash due to null pointer dereferences [CDDG⁺15]. In what follows, we recap two prominent approaches for reasoning about invariant properties. We start with *bounded model checking* for *refuting* invariant properties in Section 4.2.1. We then consider k-induction for *verifying* them in Section 4.2.2.

4.2.1 Bounded Model Checking

Let $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ be a transition system and let $P \subseteq \mathcal{S}$ be an invariant property. The essence of bounded model checking (BMC, for short) — a method introduced by Clarke et al. [CBRZ01] — is to perform a breadth-first search in the transition system TS. We first check whether P covers all states in $\mathsf{Reach}^{\leq 0}(\mathsf{TS})$ — the initial states. If not, the invariant property is refuted. Otherwise, we continue with $\mathsf{Reach}^{\leq 1}(\mathsf{TS})$ — states reachable in at most 1 step — and check whether $\mathsf{Reach}^{\leq 1}(\mathsf{TS}) \subseteq P$, and so on. Whenever we determine that $\mathsf{Reach}^{\leq k}(\mathsf{TS}) \subseteq P$ for some $k \in \mathbb{N}$, we thus verified that P covers all states reachable within k steps (hence the word "bounded"). If, for some k, we determine that this is *not* the case, i.e., if $\mathsf{Reach}^{\leq k}(\mathsf{TS}) \not\subseteq P$, we have refuted that P holds for TS since it follows that there is some reachable state $s \in \mathsf{Reach}(\mathsf{TS})$ with $s \notin P$. In case P does *not* hold for TS, the existence of such a k is guaranteed, which makes BMC particularly suitable for refuting invariant properties.

We adapt the typical formulation of BMC as follows:

Theorem 4.1 (Classical Bounded Model Checking [CBRZ01]).
Let $k \in \mathbb{N}$. We have

$$\text{for all } s_0, \ldots, s_k \in \mathcal{S}: \quad s_0 \in \mathcal{S}_I \wedge \bigwedge_{i=1}^{k} s_{i-1} \longrightarrow s_i \qquad \Longrightarrow \qquad \bigwedge_{i=0}^{k} s_i \in P$$

iff P covers all states of TS reachable in at most k steps, i.e.,

$$\mathsf{Reach}^{\leq k}(\mathsf{TS}) \subseteq P .$$

BMC was originally introduced as a symbolic method based on suitable encodings of transition systems as propositional formulae (consider, e.g., [Lan18, Example 3.2]). By encoding the set $\mathcal{S}$ of states, the transition relation $\longrightarrow$, and the set $\mathcal{S}_I$ of initial states as propositional formulae, checking $\mathsf{Reach}^{\leq k}(\mathsf{TS}) \subseteq P$ can be encoded equivalently as the validity a propositional formula of the form

$$s_0 \in \mathcal{S}_I \wedge \bigwedge_{i=1}^{k} s_{i-1} \longrightarrow s_i \qquad \Longrightarrow \qquad \bigwedge_{i=0}^{k} s_i \in P ,$$

which is then offloaded to a *Satisfiability solver* (*SAT solver*, for short) — tools deciding the formula's validity[1]. Since we do not treat such encodings in this thesis, suffice it to say here that these encodings are particularly suited for modeling hardware circuits and often allow for concise representations of huge transition systems. We exemplify how such encodings are employed in the field of software model checking in Section 4.2.3, which is similar to the aforementioned SAT-based approach for reasoning about hardware circuits.

4.2.2 k-Induction

Whereas BMC is suitable for refuting invariant properties, k-induction — a method introduced by Sheeran et al. [SSS00] — is a method for *verifying* them. As with BMC, k-induction was originally introduced as a symbolic method based on suitable encodings of transition systems as propositional formulae. Let $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ be a transition system and let $P \subseteq \mathcal{S}$ be an invariant property. We adapt the typical formulation of k-induction [DHKR11; JD16; GI17]:

[1] SAT solvers actually decide a formula's satisfiability but this subsumes validity checking since a formula is valid iff its negation is unsatisfiable.

Theorem 4.2 (Classical *k*-Induction [SSS00]).
Let $k \geq 1$. If for all states $s_1, \ldots, s_{k+1} \in \mathcal{S}$, we have

$$s_1 \in \mathcal{S}_I \wedge \bigwedge_{i=1}^{k-1} s_i \longrightarrow s_{i+1} \quad \Longrightarrow \quad \bigwedge_{i=1}^{k} s_i \in P\,, \tag{4.3a}$$

$$s_1 \in P \wedge \left(\bigwedge_{i=1}^{k-1} s_i \longrightarrow s_{i+1} \wedge s_{i+1} \in P \right) \wedge s_k \longrightarrow s_{k+1} \quad \Longrightarrow \quad s_{k+1} \in P\,, \tag{4.3b}$$

then P covers all reachable states of TS, i.e.,

$$\mathsf{Reach}(\mathsf{TS}) \subseteq P\,.$$

Recall from Definition 2.1 on page 12 that we assume the transition relation $\longrightarrow$ to be *total*, which is crucial for the above theorem to be sound. If both of the above conditions hold for some given $k \geq 1$, we say that P is *k-inductive*.

Condition (4.3a) is the *base case* asserting that all states reachable within $k-1$ transition steps from some initial state are in P; Condition (4.3b) is the *induction step* asserting that if we start in P and stay in P for $k-1$ steps, which is the *induction hypothesis*, then we also end up in P after taking the k-th step.

Example 4.1.
Consider the transition system TS depicted in Figure 4.1, which already served us as a counterexample to the converse direction of Park induction in Example 2.12 on page 30. We employ k-induction to prove that $P = \{s_1, \ldots s_4\}$ covers all reachable states of TS. We start with $k = 1$, which fails: P is *not* 1-inductive since s_4 and s_5 violate the induction step. We have $s_4 \in P$ and $s_4 \longrightarrow s_5$ but $s_5 \notin P$. Next, we check whether P is 2-inductive. This *is* the case: s_4 and s_5 no longer violate the induction step since $s_4 \in P \wedge s_4 \longrightarrow s_5 \wedge s_5 \in P$ does *not* hold. Hence, the *stronger induction hypothesis* of 2-induction enables us to verify that P holds for TS.

The above example demonstrates that increasing k yields increasingly powerful proof rules in the sense that P might not be k-inductive but $(k+1)$-inductive. On the other hand, if P is k-inductive, it is also k'-inductive for all $k' \geq k$. In practice, checking k'-inductivity for $k' > k$ is usually computationally more expensive than checking k-inductivity since the corresponding conditions require to reason about ever more steps of the transition relation. For this reason, one typically starts with small k (e.g., $k = 1$), and gradually increases k until establishing k-

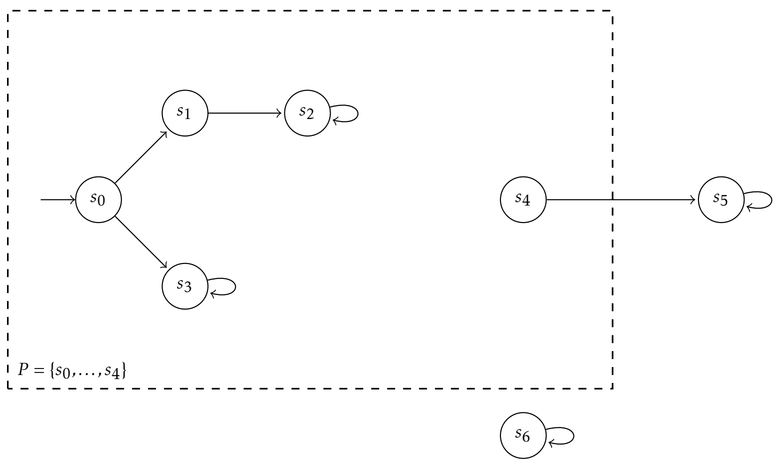


Figure 4.1: The transition system considered in Example 4.1 on page 126.

inductivity of P. Since k-induction is incomplete, this procedure might diverge: There are (both finite-state and infinite-state) transition systems TS and P such that Reach (TS) $\subseteq P$ but P is not k-inductive for any k. For finite-state transition systems, k-induction can be extended to be complete by excluding cycles in the transition system [SSS00], which is not considered here.

4.2.3 Excursus: Infinite-State Software Model Checking

One of the key aspects of k-induction is that — even for *infinite-state* transition systems — it suffices to reason about finitely many steps of the transition relation in order to verify a given invariant property. This makes k-induction suitable for the automatic verification of software [BK11; BDW15; DHKR11]. There, one of the most challenging tasks is to reason about loops. With k-induction, however, it may suffice to reason about a *finite* number of loop iterations to verify properties about an *unbounded* number of loop iterations.

Let us exemplify this by means of a simple example. Analogously to how probabilistic programs can be understood as finite representations of possibly infinite-state MDPs (cf. Section 2.3.2), *non-probabilistic* programs can be understood as finite representations of *infinite-state transition systems*. Consider the loop C over the $\mathbb{N}$-valued variables x and y given by

$$\texttt{while}(x \neq 0)\{y := y + 1\}$$

and fix the initial program state with $x = 0, y = 0$. We can model the execution behavior of C as the transition system

$$\mathsf{TS} \;=\; \left(\mathbb{N} \times \mathbb{N}, \longrightarrow, \{(0,0)\}\right),$$

where for every $i, j \in \mathbb{N}$, we have

$$(i,j) \longrightarrow (i',j') \quad \text{iff} \quad \begin{cases} i' = i \wedge j' = j + 1 & \text{if } i \neq 0 \\ i' = i \wedge j = j' & \text{if } \neg(i \neq 0)\,. \end{cases}$$

A state (i,j) of TS represents the program state σ with $\sigma(x) = i$ and $\sigma(y) = j$ at the beginning of a guarded loop iteration. Naturally, TS is infinite-state since there are infinitely many variable valuations. The transition relation $\longrightarrow$ models the state changes resulting from executing *one* guarded iteration of C. Now suppose we want to verify that $y < 1$ holds at the beginning of each guarded loop iteration when executing C on the given initial state, i.e., the invariant property

$$P \quad = \quad \{(i,j) \in \mathbb{N} \times \mathbb{N} \mid j < 1\}$$

holds for TS.

How can we employ k-induction to *automatically* check whether this property holds? The idea is to *encode* the two conditions from Theorem 4.2 on page 126 — the base case and the induction step — equivalently as the validity of certain logical formulae over the program variables. These logical formulae are then offloaded to off-the-shelf tools called *Satisfiability Modulo Theory (SMT, for short) solvers*, which are able to *decide*[2] whether these formulae are valid. Prime examples of SMT solvers include Z3 [MB08], CVC5 [BBBK+22], and MathSat5 [CGSS13]. The idea is to start with $k = 1$. If the SMT solver reports that the conditions for $k = 1$ hold, we are done by Theorem 4.2. If not, we proceed with $k = 2, 3, \ldots$ until we establish k-inductivity of P, or diverge.

We model the set $\mathcal{S}_I$ of initial states, the transition relation $\longrightarrow$, and the set P of states, respectively, by the following Linear Integer Arithmetic (LIA) formulae:

$$\mathsf{F}_{\mathsf{init}}(x,y) \quad = \quad x = 0 \wedge y = 0$$

[2] if all arithmetic involved is *linear*, which is the case here. In the presence of non-linear integer arithmetic, validity checking is undecidable. SMT solvers may still be able to check their validity by applying (incomplete) heuristics (see, e.g., [KCÁ16; BLNR+09]).

$$\begin{aligned}
\mathsf{F}_{\text{trans}}(x,y,x',y') \quad &= \quad \underbrace{\big((x \neq 0) \wedge (x' = x \wedge y' = y+1)\big)}_{\text{loop guard satisfied: } y \text{ is incremented}} \\
&\qquad\qquad \vee \quad \underbrace{\big(\neg(x \neq 0) \wedge (x' = x \wedge y' = y)\big)}_{\text{loop guard not satisfied: no state change}} \\
\mathsf{F}_{\text{prop}}(x,y) \quad &= \quad y < 1
\end{aligned}$$

Here x' and y' are copies of x and y often called *next-state-variables*: The formula $\mathsf{F}_{\text{trans}}$ encodes the transition relation $\longrightarrow$ in the sense that for all $i,j,i',j' \in \mathbb{N}$,

$$(i,j) \longrightarrow (i',j') \qquad \text{iff} \qquad \mathsf{F}_{\text{trans}}(i,j,i',j') \text{ holds .}$$

The above formulae can be generated automatically via strongest postconditions (see, e.g., [BDW18]). Now, P is 1-inductive, i.e., the two conditions from Theorem 4.2 hold, iff the following formula is valid:

$$\begin{aligned}
&\underbrace{\big(\mathsf{F}_{\text{init}}(x_1,y_1) \quad \Longrightarrow \quad \mathsf{F}_{\text{prop}}(x_1,y_1)\big)}_{\text{base case}} \\
\wedge &\underbrace{\big(\mathsf{F}_{\text{prop}}(x_1,y_1) \wedge \mathsf{F}_{\text{trans}}(x_1,y_1,x_2,y_2) \quad \Longrightarrow \quad \mathsf{F}_{\text{prop}}(x_2,y_2)\big)}_{\text{induction step}}
\end{aligned}$$

This can be checked automatically by, e.g., the SMT solver Z3. It turns out that P is *not* 1-inductive since Z3 reports that the induction step is violated for $x_1 = 1 \wedge y_1 = 0$ and $x_2 = 1 \wedge y_2 = 1$. Indeed, we have $(1,0) \in P$ and $(1,0) \longrightarrow (1,1)$ but $(1,1) \notin P$. We thus proceed by checking whether P is 2-inductive, which is the case iff the following formula is valid:

$$\begin{aligned}
&\underbrace{\big(\mathsf{F}_{\text{init}}(x_1,y_1) \wedge \mathsf{F}_{\text{trans}}(x_1,y_1,x_2,y_2) \quad \Longrightarrow \quad \mathsf{F}_{\text{prop}}(x_1,y_1) \wedge \mathsf{F}_{\text{prop}}(x_2,y_2)\big)}_{\text{base case}} \\
\wedge &\underbrace{\begin{aligned}\big(&\mathsf{F}_{\text{prop}}(x_1,y_1) \wedge \mathsf{F}_{\text{trans}}(x_1,y_1,x_2,y_2) \wedge \mathsf{F}_{\text{prop}}(x_2,y_2) \wedge \mathsf{F}_{\text{trans}}(x_2,y_2,x_3,y_3) \\ &\quad \Longrightarrow \quad \mathsf{F}_{\text{prop}}(x_3,y_3)\big)\end{aligned}}_{\text{induction step}}
\end{aligned}$$

Z3 reports that this is the case. Hence, P covers all states reachable in the infinite-state transition system TS by Theorem 4.2. The crucial point here is that k-

induction only requires us to reason about a *finite* number loop iterations, which can be automated by offloading the above LIA formulae to SMT solvers. Our fixpoint-theoretic understanding of k-induction presented in the next section enables a similar approach for the automatic verification of probabilistic loops.

4.3 Latticed k-Induction: Theory and Algorithm

In this section, we generalize the well-established k-induction verification technique to *latticed k-induction* (for short: *κ-induction*; reads: "kappa induction") — a generalization of Park induction (cf. Lemma 2.4 on page 28). We start with recapping this principle. Fix some complete lattice $(D, \sqsubseteq)$, some monotone function $\Phi\colon D \to D$, and some $u \in D$ for the rest of this section. Our aim is to *prove* that $\mathsf{lfp}\, \Phi \sqsubseteq u$. To this end, we attempt Park induction, i.e.,

$$\Phi(u) \sqsubseteq u \quad \text{implies} \quad \mathsf{lfp}\, \Phi \sqsubseteq u\,,$$

which, intuitively, says: if pushing our candidate upper bound u through Φ does *not take us up* in the partial order $\sqsubseteq$, we have verified that u is indeed an upper bound on $\mathsf{lfp}\, \Phi$. If $\Phi(u) \sqsubseteq u$, then we say that u is *inductive*.

We have seen in Example 2.12 on page 30 that Park induction, unfortunately, does *not* necessarily work in the converse direction: If we are unlucky, u *is* an upper bound on $\mathsf{lfp}\, \Phi$, but nevertheless *not* inductive, i.e., $\Phi(u) \not\sqsubseteq u$. But how can we verify that u is indeed an upper bound in such a non-inductive scenario? We search *below* u for a *different, but inductive*, upper bound on $\mathsf{lfp}\, \Phi$, that is, we

$$\text{search for an } e \in D \qquad \text{such that} \qquad \mathsf{lfp}\, \Phi \sqsubseteq \Phi(e) \sqsubseteq e \sqsubseteq u\,.$$

In order to perform a *guided* search for such an e, we introduce the κ-induction operator[3] — a modified version of Φ that is parameterized by our candidate u:

Definition 4.1 (κ-Induction Operator).
We call the function $\Psi_u\colon D \to D$ defined as

$$\Psi_u(d) = \Phi(d) \sqcap u$$

the *κ-induction operator (w.r.t. u and Φ)*.

What does Ψ_u do? As illustrated in Figure 4.2, if $\Phi(u) \not\sqsubseteq u$ (i.e. u is not inductive) then *"at least some part of $\Phi(u)$ is greater than u"*. If the whole of $\Phi(u)$ is greater than u, then $u \sqsubset \Phi(u)$; if only some part of $\Phi(u)$ is greater and some is smaller

[3]Notice that we use the Greek letter κ to emphasize that κ denotes possibly transfinite ordinals.

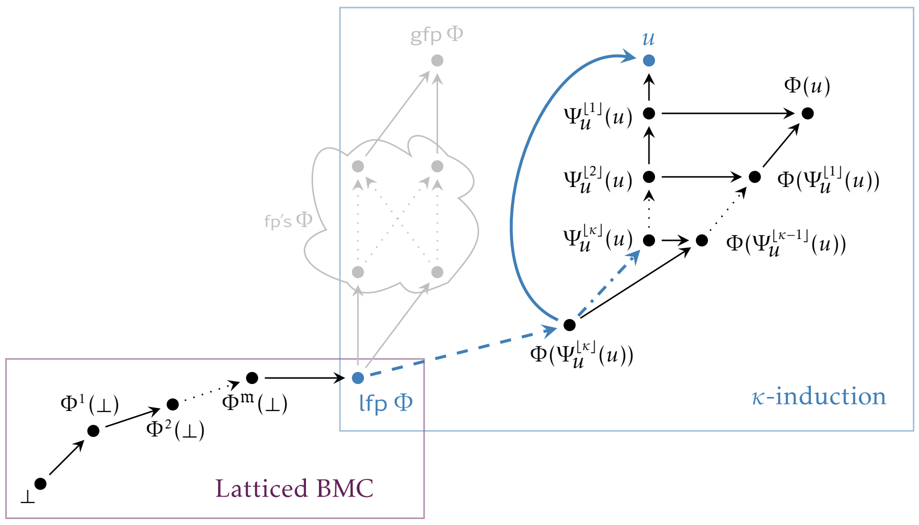

gfp Φ
fp's Φ
u
$\Phi(u)$
$\Psi_u^{\lfloor 1 \rfloor}(u)$
$\Psi_u^{\lfloor 2 \rfloor}(u)$
$\Phi(\Psi_u^{\lfloor 1 \rfloor}(u))$
$\Psi_u^{\lfloor \kappa \rfloor}(u)$
$\Phi(\Psi_u^{\lfloor \kappa-1 \rfloor}(u))$
$\Phi(\Psi_u^{\lfloor \kappa \rfloor}(u))$
$\Phi^1(\bot)$
$\Phi^2(\bot)$
$\Phi^m(\bot)$
lfp Φ
$\bot$
κ-induction
Latticed BMC

Figure 4.2: κ-induction and latticed BMC in case that lfp $\Phi \sqsubseteq u$. An arrow from d to e indicates $d \sqsubseteq e$. The solid blue arrow from $\Phi(\Psi_u^{\lfloor \kappa \rfloor}(u))$ to u is the premise of κ-induction, i.e., the LHS of Lemma 4.4, which implies the dash-dotted blue arrow from $\Phi(\Psi_u^{\lfloor \kappa \rfloor}(u))$ to $\Psi_u^{\lfloor \kappa \rfloor}(u)$, i.e., the RHS of Lemma 4.4. The dashed blue arrow from lfp u to $\Phi(\Psi_u^{\lfloor \kappa \rfloor}(u))$ is a consequence of the dash-dotted arrow (by Park induction, Lemma 2.4 on page 28) and ultimately proves that lfp $\Phi \sqsubseteq u$.

than u, then u and $\Phi(u)$ are incomparable (recall that $\sqsubseteq$ is generally not total). The κ-induction operator Ψ_u now *rectifies* $\Phi(u)$ being (partly) greater than u by *pulling* $\Phi(u)$ *down* via the meet with u (i.e., via $\ldots \sqcap u$), so that the result — $\Phi(u) \sqcap u$ — is in no part greater than u. Applying Ψ_u to u hence always yields something below or equal to u.

Let $\mathfrak{n}$ be an ordinal. Towards formalizing κ-induction, we denote by $\Psi_u^{\lfloor \mathfrak{n} \rfloor}(d)$ the *lower $\mathfrak{n}$-fold iteration of Ψ_u on $d \in D$* defined by transfinite recursion as[4]

$$\Psi_u^{\lfloor \mathfrak{n} \rfloor}(d) \;=\; \begin{cases} d & \text{if } \mathfrak{n} = 0\,, \\ \Psi_u\left(\Psi_u^{\lfloor \mathfrak{m} \rfloor}(d)\right) & \text{if } \mathfrak{n} = \mathfrak{m} + 1 \text{ is a successor ordinal}\,, \\ \bigsqcap \left\{ \Psi_u^{\lfloor \mathfrak{m} \rfloor}(d) \;\middle|\; \mathfrak{m} < \mathfrak{n} \right\} & \text{if } \mathfrak{n} \text{ is a limit ordinal}\,. \end{cases}$$

If $\mathsf{lfp}\, \Phi \sqsubseteq u$, then monotonicity of Ψ_u implies that iterating Ψ_u on u *descends* from u downwards in the direction of $\mathsf{lfp}\, \Phi$ (and never below):

Lemma 4.3 (Properties of the κ-Induction Operator).
We have:

1. Ψ_u is monotone, i.e.,
$$\text{for all } e_1, e_2 \in D\colon \quad e_1 \sqsubseteq e_2 \;\text{ implies }\; \Psi_u(e_1) \sqsubseteq \Psi_u(e_2)\,.$$
2. Iterations of Ψ_u starting from u are descending, i.e.,
$$\text{for all ordinals } \mathfrak{n}, \mathfrak{m}\colon \quad \mathfrak{n} < \mathfrak{m} \quad \text{implies} \quad \Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \sqsubseteq \Psi_u^{\lfloor \mathfrak{n} \rfloor}(u)\,.$$
3. Ψ_u is dominated by Φ, i.e.,
$$\text{for all } e \in D\colon \quad \Psi_u(e) \sqsubseteq \Phi(e)\,.$$
4. If $\mathsf{lfp}\, \Phi \sqsubseteq u$, then for any ordinal $\mathfrak{m}$,
$$\mathsf{lfp}\, \Phi \;\sqsubseteq\; \Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \;\sqsubseteq\; \ldots \;\sqsubseteq\; \Psi_u^{\lfloor 2 \rfloor}(u) \;\sqsubseteq\; \Psi_u(u) \;\sqsubseteq\; u\,.$$

Proof. For Lemma 4.3.1, we have

$$\begin{aligned} &\Psi_u(e_1) \\ =\; &\Phi(e_1) \sqcap u && \text{(Definition 4.1)} \end{aligned}$$

[4]Recall from Section 2.1.4 that we fix an ambient ordinal $\mathfrak{k}$, which is the smallest ordinal such that $|\mathfrak{k}| > |D|$, and tacitly assume $\mathfrak{n} < \mathfrak{k}$ for all ordinals $\mathfrak{n}$ considered throughout.

$$
\begin{aligned}
&\sqsubseteq \Phi(e_2) \sqcap u && \text{(monotonicity of } \Phi \text{ and } \sqcap) \\
&= \Psi_u(e_2)\,. && \text{(Definition 4.1)}
\end{aligned}
$$

For Lemma 4.3.2, we proceed by transfinite induction on $\mathfrak{m}$. The case $\mathfrak{m} = 0$ is trivial. For the remaining cases, we reason as follows:

The case $\mathfrak{m} = \mathfrak{l} + 1$. Let $\mathfrak{n} < \mathfrak{m}$ and notice that the I.H. and reflexivity of $\sqsubseteq$ (for $\mathfrak{l} = \mathfrak{n}$) yield $\Psi_u^{\lfloor \mathfrak{l} \rfloor}(u) \sqsubseteq \Psi_u^{\lfloor \mathfrak{n} \rfloor}(u)$. This gives us

$$
\begin{aligned}
&\Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \\
&= \Psi_u(\Psi_u^{\lfloor \mathfrak{l} \rfloor}(u)) \\
&\sqsubseteq \Psi_u(\Psi_u^{\lfloor \mathfrak{n} \rfloor}(u)) && \text{(Lemma 4.3.1 and I.H.)} \\
&= \Psi_u^{\lfloor \mathfrak{n}+1 \rfloor}(u)\,.
\end{aligned}
$$

It remains to show that $\Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \sqsubseteq \Psi_u^{\lfloor 0 \rfloor}(u)$. We have

$$
\begin{aligned}
&\Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \\
&= \Psi_u(\Psi_u^{\lfloor \mathfrak{l} \rfloor}(u)) \\
&= \Phi(\Psi_u^{\lfloor \mathfrak{l} \rfloor}(u)) \sqcap u && \text{(Definition 4.1)} \\
&\sqsubseteq u && \text{(definition of infima)} \\
&= \Psi_u^{\lfloor 0 \rfloor}(u)\,.
\end{aligned}
$$

The case $\mathfrak{m}$ *limit ordinal.* We have

$$
\Psi_u^{\lfloor \mathfrak{m} \rfloor}(u) \;=\; \bigsqcap \left\{ \Psi_u^{\lfloor \mathfrak{n} \rfloor}(d) \;\middle|\; \mathfrak{n} < \mathfrak{m} \right\}
$$

which immediately implies the claim by definition of infima.

For Lemma 4.3.4, we have

$$
\begin{aligned}
&\Psi_u(e) \\
&= \Phi(e) \sqcap u && \text{(Definition 4.1)} \\
&\sqsubseteq \Phi(e)\,. && \text{(definition of infima)}
\end{aligned}
$$

For Lemma 4.3.3, it suffices to prove $\mathsf{lfp}\,\Phi \sqsubseteq \Psi_u^{\lfloor \mathfrak{m} \rfloor}(u)$ since the remaining inequalities follow from Lemma 4.3.2. We proceed by transfinite induction on $\mathfrak{m}$. The base case $\mathfrak{m} = 0$ holds by assumption.

The case $\mathfrak{m} = \mathfrak{l} + 1$. We have

$$\begin{aligned}
& \mathsf{lfp}\ \Phi \\
\sqsubseteq\ & \mathsf{lfp}\ \Phi \sqcap u && \text{(assumption and reflexivity of } \sqsubseteq\text{, definition of infima)} \\
=\ & \Phi(\mathsf{lfp}\ \Phi) \sqcap u && \text{(fixpoint property)} \\
\sqsubseteq\ & \Phi(\Psi_u^{\lfloor \mathfrak{l} \rfloor}(u)) \sqcap u && \text{(monotonicity of } \Phi \text{ and } \sqcap\text{, I.H.)} \\
=\ & \Psi_u^{\lfloor \mathfrak{l}+1 \rfloor}(u) && \text{(Definition 4.1)} \\
=\ & \Psi_u^{\lfloor \mathfrak{m} \rfloor}(u)\,.
\end{aligned}$$

The case $\mathfrak{m}$ *limit ordinal.* We have

$$\begin{aligned}
& \mathsf{lfp}\ \Phi \\
=\ & \bigsqcap \{\mathsf{lfp}\ \Phi\} \\
\sqsubseteq\ & \bigsqcap \left\{ \Psi_u^{\lfloor \mathfrak{n} \rfloor}(u) \mid \mathfrak{n} < \mathfrak{m} \right\} && (\mathsf{lfp}\ \Phi \sqsubseteq \Psi_u^{\lfloor \mathfrak{n} \rfloor}(u) \text{ for all } \mathfrak{n} < \mathfrak{m} \text{ by I.H.}) \\
=\ & \Psi_u^{\lfloor \mathfrak{m} \rfloor}\,.
\end{aligned}$$

■

The descending chain $u \sqsupseteq \Psi_u(u) \sqsupseteq \Psi_u^{\lfloor 2 \rfloor}(u) \sqsupseteq \ldots$ constitutes our guided search for an inductive upper bound on $\mathsf{lfp}\ \Phi$. For each ordinal κ (hence the short name: κ-induction), $\Psi_u^{\lfloor \kappa \rfloor}(u)$ is a potential candidate for Park induction:

$$\Phi\left(\Psi_u^{\lfloor \kappa \rfloor}(u)\right) \overset{\text{potentially}}{\sqsubseteq} \Psi_u^{\lfloor \kappa \rfloor}(u)\,. \tag{‡}$$

For efficiency reasons, e.g., when offloading the above inequality check to an SMT solver, we will not check the inequality (‡) directly but a property equivalent to (‡), namely whether $\Phi\left(\Psi_u^{\lfloor \kappa \rfloor}(u)\right)$ is below u instead of $\Psi_u^{\lfloor \kappa \rfloor}(u)$:

Lemma 4.4 (Park Induction from κ-Induction).
Let κ be an ordinal. We have

$$\Phi\left(\Psi_u^{\lfloor \kappa \rfloor}(u)\right) \sqsubseteq u \quad \text{iff} \quad \Phi\left(\Psi_u^{\lfloor \kappa \rfloor}(u)\right) \sqsubseteq \Psi_u^{\lfloor \kappa \rfloor}(u)\,.$$

Proof. The if-direction is trivial, as $\Psi_u^{\lfloor\kappa\rfloor}(u) \sqsubseteq u$ (Lemma 4.3.4). For only-if:

$$\begin{aligned}
\Psi_u^{\lfloor\kappa\rfloor}(u) &\sqsupseteq \Psi_u^{\lfloor\kappa+1\rfloor}(u) && \text{(Lemma 4.3.2)}\\
&= \Psi_u\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) && \text{(definition of } \Psi_u^{\lfloor\kappa+1\rfloor}(u))\\
&= \Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) \sqcap u && \text{(definition of } \Psi_u)\\
&\sqsupseteq \Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) . && \text{(premise)}
\end{aligned}$$

■

If $\Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) \sqsubseteq u$, then Lemma 4.4 tells us that $\Psi_u^{\lfloor\kappa\rfloor}(u)$ is inductive and thereby an upper bound on $\mathsf{lfp}\,\Phi$. Since iterating Ψ_u on u yields a descending chain by Lemma 4.3.2, $\Psi_u^{\lfloor\kappa\rfloor}(u)$ is at most u and therefore u is also an upper bound on $\mathsf{lfp}\,\Phi$. Formulated as a proof rule, we obtain the following induction principle:

Theorem 4.5 (κ-Induction).
Let κ be an ordinal. Then

$$\Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) \sqsubseteq u \qquad \text{implies} \qquad \mathsf{lfp}\,\Phi \sqsubseteq u .$$

Proof. We have

$$\begin{aligned}
& \Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) \sqsubseteq u \\
\text{implies}\quad & \Phi\left(\Psi_u^{\lfloor\kappa\rfloor}(u)\right) \sqsubseteq \Psi_u^{\lfloor\kappa\rfloor}(u) && \text{(Lemma 4.4)}\\
\text{implies}\quad & \mathsf{lfp}\,\Phi \sqsubseteq \Psi_u^{\lfloor\kappa\rfloor}(u) && \text{(Lemma 2.4)}\\
\text{implies}\quad & \mathsf{lfp}\,\Phi \sqsubseteq u . && \text{(Lemma 4.3.2 and transitivity of } \sqsubseteq)
\end{aligned}$$

■

An illustration of κ-induction is shown in (the right frame of) Figure 4.2. If $\Phi(\Psi_u^{\lfloor\kappa\rfloor}(u)) \sqsubseteq u$ for some ordinal κ, then we call *u (κ+1)-inductive (for Φ)*. In particular, κ-induction generalizes Park induction, in the sense that 1-induction *is* Park induction and, (κ>1)-induction is a *more general principle of induction*.

Algorithm 1 on page 136 depicts a (semi-)algorithm that performs *latticed k-induction* (for $k < \omega$) to prove $\mathsf{lfp}\,\Phi \sqsubseteq u$ by iteratively increasing k. We assume here that both Φ and Ψ_u are computable and that $\sqsubseteq$ is decidable. In practice, this is often not the case, which, on top of the incompleteness of κ-induction considered next, yields an additional source of incompleteness of Algorithm 1.

Algorithm 1 is generally a *proper* semi-algorithm: even if $\mathsf{lfp}\,\Phi \sqsubseteq u$, then

Algorithm 1: Latticed k-induction

```
input: $\Phi\colon D \to D$ and $u \in D$.
output: "verify" if $u$ is $k$-inductive for some $k$, diverge otherwise.
1 $e \leftarrow u$
2 while $\Phi(e) \not\sqsubseteq u$ do
3     $e \leftarrow \Psi_u(e)$
      // recall: $\Psi_u(e) = \Phi(e) \sqcap u$
4 return verify
```

Algorithm 2: Latticed BMC

```
input: $\Phi\colon D \to D$ and $u \in D$.
output: "refute" if there exists $k \in \mathbb{N}$ with $\Phi^k(\bot) \not\sqsubseteq u$, diverge otherwise.
1 $e \leftarrow \bot$
2 repeat
3     $e \leftarrow \Phi(e)$
4 until $e \not\sqsubseteq u$
5 return refute
```

Figure 4.3: Algorithmic perspective on latticed k-Induction and BMC.

u is still not guaranteed to be k-inductive for some $k < \omega$. And even if an algorithm *could* somehow perform transfinitely many iterations, then u is still not guaranteed to be κ-inductive for some ordinal κ:

Example 4.2 (Incompleteness of κ-Induction).
Consider the complete lattice $(\{0,1,2\}, \sqsubseteq)$ where $0 \sqsubseteq 1 \sqsubseteq 2$, and the monotone operator Φ with $\Phi(0) = 0 = \mathsf{lfp}\ \Phi$, and $\Phi(1) = 2$, and $\Phi(2) = 2 = \mathsf{gfp}\ \Phi$. Then $\mathsf{lfp}\ \Phi \sqsubseteq 1$, but for any ordinal κ, $\Psi_1^{\lfloor\kappa\rfloor}(1) = 1$ and $\Phi(1) = 2 \not\sqsubseteq 1$. Hence 1 is not κ-inductive for every ordinal κ.

We now provide a *sufficient* criterion which ensures that *every* upper bound on $\mathsf{lfp}\ \Phi$ is κ-inductive for some ordinal κ. This is the case if Φ has *exactly one fixpoint*, i.e., if the least and the greatest fixpoint of Φ coincide. Dually to least fixpoints, a fixpoint $d \in D$ of Φ is called the *greatest fixpoint of* Φ, if

$$\text{for all fixpoints } d' \text{ of } \Phi\colon \quad d' \sqsubseteq d\,.$$

For monotone Φ over a complete lattice, the greatest fixpoint is guaranteed to exist uniquely [Tar55] and we denote it by $\mathsf{gfp}\ \Phi$.

Theorem 4.6 (Completeness of κ-Induction for Unique Fixpoint).
If $\mathsf{lfp}\,\Phi = \mathsf{gfp}\,\Phi$, then, for every $u \in D$,

$$\mathsf{lfp}\,\Phi \sqsubseteq u \quad \text{implies} \quad u \text{ is } \kappa\text{-inductive for some ordinal } \kappa\ .$$

Proof. By Cousot & Cousot's fixpoint theorem for greatest fixpoints [CC79], we have $\Phi^{\lfloor \mathfrak{m} \rfloor}(\top) = \mathsf{gfp}\,\Phi$ for some ordinal $\mathfrak{m}$. We then show that u is $(\mathfrak{m}{+}1)$-inductive, see [10, Appendix A.3]. ∎

The proof of the above theorem immediately yields that, if the unique fixpoint can be reached through *finite* fixpoint iterations starting at $\top$, then u is k-inductive for some *natural* number k.

Corollary 4.7.
Let $n \in \mathbb{N}$. If $\Phi^{\lfloor n \rfloor}(\top) = \mathsf{lfp}\,\Phi$, then, for every $u \in D$,

$$\mathsf{lfp}\,\Phi \sqsubseteq u \quad \text{implies} \quad u \text{ is } (n+1)\text{-inductive}\ .$$

4.4 Latticed versus Classical k-Induction

We show that our fixpoint-theoretic κ-induction from Section 4.3 generalizes classical k-induction for transition systems as formalized in Theorem 4.2 on page 126. For that, let $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ be a transition system and let $P \subseteq \mathcal{S}$ be an invariant property. We instantiate κ-induction as follows. The complete lattice is induced by TS and given by (cf. Section 2.1.2)

$$(\mathcal{P}(\mathcal{S}), \subseteq)\ .$$

The monotone function is the reachability operator of TS given by

$$\Phi_{\mathsf{TS}} \colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}), \qquad \Phi_{\mathsf{TS}}(X) \;=\; \mathcal{S}_I \cup \{ s \mid \text{exists } t \in X \colon t \longrightarrow s \}\ .$$

Recall from Remark 2.1 on page 23 that we have

$$\mathsf{lfp}\,\Phi_{\mathsf{TS}} \;=\; \mathsf{Reach}(\mathsf{TS}) \qquad \text{and hence} \qquad \mathsf{Reach}(\mathsf{TS}) \subseteq P \;\text{ iff }\; \mathsf{lfp}\,\Phi_{\mathsf{TS}} \subseteq P\ .$$

Using the κ-induction operator Ψ_P constructed from Φ_{TS} and P according to Definition 4.1, the principle of κ-induction (Theorem 4.5) then tells us that

$$\text{for all } k \in \mathbb{N}\colon \qquad \Phi_{\mathsf{TS}}\left(\Psi_P^{\lfloor k \rfloor}(P)\right) \;\subseteq\; P \quad \text{implies} \quad \mathsf{Reach}(\mathsf{TS}) \;\subseteq\; P\ .$$

How is classical k-induction for transition systems (Theorem 4.2 on page 126) reflected in *latticed* k-induction? We show that, for the given instance, the premise of κ-induction — $\Phi_{\mathsf{TS}}\left(\Psi_P^{\lfloor k \rfloor}(P)\right) \subseteq P$ — is equivalent to the premises of classical k-induction — Conditions (4.3a) and (4.3b) from Theorem 4.2. Towards this end, define the function

$$\mathsf{Succs}\colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}), \quad \mathsf{Succs}(X) \;=\; \{s \mid \text{exists } t \in X\colon t \longrightarrow s\} ~,$$

which returns the set of direct successors of states in X. Moreover, define

$$\Lambda_P\colon \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S}), \quad \Lambda_P(X) \;=\; \mathsf{Succs}(X) \cap P ~.$$

Now let $k \geq 1$. For the base case of classical k-induction, we have

$$\begin{aligned} & \text{for all } s_1, \ldots, s_k \in \mathcal{S}\colon \quad s_1 \in \mathcal{S}_I \wedge \bigwedge_{i=1}^{k-1} s_i \longrightarrow s_{i+1} \quad \Longrightarrow \quad \bigwedge_{i=1}^{k} s_i \in P \\ \text{iff} \quad & \text{for all } i \in \{0, \ldots, k-1\}\colon \quad \mathsf{Succs}^i(\mathcal{S}_I) \subseteq P ~. \end{aligned}$$

Moreover, for the induction step of classical k-induction, we have

$$\begin{aligned} & \text{for all } s_1, \ldots, s_k, s_{k+1} \in \mathcal{S}\colon \\ & s_1 \in P \wedge \left(\bigwedge_{i=1}^{k-1} s_i \longrightarrow s_{i+1} \wedge s_{i+1} \in P \right) \wedge s_k \longrightarrow s_{k+1} \quad \Longrightarrow \quad s_{k+1} \in P \\ \text{iff} \quad & \mathsf{Succs}\left(\Lambda_P^{\lfloor k-1 \rfloor}(P)\right) \subseteq P ~. \end{aligned}$$

Hence, the premises of classical k-induction are equivalent to

$$\text{for all } i \in \{0, \ldots, k-1\}\colon \mathsf{Succs}^i(\mathcal{S}_I) \subseteq P \text{ and } \mathsf{Succs}\left(\Lambda_P^{\lfloor k-1 \rfloor}(P)\right) \subseteq P ~.$$

Since the above conjunction is equivalent to $\Phi_{\mathsf{TS}}\left(\Psi_P^{\lfloor k-1 \rfloor}(P)\right) \subseteq P$, we get:

Theorem 4.8 (Equivalence of Latticed and Classical k-Induction). For every natural number $k \geq 1$,

$$\begin{aligned} & \Phi_{\mathsf{TS}}\left(\Psi_P^{\lfloor k-1 \rfloor}(P)\right) \;\subseteq\; P \\ \text{iff} \quad & P \text{ is } k\text{-inductive in the sense of Theorem 4.2} ~. \end{aligned}$$

Proof. See [10, Appendix A.4] for a detailed proof. ■

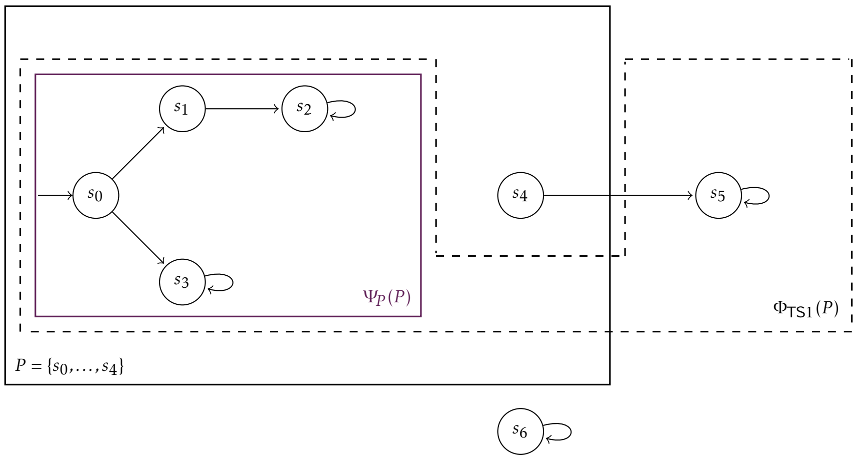


Figure 4.4: Applying κ-induction to Example 4.1 on page 126.

Example 4.3.
We reconsider Example 4.1 on page 126 through the lens of κ-induction. Consider the corresponding transition TS and the invariant property $P = \{s_0, \ldots, s_4\}$ shown in Figure 4.4 on page 139. P is not 1-inductive since

$$\Phi_{\mathsf{TS}}(P) \;=\; (P \setminus \{s_4\}) \cup \{s_5\} \;\not\subseteq\; P\,.$$

P is, however, 2-inductive since

$$\Phi_{\mathsf{TS}}(\Psi_P(P)) \;=\; \{s_0, \ldots, s_3\} \;\subseteq\; P\,.$$

Hence, we have by Theorem 4.5,

$$\mathsf{Reach}(\mathsf{TS}) \;=\; \mathsf{lfp}\,\Phi_{\mathsf{TS}} \;\subseteq\; P\,.$$

4.5 Latticed Bounded Model Checking

We complement κ-induction with a latticed analog of bounded model checking (cf. Section 4.2.1) for *refuting* $\mathsf{lfp}\,\Phi \sqsubseteq u$. In fixpoint-theoretic terms, bounded model checking amounts to a *fixpoint iteration* of u on $\bot$ while continually checking whether the iteration exceeds our candidate upper bound u. If so, then we have indeed refuted $\mathsf{lfp}\,\Phi \sqsubseteq u$:

Theorem 4.9 (Soundness of Latticed BMC [CC79]).
We have

$$\text{exists ordinal } \mathfrak{m}\colon \Phi^{\mathfrak{m}}(\bot) \not\sqsubseteq u \qquad \text{implies} \qquad \mathsf{lfp}\,\Phi \not\sqsubseteq u\,.$$

Proof. This is an immediate consequence of Theorem 2.3 on page 27. ■

Furthermore, if we were actually able to perform transfinite iterations of Φ on $\bot$, then latticed bounded model checking is also complete: If u is in fact *not* an upper bound on $\mathsf{lfp}\,\Phi$, this *will* be witnessed at some ordinal:

Theorem 4.10 (Completeness of Latticed BMC [CC79]).
We have

$$\mathsf{lfp}\,\Phi \not\sqsubseteq u \qquad \text{implies} \qquad \text{exists ordinal } \mathfrak{m}\colon \quad \Phi^{\mathfrak{m}}(\bot) \not\sqsubseteq u\,.$$

Proof. This is an immediate consequence of Theorem 2.3 on page 27. ■

More practically relevant, if Φ is continuous, then a simple *finite* fixpoint iteration, see Algorithm 2, is sound and complete for refutation:

Corollary 4.11 (Latticed BMC for Continuous Functions [LNS82]).
If Φ is continuous, then

$$\text{exists } n \in \mathbb{N}\colon \Phi^{n}(\bot) \not\sqsubseteq u \qquad \text{iff} \qquad \mathsf{lfp}\,\Phi \not\sqsubseteq u\,.$$

Proof. This is an immediate consequence of Theorem 2.2 on page 22. ■

4.6 k-Induction and BMC for Probabilistic Programs

We instantiate latticed k-Induction and BMC for reasoning about upper bounds on expected outcomes of pGCL loops. In Section 4.6.1, we first instantiate our theory for *arbitrary* pGCL loops and expectations. Towards automating the so-obtained techniques, we then consider sufficient conditions on classes of pGCL loops and expectations for which Algorithms 1 and 2 on page 136 are *semi-decision procedures* in Section 4.6.2. Section 4.6.3 introduces concrete instances of such classes. Finally, in Section 4.6.4, we present our implementation.

4.6.1 Instantiating Latticed k-Induction and BMC

We now instantiate latticed k-induction and BMC to enable verification of loops written in pGCL. Fix some $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$ and a loop

$$C \;=\; \texttt{while}(B)\{C'\}\,.$$

Given a postexpectation $X \in \mathbb{E}$ and a candidate upper bound $Y \in \mathbb{E}$ on $\mathcal{T}[\![C]\!](X)$, we will apply latticed verification techniques to verify or refute that

$$\mathcal{T}[\![C]\!](X) \;\sqsubseteq\; Y\,.$$

We operate in the complete lattice $(\mathbb{E}, \sqsubseteq)$ of expectations (cf. Section 2.4.2). The monotone function is C's characteristic function (cf. Definition 2.21 on page 63)

$${}^{\mathcal{T}}_{C}\Phi_X \colon \mathbb{E} \to \mathbb{E}\,, \quad {}^{\mathcal{T}}_{C}\Phi_X(Z) \;=\; [B]\cdot \mathcal{T}[\![C']\!](Y) + [\neg B]\cdot X\,.$$

We remark that ${}^{\mathcal{T}}_{C}\Phi_X$ is a monotone — and in fact even continuous — function w.r.t. $(\mathbb{E}, \sqsubseteq)$. Recall that, in this lattice, the meet is a pointwise minimum, i.e.,

$$Z \sqcap Z' \;=\; \lambda\sigma.\,\min\{Z(\sigma), Z'(\sigma)\}\,.$$

By Definition 4.1 on page 130, ${}^{\mathcal{T}}_{C}\Phi_X$ and Y then induce the κ-induction operator

$$\Psi_Y \colon \mathbb{E} \to \mathbb{E}, \qquad \Psi_Y(Z) \;=\; {}^{\mathcal{T}}_{C}\Phi_X(Z) \sqcap Y\,.$$

With this setup, we obtain the following proof rule for reasoning about probabilistic loops as an immediate consequence of Theorem 4.5:

Corollary 4.12 (k-Induction for pGCL).
For every natural number $k \in \mathbb{N}$,

$${}^{\mathcal{T}}_{C}\Phi_X\left(\Psi_Y^{\lfloor k \rfloor}(Y)\right) \;\sqsubseteq\; Y \quad \text{implies} \quad \mathcal{T}[\![C]\!](X) \;\sqsubseteq\; Y\,.$$

If the above premise holds, then we say that *Y is $(k+1)$-inductive for C w.r.t. $\mathcal{T}$ and X*. If clear from the context, we omit $\mathcal{T}$. Notice that Y is 1-inductive iff Y is a $\mathcal{T}$-superinvariant of C w.r.t. X (cf. Theorem 2.12 on page 66).

Analogously, refuting that Y upper-bounds the expected value of X after execution of C via bounded model checking is an instance of Corollary 4.11:

Corollary 4.13 (Bounded Model Checking for pGCL).
We have

$$\text{exists } n \in \mathbb{N}\colon\ {}^{\mathcal{T}}_{C}\Phi_{X}{}^{n}(0) \not\sqsubseteq Y \qquad \text{iff} \qquad \mathcal{T}[\![C]\!](X) \not\sqsubseteq Y\,.$$

Example 4.4.
Reconsider the geometric loop from Example 2.24 on page 66 given by

$$C \;=\; \mathtt{while}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x+1\}\}$$

and fix the postexpectation $X = x$.

1. We have seen that
$$Y_1 \;=\; [y = 1] \cdot (x+1) + [y \neq 1] \cdot x$$
is an upper bound on $\mathsf{wp}[\![C]\!](x)$ since Y_1 is a wp-superinvariant of C w.r.t. postexpectation x and hence also 1-inductive.

2. Let us now consider
$$Y_2 \;=\; x + 1\,.$$
Clearly, Y_2 is also an upper bound on $\mathsf{wp}[\![C]\!](x)$ since $Y_1 \sqsubseteq Y_2$. Y_2 is, however, *not* 1-inductive since
$$\begin{aligned}
&{}^{\mathsf{wp}}_{C}\Phi_{x}(Y_2)\\
=\;& [y = 1] \cdot \mathsf{wp}[\![\{y := 0\}\,[1/2]\,\{x := x+1\}]\!](Y_2) + [y \neq 1] \cdot x\\
=\;& [y = 1] \cdot (1/2 \cdot (x+1) + 1/2 \cdot (x+2)) + [y \neq 1] \cdot x\\
=\;& [y = 1] \cdot (x + 1.5) + [y \neq 1] \cdot x\\
\not\sqsubseteq\;& x + 1 \;=\; Y_2\,.
\end{aligned}$$
Y_2 is, however, 2-inductive because
$$\begin{aligned}
&{}^{\mathsf{wp}}_{C}\Phi_{x}\big(\Psi_{Y_2}(Y_2)\big)\\
=\;& {}^{\mathsf{wp}}_{C}\Phi_{x}\big({}^{\mathsf{wp}}_{C}\Phi_{x}(Y_2) \sqcap Y_2\big)\\
=\;& {}^{\mathsf{wp}}_{C}\Phi_{x}\big(([y = 1] \cdot (x + 1.5) + [y \neq 1] \cdot x) \sqcap (x+1)\big) && \text{(see above)}\\
=\;& {}^{\mathsf{wp}}_{C}\Phi_{x}\big([y = 1] \cdot (x+1) + [y \neq 1] \cdot x\big)\\
=\;& [y = 1] \cdot (x+1) + [y \neq 1] \cdot x && \text{(see Example 2.24 on page 66)}\\
\sqsubseteq\;& Y_2\,.
\end{aligned}$$

3. Some upper bounds on $\mathsf{wp}[\![C]\!](x)$ only become κ-inductive for *transfinite* ordinals κ. For instance,

   $$Y_3 \;=\; 2 \cdot x + 1$$

   is not k-inductive for any natural number k, but can be shown to be $\omega + 1$-inductive since

   $${}^{\mathsf{wp}}_{C}\Phi_x\left(\Psi_{Y_3}^{\lfloor\omega\rfloor}(Y_3)\right) \;\sqsubseteq\; Y_3\,.$$

4. κ-induction for probabilistic loops is not complete: Consider

   $$Y_4 \;=\; [y = 1] \cdot ([x = 0] \cdot 1 + [x \neq 0] \cdot \infty) \cdot [y \neq 1] \cdot x\,.$$

   Y_4 is not 1-inductive and we have

   $$\Psi_{Y_4}(Y_4) = Y_4 \quad \text{and thus} \quad \text{for all ordinals } \kappa \geq 0\colon\, \Psi_{Y_4}^{\lfloor\kappa\rfloor}(Y_4) = Y_4\,.$$

   Hence, Y_4 is not κ-inductive for any κ.

5. Finally, we can refute via BMC that

   $$Y_5 \;=\; x + 0.99$$

   is an upper bound on $\mathsf{wp}[\![C]\!](x)$ since ${}^{\mathsf{wp}}_{C}\Phi_x^{11}(0) \not\sqsubseteq Y_5$.

4.6.2 Automatic Reasoning about Loops

Given $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$, expectations $X, Y \in \mathbb{E}$, and a loop

$$C \;=\; \mathtt{while}(B)\{C'\}\,,$$

we now automate our latticed techniques for verifying or refuting whether

$$\mathcal{T}[\![C]\!](X) \;\sqsubseteq\; Y$$

holds. More precisely, we semi-decide whether (i) Y is k-inductive for C w.r.t. X for some k or (ii) $\mathcal{T}[\![C]\!](X) \sqsubseteq Y$ can be refuted via BMC. For simplicity, we assume that the loop body C' is loop-free. We call such loops *single loops*[5].

In principle, it suffices to run Algorithms 1 and 2 on page 136 in parallel. For these algorithms to be semi-decision procedures, we can, however, neither admit arbitrary loops C (more precisely: arbitrary arithmetic and Boolean expressions occurring in assignments and guards) nor arbitrary expectations. The reason is that we require the checks in the respective termination conditions of Algorithms 1 and 2 to be decidable. Let us therefore investigate sufficient

[5]Every probabilistic program can be rewritten as a single while loop with loop-free body [RS09].

conditions on classes of single loops and expectations for which this is the case. To formalize subclasses of expectations, we proceed similarly to Chapter 3 by considering appropriate formal languages of expectations.

Definition 4.2 (Fragments of pGCL and $\mathbb{E}$ Suitable for Automation).
Let AutpGCL $\subseteq$ pGCL be a set of single loops. Moreover, let AutExp be a computable formal language of syntactic expectations equipped with a semantic function $[\![\cdot]\!] : \mathsf{AutExp} \to \mathbb{E}$. We say that the pair

$$(\mathsf{AutpGCL}, \mathsf{AutExp})$$

is *suitable for automation*, if the following conditions hold:

1. AutExp is effectively closed under characteristic functions of loops in AutpGCL for both dwp and awp, i.e., for $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$,

$$\begin{aligned}&\text{for all } C \in \mathsf{AutpGCL} \text{ and } f, h \in \mathsf{AutExp}\colon\\ &\qquad \text{exists effectively constructible } h' \in \mathsf{AutExp}\colon\\ &\qquad\qquad [\![h']\!] ~=~ {}^{\mathcal{T}}_{C}\Phi_{[\![f]\!]}([\![h]\!])~.\end{aligned}$$

2. *Quantitative entailments* in AutExp are decidable, i.e.,

   Given $h_1, h_2 \in \mathsf{AutExp}$, *does* $[\![h_1]\!] \sqsubseteq [\![h_2]\!]$ *hold?*

   is decidable.

3. AutExp is effectively closed under pointwise minima, i.e.,

$$\begin{aligned}&\text{for all } h_1, h_2 \in \mathsf{AutExp}\colon\\ &\qquad \text{exists effectively constructible } h' \in \mathsf{AutExp}\colon\\ &\qquad\qquad [\![h']\!] ~=~ [\![h_1]\!] \sqcap [\![h_2]\!]~.\end{aligned}$$

Restricting to loops in AutpGCL and expectations in AutExp thus yield Algorithms 1 and 2 to be semi-decision procedures and we get:

Theorem 4.14 (Automatic Reasoning about Loops).
If (AutpGCL, AutExp) is suitable for automation, then for $\mathcal{T} \in \{\mathsf{dwp}, \mathsf{awp}\}$, both of the following problems are semi-decidable:

1. *Given* $C \in \mathsf{AutpGCL}, f, g \in \mathsf{AutExp}$, *is there some* $k \in \mathbb{N}$ *such that* $[\![g]\!]$ *is* k*-inductive for* C *w.r.t.* $\mathcal{T}$ *and* $[\![f]\!]$*?*
2. *Given* $C \in \mathsf{AutpGCL}, f, g \in \mathsf{AutExp}$, *does* $\mathcal{T}[\![C]\!]([\![f]\!]) \not\sqsubseteq [\![g]\!]$ *hold?*

Remark 4.1 (On the Language of Expectations from Chapter 3).
The expressive language Exp of syntactic expectations and the pGCL language presented in Chapter 3 is not suitable for automation in the sense of Definition 4.2: Quantitative entailments in Exp are undecidable by Theorem 3.20. We may nonetheless attempt to automate the reasoning with Exp as is done by the automated deductive verifier Caesar presented in [18]. Using the full power of Exp yields additional sources of incompleteness of Algorithms 1 and 2: In general, we must rely on incomplete heuristics or user-provided verification hints for discharging their termination conditions. Since such heuristics or user-provided hints are outside the scope of this thesis, suffice it to say here that they often succeed for various verification problems taken from the literature. See [18] for details.

4.6.3 Linear pGCL and Piecewise Linear Expectations

We now provide a concrete class of pGCL loops and expectations suitable for automation, i.e., for which Algorithms 1 and 2 are semi-decision procedures. Inspired by existing approaches for the automatic verification of both non-probabilistic [CSS03] and probabilistic programs [KMMM10], the key idea is to restrict to programs and expectations where all arithmetic involved is *linear*[6]. This enables to encode the necessary decision problems — the termination conditions of Algorithms 1 and 2 — equivalently as satisfiability problems of formulae over a decidable theory. In our case, these are formulae in quantifier-free fragment of linear mixed real-integer arithmetic (QF_LIRA). Similarly to classical software model checking (cf. Section 4.2.3), we can then decide the termination conditions by offloading these formulae to an SMT solver.

4.6.3.1 Definition

We assume the domain of the program variables to be $\mathsf{Vals} = \mathbb{N}$ so that the set of program states is given by

$$\mathsf{States} \;=\; \{\sigma \colon \mathsf{Vars} \to \mathbb{N} \;\mid\; \text{the set } \{x \in \mathsf{Vars} \;\mid\; \sigma(x) > 0\} \text{ is finite}\}\ .$$

This is a design choice based on the benchmarks considered in Section 4.6.4. The results presented here can be straightforwardly adapted to $\mathbb{Q}_{\geq 0}$-valued variables. Let us now consider the linear fragment of single loops in pGCL.

[6] In fact, we allow for *affine* arithmetic expressions but we adopt SMT terminology here.

Definition 4.3 (The Linear Fragment of pGCL).
Programs C in the *linear fragment of* pGCL, denoted by LpGCL, adhere to

$$
\begin{aligned}
C &\longrightarrow \texttt{while}(\varphi)\{C'\} && \text{(linear (single) loops)}\\
C' &\longrightarrow \texttt{skip} && \text{(effectless program)}\\
&\quad\mid x := a && \text{(assignment)}\\
&\quad\mid C' ; C' && \text{(sequential composition)}\\
&\quad\mid \{C'\}\,[p]\,\{C'\} && \text{(probabilistic choice)}\\
&\quad\mid \{C'\}\,\square\,\{C'\} && \text{(nondeterministic choice)}\\
&\quad\mid \texttt{if}\,(\varphi)\,\{C'\}\,\texttt{else}\,\{C'\}\,, && \text{(conditional choice)}
\end{aligned}
$$

where

1. $a \in \mathsf{AExpr}_{\mathbb{Z}}$ is a *linear arithmetic expression over* $\mathbb{Z}$ adhering to

$$
\begin{aligned}
a &\longrightarrow z \in \mathbb{Q}_{\geq 0} && \text{(integers)}\\
&\quad\mid x \in \mathsf{Vars} && \text{($\mathbb{N}$-valued variables)}\\
&\quad\mid a + a && \text{(addition)}\\
&\quad\mid z \cdot a\,, && \text{(scaling by constants in $\mathbb{Z}$)}
\end{aligned}
$$

2. $\varphi \in \mathsf{LBExpr}_{\mathbb{Z}}$ is a *linear Boolean expression over* $\mathbb{Z}$ adhering to

$$
\begin{aligned}
\varphi &\longrightarrow a < a && \text{(strict inequality of arithmetic expressions)}\\
&\quad\mid \varphi \wedge \varphi && \text{(conjunction)}\\
&\quad\mid \neg \varphi\,, && \text{(negation)}
\end{aligned}
$$

3. and where $p \in [0,1] \cap \mathbb{Q}$ is a rational probability.

The semantics $[\![a]\!] : \mathsf{States} \to \mathbb{Z}$ and $[\![\varphi]\!] \in \mathcal{P}(\mathsf{States})$ is defined by induction on the structure of a and φ, respectively, which is completely analogous to Definition 3.7 on page 85. We additionally require that every $C \in \mathsf{LpGCL}$ is type-safe[7] in the sense that whenever an assignment $x := a$ is executed on some state σ, then $[\![a]\!](\sigma) \in \mathbb{N}$. We adapt the usual order of precedence and syntactic sugar for arithmetic and Boolean operations from Section 3.2.

Next, we consider the formal language of piecewise linear expectations.

[7]Our implementation presented in Section 4.6.4 ensures this statically.

Definition 4.4 (Piecewise Linear Expectations adapted from [KMMM10]). *Piecewise linear (syntactic) expectations* in the set LExp adhere to the grammar

$$f \quad \longrightarrow \quad [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n ,$$

where $n \geq 1$ and

1. $\tilde{e}_i \in \mathsf{AExpr}_{\mathbb{Q}}$ is a *linear arithmetic expression over* $\mathbb{Q}$ adhering to

$$\begin{aligned} \tilde{e} \quad &\longrightarrow \quad \infty \mid e && \text{(infinity or finitely-valued expression)} \\ e \quad &\longrightarrow \quad q \in \mathbb{Q} && \text{(rationals)} \\ &\quad\mid\; x \in \mathsf{Vars} && \text{($\mathbb{N}$-valued variables)} \\ &\quad\mid\; e + e && \text{(addition)} \\ &\quad\mid\; q \cdot e , && \text{(scaling by constants in $\mathbb{Q}$)} \end{aligned}$$

2. $\vartheta_i \in \mathsf{LBExpr}_{\mathbb{Q}}$ is a *linear Boolean expression over* $\mathbb{Q}$ adhering to

$$\begin{aligned} \vartheta \quad &\longrightarrow \quad e < e && \text{(strict inequality of arithmetic expressions)} \\ &\quad\mid\; \vartheta \wedge \vartheta && \text{(conjunction)} \\ &\quad\mid\; \neg \vartheta , && \text{(negation)} \end{aligned}$$

3. and where for all $\sigma \in \mathsf{States}$, we have $[\![f]\!](\sigma) \geq 0$.

The *size of* f, denoted by $|f|$, is n — the number of summands in f. The semantics[8] $[\![f]\!] \colon \mathsf{States} \to \mathbb{Q} \cup \{\infty\}, [\![\tilde{e}]\!] \colon \mathsf{States} \to \mathbb{Q} \cup \{\infty\}$, and $[\![\vartheta]\!] \in \mathcal{P}(\mathsf{States})$ is defined by induction on the structure of f, e, and ϑ, respectively, which is completely analogous to Section 3.2.4. As usual, we write $\sigma \models \vartheta$ instead of $\sigma \in [\![\vartheta]\!]$. The last requirement[9] ensures that $[\![f]\!]$ is a well-defined expectation of type $\mathbb{E}$. Since this condition is decidable (cf. Lemma A.5 on page 275), LExp is indeed a computable formal language. Notice that, whereas arithmetic expressions occurring in pGCL range over $\mathbb{Z}$, arithmetic expressions in LExp range over $\mathbb{Q}$. This is reasonable since, with expectations, we want to reason about (rational) probabilities or more general expected outcomes. We will make heavy use of the following:

[8]We often omit the semantic brackets if their necessity is clear from the context.

[9]Analogously to Definition 3.4 on page 81, we could as well restrict to non-negative constants in $\mathbb{Q}_{\geq 0}$ and add a rule $e \mathbin{\dot{-}} e$ for monus. We opted for the presented choices to align the results presented here with the results from Chapter 5.

Lemma 4.15 (Decidability of Satisfiability for $\mathsf{LBExpr}_{\mathbb{Q}}$).
The satisfiability problem for $\mathsf{LBExpr}_{\mathbb{Q}}$

> *Given* $\vartheta \in \mathsf{LBExpr}_{\mathbb{Q}}$*, is there some* $\sigma \in \mathsf{States}$ *with* $\sigma \models \vartheta$*?*

is decidable.

Proof. ϑ is a formula in the quantifier-free fragment of linear mixed real-integer arithmetic (QF_LIRA) with free $\mathbb{N}$-valued variables in Vars. Since satisfiability of these formulae is decidable [KBT14], the claim follows. ∎

The reason why we need *mixed* arithmetic over both reals *and* integers is that variables are $\mathbb{N}$-valued whereas arithmetic expressions possibly contain constants in $\mathbb{Q}$. Hence, intuitively, we require *integer* arithmetic to reason about *program states* and *real* arithmetic to reason about *probabilities or more general expected outcomes*. From now on, we assume a black box — an SMT solver[10] — for deciding a linear Boolean expression's satisfiability.

Example 4.5.

1. Reconsider the geometric loop C from Example 4.4 given by
$$C \;=\; \texttt{while}(y=1)\{\{y:=0\}\,[1/2]\,\{x:=x+1\}\}\;.$$
We have $C \in \mathsf{LpGCL}$. Moreover, all the specifications[a] from Example 4.4 can be expressed in LExp, e.g.,
$$\mathsf{wp}[\![C]\!](x) \;\sqsubseteq\; [y=1]\cdot(x+1)+[y\neq 1]\cdot x\;.$$

2. Including ∞ in LExp is convenient for declaring certain initial states as irrelevant. For instance,
$$\mathsf{wp}[\![C]\!](x) \;\sqsubseteq\; [x=0]\cdot 1+[x\neq 0]\cdot\infty$$
specifies that the expected final value of x obtained from executing C on each initial state satisfying $x=0$ is at most 1. For all other initial states, the expected final value of x is irrelevant.

3. We have
$$[\mathsf{true}]\cdot(x-2)\notin \mathsf{LExp}$$
since for, e.g., a state σ with $\sigma(x)=0$, the above evaluates to -2.

[10] Our implementation uses the SMT solver Z3 [MB08]. See Section 4.6.4 for details.

4. Piecewise linear expectations impose restrictions on the properties that can be specified. Consider the loop $C \in \mathsf{LpGCL}$ given by

$$\texttt{while}(y > 0)\{x := x + z; y := y - 1\} .$$

C adds to the initial value of x the product $y \cdot z$ of the initial values of y and z. Hence, the expected (in fact: certain) final values of x satisfy

$$\mathsf{wp}[\![C]\!](x) \quad \sqsubseteq \quad \lambda\sigma.\sigma(x) \quad + \quad \underbrace{\sigma(y)\cdot\sigma(x)}_{\text{not expressible in LExp}} \quad .$$

This can not be specified within LExp since non-linear products of program variables are not expressible. Moreover, it follows that LExp is not expressive (cf. Definition 3.1).

[a]We often write $f = \tilde{e}$ instead of $f = [\mathsf{true}] \cdot \tilde{e}$ for the sake of readability.

4.6.3.2 The Guarded Normal Form

Next, we consider an effective normal form for LExp, which is convenient for both deciding quantitative entailments and constructing pointwise minima.

Definition 4.5 (Guarded Normal Form adapted from [KMMM10]). A piecewise linear expectation $f \in \mathsf{LExp}$

$$f \;=\; [\vartheta_1]\cdot\tilde{e}_1 + \ldots + [\vartheta_n]\cdot\tilde{e}_n$$

is in *guarded normal form (GNF)*, if the ϑ_i partition the set States, i.e., if for all $\sigma \in \mathsf{States}$, there is exactly one $i \in \{1, \ldots, n\}$ with $\sigma \models \vartheta_i$.

The intuition is that $f \in \mathsf{LExp}$ in GNF can be considered a complete case distinction: Given $\sigma \in \mathsf{States}$, the value $[\![f]\!](\sigma)$ is equal to $[\![\tilde{e}_i]\!](\sigma)$, where i is the *uniquely determined* index with $\sigma \models \vartheta_i$.

Example 4.6.

1. Consider the piecewise linear expectation

$$f \;=\; [x \le 1]\cdot y + [x \le 2]\cdot z\,.$$

f is not in GNF since $x \le 1$ and $x \le 2$ do not partition States. For, e.g., $\sigma \in \mathsf{States}$ with $\sigma(x) = 1$, we have $\sigma \models x \le 1$ and $\sigma \models x \le 2$.

Algorithm 3: PrunedGNF (f)

```
   input: $f \in \mathsf{LExp}$.
   output: $g \in \mathsf{LExp}$ in GNF with $[\![f]\!] = [\![g]\!]$.
1  return PrunedGNF ( $f, \mathsf{true}, 0$ )
2  Procedure PrunedGNF ( $f, \vartheta, \tilde{e}$ )
3      if $f = \epsilon$ then
4          return $[\vartheta] \cdot \tilde{e}$
       // Let $f = [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n$
5      $f_1, f_2 \leftarrow \epsilon$
6      if $\vartheta \wedge \vartheta_1$ is satisfiable then
7          $f_1 \leftarrow$ PrunedGNF ( $[\vartheta_2] \cdot \tilde{e}_2 + \ldots + [\vartheta_n] \cdot \tilde{e}_n, \vartheta \wedge \vartheta_1, \tilde{e} + \tilde{e}_1$ )
8      if $\vartheta \wedge \neg\vartheta_1$ is satisfiable then
9          $f_2 \leftarrow$ PrunedGNF ( $[\vartheta_2] \cdot \tilde{e}_2 + \ldots + [\vartheta_n] \cdot \tilde{e}_n, \vartheta \wedge \neg\vartheta_1, \tilde{e}$ )
10     return $f_1 + f_2$
```

Figure 4.5: Computing guarded normal forms with on-the-fly pruning of unsatisfiable Boolean expressions, where we use procedure overloading for a recursive implementation. Here ϵ denotes the empty word. Moreover, we let $\tilde{e} + \infty = \infty + \tilde{e} = \infty$ and $\epsilon + f = f + \epsilon = f$.

2. The piecewise linear expectation g given by
$$g \;=\; [x \leq 1] \cdot (y + z) + [x > 1 \wedge x \leq 2] \cdot z + [x > 2] \cdot 0\,,$$
on the other hand, *is* in GNF and moreover equivalent to f.

Lemma 4.16 (Effectively Constructing Guarded Normal Forms).
Let $f \in \mathsf{LExp}$. There is an effectively constructible piecewise linear expectation $\mathsf{GNF}(f) \in \mathsf{LExp}$ which is in GNF and equivalent to f, i.e,
$$[\![f]\!] \;=\; [\![\mathsf{GNF}(f)]\!] \quad \text{and} \quad \mathsf{GNF}(f) \text{ is in guarded normal form}\,.$$

Proof. We adapt the construction from [KMMM10]. Let
$$f \;=\; [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n \quad \in \quad \mathsf{LExp}\,.$$

The construction enumerates the assignments of truth values to the ϑ_i:
$$\begin{aligned} &\mathsf{GNF}(f) \\ =\; &\sum_{\left((\rho_1,\tilde{a}_1),\ldots,(\rho_n,\tilde{a}_n)\right) \in \bigtimes_{i=1}^{n} \left\{(\vartheta_i,\tilde{e}_i),(\neg\vartheta_i,0)\right\}} \left[\bigwedge_{i=1}^{n} \rho_i\right] \cdot \left(\sum_{i=0}^{n} \tilde{a}_i\right) \end{aligned} \tag{4.4}$$

where we rewrite every expression of the form $\tilde{a} + \infty$ or $\infty + \tilde{a}$ by ∞ to ensure that $\mathsf{GNF}(f)$ adheres to the grammar from Definition 4.4. It is immediate that the so-obtained Boolean expressions attached to each summand partition States. Moreover, $[\![f]\!] = [\![\mathsf{GNF}(f)]\!]$ holds since we have for every $\sigma \in \mathsf{States}$,
$$\begin{aligned} &[\![f]\!](\sigma) \\ =\; &\sum_{\{i \in \{1,\ldots,n\} \mid \sigma \models \vartheta_i\}} \tilde{e}_i \\ =\; &\sum_{\{i \in \{1,\ldots,n\} \mid \sigma \models \vartheta_i\}} \tilde{e}_i + \sum_{\{i \in \{1,\ldots,n\} \mid \sigma \models \neg\vartheta_i\}} 0 \\ =\; &[\![\mathsf{GNF}(f)]\!](\sigma)\,. \end{aligned}$$
■

Naively applying the construction from (4.4) yields $|\mathsf{GNF}(f)| = 2^{|f|}$. In practice, we can often avoid this exponential blow-up by pruning unsatisfiable Boolean expressions resulting from the construction on-the-fly: Let

$$(\rho_1, \tilde{a}_1), \ldots, (\rho_n, \tilde{a}_n) \in \bigtimes_{i=1}^{n} \big\{(\vartheta_i, \tilde{e}_i), (\neg\vartheta_i, 0)\big\}$$

and assume we determine for some $j \in \{1, \ldots, n\}$ that the Boolean expression

$$\bigwedge_{i=1}^{j} \rho_j$$

is unsatisfiable. It follows that *all* summands in $\mathsf{GNF}(f)$ resulting from

$$(\rho_1, \tilde{a}_1), \ldots, (\rho_j, \tilde{a}_j), (\rho'_{j+1}, \tilde{a}'_{j+1}), \ldots, (\rho'_n, \tilde{a}'_n) \in \bigtimes_{i=1}^{n} \big\{(\vartheta_i, \tilde{e}_i), (\neg\vartheta_i, 0)\big\}$$

can be omitted — and must hence not be enumerated — because

$$\underbrace{\underbrace{\bigwedge_{i=1}^{j} \rho_i}_{\text{unsatisfiable}} \quad \wedge \quad \bigwedge_{i=j+1}^{n} \rho'_i}_{\text{unsatisfiable}}$$

will as well be unsatisfiable. Algorithm 3 on page 150 implements this optimization, where we enumerate the possible assignments of truth values to the ϑ_i recursively. If we encounter the situation described above, i.e., if during enumeration we determine that the currently constructed Boolean expression is unsatisfiable, we avoid enumerating the remaining conjuncts (lines 6 and 8).

Example 4.7.
Let $n \geq 1$ and consider the family of piecewise linear expectations

$$f_n \;=\; [x = 1] \cdot y_1 + \ldots + [x = n] \cdot y_n \,.$$

f_n is not in GNF because, e.g., for $\sigma \in \mathsf{States}$ with $\sigma(x) = n + 1$, there is no $i \in \{1, \ldots, n\}$ with $\sigma \models x = i$. Applying the construction from Lemma 4.16 yields $|\mathsf{GNF}(f_n)| \;=\; 2^n$. With Algorithm 3, on the other hand, we get

$$\mathtt{PrunedGNF}(f_n) \quad = \quad \left[\bigwedge_{i=1}^{n} x \neq i\right] \cdot 0 \;+\; \sum_{i=1}^{n} \left[x = i \bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} x \neq j\right] \cdot y_i \,.$$

with $|\mathtt{PrunedGNF}(f_n)| = n + 1$. We thus obtain much more concise GNFs.

4.6.3.3 Deciding Quantitative Entailments

Let $f_1, f_2 \in \mathsf{LExp}$. To decide the validity of the quantitative entailment

$$[\![f_1]\!] \sqsubseteq [\![f_2]\!] ,$$

we effectively construct[11] a Boolean expression $\mathsf{CEX}_{\sqsubseteq}(f_1, f_2) \in \mathsf{LBExpr}_{\mathbb{Q}}$ such that

$$\begin{aligned} & \mathsf{CEX}_{\sqsubseteq}(f_1, f_2) \text{ is satisfiable} \\ \text{iff} \quad & \text{exists } \sigma \in \mathsf{States}\colon\ [\![f_1]\!](\sigma) > [\![f_2]\!](\sigma) . \end{aligned} \tag{4.5}$$

Thus, intuitively, $\mathsf{CEX}_{\sqsubseteq}(f_1, f_2)$ is satisfiable iff there is a *counterexample* to the validity of the quantitative entailment $[\![f_1]\!] \sqsubseteq [\![f_2]\!]$, and thus *unsatisfiable* iff the quantitative entailment holds. To obtain $\mathsf{CEX}_{\sqsubseteq}(f_1, f_2)$, we first construct

$$\mathsf{GNF}(f_1) = [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n \quad \text{and} \quad \mathsf{GNF}(f_2) = [\eta_1] \cdot \tilde{a}_1 + \ldots + [\eta_m] \cdot \tilde{a}_m .$$

using Algorithm 3. The sought-after Boolean expression is then given by

$$\begin{aligned} & \mathsf{CEX}_{\sqsubseteq}(f_1, f_2) \\ = \quad & \bigvee_{i=1}^{n} \bigvee_{j=1}^{m} \quad \vartheta_i \wedge \eta_j \wedge \begin{cases} \mathsf{true} & \text{if } \tilde{e}_i = \infty, \tilde{a}_j \neq \infty \\ \mathsf{false} & \text{if } \tilde{a}_j = \infty \\ \tilde{e}_i > \tilde{a}_j & \text{otherwise} . \end{cases} \end{aligned}$$

The latter is indeed a linear Boolean expression over $\mathbb{Q}$ adhering to the grammar from Definition 4.4.2 since we eliminated all potential occurrences of ∞. Moreover, $\mathsf{CEX}_{\sqsubseteq}(f_1, f_2)$ satisfies (4.5) because we have

$$\begin{aligned} & [\![f_1]\!] \sqsubseteq [\![f_2]\!] \\ \text{iff} \quad & [\![\mathsf{GNF}(f_1)]\!] \sqsubseteq [\![\mathsf{GNF}(f_2)]\!] && \text{(Lemma 4.16)} \\ \text{iff} \quad & \text{for all } \sigma \in \mathsf{States}\colon\ [\![\mathsf{GNF}(f_1)]\!](\sigma) \leq [\![\mathsf{GNF}(f_2)]\!](\sigma) \\ \text{iff} \quad & \text{for all } \sigma \in \mathsf{States} \text{ and } (i,j) \in \{1,\ldots,n\} \times \{1,\ldots,m\}\colon \\ & \sigma \models \vartheta_i \text{ and } \sigma \models \eta_j \quad \text{implies} \quad [\![\tilde{e}_i]\!](\sigma) \leq \left[\!\!\left[\tilde{a}_j\right]\!\!\right](\sigma) \\ & && \text{(both the } \vartheta_i \text{ and the } \eta_j \text{ partition } \mathsf{States}) \\ \text{iff} \quad & \text{for all } \sigma \in \mathsf{States} \text{ and } (i,j) \in \{1,\ldots,n\} \times \{1,\ldots,m\}\colon \end{aligned}$$

[11] This construction is adapted from [KMMM10, Lemma 3].

$$\sigma \models \vartheta_i \text{ and } \sigma \models \eta_j \quad \text{implies} \quad \begin{cases} \mathsf{true} & \text{if } \tilde{a}_j = \infty \\ \mathsf{false} & \text{if } \tilde{e}_i = \infty, \tilde{a}_j \neq \infty \\ [\![\tilde{e}_i]\!](\sigma) \leq [\![\tilde{a}_j]\!](\sigma) & \text{otherwise} \end{cases}$$

(eliminate ∞)

iff for all $\sigma \in \mathsf{States}$:

$$\sigma \models \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \vartheta_i \wedge \eta_j \quad \Longrightarrow \quad \begin{cases} \mathsf{true} & \text{if } \tilde{a}_j = \infty \\ \mathsf{false} & \text{if } \tilde{e}_i = \infty, \tilde{a}_j \neq \infty \\ \tilde{e}_i \leq \tilde{a}_j & \text{otherwise} \end{cases}$$

(rewrite as Boolean expression)

iff for all $\sigma \in \mathsf{States}$:

$$\sigma \not\models \bigvee_{i=1}^{n} \bigvee_{j=1}^{m} \vartheta_i \wedge \eta_j \wedge \begin{cases} \mathsf{true} & \text{if } \tilde{e}_i = \infty, \tilde{a}_j \neq \infty \\ \mathsf{false} & \text{if } \tilde{a}_j = \infty \\ \tilde{e}_i > \tilde{a}_j & \text{otherwise} \end{cases}$$

(double-negate)

iff $\mathsf{CEX}_{\sqsubseteq}(f_1, f_2)$ is unsatisfiable .

Since the latter is decidable by Lemma 4.15, we conclude:

Lemma 4.17 (Decidability of Quantitative Entailments in LExp). *Quantitative entailments* in LExp are decidable, i.e.,

Given $f_1, f_2 \in \mathsf{LExp}$, *does* $[\![f_1]\!] \sqsubseteq [\![f_2]\!]$ *hold?*

is decidable.

4.6.3.4 Constructing Pointwise Minima and Maxima

Lemma 4.18 (Constructing Pointwise Minima and Maxima for LExp). LExp is effectively closed under both pointwise minima and maxima, i.e., for all $f_1, f_2 \in \mathsf{LExp}$, there are effectively constructible

1. $f_1 \sqcap f_2 \in \mathsf{LExp}$ with $[\![f_1 \sqcap f_2]\!] = [\![f_1]\!] \sqcap [\![f_2]\!]$, and
2. $f_1 \sqcup f_2 \in \mathsf{LExp}$ with $[\![f_1 \sqcup f_2]\!] = [\![f_1]\!] \sqcup [\![f_2]\!]$.

Proof. We first construct the GNF's of f_1 and f_2 using Algorithm 3 and let

$$\mathsf{GNF}(f_1) \;=\; [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n \text{ ,and}$$

$$\mathsf{GNF}(f_2) \;=\; [\eta_1]\cdot\tilde{a}_1 + \ldots + [\eta_m]\cdot\tilde{a}_m\,.$$

We then construct $f_1 \sqcap f_2$ as follows:

$$\sum_{(i,j)\in\{1,\ldots,n\}\times\{1,\ldots,m\}} \begin{cases} \left[\vartheta_i \wedge \eta_j\right]\cdot\tilde{a}_j & \text{if } \tilde{e}_i = \infty \\ \left[\vartheta_i \wedge \eta_j\right]\cdot\tilde{e}_i & \text{if } \tilde{a}_j = \infty \\ \left[\vartheta_i \wedge \eta_j \wedge \tilde{e}_i \leq \tilde{a}_j\right]\cdot\tilde{e}_i + \left[\vartheta_i \wedge \eta_j \wedge \tilde{e}_i > \tilde{a}_j\right]\cdot\tilde{a}_j & \text{otherwise} \end{cases}$$

The soundness of this construction is an immediate consequence of the fact that both the ϑ_i and the η_j partition the state space by Definition 4.5. Notice that $f_1 \sqcap f_2$ is in GNF. The construction for $f_1 \sqcup f_2$ is dual:

$$\sum_{(i,j)\in\{1,\ldots,n\}\times\{1,\ldots,m\}} \begin{cases} \left[\vartheta_i \wedge \eta_j\right]\cdot\tilde{e}_i & \text{if } \tilde{e}_i = \infty \\ \left[\vartheta_i \wedge \eta_j\right]\cdot\tilde{a}_j & \text{if } \tilde{a}_j = \infty \\ \left[\vartheta_i \wedge \eta_j \wedge \tilde{e}_i \leq \tilde{a}_j\right]\cdot\tilde{a}_i + \left[\vartheta_i \wedge \eta_j \wedge \tilde{e}_i > \tilde{a}_j\right]\cdot\tilde{e}_j & \text{otherwise} \end{cases}$$

Notice that $f_1 \sqcup f_2$ is in GNF. ■

With the preceding results at hand, we finally conclude:

Theorem 4.19 (Suitability for Automation).
$(\mathsf{LpGCL}, \mathsf{LExp})$ is suitable for automation (cf. Definition 4.2 on page 144). Hence, Algorithms 4 and 5 on page 156 are semi-decision procedures.

Proof. The fact that LExp is effectively closed under characteristic functions of loops in $C \in \mathsf{LpGCL}$ follows by induction on the structure of C. This is essentially analogous to the proof of Theorem 3.2 on page 90 where the additional cases for nondeterministic choices follow from Lemma 4.18. Together with Lemma 4.17, this implies the claim. ■

4.6.4 Implementation and Experiments

We have implemented a prototype called KIPRO2[12] — k-Induction for PRObabilistic PROgrams — in Python 3.11 using the SMT solver Z3 [MB08] and the solver-API PySMT [GM15]. KIPRO2 performs latticed k-induction or BMC as depicted in Figure 4.6 to fully automatically verify or refute upper bounds

[12]We have adapted our original implementation presented in [11; 10] so that it matches the formalization presented in this thesis. The prototype is available open-source from https://github.com/KevinBatz/kipro2-diss.

Algorithm 4: k-induction for LpGCL

```
input: $C \in \mathsf{LpGCL}, f, g \in \mathsf{LExp}$.
output: "verify" if there is $k$ such that $g$ is $k$-inductive for $C$ w.r.t. $f$,
        diverge otherwise.
$h \leftarrow g$
while true do
    $h \leftarrow {}^{\mathsf{wp}}_{C}\Phi_{f}(h)$   // Table 2.1 on page 60
    if $\mathsf{CEX}_{\sqsubseteq}(h, g)$ is unsatisfiable then // Section 4.6.3.3
        return verify
    $h \leftarrow h \sqcap g$ // Section 4.6.3.4
```

Algorithm 5: BMC for LpGCL

```
input: $C \in \mathsf{LpGCL}, f, g \in \mathsf{LExp}$.
output: "refute" if $\mathsf{wp}[\![C]\!](f) \not\sqsubseteq g$, diverge otherwise.
$h \leftarrow 0$
while true do
    $h \leftarrow {}^{\mathsf{wp}}_{C}\Phi_{f}(h)$   // Table 2.1 on page 60
    if $\mathsf{CEX}_{\sqsubseteq}(h, g)$ is satisfiable then // Section 4.6.3.3
        return refute
```

Figure 4.6: k-Induction and BMC for LpGCL as implemented in KIPRO2 using the techniques from Section 4.6.3. We omit semantic brackets of piecewise linear expectations for the sake of readability.

on expected values of programs C in the *fully probabilistic*[13] linear fragment of pGCL and piecewise linear expectations. We describe the benchmarks in Section 4.6.4.1 and discuss empirical results in Section 4.6.4.2.

4.6.4.1 Benchmarks

Table 4.1 on page 158 depicts the postexpectations f and the candidate upper bounds g we considered for each of the programs given below. Moreover, column "method" indicates whether the given bound is to be verified by means of k-induction (Algorithm 4) or refuted by means of BMC (Algorithm 5).

The Bounded Retransmission Protocol (cf. Section 4.1).

```
while($sent < N \wedge failed \leq M$){
    {
        $failed := failed + 1$;
        $totalFailed := totalFailed + 1$
    } [0.1] {
        $failed := 0$;
        $sent := sent + 1$
    }
}
```

We reason about candidate upper bounds on the expected final value of *totalFailed* for varying bounds on the number N of packets to transmit.

*The Geometric Loop (*geo*).* This is the program from Example 4.5 on page 148.

```
while($y = 1$){{$y := 0$} [1/2] {$x := x + 1$}}
```

We reason about candidate upper bounds on the expected final value of x.

*Rabin's Mutual Exclusion Protocol (*rabin*) [KR92].* The program shown in Figure 4.7 on page 159 models Rabin's protocol, which utilizes randomization to break the symmetry in a scenario where a unique leader among x participating

[13]The deductive verifier Caesar [18], which we developed two years later, also supports nondeterministic programs but is outside the scope of this thesis.

	variant	method	f	g
brp	1	k-ind.	$\mathit{totalFailed}$	$[N \leq 4] \implies (\mathit{totalFailed} + 1)$
	2	k-ind.		$[N \leq 6] \implies (\mathit{totalFailed} + 3)$
	3	k-ind.		$[N \leq 8] \implies (\mathit{totalFailed} + 3)$
	4	k-ind.		$[N \leq 20] \implies (\mathit{totalFailed} + 8)$
	5	BMC		$\mathit{totalFailed}0.25$
	6	BMC		$\mathit{totalFailed} + 0.5$
	7	BMC		$\mathit{totalFailed} + 0.75$
	8	BMC		$\mathit{totalFailed} + 1$
geo	1	k-ind.	x	$[y = 1] \cdot (x + 1) + [y \neq 1] \cdot x$
	2	k-ind.		$x + 1$
	3	BMC		x
	4	BMC		$x + 0.9$
	5	BMC		$x + 0.9999999999999$
rabin	1	k-ind.	$[x \neq 1]$	$[1 < x \wedge x < 3 \wedge \mathit{phase} = 0] \implies 1/3$
	2	k-ind.		$[1 < x \wedge x < 4 \wedge \mathit{phase} = 0] \implies 1/3$
	3	k-ind.		$[1 < x \wedge x < 5 \wedge \mathit{phase} = 0] \implies 1/3$
	4	k-ind.		$[1 < x \wedge x < 6 \wedge \mathit{phase} = 0] \implies 1/3$
	5	BMC		$[1 < x \wedge \mathit{phase} = 0] \implies 0.2$
	6	BMC		$[1 < x \wedge \mathit{phase} = 0] \implies 0.225$
	7	BMC		$[1 < x \wedge \mathit{phase} = 0] \implies 0.25$
fdr	1	k-ind.	$[x = i]$	$[\vartheta_1] \implies 1/2$
	2	k-ind.		$[\vartheta_2] \implies 1/3$
	3	k-ind.		$[\vartheta_3] \implies 1/4$
	4	k-ind.		$[\vartheta_4] \implies 1/5$
	5	BMC		$[\vartheta_2] \implies 1/4$
	6	BMC		$[\vartheta_3] \implies 1/5$
	7	BMC		$[\vartheta_4] \implies 1/6$

Table 4.1: Specifications for the programs given in Section 4.6.4.1. Here $[\vartheta] \implies \tilde{e}$ is a shorthand for $[\vartheta] \cdot \tilde{e} + [\neg\vartheta] \cdot \infty$.

```
while($1 < x \vee phase = 1$){
    if ($phase = 0$) {
        $y := x$; $phase := 1$
    }else{
        if ($0 < y \wedge 1 \leq x$) {
            {$z := 0$} [$1/2$] {$z := 1$};
            $x := x - z$;
            $y := y - 1$
        }else{
            $phase := 0$
        }
    }
}
```

Figure 4.7: A pGCL program adapted from [HMM05] modeling Rabin's mutual exclusion protocol [KR92].

```
while($running = 0$){
    $y := 2 \cdot y$;
    {$x := 2 \cdot x + 1$} [$1/2$] {$x := 2 \cdot x$};
    if ($y \geq N$) {
        if ($x < N$) {
            $running := 1$
        }else{
            $y := y - N$; $x := x - N$
        }
    }else{
        if ($running \neq 0$) {$x := low + x$} else {skip}
    }
}
```

Figure 4.8: A pGCL program adapted from [Lum13] modeling Lumbroso's algorithm for sampling uniformly from a finite interval of natural numbers using fair coin flips only.

processes is to be elected. In this model, the protocol succeeds if it terminates with $x = 1$. If it terminates in $x \neq 1$, the protocol fails.

We reason about candidate upper bounds on the probability that the protocol fails for varying bounds on the number x of participating processes.

*The Fast Dice Roller (*fdr*) [Lum13].* The program shown in Figure 4.8 on page 160 models Lumbroso's algorithm for sampling uniformly from a given finite interval of natural numbers using fair coin flips only. If initially $running = 0, x = 0, y = 1, low \leq high$, and $N = high - low + 1$, then the probability of terminating in some state with $x \in \{low, \ldots, high\}$ is $1/N$.

We reason about candidate upper bounds on the probability of terminating in some state $x \in \{low, \ldots, high\}$ for varying sizes of the interval. To specify this in terms of weakest preexpectations, we introduce an auxiliary program variable

$i \in$ Vars to fix some final value of variable x. We then specify the initial program states of interest by defining for each $n \in \mathbb{N}$ the Boolean exprresion ϑ_n given by

$$\begin{aligned}&\mathit{running} = 0 \wedge x = 0 \wedge y = 1\\ &\wedge \mathit{low} + n = \mathit{high} \wedge N = \mathit{high} - \mathit{low} + 1 \wedge \mathit{low} \leq i \leq \mathit{high}\ .\end{aligned}$$

ϑ_n specifies the initial values of $\mathit{running}, x, y$, and that we aim to sample from some interval $\{\mathit{low}, \ldots, \mathit{high}\}$ with $|\{\mathit{low}, \ldots, \mathit{high}\}| = n + 1$.

4.6.4.2 Results

We evaluate KIPRO2 on the benchmark set described in the previous section. The experiments were run on an Apple M2 with 24GB RAM and a 5-minute timeout. The results are depicted in Table 4.2. Runtimes are given in seconds. Column "variant" indicates the candidate upper bound. "result" indicates whether the candidate upper bound was verified or refuted, and "k" indicates that the candidate was shown to be k-inductive or that k iterations of the wp-characteristic functional were needed to refute the bound via BMC. "size" indicates $|h|$, where h is the piecewise linear expectation in GNF constructed for the last satisfiability check. "construction_t" is the time spent on wp computations, computing GNFs, and pointwise minima. "entailment_t" is the time spent on checking the satisfiability of $\mathsf{CEX}_{\sqsubseteq}(h, g)$. Finally, "total_t" is the total time.

The results depicted in Table 4.2 empirically underline that deductive probabilistic program verification can benefit from k-induction and BMC to the same extent as classical software verification: KIPRO2 *fully automatically* verifies or refutes relevant properties of *infinite-state* randomized algorithms and stochastic processes from the literature that require *k to be strictly larger than* 1. That is, proving these properties using 1-induction, i.e., wp-superinvariants, requires either invariant synthesis or additional user annotations.

We observe that k-induction tends to succeed if *some* variable is bounded in the candidate upper bound under consideration (cf. brp, rabin, fdr). However, k-induction can also succeed without any bounds (cf. geo). The size of piecewise linear expectations and the time required for checking k-inductivity can increase rapidly for larger k; this is particularly striking for brp and rabin. When refuting candidate bounds with BMC, we obtain a similar picture.

Finally, we observe that most of the computation time is spent on computing GNFs and pointwise minima. The satisfiability checks for $\mathsf{CEX}_{\sqsubseteq}(h, g)$, on the other hand, are rather quick. We conjecture that the scalability of k-induction and BMC can be improved by investigating more sophisticated pruning tech-

	variant	result	k	size	construction_t	entailment_t	total_t
brp	1	ind	5	52	0.52	0.01	0.56
	2	ind	7	174	8.4	0.03	8.45
	3	ind	9	424	121.02	0.76	121.81
	4	TO	–	–	–	–	–
	5	ref	4	26	0.04	<0.01	0.07
	6	ref	8	212	6.63	0.03	6.68
	7	TO	–	–	–	–	–
	8	TO	–	–	–	–	–
geo	1	ind	1	2	<0.01	<0.01	0.03
	2	ind	2	2	<0.01	<0.01	0.02
	3	ref	3	2	<0.01	<0.01	0.01
	4	ref	8	2	0.01	<0.01	0.03
	5	ref	50	2	0.21	0.01	0.23
rabin	1	ind	5	36	0.95	0.01	0.99
	2	ind	6	55	3.18	0.02	3.24
	3	ind	7	78	10.07	0.04	10.16
	4	ind	8	105	30.52	0.12	30.68
	5	ref	5	44	0.47	0.01	0.51
	6	ref	5	44	0.47	0.01	0.51
	7	ref	9	130	31.47	0.15	31.65
fdr	1	ind	2	7	3.54	<0.01	3.62
	2	ind	3	26	1.5	0.01	1.59
	3	ind	3	24	1.38	0.01	1.47
	4	TO	–	–	–	–	–
	5	TO	–	–	–	–	–
	6	ref	3	110	1.01	0.01	1.12
	7	TO	–	–	–	–	–

Table 4.2: Experimental results for the benchmarks described in Section 4.6.4.1. TO = 5min. Runtimes are given in seconds.

niques. For instance, KIPRO2 first computes ${}^{\mathsf{wp}}_{C}\Phi_f(h)$ monolithically and then computes the GNF of the so-obtained expectation. It might pay off to already simplify the intermediate expectations obtained *during* the computation of ${}^{\mathsf{wp}}_{C}\Phi_f(h)$. Such improvements are left for future work.

4.7 Future and Related Work

Future Work. Latticed k-induction applies to all verification problems that can be phrased as upper-bounding the least fixpoint of a monotone function over a complete lattice. A natural direction for future work is to investigate whether latticed k-induction can be applied — both theoretically and practically — to other domains. A particularly promising direction is to apply latticed k-induction to the verification of *weighted programs* [15]. The bachelor's thesis of Ben Sturgis [Stu22] shows promise: Sturgis successfully applied latticed k-induction to automatically verify properties of weighted programs over the tropical semiring. We strongly conjecture that latticed k-induction yields automated verification techniques for more instances of weighted programming.

Related Work. k-induction for transition systems has been introduced by Sheeran et al. [SSS00]. Bounded model checking for transition systems is due to Clarke et al. [CBRZ01]. Since then, k-Induction and BMC have been extensively studied and applied in the field of hardware and software model checking [DHKR11; BDW15; JD16; GI17; RICB15; KVGG19; DKR10; DKR11]. To the best of our knowledge, we were the first to apply k-induction to the automated deductive verification of probabilistic programs. BMC has been applied for reasoning about conditional reachability probabilities in finite unfoldings of probabilistic programs [JDKK+16], and the enumerative generation of counterexamples in *finite* Markov chains [WBB09; JÁZW+12]. The aforementioned approaches either restrict to finite-state programs or fix some initial program state. Our approach does not impose these restrictions. However, our implementation is restricted to linear programs and specifications, and does currently not support reasoning about *conditional* reachability probabilities. Moreover, latticed k-induction has been integrated into techniques for the automatic verification of probabilistic pushdown automata [WK23a]. We discuss more approaches for the automatic verification of probabilistic programs in Section 5.9.

When applied to finite-state Markov chains, our κ-induction operator is related to other operators that have been employed for determining reachability probabilities through value iteration [QK18; BKLP+17; HK20]. In particular, when iterated on the candidate upper bound, the κ-induction operator coincides with the (upper value iteration) operator in interval iteration [BKLP+17]; the latter operator can be used together with up-to techniques[14] (cf. [Mil89; Pou07; PS12]) to prove our κ-induction (Theorem 4.5 on page 135) sound.

[14]The author thanks Joshua Moerman for this insight.

5 Automatic Loop Invariant Synthesis

This chapter is based on our prior publications [16; 13].

In this chapter, we present a *Counterexample-Guided Inductive Synthesis (CEGIS)* [ABDF⁺15; SRBE05; STBS⁺06] approach for synthesizing quantitative invariants of probabilistic loops. Our approach enables the fully automatic verification of both *upper and lower bounds on expected outcomes* of loops as well as the automatic verification of *universal positive almost-sure termination* by synthesizing upper bounds on a loop's expected runtime. Our implementation shows promise: It finds invariants of various loops taken from the literature, can beat state-of-the-art probabilistic model checkers, and is competitive with modern tools dedicated to invariant synthesis or expected runtime reasoning.

Assumptions. In order to simplify the presentation, we assume the set Vars of program variables to be *finite*. We moreover assume that program variables $x \in \mathsf{Vars}$ are $\mathbb{N}$-valued, i.e, the countable set of program states is given by

$$\mathsf{States} \;=\; \{\sigma \mid \sigma\colon \mathsf{Vars} \to \mathbb{N}\}\;.$$

Finally, we assume that all programs considered throughout are fully probabilistic and that all piecewise linear expectations (cf. Definition 4.4 on page 147) are in Guarded Normal Form (GNF) (cf. Definition 4.5 on page 149).

5.1 Motivation and Problem Statement

We start by considering the conceptually simpler problem of automatically verifying upper bounds on expected outcomes of probabilistic loops:

Given a loop $C = \mathtt{while}\,(B)\,\{C'\}$ *and* $X, Y \in \mathbb{E}$,
automatically verify that $\mathsf{wp}[\![C]\!]\,(X) \sqsubseteq Y$ *holds.*

Let us recap the approaches from the preceding chapters for tackling this problem, which will yield our motivation for considering loop invariant synthesis.

We have seen in Section 2.4.4 that the above problem can be tackled via *quantitative loop invariants*. By Theorem 2.12 on page 66, we have[1]

$$\underbrace{{}^{\mathsf{wp}}_{C}\Phi_X(Y) \sqsubseteq Y}_{Y \text{ is wp-superinvariant of } C \text{ w.r.t. } X} \qquad \text{implies} \qquad \mathsf{wp}[\![C]\!](X) \sqsubseteq Y\,. \tag{5.1}$$

If C is in the linear fragment[2] of pGCL and X, Y are piecewise linear expectations, we can employ the techniques from Section 4.6.3 implemented by our tool KIPRO2 to *decide* whether Y is a wp-superinvariant of C w.r.t. X. For instance, KIPRO2 automatically verifies that for the geometric loop

$$C \;=\; \mathtt{while}\,(y=1)\,\{\{y:=0\}\,[1/2]\,\{x:=x+1\}\}\,,$$

the following specification holds by (5.1):

$$\mathsf{wp}[\![C]\!](x) \;\sqsubseteq\; [y=1]\cdot(x+1)+[y\neq 1]\cdot x$$

The converse direction of (5.1) does, however, *not hold in general*, i.e., potentially

$$\mathsf{wp}[\![C]\!](X) \sqsubseteq Y \qquad \text{but} \qquad \underbrace{{}^{\mathsf{wp}}_{C}\Phi_X(Y) \not\sqsubseteq Y}_{Y \text{ is } \underline{\textit{not}} \text{ a wp-superinvariant of } C \text{ w.r.t. } X}\,.$$

This is, e.g., the case for the geometric loop C and the specification

$$\mathsf{wp}[\![C]\!](x) \;\sqsubseteq\; x+1\,. \tag{5.2}$$

We have then seen in Chapter 4 that k-induction for probabilistic programs can mitigate this problem: Even though $Y = x+1$ is not a wp-superinvariant of C w.r.t. $X = x$ (or, phrased in the terminology of k-induction: Y is not 1-inductive) Y is 2-inductive. We have

$$\underbrace{{}^{\mathsf{wp}}_{C}\Phi_X\left({}^{\mathsf{wp}}_{C}\Phi_X(Y)\sqcap Y\right) \sqsubseteq Y}_{Y \text{ is 2-inductive for } C \text{ w.r.t. } X} \qquad \underbrace{\text{and thus}}_{\text{Corollary 4.12 on page 141}} \qquad \mathsf{wp}[\![C]\!](X) \sqsubseteq Y\,.$$

In the piecewise linear setting, KIPRO2 semi-decides whether there is some $k \in \mathbb{N}$ such that Y is k-inductive for C w.r.t. X, thereby verifying (5.2) automatically. However, k-induction is incomplete in the sense that we might have

$$\mathsf{wp}[\![C]\!](X) \sqsubseteq Y \qquad \text{but} \qquad \begin{array}{c}\text{there is no } k\in\mathbb{N} \text{ such that}\\ Y \text{ is } k\text{-inductive for } C \text{ w.r.t. } X\,.\end{array}$$

[1] Recall from Definition 2.21 on page 63 that ${}^{\mathsf{wp}}_{C}\Phi_X$ is the characteristic function of C w.r.t. X.
[2] Recall the (piecewise) linear setting from Section 4.6.3.

This is, e.g., the case for the geometric loop C and the specification

$$\mathsf{wp}[\![C]\!](x) \;\sqsubseteq\; [x=0]\cdot 1 + [x \neq 0]\cdot \infty\,. \tag{5.3}$$

What can we do in such a non-k-inductive scenario? We can attempt to synthesize what we call an *admissible invariant* I, i.e., we

$$\text{search for an } I \in \mathbb{E} \qquad \text{satisfying} \qquad \underbrace{\underbrace{{}^{\mathsf{wp}}_{C}\Phi_X(I) \sqsubseteq I}_{I \text{ is } \textit{inductive}} \text{ and } \underbrace{I \sqsubseteq Y}_{I \text{ is } \textit{safe}}}_{I \text{ is } \textit{admissible}}\,.$$

The existence of an admissible invariant I witnesses that $\mathsf{wp}[\![C]\!](X) \sqsubseteq Y$ holds:

$$\begin{aligned}
& {}^{\mathsf{wp}}_{C}\Phi_X(I) \sqsubseteq I \text{ and } I \sqsubseteq Y && (I \text{ is admissible invariant})\\
\text{implies}\quad & \mathsf{wp}[\![C]\!](X) \sqsubseteq I \text{ and } I \sqsubseteq Y && (\text{Theorem 2.12 on page 66})\\
\text{implies}\quad & \mathsf{wp}[\![C]\!](X) \sqsubseteq Y && (\text{transitivity of } \sqsubseteq)
\end{aligned}$$

For instance, we can verify (5.3) by synthesizing the admissible invariant

$$I \;=\; [y=1]\cdot(x+1) + [y \neq 1]\cdot x\,.$$

The techniques presented in this chapter *enable the fully automatic synthesis of* I.

5

Formal Problem Statement. We will heavily build upon the techniques from Section 4.6.2, exploiting that the piecewise linear setting is particularly suitable for automation as both inductivity and safety of $I \in \mathsf{LExp}$ are decidable for loops in LpGCL and postexpectations in LExp (cf. Theorem 4.19 on page 155).

We denote by FinLExp $\subset$ LExp the *set of piecewise linear expectations not containing* ∞. In this chapter, we restrict to postexpectations and invariants in FinLExp. Given $h \in \mathsf{LExp}$, we often omit the semantic brackets in $[\![h]\!]$, if it is clear from the context that we speak about the expectation in $\mathbb{E}$ described by h.

Definition 5.1 (Admissible Invariants).
Let $C = \mathtt{while}(\varphi)\{C'\} \in \mathsf{LpGCL}$, $f \in \mathsf{FinLExp}$, and let $g \in \mathsf{LExp}$. The *set of admissible invariants (for verifying* $\mathsf{wp}[\![C]\!](f) \sqsubseteq g$*)* is defined as

$$\mathsf{AdmInv} \;=\; \left\{ I \in \mathsf{FinLExp} \;\middle|\; {}^{\mathsf{wp}}_{C}\Phi_f(I) \sqsubseteq I \text{ and } I \sqsubseteq g \right\}.$$

The loop C as well as f, g will always be clear from the context. With these notions at hand, we formalize our problem statement.

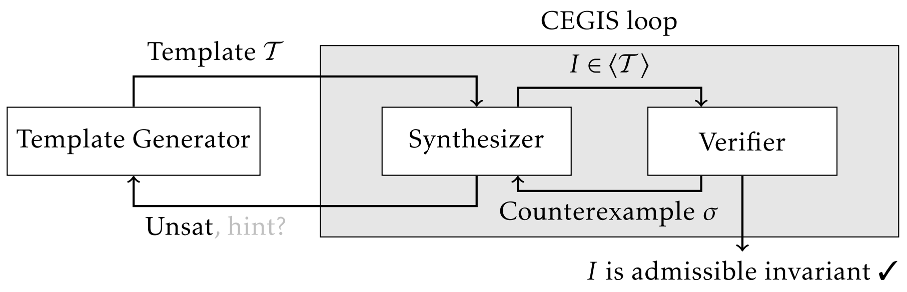


Figure 5.1: Our CEGIS framework for synthesizing admissible invariants.

Given $C = \texttt{while}(\varphi)\{C'\} \in \mathsf{LpGCL}$, $f \in \mathsf{FinLExp}$, and $g \in \mathsf{LExp}$,
if it exists, synthesize — in a fully automatic manner — some $I \in \mathsf{AdmInv}$.

Remark 5.1 (Extensions for Verifying Termination and Lower Bounds). We have so far considered the verification of about upper bounds on expected outcomes of probabilistic loops. We discuss extensions of our techniques for verifying (i) *lower bounds* on expected outcomes and (ii) *universal positive almost-sure termination* in Section 5.7.

5.2 Overview: Loop Invariants via CEGIS

While *finding* an admissible invariant I is challenging, *checking* whether a given candidate $I \in \mathsf{LExp}$ is indeed admissible is easier:

Is $I \in \mathsf{AdmInv}$ *?*

can be decided[3] via SMT solving using the techniques from Section 4.6.3. This motivates the development of techniques that generate decent candidates I fast and then check their admissibility. One of our key empirical results is that this enables the automatic synthesis of admissible invariants for probabilistic loops with huge or infinite state spaces (cf. Section 5.8).

[3]See Section 5.5.1 for details.

We present the *Counterexample-Guided Inductive Synthesis (CEGIS)* framework depicted in Figure 5.1 for finding admissible invariants, which builds upon the idea of template-based invariant synthesis for probabilistic programs [KMMM10]: A *template generator* automatically generates templates for admissible invariants. A template $\mathcal{T}$ syntactically describes a set $\langle \mathcal{T} \rangle$ of *candidate invariants* in $\mathsf{FinLExp}$. The inner loop (shaded box in Figure 5.1) then searches for an admissible invariant $I \in \langle \mathcal{T} \rangle$. If it succeeds, an admissible invariant has been synthesized. Otherwise, the template cannot be instantiated to an admissible invariant, i.e., $\langle \mathcal{T} \rangle \cap \mathsf{AdmInv} = \emptyset$. The inner loop then reports this back to the template generator (possibly with some hint) and asks for a refined template.

We illustrate this procedure by means of an example. Consider the following program C modeling a variant of the bounded retransmission protocol [HSV93; DJJL01], which already served us several times as a running example.

```
while( $sent < 8\,000\,000 \wedge failed < 10$ ){
    {
        $failed := failed + 1$
    } [ 0.001 ] {
        $failed := 0 ;\ sent := sent + 1$
    }
}
```

In this model, there are 8 000 000 packets to send and the protocol fails as soon as, for some packet, the threshold of 10 retransmissions is reached. Now let $\lambda \in [0,1] \cap \mathbb{Q}$ be some rational probability and suppose we wish to verify that λ upper-bounds the probability that the protocol fails. In terms of weakest preexpectations, we thus aim to verify that

$$
\begin{aligned}
& \mathsf{wp}[\![C]\!](\overbrace{[\mathit{failed} = 10]}^{=f}) \\
\sqsubseteq\quad & \underbrace{[\mathit{sent} = 0 \wedge \mathit{failed} = 0] \cdot \lambda + [\neg(\mathit{sent} = 0 \wedge \mathit{failed} = 0)] \cdot \infty}_{=g}\ .
\end{aligned}
\tag{5.4}
$$

We start by constructing the following template from the syntax of C and f:

$$\mathcal{T} \;=\; [\mathit{failed} < 10 \wedge \mathit{sent} < 8\,000\,000] \cdot (\mathfrak{a} \cdot \mathit{sent} + \mathfrak{b} \cdot \mathit{failed} + \mathfrak{c})$$

$$+ [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \wedge \mathit{sent} \geq 8\,000\,000] \cdot 0$$

$\mathcal{T}$ contains two kinds of variables: $\mathbb{N}$-valued program variables *failed*, *sent* and $\mathbb{Q}$-valued *template variables* $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$. While the template is nonlinear, substituting $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$ with values yields a piecewise linear candidate I.

Our inner CEGIS loop checks whether there exists an assignment from these template variables $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$ to concrete values in $\mathbb{Q}$ such that the resulting piecewise linear candidate is an admissible invariant. For that, the synthesizer first *guesses* values for $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$, say all 0's, which yields the *instance* $I \in \langle \mathcal{T} \rangle$ of $\mathcal{T}$ given by

$$\begin{aligned} I \;=\; & [\mathit{failed} < 10 \wedge \mathit{sent} < 8\,000\,000] \cdot (0 \cdot \mathit{sent} + 0 \cdot \mathit{failed} + 0) \\ & + [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \wedge \mathit{sent} \geq 8\,000\,000] \cdot 0 \,, \end{aligned}$$

and asks a *verifier* whether this I is indeed admissible, i.e., non-negative[4], inductive, and safe. The verifier determines that I is not admissible since I is *not inductive*, and explains this fact by providing a *counterexample to inductivity* — a state $\sigma \in \mathsf{States}$ with $\sigma(\mathit{failed}) = 9$, $\sigma(\mathit{sent}) = 7999999$ for which we have

$$ {}^{\mathsf{wp}}_{C}\Phi_{f}(I)(\sigma) \;>\; [\![I]\!](\sigma)\,. $$

This counterexample is reported to the synthesizer, which ensures that the instances of $\mathcal{T}$ it will produce in the future will no longer violate inductivity for this state σ. For that, it learns the following lemma for the template variables:

$$\overbrace{0.001}^{\text{quantity obtained from applying } {}^{\mathsf{wp}}_{C}\Phi_{f} \text{ to instances of } \mathcal{T} \text{ and evaluated at } \sigma}$$

$$\overset{!}{\leq} \qquad \underbrace{\mathfrak{a} \cdot 7999999 + \mathfrak{b} \cdot 9 + \mathfrak{c}}_{\text{quantity instances of } \mathcal{T} \text{ evaluate to at } \sigma \text{ depending on } \mathfrak{a}, \mathfrak{b}, \mathfrak{c}}$$

Observe that this lemma is linear in $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$. The synthesizer will now keep "guessing" assignments to the template variables which are consistent with the learned lemmas. If, eventually, no such parameter assignment exists anymore, we have determined that $\mathcal{T}$ cannot be instantiated to an admissible invariant, which will then be reported to the template generator. On the other hand, the synthesizer might eventually produce an admissible invariant. In our running example, for

[4]Some instances of $\mathcal{T}$ do not denote expectations in $\mathbb{E}$ as they might evaluate to negative numbers.

$\lambda = 0.9$, after 6 lemmas, our synthesizer produces the admissible invariant

$$\begin{aligned} I \;=\;& \left[\mathit{failed} < 10 \land \mathit{sent} < 8 \cdot 10^6\right] \cdot \left(-\tfrac{9}{8\cdot 10^7} \cdot \mathit{sent} + \tfrac{79\,991}{72\cdot 10^7} \cdot \mathit{failed} + \tfrac{9}{10}\right) \\ &+ [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \land \mathit{sent} \geq 8\,000\,000] \cdot 0 \,. \end{aligned}$$

For a tighter λ, the simple template $\mathcal{T}$ might not suffice. For example, it is impossible to instantiate $\mathcal{T}$ to an admissible invariant for $\lambda = 0.8$, even though (5.4) holds. We therefore support *more general templates* of the form

$$\begin{aligned} &\sum_i [\mathcal{B}_i] \cdot (\mathfrak{a}_i \cdot \mathit{sent} + \mathfrak{b}_i \cdot \mathit{failed} + \mathfrak{c}_i) \\ &+ [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \land \mathit{sent} \geq 8\,000\,000] \cdot 0 \,, \end{aligned}$$

where the $\mathcal{B}_i$ are (possibly templated) Boolean expressions partitioning States. In particular, we allow for a template such as

$$\begin{aligned} \mathcal{T}_2 \;=\;& [\mathit{failed} < 10 \land \mathit{sent} < \mathfrak{e}] \cdot (\mathfrak{a}_1 \cdot \mathit{sent} + \mathfrak{b}_1 \cdot \mathit{failed} + \mathfrak{c}_1) \\ &+ [\mathit{failed} < 10 \land \mathit{sent} \geq \mathfrak{e}] \cdot (\mathfrak{a}_2 \cdot \mathit{sent} + \mathfrak{b}_2 \cdot \mathit{failed} + \mathfrak{c}_2) \\ &+ [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \land \mathit{sent} \geq 8\,000\,000] \cdot 0 \,. \end{aligned}$$

However, such templates containing *templated* Boolean expressions are challenging for the CEGIS loop. Thus, we additionally consider templates where the $\mathcal{B}_i$'s range only over program variables, e.g.,

$$\begin{aligned} \mathcal{T}_3 \;=\;& [\mathit{failed} < 10 \land \mathit{sent} < 4\,000\,000] \cdot (\mathfrak{a}_1 \cdot \mathit{sent} + \mathfrak{b}_1 \cdot \mathit{failed} + \mathfrak{c}_1) \\ &+ [\mathit{failed} < 10 \land \mathit{sent} \geq 4\,000\,000] \cdot (\mathfrak{a}_2 \cdot \mathit{sent} + \mathfrak{b}_2 \cdot \mathit{failed} + \mathfrak{c}_2) \\ &+ [\mathit{failed} = 10] \cdot 1 + [\mathit{failed} < 10 \land \mathit{sent} \geq 8\,000\,000] \cdot 0 \,. \end{aligned}$$

Our template refinement algorithms produce these templates automatically.

Remark 5.2 (On Probabilistic Model Checking).
The specification from (5.4) on page 169 describes a *finite-state* verification problem: That is, the fragment of pGCL's operational MDP $\mathcal{O}$ reachable from the initial configuration (C, σ) with $\sigma(\mathit{sent}) = 0$ and $\sigma(\mathit{failed}) = 0$ is *finite*.

Moreover, by Corollary 2.16.3 on page 72, (5.4) is equivalent to

$$\underbrace{\mathsf{Pr}\Big(\mathcal{O}, (C, \sigma) \models \Diamond\Big\{(\Downarrow, \tau) \mid \tau \models \mathit{failed} = 10\Big\}\Big)}_{\text{reachability probability of } \mathcal{O}\text{, see Section 2.2.3}} \leq \lambda \,.$$

Verifying such bounds on reachability probabilities of finite-state MDPs is among the key applications of probabilistic model checkers such as Storm [HJKQ+22; DJKV17]. Interestingly, even though C appears to be rather simple, the explicit-state engine of Storm cannot verify the above bound for $\lambda = 0.9$ within 2 hours. This is because Storm enumerates *all* configurations in $\mathcal{O}$ reachable from (C, σ), which is infeasible due to the large model parameters — the number of packets to send and the maximal number of retransmissions per packet. On the other hand, our approach verifies the given bound by considering 6 program states, thus *avoiding* the enumeration of all states. It is evident that, for the linear fragment of pGCL, our approach can be considered an alternative means for tackling such probabilistic model checking tasks. We provide an empirical comparison of our approach and Storm in Section 5.8. The results are promising: Our tool cegispro2 is capable of verifying bounds on reachability probabilities of models taken from the literature that cannot be verified by Storm within the time limit of 2 hours. However, the contrary is also the case: We show that there are simple programs and specifications which Storm verifies within a few seconds whereas cegispro2 times out. We refer to Section 5.9 for a discussion on how these examples could guide future investigations for improving our approach.

Chapter Outline. The remainder of this chapter is structured as follows: Section 5.3 introduces templates. Section 5.4 provides the prerequisites for our CEGIS loop, which is treated in Section 5.5. Section 5.6 presents the template generator. Extensions of our approach are presented in Section 5.7. Section 5.8 provides details on our implementation and the corresponding empirical results. Finally, in Section 5.9, we discuss future and related work.

5.3 Templated Piecewise Linear Expectations

We formalize the templates described in the previous section by introducing *templated piecewise linear expectations* (or *templates*, for short). As the name suggests, (appropriate) instances of these templates will be piecewise linear expectations as introduced in Definition 4.4 on page 147.

Let $\mathsf{TVars} = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}, \ldots\}$ be a countably infinite *set of $\mathbb{Q}$-valued template variables*. A *template (variable) valuation* is a function $\mathfrak{I} \colon \mathsf{TVars} \to \mathbb{Q}$, assigning to each template variable a rational number. We start by introducing the basic building blocks of templates — templated arithmetic and Boolean expressions.

Definition 5.2 (Templated Arithmetic Expressions).
Templated (linear) arithmetic expressions, which we denote by $\mathcal{E}$ and variations thereof, in the set TAExpr adhere to the grammar

$$\begin{aligned}
\mathcal{E} \quad \longrightarrow \quad & q \in \mathbb{Q} && \text{(rationals)}\\
\mid\ & \mathfrak{a} \in \mathsf{TVars} && \text{(template variables)}\\
\mid\ & x \in \mathsf{Vars} && \text{($\mathbb{N}$-valued program variables)}\\
\mid\ & \mathfrak{a} \cdot x && \text{(multiplication of template and program variables)}\\
\mid\ & q \cdot \mathcal{E} && \text{(scaling by constants in $\mathbb{Q}$)}\\
\mid\ & \mathcal{E} + \mathcal{E}\,. && \text{(addition)}
\end{aligned}$$

Given a template valuation $\mathfrak{I}$, we denote by $\mathcal{E}[\mathfrak{I}] \in \mathsf{AExpr}_{\mathbb{Q}}$ the *instance of $\mathcal{E}$ under $\mathfrak{I}$*, which is the linear arithmetic expression over $\mathbb{Q}$ (cf. Definition 4.4 on page 147) obtained from substituting every template variable $\mathfrak{a}$ in $\mathcal{E}$ by $\mathfrak{I}(\mathfrak{a})$. Recall that the semantics $[\![\mathcal{E}[\mathfrak{I}]]\!]$ of $\mathcal{E}[\mathfrak{I}]$ is of type $\mathsf{States} \to \mathbb{Q}$.

Definition 5.3 (Templated Boolean Expressions).
Templated Boolean expressions, which we denote by $\mathcal{B}$ and variations thereof, in the set TBExpr adhere to the grammar

$$\begin{aligned}
\mathcal{B} \quad \longrightarrow \quad & \mathcal{E} < \mathcal{E} && \text{(strict inequality of arithmetic expressions)}\\
\mid\ & \mathcal{B} \wedge \mathcal{B} && \text{(conjunction)}\\
\mid\ & \neg \mathcal{B}\,. && \text{(negation)}
\end{aligned}$$

We adapt the usual order of precedence and syntactic sugar for arithmetic and Boolean operations from Section 3.2. Given a template valuation $\mathfrak{I}$, we denote by $\mathcal{B}[\mathfrak{I}] \in \mathsf{AExpr}_{\mathbb{Q}}$ the *instance of $\mathcal{B}$ under $\mathfrak{I}$*, which is the linear Boolean expression over $\mathbb{Q}$ (cf. Definition 4.4 on page 147) obtained from substituting every template variable $\mathfrak{a}$ in $\mathcal{B}$ by $\mathfrak{I}(\mathfrak{a})$. Recall that the semantics $[\![\mathcal{B}[\mathfrak{I}]]\!]$ is a predicate in $\mathcal{P}(\mathsf{States})$. As usual, we write $\sigma \models \mathcal{B}[\mathfrak{I}]$ instead of $\sigma \in [\![\mathcal{B}[\mathfrak{I}]]\!]$.

Definition 5.4 (Templates (adapted from [KMMM10])).
Templated piecewise linear expectations (*templates*, for short), which we denote

by $\mathcal{T}$ and variations thereof, in the set TExp adhere to the grammar

$$\mathcal{T} \quad \longrightarrow \quad [\mathcal{B}_1] \cdot \mathcal{E}_1 + \ldots + [\mathcal{B}_n] \cdot \mathcal{E}_n \,,$$

where $n \geq 1$ and where the $\mathcal{B}_i$ partition States, i.e., for all template valuations $\mathfrak{I}$ and all states σ, there is *exactly one* i with $\sigma \models \mathcal{B}_i[\mathfrak{I}]$.

We call a template $\mathcal{T}$ *fixed-partition*, if none of the $\mathcal{B}_i$ contains a template variable. Otherwise, we call $\mathcal{T}$ *variable-partition*. Given a template valuation $\mathfrak{I}$, we denote by $\mathcal{T}[\mathfrak{I}]$ the *instance of $\mathcal{T}$ under $\mathfrak{I}$*, which is obtained from substituting every template variable $\mathfrak{a}$ in $\mathcal{T}$ by $\mathfrak{I}(\mathfrak{a})$. We denote the *set of instances of $\mathcal{T}$* by

$$\langle \mathcal{T} \rangle \;=\; \{\mathcal{T}[\mathfrak{I}] \mid \mathfrak{I} \colon \mathsf{TVars} \to \mathbb{Q}\} \,.$$

The semantics $[\![\mathcal{T}[\mathfrak{I}]]\!] : \mathsf{States} \to \mathbb{Q}$ is defined by induction on the structure of $\mathcal{T}[\mathfrak{I}]$, which is analogous to Section 3.2.4. Notice that we might have $\mathcal{T}[\mathfrak{I}] \notin$ LExp, i.e., that $[\![\mathcal{T}[\mathfrak{I}]]\!]$ is not a well-defined expectation in $\mathbb{E}$ since we might have $[\![\mathcal{T}[\mathfrak{I}]]\!](\sigma) < 0$ for some state σ. It is therefore convenient to introduce:

Definition 5.5 (Piecewise Linear Quantities).
We denote by LQuant the set of all expressions f adhering to the grammar from Definition 4.4 on page 147 which do not contain ∞ and where we *omit* the requirement that for all $\sigma \in$ States, we have $[\![f]\!](\sigma) \geq 0$. We call elements of LQuant *piecewise linear quantities*.

The semantics $[\![f]\!]$ of $f \in$ LQuant is of type States $\to \mathbb{Q}$. We lift the partial order $\sqsubseteq$ to functions of type States $\to \mathbb{Q}$ in the obvious way. Notice that

$$\mathsf{FinLExp} \;\subset\; \mathsf{LQuant} \;\subset\; \mathsf{TExp}$$

and that, for all templates $\mathcal{T} \in$ TExp, we have $\langle \mathcal{T} \rangle \subseteq$ LQuant.

We will often consider what we call *natural templates*, which take the given loop C, the postexpectation f, and the candidate upper bound g into account.

Definition 5.6 (Natural Templates).
Let $C = \mathtt{while}(\varphi)\{C'\} \in$ LpGCL and $f \in$ FinLExp, $g \in$ LExp. We say that a template $\mathcal{T} \in$ TExp is *natural for C, f, and g*, if the following conditions hold:

1. $\mathcal{T}$ is of the form
$$\underbrace{[\neg\varphi \wedge \varphi_1] \cdot e_1 + \ldots + [\neg\varphi \wedge \varphi_n] \cdot e_n}_{\text{equivalent to } [\neg\varphi]\cdot f} + [\mathcal{B}_1] \cdot \mathcal{E}_1 + \ldots + [\mathcal{B}_m] \cdot \mathcal{E}_m \,.$$

2. We have $[\neg\varphi] \cdot f \sqsubseteq [\neg\varphi] \cdot g$.

We denote the *set of all natural templates for C, g, and f* by TnExp.

5.4 Reasoning about Template Instances

In this section, we characterize when a given template $\mathcal{T}$ can be instantiated to an admissible invariant. This provides us with the prerequisites for obtaining appropriate synthesizers as described in Section 5.2. Throughout this section, fix a loop $C = \mathtt{while}(\varphi)\{C'\}$ and some $f \in \mathsf{FinLExp}, g \in \mathsf{LExp}$.

As a first step, we lift the characteristic function ${}^{\mathsf{wp}}_{C}\Phi_{f}$ to templates.

Theorem 5.1 (Lifting Characteristic Functions of Loops to Templates). There is a computable function

$$ {}^{\mathsf{wp}}_{C}\Theta_{f} \colon \mathsf{TExp} \to \mathsf{TExp} $$

such that for all $\mathcal{T} \in \mathsf{TExp}$ and template valuations $\mathfrak{I}$, we have

$$ \mathcal{T}[\mathfrak{I}] \in \mathsf{LExp} \qquad \text{implies} \qquad \left[\!\!\left[{}^{\mathsf{wp}}_{C}\Theta_{f}(\mathcal{T})[\mathfrak{I}] \right]\!\!\right] = {}^{\mathsf{wp}}_{C}\Phi_{f}\left(\llbracket \mathcal{T}[\mathfrak{I}] \rrbracket\right) . $$

Moreover, if $\mathcal{T}$ is fixed-partition (resp. natural), so is ${}^{\mathsf{wp}}_{C}\Theta_{f}(\mathcal{T})$.

Proof. We compute ${}^{\mathsf{wp}}_{C}\Theta_{f}(\mathcal{T})$ as follows. First, we apply the syntactic analogous of the rules from Table 2.1 on page 60 to compute

$$ [\varphi] \cdot \mathsf{wp}\llbracket C' \rrbracket(\mathcal{T}) + [\neg\varphi] \cdot f , $$

treating the template variables as constants and ignoring that we deal with possibly negative quantities, which is possible since the loop body C' is loop-free so that we do not rely on the well-definedness of least fixpoints. Second, we apply the construction from Lemma 4.16 on page 151 to ensure that the so-obtained (possibly templated) Boolean expressions partition States. If $\mathcal{T}$ is fixed-partition, then none of the Boolean expressions obtained from applying the rules from Table 2.1 contain template variables, in which case step (2) can be done using the optimized Algorithm 3 on page 150. ∎

5

Example 5.1.
Consider the geometric loop

$$C \;=\; \texttt{while}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x + 1\}\}$$

with

$$f \;=\; [\mathsf{true}] \cdot x \qquad \text{and} \qquad g \;=\; [\mathsf{true}] \cdot (x + 1)\,.$$

The fixed-partition template f given by

$$\mathcal{T} \;=\; [y = 1] \cdot (\mathfrak{a} \cdot x + \mathfrak{b} \cdot y + \mathfrak{c}) + [y \neq 1] \cdot x$$

is natural for C, f, g and we have

$$^{\mathsf{wp}}_{C}\Theta_{f}(\mathcal{T}) \;=\; [y = 1] \cdot (1/2 \cdot \mathfrak{a} \cdot x + 1/2 \cdot \mathfrak{a} + 1/2 \cdot \mathfrak{b} \cdot y + 1/2 \cdot \mathfrak{c} + 1/2 \cdot x) + [y \neq 1] \cdot x\,.$$

Next, we treat the construction of what we called *lemmas* in Section 5.2 — constraints on the template variables asserting, e.g., inductivity of template instances at a particular program state. For that, let $\sigma \in \mathsf{States}$ be a program state. Given $\mathcal{E} \in \mathsf{TAExpr}$, we denote by $\mathcal{E}(\sigma)$ the linear[5] expression over the $\mathbb{Q}$-valued variables TVars with coefficients in $\mathbb{Q}$ obtained from substituting every program variable x in $\mathcal{E}$ by $\sigma(x)$. Analogously, given $\mathcal{B} \in \mathsf{TBExpr}$, we denote by $\mathcal{B}(\sigma)$ the Boolean combination of linear inequalities over TVars with coefficients in $\mathbb{Q}$ obtained from substituting every program variable x in $\mathcal{B}$ by $\sigma(x)$. Notice that $\mathcal{B}(\sigma)$ is a formula in the quantifier-free fragment of linear rational arithmetic (QF_LRA). We write $\mathfrak{I} \models \mathcal{B}(\sigma)$ to indicate that $\mathcal{B}(\sigma)$ evaluates to true under $\mathfrak{I}$. Satisfiability of QF_LRA formulae, i.e.,

Does there exist $\mathfrak{I}$ *such that* $\mathfrak{I} \models \mathcal{B}(\sigma)$ *?*

is decidable using, e.g., Fourier-Motzkin variable elimination [Fou25; Mot36]. From now on, we assume a black box — an SMT solver[6] — for deciding the satisfiability of these formulae. Now, the lemmas from Section 5.2 are obtained from what we call *state-specific formulae*:

Definition 5.7 (State-Specific Linear Arithmetic Formulae).
Let $\mathcal{T}, \mathcal{T}' \in \mathsf{TExp}$ be given by

$$\mathcal{T} \;=\; [\mathcal{B}_1] \cdot \mathcal{E}_1 + \ldots + [\mathcal{B}_n] \cdot \mathcal{E}_n \quad \text{and} \quad \mathcal{T}' \;=\; [\mathcal{B}'_1] \cdot \mathcal{E}'_1 + \ldots + [\mathcal{B}'_m] \cdot \mathcal{E}'_m\,.$$

[5]More precisely, this expression is generally affine but we adopt SMT terminology here.
[6]Our implementation uses the SMT solver Z3 [MB08]. See Section 5.8 for details.

Moreover, let $h \in \mathsf{LExp}$ in GNF be given by

$$h \;=\; [\vartheta_1]\cdot\tilde{e}_1 + \ldots + [\vartheta_k]\cdot\tilde{e}_k \,.$$

Finally, let $\sigma \in \mathsf{States}$. We define the following QF_LRA formulae:

1. The formula $0 \trianglelefteq_\sigma \mathcal{T}$ is defined as

$$\bigwedge_{i=1}^{n} \quad \mathcal{B}_i(\sigma) \implies 0 \leq \mathcal{E}_i(\sigma)\,.$$

2. The formula $\mathcal{T} \trianglelefteq_\sigma \mathcal{T}'$ is defined as

$$\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{m} \quad \big(\mathcal{B}_i(\sigma) \wedge \mathcal{B}_j(\sigma)\big) \implies \mathcal{E}_i(\sigma) \leq \mathcal{E}_j(\sigma)\,.$$

3. The formula $\mathcal{T} \trianglelefteq_\sigma h$ is defined as

$$\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{k} \quad \big(\mathcal{B}_i(\sigma) \wedge \underbrace{[\![\vartheta_j]\!](\sigma)}_{\in\{\mathsf{true},\mathsf{false}\}}\big) \implies \begin{cases}\mathsf{true} & \text{if } [\![\tilde{e}_j]\!](\sigma) = \infty \\ \mathcal{E}_i(\sigma) \leq [\![\tilde{e}_j]\!] & \text{otherwise}\,.\end{cases}$$

If both $\mathcal{T}$ and $\mathcal{T}'$ are fixed-partition, then each of the above formulae simplifies to a *single* linear inequality (or true) since, in each formula, all but one left-hand side of the implications are equivalent to false.

5

Example 5.2.
Reconsider the program

$$C \;=\; \mathtt{while}(y=1)\{\{y := 0\}\,[1/2]\,\{x := x+1\}\}\,,$$

and

$$f \;=\; [\mathsf{true}]\cdot x \qquad \text{and} \qquad g \;=\; [\mathsf{true}]\cdot(x+1)\,,$$

and the following template for which we have

$$\begin{aligned}\mathcal{T} \;&=\; [y=1]\cdot(\mathfrak{a}\cdot x + \mathfrak{b}\cdot y + \mathfrak{c}) + [y\neq 1]\cdot x \\ {}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \;&=\; [y=1]\cdot(1/2\cdot\mathfrak{a}\cdot x + 1/2\cdot\mathfrak{a} + 1/2\cdot\mathfrak{b}\cdot y + 1/2\cdot\mathfrak{c} + 1/2\cdot x) \\ &\quad\; + [y\neq 1]\cdot x\,.\end{aligned}$$

Now let σ be a state with $\sigma(x) = 1$ and $\sigma(y) = 1$.

1. The formula $0 \trianglelefteq_\sigma \mathcal{T}$ asserts *non-negativity* at σ and is given by

$$\begin{aligned} & 1 = 1 \implies 0 \leq \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \\ \wedge \quad & 1 \neq 1 \implies 0 \leq 1 \,, \end{aligned}$$

   which can be simplified to

$$0 \leq \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \,.$$

2. The formula ${}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T}$ asserts *inductivity* at σ and is given by

$$\begin{aligned} & 1 = 1 \wedge 1 = 1 \implies 1/2 \cdot \mathfrak{a} \cdot 1 + 1/2 \cdot \mathfrak{a} + 1/2 \cdot \mathfrak{b} \cdot 1 + 1/2 \cdot \mathfrak{c} + 1/2 \cdot 1 \\ & \qquad\qquad\qquad\quad \leq \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \\ \wedge \quad & 1 = 1 \wedge 1 \neq 1 \implies 1/2 \cdot \mathfrak{a} \cdot 1 + 1/2 \cdot \mathfrak{a} + 1/2 \cdot \mathfrak{b} \cdot 1 + 1/2 \cdot \mathfrak{c} + 1/2 \cdot 1 \\ & \qquad\qquad\qquad\quad \leq 1 \\ \wedge \quad & 1 \neq 1 \wedge 1 = 1 \implies 1 \leq \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \\ \wedge \quad & 1 \neq 1 \wedge 1 \neq 1 \implies 1 \leq 1 \,, \end{aligned}$$

   which can be simplified to

$$1/2 \cdot \mathfrak{a} \cdot 1 + 1/2 \cdot \mathfrak{a} + 1/2 \cdot \mathfrak{b} \cdot 1 + 1/2 \cdot \mathfrak{c} + 1/2 \cdot 1 \leq \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \,.$$

3. The formula $\mathcal{T} \trianglelefteq_\sigma g$ asserts *safety* at σ and is given by

$$\begin{aligned} & 1 = 1 \wedge \mathsf{true} \implies \mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \leq 1 + 1 \\ \wedge \quad & 1 \neq 1 \wedge \mathsf{true} \implies 1 \leq 1 + 1 \,, \end{aligned}$$

   which can be simplified to

$$\mathfrak{a} \cdot 1 + \mathfrak{b} \cdot 1 + \mathfrak{c} \leq 1 + 1 \,.$$

Notice that, if the Boolean expressions $\mathcal{T}$ contained template variables, those simplifications might not be applicable, yielding the state-specific formulae to potentially have a complex Boolean structure.

The fact that state-specific formulae indeed assert non-negativity, inductivity, or safety of template instances at a given program state formally reads as follows:

Lemma 5.2 (Properties of State-Specific Formulae).
Let $\mathcal{T}, \mathcal{T}' \in \mathsf{TExp}$, $h \in \mathsf{LExp}$, $\sigma \in \mathsf{States}$, and $\mathfrak{I}\colon \mathsf{TVars} \to \mathbb{Q}$. Then:

1. $\mathfrak{I} \models 0 \trianglelefteq_\sigma \mathcal{T}$ if and only if $0 \leq [\![\mathcal{T}[\mathfrak{I}]]\!](\sigma)$.
2. $\mathfrak{I} \models \mathcal{T} \trianglelefteq_\sigma \mathcal{T}'$ if and only if $[\![\mathcal{T}[\mathfrak{I}]]\!](\sigma) \leq [\![\mathcal{T}'[\mathfrak{I}]]\!](\sigma)$.
3. $\mathfrak{I} \models \mathcal{T} \trianglelefteq_\sigma h$ if and only if $[\![\mathcal{T}[\mathfrak{I}]]\!](\sigma) \leq [\![h]\!](\sigma)$.

Proof. The reasoning is analogous to Section 4.6.3.3, exploiting that the (templated) Boolean expressions in $\mathcal{T}$, $\mathcal{T}'$, and h partition States. ■

With state-specific formulae, we characterize the existence of an admissible invariant in $\langle \mathcal{T} \rangle$ for some given template $\mathcal{T} \in \mathsf{TExp}$ as follows:

Theorem 5.3 (Existence of Admissible Invariants).
For every template $\mathcal{T} \in \mathsf{TExp}$, we have

$$\begin{aligned} & \text{exists } I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv} \\ \text{iff}\quad & \text{exists template valuation } \mathfrak{I}\colon \text{for all } \sigma \in \mathsf{States}\colon \\ & \mathfrak{I} \models \underbrace{0 \trianglelefteq_\sigma \mathcal{T}}_{\text{non-negativity}} \;\wedge\; \underbrace{{}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T}}_{\text{inductivity}} \;\wedge\; \underbrace{\mathcal{T} \trianglelefteq_\sigma g}_{\text{safety}} . \end{aligned}$$

Proof. This is an immediate consequence of Lemma 5.2. ■

The question of whether the existence of an admissible invariant in $\langle \mathcal{T} \rangle$ is decidable is open. For the special case of *finite-state loops* and *natural templates*, the existence of an admissible invariant in $\langle \mathcal{T} \rangle$ *is* decidable. We say that a loop $C = \texttt{while}(\varphi)\{C'\}$ is *finite-state*, if $|[\![\varphi]\!]| < \infty$, i.e., if the number of states that satisfy the loop guard φ is finite[7].

Theorem 5.4 (Decidability for Finite-State Loops).
For finite-state C and natural templates $\mathcal{T} \in \mathsf{TnExp}$, the problem

Is there some $I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv}$ *?*

is decidable. If $\mathcal{T}$ is fixed-partition, it is decidable via linear programming.

[7] Recall that, in this chapter, we assume the set Vars of program variables to be finite.

Proof. We have

$$
\begin{aligned}
& \text{exists } I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv} \\
\text{iff} \quad & \text{exists template valuation } \mathfrak{I}\colon \text{for all } \sigma \models \mathsf{States}\colon && \text{(Theorem 5.3)} \\
& \quad \mathfrak{I} \models 0 \trianglelefteq_\sigma \mathcal{T} \wedge {}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T} \wedge \mathcal{T} \trianglelefteq_\sigma g \\
\text{iff} \quad & \text{exists template valuation } \mathfrak{I}\colon \text{for all } \sigma \models \varphi\colon && (\mathcal{T} \text{ is natural}) \\
& \quad \mathfrak{I} \models 0 \trianglelefteq_\sigma \mathcal{T} \wedge {}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T} \wedge \mathcal{T} \trianglelefteq_\sigma g \\
\text{iff} \quad & \text{exists template valuation } \mathfrak{I}\colon && (\llbracket \varphi \rrbracket \text{ is finite}) \\
& \quad \mathfrak{I} \models \underbrace{\bigwedge_{\sigma \models \varphi} 0 \trianglelefteq_\sigma \mathcal{T} \wedge {}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T} \wedge \mathcal{T} \trianglelefteq_\sigma g}_{\mathsf{QF\_LRA\ formula}} .
\end{aligned}
$$

Hence, we have $\langle \mathcal{T} \rangle \cap \mathsf{AdmInv} \neq \emptyset$ iff the above QF_LRA formula is satisfiable. Since the latter is decidable, the claim follows. Moreover, if $\mathcal{T}$ is fixed-partition, then the above QF_LRA formula is a conjunction of linear inequalities, which implies the claim concerning linear programming. ■

5.5 Constructing an Efficient CEGIS Loop

With the prerequisites from the preceding section at hand, we construct the verifier and the synthesizer for our inner CEGIS loop as described in Section 5.2. For that, fix some loop $C = \mathtt{while}(\varphi)\{C'\}$ and $f \in \mathsf{FinLExp}, g \in \mathsf{LExp}$ throughout this section. Recall that the goal of the inner CEGIS loop is to determine whether there is an admissible invariant $I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv}$ for a *given* template $\mathcal{T}$.

5.5.1 The Verifier

The verifier determines whether a given candidate invariant $I \in \mathsf{LQuant}$ is admissible, i.e., whether $I \in \mathsf{AdmInv}$. If not, the verifier returns a counterexample to non-negativity, inductivity, or safety of I. To formalize this, we introduce:

Definition 5.8 (σ-Admissible Invariants).
For a state $\sigma \in \mathsf{States}$, the *set of σ-admissible invariants* is

$$
\begin{aligned}
\mathsf{AdmInv}(\sigma) \;=\; \Big\{ I \in \mathsf{LQuant} \quad \Big| \quad & \underbrace{0 \leq [\![I]\!](\sigma)}_{\sigma\text{-non-negative}} \\
\text{and} \quad & \underbrace{\left[\!\!\left[{}^{\mathsf{wp}}_{C}\Theta_f(I) \right]\!\!\right](\sigma) \leq [\![I]\!](\sigma)}_{\sigma\text{-inductive}} \\
\text{and} \quad & \underbrace{[\![I]\!](\sigma) \leq [\![g]\!](\sigma)}_{\sigma\text{-safe}} \Big\} .
\end{aligned}
$$

For a subset $\mathsf{States}' \subseteq \mathsf{States}$ of states, we define

$$
\mathsf{AdmInv}(\mathsf{States}') = \bigcap_{\sigma \in \mathsf{States}'} \mathsf{AdmInv}(\sigma) .
$$

Now let $I \in \mathsf{LQuant}$. Clearly, if $I \notin \mathsf{AdmInv}$, then $I \notin \mathsf{AdmInv}(\sigma)$ for some $\sigma \in \mathsf{States}$, i.e., state σ is a *counterexample* to non-negativity, inductivity, or safety of I. We denote by $\mathsf{CounterEx}(I) \subseteq \mathsf{States}$ the *set of all counterexamples of I*.

Theorem 5.5 (Existence of Effective Verifiers).
There is a computable function

$$
\mathsf{Verify}\colon \mathsf{LQuant} \to \{\mathsf{true}\} \cup \mathsf{States} ,
$$

called a *verifier*, such that for all $I \in \mathsf{LQuant}$, we have

1. $\mathsf{Verify}(I) = \mathsf{true}$ if and only if $I \in \mathsf{AdmInv}$, and
2. $\mathsf{Verify}(I) = \sigma$ implies $\sigma \in \mathsf{CounterEx}(I)$.

Proof. We employ the reduction from quantitative entailments to satisfiability of Boolean expressions in $\mathsf{LBExpr}_{\mathbb{Q}}$ from Section 4.6.3.3, which generalizes from LExp to LQuant since it does not rely on non-negativity. We have

$$
\begin{aligned}
& I \in \mathsf{AdmInv} \\
\text{iff} \quad & \underbrace{\mathsf{CEX}_{\sqsubseteq}([\mathsf{true}] \cdot 0, I) \vee \mathsf{CEX}_{\sqsubseteq}\left({}^{\mathsf{wp}}_{C}\Theta_f(I), I\right) \vee \mathsf{CEX}_{\sqsubseteq}(I, g)}_{\in \mathsf{LBExpr}_{\mathbb{Q}} \text{ (cf. Definition 4.4 on page 147)}} \text{ is unsatisfiable} .
\end{aligned}
$$

The latter is decidable by Lemma 4.15. Thus, in case of unsatisfiability, we let

$$\mathsf{Verify}(I) \;=\; \mathsf{true}\,.$$

In case of satisfiability, SMT solvers such as Z3 provide a concrete program state σ satisfying at least one of the above disjuncts and we let

$$\mathsf{Verify}(I) \;=\; \sigma\,. \qquad \blacksquare$$

5.5.2 The Synthesizer

Let $\mathcal{T} \in \mathsf{TExp}$ be a template. The synthesizer is provided with a finite subset $\mathsf{States}' \subseteq \mathsf{States}$ of states — the counterexamples provided by the verifier — and determines whether there is some instance $I \in \langle \mathcal{T} \rangle$ of $\mathcal{T}$ which is admissible for all states in States', i.e., $I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv}(\mathsf{States}')$. If so, the synthesizer returns such an I. If not, the synthesizer returns false.

Now let $\mathsf{FinStates} \subseteq \mathcal{P}(\mathsf{States})$ be the *set of finite subsets of states*.

Theorem 5.6 (Existence of Effective Synthesizers).
Let $\mathcal{T} \in \mathsf{TExp}$ be a template. There is a computable function

$$\mathsf{Synt}_{\mathcal{T}} \colon \mathsf{FinStates} \to \langle \mathcal{T} \rangle \cup \{\mathsf{false}\}$$

such that for all $\mathsf{States}' \in \mathsf{FinStates}$ and all $I \in \mathsf{LQuant}$, we have

1. if $\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}') = I$, then $I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv}(\mathsf{States}')$, and
2. $\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}') = \mathsf{false}$ if and only if $\langle \mathcal{T} \rangle \cap \mathsf{AdmInv}(\mathsf{States}') = \emptyset$.

For fixed-partition $\mathcal{T}$, $\mathsf{Synt}_{\mathcal{T}}$ can be implemented via linear programming.

Proof. We reduce computing $\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}')$ to a satisfiability problem of QF_LRA formulae, which is decidable. Lemma 5.2 and Theorem 5.3 yield

$$\begin{aligned} & \text{exists } I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv}(\mathsf{States}') \\ \text{iff} \quad & \text{exists template valuation } \mathfrak{I}\colon \\ & \mathfrak{I} \models \underbrace{\bigwedge_{\sigma \in \mathsf{States}'} 0 \trianglelefteq_\sigma \mathcal{T} \wedge {}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_\sigma \mathcal{T} \wedge \mathcal{T} \trianglelefteq_\sigma g}_{\text{QF\_LRA formula}}\,. \end{aligned}$$

Algorithm 6: Synthesis of Admissible Template Instances

```
input: A template $\mathcal{T} \in \mathsf{TExp}$
output: $\begin{cases} \mathsf{false} & \text{only if } \langle \mathcal{T} \rangle \cap \mathsf{AdmInv} = \emptyset \\ I & \text{only if } I \in \langle \mathcal{T} \rangle \cap \mathsf{AdmInv} \end{cases}$
1 $\mathsf{States}' \leftarrow \emptyset$
2 while $\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}') \neq \mathsf{false}$ do
3     $I \leftarrow \mathsf{Synt}_{\mathcal{T}}(\mathsf{States}')$
4     $result \leftarrow \mathsf{Verify}(I)$
5     if $result = \mathsf{true}$ then
6         return $I$                /* Verifier returns true, we have $I \in \mathsf{AdmInv}$ */
7     $\mathsf{States}' \leftarrow \mathsf{States}' \cup \{result\}$            /* $result$ is a counterexample */
8 return false                          /* $\langle \mathcal{T} \rangle \cap \mathsf{AdmInv} = \emptyset$ */
```

Figure 5.2: Implementation of the inner CEGIS loop from Figure 5.1.

Hence, if the above QF_LRA formula is unsatisfiable, we let

$$\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}') \;=\; \mathsf{false}\,.$$

In case of satisfiability, SMT solvers such as Z3 provide a concrete template valuation $\mathfrak{I}$ satisfying all of the above conjuncts and we let

$$\mathsf{Synt}_{\mathcal{T}}(\mathsf{States}') \;=\; \mathcal{T}[\mathfrak{I}]\,.$$

If $\mathcal{T}$ is fixed-partition, then the above QF_LRA formula is a conjunction of linear inequalities, which implies the claim concerning linear programming.■

For finite-state loops, we obtain the following completeness result:

Theorem 5.7 (Synthesizer Completeness for Finite-State Loops).
For finite-state C and natural templates $\mathcal{T} \in \mathsf{TnExp}$, we have

$$\mathsf{Synt}_{\mathcal{T}}(\llbracket \varphi \rrbracket) \in \mathsf{AdmInv} \quad \text{or} \quad \langle \mathcal{T} \rangle \cap \mathsf{AdmInv} = \emptyset\,.$$

Using the synthesizer and the verifier in concert is then intuitive as in Algorithm 6 on page 183. We incrementally ask our synthesizer to provide an instance $I \in \langle \mathcal{T} \rangle$ of $\mathcal{T}$ which is σ-admissible for all states $\sigma \in \mathsf{States}'$. Unless the synthesizer returns false, we ask the verifier whether I is admissible for *all* states. If so, we return I; otherwise, we get a counterexample σ and add it to States' before synthesizing the next candidate. If the synthesizer returns false, we ask the template generator for a refined template. See Section 5.6 for details.

Cooperative Verifiers. The verifier from Theorem 5.5 may return any counterexample. Without further restrictions, a verifier might start enumerating these counterexamples, which can yield a bad performance.

Example 5.3.
Consider the finite-state loop C given by

$$\mathtt{while}(y = 0 \land z < 100)\{\{x := x+1\}\,[0.99]\,\{y := 1\}\}\,,$$

the postexpectation

$$f \;=\; [y=1]\cdot 1 + [y \neq 1]\cdot 0\,,$$

and the fixed-partition template

$$\mathcal{T} \;=\; [y = 0 \land z < 100]\cdot(\mathfrak{a}\cdot x + \mathfrak{b}\cdot y + \mathfrak{c}) + [y=1] + [y = 0 \land z \geq 100]\cdot 0\,.$$

Assume that the verifier returns two consecutive states σ_1, σ_2 with $\sigma_i(y) = 0$ and $\sigma_i(x) = i$ for $i \in \{1,2\}$. The formulae asserting inductivity at σ_i are

$${}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_{\sigma_i} \mathcal{T} \quad=\quad 0.99\cdot((i+1)\cdot\mathfrak{a} + \mathfrak{c}) + 0.01 \leq i\cdot\mathfrak{a} + \mathfrak{c}\,.$$

The two constraints are very similar in the sense that there is only a little information gained from the formula ${}^{\mathsf{wp}}_{C}\Theta_f(\mathcal{T}) \trianglelefteq_{\sigma_2} \mathcal{T}$ obtained from σ_2. Formulae obtained from more diverse states such as σ' with, e.g., $\sigma'(x) = 98$, prune more undesired template instances.

We therefore consider *cooperative* verifiers, which are parameterized by a distance measure $\mu\colon \mathsf{States}\times\mathsf{States} \to \mathbb{Q}$ on states[8]. *Additionally* to I, the verifier is provided with the last counterexample σ_{last} and a lower bound $q \in \mathbb{Q}$. The verifier then prefers to return new counterexamples σ with $\mu(\sigma, \sigma_{\mathsf{last}}) \geq q$.

[8]In our implementation, we use the Manhattan distance on the (relevant) program variables, i.e., the distance measure μ is defined as $\mu(\sigma,\sigma') = \sum_{x\in\mathsf{Vars}} |\sigma(x) - \sigma'(x)|$.

Definition 5.9 (Cooperative Verifiers).
A *cooperative verifier* is a computable function

$$\mathsf{CVerify}_\mu\colon \mathsf{States}\times\mathbb{Q}\to(\mathsf{LQuant}\to\{\mathsf{true}\}\cup\mathsf{States})$$

such that for all $I\in\mathsf{LQuant}$, $\sigma,\sigma_{\text{last}}\in\mathsf{States}$, and $q\in\mathbb{Q}$, we have

1. $\mathsf{CVerify}_\mu(\sigma,q)$ is a verifier, and
2. if $\mathsf{CVerify}_\mu(\sigma_{\text{last}},q)(I)=\sigma$, then
$$\mu(\sigma,\sigma_{\text{last}})\geq q \quad\text{or}\quad \text{for all } \sigma'\in\mathsf{CounterEx}(I)\colon\ \mu(\sigma',\sigma_{\text{last}})\leq q\ .$$

Furthermore, we can easily generalize the cooperative verifier beyond taking into account only the last counterexample. In our implementation, rather than fixing a value q, we adapt q during runtime: We start with $q=1$. If we succeed in finding two counterexamples that were q apart, we update $q\leftarrow q\cdot d$, otherwise, $q\leftarrow q\cdot 1/d$ for suitable values of d.

5.6 The Template Generator

Throughout this section, fix some loop $C=\texttt{while}(\varphi)\{C'\}$ and $f\in\mathsf{FinLExp}, g\in\mathsf{LExp}$ and let $\mathsf{Vars}=\{x_1,\ldots,x_m\}$. We describe (i) the construction of the initial template $\mathcal{T}_1$ given to the synthesizer and (ii) the three approaches we have implemented for obtaining a *refined* template $\mathcal{T}_{i+1}$ constructed when the synthesizer reports that for the current template $\mathcal{T}_i$ we have $\langle\mathcal{T}_i\rangle\cap\mathsf{AdmInv}=\emptyset$.

5.6.1 Constructing the Initial Template

We start with a natural[9] fixed-partition template $\mathcal{T}_1\in\mathsf{TnExp}$ constructed automatically from the syntactic structure of the given loop $C=\texttt{while}(\varphi)\{C'\}$ and the piecewise linear postexpectation $f\in\mathsf{LExp}$ in GNF. Intuitively, $\mathcal{T}$ partitions the state space according to the possible branches through the loop body C' and the postexpectation f. To put this more formally, let

$$f\ =\ [\vartheta_1]\cdot e_1+\ldots+[\vartheta_k]\cdot e_k\ .$$

Since the loop body C' is loop-free, we can compute linear Boolean expressions $\varphi_1,\ldots,\varphi_n\in\mathsf{LBExpr}_\mathbb{Z}$ partitioning the set $[\![\varphi]\!]$ of states satisfying the loop guard

[9]We assume that $[\neg\varphi]\cdot f\sqsubseteq[\neg\varphi]\cdot g$, which is decidable by Lemma 4.17 on page 154.

φ such that the lifted characteristic function ${}^{\mathsf{wp}}_{C}\Theta_f$ of C w.r.t. f can be written in such a way that for all $h \in \mathsf{LExp}$, we have

$$
\begin{aligned}
&{}^{\mathsf{wp}}_{C}\Theta_f(h) \\
&= \underbrace{\Bigl(\sum_{i=1}^{n} [\varphi \wedge \varphi_i] \cdot \Bigl(\sum_{j=1}^{n_i} p_{i,j} \cdot h\bigl[\mathsf{upd}_{i,j}\bigr]\Bigr)\Bigr)}_{[\varphi]\cdot \mathsf{wp}[\![C']\!](h)} + \underbrace{[\neg\varphi \wedge \vartheta_1] \cdot e_1 + \ldots + [\neg\varphi \wedge \vartheta_k] \cdot e_k}_{[\neg\varphi]\cdot f} ,
\end{aligned}
$$

where the $p_{i,j}$ are appropriate rational probabilities and the $\mathsf{upd}_{i,j}$ are appropriate substitutions of variables by arithmetic expressions corresponding to the assignment statements in the loop body C'. We construct $\mathcal{T}_1$ by associating to each partition $\varphi \wedge \varphi_i$ a templated arithmetic expression over $\{x_1, \ldots, x_m\}$:

$$
\mathcal{T}_1 \;=\; \Bigl(\sum_{i=1}^{n} [\varphi \wedge \varphi_i] \cdot \Bigl(\mathfrak{b}_i + \sum_{j=1}^{m} \mathfrak{a}_{i,j} \cdot x_j\Bigr)\Bigr) + \underbrace{[\neg\varphi \wedge \vartheta_1] \cdot e_1 + \ldots + [\neg\varphi \wedge \vartheta_k] \cdot e_k}_{[\neg\varphi]\cdot f} ,
$$

where the Boolean expressions in $\mathcal{T}_1$ partition States by construction.

Example 5.4.
Consider the following loop $C \in \mathsf{LpGCL}$:

```
while($x \leq 1$){
    {$y := 1$} [$1/2$] {$y := 2$};
    if ($z = 0$){
        $x := x + 1$
    }else{
        $x := x + 2$
    }
}
```

Moreover, fix the postexpectation

$$
f \;=\; [y = 1] \cdot 1 + [y \neq 1] \cdot 0 .
$$

We have for every $h \in \mathsf{LExp}$,

$$
\begin{aligned}
& {}^{\mathsf{wp}}_{C}\Theta_{f}(h) \\
= \quad & [x \leq 1 \wedge z = 0] \cdot (1/2 \cdot h[y, x/1, x+1] + 1/2 \cdot h[y, x/2, x+1]) \\
& + [x \leq 1 \wedge \neg(z = 0)] \cdot (1/2 \cdot h[y, x/1, x+2] + 1/2 \cdot h[y, x/2, x+2]) \\
& + [\neg(x \leq 1) \wedge y = 1] \cdot 1 + [\neg(x \leq 1) \wedge y \neq 1] \cdot 0 \,.
\end{aligned}
$$

Hence, the initial template $\mathcal{T}_1$ is given by

$$
\begin{aligned}
\mathcal{T}_1 \quad = \quad & [x \leq 1 \wedge z = 0] \cdot (\mathfrak{b}_1 + \mathfrak{a}_{1,1} \cdot x + \mathfrak{a}_{1,2} \cdot y + \mathfrak{a}_{1,3} \cdot z) \\
& + [x \leq 1 \wedge \neg(z = 0)] \cdot (\mathfrak{b}_2 + \mathfrak{a}_{2,1} \cdot x + \mathfrak{a}_{2,2} \cdot y + \mathfrak{a}_{2,3} \cdot z) \\
& + [\neg(x \leq 1) \wedge y = 1] \cdot 1 + [\neg(x \leq 1) \wedge y \neq 1] \cdot 0 \,.
\end{aligned}
$$

5.6.2 Template Refinement

We describe three template refinement strategies. The performance of these strategies is evaluated empirically in Section 5.8.

5

Static Hyperrectangle Refinement for Finite-State Loops. This strategy produces fixed-partition templates. For *finite-state* loops, we assume that each program variable is upper-bounded by a given constant. Hence, the (relevant) state space is (a subset of) a bounded hyperrectangle. We obtain template $\mathcal{T}_i$ for $i > 1$ by splitting each dimension of this hyperrectangle into i equally-sized parts[10], thus obtaining (at most) i^m hyperrectangles. Let these hyperrectangles be described by the Boolean expressions $\eta_1, \ldots, \eta_k$ and assume that the *initial* template $\mathcal{T}_1$ is given by

$$
[\mathcal{B}_1] \cdot \mathcal{E}_1 + \ldots + [\mathcal{B}_n] \cdot \mathcal{E}_n + \underbrace{\sum_i [\vartheta_i] \cdot e_i}_{[\neg\varphi] \cdot f} \;.
$$

[10] if possible, i.e., if i does not exceed the size of the dimension

Then, $\mathcal{T}_{i+1}$ is given by

$$\Big(\sum_{j=1}^{k}\big[\mathcal{B}_1 \wedge \eta_j\big]\cdot \mathcal{E}_{1,j}\Big) + \ \ldots\ + \Big(\sum_{j=1}^{k}\big[\mathcal{B}_n \wedge \eta_j\big]\cdot \mathcal{E}_{n,j}\Big) + \underbrace{\sum_{i}[\vartheta_i]\cdot e_i}_{[\neg\varphi]\cdot f}\ ,$$

where the $\mathcal{E}_{l,j}$ are templated arithmetic expressions over $\{x_1,\ldots x_m\}$.

Example 5.5.
Let $\mathcal{T}_1$ be the template

$$\mathcal{T}_1 \ = \ [x < 10 \wedge y < 10]\cdot(\mathfrak{a}\cdot x + \mathfrak{b}\cdot y + \mathfrak{c}) + [\neg(x < 10 \wedge y < 10)]\cdot 1\ .$$

The static hyperrectangle refinement produces

$$\begin{aligned}\mathcal{T}_2 \ = \ \ & [x < 10 \wedge y < 10 \wedge x < 5 \wedge y \geq 5]\cdot(\mathfrak{a}_1\cdot x + \mathfrak{b}_1\cdot y + \mathfrak{c}_1)\\ & + [x < 10 \wedge y < 10 \wedge x < 5 \wedge y < 5]\cdot(\mathfrak{a}_2\cdot x + \mathfrak{b}_2\cdot y + \mathfrak{c}_2)\\ & + [x < 10 \wedge y < 10 \wedge x \geq 5 \wedge y \geq 5]\cdot(\mathfrak{a}_3\cdot x + \mathfrak{b}_3\cdot y + \mathfrak{c}_3)\\ & + [x < 10 \wedge y < 10 \wedge x \geq 5 \wedge y < 5]\cdot(\mathfrak{a}_4\cdot x + \mathfrak{b}_4\cdot y + \mathfrak{c}_4)\\ & + [\neg(x < 10 \wedge y < 10)]\cdot 1\ .\end{aligned}$$

Dynamic Hyperrectangle Refinement. This technique does not require the given loop to be finite-state and produces variable-partition templates. We proceed as for static hyperrectangle refinement but do not fix *where* we split the hyperrectangle, i.e., we introduce template variables in the Boolean expressions encoding the boundaries of the hyperrectangles.

Example 5.6.
Let $\mathcal{T}_1$ be the template

$$\mathcal{T}_1 \ = \ [x < 10 \wedge y < 10]\cdot(\mathfrak{a}\cdot x + \mathfrak{b}\cdot y + \mathfrak{c}) + [\neg(x < 10 \wedge y < 10)]\cdot 1\ .$$

The dynamic hyperrectangle refinement produces

$$\begin{aligned}\mathcal{T}_2 \ = \ \ & [x < 10 \wedge y < 10 \wedge x < \mathfrak{d}_1 \wedge y \geq \mathfrak{d}_2]\cdot(\mathfrak{a}_1\cdot x + \mathfrak{b}_1\cdot y + \mathfrak{c}_1)\\ & + [x < 10 \wedge y < 10 \wedge x < \mathfrak{d}_1 \wedge y < \mathfrak{d}_2]\cdot(\mathfrak{a}_2\cdot x + \mathfrak{b}_2\cdot y + \mathfrak{c}_2)\end{aligned}$$

$$
\begin{aligned}
&+ [x < 10 \land y < 10 \land x \geq \mathfrak{d}_1 \land y \geq \mathfrak{d}_2] \cdot (\mathfrak{a}_3 \cdot x + \mathfrak{b}_3 \cdot y + \mathfrak{c}_3) \\
&+ [x < 10 \land y < 10 \land x \geq \mathfrak{d}_1 \land y < \mathfrak{d}_2] \cdot (\mathfrak{a}_4 \cdot x + \mathfrak{b}_4 \cdot y + \mathfrak{c}_4) \\
&+ [\neg(x < 10 \land y < 10)] \cdot 1 \,.
\end{aligned}
$$

Inductivity-Guided Refinement. This technique produces fixed-partition templates. Suppose the synthesizer reports that for the current template $\mathcal{T}_i$, we have $\langle \mathcal{T}_i \rangle \cap \mathsf{AdmInv} = \emptyset$. We refine using a hint, namely the last partially admissible instance $I \in \langle \mathcal{T}_i \rangle$ provided by the synthesizer. Let

$$
\begin{aligned}
I &= [\eta_1'] \cdot a_1' + \ldots + \left[\eta_k'\right] \cdot a_{k'}' + \underbrace{\sum_i [\vartheta_i] \cdot e_i}_{[\neg\varphi]\cdot f} \text{, and} \\
{}^{\mathsf{wp}}_{C}\Theta_f(I) &= [\eta_1] \cdot a_1 + \ldots + [\eta_k] \cdot a_k + \underbrace{\sum_i [\vartheta_i] \cdot e_i}_{[\neg\varphi]\cdot f} \,.
\end{aligned}
$$

We adapt the construction for computing pointwise minima of piecewise linear expectations from Section 4.6.3.4 to partition States into those parts where I is inductive, i.e., ${}^{\mathsf{wp}}_{C}\Theta_f(I) \sqsubseteq I$, and where it is not. We obtain $\mathcal{T}_{i+1}$ by letting

$$
\begin{aligned}
\mathcal{T}_{i+1} = \quad & \sum_{i=1}^{k} \sum_{j=1}^{k'} \left[\eta_i \land \eta_j' \land a_i \leq a_j'\right] \cdot \mathcal{E}_{i,j,1} + \left[\eta_i \land \eta_j' \land a_i > a_j'\right] \cdot \mathcal{E}_{i,j,2} \\
& + \underbrace{\sum_i [\vartheta_i] \cdot e_i}_{[\neg\varphi]\cdot f} \,.
\end{aligned}
$$

Where the $\mathcal{E}_{i,j,1}$ and $\mathcal{E}_{i,j,2}$ are templated arithmetic expressions over $\{x_1, \ldots, x_m\}$.

5.7 Extensions for Termination and Lower Bounds

We extend our approach to (i) proving *universal positive almost-sure termination* (UPAST) — termination in finite expected runtime on all inputs — by synthesizing piecewise linear upper bounds on expected runtimes, and to (ii) verifying *lower bounds* on possibly unbounded expected outcomes.

5.7.1 Verifying Positive Almost-Sure Termination

We leverage Kaminski et al.'s weakest preexpectation-style ert calculus for reasoning about expected runtimes [KKMO16; KKMO18]:

Theorem 5.8 (Reasoning about Expected Runtimes of Loops).
For every loop $C = \mathtt{while}(B)\{C'\} \in \mathsf{pGCL}$, the monotone function

$$ {}^{\mathsf{ert}}_{C}\Phi \colon \mathbb{E} \to \mathbb{E}\,, \qquad {}^{\mathsf{ert}}_{C}\Phi(X) \;=\; 1 + {}^{\mathsf{wp}}_{C}\Phi_0(X)\,, $$

obtained from ${}^{\mathsf{wp}}_{0}\Phi_C$ (cf. Definition 2.21 on page 63) satisfies

$$ \left(\mathsf{lfp}\ {}^{\mathsf{ert}}_{C}\Phi\right)(\sigma) \;=\; \text{“expected number of loop guard evaluations when executing } C \text{ on } \sigma\text{"}\,. $$

In particular,

$$ C \text{ is UPAST} \qquad \text{iff} \qquad \text{for all } \sigma \in \mathsf{States}\colon \left(\mathsf{lfp}\ {}^{\mathsf{ert}}_{C}\Phi\right)(\sigma) < \infty\,. $$

All properties of ${}^{\mathsf{wp}}_{C}\Phi_0$ relevant to our approach carry over to ${}^{\mathsf{ert}}_{C}\Phi$, thus enabling the synthesis ert-*superinvariants* from $\mathsf{FinLExp}$, i.e.,

$$ I \in \mathsf{FinLExp} \qquad \text{with} \qquad \underbrace{{}^{\mathsf{ert}}_{C}\Phi(I) \sqsubseteq I}_{I \text{ is ert-superinvariant (of } C)}\,. $$

Such I *upper-bound the expected number of loop iterations for every initial state* [KKMO16, Theorem 3] and, since instances $I \in \langle \mathcal{T} \rangle$ of templates $\mathcal{T} \in \mathsf{TExp}$ never evaluate to infinity, I witnesses UPAST of the given loop. We refer to [Kam19, Chapter 7] for an in-depth treatment of the ert-calculus.

Example 5.7.
Consider the geometric loop C given by

$$ \mathtt{while}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x + 1\}\}\,. $$

Then C is UPAST as witnessed by the ert-superinvariant

$$ I \;=\; [y = 1] \cdot 3 + [y \neq 1] \cdot 1\,. $$

I being an ert-superinvariant tells us that the expected number of loop guard evaluations when executing C on an initial state σ with $\sigma(y) = 1$ is at most 3. To see that I is an ert-superinvariant, consider the following:

$$\begin{aligned} & {}^{\mathsf{ert}}_{C}\Phi(I) \\ = \; & 1 + {}^{\mathsf{wp}}_{C}\Phi_0(I) \\ = \; & 1 + [y = 1] \cdot (1/2 \cdot 1 + 1/2 \cdot 3) + [y \neq 1] \cdot 0 \\ = \; & [y = 1] \cdot 3 + [y \neq 1] \cdot 1 \\ \sqsubseteq \; & I \,. \end{aligned}$$

Our implementation synthesizes I automatically.

5.7.2 Verifying Lower Bounds on Expected Outcomes

Consider the problem of verifying a *lower* bound $g \sqsubseteq \mathsf{wp}[\![C]\!](f)$ for some loop $C = \mathtt{while}(\varphi)\{C'\} \in \mathsf{LpGCL}$ and $f, g \in \mathsf{FinLExp}$. It is straightforward to modify our approach for the synthesis of wp-*subinvariants* from $\mathsf{FinLExp}$, i.e.,

$$I \in \mathsf{LExp} \qquad \text{with} \qquad \underbrace{I \sqsubseteq {}^{\mathsf{wp}}_{C}\Phi_f(I)}_{I \text{ is wp-subinvariant (of } C \text{ w.r.t. } f)} \,.$$

As mentioned in Section 2.4.4, it is important to note that wp-subinvariants *do not necessarily lower-bound* $\mathsf{wp}[\![C]\!](f)$. Therefore, Hark et al. [HKGK20] have proposed a more involved yet sound induction rule for lower bounds.

Theorem 5.9 (Adapted from Hark et al. [HKGK20]).
Let $\mathcal{T} \in \mathsf{TnExp}$ be a natural template and let $C = \mathtt{while}(\varphi)\{C'\}$ and $f \in \mathsf{LExp}$ not containing ∞. Moreover, let $I \in \langle \mathcal{T} \rangle$. If

1. $I \in \mathsf{FinLExp}$,
2. $I \sqsubseteq {}^{\mathsf{wp}}_{C}\Phi_f(I)$,
3. C is UPAST (cf. Section 5.7.1), and
4. there is $\beta \in \mathbb{R}_{\geq 0}$ such that for all $\sigma \models \varphi$, we have[a]
$$\underbrace{{}^{\mathsf{wp}}_{C}\Phi_f\Big(|I - [\![I]\!](\sigma)|\Big)(\sigma) \;\leq\; \beta}_{I \text{ is } \textit{conditionally difference bounded (c.d.b.)}} \,,$$

then

$$I \sqsubseteq \mathsf{wp}[\![C]\!](f) \,.$$

[a]For $X \in \mathbb{E}$ and $\alpha \in \mathbb{R}_{\geq 0}$, we define $|X - \alpha| = \lambda\sigma. |X(\sigma) - \alpha|$.

Analogously to Theorem 5.1, given $\mathcal{T} \in \mathsf{TExp}$, we can *compute* $\mathcal{T}' \in \mathsf{TExp}$ such that

$$\text{for all template valuations } \mathfrak{I}\colon \quad \mathcal{T}[\mathfrak{I}] \in \mathsf{LExp} \quad \text{implies} \quad [\![\mathcal{T}'[\mathfrak{I}]]\!] = \lambda\sigma.\, {}^{\mathsf{wp}}_{C}\Phi_f\,(|\mathcal{T}[\mathfrak{I}] - [\![\mathcal{T}[\mathfrak{I}]]\!](\sigma)|)(\sigma)\,,$$

which facilitates the extension of our verifier and synthesizer for encoding and checking conditional difference boundedness. Hence, we can employ our framework for verifying $g \sqsubseteq \mathsf{wp}[\![C]\!](f)$ by (i) proving UPAST of C as described in Section 5.7.1 and (ii) synthesizing a c.d.b. sub-invariant I with $g \sqsubseteq I$. We refer to [Har21] for an in-depth and more general treatment of Theorem 5.9.

Example 5.8.
Reconsider the geometric loop C given by

$$\mathtt{while}(y = 1)\{\{y := 0\}\,[1/2]\,\{x := x+1\}\}\,.$$

and suppose we aim to verify by Theorem 5.9 that $g \sqsubseteq \mathsf{wp}[\![C]\!](f)$ holds for

$$g = f = x\,.$$

For that, we choose

$$I = [y = 1]\cdot(x+1) + [y \neq 1]\cdot x \quad \in \quad \mathsf{FinLExp}\,.$$

We have seen that ${}^{\mathsf{wp}}_{C}\Phi_f(I) = I$ so, in particular, $I \sqsubseteq {}^{\mathsf{wp}}_{C}\Phi_f(I)$, i.e., I is a wp-subinvariant of C w.r.t. f. Moreover, C is UPAST by Example 5.7. Finally, I is c.d.b. for $\beta = 1$ because we have for all $\sigma \models \varphi$,

$$\begin{aligned}
&{}^{\mathsf{wp}}_{C}\Phi_f\big(|I - [\![I]\!](\sigma)|\big)(\sigma)\\
=\;& \Big(1/2\cdot(|I - [\![I]\!](\sigma)|)[y/0] + 1/2\cdot(|I - [\![I]\!](\sigma)|)[x/x+1]\Big)(\sigma)\\
=\;& \Big(1/2\cdot(|x - [\![I]\!](\sigma)|) + 1/2\cdot(|x+1 - [\![I]\!](\sigma)|)\Big)(\sigma)\\
=\;& \Big(1/2\cdot(|\sigma(x) - (\sigma(x)+1)|) + 1/2\cdot(|\sigma(x)+1 - (\sigma(x)+1)|)\Big)(\sigma)\\
=\;& 1/2\cdot 1 + 1/2\cdot 0 \;\leq\; 1\,.
\end{aligned}$$

Our implementation automatically proves UPAST and synthesizes I and β.

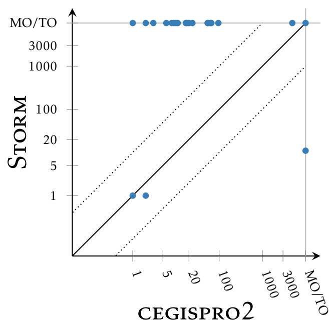
5.8 Implementation and Experiments

We have implemented a prototype[11] of our techniques called CEGISPRO2 — CEGIS for PRObabilistic PROgrams — in Python 3.11 using the SMT solver Z3 [MB08] and the solver-API PySMT [GM15]. We investigate the applicability and scalability of our approach with a focus on the expressive power of piecewise linear expectations. Moreover, we compare with three state-of-the-art tools — Storm [HJKQ+22; DJKV17], Absynth [NCH18], and Exist [BTPH+22] — on subsets of their benchmarks fitting into our framework.

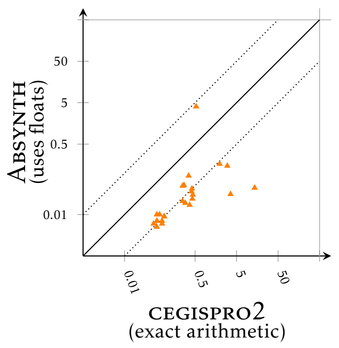
Finite-State Loops. Figure 5.3a depicts experiments on verifying upper bounds on expected outcomes of *finite-state* loops, including the model of the bounded retransmission protocol (brp) from Section 5.2 and the IPv4 zeroconf protocol (zeroconf) [CAG05] adapted from [BK08, Chapter 10]. The size of the relevant state space — the number of states satisfying the loop guard — ranges from 10^5 to 10^{16}. For each benchmark, i.e., program and specification with increasingly sharper upper bounds on expected outcomes, we evaluate CEGISPRO2 on all template refinement strategies (cf. Section 5.6.2). More details on these experiments are provided in Appendix 3.1. We compare to both explicit- and symbolic-state engines of the probabilistic model checker Storm 1.6.3 with exact arithmetic. Storm implements efficient linear programming-based model checking methods. Figure 5.3a depicts the runtime of the best configuration for Storm and CEGISPRO2, respectively. Details are provided in Table 1 on page 276.
Results. (i) Our CEGIS approach synthesizes admissible invariants for a variety of programs. We mostly find *syntactically small* invariants with a *small number of counterexamples* compared to the size of the relevant state space (cf. Table 1). This indicates that piecewise linear expectations can be sufficiently expressive for the verification of finite-state programs. The overall performance of CEGISPRO2 depends highly on the sharpness of the given bounds. (ii) Our approach can outperform state-of-the-art *explicit- and symbolic-state* model checking techniques and can scale to huge state spaces. There are also simple programs where our method fails to find an admissible invariant (gridbig, cf. Appendix 3.1) or finds admissible invariants only for rather simple properties while requiring many counterexamples (gridsmall). Whether we need more sophisticated template refinements or whether these programs are not amenable to piecewise linear expectations is left for future work. (iii) There is no clear winner between the template refinement strategies producing *fixed-partition* templates

[11]The prototype is available open-source from `https://github.com/moves-rwth/cegispro2`.

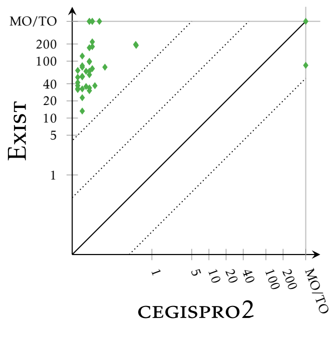
5



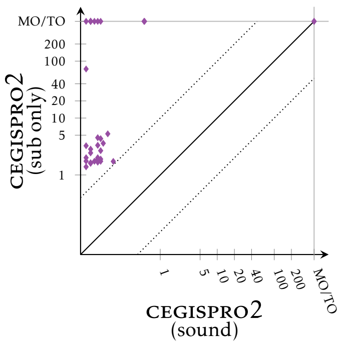
(a) Verifying upper bounds for finite-state loops (TO=2h).



(b) Verifying UPAST.



(c) Synthesizing wp-subinvariants (TO=5min).



(d) Synthesizing wp-subinvariants only vs. verifying lower bounds by Theorem 5.9 (TO=5min).

Figure 5.3: Performance of cegispro2 vs. state-of-the-art tools on three verification tasks (time in seconds, log-scaled; MO=8GB). Markers above the solid line depict benchmarks where cegispro2 is faster (in different orders of magnitude marked by the dashed lines), and vice versa.

— the static hyperrectangle refinement and the inductivity-guided refinement (cf. Table 1). We further observe that the strategy producing *variable partition* templates — the dynamic hyperrectangle refinement — is not competitive as significantly more time is spent in the synthesizer to solve formulae with Boolean structures[12]. We conclude that searching for suitable fixed-partition templates in a separate outer loop (cf. Figure 5.1 on page 168) can pay off.

Verifying UPAST. Figure 5.3b depicts experiments on proving UPAST of (possibly infinite-state) loops taken from [NCH18] restricted to linear loops with flattened nested loops. We compare to the linear programming-based tool Absynth [NCH18] for computing upper bounds on expected runtimes. These benchmarks do not require template refinements. More details on these experiments are provided in Appendix 3.2 and Table 2 on page 282.
Results. cegispro2 can prove UPAST of various infinite-state programs from the literature using very few counterexamples. Absynth mostly outperforms cegispro2[13]. The runtime bounds synthesized by cegispro2 are often as tight as the bounds synthesized by Absynth (cf. Table 2).

Verifying Lower Bounds. Figure 5.3c depicts experiments aiming to verify *lower* bounds on expected outcomes of (possibly infinite-state) loops taken from [BTPH+22]. We compare to Exist [BTPH+22][14], which combines CEGIS with sampling- and ML-based techniques. However, Exist synthesizes wp-subinvariants only, which might be unsound for proving lower bounds (cf. [Kam19, Counterexample 5.8]). Thus, for a fair comparison, Figure 5.3c depicts experiments where *both* Exist and cegispro2 synthesize wp-subinvariants only. More details on these experiments are provided in Appendix 3.3. In Figure 5.3d, we compare cegispro2 that finds wp-subinvariants only with cegispro2 that *additionally* proves UPAST and c.d.b., thus soundly verifying lower bounds by Theorem 5.9 on page 191. No benchmark requires template refinements.
Results. cegispro2 is capable of verifying lower bounds on expected outcomes and outperforms Exist (on 30/32 benchmarks) for synthesizing wp-subinvariants. Additionally verifying UPAST and c.d.b. naturally requires more time. A manual inspection reveals that, for most TO/MO cases in Figure 5.3d,

[12]Recall from Theorem 5.6 that, for fixed-partition templates, the synthesizer can be implemented via linear programming.
[13]Absynth uses floating-point arithmetic whereas cegispro2 uses exact arithmetic.
[14]Exist supports parametric probabilities, which are not supported by our tool. We have instantiated these parameters with varying probabilities to enable a comparison.

there is no suitable c.d.b. wp-subinvariant. One benchmark times out because we could not prove UPAST for that benchmark.

5.9 Future and Related Work

Future Work. We have restricted to piecewise *linear* expectations. A natural direction for future work is to investigate whether our approach can be extended to piecewise *polynomial* expectations in a practically feasible manner. This is unclear since, e.g., a straightforward extension would yield the verifier to reason about undecidable nonlinear integer arithmetic. Another direction for future work is to investigate more sophisticated template refinement strategies.

The gridbig benchmark mentioned in the previous section emphasizes the relevance of pursuing the aforementioned directions: For C given by

```
while($x < 1000 \wedge y < 1000$){
    {$x := x+1$} [$1/2$] {$y := y+1$}
},
```

Storm verifies the specification

$$\mathsf{wp}[\![C]\!]([y=1000]) \;\sqsubseteq\; [x=0 \wedge y=0]\cdot 0.99 + [\neg(x=0\wedge y=0)]\cdot\infty$$

within 11 seconds whereas cegispro2 times out after 2 hours. One could argue that both the program C itself and the above specification are rather simple since for the initial state σ with $\sigma(x)=\sigma(y)=0$, we have

$$\mathsf{wp}[\![C]\!]([y=1000])(\sigma) \;=\; 0.5\,.$$

The reason for cegispro2's bad performance is unclear and needs to be investigated: While there certainly *is* a piecewise linear admissible invariant, namely

$$\begin{aligned} I \;=\; & \sum_{\sigma\models x<1000\wedge y<1000} [x=\sigma(x)\wedge y=\sigma(y)]\cdot \mathsf{wp}[\![C]\!]([y=1000])(\sigma) \\ & + [\neg(x<1000\wedge y<1000)\wedge y=1000]\cdot 1 \\ & + [\neg(x<1000\wedge y<1000)\wedge y\neq 1000]\cdot 0\,, \end{aligned}$$

this invariant is not particularly *concise* in the sense that its syntactic representation enumerates all (relevant) states. The question of whether there is a *significantly more concise* piecewise linear admissible invariant remains open.

If this is the case, we could investigate how to improve our template refinement strategies for finding suitable templates. Otherwise, we could investigate whether, e.g., *polynomial* expectations are more suited.

Moreover, our CEGIS framework can be combined with k-induction for probabilistic programs from Chapter 4: One could try to synthesize k-inductive invariants for $k > 1$, which might mitigate the need for template refinements and yield more concise invariants. Finally, it would be interesting to extend our framework to *weighted programs* [15], which might, e.g., enable the automatic competitive analysis of online algorithms [BE98].

Related Work. Besides the comparisons in Section 5.8, we discuss works in invariant synthesis, probabilistic model checking, and symbolic inference.

Invariant Synthesis. Template-based (qualitative) invariant synthesis for nonprobabilistic programs was pioneered by Colón and et al. [CSS03]. In the nonprobabilistic setting, ICE [GLMN14] is a template-based, counterexample guided technique for learning invariants. [UTK21] presents a CEGIS-based approach for the relational verification of programs. [KUH21] combines decision-tree learning with CEGIS for verifying program termination. More inductive synthesis approaches are surveyed in [ABDF$^+$15; FB18].

Our approach is largely inspired by template-based *quantitative* invariant synthesis for probabilistic programs, which was pioneered by Katoen et al. [KMMM10]. They consider the synthesis of wp-subinvariants for loops over $\mathbb{R}$-valued program variables, which allows to leverage Motkzin's transposition theorem [Mot36] to derive constraints on the template variables characterizing *all* inductive instances of a given template. Rather than solving these constraints in a counterexample-guided manner, they are solved in one shot using quantifier elimination for nonlinear real arithmetic [Tar48]. The approach has been implemented in the tool Prinsys [GKM13; Gre16], which, in contrast to our approach, also supports *nonlinear* programs and invariants as well as *parametric* probabilities for probabilistic choices. However, templates have to be provided by the user and automatic template refinement — in case, e.g., the given template does not admit an inductive instance — is not considered. Moreover, [GKM13; Gre16] considers benchmarks with rather small templates (at most 3 summands and at most 3 template variables). On the other hand, we have shown that our incremental counterexample-guided approach for the piecewise linear setting scales to templates with up to 23 summands and ca. 60 template variables. It would be interesting to investigate whether the quantifier elimination-based techniques employed by Prinsys can also scale to such larger templates. A direct comparison of cegispro2 and Prinsys is, however, not meaningful since cegispro2 synthesizes inductive invariants for verifying a given bound, whereas Prinsys synthesizes *all* inductive instances of a given template without considering a given bound. Moreover, Prinsys cannot synthesize wp-superinvariants for upper-bounding expected outcomes.

Other constraint solving-based approaches [FZJZ$^+$17; CHWZ15] also aim to synthesize (non-piecewise) *polynomial* invariants for proving lower bounds on expected outcomes of programs over $\mathbb{R}$-valued program variables. In particular, [CHWZ15] also obtains constraints from counterexamples ensuring certain conditions on candidate invariants. Apart from various technical differences, we identify the following conceptual differences: (i) we support *piecewise* defined expectations; (ii) we consider strategies for refining templates; (iii) we focus on the integration of fast verifiers over efficiently decidable theories; and (iv) we do not need assumptions on the boundedness of expectations.

Various *martingale-based approaches*, such as [CS13; CFG16; CNZ17; FC19; FH15; ACN18; MBKK21], aim to synthesize quantitative invariants over $\mathbb{R}$-valued variables. In particular, [TOUH21; TOUH18] provide surveys on martingale-based approaches for approximating reachability probabilities in (possibly nondeterministic and polynomial) probabilistic programs. By combining martingale-, fixpoint-, and category-theoretic observations, they obtain new martingale-based proof rules. These rules are combined with (linear and polynomial) template-based techniques, yielding new synthesis algorithms. Our approach — when restricted to reachability probabilities[15] — might benefit from these observations. Finally, [BEFH16] synthesizes bounds on expected outcomes using a symbolic construction based on Doob's decomposition, which, however requires user-supplied hints.

[AGR21] employs a CEGIS loop to train a neural network dedicated to learning a ranking supermartingale witnessing UPAST of (possibly continuous) probabilistic programs. They also use SMT solvers to check the supermartingale condition and use the provided counterexamples to guide the learning process. Besides the discussed tool Absynth [NCH18], the tools KoAT [MHG21], eco-imp [AMS20], pRaML [WKH20], and the approach from [WFGC$^+$19] also consider the automatic synthesis of bounds on expected runtimes (or more general resources) of (possibly nondeterministic) probabilistic programs. [KG23; GABE$^+$17] combine constraint solving with dependency pairs [AG00] to automatically prove almost-sure termination of probabilistic term rewriting.

All of these synthesis techniques can be classified into two categories:

[15]e.g., when reasoning about expected outcomes of the form $\mathsf{wp}[\![C]\!]([\vartheta])$

1. *Correct-by-construction* techniques use, e.g., quantifier elimination over the reals to decide whether there exist an admissible invariant I in a given template $\mathcal{T}$. This obviates the need for a verifier.

2. *Counterexample-guided* techniques, like our CEGIS approach, which potentially produce non-admissible invariants in a guess-and-check loop.

Except for our CEGIS approach and [GLMN14; AGR21; UTK21; KUH21; CHWZ15; BTPH+22], all of the above-mentioned works apply correct-by-construction techniques. An advantage of correct-by-construction techniques is that[16] they can decide whether there is an admissible invariant in a given template $\mathcal{T}$. For our approach, this is guaranteed only for finite-state loops (cf. Theorem 5.7). We have opted for a counterexample-guided approach for two reasons: First, to the best of our knowledge, all correct-by-construction techniques assume a continuous state space (e.g., $\mathbb{R}_{\geq 0}$-valued program variables) rather than a discrete one (e.g., $\mathbb{N}$-valued program variables) so that quantifier elimination techniques apply. In a setting like ours where we are actually considering $\mathbb{N}$-valued program variables, this is a *problem relaxation*: A template $\mathcal{T}$ might give rise to an admissible invariant for $\mathbb{N}$-valued program variables even though this is *not* the case for $\mathbb{R}_{\geq 0}$-valued program variables. Correct-by-construction methods over continuous state spaces might, therefore, "miss" admissible invariants. Investigating the practical relevance of this aspect and whether there are correct-by-construction methods for $\mathbb{N}$-valued program variables is an interesting direction for future work. Second, producing counterexamples and thereby *partially* admissible invariants is crucial for our inductivity-guided template refinement strategy (cf. Section 5.6.2).

Our CEGIS framework is related to quantifier instantiation techniques implemented in modern SMT solvers. These can be classified into conflict-based [RTM14; Bar17], model-based [GM09; RTGK+13], enumerative [RBF18], counterexample-guided [BJ15; RDKB+15], and syntax-guided techniques [PNB17; NPRB+21]. Our CEGIS framework can be considered a mixture of model-based and syntax-guided instantiation techniques tailored to the verification of bounds on expected outcomes of probabilistic loops.

The recurrence solving-based approaches in [BKS19; BKS20b] synthesize non-linear invariants for moments of program variables. The underlying algebraic techniques are confined to the subclass of *prob-solvable loops*.

Probabilistic Model Checking. Symbolic probabilistic model checking mostly uses

[16] when restricting the class of considered programs and specifications appropriately (e.g., when restricting to linear loops and piecewise linear expectations over $\mathbb{R}_{\geq 0}$-valued program variables)

algebraic decision diagrams [BCHK+97; AKNP+00], representing the transition relation symbolically and using equation solving or value iteration [BKLP+17; HK20; QK18] on that representation. Alternative CEGIS approaches synthesize Markov chains [CHJK21] and probabilistic programs [ACJK21; ACJK+21].

Symbolic Inference. Finite-horizon probabilistic inference employs weighted model counting via either decision diagrams annotated with probabilities as in Dice [HvM20; HJVM+21] or approximate versions by SAT/SMT-solvers [CFMV15; CMMV16; CDM17; RWKY+14; BPv15]. PSI [GMV16] determines symbolic representations of exact distributions. Prodigy [CKKW22] decides whether a probabilistic loop agrees with an (invariant) specification.

6 Property Directed Reachability

This chapter is based on our prior publications [7; 6].

Aaron R. Bradley's algorithm IC3 (shorthand for ***I****ncremental construction of* ***I****nductive clauses for* ***I****ndubitable correctness*) [Bra11a] has been a leap forward in symbolic model checking of transition systems. One year after IC3 was published, Een, Mischenko, and Brayton [EMB11] proposed some improvements for Bradley's algorithm and coined their variant *Property Directed Reachability* (PDR, for short), which is nowadays among the state-of-the-art approaches in the field of symbolic hardware and software model checking[1].

Markov decision processes can be thought of as probabilistic extensions of transition systems. Given the success of IC3, this raises the question:

Can we transfer the key principles underlying IC3 *to the probabilistic setting to obtain a scalable algorithm for reasoning about safety of Markov decision processes?*

Our goal is to develop theoretical foundations to answer this question. Towards this end, we present PrIC3 (pronounced pricy-three) — a *conservative* and *truly quantitative* extension of IC3 to symbolic model checking of MDPs. Our main focus is to develop the theory underlying PrIC3. Alongside, we present a first implementation of PrIC3 for model checking finite-state probabilistic programs in a symbolic manner by leveraging the weakest preexpectation calculus.

Chapter Outline. In Section 6.1, we treat the key principles underlying IC3. Equipped with these key principles, we then present our PrIC3 framework for reasoning about safety of Markov decision processes in Section 6.2.

6.1 Foundations of IC3 for Transition Systems

In this section, we aim to obtain an understanding of the key principles underlying Bradley's IC3 algorithm with adaptions from PDR. The results presented

[1] Even though we adapt some of the improvements from PDR, we will mostly stick to the original name IC3 since most of our results are inspired by Bradley's seminal work.

in this section are not new. However, the presentation differs from [Bra11a; EMB11]: Bradley as well as Een, Mischenko, and Brayton's presentation is tailored to the symbolic nature of IC3, i.e., to the fact that IC3 heavily exploits the *symbolic encodings* of the transition systems it operates on. We, on the other hand, *abstract* from such encodings and present IC3 in fixpoint-theoretic terms[2] to focus on its essence. It is for this reason that we will moreover omit several *optimizations* of IC3, which are inessential to its soundness.

Section Outline. Section 6.1.1 details the input to IC3 and the decision problem it tackles. Inductive invariants for verifying safety and the (loop) invariants IC3 maintains throughout its execution are treated in Section 6.1.1.1 and Section 6.1.1.2, respectively. The IC3 main loop is presented in Section 6.1.2. *Strengthening* in IC3 — a procedure invoked by the IC3 main loop — is discussed in Section 6.1.3. Finally, in Section 6.1.3, we discuss a key ingredient for IC3's scalability called *(inductive) generalization*.

6.1.1 Setting

As with k-induction and BMC (cf. Section 4.2), IC3 verifies or refutes invariant properties of (finite-state) transition systems, i.e., whether the set of states reachable in a given transition system covers all states in a given set of "safe" states. To align the problem tackled by IC3 with the one tackled by PrIC3 for MDPs (cf. Section 6.2.1), we present a variant[3] of IC3 for solving an equivalent problem: Throughout this section, fix a finite-state transition system[4]

$$\mathsf{TS} = (\mathcal{S}, \longrightarrow, \{s_I\}) \qquad \text{and} \qquad \text{a set } B \subseteq \mathcal{S} \text{ of "bad" states}\ .$$

IC3 verifies or refutes whether some state in B is reachable from the initial state s_I. To formalize this, we define *the set of states reaching B* as

$$\Diamond B \;=\; \{s \in \mathcal{S} \mid \text{exists finite execution fragment } s_0 \ldots s_n \colon s_0 = s \text{ and } s_n \in B\}\ .$$

We call TS *safe* if $s_I \not\in \Diamond B$. The goal of IC3 can thus be cast as:

Verify or refute that TS is safe.

[2]The foundations of this perspective on IC3 have been laid in [4, Chapter 3] of the author's master thesis. The formalization presented here is improved and aligned with the subsequently developed results presented in Section 6.2 and treats inductive generalization more formally.
[3]bearing close resemblance to what [EMB11] calls "dual PDR ", which is detailed in [SS17].
[4]cf. Section 2.1.1.

Refuting safety of TS boils down to determining a *single* finite execution fragment from s_I to some state in B — *a counterexample to safety*. For *verifying* safety, IC3 computes an inductive invariant, which is detailed in Section 6.1.1.1.

Adopting IC3 terminology, we call sets $\mathcal{F} \subseteq \mathcal{S}$ of states *frames*. We will often identify frames $\mathcal{F}$ with their indicator functions of type $\mathcal{S} \to \{0,1\}$, i.e.,

$$\mathcal{F}(s) \;=\; \begin{cases} 1 & \text{if } s \in \mathcal{F} \\ 0 & \text{otherwise}\,. \end{cases}$$

In particular, we have $\mathcal{F}(s) \leq 0$ iff $s \notin \mathcal{F}$ and $\mathcal{F}(s) > 0$ iff $s \in \mathcal{F}$. Now, given a frame $\mathcal{F}$, we define *set of (direct) predecessors of states in $\mathcal{F}$* as

$$\mathsf{Preds}(\mathcal{F}) \;=\; \{s \in \mathcal{S} \mid \text{exists } t \in \mathcal{F} \colon s \longrightarrow t\}\;.$$

We moreover define the *qualitative Bellman operator* as the frame transformer

$$\Psi \colon (\mathcal{S} \to \{0,1\}) \to (\mathcal{S} \to \{0,1\})\,, \qquad \Psi(\mathcal{F}) \;=\; B \cup \mathsf{Preds}(\mathcal{F})\,.$$

The operator Ψ can be shown to be monotone and continuous w.r.t. complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$ induced by TS. Hence, $\mathsf{lfp}\ \Psi$ exists uniquely by Tarski's Theorem 2.1 on page 20. Crucially, we have

$$\Diamond B \;=\; \mathsf{lfp}\ \Psi\;.$$

6.1.1.1 Inductive Frames

If TS is safe, IC3 finds a frame $\mathcal{F} \in \mathcal{S} \to \{0,1\}$ such that (i) the initial state s_I does not belong to $\mathcal{F}$ and (ii) applying the qualitative Bellman operator Ψ to $\mathcal{F}$ does not take us up in the partial order $\subseteq$ on frames, i.e.,

$$\text{(i)} \quad \mathcal{F}(s_I) \;\leq\; 0 \qquad \text{and} \qquad \text{(ii)} \quad \Psi(\mathcal{F}) \;\subseteq\; \mathcal{F}\;.$$

Intuitively, (i) postulates the *hypothesis* that s_I cannot reach B and (ii) expresses that $\mathcal{F}$ is closed under adding bad states and taking predecessors, thus affirming the hypothesis. Frames satisfying the above conditions are called *inductive invariants* in IC3. We adopt this terminology. By *Park induction* (Lemma 2.4 on page 28), which in our setting reads

$$\Psi(\mathcal{F}) \;\subseteq\; \mathcal{F} \qquad \text{implies} \qquad \mathsf{lfp}\ \Psi \;=\; \Diamond B \;\subseteq\; \mathcal{F}\,,$$

an inductive invariant $\mathcal{F}$ *witnesses* that TS is safe because

$$(\Diamond B)(s_I) \;=\; \Big(\mathsf{lfp}\ \Psi\Big)(s_I) \;\leq \mathcal{F}(s_I) \;\leq\; 0\,.$$

Such an inductive invariant exists iff TS is safe, which can be seen by taking $\mathcal{F} = \Diamond B$. If no inductive invariant exists, then IC3 will find a *counterexample to safety*: a *finite execution fragment* from the initial state s_I to a bad state in B.

6.1.1.2 The IC3 invariants

IC3 aims to find the inductive invariant by maintaining a *sequence of frames*

$$\mathcal{F}_0 \;\subseteq\; \mathcal{F}_1 \;\subseteq\; \mathcal{F}_2 \;\subseteq\; \ldots$$

such that $\mathcal{F}_i$ overapproximates the set

$$\Diamond^{\leq i} B \textit{ of states reaching } B \textit{ within at most } i \textit{ steps}\,,$$

which we characterize using the qualitative Bellman operator: For any $i \geq 0$,

$$\Diamond^{\leq i} B \;=\; \Psi^{i+1}(\emptyset)\,.$$

Notice that the set $\Diamond B$ of states reaching B in an *unbounded* number of steps is

$$\begin{aligned}
&\Diamond B\\
=\;& \mathsf{lfp}\ \Psi\\
\overset{(*)}{=}\;& \bigcup\{\Psi^n(\emptyset) \mid n \in \mathbb{N}\}\\
=\;& \bigcup\Big\{\Diamond^{\leq n} B \mid n \in \mathbb{N}\Big\}\,,
\end{aligned}$$

where $(*)$ is a consequence of Kleene's Theorem 2.2 on page 22.

The sequence $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \subseteq \ldots$ maintained by IC3 should frame-wise overapproximate the sequence $\Psi(\emptyset) \subseteq \Psi^2(\emptyset) \subseteq \Psi^3(\emptyset) \ldots$. Pictorially:

$$\begin{array}{ccccccccccc}
 & & \mathcal{F}_0 & \subseteq & \mathcal{F}_1 & \subseteq & \mathcal{F}_2 & \subseteq & \ldots & \subseteq & \mathcal{F}_k\\
 & & \cup\!\!\shortmid & & \cup\!\!\shortmid & & \cup\!\!\shortmid & & & & \cup\!\!\shortmid\\
\emptyset & \subseteq & \Psi(\emptyset) & \subseteq & \Psi^2(\emptyset) & \subseteq & \Psi^3(\emptyset) & \subseteq & \ldots & \subseteq & \Psi^{k+1}(\emptyset)
\end{array}$$

However, the sequence $\Psi(\emptyset), \Psi^2(\emptyset), \Psi^3(\emptyset), \ldots$ will never explicitly be known to IC3. Instead, IC3 will ensure the above frame-wise overapproximation property implicitly by enforcing the so-called IC3 *invariants* on the frame sequence:

Definition 6.1 (IC3 Invariants).
Let $k \geq 0$. We say that the frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ satisfy the IC3 *invariants*, a fact we denote by $\mathsf{IC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$, if all of the following hold:

1.	***Initiality:***	$\mathcal{F}_0 \;=\; \Psi(\emptyset)$	
2.	***Chain Property:***	for all $0 \leq i < k$:	$\mathcal{F}_i \subseteq \mathcal{F}_{i+1}$
3.	***Frame-Safety:***	for all $0 \leq i < k$:	$\mathcal{F}_i(s_I) \leq 0$
4.	***Relative Inductivity:***	for all $0 \leq i < k$:	$\Psi(\mathcal{F}_i) \subseteq \mathcal{F}_{i+1}$

Notice that $\mathcal{F}_k$ possibly contains s_I, which is an adaption from PDR [EMB11]. The IC3 invariants enforce the above picture: The *chain property* ensures

$$\mathcal{F}_0 \;\subseteq\; \mathcal{F}_1 \;\subseteq\; \ldots \;\subseteq\; \mathcal{F}_k\,.$$

Moreover, we have $\Psi(\emptyset) = \mathcal{F}_0 \subseteq \mathcal{F}_0$ by *initiality*. Assuming $\Psi^{i+1}(\emptyset) \subseteq \mathcal{F}_i$ as induction hypothesis, monotonicity of Ψ and *relative inductivity* imply

$$\Psi^{i+2}(\emptyset) \;\subseteq\; \Psi(\mathcal{F}_i) \;\subseteq\; \mathcal{F}_{i+1}\,.$$

By overapproximating $\Psi(\emptyset), \Psi^1(\emptyset), \Psi^2(\emptyset), \ldots$, the frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ indeed overapproximate the sets $\Diamond^{\leq 0}B$, $\Diamond^{\leq 1}B$, $\Diamond^{\leq 2}B, \ldots$:

Lemma 6.1 (IC3 Invariants yield Overapproximations).
Let $\mathcal{F}_0, \ldots, \mathcal{F}_k$ be frames satisfying the IC3 invariants. Then

$$\text{for all } i \in \{0, \ldots, k\}\colon \quad \Diamond^{\leq i}B \;\subseteq\; \mathcal{F}_i\,.$$

In particular, a sequence of frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ satisfying the IC3 invariants witnesses that s_I cannot reach B within $i \in \{0, \ldots, k-1\}$ steps by frame-safety.

Given two frames $\mathcal{F}, \mathcal{F}'$, we say that $\mathcal{F}'$ is a *strengthening* of $\mathcal{F}$, if $\mathcal{F}' \subseteq \mathcal{F}$. We say that a sequence $\mathcal{F}'_0, \ldots, \mathcal{F}'_k$ of frames is a strengthening of $\mathcal{F}_0, \ldots, \mathcal{F}_k$, if $\mathcal{F}'_i$ is a strengthening of $\mathcal{F}_i$ for all $i \in \{0, \ldots, k\}$.

Lemma 6.2 (Step-Bounded Safety via Strengthenings).
Let $\mathcal{F}_0, \ldots, \mathcal{F}_k$ be frames satisfying the IC3 invariants. There is a strengthening $\mathcal{F}'_0, \ldots, \mathcal{F}'_k$ of $\mathcal{F}_0, \ldots, \mathcal{F}_k$ satisfying

1. $\mathsf{IC3Inv}\left(\mathcal{F}'_0, \ldots, \mathcal{F}'_k\right)$, and
2. $\mathcal{F}'_k(s_I) \leq 0$

if and only if $s_I \not\in \Diamond^{\leq k}B$.

As for proving that s_I cannot reach B in an *unbounded* number of steps, it suffices to find two consecutive frames, say $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$, that coincide:

Algorithm 7: IC3 (TS, B)

```
input: Finite transition system TS, set of bad states $B$ with $s_I \notin B$
output: true, if TS is safe; otherwise false
$\mathcal{F}_0 \leftarrow \Psi(\emptyset)$;   $\mathcal{F}_1 \leftarrow \mathcal{S}$;   $k \leftarrow 1$
while true do
    $success, \mathcal{F}_0, \ldots, \mathcal{F}_k \leftarrow$ Strengthen $(\mathcal{F}_0, \ldots, \mathcal{F}_k)$
    if $\neg success$ then return false                          // $s_I \in \Diamond B$
    $\mathcal{F}_{k+1} \leftarrow \mathcal{S}$
    $\mathcal{F}_0, \ldots, \mathcal{F}_{k+1} \leftarrow$ Propagate $(\mathcal{F}_0, \ldots, \mathcal{F}_{k+1})$
    if $\exists\, 1 \leq i \leq k \colon \mathcal{F}_i = \mathcal{F}_{i+1}$ then return true     // $s_I \notin \Diamond B$
    $k \leftarrow k + 1$
```

Figure 6.1: The IC3 main loop.

Lemma 6.3 (Inductive Invariants from IC3 Invariants).
Let $\mathcal{F}_0, \ldots, \mathcal{F}_k$ be frames satisfying the IC3 invariants. Then

$$\text{exists } i < k \colon \mathcal{F}_i = \mathcal{F}_{i+1} \qquad \text{implies} \qquad s_I \notin \Diamond B\,.$$

Proof. $\mathcal{F}_i = \mathcal{F}_{i+1}$ and relative inductivity yield

$$\Psi(\mathcal{F}_i) \subseteq \mathcal{F}_{i+1} = \mathcal{F}_i\,.$$

With frame-safety, i.e., $\mathcal{F}_i(s_I) \leq 0$, this renders $\mathcal{F}_i$ an inductive invariant. ■

6.1.2 The IC3 Main Loop

Lemma 6.3 provides IC3's angle of attack for proving a TS safe. Repeatedly add and strengthen frames while preserving the IC3 invariants until two consecutive frames coincide, yielding an inductive invariant. This approach is taken by the main loop IC3 depicted in Algorithm 7. As input, IC3 takes a transition system $\mathsf{TS} = (\mathcal{S}, \longrightarrow, \mathcal{S}_I)$ and a set $B \subseteq \mathcal{S}$ of bad states with[5] $s_I \notin B$. Since the input is never changed, we assume it to be *globally available*, also to subroutines. As output, IC3 returns true if TS is safe, and it returns false if B is

[5] This assumption simplifies the presentation and can be easily checked upfront.

reachable from s_I. Let us go through the individual steps of IC3.

How IC3 *works.* Recall that IC3 maintains a sequence of frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ which is initialized in l. 1 with $k = 1$, $\mathcal{F}_0 = \Psi(\emptyset)$, and $\mathcal{F}_1 = \mathcal{S}$, which implies IC3Inv$(\mathcal{F}_0, \ldots, \mathcal{F}_k)$. The **while**-loop in l. 2 maintains IC3Inv$(\mathcal{F}_0, \ldots, \mathcal{F}_k)$ at all times. In l. 3, procedure Strengthen — detailed in Section 6.1.3 — is called to determine whether $s_I \in \lozenge^{\leq k} B$. By Lemma 6.2, this is equivalent to determining whether the sequence of frames can be strengthened while preserving the IC3 invariants and ensuring $\mathcal{F}_k(s_I) \leq 0$. It either returns true if successful or returns false if it was unable to do so. The latter case indicates that TS is unsafe. Formally, Strengthen meets the following specification:

Definition 6.2 (Specification of Strengthen).
Procedure Strengthen is *sound*, if for all $k \geq 1$ and all sequences $\mathcal{F}_0, \ldots, \mathcal{F}_k$ of frames with IC3Inv$(\mathcal{F}_0, \ldots, \mathcal{F}_k)$, we have:

1. If
$$\text{Strengthen}\,(\mathcal{F}_0, \ldots, \mathcal{F}_k) = \text{true}, \mathcal{F}_0', \ldots, \mathcal{F}_k'\,,$$
then
$$\text{IC3Inv}\left(\mathcal{F}_0', \ldots, \mathcal{F}_k'\right) \quad \text{and} \quad \mathcal{F}_k(s_I) \leq 0\,.$$
2. If Strengthen$(\mathcal{F}_0, \ldots, \mathcal{F}_k) = $ false, then $s_I \in \lozenge B$.

If Strengthen returns true, then a new frame $\mathcal{F}_{k+1} = \mathcal{S}$ is created in l. 5 and the extended sequence $\mathcal{F}_0, \ldots, \mathcal{F}_{k+1}$ of frames again satisfies the IC3 invariants.

Propagation (l. 6) aims to speed up termination by updating $\mathcal{F}_{i+1}(s)$ by $\mathcal{F}_i(s)$ if this does not violate the IC3 invariants. Consequently, the previously mentioned properties remain unchanged. Since propagation can be regarded as an optimization, we omit the details to focus on the fundamental aspects of IC3.

In l. 7, we check whether we have produced two identical consecutive frames. If so, TS is safe by Lemma 6.3. Consequently, IC3 returns true. Otherwise, we increment k and are in the same setting as upon entering the loop, now with an extended frame sequence; IC3 then performs another iteration.

IC3 *terminates for unsafe TS.* The following argument requires termination of Strengthen and Propagate, which we justify in the next section. If TS is unsafe, then there exists a step-bound $n \in \mathbb{N}$, such that

$$s_I \in \lozenge^{\leq n} B\,.$$

Algorithm 8: Strengthen $(\mathcal{F}_0,\ldots,\mathcal{F}_k)$

```
input: Frames $\mathcal{F}_0,\ldots,\mathcal{F}_k$ with $k \geq 1$ and IC3Inv $(\mathcal{F}_0,\ldots,\mathcal{F}_k)$
output: See Definition 6.2 on page 207
1  $Q \leftarrow \{(k, s_I)\}$
2  while $Q$ not empty do
3      $(i,s) \leftarrow Q$.popMin()    /* pop obligation with minimal frame index */
4      if $i = 0$ then
           /* TS is unsafe */
5          return false, $Q$.states();
       /* check whether $\mathcal{F}_i(s) \leftarrow 0$ violates relative inductivity */
6      if $\Psi(\mathcal{F}_{i-1})(s) > 0$ then choose $s' \in \mathcal{F}_{i-1}$ with $s \longrightarrow s'$
           /* determine whether $s'$ reaches $B$ within $i-1$ steps */
7          $Q$.push $((i-1, s'), (i, s))$
8      else
           /* resolve $(i,s)$ without violating relative inductivity */
9          $\mathcal{F}_1(s) \leftarrow 0; \ldots; \mathcal{F}_i(s) \leftarrow 0$
   /* $Q$ empty; all obligations have been resolved */
10 return true, $\mathcal{F}_0,\ldots,\mathcal{F}_k$
```

Figure 6.2: Strengthening in IC3. If Strengthen returns false, then Q.states() is a set of states forming a finite execution fragment from s_I to B.

The specification of Strengthen implies that it returns false as soon as it is invoked on some sequence $\mathcal{F}_0,\ldots\mathcal{F}_{n+1}$ of length $n+2$ because the frames cannot be strengthened accordingly by Lemma 6.2. Consequently, IC3 terminates in l. 4.

IC3 *terminates for safe TS.* Assume again that Strengthen and Propagate terminate. The specification of Strengthen implies that it never returns false. Since TS is finite-state, every infinite chain $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \ldots$ eventually stabilizes[6], yielding IC3's inductivity check in l. 7 to eventually succeed.

6.1.3 Strengthening in IC3

When the loop of IC3 invokes Strengthen, we have $\mathsf{IC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$ and $\mathcal{F}_k(s_I) = 1$. The task of Strengthen is to determine whether the frames can be strengthened in such a way that the IC3 invariants remain valid and $\mathcal{F}_k(s_I) \leq 0$, which, by Lemma 6.2, is equivalent to determining whether it holds that

$$s_I \in \Diamond^{\leq k} B\,.$$

To this end, Strengthen generates *(proof) obligations*: The first obligation is to prove that state $s = s_I$ cannot reach B within $i = k$ steps. Such an obligation is denoted by (i, s). To prove that s cannot reach B within i steps, we have to prove that each *direct successor* s' of s cannot reach B within $i-1$ steps. For that, Strengthen considers the frame $\mathcal{F}_{i-1}$ which, by Lemma 6.1, *overapproximates* the set of states reaching B within $i-1$ steps: If there is no $s' \in \mathcal{F}_{i-1}$ with $s \longrightarrow s'$, we have proven that indeed $s \notin \Diamond^{\leq i} B$. We can then *resolve* the obligation (i, s) by excluding s from $\mathcal{F}_i$ *without violating the* IC3 *invariants* because, intuitively, $\mathcal{F}_{i-1}$ is strong enough to support the claim $s \notin \Diamond^{\leq i} B$.

If, on the other hand, there *is* an $s' \in \mathcal{F}_{i-1}$ with $s \longrightarrow s'$, there are two possibilities: We *either* have (i) $s \notin \Diamond^{\leq i} B$ but $\mathcal{F}_{i-1}$ is not yet strong enough to support this claim and must hence be strengthened accordingly *or* (ii) we have $s' \in \Diamond^{\leq i-1} B$ and thus $s \in \Diamond^{\leq i} B$. To determine which of these possibilities holds, Strengthen spawns a new obligation $(i-1, s')$, which is attempted to be resolved before reconsidering (i, s).

This way, Strengthen eventually excludes s_I from $\mathcal{F}_k$ without violating the IC3 invariants, thereby proving that $s_I \notin \Diamond^{\leq k} B$, or eventually spawns an obligation of the form $(0, s)$ with $s \in B$. In the latter case, Strengthen determined a finite execution fragment from s_I to some state in B, witnessing unsafety of TS.

Strengthen is given by the pseudo code in Algorithm 8. It preserves $\mathsf{IC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$ at all times. To keep track of unresolved obligations (i, s), Strengthen employs a priority queue Q which pops obligations with minimal frame index i first. This queue is initialized in l. 1 with (s_I, k). The subsequently executed **while**-loop then checks whether Q is empty. If so, all obligations have been resolved and $\mathcal{F}_k(s_I) \leq 0$. If not, we pop some obligation (i, s) with minimal frame index i from Q in l. 3. If $i = 0$, then $s \in B$ and the states Q.states() form a finite execution fragment from s_I to $s \in B$, yielding Strengthen to return false in l. 5. Otherwise, Strengthen checks whether frame $\mathcal{F}_{i-1}$ is strong enough to

[6] This is because the complete lattice $(\mathcal{P}(\mathcal{S}), \subseteq)$ induced by TS has *finite height*.

support the claim $s \not\in \Diamond^{\leq i} B$ in l. 6. Notice that

$$\begin{aligned}
& \Psi(\mathcal{F}_{i-1})(s) > 0 \\
\text{iff} \quad & s \in B \cup \mathsf{Preds}(\mathcal{F}_{i-1}) \\
\text{iff} \quad & s \in \mathsf{Preds}(\mathcal{F}_{i-1}) \\
& (s \not\in B \text{ because } s_I \text{ reaches } s \text{ in } k-i \text{ steps and } s_I \not\in \Diamond^{\leq k-i} B \text{ by Lemma 6.1}) \\
\text{iff} \quad & \text{exists } s' \in \mathcal{F}_{i-1} : s \longrightarrow s' .
\end{aligned}$$

If such an s' exists, we push both the new obligation $(i-1, s')$ and the current obligation (i, s) onto Q. If $\Psi(\mathcal{F}_{i-1})(s) \leq 0$, then $\mathcal{F}_{i-1}$ is strong enough to support the claim $s \not\in \Diamond^{\leq i} B$ and (i, s) can be resolved by excluding s from all frames $\mathcal{F}_1, \ldots, \mathcal{F}_i$ without violating the IC3 invariants.

Strengthen *terminates.* We now show that Strengthen as in Algorithm 8 terminates. The only scenario in which Strengthen may not terminate is if it keeps spawning obligations in l. 7. Let us thus look closer at how obligations are spawned: Whenever we detect that resolving an obligation (i, s) would violate relative inductivity for some direct successor s' of s (l. 6), we first need to update the value of this successor in frame $i-1$, i.e., we push the obligation $(i-1, s')$. Since the obligation queue Q pops obligations with minimal frame index first, resolving $(i-1, s')$ either eventually succeeds or yields Strengthen to return false. Hence, for every state s, Strengthen spawns at most $|\mathsf{Succs}(s)|$ obligations, which implies termination since TS is finite-state.

Example 6.1 (IC3 on a Safe Transition System).
Let us consider an example run of IC3 on a safe transition system. Consider the transition system depicted in Figures 6.3 to 6.5. We have $B = \{s_4\}$ (colored in red). Figure 6.3 depicts the frames maintained by IC3 right *after* the first iteration. Figure 6.4 and Figure 6.5 depict the frames right after the second and third iteration, respectively. IC3 terminates after the third iteration since $\mathcal{F}_1$ and $\mathcal{F}_2$ coincide.

Example 6.2 (IC3 on an Unsafe Transition System).
Figure 6.6 depicts the frames maintained by IC3 on the given transition system at the *beginning* of the third iteration. The transition system is unsafe since s_I reaches B in 3 steps. IC3 determines this via the subsequent

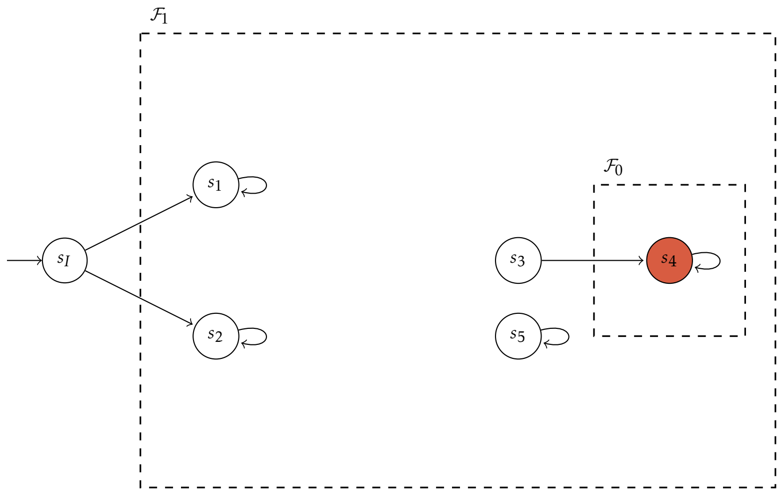


Figure 6.3: Frames for the transition system considered in Example 6.1 right *after* the first iteration of IC3. Here $B = \{s_4\}$.

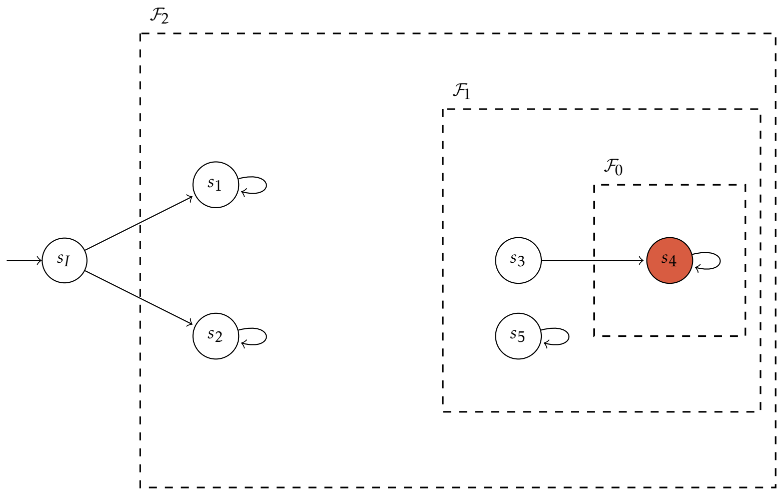


Figure 6.4: Frames for the transition system considered in Example 6.1 right *after* the second iteration of IC3. Here $B = \{s_4\}$.

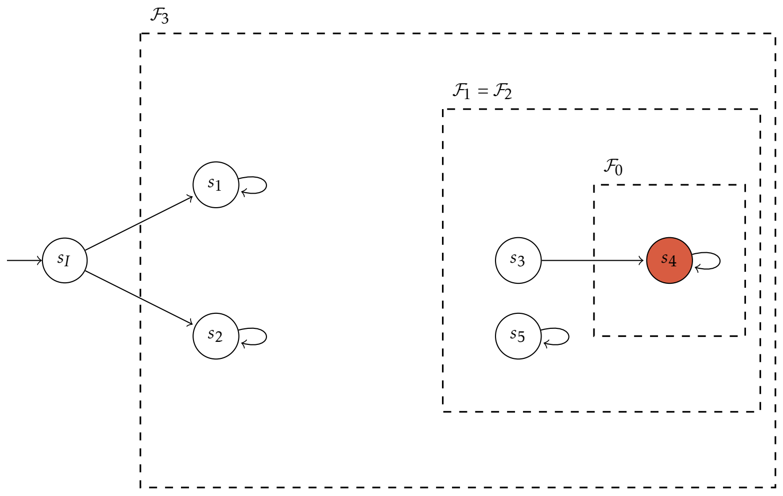


Figure 6.5: Frames for the transition system considered in Example 6.1 right *after* the third iteration of IC3. Here $B = \{s_4\}$.

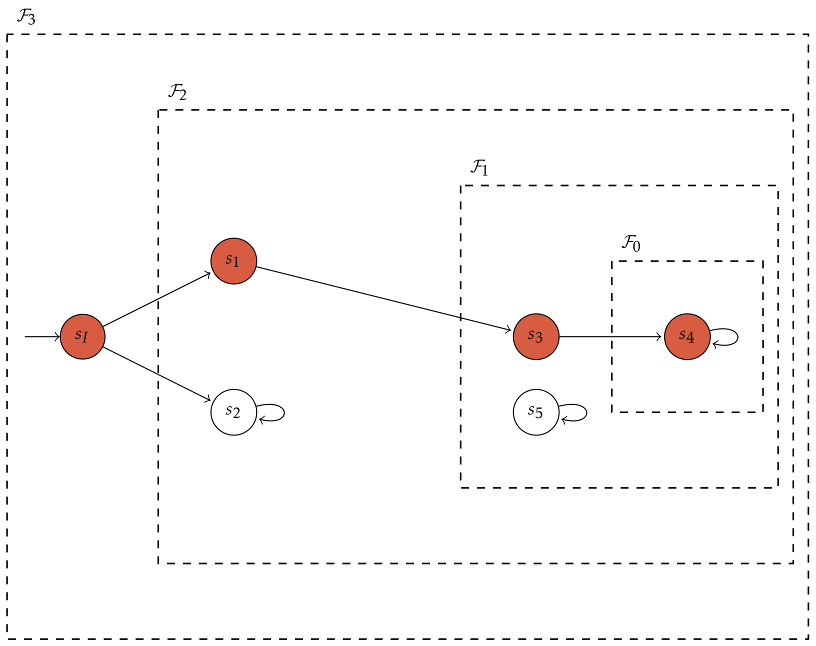


Figure 6.6: Frames for the transition system considered in Example 6.2 at the *beginning* of the third iteration of IC3. Here $B = \{s_4\}$.

call to Strengthen, which spawns the obligations

$$(3, s_I) \quad \text{then} \quad (2, s_1) \quad \text{then} \quad (1, s_3) \quad \text{then} \quad (0, s_4)\,.$$

Since s_4 cannot be excluded from $\mathcal{F}_0$ without violating initiality, the obligation queue (Q.states()) contains a finite execution fragment from the initial state to the bad state s_4. Consequently, Strengthen and IC3 return false.

6.1.4 On Generalization in IC3

Examples 6.1 and 6.2 illustrate that IC3 basically performs a breadth-first search from the initial state until it either finds an inductive invariant or a finite execution fragment to some bad state. One might — for good reason — ask: Why does IC3 even work in such an incremental manner by maintaining a sequence of frames? Why not simply perform the breadth-first search in a *single* frame? These questions are reasonable as maintaining and refining ever more frames even seems to introduce some overhead.

So far, we have omitted one of the key components for IC3's scalability. This ingredient is called *(inductive) generalization*. Generalization is not essential for soundness or termination but makes IC3 scale. For generalization, it is crucial that the given transition system is represented in a *symbolic* manner. The idea is then to exploit this symbolic representation to abstract — or "generalize" — individual states appearing in some proof obligation to a whole *set of states*, thereby excluding more than just a single state at a time when resolving an obligation in Strengthen. In what follows, we first describe the conceptual idea underlying generalization in Section 6.1.4.1. We then exemplify in Section 6.1.4.2 how generalization is realized in bit-level IC3 for finite transition systems. An in-depth treatment of generalization and additional optimizations of IC3 is outside the scope of this thesis. Rather, our goal is to identify the fundamental ideas in order to transfer them to the probabilistic setting.

6

6.1.4.1 Generalization: Conceptually

Generalization in IC3 takes place right before l. 9 in Strengthen (Algorithm 8 on page 208). Suppose an obligation (i, s) can be resolved without violating the IC3 invariants by executing

$$\mathcal{F}_1(s) \leftarrow 0; \ldots; \mathcal{F}_i(s) \leftarrow 0\,. \qquad (6.1)$$

Generalization aims to exclude *more* states than just s without violating the IC3 invariants to accelerate the search for an inductive invariant. Put more formally, the goal is to *heuristically guess* a frame $\mathcal{F}$ such that

$$\text{(i)}\ \ \mathcal{F}(s) \leq 0 \qquad \text{and} \qquad \text{(ii)}\ \ \Psi(\mathcal{F}_i) \subseteq \mathcal{F}$$

We call such an $\mathcal{F}$ a *generalization of s w.r.t. $\mathcal{F}_i$*. Instead of updating the frames by executing (6.1), we then update the frames using the generalization $\mathcal{F}$:

$$\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cap \mathcal{F}; \ldots; \mathcal{F}_i \leftarrow \mathcal{F}_i \cap \mathcal{F}\ ,$$

Condition (ii) implies that this update will not violate the IC3 invariants, and Condition (i) yields that we indeed resolve the obligation (i,s).

This already reveals an advantage of maintaining a sequence $\mathcal{F}_0,\ldots,\mathcal{F}_k$ of frames satisfying the IC3 invariants: By Lemma 6.1, frame $\mathcal{F}_i$ is an overapproximation of the states reaching B within i steps, i.e.,

$$\Diamond^{\leq i} B \subseteq \mathcal{F}_i\ .$$

Hence, for any frame $\mathcal{F}$,

$$\Psi(\mathcal{F}_i) \subseteq \mathcal{F}$$

is a sufficient condition for $\mathcal{F}$ being an overapproximation of the states reaching B within $i+1$ steps *that only requires us to reason about a single step of the transition relation*. To see this, consider the following:

$$\begin{aligned}
& \Diamond^{\leq i} B \subseteq \mathcal{F}_i && \text{(Lemma 6.1 on page 205)}\\
\text{implies}\quad & \Psi\left(\Diamond^{\leq i} B\right) \subseteq \Psi(\mathcal{F}_i) && \text{(monotonicity of } \Psi)\\
\text{implies}\quad & \Diamond^{\leq i+1} B \subseteq \Psi(\mathcal{F}_i) && \text{(by definition)}\\
\text{implies}\quad & \Diamond^{\leq i+1} B \subseteq \mathcal{F}\ . && \text{(assumption and transitivity of } \subseteq)
\end{aligned}$$

6.1.4.2 Generalization in Bit-Level IC3 by Example

In bit-level IC3 [Bra11a], states are represented symbolically as assignments from propositional variables — called *state-variables* — to truth values. The transition relation is represented symbolically as a propositional formula over these state variables. This enables (i) to represent gigantic but structured transition systems in a concise manner and (ii) to leverage the efficiency of

$$\begin{tikzcd}
{} \arrow[r] & \neg x_1\neg x_2x_3x_4 \arrow[r] & \neg x_1x_2x_3x_4 \arrow[r] & x_1\neg x_2x_3x_4 \arrow[r] & x_1x_2x_3x_4 \arrow[loop right] & \\
& x_1\neg x_2\neg x_3x_4 \arrow[d] & & & & \\
& \neg x_1\neg x_2\neg x_3x_4 \arrow[r] & \neg x_1x_2\neg x_3x_4 \arrow[r] & x_1x_2\neg x_3x_4 \arrow[r] & \neg x_1\neg x_2x_3\neg x_4 \arrow[r] & \neg x_1x_2x_3\neg x_4 \arrow[loop right]
\end{tikzcd}$$

Figure 6.7: A symbolic transition system with four $\{0,1\}$-valued state-variables. 6 irrelevant states are omitted. $B = \{\neg x_1 x_2 x_3 \neg x_4\}$.

SAT solvers for reasoning about, e.g., direct successors of sets of states, as it is required by the relative inductivity, check in l. 6 of Strengthen (Algorithm 8 on page 208). We do not treat such symbolic encodings in great detail. Rather, we outline — on a conceptual level — how bit-level IC3 leverages these symbolic encodings to obtain suitable generalizations, which is key to making IC3 scale.

Consider the transition system TS depicted in Figure 6.7 on page 217 as an example. IC3 *without* generalization requires 5 iterations to find an inductive invariant. We will now see that IC3 *with* generalization requires only 3 iterations. The states of this transition system are given as assignments from the finite ordered sequence $\mathsf{Vars} = (x_1, \ldots, x_4)$ of state-variables to $\{0,1\}$, i.e.,

$$\mathcal{S} \;=\; \{s \mid s\colon \mathsf{Vars} \to \{0,1\}\}\;.$$

Frames can thus be represented as propositional formulae over Vars. We introduce some terminology. A *cube* is a conjunction of literals, e.g.,

$$\neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4 \quad \text{which we denote more concisely by} \quad \neg x_1 x_2 x_3 \neg x_4\;.$$

A *clause* is a negated cube, e.g.,

$$\neg(x_3 \neg x_4)\;.$$

The above clause represents the set of states satisfying it, i.e., the frame

$$\{s \mid s(x_3) = 0 \text{ or } s(x_4) = 1\}\;.$$

A run of IC3 *with generalization.* In the first iteration, IC3 maintains the frames

$$\mathcal{F}_1 \;=\; \mathsf{true} \quad \text{and} \quad \mathcal{F}_0 \;=\; B \;=\; \neg x_1 x_2 x_3 \neg x_4\;.$$

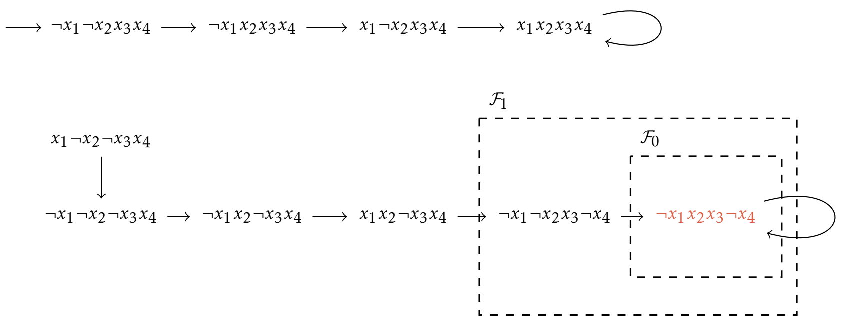

$\neg x_1 \neg x_2 x_3 x_4$
$\neg x_1 x_2 x_3 x_4$
$x_1 \neg x_2 x_3 x_4$
$x_1 x_2 x_3 x_4$
$\mathcal{F}_1$
$\mathcal{F}_0$
$x_1 \neg x_2 \neg x_3 x_4$
$\neg x_1 \neg x_2 \neg x_3 x_4$
$\neg x_1 x_2 \neg x_3 x_4$
$x_1 x_2 \neg x_3 x_4$
$\neg x_1 \neg x_2 x_3 \neg x_4$
$\neg x_1 x_2 x_3 \neg x_4$

Figure 6.8: The frames after the first iteration of IC3 with generalization.

The subsequent call to Strengthen excludes

$$s_I \;=\; \neg x_1 \neg x_2 x_3 x_4$$

from $\mathcal{F}_1$. This can be done by conjoining the clause obtained from negating this cube to $\mathcal{F}_1$, i.e.,

$$\mathcal{F}_1 \;\leftarrow\; \mathcal{F}_1 \wedge \neg(\neg x_1 \neg x_2 x_3 x_4)\,.$$

Now, when applying generalization, rather than just excluding this single state from $\mathcal{F}_1$, IC3 *heuristically guesses* an entire *set of states* containing s_I that can be excluded from $\mathcal{F}_1$ without violating the IC3 invariants. The heuristic proposed by Bradley [Bra11a] works as follows: We gradually "drop literals" — from left to right, say — from the cube representing s_I. We drop as many literals as possible such that excluding all states represented by the so-obtained cube from $\mathcal{F}_1$ does not violate relative inductivity. In our example, it turns out that we can exclude all states represented by the cube x_4 because we have

$$\Psi(\mathcal{F}_0) \;\subseteq\; \neg(x_4)\,.$$

We thus update $\mathcal{F}_1$, which is now given by

$$\mathcal{F}_1 \;=\; \neg(x_4)\,.$$

So, instead of one state, we have excluded $2^3 = 8$ states from $\mathcal{F}_1$. This situation is depicted in Figure 6.8 on page 218. Notice that we excluded a state which reaches B, namely $x_1x_2\neg x_3x_4$. Consequently, *there is no way that* $\mathcal{F}_1$ *becomes the sought-after inductive invariant* in future iterations since IC3 never adds states back to some frame. We are now facing one of the main reasons for IC3's incremental nature. Let us consider the next iteration.

We initialize a new frame $\mathcal{F}_2$ so that our sequence of frames is given by

$$\mathcal{F}_2 = \mathsf{true} \qquad \mathcal{F}_1 = \neg(x_4) \qquad \mathcal{F}_0 = B\,.$$

The subsequent call to Strengthen now aims to exclude s_I from $\mathcal{F}_2$ without violating relative inductivity. First, observe that this is possible *without spawning obligations for* $\mathcal{F}_1$ because there is no direct successor of s_I in $\mathcal{F}_1$. Again, IC3 aims to exclude more states from $\mathcal{F}_2$ by dropping literals from

$$s_I = \neg x_1\neg x_2x_3x_4\,.$$

This time, however, we *cannot* exclude all states satisfying x_4 from $\mathcal{F}_2$ without violating relative inductivity: The state $x_1x_2\neg x_3x_4$ has a direct successor in $\mathcal{F}_1$, namely $\neg x_1\neg x_2x_3\neg x_4$. We *can*, however, exclude the states represented by

$$x_3x_4\,,$$

from $\mathcal{F}_2$ without violating inductivity. Hence, the updated sequence of frames is

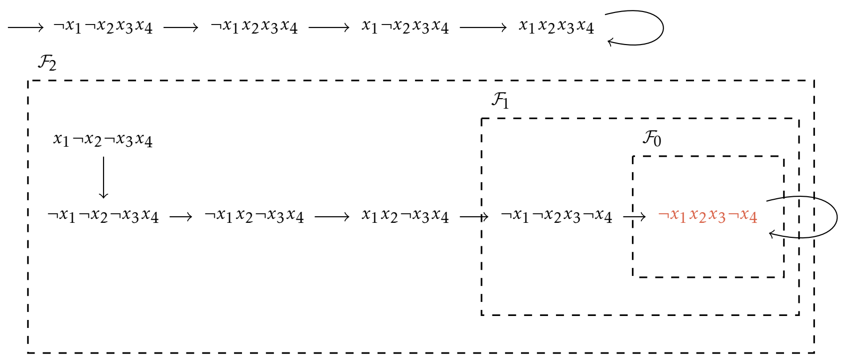
$$\mathcal{F}_2 = \neg(x_3x_4) \qquad \mathcal{F}_1 = \neg(x_4) \wedge \neg(x_3x_4) \qquad \mathcal{F}_0 = B\,.$$

The situation is depicted in Figure 6.9 on page 220. Notice that $\mathcal{F}_2$ is an inductive invariant. In the next iteration[7], IC3 will determine this by strengthening the new frame $\mathcal{F}_3$ in such a way that

$$\mathcal{F}_2 = \mathcal{F}_3\,.$$

Discussion. Bradley's heuristic is rather aggressive as it tries to drop as many literals as possible. We have seen that this heuristic can — in some sense – be *too* aggressive as it potentially excludes states from frames that reach B. However, as the maintained sequence of frames grows, IC3 learns which literals to drop to obtain an inductive invariant. In our example, $\mathcal{F}_1$ merely served as an intermediate proof step — or "lemma" [Bra11a] — yielding IC3 to realize in subsequent iterations that x_3 should not be dropped from s_I.

[7]With *propagation* (l. 6 in Algorithm 7 on page 206), IC3 even terminates after 2 iterations: Whenever this does not violate relative inductivity, Propagate$(\mathcal{F}_0,\ldots,\mathcal{F}_{k+1})$ conjoins the clauses comprising $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$. Since $\mathcal{F}_2$ is an inductive invariant, *all* clauses comprising $\mathcal{F}_2$ can be conjoined to the new frame $\mathcal{F}_3$, thus yielding $\mathcal{F}_2 = \mathcal{F}_3$ without a further iteration. We omit further details.



$\neg x_1 \neg x_2 x_3 x_4$
$\neg x_1 x_2 x_3 x_4$
$x_1 \neg x_2 x_3 x_4$
$x_1 x_2 x_3 x_4$
$\mathcal{F}_2$
$\mathcal{F}_1$
$x_1 \neg x_2 \neg x_3 x_4$
$\mathcal{F}_0$
$\neg x_1 \neg x_2 \neg x_3 x_4$
$\neg x_1 x_2 \neg x_3 x_4$
$x_1 x_2 \neg x_3 x_4$
$\neg x_1 \neg x_2 x_3 \neg x_4$
$\neg x_1 x_2 x_3 \neg x_4$

Figure 6.9: The frames after the second iteration of IC3 with generalization.

As mentioned in the previous section, it is evident that a crucial advantage of maintaining a sequence of frames satisfying the IC3 invariants is that, at any point in time, it suffices to reason about *one* step of the transition relation. Hence, in practice, IC3's relative inductivity checks can be implemented as (relatively) cheap SAT queries. This is in contrast to, e.g., bounded model checking (cf. Section 4.2.1) or k-induction for transition systems (cf. Section 4.2.2), which potentially reason about many steps of the transition relation at once, which can be expensive.

6.2 PrIC3: Property Directed Reachability for MDPs

In this section, we present our conservative and quantitative extension of IC3 for reasoning about safety of Markov decision processes.

Section Outline. We set the stage in Section 6.2.1 by describing the input to PrIC3 and the decision problem it tackles. Section 6.2.2 discusses in which sense our algorithm conservatively extends IC3 for transition systems. In Section 6.2.3, we address the challenges of developing PrIC3. Section 6.2.4 details the core loop of PrIC3, quantitative inductive invariants for verifying safety of Markov decision processes, and the loop invariants PrIC3 preserves throughout its execution. *Strengthening* in PrIC3 is discussed in Section 6.2.5. In contrast to IC3, PrIC3 requires an *additional outermost loop*, which is presented in Section 6.2.6. Finally in Section 6.2.7, we present *practical* PrIC3 for probabilistic programs, where we marry PrIC3 with the weakest preexpectation calculus, yielding a symbolic model checking algorithm. We provide first ideas for *(inductive) generalization* in PrIC3 and present our implementation alongside experimental results. Finally, in Section 6.3, we discuss future and related work.

6.2.1 Setting

Throughout this section, fix a finite-state[8]

$$\begin{gathered}\text{MDP } \mathcal{M} \;=\; (\mathcal{S}, \mathsf{Act}, P),\ \text{an initial state } s_I \in \mathcal{S}, \\ \text{a set } B \subseteq \mathcal{S} \text{ of “bad” states, and a threshold probability } \lambda \in [0,1] \cap \mathbb{Q}\,.\end{gathered}$$

PrIC3 verifies or refutes that the *maximal* probability[9] to reach some state in B from the initial state s_I is at most λ, i.e.,

$$\mathsf{MaxPr}\left(s_I \models \Diamond B\right) \;\leq\; \lambda\,.$$

We call $\mathcal{M}$ *safe* if the above inequality holds. In other words, $\mathcal{M}$ is safe if the probability of reaching B from s_I is at most λ *regardless of how the nondeterminism in $\mathcal{M}$ is resolved*. The goal of PrIC3 can thus be cast as:

$$\boxed{\text{Verify or refute that } \mathcal{M} \text{ is safe.}}$$

[8]cf. Section 2.2
[9]cf. Section 2.2.3

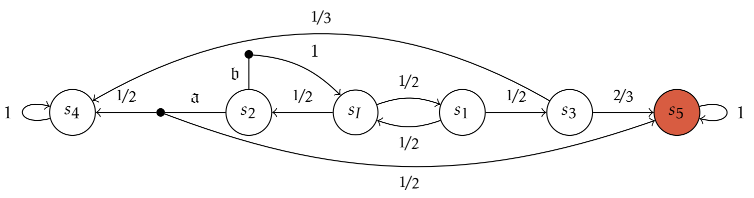


Figure 6.10: A finite-state MDP with $B = \{s_5\}$ serving as a running example. For states s without outgoing action label, we assume that $\mathsf{Act}(s) = \{\mathfrak{a}\}$.

Whereas IC3 operates on functions of type States $\to \{0,1\}$, PrIC3 operates on functions of type States $\to [0,1]$, assigning a probability to each state of $\mathcal{M}$. Adapting IC3 terminology, throughout this section, *frames* $\mathcal{F}$ are functions of type States $\to [0,1]$. We denote by $(\mathbb{V}_{\leq 1}, \sqsubseteq)$ the *complete lattice of frames*, where

$$\mathbb{V}_{\leq 1} \;=\; \{\mathcal{F} \mid \mathcal{F}\colon \mathcal{S} \to [0,1]\} \qquad \text{and} \qquad \mathcal{F} \sqsubseteq \mathcal{F}' \text{ iff for all } s \in \mathcal{S}\colon \mathcal{F}(s) \leq \mathcal{F}'(s)\,.$$

Our theoretical development is based on the max-Bellman operator from Definition 2.15 on page 41 characterizing maximal reachability probabilities. Since we restrict to reachability probabilities rather than more general reachability-rewards, we simplify the presentation by adapting its definition as follows:

Definition 6.3 (Bellman Operator for Reachability Probabilities). Given a set $\mathsf{Act}' \subseteq \mathsf{Act}$ of actions, we define the *Bellman operator for* Act' as

$$\Phi_{\mathsf{Act}'}\colon \mathbb{V}_{\leq 1} \to \mathbb{V}_{\leq 1}\,, \quad \Phi_{\mathsf{Act}'}(\mathcal{F}) \;=\; \lambda s. \begin{cases} 1 & \text{if } s \in B \\ \max\limits_{\mathfrak{a} \in \mathsf{Act}'} \sum\limits_{s' \in \mathcal{S}} P(s,\mathfrak{a},s') \cdot \mathcal{F}(s') & \text{otherwise}\,. \end{cases}$$

Moreover, we write $\Phi_{\mathfrak{a}}$ for $\Phi_{\{\mathfrak{a}\}}$ and Φ for Φ_{Act} and call Φ the *Bellman operator*.

The Bellman operator Φ is continuous w.r.t. the complete lattice $(\mathbb{V}_{\leq 1}, \sqsubseteq)$ and we have [BK08, Theorem 10.100]

$$\text{for all } s \in \mathcal{S}\colon \quad \mathsf{MaxPr}(s \models \lozenge B) \;=\; (\mathsf{lfp}\;\Phi)(s)\,,$$

where the least fixpoint $\mathsf{lfp}\;\Phi$ of the Bellman operator is guaranteed to exist uniquely by Tarski's Fixpoint Theorem 2.1 on page 20.

Example 6.3.
The MDP $\mathcal{M}$ in Figure 6.10 consists of 6 states with initial state s_I and bad states $B = \{s_5\}$. In s_2, actions $\mathfrak{a}$ and $\mathfrak{b}$ are enabled; in all other states, only $\mathfrak{a}$ is enabled. We have

$$(\mathsf{lfp}\ \Phi)(s_I) \;=\; \mathsf{MaxPr}(s_I \models \Diamond B) \;=\; 2/3\,.$$

Hence, $\mathcal{M}$ is safe for all $\lambda \geq 2/3$ and unsafe for all $\lambda < 2/3$.

6.2.2 Recovering IC3 for Transition Systems

We aim to develop PrIC3 in such a way that it conservatively extends IC3 for transition systems. We will therefore accompany the presentation by *recovery statements* to justify this backward compatibility.

As a first prerequisite, we observe that the verification problem tackled by PrIC3 generalizes the problem tackled by IC3. We say that MDP $\mathcal{M}$ is a transition system[10], if there is at most one successor for each state-action-pair, i.e., if

$$\text{for all } s \in \mathcal{S} \text{ and all } \mathfrak{a} \in \mathsf{Act}\colon \quad |\mathsf{Succs}_{\mathfrak{a}}(s)| \in \{0,1\}\,.$$

This terminology is justified by the fact that, in case the above property holds, $\mathcal{M}$ can indeed be identified with the transition system

$$\mathsf{TS}_{\mathcal{M}} \;=\; (\mathcal{S}, \longrightarrow, \{s_I\})\,, \quad \text{where} \quad s \longrightarrow s' \text{ iff exists } \mathfrak{a} \in \mathsf{Act}\colon s' \in \mathsf{Succs}_{\mathfrak{a}}(s)\,.$$

With this notion at hand, it becomes evident that the verification problem tackled by PrIC3 generalizes IC3's objective: We have

$$\text{MDP } \mathcal{M} \text{ is safe} \qquad \text{iff} \qquad \text{transition system } \mathsf{TS}_{\mathcal{M}} \text{ is safe}$$

or, put more symbolically,

$$\mathsf{MaxPr}(s \models \Diamond B) \leq 0 \qquad \text{iff} \qquad s_I \not\in \Diamond B \text{ holds for } \mathsf{TS}_{\mathcal{M}}\,.$$

In the aforementioned recovery statements, we will speak of the *qualitative setting*, referring to the situation where the given MDP $\mathcal{M}$ is a transition system and the threshold probability λ is 0. We moreover call a frame $\mathcal{F}$ *qualitative*, if $\mathcal{F}(s) \in \{0,1\}$ for all $s \in \mathcal{S}$, and we denote by Ψ the *qualitative* Bellman operator of $\mathsf{TS}_{\mathcal{M}}$ as introduced in Section 6.1. Recall that in Section 6.1, we often identified

[10] Notice furthermore that every transition system can be identified with an MDP where all transition probabilities are either 0 or 1.

sets of states with their indicator function, which we adopt here: Given a qualitative frame $\mathcal{F}$, we consider $\Psi(\mathcal{F})$ a well-defined application of Ψ to the set of states described by $\mathcal{F}$. This yields our first recovery statement:

Recovery Statement 1.
In the qualitative setting, we have for all qualitative frames $\mathcal{F}$,

$$\Phi(\mathcal{F}) = \Psi(\mathcal{F}).$$

6.2.3 Challenges

Our motivation is to investigate *whether and how* IC3 *for transition systems can be generalized to reason about probabilistic safety of MDPs*. In this section, we address the challenges of answering this question. Throughout, we will refer to these challenges to justify the design choices we made.

Challenge 1 (Leaving the Boolean Domain).
IC3 iteratively computes *frames*, which are overapproximations of sets of states that reach B in a bounded number of steps. For MDPs, Boolean reachability becomes a *quantitative reachability probability*. This requires a shift: frames become real-valued functions rather than sets of states. Thus, there are infinitely many possible frames — even for finite-state MDPs — just as for *infinite*-state software [CGMT16; LNNK20a; Lan18] and hybrid systems [SI20]. Moreover, if in a given TS a state reaches B within k steps, this fact remains true when increasing k. The step-bounded reachability probabilities in MDPs, on the other hand, may increase for increasing step bounds k. This complicates ensuring the termination of an IC3-style algorithm for MDPs.

Challenge 2 (Single Paths as Counterexamples are Insufficient).
For TSs, a single finite execution fragment[a] from the initial to a bad state suffices to refute safety. This is not true in the probabilistic setting [HKD09]. Instead, proving that the maximal probability of reaching B exceeds the threshold λ possibly requires *a set of possibly cyclic paths* — e.g., represented as a subMDP [CV10] — whose probability mass exceeds λ.

[a] In [HB12], tree-like counterexamples are used for non-linear predicate transformers in IC3.

Example 6.4.
For our running example from Figure 6.10 on page 222, the most probable

path from s_I to B has probability $1/4$ whereas we have $\mathsf{MaxPr}(s_I \models \Diamond B) = 2/3$. Hence, refuting safety for any λ with

$$1/4 \;<\; \lambda \;<\; 2/3$$

requires to consider more than just a single path from s_I to B.

Challenge 3 (Strengthening).
Recall from Section 6.1.3 that IC3 turns a proof obligation of type (i) "state s does not reach B" into type (ii) "the successors of s do not reach B". In the probabilistic setting, obligations of type (i) read "s reaches B with at most probability δ". However, the strengthened type (ii) obligation must then read: "*the weighted sum over the reachability probabilities of the successors of s is at most* δ". In general, there are infinitely many possible choices of obligations for the successors of s in order to satisfy the original obligation because — grossly simplified — there are infinitely many possibilities for α and β to satisfy weighted sums such as

$$\tfrac{1}{3} \cdot \alpha + \tfrac{2}{3} \cdot \beta \;\leq\; \delta\,.$$

While we only need one choice of probabilities, picking a *good* one is approximately as hard as solving the entire problem altogether. We hence require a *heuristic*, which is guided by a *user-provided oracle*.

Challenge 4 (Generalization).
"One of the key components of IC3 is [inductive] generalization" [Bra11b]. Recall from Section 6.1.4 that generalization abstracts single states. It makes IC3 scale but is *not* essential for soundness or termination. To facilitate generalization, systems should be encoded symbolically via, e.g., propositional formulae. IC3's generalization guesses *sets of states*. We, however, need to guess this set *and* a probability for each state. To be effective, these guesses should moreover eventually yield an inductive invariant, which is often highly nonlinear. We proceed as in Section 6.1: We first present PrIC3 for MDPs *without* generalization in a *non*-symbolic manner to understand its fundamentals. Then, in Section 6.2.7, we marry PrIC3 with the weakest preexpectation calculus, yielding a more *symbolic* variant of PrIC3 operating on probabilistic programs, which can be thought of as symbolic representations of MDPs. We then provide first ideas on how to exploit such symbolic encodings to obtain suitable generalizations.

6.2.4 The Core PrIC3 Algorithm

In this section, we explain the rationale underlying PrIC3. Moreover, we describe the core of PrIC3 — called PrIC3$_{\mathcal{H}}$— which bears a close resemblance to the main loop of IC3 for transition systems (Algorithm 7 on page 206).

6.2.4.1 Inductive Frames

Analogously to IC3, the rationale of PrIC3 is to find a frame $\mathcal{F} \in \mathbb{V}_{\leq 1}$ such that (i) $\mathcal{F}$ postulates that the maximal probability of s_I to reach B is at most the threshold λ and (ii) applying the Bellman operator Φ to $\mathcal{F}$ does not take us up in the partial order on frames, i.e.,

$$\text{(i)} \quad \mathcal{F}(s_I) \leq \lambda \qquad \text{and} \qquad \text{(ii)} \quad \Phi(\mathcal{F}) \sqsubseteq \mathcal{F} \,.$$

As with IC3, we call frames satisfying the above conditions *inductive invariants*. By *Park induction* (Lemma 2.4 on page 28), which in our setting reads

$$\Phi(\mathcal{F}) \sqsubseteq \mathcal{F} \quad \text{implies} \quad \mathsf{lfp}\,\Phi \;=\; \lambda s.\,\mathsf{MaxPr}(s \models \Diamond B) \;\sqsubseteq\; \mathcal{F}\,,$$

an inductive invariant $\mathcal{F}$ would indeed *witness* that $\mathcal{M}$ is safe because

$$\mathsf{MaxPr}(s_I \models \Diamond B) \;=\; \big(\mathsf{lfp}\,\Phi\big)(s_I) \;\leq\; \mathcal{F}(s_I) \;\leq\; \lambda\,.$$

Such an inductive invariant exists iff $\mathcal{M}$ is safe, which can be seen by taking $\mathcal{F} = \mathsf{lfp}\,\Phi$. If no inductive invariant exists, then IC3 finds a counterexample: a *finite execution fragment* from the initial state s_I to a bad state in B, which serves as a witness to refute. Analogously, PrIC3 will find a counterexample, but of a different kind: Since single paths are insufficient as counterexamples in the probabilistic realm (Challenge 2), PrIC3 will instead find a *subsystem* of $\mathcal{M}$ witnessing $\mathsf{MaxPr}(s_I \models \Diamond B) > \lambda$, which we will define formally in Section 6.2.6.1.

6.2.4.2 The PrIC3 Invariants

Analogously to IC3, PrIC3 aims to find the inductive invariant by maintaining a *sequence of frames* $\mathcal{F}_0 \sqsubseteq \mathcal{F}_1 \sqsubseteq \mathcal{F}_2 \sqsubseteq \ldots$ such that $\mathcal{F}_i(s)$ overapproximates the maximal probability of reaching B from s within *at most i steps*. This *i-step-bounded maximal reachability probability* $\mathsf{MaxPr}\big(s \models \Diamond^{\leq i} B\big)$ can be characterized using the Bellman operator: $\Phi(0)$ is[11] the 0-step probability; it is 1 for every

[11]We slightly abuse notation and often denote by 0 (resp. 1) the constant frame $\lambda s.\,0$ (resp. $\lambda s.\,1$).

$s \in B$ and 0 otherwise. For any $i \geq 0$, we have

$$\mathsf{MaxPr}\left(s \models \Diamond^{\leq i} B\right) = \Phi^i(\Phi(0))(s) = \Phi^{i+1}(0)(s)\,.$$

The *unbounded* reachability probability is given by

$$\begin{aligned}
&\mathsf{MaxPr}\,(s \models \Diamond B) \\
=\;& \left(\mathsf{lfp}\ \Phi\right)(s) \\
\overset{(*)}{=}\;& \sup\left\{\Phi^n(0)(s) \mid n \in \mathbb{N}\right\} \\
=\;& \sup\left\{\mathsf{MaxPr}\left(s \models \Diamond^{\leq n} B\right) \mid n \in \mathbb{N}\right\},
\end{aligned}$$

where $(*)$ is a consequence of Kleene's Fixpoint Theorem 2.2.

The sequence $\mathcal{F}_0 \sqsubseteq \mathcal{F}_1 \sqsubseteq \mathcal{F}_2 \sqsubseteq \ldots$ maintained by PrIC3 should frame-wise overapproximate the sequence $\Phi(0) \sqsubseteq \Phi^2(0) \sqsubseteq \Phi^3(0) \ldots$. Pictorially:

$$\begin{array}{ccccccccc}
 & & \mathcal{F}_0 & \sqsubseteq & \mathcal{F}_1 & \sqsubseteq & \mathcal{F}_2 & \sqsubseteq & \ldots \\
 & & \sqcup\!\!\!\!\!\!\mid & & \sqcup\!\!\!\!\!\!\mid & & \sqcup\!\!\!\!\!\!\mid & & \\
0 & \sqsubseteq & \Phi(0) & \sqsubseteq & \Phi^2(0) & \sqsubseteq & \Phi^3(0) & \sqsubseteq & \ldots
\end{array}$$

However, the sequence $\Phi(0), \Phi^2(0), \Phi^3(0), \ldots$ will never explicitly be known to PrIC3. Instead, PrIC3 will ensure the above frame-wise overapproximation property implicitly by enforcing the so-called PrIC3 *invariants* on the frame sequence $\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2, \ldots$. Apart from allowing for a threshold $0 \leq \lambda \leq 1$ on the maximal reachability probability, these invariants are analogous to the IC3 invariants from Definition 6.1 on page 205 (where $\lambda = 0$ is fixed). Formally:

Definition 6.4 (PrIC3 Invariants).
Let $k \geq 0$. We say that the frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ satisfy the PrIC3 *invariants*, a fact we denote by $\mathsf{PrIC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$, if all of the following conditions hold:

1. ***Initiality:*** $\mathcal{F}_0 = \Phi(0)$
2. ***Chain Property:*** for all $0 \leq i < k$: $\mathcal{F}_i \sqsubseteq \mathcal{F}_{i+1}$
3. ***Frame-Safety:*** for all $0 \leq i < k$: $\mathcal{F}_i(s_I) \leq \lambda$
4. ***Relative Inductivity:*** for all $0 \leq i < k$: $\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}_{i+1}$

Notice that $\mathcal{F}_k$ must *not necessarily* satisfy $\mathcal{F}_k(s_I) \leq \lambda$. The PrIC3 invariants enforce the above picture: The *chain property* ensures

$$\mathcal{F}_0 \sqsubseteq \mathcal{F}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{F}_k\,.$$

Moreover, we have $\Phi(0) = \mathcal{F}_0 \sqsubseteq \mathcal{F}_0$ by *initiality*. Assuming $\Phi^{i+1}(0) \sqsubseteq \mathcal{F}_i$ as induction hypothesis, monotonicity of Φ and *relative inductivity* imply

$$\Phi^{i+2}(0) \sqsubseteq \Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}_{i+1} .$$

Now, by overapproximating $\Phi(0), \Phi^2(0), \ldots, \Phi^{k+1}(0)$, the frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$ in effect bound the maximal step-bounded reachability probability of every state:

Lemma 6.4 (PrIC3 Invariants yield Overapproximations).
Let $\mathcal{F}_0,\ldots,\mathcal{F}_k$ be frames satisfying the PrIC3 invariants. Then

$$\text{for all } s \in \mathcal{S} \text{ and all } i \leq k\colon \quad \mathsf{MaxPr}\left(s \models \Diamond^{\leq i} B\right) \leq \mathcal{F}_i(s) .$$

In particular, a sequence $\mathcal{F}_0,\ldots,\mathcal{F}_k$ of frames satisfying the PrIC3 invariants witnesses that the maximal probability to reach B from s_I within $i \in \{0,\ldots,k-1\}$ steps is at most λ by Lemma 6.4 and frame-safety.

Given two frames $\mathcal{F}, \mathcal{F}'$, we say that $\mathcal{F}'$ is a *strengthening* of $\mathcal{F}$, if $\mathcal{F}' \sqsubseteq \mathcal{F}$. We say that a sequence $\mathcal{F}_0',\ldots,\mathcal{F}_k'$ of frames is a strengthening of $\mathcal{F}_0,\ldots,\mathcal{F}_k$, if $\mathcal{F}_i'$ is a strengthening of $\mathcal{F}_i$ for all $i \in \{0,\ldots,k\}$.

Lemma 6.5 (Step-Bounded Probabilistic Safety via Strengthenings).
Let $\mathcal{F}_0,\ldots,\mathcal{F}_k$ be frames satisfying the PrIC3 invariants. There is a strengthening $\mathcal{F}_0',\ldots,\mathcal{F}_k'$ of $\mathcal{F}_0,\ldots,\mathcal{F}_k$ satisfying

1. $\mathsf{PrIC3Inv}\left(\mathcal{F}_0',\ldots,\mathcal{F}_k'\right)$, and
2. $\mathcal{F}_k'(s_I) \leq \lambda$

if and only if $\mathsf{MaxPr}\left(s_I \models \Diamond^{\leq k} B\right) \leq \lambda$.

As for proving that the *unbounded* reachability probability is also at most λ, it suffices to find two consecutive frames, say $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$, that coincide:

Lemma 6.6 (Inductive Invariants from PrIC3 Invariants).
Let $\mathcal{F}_0,\ldots,\mathcal{F}_k$ be frames satisfying the PrIC3 invariants. Then

$$\text{exists } i < k\colon \mathcal{F}_i = \mathcal{F}_{i+1} \qquad \text{implies} \qquad \mathsf{MaxPr}(s_I \models \Diamond B) \leq \lambda .$$

Proof. $\mathcal{F}_i = \mathcal{F}_{i+1}$ and relative inductivity yield

$$\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}_{i+1} = \mathcal{F}_i .$$

With frame-safety, i.e., $\mathcal{F}_i(s_I) \leq \lambda$, this renders $\mathcal{F}_i$ an inductive invariant. ∎

Algorithm 9: $\mathsf{PrIC3}_{\mathcal{H}}(\mathcal{M}, s_I, B, \lambda)$

```
input: MDP $\mathcal{M}$, initial state $s_I \in \mathcal{S}$, set of bad states $B$ with $s_I \notin B$,
       threshold $\lambda \in [0,1] \cap \mathbb{Q}$
output: true or (false and a subset of the states of $\mathcal{M}$)
1 $\mathcal{F}_0 \leftarrow \Phi(0)$;   $\mathcal{F}_1 \leftarrow 1$;   $k \leftarrow 1$;   $oldSubsystem \leftarrow \emptyset$
2 while true do
3     $success, \mathcal{F}_0, \ldots, \mathcal{F}_k, subsystem \leftarrow \mathsf{Strengthen}_{\mathcal{H}}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$
4     if $\neg success$ then return false, $subsystem$
5     $\mathcal{F}_{k+1} \leftarrow 1$
6     $\mathcal{F}_0, \ldots, \mathcal{F}_{k+1} \leftarrow \mathsf{Propagate}(\mathcal{F}_0, \ldots, \mathcal{F}_{k+1})$
7     if $\exists 1 \leq i \leq k \colon \mathcal{F}_i = \mathcal{F}_{i+1}$ then return true
8     if $oldSubsystem = subsystem$ then return false, $subsystem$
9     $k \leftarrow k+1$;   $oldSubsystem \leftarrow subsystem$
```

Figure 6.11: $\mathsf{PrIC3}_{\mathcal{H}}$ parameterized by a heuristic $\mathcal{H} \colon \mathcal{S} \times \mathsf{Act} \times [0,1] \to [0,1]^*$.

Recovery Statement 2.
In the qualitative setting, we have for all qualitative frames $\mathcal{F}_0, \ldots, \mathcal{F}_k$,

$$\mathsf{PrIC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k) \qquad \text{iff} \qquad \mathsf{IC3Inv}(\mathcal{F}_0, \ldots, \mathcal{F}_k)\ .$$

6

6.2.4.3 Operationalizing the PrIC3 Invariants for Proving Safety

Lemma 6.6 gives us a clear angle of attack for proving an MDP safe: Repeatedly add and strengthen frames overapproximating maximal step-bounded reachability probabilities for more and more steps by preserving the PrIC3 invariants until two consecutive frames coincide.

Analogously to IC3, this approach is taken by the core loop $\mathsf{PrIC3}_{\mathcal{H}}$ depicted in Algorithm 9; differences to the main loop of IC3 (Algorithm 7 on page 206) are highlighted in red. A particular difference is that $\mathsf{PrIC3}_{\mathcal{H}}$ is parameterized by a heuristic $\mathcal{H}$ for finding suitable probabilities (see Challenge 3). Since the precise choice of $\mathcal{H}$ is irrelevant to the soundness of $\mathsf{PrIC3}_{\mathcal{H}}$, we defer a detailed discussion of suitable heuristics to Section 6.2.7.3.

As input, $\mathsf{PrIC3}_{\mathcal{H}}$ takes an MDP $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$, an initial state $s_I \in \mathcal{S}$, a set

$B \subseteq \mathcal{S}$ of bad states with[12] $s_I \notin B$, and a threshold $\lambda \in [0,1] \cap \mathbb{Q}$. Since the input is never changed, we assume it to be *globally available*, also to subroutines. As output, $\mathsf{PrIC3}_{\mathcal{H}}$ returns true if two consecutive frames become equal. We hence say that $\mathsf{PrIC3}_{\mathcal{H}}$ is *sound*, if it returns true only if $\mathcal{M}$ is safe, i.e., if

$$\mathsf{PrIC3}_{\mathcal{H}}(\mathcal{M}, s_I, B, \lambda) \;=\; \mathsf{true} \qquad \text{implies} \qquad \mathsf{MaxPr}(s_I \models \lozenge B) \;\leq\; \lambda\,.$$

Let us briefly go through the individual steps of $\mathsf{PrIC3}_{\mathcal{H}}$ and convince ourselves that it is indeed sound. After that, we discuss why $\mathsf{PrIC3}_{\mathcal{H}}$ terminates and what happens if it is unable to prove safety by finding two equal consecutive frames.

How $\mathsf{PrIC3}_{\mathcal{H}}$ *works.* The sequence of frames $\mathcal{F}_0,\ldots,\mathcal{F}_k$ maintained by $\mathsf{PrIC3}_{\mathcal{H}}$ is initialized in l. 1 with $k = 1$, $\mathcal{F}_0 = \Phi(0)$, and $\mathcal{F}_1 = 1$, which implies $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$. The **while**-loop in l. 2 maintains $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$ at all times. In l. 3, procedure $\mathsf{Strengthen}_{\mathcal{H}}$ — detailed in Section 6.2.5 — is called to determine whether $\mathsf{MaxPr}\big(s_I \models \lozenge^{\leq k} B\big) \leq \lambda$. By Lemma 6.5, this is equivalent to determining whether the frames can be strengthened while preserving the PrIC3 invariants and such that $\mathcal{F}_k(s_I) \leq \lambda$. $\mathsf{Strengthen}_{\mathcal{H}}$ either returns true if successful or returns false and a subsystem of $\mathcal{M}$ if it was unable to do so.

Definition 6.5 (Specification of $\mathsf{Strengthen}_{\mathcal{H}}$).
Procedure $\mathsf{Strengthen}_{\mathcal{H}}$ is *sound*, if for all $k \geq 1$ and all sequences $\mathcal{F}_0,\ldots,\mathcal{F}_k$ of frames with $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$, we have

$$\begin{aligned} & \mathsf{Strengthen}_{\mathcal{H}}(\mathcal{F}_0,\ldots,\mathcal{F}_k) = \mathsf{true}, \mathcal{F}_0',\ldots,\mathcal{F}_k', \_\_ \\ \text{implies}\quad & \mathsf{PrIC3Inv}\big(\mathcal{F}_0',\ldots,\mathcal{F}_k'\big) \text{ and } \mathcal{F}_k'(s_I) \leq \lambda\,. \end{aligned}$$

If $\mathsf{Strengthen}_{\mathcal{H}}$ returns false, then it failed to strengthen the frames accordingly and $\mathsf{PrIC3}_{\mathcal{H}}$ terminates returning false (l. 4). In contrast to IC3, *this does not necessarily mean that $\mathcal{M}$ is unsafe*. Returning false (also possible in l. 8) has by specification no effect on the soundness of $\mathsf{PrIC3}_{\mathcal{H}}$.

If $\mathsf{Strengthen}_{\mathcal{H}}$ returns true, then a new frame $\mathcal{F}_{k+1} = 1$ is created in l. 5 and the extended sequence $\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$ of frames again satisfies the PrIC3 invariants. *Propagation* (l. 6) aims to speed up termination by updating $\mathcal{F}_{i+1}(s)$ by $\mathcal{F}_i(s)$ if this does not violate the PrIC3 invariants. Consequently, the previously mentioned properties remain unchanged. We then check whether there exist two identical consecutive frames (l. 7). If so, Lemma 6.6 yields that the MDP $\mathcal{M}$ is safe; consequently, $\mathsf{PrIC3}_{\mathcal{H}}$ returns true. Otherwise, we increment k and are in the

[12]This assumption simplifies the presentation and can be easily checked upfront.

same setting as upon entering the loop, now with an extended frame sequence; $\mathsf{PrIC3}_{\mathcal{H}}$ then performs another iteration. Hence:

Theorem 6.7 (Soundness of $\mathsf{PrIC3}_{\mathcal{H}}$).
If $\mathsf{Strengthen}_{\mathcal{H}}$ is sound and Propagate preserves the PrIC3 invariants, then $\mathsf{PrIC3}_{\mathcal{H}}$ is sound, i.e.,

$$\mathsf{PrIC3}_{\mathcal{H}}(\mathcal{M}, s_I, B, \lambda) \;=\; \mathsf{true} \qquad \text{implies} \qquad \mathsf{MaxPr}(s_I \models \lozenge B) \;\leq\; \lambda\,.$$

Next, we discuss why $\mathsf{PrIC3}_{\mathcal{H}}$ terminates by distinguishing the cases where $\mathcal{M}$ is safe and where it is unsafe. We moreover assume that $\mathsf{Strengthen}_{\mathcal{H}}$ and Propagate terminate, which we justify in Section 6.2.5.

$\mathsf{PrIC3}_{\mathcal{H}}$ *terminates for unsafe MDPs.* If $\mathcal{M}$ is unsafe, then there exists a step-bound $n \in \mathbb{N}$ witnessing this fact, i.e., there exists some $n \in \mathbb{N}$ with

$$\mathsf{MaxPr}\big(s_I \models \lozenge^{\leq n} B\big) > \lambda\,.$$

Any sound implementation of $\mathsf{Strengthen}_{\mathcal{H}}$ (cf. Definition 6.5) either immediately terminates $\mathsf{PrIC3}_{\mathcal{H}}$ by returning false or keeps returning strengthened frames. If the former case never arises, then $\mathsf{Strengthen}_{\mathcal{H}}$ will eventually return true on a sequence of frames $\mathcal{F}_0, \ldots, \mathcal{F}_{n+1}$ of length $n+2$. By Lemma 6.4, we then have $\mathcal{F}_n(s_I) \geq \mathsf{MaxPr}\big(s_I \models \lozenge^{\leq n} B\big) > \lambda$, contradicting frame-safety.

$\mathsf{PrIC3}_{\mathcal{H}}$ *terminates for safe MDPs.* IC3 terminates on safe finite TSs as there are only finitely many different frames, making every ascending chain of frames eventually stabilize. For us, frames map states to probabilities (Challenge 1), yielding *infinitely many possible frames* even for finite-state MDPs. In particular, the complete lattice $(\mathbb{V}_{\leq 1}, \sqsubseteq)$ of frames does *not* have finite height. Hence, $\mathsf{Strengthen}_{\mathcal{H}}$ need not ever yield a stabilizing chain of frames. In this case, $\mathsf{Strengthen}_{\mathcal{H}}$ continuously reasons about the same set of states and we give up. $\mathsf{PrIC3}_{\mathcal{H}}$ checks this by comparing the subsystem $\mathsf{Strengthen}_{\mathcal{H}}$ operates on with the one it operated on in the previous loop iteration (l. 8). In summary:

Theorem 6.8 (Termination of $\mathsf{PrIC3}_{\mathcal{H}}$).
If $\mathsf{Strengthen}_{\mathcal{H}}$ and Propagate terminate, then $\mathsf{PrIC3}_{\mathcal{H}}$ terminates.

$\mathsf{PrIC3}_{\mathcal{H}}$ *is incomplete.* If $\mathsf{Strengthen}_{\mathcal{H}}$ returns false, $\mathsf{PrIC3}_{\mathcal{H}}$ returns a *subsystem* of the MDP $\mathcal{M}$ provided by $\mathsf{Strengthen}_{\mathcal{H}}$. However, $\mathsf{Strengthen}_{\mathcal{H}}$ cannot detect whether $\mathcal{M}$ is indeed unsafe. Reporting false thus only means that the given MDP *may* be unsafe; the returned subsystem has to be analyzed further. The full PrIC3 algorithm presented in Section 6.2.6 addresses this issue.

It.	1	2		3			4				5				
$\mathcal{F}_i$	$\mathcal{F}_1$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_5$
s_I	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9	5/9
s_1	1	11/18	1	11/18	11/18	1	11/18	11/18	11/18	1	11/18	11/18	11/18	11/18	1
s_2	1	1/2	1	1/2	1/2	1	1/2	1/2	1/2	1	1/2	1/2	1/2	1/2	1
s_3	1	1	1	2/3	1	1	2/3	2/3	1	1	2/3	2/3	2/3	1	1
s_4	1	1	1	1	1	1	0	1	1	1	0	0	1	1	1
s_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a) Threshold $\lambda = 5/9$.

It.	1	2		3			4			
$\mathcal{F}_i$	$\mathcal{F}_1$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$
s_I	9/10	9/10	9/10	9/10	9/10	9/10	9/10	9/10	9/10	9/10
s_1	1	99/100	1	99/100	99/100	1	99/100	99/100	99/100	1
s_2	1	81/100	1	81/100	81/100	1	81/100	81/100	81/100	1
s_3	1	1	1	1	1	1	1	1	1	1
s_4	1	1	1	0	1	1	0	0	1	1
s_5	1	1	1	1	1	1	1	1	1	1

(b) Threshold $\lambda = 9/10$.

Figure 6.12: Two runs of $\mathsf{PrIC3}_{\mathcal{H}}$ on the Markov chain induced by selecting action $\mathfrak{a}$ in the MDP from Figure 6.10 on page 222. For every iteration, frames are recorded after invocation of $\mathsf{Strengthen}_{\mathcal{H}}$.

Example 6.5.
We consider two example executions of $\mathsf{PrIC3}_{\mathcal{H}}$ on a simplified version of the MDP from Figure 6.10 on page 222, where $\mathfrak{b}$ has been removed. For every state, exactly one action is enabled, i.e., we consider a Markov chain. Figure 6.12 depicts the frame sequences computed by $\mathsf{PrIC3}_{\mathcal{H}}$ (for a reasonable $\mathcal{H}$) on that Markov chain for two safe thresholds:

$$5/9 \;=\; \mathsf{MaxPr}(s_I \models \Diamond B) \qquad \text{and} \qquad 9/10\,.$$

In particular, notice that *proving the coarser bound of* 9/10 *requires fewer frames than proving the exact bound of* 5/9. See Example 6.6 on page 235 for more details on the run of $\mathsf{PrIC3}_{\mathcal{H}}$ for the threshold 5/9.

6.2.5 Strengthening in PrIC3

When the loop of $\mathsf{PrIC3}_{\mathcal{H}}$ invokes Strengthen, we have $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$ and $\mathcal{F}_k(s_I) = 1$. The task of $\mathsf{Strengthen}_{\mathcal{H}}$ is to determine whether it holds that

$$\mathsf{MaxPr}\left(s_I \models \Diamond^{\leq k} B\right) \;\leq\; \lambda\,.$$

By Lemma 6.5, this is equivalent to determining whether the frames can be strengthened while preserving the PrIC3 invariants and such that $\mathcal{F}_k(s_I) \leq \lambda$.

Analogously to IC3, our first *obligation* is to lower the value in frame $i = k$ for state $s = s_I$ to $\delta = \lambda \in [0,1]$. We denote such an obligation by (i,s,δ). Notice that obligations in PrIC3 have an additional entry δ. In the *qualitative* setting, this entry is always equal to 0. An obligation (i,s,δ) requires us to prove that

$$\mathsf{MaxPr}\left(s \models \Diamond^{\leq i} B\right) \;\leq\; \delta\,, \tag{6.2}$$

and it is *resolved* by updating the values assigned to state s in *all frames* $\mathcal{F}_1,\ldots,\mathcal{F}_i$ to at most δ without violating the PrIC3 invariants. That is, for all $j \leq i$, we set $\mathcal{F}_j(s)$ to the minimum of δ and the original value $\mathcal{F}_j(s)$. Since such an update may violate relative inductivity, i.e., $\Phi(\mathcal{F}_{i-1}) \sqsubseteq \mathcal{F}_i$, we thus first check whether $\mathcal{F}_{i-1}$ is strong enough to support (6.2), i.e., if

$$\Phi(\mathcal{F}_{i-1})(s) \;\leq\; \delta \quad \text{which is the case iff} \quad \text{for all } \mathfrak{a} \in \mathsf{Act}(s)\colon\ \Phi_{\mathfrak{a}}(\mathcal{F}_{i-1})(s) \;\leq\; \delta\,.$$

If so, (i,s,δ) can be resolved without violating the PrIC3 invariants. If not, there is some action $\mathfrak{a} \in \mathsf{Act}(s)$ with

$$\Phi_{\mathfrak{a}}(\mathcal{F}_{i-1})(s) \;>\; \delta\,,$$

Algorithm 10: $\mathsf{Strengthen}_{\mathcal{H}}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$

```
input: Frames $\mathcal{F}_0,\ldots,\mathcal{F}_k$ with $k \geq 1$ and $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$
output: See Definition 6.5 on page 230
1  $Q \leftarrow \{(k, s_I, \lambda)\}$
2  while $Q$ not empty do
3      $(i, s, \delta) \leftarrow Q.\mathrm{popMin}()$  /* pop obligation with minimal frame index */
4      if $i = 0 \quad \vee \quad (s \in B \wedge \delta < 1)$ then
           /* possible counterexample given by subsystem consisting of
              states popped from $Q$ at some point */
5          return false, ___, $Q$.touched();
       /* check whether $\mathcal{F}_i(s) \leftarrow \delta$ violates relative inductivity */
6      if $\exists \mathfrak{a} \in \mathsf{Act}(s)$: $\Phi_{\mathfrak{a}}(\mathcal{F}_{i-1})(s) > \delta$ then for such an $\mathfrak{a}$
7          $\{s_1,\ldots,s_n\} \leftarrow \mathsf{Succs}_{\mathfrak{a}}(s)$
8          $\delta_1,\ldots,\delta_n \leftarrow \mathcal{H}(s,\mathfrak{a},\delta)$
9          $Q.\mathrm{push}((i-1, s_1, \delta_1),\ldots,(i-1, s_n, \delta_n),(i, s, \delta))$
10     else /* resolve $(i, s, \delta)$ without violating relative inductivity */
11         $\mathcal{F}_1(s) \leftarrow \min(\mathcal{F}_1(s), \delta);\ldots;\mathcal{F}_i(s) \leftarrow \min(\mathcal{F}_i(s), \delta)$
   /* $Q$ empty; all obligations have been resolved */
12 return true, $\mathcal{F}_0,\ldots,\mathcal{F}_k$, $Q$.touched()
```

Figure 6.13: Strengthening in PrIC3. Here $\mathcal{H}$ is a fixed heuristic of type $\mathcal{S} \times \mathsf{Act} \times [0,1] \to [0,1]^*$ for selecting probabilities and Q.touched() is the set of states that eventually appeared in Q.

and we have to spawn further obligations for the states in $\mathsf{Succs}_{\mathfrak{a}}(s)$ to decrease the entries for these states in $\mathcal{F}_{i-1}$ before being able to resolve (i, s, δ).

$\mathsf{Strengthen}_{\mathcal{H}}$ *by example.* $\mathsf{Strengthen}_{\mathcal{H}}$ is given by the pseudo code in Algorithm 10; differences to IC3's strengthening (Algorithm 8 on page 208) are highlighted in red. Intuitively, $\mathsf{Strengthen}_{\mathcal{H}}$ attempts to recursively resolve all obligations while preserving the PrIC3 invariants or it detects a *potential counterexample* justifying why it is unable to do so. We first consider an execution where the latter does not arise:

Example 6.6.
We zoom in on Example 6.5: Prior to the second iteration, we have created the following three frames:

$$\mathcal{F}_0 = (0,0,0,0,0,1), \qquad \mathcal{F}_1 = (5/9,1,1,1,1,1), \qquad \text{and} \qquad \mathcal{F}_2 = 1 .$$

To keep track of unresolved obligations (i,s,δ), $\mathsf{Strengthen}_{\mathcal{H}}$ employs a priority queue Q which pops obligations with minimal frame index i first. Our first step is to ensure frame-safety of $\mathcal{F}_2$, i.e., alter $\mathcal{F}_2$ so that $\mathcal{F}_2(s_I) \leq 5/9$; we thus initialize the queue Q with the initial obligation $(2,s_I,5/9)$ (l. 1). To do so, we check whether updating $\mathcal{F}_2(s_I)$ to $5/9$ would violate relative inductivity (l. 6). This is indeed the case:

$$\Phi(\mathcal{F}_1)(s_I) = 1/2 \cdot \mathcal{F}_1(s_1) + 1/2 \cdot \mathcal{F}_1(s_2) = 1 \not\leq 5/9 .$$

$\mathsf{Strengthen}_{\mathcal{H}}$ thus spawns one new obligation for each relevant successor of s_I. These have to be resolved before retrying to resolve the old obligation.[a] *In contrast to standard* IC3, *spawning obligations involves finding suitable probabilities* δ (l. 8). In our example, we have to spawn two obligations

$$(1,s_1,\delta_1) \text{ and } (1,s_2,\delta_2) \qquad \text{such that} \qquad 1/2 \cdot \delta_1 + 1/2 \cdot \delta_2 \leq 5/9 .$$

There are *infinitely many choices* for δ_1 and δ_2 satisfying this inequality. Assume some heuristic $\mathcal{H}$ chooses $\delta_1 = 11/18$ and $\delta_2 = 1/2$; we push obligations $(1,s_1,11/18)$, $(1,s_2,1/2)$, and $(2,s_I,5/9)$ (ll. 7, 9). In the next iteration, we first pop obligation $(1,s_1,11/18)$ (l. 3) and find that it can be resolved without violating relative inductivity (l. 6). Hence, we set $\mathcal{F}_1(s_1)$ to $11/18$ (l. 11); no new obligation is spawned. Obligation $(1,s_2,1/2)$ is resolved analogously; the updated frame is

$$\mathcal{F}_1 = (5/9, 11/18, 1/2, 1, 1, 1) .$$

Thereafter, our initial obligation $(2,s_I,5/9)$ can be resolved; we have $\mathsf{PrIC3Inv}(\mathcal{F}_0,\mathcal{F}_1,\mathcal{F}_2)$ and $\mathcal{F}_2(s_I) \leq \lambda$. Hence, $\mathsf{Strengthen}_{\mathcal{H}}$ returns true, the updated frames, and the set Q.touched() that eventually appeared in Q.

[a] We assume that the set $\mathrm{Succs}_{\mathfrak{a}}(s)$ of $\mathfrak{a}$-*successors* of state s is given in some fixed order.

$\mathsf{Strengthen}_{\mathcal{H}}$ *is sound.* Let us discuss why Algorithm 10 meets the specification from Definition 6.5: First, we observe that Algorithm 10 alters the frames — and thus potentially invalidates the PrIC3 invariants — only in l. 11 by resolving an obligation (i,s,δ) with $\Phi(F_{i-1})(s) \leq \delta$ (due to the check in l. 6). Resolving obligation (i,s,δ) in l. 11 lowers the values assigned to state s to at most δ *without* violating the PrIC3 invariants, which can be seen by taking

$$\mathcal{F} \;=\; \lambda s'.\begin{cases}\delta & \text{if } s' = s\\ 1 & \text{otherwise}\end{cases}$$

in the following more general theorem:

Theorem 6.9 (Preservation of the PrIC3 Invariants).
Let $k \geq 1$ and let $\mathcal{F}_0,\ldots,\mathcal{F}_k$ be frames with $\mathsf{PrIC3Inv}(\mathcal{F}_0,\ldots,\mathcal{F}_k)$. Moreover, let $i \in \{1,\ldots,k\}$ and let $\mathcal{F}$ be a frame with $\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}$. Then the strengthened sequence of frames $\mathcal{F}_0',\ldots,\mathcal{F}_k'$ of frames given by

$$\mathcal{F}_j' \;=\; \begin{cases}\mathcal{F}_j \sqcap \mathcal{F} & \text{if } j \in \{1,\ldots,i\}\\ \mathcal{F}_j & \text{otherwise}\end{cases}$$

satisfies the PrIC3 invariants as well, i.e., $\mathsf{PrIC3Inv}\big(\mathcal{F}_0',\ldots,\mathcal{F}_k'\big)$. If moreover $i = k$ and $\mathcal{F}(s_I) \leq \lambda$, then additionally $\mathcal{F}_k'(s_I) \leq \lambda$.

The specification of $\mathsf{Strengthen}_{\mathcal{H}}$ (Definition 6.5 on page 230) guarantees that the PrIC3 invariants hold initially. Since these invariants are preserved by Theorem 6.9, Algorithm 10 is a sound implementation of $\mathsf{Strengthen}_{\mathcal{H}}$ if $\mathcal{F}_k(s_I) \leq \lambda$ whenever it returns true, i.e., once it leaves the loop with an empty obligation queue Q (ll. 11–12). Now, an obligation (i,s,δ) is only popped from Q in l. 3. As (i,s,δ) is added to Q upon reaching l. 9, the size of Q can only ever be reduced (without returning false) by resolving (i,s,δ) in l. 11. Hence, Algorithm 10 does not return true unless it establishes $\mathcal{F}_k(s_I) \leq \lambda$ by resolving, amongst all other obligations, the initial obligation (k, s_I, λ). Consequently:

Lemma 6.10 (Soundness of $\mathsf{Strengthen}_{\mathcal{H}}$).
$\mathsf{Strengthen}_{\mathcal{H}}$ satisfies the specification from Definition 6.5.

Theorem 6.11 (Soundness of $\mathsf{PrIC3}_{\mathcal{H}}$).
Procedure $\mathsf{PrIC3}_{\mathcal{H}}$ satisfies the specification from Theorem 6.7.

Remark 6.1 (On Generalization).
Analogously to IC3 (cf. Section 6.1.4), resolving an obligation in l. 11 may be accompanied by *generalization*. That is, we attempt to update the values of multiple states at once without violating the PrIC3 invariants. Generalization is, however, highly non-trivial in a probabilistic setting. We discuss three

possible approaches to generalization in Section 6.2.7.4.

$\mathsf{Strengthen}_{\mathcal{H}}$ *terminates.* We now show that $\mathsf{Strengthen}_{\mathcal{H}}$ as in Algorithm 10 terminates. The only scenario in which $\mathsf{Strengthen}_{\mathcal{H}}$ may not terminate is if it keeps spawning obligations in l. 9. Let us thus look closer at how obligations are spawned: Whenever we detect that resolving an obligation (i, s, δ) would violate relative inductivity for some action $\mathfrak{a}$ (l. 6), we first need to update the values of the successor states $s_1, \ldots, s_n \in \mathsf{Succs}_{\mathfrak{a}}(s)$ in frame $i-1$, i.e., we push the obligations $(i-1, s_1, \delta_1), \ldots, (i-1, s_n, \delta_n)$ which have to be resolved first (ll. 8–9). It is noteworthy that, in the qualitative setting, i.e., if $\mathcal{M}$ is a transition system, then a single action leads to a single successor state s_1. $\mathsf{Strengthen}_{\mathcal{H}}$ employs a heuristic $\mathcal{H}$ to determine the probabilities required for pushing obligations (l. 8). Assume for an obligation (i, s, δ) that the check in l. 6 yields $\exists \mathfrak{a} \in \mathsf{Act}(s)\colon \Phi_{\mathfrak{a}}(\mathcal{F}_{i-1})(s) > \delta$. Then $\mathcal{H}$ takes s, $\mathfrak{a}$, δ and reports some probability δ_j for every $\mathfrak{a}$-successor s_j of s. However, an arbitrary heuristic of type $\mathcal{H}\colon \mathcal{S} \times \mathsf{Act} \times [0,1] \to [0,1]^*$ may lead to non-terminating behavior: If $\delta_1, \ldots, \delta_n = \mathcal{F}_{i-1}(s_1), \ldots, \mathcal{F}_{i-1}(s_n)$, then the heuristic has no effect. It is thus natural to require that an *adequate* heuristic $\mathcal{H}$ yields probabilities such that the check $\Phi_{\mathfrak{a}}(\mathcal{F}_{i-1})(s) > \delta$ in l. 6 cannot succeed twice for the *same obligation* (i, s, δ) and *same action* $\mathfrak{a}$. Formally, this is guaranteed by the following:

Definition 6.6 (Adequate Heuristics).
We say that heuristic $\mathcal{H}$ is *adequate*, if for all frames $\mathcal{F}$, states $s, s_1, \ldots, s_n$, actions $\mathfrak{a}$ with $\mathsf{Succs}_{\mathfrak{a}}(s) = \{s_1, \ldots, s_n\}$, and probabilities δ, we have:

1. If there exist probabilities $\delta_1, \ldots, \delta_n$ with
$$\Phi_{\mathfrak{a}}(\mathcal{F}[s_1 \mapsto \delta_1] \ldots [s_n \mapsto \delta_n])(s) \;\leq\; \delta\,,$$
then $\mathcal{H}$ returns such probabilities, i.e.,
$$\begin{aligned} & \mathcal{H}(s, \mathfrak{a}, \delta) \;=\; \delta_1', \ldots, \delta_n' \\ \text{implies}\quad & \Phi_{\mathfrak{a}}(\mathcal{F}[s_1 \mapsto \delta_1'] \ldots [s_n \mapsto \delta_n'])(s) \;\leq\; \delta\,. \end{aligned}$$

2. If such probabilities do not exist, then
$$\mathcal{H}(s, \mathfrak{a}, \delta) \;=\; 0, \ldots, 0\,.$$

Details regarding our implementation of heuristic $\mathcal{H}$ are given in Section 6.2.7.3.

For an adequate heuristic, the termination argument is analogous to the argument for IC3's strengthening: Attempting to resolve an obligation (i, s, δ) (ll. 3 – 11) either succeeds after spawning it at most $|\mathsf{Act}(s)|$ times or $\mathsf{Strengthen}_{\mathcal{H}}$

returns false. By a similar argument, attempting to resolve an obligation $(i > 0, s, \_\_)$ leads to at most $\sum_{\mathfrak{a} \in \mathsf{Act}(s)} |\{s' \in \mathcal{S} \mid P(s, \mathfrak{a}, s') > 0\}|$ other obligations of the form $(i{-}1, s', \_\_)$. Consequently, the total number of obligations spawned by Algorithm 10 is bounded since $\mathcal{M}$ is finite-state. Since Algorithm 10 terminates if all obligations have been resolved (l. 11) and each of its loop iterations either returns false, spawns obligations, or resolves an obligation, we conclude:

Lemma 6.12 (Termination of $\mathsf{Strengthen}_{\mathcal{H}}$).
$\mathsf{Strengthen}_{\mathcal{H}}(\mathcal{F}_0, \ldots, \mathcal{F}_k)$ terminates for every adequate heuristic $\mathcal{H}$.

Recovery Statement 3.
Let $\mathcal{H}$ be adequate. Then, in the qualitative setting, all obligations spawned by $\mathsf{Strengthen}_{\mathcal{H}}$ are of the form $(i, s, 0)$.

$\mathsf{Strengthen}_{\mathcal{H}}$ *returns* false. There are two cases in which $\mathsf{Strengthen}_{\mathcal{H}}$ returns false. The first case (the left disjunct of l. 4) is that we encounter an obligation for frame $\mathcal{F}_0$. Resolving such an obligation would inevitably violate *initiality*; we thus return false. The second case (the right disjunct of l. 4) is that we encounter an obligation (i, s, δ) for a bad state $s \in B$ with a probability $\delta < 1$ (though, obviously, all $s \in B$ have probability 1). Resolving such an obligation would inevitably prevent us from preserving *relative inductivity*: If we updated $\mathcal{F}_i(s)$ to δ, we would have $\Phi(\mathcal{F}_{i-1})(s) = 1 > \delta = \mathcal{F}_i(s)$. Notice that, in contrast to IC3, this second case *can* occur in PrIC3:

Example 6.7.
Assume we have to resolve an obligation $(i, s_3, 1/2)$ for the MDP in Figure 6.10 on page 222. This involves spawning obligations $(i{-}1, s_4, \delta_1)$ and $(i{-}1, s_5, \delta_2)$, where s_5 is a bad state, such that $1/3 \cdot \delta_1 + 2/3 \cdot \delta_2 \leq 1/2$. Even for $\delta_1 = 0$, this is only possible if

$$\delta_2 \leq 3/4 < 1 .$$

$\mathsf{Strengthen}_{\mathcal{H}}$ *cannot refute safety.* If $\mathsf{Strengthen}_{\mathcal{H}}$ returns false, there are two possible reasons: *Either* the MDP is indeed unsafe, *or* the heuristic $\mathcal{H}$ at some point selected probabilities in a way such that $\mathsf{Strengthen}_{\mathcal{H}}$ is unable to restore the PrIC3 invariants (even though the MDP might in fact be safe). $\mathsf{Strengthen}_{\mathcal{H}}$ thus only returns a *potential* counterexample which either proves unsafety or indicates that our heuristic $\mathcal{H}$ was inappropriate.

Counterexamples in our case consist of subsystems rather than a single path (see Challenge 2). $\mathsf{Strengthen}_{\mathcal{H}}$ hence returns the set Q.touched() of all states that eventually appeared in the obligation queue. Finally, in the special case of the qualitative setting, $\mathsf{Strengthen}_{\mathcal{H}}$ *can* detect unsafety of $\mathcal{M}$:

Recovery Statement 4.
Let $\mathcal{H}_0$ be the adequate heuristic mapping every state to 0. Then, in the qualitative setting, $\mathsf{Strengthen}_{\mathcal{H}_0}$ returns false only if $\mathcal{M}$ is unsafe.

This yields our final recovery statement:

Recovery Statement 5.
Let $\mathcal{H}_0$ be the adequate heuristic mapping every state to 0. Then, in the qualitative setting, $\mathsf{PrIC3}_{\mathcal{H}_0}$ and $\mathsf{IC3}$ coincide, i.e.,

$$\mathsf{PrIC3}_{\mathcal{H}_0}(\mathcal{M}, s_I, B, 0) \;=\; \mathsf{IC3}(\mathsf{TS}_{\mathcal{M}}, B) \;.$$

Hence, in particular, $\mathsf{PrIC3}_{\mathcal{H}_0}(\mathcal{M}, s_I, B, 0)$ returns true iff $\mathcal{M}$ is safe. Moreover, the frames computed by $\mathsf{PrIC3}_{\mathcal{H}_0}$ and $\mathsf{IC3}$ coincide on every loop iteration.

6.2.6 Dealing with Potential Counterexamples

Recall that our core algorithm $\mathsf{PrIC3}_{\mathcal{H}}$ is incomplete for a fixed heuristic $\mathcal{H}$: It cannot give a conclusive answer whenever it finds a potential counterexample for two possible reasons: Either the heuristic $\mathcal{H}$ turned out to be inappropriate or the MDP is indeed unsafe. The idea to overcome the former is to call $\mathsf{PrIC3}_{\mathcal{H}}$ finitely often in an outer loop that generates new heuristics until we find an appropriate one: If $\mathsf{PrIC3}_{\mathcal{H}}$ still does not report safety of the MDP, then it is indeed unsafe. We do not blindly generate new heuristics, but use the potential counterexamples returned by $\mathsf{PrIC3}_{\mathcal{H}}$ to refine the previous one.

Let us consider the procedure $\mathsf{PrIC3}$ in Algorithm 11 which wraps our core algorithm $\mathsf{PrIC3}_{\mathcal{H}}$ in more detail: First, we create an *oracle* $\Omega\colon \mathcal{S} \to [0,1]$ which (roughly) *estimates* the maximal probability of reaching B for every state. A *perfect oracle* would yield *precise* maximal reachability probabilities, i.e.,

$$\text{for all } s \in \mathcal{S}\colon \quad \Omega(s) \;=\; \mathsf{MaxPr}(s \models \lozenge B) \;.$$

We construct oracles by user-supplied methods (highlighted in blue). Our implementation of this method is discussed in Section 6.2.7.2.

Assuming the oracle is good, but not perfect, we construct an adequate heuristic $\mathcal{H}$ selecting probabilities based on the oracle[13] for all successors of a given state: There are various options. The simplest is to pass-through the oracle values. A version that is more robust against noise in the oracle is discussed in

[13]We thus assume that heuristic $\mathcal{H}$ invokes the oracle whenever it needs to guess some probability.

```
Algorithm 11: PrIC3($\mathcal{M}, s_I, B, \lambda$)
input: MDP $\mathcal{M}$, initial state $s_I \in \mathcal{S}$, set of bad states $B$ with $s_I \notin B$,
       threshold $\lambda \in [0,1] \cap \mathbb{Q}$
output: true if $\mathcal{M}$ is safe; false otherwise
1  $\Omega \leftarrow$ Initialize()
2  $touched \leftarrow \{s_I\}$
3  do
4      $\mathcal{H} \leftarrow$ CreateHeuristic($\Omega$)
5      $safe, subsystem \leftarrow$ PrIC3$_{\mathcal{H}}$($\mathcal{M}, s_I, B, \lambda$)
6      if $safe$ then return true
7      if CheckRefutation($subsystem$) then return false
8      $touched \leftarrow$ Enlarge($touched, subsystem$)
9      $\Omega \leftarrow$ Refine($\Omega, touched$)
10 while $touched \neq \mathcal{S}$
11 return $\Omega(s_I) \leq \lambda$
```

Figure 6.14: The outermost loop dealing with possibly imprecise heuristics and potential counterexamples to safety.

Section 6.2.7.3. We then invoke PrIC3$_{\mathcal{H}}$. If PrIC3$_{\mathcal{H}}$ reports safety, the MDP is indeed safe by the soundness of PrIC3$_{\mathcal{H}}$.

6.2.6.1 Check Refutation

If PrIC3$_{\mathcal{H}}$ does not report safety, it reports a subsystem that hints to a *potential* counterexample. Formally, this subsystem is a subMDP[14] of $\mathcal{M}$ containing the states that were "visited" during the invocation of Strengthen$_{\mathcal{H}}$.

Definition 6.7 (subMDPs (adapted from [CV10])).
Given a subset $\mathcal{S}' \subseteq \mathcal{S}$ of states with $s_I \in \mathcal{S}$, we call the structure

$$\mathcal{M}_{\mathcal{S}'} \;=\; (\mathcal{S}', \mathsf{Act}, P')\;,$$

where $P' \colon \mathcal{S}' \times \mathsf{Act} \times \mathcal{S}' \to [0,1]$ is given by

$$\text{for all } s, s' \in \mathcal{S}' \text{ and all } \mathfrak{a} \in \mathsf{Act}\colon \quad P'(s, \mathfrak{a}, s') \;=\; P(s, \mathfrak{a}, s')\;,$$

the *subMDP of $\mathcal{M}$ induced by $\mathcal{S}'$*.

$\mathcal{M}_{\mathcal{S}'}$ may be substochastic where, intuitively, the missing probability mass never reaches a bad state. Definition 2.8 on page 32 is thus relaxed: For all states $s \in \mathcal{S}'$ and all actions $\mathfrak{a} \in \mathsf{Act}$, we require that $\sum_{s' \in \mathcal{S}'} P(s, \mathfrak{a}, s') \leq 1$. The notion of maximal reachability probabilities carries over to subMDPs. Crucially, if the subMDP is unsafe, we can conclude that the original MDP $\mathcal{M}$ is also unsafe:

Lemma 6.13 (Unsafety of subMDPs [JDKK⁺16]).
If $\mathcal{M}_{\mathcal{S}'}$ is a subMDP of $\mathcal{M}$, then

$$\mathsf{MaxPr}(\mathcal{M}_{\mathcal{S}'}, s_I \models \lozenge B) \;>\; \lambda \qquad \text{implies} \qquad \mathsf{MaxPr}(\mathcal{M}, s_I \models \lozenge B) \;>\; \lambda\,.$$

The role of CheckRefutation is to establish whether the subsystem is indeed a true counterexample or a spurious one. Formally, we require that

$$\mathsf{CheckRefutation}(\mathit{subsystem}) \;=\; \mathsf{true} \qquad \text{iff} \qquad \mathcal{M}_{\mathit{subsystem}} \text{ is unsafe}\,.$$

We remark that the procedure CheckRefutation invoked in l. 7 is a classical fallback; it can be implemented using the standard LP-based characterization of maximal reachability probabilities in finite-state MDPs ([BK08, Theorem 10.105]). In the worst case, i.e., for $\mathcal{S}' = \mathcal{S}$, we thus solve exactly our problem statement via this standard approach.

[14]Recall that we fix an $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$, an initial state $s_I \in \mathcal{S}$, a set $B \subseteq \mathcal{S}$ of bad states, and a threshold probability λ.

6.2.6.2 Refine Oracle

Whenever we have neither proven the MDP safe nor unsafe, we refine the oracle to prevent generating the same subsystem in the next invocation of $\mathsf{PrIC3}_{\mathcal{H}}$. To ensure termination, oracles should only be refined finitely often. That is, we need some progress measure. The set *touched* overapproximates all counterexamples encountered in some invocation of $\mathsf{PrIC3}_{\mathcal{H}}$ and we propose to use its size as the progress measure. While there are several possibilities to update *touched* through the user-defined procedure Enlarge (l. 8), every implementation should hence satisfy

$$|\mathsf{Enlarge}(\mathit{touched}, \_\_)| > |\mathit{touched}| .$$

Consequently, after finitely many iterations, the oracle is refined with respect to all states. We may then as well rely on solving the characteristic LP problem:

Lemma 6.14 (Soundness and Termination of PrIC3).
The algorithm PrIC3 in Algorithm 11 terminates and is sound, if $\mathsf{Refine}(\Omega, \mathcal{S})$ returns a perfect oracle Ω (with $\mathcal{S}$ is the set of all states).

Notice that the above lemma does not rely on the abstract concept that heuristic $\mathcal{H}$ provides suitable probabilities after finitely many refinements.

6.2.7 Practical PrIC3 for Probabilistic Programs

So far, we gave a conceptual view on PrIC3. In this section, we take a more practical stance: We marry PrIC3 with the weakest preexpectation calculus to obtain a practical algorithm for model checking finite-state probabilistic loops.

6.2.7.1 Setting

We operate in the setting from Chapter 5: We assume the set Vars of program variables to be finite and that program variables $x \in \mathsf{Vars}$ are $\mathbb{N}$-valued. We employ PrIC3 to tackle the following problem: Given a *fully probabilistic*[15] *and finite-state*[16] loop $C = \mathtt{while}(\varphi)\{C'\} \in \mathsf{LpGCL}$ in the linear fragment of pGCL (cf. Definition 4.3 on page 146), a linear Boolean expression $\eta \in \mathsf{LBExpr}_{\mathbb{Z}}$ describing a set of “bad” final states, an initial program state σ_I with $\sigma_I \models \varphi$, and a threshold probability $\lambda \in [0,1] \cap \mathbb{Q}$,

[15]Extending our results to nondeterministic programs is possible but left for future work.
[16]Recall that we call C finite-state, if $|[\![\varphi]\!]| < \infty$.

$$\boxed{\text{Verify or refute that } \mathsf{wp}[\![C]\!]([\eta])(\sigma_I) \leq \lambda.}$$

Due to the tight connection between weakest preexpectations and expected reachability-rewards in pGCL's operational MDP $\mathcal{O}$ discussed in Section 2.4.5, instantiating PrIC3 for tackling the above problem *requires almost no adaptions*. To see this, we associate with the loop C the Markov chain[17]

$$\mathcal{M}_C = (\mathsf{States}, P_C) \ , \text{ where } \quad P_C(\sigma, \sigma') = \underbrace{[\![\mathtt{if}\ (\varphi)\ \{C'\}\ \mathtt{else}\ \{\mathtt{skip}\}]\!]^{\sigma}(\sigma')}_{\substack{\text{probability of reaching } \sigma' \text{ from } \sigma \\ \text{via } one \text{ guarded loop iteration} \\ \text{(cf. Definition 2.18 on page 50)}}} \ .$$

Notice that, even though $\mathcal{M}_C$ is infinite-state since States is infinite, there are only finitely many *relevant* states — those reachable from σ_I. Notice furthermore that both P_C and the set

$$\mathsf{Succs}(\sigma) \ = \ \underbrace{\{\sigma' \in \mathsf{States} \mid P_C(\sigma, \sigma') > 0\}}_{\text{finite}}$$

are computable[18] for all $\sigma \in$ States since the loop body C' is loop-free. We invoke PrIC3 on this Markov chain. Consequently, frames $\mathcal{F}$ are of type States $\to [0, 1]$, i.e., 1-bounded expectations. The set of bad states is

$$B \ = \ [\![\neg\varphi \wedge \eta]\!] \ ,$$

i.e., states where C has terminated and that satisfy η. Exploiting that states satisfying $\neg\varphi \wedge \neg\eta$ never reach a bad state, we define the Bellman operator as

$$\Phi(\mathcal{F}) \ = \ \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models \neg\varphi \wedge \neg\eta \\ 1 & \text{if } \sigma \models \neg\varphi \wedge \eta \\ \displaystyle\sum_{\sigma' \in \mathsf{States}} P_C(\sigma, \sigma') \cdot \mathcal{F}(\sigma') & \text{otherwise} \ . \end{cases}$$

This Bellman operator in fact *coincides* with the wp-characteristic function ${}^{\mathsf{wp}}_{C}\Phi_{[\eta]}$ of C w.r.t. $[\eta]$ when restricting its domain to 1-bounded expectations because we have for every frame $\mathcal{F}$,

$${}^{\mathsf{wp}}_{C}\Phi_{[\eta]}(\mathcal{F})$$

[17] Recall that a Markov chain is an MDP where, in every state, exactly one action is enabled. We assume here that the set Act of actions is a singleton and omit it for the sake of readability.

[18] Our implementation uses Storm.

$$= [\varphi] \cdot \mathsf{wp}[\![C']\!](\mathcal{F}) + [\neg\varphi] \cdot [\eta] \qquad \text{(Definition 2.21 on page 63)}$$

$$= \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models \neg\varphi \wedge \neg\eta \\ 1 & \text{if } \sigma \models \neg\varphi \wedge \eta \\ \mathsf{wp}[\![C']\!](\mathcal{F})(\sigma) & \text{otherwise}. \end{cases}$$

$$= \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models \neg\varphi \wedge \neg\eta \\ 1 & \text{if } \sigma \models \neg\varphi \wedge \eta \\ \sum_{\sigma' \in \mathsf{States}} [\![C']\!]^{\sigma}(\sigma') \cdot \mathcal{F}(\sigma') & \text{otherwise}. \end{cases}$$

(Corollary 2.16 on page 72)

$$= \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models \neg\varphi \wedge \neg\eta \\ 1 & \text{if } \sigma \models \neg\varphi \wedge \eta \\ \sum_{\sigma' \in \mathsf{States}} [\![\mathtt{if}\,(\varphi)\,\{C'\}\,\mathtt{else}\,\{\mathtt{skip}\}]\!]^{\sigma}(\sigma') \cdot \mathcal{F}(\sigma') & \text{otherwise}. \end{cases}$$

(we have $\sigma \models \varphi$ in the third case)

$$= \lambda\sigma. \begin{cases} 0 & \text{if } \sigma \models \neg\varphi \wedge \neg\eta \\ 1 & \text{if } \sigma \models \neg\varphi \wedge \eta \\ \sum_{\sigma' \in \mathsf{States}} P_C(\sigma, \sigma') \cdot \mathcal{F}(\sigma') & \text{otherwise}. \end{cases} \qquad \text{(by definition)}$$

$$= \Phi(\mathcal{F}).$$

In particular, we have for every state σ,

$$(\mathsf{lfp}\ \Phi)(\sigma) \;=\; \mathsf{Pr}(\sigma \models \Diamond B) \;=\; \left(\mathsf{lfp}\ {}^{\mathsf{wp}}_{C}\Phi_{[\eta]}\right)(\sigma) \;=\; \mathsf{wp}[\![C]\!]([\eta])(\sigma).$$

Hence, we can verify or refute whether $\mathsf{wp}[\![C]\!]([\eta])(\sigma_I) \leq \lambda$ holds by invoking PrIC3($\mathcal{M}_C, \sigma_I, B, \lambda$), where the wp-characteristic function replaces the Bellman operator. Notice that, due to the above reasoning, PrIC3 actually computes a wp-superinvariant of C w.r.t. $[\eta]$ to prove $\mathcal{M}_C$ safe.

We represent frames as piecewise linear[19] expectations in LExp (cf. Definition 4.4 on page 147). Resolving an obligation (i, σ, δ) in Strengthen$_{\mathcal{H}}$ (l. 11 in Algorithm 10) is realized using the construction for pointwise minima from Lemma 4.18 on page 154, i.e.,

$$\mathcal{F}_i(\sigma) \leftarrow \min(\mathcal{F}_i(\sigma), \delta) \quad \text{becomes} \quad \mathcal{F}_i \leftarrow \mathcal{F}_i \sqcap ([\sigma] \cdot \delta + [\neg\sigma] \cdot \infty),$$

[19] In Section 6.2.7.4, we present a technique which produces piecewise *polynomial* expectations but all required properties carry over.

where $[\sigma]$ is the *characteristic expectation* of state σ given by

$$[\sigma] \;=\; \Big[\bigwedge_{x \in \mathsf{Vars}} x = \sigma(x)\Big] \;=\; \lambda\sigma'.\begin{cases} 1 & \text{if } \sigma' = \sigma \\ 0 & \text{otherwise}\,. \end{cases}$$

Since every frame $\mathcal{F}_i$ maintained by the core $\mathsf{PrIC3}_{\mathcal{H}}$ loop is initialized by the constant-1-frame, $\mathcal{F}_i$ will always be of the form

$$\mathcal{F}_i \;=\; \mathcal{F}_1' \sqcap \ldots \sqcap \mathcal{F}_m'$$

for some $m \in \mathbb{N}$ and some frames $\mathcal{F}_1', \ldots, \mathcal{F}_m'$. In our implementation, we track this set of "conjuncts" for each frame $\mathcal{F}_i$ maintained by $\mathsf{PrIC3}_{\mathcal{H}}$. Since Algorithm 10, l. 11 always updates not only $\mathcal{F}_i$ but *all* $\mathcal{F}_1, \ldots, \mathcal{F}_i$, we can check whether two frames $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ coincide (l. 7 in Algorithm 9) in a syntactic manner by checking whether the set of conjuncts of $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ coincide[20].

6.2.7.2 Obtaining and Refining Oracles

Recall that the outermost PrIC3 loop (Algorithm 11) requires user-supplied methods for initializing (l. 1) and refining (l. 9) an oracle $\Omega\colon \mathsf{States} \to [0,1]$, which roughly estimates the probability of reaching B for every state σ. In this section, we describe the methods we chose for our implementation.

Orcale initialization. States $\sigma \models \neg\varphi \wedge \eta$ are bad. For such states, we thus let

$$\Omega(\sigma) \;=\; 1\,.$$

Conversely, states satisfying $\sigma \models \neg\varphi \wedge \neg\eta$ never reach a bad state and we let

$$\Omega(\sigma) \;=\; 0\,.$$

It remains to assign probabilities to the states satisfying the loop guard φ. For that, we use Storm to compute a finite set States' of states reachable from the initial state σ_I in breadth-first search manner and such that $|\mathsf{States}| = \min(|[\![\varphi]\!]|, 5000)$. For all states with $\sigma \not\in \mathsf{States}'$ and $\sigma \models \varphi$, we let

$$\Omega(\sigma) \;=\; 1\,.$$

[20]This is analogous to bit-level IC3's syntactic termination check (cf. Section 6.1.4.2): There, frames $\mathcal{F}_i$ are conjunctions of clauses and two frames $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ coincide iff their corresponding sets of clauses coincide.

For all remaining states, i.e., for all $\sigma \in \mathsf{States}'$ with $\sigma \models \varphi$, we introduce a $[0,1]$-valued variable $\mathfrak{a}_\sigma$ and solve the following linear equation system:

$$\bigwedge_{\substack{\sigma \in \mathsf{States}' \\ \sigma \models \varphi}} \mathfrak{a}_\sigma \;=\; \sum_{\sigma' \in \mathsf{Succs}(\sigma)} P_C(\sigma, \sigma') \cdot \begin{cases} 0 & \text{if } \sigma' \models \neg\varphi \land \neg\eta \\ 1 & \text{if } \sigma' \models \neg\varphi \land \eta \text{ or } \sigma' \notin \mathsf{States}' \\ \mathfrak{a}_{\sigma'} & \text{if } \sigma' \in \mathsf{States}' \,. \end{cases}$$

We then let

$$\Omega(\sigma) \;=\; \mathfrak{a}_\sigma \,.$$

The above equation system bears a close resemblance to the standard equation system for computing reachability probabilities in finite-state Markov chains (cf. [BK08, Theorem 10.19]), where states $\sigma' \notin \mathsf{States}'$ are considered bad.

Oracle refinement. Denote by Ω the current oracle and by *touched* $\subseteq$ States the finite set of states maintained by the outermost PrIC3 loop (Algorithm 11). Now introduce a $[0,1]$-valued variable $\mathfrak{a}_\sigma$ for all $\sigma \in$ *touched*. We obtain the refined oracle Ω' by solving the following linear equation system:

$$\bigwedge_{\substack{\sigma \in \mathsf{States}' \\ \sigma \models \varphi}} \mathfrak{a}_\sigma \;=\; \sum_{\sigma' \in \mathsf{Succs}(\sigma)} P_C(\sigma, \sigma') \cdot \begin{cases} 0 & \text{if } \sigma' \models \neg\varphi \land \neg\eta \\ 1 & \text{if } \sigma' \models \neg\varphi \land \eta \\ \Omega(\sigma') & \text{if } \sigma' \models \varphi \text{ and } \sigma' \notin \textit{touched} \\ \mathfrak{a}_{\sigma'} & \text{if } \sigma' \models \varphi \text{ and } \sigma' \in \textit{touched} \,. \end{cases}$$

We then let

$$\Omega' \;=\; \lambda\sigma. \begin{cases} \Omega(\sigma) & \text{if } \sigma \not\models \varphi \text{ or } \sigma \notin \textit{touched} \\ \mathfrak{a}_\sigma & \text{otherwise} \,. \end{cases}$$

Finally, if *touched* covers all states in $[\![\varphi]\!]$, we fall back to the standard procedure for computing reachability probabilities in finite-state Markov chains ([BK08, Theorem 10.19])) to ensure that

$$\Omega'(\sigma_I) \;=\; \mathsf{Pr}(\sigma_I \models \lozenge B) \,.$$

Enlarging touched. Finally, we implement Enlarge as

$$\begin{aligned} &\mathsf{Enlarge}(\textit{touched}, \textit{subsystem}) \\ =\;& \textit{touched} \cup \begin{cases} \textit{subsystem} & \text{if } \textit{subsystem} \not\subseteq \textit{touched} \\ \bigcup_{\sigma \in \textit{touched}} \mathsf{Succs}(\sigma) & \text{otherwise} \end{cases} \end{aligned}$$

so that *touched* contains all relevant states after finitely many iterations, which yields PrIC3 (Algorithm 11) to terminate.

6.2.7.3 Obtaining Adequate Heuristics

Recall that $\mathrm{Strengthen}_{\mathcal{H}}$ invokes a heuristic[21] $\mathcal{H}\colon \mathrm{States} \times [0,1] \to [0,1]^*$ (l. 8 in Algorithm 10). We use the oracle $\Omega\colon \mathrm{States} \to [0,1]$ maintained by the outermost PrIC3 loop to construct an adequate heuristic: Suppose we encounter an obligation of the form (i, σ, δ) with $\mathrm{Succs}(\sigma) = \{\sigma_1, \ldots, \sigma_n\}$. We obtain the probabilities $\delta_1, \ldots, \delta_n$ for these successors as follows: If $\delta = 0$, we let

$$\mathcal{H}(\sigma, \delta) \;=\; \delta_1, \ldots, \delta_n \;=\; 0, \ldots, 0\,.$$

If $\delta > 0$, then we attempt to solve the following optimization problem[22]:

$$\begin{aligned}
\text{minimize} \quad & \sum_{\substack{i=1 \\ \sigma_i \notin B}}^{n} \left| \frac{\delta_i}{\sum_{j=1}^{n} \delta_j} - \frac{\Omega(\sigma_i)}{\sum_{j=1}^{n} \Omega(\sigma_j)} \right| \\
\text{subject to} \quad & \delta = \sum_{i=1}^{n} P_C(\sigma, \sigma_i) \cdot \delta_i \\
& \wedge \bigwedge_{\substack{i=1 \\ \sigma_i \in B}}^{n} \delta_i = 1 \\
& \wedge\, 0 \leq \delta_1, \ldots, \delta_n \leq 1\,,
\end{aligned}$$

where we let $0.5 = \Omega(\sigma_1) = \ldots = \Omega(\sigma_n)$ in case $\sum_{j=1}^{n} \Omega(\sigma_j) = 0$. The objective function aims to preserve the ratio between the probabilities suggested by the oracle Ω. Now, if the above optimization problem has a solution, we let

$$\mathcal{H}(\sigma, \delta) \;=\; \delta_1, \ldots, \delta_n\,.$$

If the problem does *not* have a solution, then it is impossible to strengthen the frames in such a way that resolving the obligation (i, σ, δ) preserves the PrIC3 invariants. In this case, at least one state $\sigma' \in \mathrm{Succs}(\sigma)$ is bad and we let

$$\mathcal{H}(\sigma, \delta) \;=\; 0, \ldots, 0\,.$$

This yields $\mathrm{Strengthen}_{\mathcal{H}}$ to return false (l. 5) as soon as it pops the obligation $(i-1, \sigma', 0)$. In our implementation, we apply a minor optimization by immediately returning false instead of spawning new obligations.

[21] We omit the Act component since we restrict to Markov chains.

[22] Our implementation uses Z3's extension for Optimization Modulo Theories [BPF15].

6.2.7.4 Generalization in PrIC3

Analogously to IC3 for transition system, generalization in PrIC3 takes place right before l. 11 in Algorithm 10. Suppose an obligation (i, σ, δ) can be resolved without violating the PrIC3 invariants by executing

$$\mathcal{F}_1(\sigma) \leftarrow \min(\mathcal{F}_1(\sigma), \delta); \ldots ; \mathcal{F}_i(\sigma) \leftarrow \min(\mathcal{F}_i(\sigma), \delta) . \tag{6.3}$$

Generalization aims to update *more* values than just those at σ without violating the PrIC3 invariants. The goal is thus to *heuristically guess* a frame $\mathcal{F}$ with

$$\text{(i)} \ \ \mathcal{F}(\sigma) \ \leq \ \delta \qquad \text{and} \qquad \text{(ii)} \ \ \Phi(\mathcal{F}_i) \ \sqsubseteq \ \mathcal{F}$$

to accelerate the search for an inductive invariant. We call such an $\mathcal{F}$ a *generalization of* (σ, δ) *w.r.t.* $\mathcal{F}_i$. Instead of updating the frames by executing (6.3), we then update the frames using the generalization $\mathcal{F}$:

$$\mathcal{F}_1 \leftarrow \mathcal{F}_1 \sqcap \mathcal{F}; \ldots ; \mathcal{F}_i \leftarrow \mathcal{F}_i \sqcap \mathcal{F} ,$$

Since we represent both the frames maintained by PrIC3 and the frame $\mathcal{F}$ as piecewise linear expectations, we can employ the techniques from Section 4.6.3 to decide the above conditions (i) and (ii) and for computing pointwise minima in a fully symbolic manner via SMT solving techniques.

In what follows, we discuss three approaches for obtaining generalizations based on *polynomial interpolation*. Since the given loop C is finite-state, we assume that each program variable x from the finite set Vars is upper-bounded by a given constant $u_x \in \mathbb{N}$.

Linear generalization. This technique is depicted in Algorithm 12 on page 249. LinearGeneralization takes as input a frame $\mathcal{F}$, a state σ, and a probability δ with $\Phi(\mathcal{F})(\sigma) \leq \delta$. The result is a generalization $\mathcal{F}'$ of (σ, δ) w.r.t. $\mathcal{F}$. For that, $\mathcal{F}'$ is initialized in l. 1 by

$$[\sigma] \cdot \delta + [\neg \sigma] \cdot \infty ,$$

which is a valid generalization by assumption. We then attempt to strengthen $\mathcal{F}'$ further by doing the following for each variable $x \in \mathsf{Vars}$: We aim to find a generalization $\mathcal{F}''$ of the form

$$[\vartheta] \cdot e + [\neg \vartheta] \cdot \infty \qquad \text{with} \qquad \vartheta \quad = \quad \sigma(x) \leq x \leq u_x \wedge \bigwedge_{y \in \mathsf{Vars} \setminus \{x\}} y = \sigma(y) ,$$

```
Algorithm 12: LinearGeneralization($\mathcal{F}, \sigma, \delta$)
  input: Frame $\mathcal{F}$, state $\sigma$, and probability $\delta$ with $\Phi(\mathcal{F})(\sigma) \leq \delta$
  output: A generalization $\mathcal{F}'$ of $(\sigma, \delta)$ w.r.t. $\mathcal{F}$
1 $\mathcal{F}' \leftarrow [\sigma] \cdot \delta + [\neg\sigma] \cdot \infty$
2 for $x \in \mathsf{Vars}$ do
      /* try to generalize between $x$'s current and maximal value */
3     $\vartheta \leftarrow \sigma(x) \leq x \leq u_x \wedge \bigwedge_{y \in \mathsf{Vars} \setminus \{x\}} y = \sigma(y)$
      /* obtain $e$ of the form $q \cdot x + r$ passing through the given points */
4     $e \leftarrow \mathsf{Interpolate}(\{(\sigma(x), \delta), (u_x, \Phi(\mathcal{F})(\sigma[x \mapsto u_x]))\})$
5     $\mathcal{F}'' \leftarrow [\vartheta] \cdot e + [\neg\vartheta] \cdot \infty$
6     if $\Phi(\mathcal{F}) \sqsubseteq \mathcal{F}''$ then
          /* generalizing with $\mathcal{F}''$ is possible */
7         $\mathcal{F}' \leftarrow \mathcal{F}' \sqcap \mathcal{F}''$
8 return $\mathcal{F}'$
```

Figure 6.15: Linear generalization in PrIC3.

i.e., we aim to lower the values of x in the interval $[\sigma(x), u_x]$. For that, we assign to e the affine expression in x interpolating between the points

$$(\sigma(x), \delta) \qquad \text{and} \qquad (u_x, \Phi(\mathcal{F})(\sigma[x \mapsto u_x])) \ .$$

If the resulting $\mathcal{F}''$ satisfies $\Phi(\mathcal{F}) \sqsubseteq \mathcal{F}''$, we strengthen $\mathcal{F}'$ by $\mathcal{F}''$ in l. 7.

Polynomial generalization. This technique, depicted in Algorithm 13 on page 250, is an extension of linear generalization. In addition to $\mathcal{F}$, σ, and δ, PolynomialGeneralization takes as input some natural number n, which bounds the number of so-called *counterexamples to generalization (CTGs)*. Considering CTGs is inspired by [HBS13] and works as follows: We proceed as for linear generalization. However, if the frame $\mathcal{F}''$ obtained from interpolating between

$$(\sigma(x), \delta) \qquad \text{and} \qquad (u_x, \Phi(\mathcal{F})(\sigma[x \mapsto u_x]))$$

does *not* satisfy $\Phi(\mathcal{F}) \sqsubseteq \mathcal{F}''$, we do not immediately give up. Rather, we obtain a state σ' with $\Phi(\mathcal{F})(\sigma') > \mathcal{F}''(\sigma')$ in l. 10 — a CTG — and add the point $(\sigma'(x), \Phi(\mathcal{F})(\sigma'))$ to the points used for interpolation. In the next iteration, we

```
Algorithm 13: PolynomialGeneralization($\mathcal{F}, \sigma, \delta, n$)
  input: Frame $\mathcal{F}$, state $\sigma$, probability $\delta$ with $\Phi(\mathcal{F})(\sigma) \leq \delta$, and $n \in \mathbb{N}$
  output: A generalization $\mathcal{F}'$ of $(\sigma, \delta)$ w.r.t. $\mathcal{F}$
1 $\mathcal{F}' \leftarrow [\sigma] \cdot \delta + [\neg\sigma] \cdot \infty$
2 for $x \in \mathsf{Vars}$ do
      /* try to generalize between $x$'s current value and maximal value */
3     $\vartheta \leftarrow \sigma(x) \leq x \leq u_x \wedge \bigwedge_{y \in \mathsf{Vars} \setminus \{x\}} y = \sigma(y)$
4     points $\leftarrow \{(\sigma(x), \delta), (u_x, \Phi(\mathcal{F})(\sigma[x \mapsto u_x]))\}$
5     $i \leftarrow 0$
6     do
          /* obtain univariate polynomial in $x$ of degree |points| − 1 */
7         $e \leftarrow$ Interpolate(points)
8         $\mathcal{F}'' \leftarrow [\vartheta] \cdot e + [\neg\vartheta] \cdot \infty$
9         if $\neg(0 \sqsubseteq [\vartheta] \cdot e \sqsubseteq 1)$ then break /* ensure well-typedness */
10        if $\exists \sigma' \colon \Phi(\mathcal{F})(\sigma') > \mathcal{F}''(\sigma')$ then for such a $\sigma'$
              /* $\sigma'$ is a counterexample to generalization */
11            points $\leftarrow$ points $\cup \{(\sigma'(x), \Phi(\mathcal{F})(\sigma')\}$
12        else
              /* generalizing with $\mathcal{F}''$ is possible */
13            $\mathcal{F}' \leftarrow \mathcal{F}' \sqcap \mathcal{F}''$
14            break
15        $i \leftarrow i + 1$
16    while $i \leq n$
17 return $\mathcal{F}'$
```

Figure 6.16: Polynomial generalization in PrIC3.

```
Algorithm 14: Propagate($\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$)
  input: $\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$ with PrIC3Inv($\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$)
  output: strengthening $\mathcal{F}_0',\ldots,\mathcal{F}_{k+1}'$ of $\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$ with PrIC3Inv$\left(\mathcal{F}_0',\ldots,\mathcal{F}_k'\right)$
1 for $i = 1,\ldots,k$ do
      /* let $\mathcal{F}_i = \mathcal{F}_1' \sqcap \ldots \sqcap \mathcal{F}_m'$ */
2     for $j = 1,\ldots,m$ do
3         if $\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}_j'$ then
              /* propagating $\mathcal{F}_j'$ to $\mathcal{F}_{i+1}$ is possible */
4             $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \sqcap \mathcal{F}_j'$
5 return $\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$
```

Figure 6.17: Propagation in PrIC3.

thus possibly obtain a univariate polynomial in x of degree greater than 1. Hence, PolynomialGeneralization produces piecewise *polynomial* expectations. The techniques from Section 4.6.3 for deciding[23] quantitative entailments and computing pointwise minima carry over to these expectations. Moreover, we have to take care: The resulting polynomials are not necessarily $[0,1]$-valued. Whenever we obtain a polynomial that is not $[0,1]$-valued, we give up (l. 9).

Hybrid generalization. We proceed as for polynomial generalization. However, to avoid reasoning about non-linear arithmetic, we soundly approximate the obtained piecewise polynomial expectations by piecewise linear expectations.

6

6.2.7.5 Propagation in PrIC3

Recall that propagation takes place in Algorithm 9, l. 6. Propagate takes as input a sequence $\mathcal{F}_0,\ldots,\mathcal{F}_{k+1}$ frames satisfying the PrIC3 invariants. For each frame $\mathcal{F}_i$ with $i \geq 1$, it then aims to "propagate" values of states to higher frames whenever this does not violate the PrIC3 invariants.

[23]For this, we relax the problem of deciding quantitative entailments by assuming the program variables to be $\mathbb{R}_{\geq 0}$-valued, i.e., we employ satisfiability checking for decidable non-linear real arithmetic. This is sound yet incomplete: It might be the case that a quantitative entailment $f \sqsubseteq g$ holds for $\mathbb{N}$-valued variables but not for $\mathbb{R}_{\geq 0}$-valued program variables.

				PrIC3								Storm	
				w/o		lin		pol		hyb			
Prog	$\lvert S\rvert$	$\Pr(\sigma_I \models \Diamond B)$	λ	t	$\lvert subsys\rvert$	t	$\lvert subsys\rvert$	t	$\lvert subsys\rvert$	t	$\lvert subsys\rvert$	sp	dd
BRP	10^3	0.035	0.1	TO	–	TO	–	TO	–	TO	–	<0.1	0.12
			0.01	**47.56**	324	509.46	324	TO	–	663.93	324		
			0.005	**10.98**	188	210.44	188	TO	–	TO	–		
ZeroConf	10^4	0.5	0.9	TO	–	TO	–	0.51	0	**0.39**	0	<0.1	281.64
			0.75	TO	–	TO	–	0.52	0	**0.46**	0		
			0.52	TO	–	TO	–	**0.40**	0	0.41	0		
			0.45	**<0.1**	1	**<0.1**	1	**<0.1**	1	**<0.1**	1		
ZeroConf	10^9	0.55	0.9	TO	–	TO	–	**1.05**	0	TO	–	MO	TO
			0.75	TO	–	TO	–	**1.06**	0	TO	–		
			0.52	TO	–	TO	–	TO	–	TO	–		
			0.45	**<0.1**	1	**<0.1**	1	**<0.1**	1	**<0.1**	1		
Chain	10^3	0.394	0.9	139.78	0	TO	–	0.76	0	**0.56**	0	<0.1	0.12
			0.4	153.42	0	TO	–	**0.78**	0	TO	–		
			0.354	**142.04**	431	TO	–	TO	–	TO	–		
			0.3	**76.24**	357	554.08	357	TO	–	TO	–		
Chain	10^4	0.394	0.9	TO	–	TO	–	0.78	0	**0.57**	0	<0.1	4.91
			0.48	TO	–	TO	–	**0.85**	0	TO	–		
			0.4	TO	–	TO	–	**0.81**	0	TO	–		
			0.3	TO	–	TO	–	TO	–	TO	–		
Chain	10^{12}	0.394	0.9	TO	–	TO	–	**0.78**	0	TO	–	MO	TO
			0.4	TO	–	TO	–	**0.79**	0	TO	–		
DoubleChain	10^3	0.215	0.9	TO	–	TO	–	**1.66**	0	1.73	0	<0.1	0.12
			0.3	TO	–	TO	–	**1.64**	0	1.74	0		
			0.2164	TO	–	TO	–	**130.48**	0	TO	–		
			0.15	TO	–	TO	–	TO	–	TO	–		
DoubleChain	10^4	0.22	0.94	TO	–	TO	–	1.68	0	**1.67**	0	<0.1	2.99
			0.3	TO	–	TO	–	**1.64**	0	1.694	0		
			0.24	TO	–	TO	–	**1.67**	0	TO	–		

Table 6.1: Experimental results. TO=15min, MO=8GB, time in seconds.

Propagate is given by the pseudo code in Algorithm 14 on page 251. We do the following for each frame $\mathcal{F}_i$, where i ranges from 1 to k: We consider each of the "conjuncts" $\mathcal{F}_j'$ comprising $\mathcal{F}_i$. Notice that, in the presence of generalization, $\mathcal{F}_j'$ could be a non-constant piecewise linear (polynomial) expectation. We then check whether $\mathcal{F}_{i+1}$ can be strengthened by $\mathcal{F}_j'$ without violating the PrIC3 invariants, which is the case if $\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}_j'$ by Theorem 6.9 on page 236.

6.2.8 Implementation and Experiments

We assess how PrIC3 may contribute to the state-of-the-art in probabilistic model checking. We do some early empirical evaluation showing that PrIC3 is feasible. We see ample room for further improvements to the prototype[24].

Setup. We have implemented a prototype of PrIC3 based on Section 6.2.7 in Python. The input[25] is represented using efficient data structures provided by the model checker Storm. We use Z3 for SMT solving, i.e., for obtaining and refining oracles, for solving the optimization problem which implements our heuristic $\mathcal{H}$, for checking relative inductivity, and for obtaining generalizations. We support all discussed generalizations (none, linear, polynomial, hybrid).

We choose a combination of models from the literature (BRP [HSV93; DJJL01], ZeroConf [CAG05]) and some structurally straightforward variants of grids (Chain, DoubleChain). Since our prototype is more sensitive to the precise encoding of a model, i.e., the program describing it, we have generated new encodings for all models. We use a 15 minute time-limit and report TO otherwise. Memory is limited to 8GB; we report MO if it is exceeded. To give an impression of the runtimes, we compare our prototype to both the explicit- (sp) and symbolic-state (dd) engine of the model checker Storm 1.5.1.

Results. In Table 6.1, we present the runtimes for various invocations of our prototype on an Intel Core i5-7360U 2.3GHz. In particular, we give the model name and the number of (relevant) states in the particular instance, and the (estimated) actual probability to reach B from the fixed initial state. For each model, we consider multiple thresholds λ. The next 8 columns report on the four variants of PrIC3 with varying generalization schemes. Besides the scheme with the run times, we report for each scheme the number of states of the largest (last) subsystem that CheckRefutation in Algorithm 11 was invoked upon (column $|subsys|$). The last two columns report on the run times for Storm that we provide for comparison. In each row, we mark with violet MDPs that are unsafe. We **highlight** the best configurations of PrIC3.

Discussion. Our experiments give a mixed picture of the performance of our implementation of PrIC3. On the one hand, Storm significantly outperforms PrIC3 on most models. On the other hand, PrIC3 is capable of reasoning about huge, yet simple, models with up to 10^{12} states that Storm is unable to analyze within the time and memory limits. There is more empirical evidence that PrIC3 may complement the state-of-the-art:

First, *the size of thresholds matters*. Our benchmarks show that more "wiggle

[24]The prototype is available open-source from https://github.com/moves-rwth/PrIC3.

[25]Our prototype accepts programs written in the PRISM language with a single module. These programs can be interpreted as a single-loop in pGCL – see [HJVM+21] for an explicit translation.

room” between the precise reachability probability and the threshold generally leads to better performance. PrIC3 may thus prove bounds for large models where a precise quantitative reachability analysis is out of scope.

Second, PrIC3 *enjoys the benefits of partial state space exploration*. In some cases, e.g., ZeroConf for $\lambda = 0.45$, PrIC3 refutes very fast as it does not need to explore the whole model.

Third, if PrIC3 proves the mode safe, it does so without relying on checking large subsystems in the CheckRefutation step.

Fourth, *generalization is crucial*. Without generalization, PrIC3 is unable to prove safety for any of the considered models with more than 10^3 states. With generalization, however, it can prove safety for very large systems and thresholds close to the exact reachability probability. For example, it proved safety of the Chain benchmark with 10^{12} states for a threshold of 0.4 which differs from the exact reachability probability by 0.006.

Fifth, *there is no best generalization*. There is no clear winner out of the considered generalization approaches. Linear generalization is the worst-performing. In fact, it performs worse than no generalization at all. The hybrid approach, however, occasionally has the edge over the polynomial approach. This indicates that more research is required to find suitable generalizations.

6.3 Future and Related Work

6.3.1 Future Work

We identify several directions for future work. A first direction is to investigate more sophisticated heuristics and generalizations. The recent works by Kori et al. [KABB+23; KUKS+22] provide various encouraging insights yielding such improvements, which we discuss in the next section. We plan to integrate the more expressive piecewise defined templates from Chapter 5 into our PrIC3 framework and to investigate whether this yields suitable generalizations.

Krishnan et al. [KVGG19] propose kAvy, which combines classical k-induction for transition systems with IC3/PDR and show that the so-obtained algorithm can be orders of magnitude faster than k-induction or PDR on their own. We hence consider it a promising direction to investigate whether our generalization of k-induction from Chapter 4 can be combined with PrIC3.

We have so far restricted to finite-state programs. However, our marriage of PrIC3 and the weakest preexpectation calculus from Section 6.2.7 strongly suggests that our framework can be extended for reasoning about *infinite-state* probabilistic programs. We plan to investigate the practical feasibility of such an extension as well as sufficient criteria for termination. We believe that insights by Kori et al. [KABB+23; KUKS+22] could yield such criteria.

Finally, we conjecture that the fact that PrIC3 requires a heuristic $\mathcal{H}$ — in a sense which needs to be formalized — is closely related to the fact that there does not exist a probabilistic analog of *strongest postconditions* [Jon90, p. 135]. The recent work by Kori et al. [KABB+23] on exploiting adjoints in IC3/PDR strengthens this conjecture.

6.3.2 Related Work

IC3 for finite-state transition system has been proposed by Bradley [Bra11a]. The improved algorithm PDR is due to Een et al. [EMB11]. IC3/PDR as has been extensively studied in the field of hardware verification [GI15; KVGG19; DKR10; VG14; HBS13; GSV14; GR16] and extended and adapted to infinite-state software model checking [Lan18; CGMT16; LNNK20b; LNN15; Wel13; LS16; CG12; CGMT14; BBW14; KBGM15; KGC14; BG15], hybrid systems [HB12; SI20], and generalized Petri nets [ADH22]. In particular, Hoder and Bjørner propose *generalized property directed reachability* [HB12] for non-linear predicate transformers where counterexamples unfold into trees rather than single finite execution fragments, thus encountering a problem similar to our Challenge 2.

To the best of our knowledge, PrIC3 was the first truly probabilistic generalization of IC3. Polgreen et al. propose pIC3 [PBFA19], where the IC3-like part of the algorithm reasons about *qualitative* reachability in Markov chains. Most closely related to PrIC3 are the recent works by Kori et al. discussed below.

LT-PDR [KUKS+22]. Our goal was to conservatively extend IC3 for transition systems to the more general quantitative setting of Markov decision processes. Kori et al. went one step further: They have generalized IC3/PDR to the even more general setting of verifying or refuting upper bounds on least fixpoints of continuous functions over complete lattices, coining their framework LT-PDR. From a conceptual point of view, this is similar to our generalization of classical k-induction for transition system to the latticed setting from Chapter 4.

Kori et al. instantiate their framework for reasoning about safety of Markov decision processes. This instance is called $\text{PDR}^{\text{IB-MDP}}$, which is compared — both theoretically and empirically — to our PrIC3 algorithm. As pointed out in [KUKS+22]: “$\text{PDR}^{\text{IB-MDP}}$ shares many essences with PrIC3 ”. Indeed, apart from minor optimizations, our core $\text{PrIC3}_{\mathcal{H}}$ loop can be considered an instance of Kori et al.’s framework with two key differences: First, as pointed out in [KUKS+22], proof obligations in our $\mathsf{Strengthen}_{\mathcal{H}}$ procedure never *directly* refute safety of the given MDP. Rather, we check for refutation in our separate outer PrIC3 loop. Kori et al., on the other hand, show how (their analogs of) proof obligations *can* be used to refute safety, which is more similar to how IC3 refutes safety of transition systems. Second $\mathsf{PDR}^{\mathsf{IB\text{-}MDP}}$ adapts an *optimized relative inductivity check* from IC3. Put more formally, the condition

$$\Phi(\mathcal{F}_i) \sqsubseteq \mathcal{F}$$

from Theorem 6.9 on page 236 is replaced by the *weaker*[26] condition

$$\Phi(\mathcal{F}_i \sqcap \mathcal{F}) \sqsubseteq \mathcal{F} .$$

This optimization is inessential to soundness or termination. It would, however, be interesting to investigate under which circumstances this optimized check yields a better performance. Finally, Kori et al. provide a further instantiation of their framework to obtain an algorithm for reasoning about upper bounds on more general *expected accumulated rewards* in Markov chains.

AdjointPDR. In [KABB⁺23], Kori et al. enhance their framework by exploiting *adjoints* — a well-studied notion from category theory [Car08] and abstract interpretation [Cou21]. In particular, they show how adjoints enable the derivation of — phrased in our terminology — *improved adequate heuristics*. These heuristics have been implemented and empirically compared to PrIC3. The results are strongly encouraging: *Without* generalization, their implementation outperforms PrIC3 on all benchmarks, indicating that their derived heuristics indeed advance the state-of-the-art which moreover enables further research on, e.g., obtaining suitable generalizations.

[26]Notice that $\Phi(\mathcal{F}_i \sqcap \mathcal{F}) \sqsubseteq \Phi(\mathcal{F}_i)$ by monotonicity of Φ.

7 Conclusion and Outlook

We have studied both foundational and practical aspects of the automated deductive verification of probabilistic programs via weakest preexpectation reasoning. We now draw some final conclusions and provide a brief outlook.

On the more foundational side, we have presented an expressive formal language Exp of expectations in Chapter 3. Our language Exp is nowadays a key ingredient of the modern semi-automated deductive probabilistic program verifier Caesar [18]. Caesar benefits from the fact that reasoning with Exp enables the relatively complete verification of probabilistic programs as discussed in Section 3.5. Moreover, the empirical results presented in [18] indicate that the semi-automated verification with Exp is feasible: Caesar verifies various specifications of programs taken from the literature. Hence, we conclude:

The results from Chapter 3 provide a principled foundation for the automated deductive verification of probabilistic programs.

On the more practical side, we have presented three different approaches for the fully automatic verification of piecewise linear bounds on expected outcomes and expected runtimes of linear loops. The empirical results are promising: Our approaches automatically verify various loops taken from the literature with both finite and infinite state spaces. A particularly promising application of automatic wp-reasoning is *probabilistic model checking*: Our CEGIS approach (Chapter 5) and PrIC3 (Chapter 6) verify huge probabilistic models for which the state-of-the-art probabilistic model checker Storm fails. This is due to the symbolic nature of our approaches, which can obviate the need for state space enumeration. Even though it is important to note that the models we have considered are structurally rather simple when compared to, e.g., many models from the Quantitative Verification Benchmark Set[1], we may conclude:

The results from Chapters 4 to 6 advance the state-of-the-art of the automatic verification of probabilistic programs. Moreover, we identify automatic wp*-reasoning as a promising direction for complementing probabilistic model checking.*

[1] https://qcomp.org/

Outlook. In the preceding chapters, we have already discussed several directions for future work. Let us take a final and more unifying perspective on possible directions for future work.

Nondeterminism. In Chapter 3, in our implementation of k-induction and BMC from Chapter 4, in Chapter 5, and in our implementation of PrIC3 from Chapter 6, we have restricted to fully probabilistic programs that do not contain nondeterministic choices. Extending the presented results to nondeterministic probabilistic programs is a promising direction for future work. This has partly been done: The semi-automated deductive verifier Caesar supports k-induction and BMC for possibly nondeterministic programs.

A particularly promising direction is to extend our CEGIS framework from Chapter 5 to nondeterministic programs. We have shown in [19] that — loosely speaking — quantitative invariants of nondeterministic probabilistic loops yield *specification-preserving program-level strategies* which resolve the nondeterminism in a program with probabilistic safety or liveness guarantees. Hence, automatic quantitative invariant synthesis also yields *automatic strategy synthesis for possibly infinite-state loops*. The synthesis of strategies for resolving the nondeterminism has various applications in planning and artificial intelligence.

Probabilistic programming and conditioning. In probabilistic programming, the main task is probabilistic inference: Given a (conditional) probability distribution encoded as a probabilistic programs, compute, e.g., the (conditional) expected value of a given random variable. There are dedicated weakest preexpectation-style calculi for reasoning about conditional expected outcomes of probabilistic programs [OGJK+18]. We consider it a promising direction to extend our results for tackling the aforementioned task, which might yield automatic quantitative loop invariant-based probabilistic inference techniques.

Combining our approaches. Finally, we have already outlined in Section 6.3 that, e.g., classical k-induction for transition systems has been combined with IC3/PDR [KVGG19]. We hence consider it a promising direction is to investigate how the strengths of our approaches from Chapters 4 to 6 can be combined to further improve automatic wp-reasoning.

Declaration of Authorship

Eidesstattliche Erklärung

I, Kevin Stefan Batz 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 research.

Hiermit erkläre ich an Eides statt / I do solemnely swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before as detailed in Section 1.5.

Kevin Stefan Batz, January 2025, Aachen

A Appendix

1 Appendix to Chapter 2

1.1 Appendix to Section 2.2

We consider Markov chains induced by memorlyess schedulers[1].

Definition A.1 (Markov Chains Induced by Memoryless Schedulers).
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathfrak{S} \in \mathsf{MLScheds}$ be a memoryless scheduler. The *Markov chain of $\mathcal{M}$ induced by $\mathfrak{S}$* is defined as

$$\mathcal{M}(\mathfrak{S}) \;=\; (\mathcal{S}, \mathsf{Act}, P_{\mathfrak{S}})\,,$$

where $P_{\mathfrak{S}}$ is the transition probability function given by

$$P_{\mathfrak{S}}(s, \mathfrak{a}, s') \;=\; \begin{cases} P(s, \mathfrak{a}, s') & \text{if } \mathfrak{a} = \mathfrak{S}(s) \\ 0, & \text{otherwise}\,. \end{cases}$$

We require the following auxiliary result: The least fixpoint of the *min*-Bellman operator ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ gives rise to a memoryless scheduler $\mathfrak{S}$ such that the least fixpoint of ${}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}$ and the least fixpoint of the Bellman operator ${}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}}$ w.r.t. the Markov chain $\mathcal{M}(\mathfrak{S})$ induced by $\mathfrak{S}$ coincide:

Lemma A.1.
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP, let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$ be a reward function, and let $\preccurlyeq$ be some total order on Act. Moreover, define the memoryless scheduler $\mathfrak{S}$ as

$$\mathfrak{S}(s) \;=\; \begin{cases} \mathfrak{a} \in \mathsf{Act}(s) & \text{if } s \in \mathcal{T} \\ \operatorname*{argmin}_{\alpha \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot \left(\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\right)(s') & \text{otherwise}\,. \end{cases}$$

where in case the above definition does not yield a unique action, we choose the least action w.r.t. $\preccurlyeq$. Then, for the Markov chain $\mathcal{M}(\mathfrak{S})$ induced by $\mathfrak{S}$,

$$\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \mathsf{lfp}\ {}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}}\,.$$

Proof. The proof is inspired by the techniques presented in [19]. We prove

[1] Restricting to memorlyess schedulers suffices for our purposes. One can more generally define Markov chains induced by history-dependent schedulers [BK08, Definition 10.92].

the two inequalities

$$\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \sqsubseteq \mathsf{lfp}\ {}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}} \quad \text{and} \quad \mathsf{lfp}\ {}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}} \sqsubseteq \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}~.$$

The claim then follows from antisymmetry of $\sqsubseteq$. The first inequality is straightforward. For the second inequality, by Park induction (Lemma 2.4), it suffices to show

$${}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}}\left(\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\right) \sqsubseteq \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}~. \qquad \blacksquare$$

For that, we reason as follows:

$$\begin{aligned}
&{}_{\mathcal{M}(\mathfrak{S})}\Phi_{\mathsf{rew}}\left(\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\right)\\
={}& \lambda s.\begin{cases}\mathsf{rew}(s) & \text{if } s \in \mathcal{T}\\ \displaystyle\sum_{s' \in \mathsf{Succs}_{\mathfrak{S}(s)}(s)} P(s, \mathfrak{S}(s), s') \cdot \left(\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\right)(s') & \text{otherwise}\end{cases}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(Definition 2.15)}\\
={}& \lambda s.\begin{cases}\mathsf{rew}(s) & \text{if } s \in \mathcal{T}\\ \displaystyle\min_{\mathfrak{a} \in \mathsf{Act}(s)} \sum_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot \left(\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}\right)(s') & \text{otherwise}~.\end{cases}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(by construction of } \mathfrak{S}\text{)}\\
={}& \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \qquad\qquad\qquad\qquad \text{(Definition 2.15.1 and fixpoint property)}\\
\sqsubseteq{}& \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}~. \qquad\qquad\qquad\qquad\qquad\qquad \text{(reflexivity of } \sqsubseteq\text{)}
\end{aligned}$$

Lemma A.2.
Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP, $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$ be a reward function, $\mathfrak{S} \in \mathsf{Scheds}$, and let $s \in \mathcal{S}$. We have:

1. $$\mathsf{MinER}(\mathcal{M}, s \models \Diamond \mathsf{rew}) = \inf_{\mathfrak{S} \in \mathsf{Scheds}} \sup_{n \in \mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n} \mathsf{rew}\right)$$

2. $$\mathsf{MaxER}(\mathcal{M}, s \models \Diamond \mathsf{rew}) = \sup_{\mathfrak{S} \in \mathsf{Scheds}} \sup_{n \in \mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n} \mathsf{rew}\right)$$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. $$\mathsf{ER}(\mathcal{M}, s \models \Diamond \mathsf{rew}) = \sup_{n \in \mathbb{N}} \mathsf{ER}\left(\mathcal{M}, s \models \Diamond^{\leq n} \mathsf{rew}\right)~.$$

Proof. It suffices to show that

$$\mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond \mathsf{rew}\right) \;=\; \sup_{n \in \mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n} \mathsf{rew}\right) .$$

We have

$$\mathsf{Paths}(s, \mathcal{T}) \;=\; \bigcup_{n \in \mathbb{N}} \mathsf{Paths}^{\leq n}(s, \mathcal{T}) .$$

Hence, the set

$$\begin{aligned}&\left\{\mathsf{ER}\left(\mathcal{M}, s \models \Diamond^{\leq n} \mathsf{rew}\right) \;\middle|\; n \in \mathbb{N}\right\}\\ =\; &\left\{ \sum_{s_0 \ldots s_m \in \mathsf{Paths}^{\leq n}(s, \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m) \cdot \mathsf{rew}(s_m) \;\middle|\; n \in \mathbb{N} \right\}\end{aligned}$$

covers all partial sums of

$$\mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond \mathsf{rew}\right) \;=\; \sum_{s_0 \ldots s_m \in \mathsf{Paths}(s, \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m) \cdot \mathsf{rew}(s_m) .$$

■

1.1.1 Proof of Lemma 2.8

We have to show Lemma 2.8 on page 42:

Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and let $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}_{\geq 0}^{\infty}$. We have:

1. for all $n \in \mathbb{N}$: $\quad {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}}^{n+1}(0) \;=\; \lambda s.\, \mathsf{MinER}\left(s \models \Diamond^{\leq n} \mathsf{rew}\right)$

2. for all $n \in \mathbb{N}$: $\quad {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}}^{n+1}(0) \;=\; \lambda s.\, \mathsf{MaxER}\left(s \models \Diamond^{\leq n} \mathsf{rew}\right)$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. for all $n \in \mathbb{N}$: $\quad {}_{\mathcal{M}}\Phi_{\mathsf{rew}}^{n+1}(0) \;=\; \lambda s.\, \mathsf{ER}\left(s \models \Diamond^{\leq n} \mathsf{rew}\right)$.

Proof. We prove the claim for the min-Bellman operator by induction on n. The proof for the max-Bellman operator is completely analogous.

Base case $n = 0$. We have

$$\begin{aligned}
&{}^{\min}_{\mathcal{M}}\Phi^{1}_{\mathsf{rew}}(0) \\
= {}& \lambda s. \begin{cases} \mathsf{rew}(s) & \text{if } s \in \mathcal{T} \\ \min\limits_{\mathfrak{a} \in \mathsf{Act}(s)} \sum\limits_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot 0 & \text{otherwise} \end{cases} && \text{(Definition 2.15.1)} \\
= {}& \lambda s. \begin{cases} \mathsf{rew}(s) & \text{if } s \in \mathcal{T} \\ 0 & \text{otherwise} \end{cases} \\
= {}& \lambda s. \inf_{\mathfrak{S} \in \mathsf{Scheds}} \sum_{s_0 \in \mathsf{Paths}^{\leq 0}(s,\mathcal{T})} \underbrace{\mathsf{Prob}_{\mathfrak{S}}(s_0)}_{=1} \cdot \mathsf{rew}(s_0) && (\mathsf{Paths}^{\leq 0}(s, \mathcal{T}) = \{s\} \cap \mathcal{T}) \\
= {}& \lambda s. \inf_{\mathfrak{S} \in \mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq 0}\mathsf{rew}\right). && \text{(Definition 2.13.3)}
\end{aligned}$$

Induction step. Let $s \in \mathcal{S}$. If $s \in \mathcal{T}$, then

$$ {}^{\min}_{\mathcal{M}}\Phi^{n+2}_{\mathsf{rew}}(0)(s) \;=\; \mathsf{rew}(s) \;=\; \mathsf{MinER}\left(\mathcal{M}, s \models \Diamond^{\leq n+1}\mathsf{rew}\right) $$

follows immediately. For $s \notin \mathcal{T}$, consider the following: Given a scheduler $\mathfrak{S}$ and some action $\mathfrak{a} \in \mathsf{Act}(s)$, define the scheduler $(s, \mathfrak{a}) \cdot \mathfrak{S}$ as

$$ ((s, \mathfrak{a}) \cdot \mathfrak{S})(s_0 \ldots s_m) \;=\; \begin{cases} \mathfrak{a} & \text{if } s_0 \ldots s_m = s \\ \mathfrak{S}(s_0 \ldots s_m) & \text{otherwise}. \end{cases} $$

■

$$\begin{aligned}
&{}^{\min}_{\mathcal{M}}\Phi^{n+2}_{\mathsf{rew}}(0)(s) \\
= {}& \min_{\mathfrak{a} \in \mathsf{Act}(s)} \sum_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot {}^{\min}_{\mathcal{M}}\Phi^{n+1}_{\mathsf{rew}}(0)(s') && \text{(Definition 2.15.1)} \\
= {}& \min_{\mathfrak{a} \in \mathsf{Act}(s)} \sum_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \cdot \mathsf{MinER}\left(\mathcal{M}, s' \models \Diamond^{\leq n}\mathsf{rew}\right) && \text{(I.H.)} \\
= {}& \min_{\mathfrak{a} \in \mathsf{Act}(s)} \sum_{s' \in \mathsf{Succs}_{\mathfrak{a}}(s)} P(s, \mathfrak{a}, s') \\
& \cdot \left(\inf_{\mathfrak{S} \in \mathsf{Scheds}} \sum_{s_0 \ldots s_m \in \mathsf{Paths}^{\leq n}(s', \mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0 \ldots s_m) \cdot \mathsf{rew}(s_m) \right) \\
& && \text{(Definition 2.13.3)}
\end{aligned}$$

$$
\begin{aligned}
&= \min_{\mathfrak{a}\in\mathsf{Act}(s)} \sum_{s'\in\mathsf{Succs}_{\mathfrak{a}}(s)} \\
&\qquad \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{s_0\ldots s_m\in\mathsf{Paths}^{\leq n}(s',\mathcal{T})} P(s,\mathfrak{a},s')\cdot\mathsf{Prob}_{\mathfrak{S}}(s_0\ldots s_m)\cdot\mathsf{rew}(s_m) \\
&\hfill \text{($\cdot$ distributes over inf and finite sums)} \\
&= \min_{\mathfrak{a}\in\mathsf{Act}(s)} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{s'\in\mathsf{Succs}_{\mathfrak{a}}(s)} \\
&\qquad \sum_{s_0\ldots s_m\in\mathsf{Paths}^{\leq n}(s',\mathcal{T})} P(s,\mathfrak{a},s')\cdot\mathsf{Prob}_{\mathfrak{S}}(s_0\ldots s_m)\cdot\mathsf{rew}(s_m) \\
&\hfill \text{(the sets $\mathsf{Paths}^{\leq n}(s',\mathcal{T})$ are pairwise disjoint)} \\
&= \min_{\mathfrak{a}\in\mathsf{Act}(s)} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{s_0\ldots s_m\in\mathsf{Paths}^{\leq n+1}(s,\mathcal{T})} \mathsf{Prob}_{(s,\mathfrak{a})\cdot\mathfrak{S}}(s_0\ldots s_m)\cdot\mathsf{rew}(s_m) \\
&\hfill \text{(de-factorize $\mathfrak{a}$-successors)} \\
&= \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sum_{s_0\ldots s_m\in\mathsf{Paths}^{\leq n+1}(s,\mathcal{T})} \mathsf{Prob}_{\mathfrak{S}}(s_0\ldots s_m)\cdot\mathsf{rew}(s_m) \\
&\hfill \text{(schedulers cover minimum over $\mathfrak{a}$-successors)} \\
&= \mathsf{MinER}\big(\mathcal{M}, s \models \Diamond^{\leq n+1}\mathsf{rew}\big)\ .
\end{aligned}
$$

1.1.2 Proof of Theorem 2.9

We have to show Theorem 2.9 on page 42:

Let $\mathcal{M} = (\mathcal{S}, \mathsf{Act}, P)$ be an MDP and $\mathsf{rew}\colon \mathcal{T} \to \mathbb{R}^{\infty}_{\geq 0}$. We have:

1. $\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\,\mathsf{MinER}(\mathcal{M}, s \models \Diamond\mathsf{rew})$
2. $\mathsf{lfp}\ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\,\mathsf{MaxER}(\mathcal{M}, s \models \Diamond\mathsf{rew})$

Moreover, if $\mathcal{M}$ is a Markov chain, then

3. $\mathsf{lfp}\ {}_{\mathcal{M}}\Phi_{\mathsf{rew}} \;=\; \lambda s.\,\mathsf{ER}(\mathcal{M}, s \models \Diamond\mathsf{rew})\ .$

Proof. Theorem 2.9.3 follows from Theorem 2.9.1 (or Theorem 2.9.2). Since the proof of Theorem 2.9.1 is more involved, we first prove Theorem 2.9.2 .

For that, consider the following:

$$
\begin{aligned}
&\mathsf{lfp}\ {}^{\max}_{\mathcal{M}}\Phi_{\mathsf{rew}} \\
&= \bigsqcup_{n\in\mathbb{N}} {}^{\max}_{\mathcal{M}}\Phi^{n}_{\mathsf{rew}}(0) && \text{(Theorem 2.7.2)} \\
&= \bigsqcup_{n\in\mathbb{N}} \lambda s.\,\mathsf{MaxER}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(Lemma 2.8.2)} \\
&= \bigsqcup_{n\in\mathbb{N}} \lambda s.\ \sup_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(Definition 2.13.5)} \\
&= \lambda s.\,\sup_{n\in\mathbb{N}}\ \sup_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(suprema defined pointwise)} \\
&= \lambda s.\ \sup_{\mathfrak{S}\in\mathsf{Scheds}}\ \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(suprema commute)} \\
&= \lambda s.\,\mathsf{MaxER}(\mathcal{M}, s \models \lozenge\mathsf{rew})\ . && \text{(Lemma A.2.2)}
\end{aligned}
$$

Recall that this already implies Theorem 2.9.3. In the above reasoning, we have exploited that suprema commute. This step is what yields the proof of Theorem 2.9.1 to be more involved. In this proof, we encounter an analogous situation but instead of swapping two suprema we have to swap suprema and infima. Suprema and infima do generally not commute. We will, however, employ Lemma A.1 to see that the suprema and infima considered in this proof *do* commute. Now consider the following:

$$
\begin{aligned}
&\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \\
&= \bigsqcup_{n\in\mathbb{N}} {}^{\min}_{\mathcal{M}}\Phi^{n}_{\mathsf{rew}}(0) && \text{(Theorem 2.7.2)} \\
&= \bigsqcup_{n\in\mathbb{N}} \lambda s.\,\mathsf{MinER}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(Lemma 2.8.1)} \\
&= \bigsqcup_{n\in\mathbb{N}} \lambda s.\ \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(Definition 2.13.3)} \\
&= \lambda s.\,\sup_{n\in\mathbb{N}}\ \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\big(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\big) && \text{(suprema defined pointwise)}
\end{aligned}
$$

Assume for the moment that for all $s \in \mathcal{S}$, we have

$$\sup_{n\in\mathbb{N}} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) = \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \tag{1}$$

With this assumption and the above reasoning, we obtain the desired claim:

$$\begin{aligned}
&\mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} \\
=\ &\lambda s.\ \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \\
&\hfill \text{(above reasoning and assumption)} \\
=\ &\lambda s.\ \mathsf{MinER}(\mathcal{M}, s \models \lozenge\mathsf{rew})\ . \qquad \text{(Lemma A.2.1)}
\end{aligned}$$

It hence remains to prove Equation (1). For that, we prove that both

$$\sup_{n\in\mathbb{N}} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \leq \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \tag{2}$$

and

$$\sup_{n\in\mathbb{N}} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \geq \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right)\ . \tag{3}$$

The claim then follows from antisymmetry of $\leq$. Inequality 2 holds since $\sup\inf\ldots \leq \inf\sup\ldots$ holds in every complete lattice. For Inequality 3, observe that for every scheduler $\mathfrak{S}'$,

$$\sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}'}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \geq \inf_{\mathfrak{S}\in\mathsf{Scheds}} \sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right)\ .$$

Hence, the claim follows if there is a scheduler $\mathfrak{S}'$ with

$$\sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}'}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) = \sup_{n\in\mathbb{N}} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right)\ .$$

We claim that $\mathfrak{S}'$ is given by the scheduler constructed in Lemma A.1, since

$$\begin{aligned}
&\sup_{n\in\mathbb{N}} \mathsf{ER}_{\mathfrak{S}'}\left(\mathcal{M}, s \models \lozenge^{\leq n}\mathsf{rew}\right) \\
=\ &\sup_{n\in\mathbb{N}} \mathsf{ER}\left(\mathcal{M}(\mathfrak{S}'), s \models \lozenge^{\leq n}\mathsf{rew}\right) \qquad \text{(Definition A.1)}
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{ER}(\mathcal{M}(\mathfrak{S}'), s \models \Diamond \mathsf{rew}) && \text{(Lemma A.2.3)}\\
&= \mathsf{lfp}\ {}_{\mathcal{M}(\mathfrak{S}')}\Phi_{\mathsf{rew}} && \text{(Theorem 2.9.3)}\\
&= \mathsf{lfp}\ {}^{\min}_{\mathcal{M}}\Phi_{\mathsf{rew}} && \text{(Lemma A.1)}\\
&= \sup_{n\in\mathbb{N}} {}^{\min}_{\mathcal{M}}\Phi^{n}_{\mathsf{rew}}(0) && \text{(Theorem 2.2)}\\
&= \sup_{n\in\mathbb{N}} \inf_{\mathfrak{S}\in\mathsf{Scheds}} \mathsf{ER}_{\mathfrak{S}}\left(\mathcal{M}, s \models \Diamond^{\leq n}\mathsf{rew}\right). && \text{(Lemma 2.8.1)}
\end{aligned}
$$

This completes the proof. ■

1.2 Appendix to Section 2.3

Lemma A.3.
Let $C \in \mathsf{pGCL}$ and $X \in \mathbb{E}$. We have:

1.

$$
\begin{aligned}
&\mathsf{dwp}[\![C]\!](X)\\
&= \lambda\sigma.\ \min_{\mathfrak{a}\in\mathsf{Act}(C,\sigma)} \sum_{(C,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow, \tau)\\ \mathsf{dwp}[\![C']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C', \sigma')\end{cases}
\end{aligned}
$$

2.

$$
\begin{aligned}
&\mathsf{awp}[\![C]\!](X)\\
&= \lambda\sigma.\ \max_{\mathfrak{a}\in\mathsf{Act}(C,\sigma)} \sum_{(C,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow, \tau)\\ \mathsf{awp}[\![C']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C', \sigma')\end{cases}
\end{aligned}
$$

Proof. By induction on C. We prove the claim for dwp. The reasoning for awp is completely analogous. Now let $\sigma \in \mathsf{States}$.

The case $C = \mathtt{skip}$. We have

$$
\begin{aligned}
&\mathsf{dwp}[\![\mathtt{skip}]\!](X)(\sigma)\\
&= X(\sigma) && \text{(Table 2.1)}
\end{aligned}
$$

$$= \min_{\mathfrak{a}\in \mathsf{Act}(C,\sigma)} \sum_{(C,\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C',\sigma') . \end{cases}$$

$$(\mathsf{Act}(\mathtt{skip},\sigma) = \{N\} \text{ and } (\mathtt{skip},\sigma) \xrightarrow{N,1} (\Downarrow,\sigma) \text{ by Figure 2.9})$$

The case $C = x := A$. We have

$$\begin{aligned} & \mathsf{dwp}[\![x := A]\!](X)(\sigma) \\ = \;& X(\sigma[x \mapsto A(\sigma)]) && \text{(Table 2.1)} \\ = \;& \min_{\mathfrak{a}\in \mathsf{Act}(C,\sigma)} \sum_{(C,\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C',\sigma') . \end{cases} \end{aligned}$$

$$(\mathsf{Act}(x := A,\sigma) = \{N\} \text{ and } (x := A,\sigma) \xrightarrow{N,1} (\Downarrow,\sigma[x \mapsto A(\sigma)]) \text{ by Figure 2.9})$$

The case $C = C_1 ; C_2$. First observe that by the rules in Figure 2.9 we encounter exactly one of the following cases:

1. We have $\mathsf{Act}(C_1,\sigma) = \{N\}$ and $C_1,\sigma \xrightarrow{N,1} \Downarrow,\tau$. Hence, also $\mathsf{Act}(C_1 ; C_2,\sigma) = \{N\}$ and $(C_1 ; C_2,\sigma) \xrightarrow{N,1} (C_2,\tau)$.
2. For every $\mathfrak{a} \in \mathsf{Act}(C_1,\sigma)$, all configurations in $\mathsf{Succs}_{\mathfrak{a}}(C_1,\sigma)$ are of the form (C_1',σ'). Hence, for every $\mathfrak{a} \in \mathsf{Act}(C_1 ; C_2,\sigma)$, all configurations in $\mathsf{Succs}_{\mathfrak{a}}(C_1 ; C_2,\sigma)$ are of the form $(C_1' ; C_2,\sigma')$.

We proceed by distinguishing these two cases. For the first case, we have

$$\begin{aligned} & \mathsf{dwp}[\![C_1 ; C_2]\!](X)(\sigma) \\ = \;& \mathsf{dwp}[\![C_1]\!](\mathsf{dwp}[\![C_2]\!](X))(\sigma) && \text{(Table 2.1)} \\ = \;& \min_{\mathfrak{a}\in \mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} \mathsf{dwp}[\![C_2]\!](X)(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C']\!](\mathsf{dwp}[\![C_2]\!](X))(\sigma') & \text{if } \mathfrak{c}' = (C',\sigma') . \end{cases} \\ & && \text{(I.H.)} \\ = \;& \mathsf{dwp}[\![C_2]\!](X)(\tau) && \text{(assumption)} \\ = \;& \min_{\mathfrak{a}\in \mathsf{Act}(C_1 ; C_2,\sigma)} \sum_{(C_1 ; C_2,\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C',\sigma') . \end{cases} \\ & && \text{(above reasoning for the first case)} \end{aligned}$$

For the second case, we have

$$\begin{aligned}
&\mathsf{dwp}[\![C_1\,;\,C_2]\!]\,(X)\,(\sigma)\\
=\;& \mathsf{dwp}[\![C_1]\!]\,(\mathsf{dwp}[\![C_2]\!]\,(X))\,(\sigma) && \text{(Table 2.1)}\\
=\;& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} p\cdot\begin{cases}\mathsf{dwp}[\![C_2]\!]\,(X)\,(\tau) & \text{if } \mathfrak{c}'=(\Downarrow,\tau)\\ \mathsf{dwp}[\![C']\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}'=(C',\sigma')\,.\end{cases} && \text{(I.H.)}\\
=\;& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}(C_1',\sigma')} p\cdot\mathsf{dwp}[\![C_1']\!]\,(\mathsf{dwp}[\![C_2]\!]\,(X))\,(\sigma')\\
& && \text{(assumption)}\\
=\;& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}(C_1',\sigma')} p\cdot\mathsf{dwp}[\![C_1'\,;\,C_2]\!]\,(X)\,(\sigma') && \text{(Table 2.1)}\\
=\;& \min_{\mathfrak{a}\in\mathsf{Act}(C_1\,;\,C_2,\sigma)} \sum_{(C_1\,;\,C_2,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} p\cdot\begin{cases}X(\tau) & \text{if } \mathfrak{c}'=(\Downarrow,\tau)\\ \mathsf{dwp}[\![C']\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}'=(C',\sigma')\,.\end{cases}\\
& && \text{(above reasoning for the second case)}
\end{aligned}$$

The cases $C = \{C_1\}\,[p]\,\{C_2\}$, $C = \{C_1\}\,\Box\,\{C_2\}$, and $C = \texttt{if}\,(B)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ follow immediately from the I.H.

The case $C = \texttt{while}(B)\{C'\}$. Since $\mathsf{dwp}[\![\texttt{while}(B)\{C'\}]\!]\,(X)$ is a fixpoint of ${}^{\mathsf{dwp}}_{C}\Phi_X$, we have

$$\mathsf{dwp}[\![\texttt{while}(B)\{C'\}]\!]\,(X) \;=\; [B]\cdot\mathsf{dwp}[\![C';\texttt{while}(B)\{C'\}]\!]\,(X)+[\neg B]\cdot X\,. \tag{4}$$

Now distinguish the cases $\sigma\models B$ and $\sigma\models\neg B$. If $\sigma\models\neg B$, then

$$\begin{aligned}
&\mathsf{dwp}[\![\texttt{while}(B)\{C'\}]\!]\,(X)\,(\sigma)\\
=\;& X(\sigma) && \text{(Equation (4))}\\
=\;& \min_{\mathfrak{a}\in\mathsf{Act}(\texttt{while}(B)\{C'\},\sigma)}
\end{aligned}$$

$$\sum_{(\mathtt{while}(B)\{C'\},\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C'']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C'',\sigma')\,, \end{cases}$$

(see below)

where the last step holds since $\sigma \not\models B$ implies $\mathsf{Act}(\mathtt{while}(B)\{C'\},\sigma) = \{N\}$ and $(\mathtt{while}(B)\{C'\},\sigma) \xrightarrow{N,1} (\Downarrow,\sigma)$ by Figure 2.10.
If $\sigma \models B$, then

$$\begin{aligned} & \min_{\mathfrak{a}\in\mathsf{Act}(\mathtt{while}(B)\{C'\},\sigma)} \\ & \qquad \sum_{(\mathtt{while}(B)\{C'\},\sigma) \xrightarrow{\mathfrak{a},p} \mathfrak{c}'} p \cdot \begin{cases} X(\tau) & \text{if } \mathfrak{c}' = (\Downarrow,\tau) \\ \mathsf{dwp}[\![C'']\!](X)(\sigma') & \text{if } \mathfrak{c}' = (C'',\sigma') \end{cases} \\ &= \mathsf{dwp}[\![C';\, \mathtt{while}(B)\{C'\}]\!](X)(\sigma) && \text{(see below)} \\ &= \mathsf{dwp}[\![\mathtt{while}(B)\{C'\}]\!](X)(\sigma)\,, && \text{(Equation (4))} \end{aligned}$$

where the one but last step holds since $\sigma \models B$ implies $\mathsf{Act}(\mathtt{while}(B)\{C'\},\sigma) = \{N\}$ and $(\mathtt{while}(B)\{C'\},\sigma) \xrightarrow{N,1} (C';\,\mathtt{while}(B)\{C'\},\sigma)$ by Figure 2.10. ■

Lemma A.4.
Let $\mathcal{T} \in \{\mathsf{dop},\mathsf{aop}\}$. For all $C_1, C_2 \in \mathsf{pGCL}$ and all $X \in \mathbb{E}$, we have

$$\mathcal{T}[\![C_1]\!](\mathcal{T}[\![C_2]\!](X)) \sqsubseteq \mathcal{T}[\![C_1\,;\,C_2]\!](X)\,.$$

Proof. We prove the claim for dop. The reasoning for aop is completely analogous. First notice that by Definition 2.23 on page 69 and Theorem 2.7 on page 41, we have for all $\sigma \in \mathsf{States}$,

$$\begin{aligned} & \mathsf{dop}[\![C_1]\!](\mathsf{dop}[\![C_2]\!](X))(\sigma) \\ &= \Big(\bigsqcup \Big\{ {}^{\min}_{\mathcal{O}}\Phi^{n}_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}(0) \;\Big|\; n \in \mathbb{N} \Big\}\Big)((C_1,\sigma))\,. \end{aligned}$$

It hence suffices to prove that for all $n \in \mathbb{N}$, $C_1 \in \mathsf{pGCL}$, $\sigma \in \mathsf{States}$, and $X \in \mathbb{E}$,

$${}^{\min}_{\mathcal{O}}\Phi^{n}_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}(0)((C_1,\sigma)) \;\leq\; \mathsf{dop}[\![C_1\,;\,C_2]\!](X)(\sigma)\,.$$ ■

We proceed by induction on n. The base case $n = 0$ is trivial. For the induction step, consider the following:

$$
\begin{aligned}
& {}^{\min}_{\mathcal{O}}\Phi^{n+1}_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}(0)((C_1,\sigma)) \\
={}& {}^{\min}_{\mathcal{O}}\Phi_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}\left({}^{\min}_{\mathcal{O}}\Phi^{n}_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}(0)\right)((C_1,\sigma)) \\
={}& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} \\
& \quad p\cdot\begin{cases}\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}(\mathfrak{c}') & \text{if } \mathfrak{c}'=(\Downarrow,\tau)\\ {}^{\min}_{\mathcal{O}}\Phi^{n}_{\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}}(0)((C',\sigma'))) & \text{if } \mathfrak{c}'=(C',\sigma')\end{cases} \\
& \hfill \text{(Definition 2.17 and Definition 2.15)} \\
\leq{}& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} \\
& \quad p\cdot\begin{cases}\mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}(\mathfrak{c}') & \text{if } \mathfrak{c}'=(\Downarrow,\tau)\\ \mathsf{dop}[\![C';\,C_2]\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}'=(C',\sigma')\end{cases} \qquad \text{(I.H.)} \\
={}& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}'} \\
& \quad p\cdot\begin{cases}\mathsf{dop}[\![C_2]\!]\,(X)\,(\tau) & \text{if } \mathfrak{c}'=(\Downarrow,\tau)\\ \mathsf{dop}[\![C';\,C_2]\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}'=(C',\sigma')\,.\end{cases} \\
& \hfill \text{(definition of } \mathsf{rew}_{\mathsf{dop}[\![C_2]\!](X)}\text{)} \\
={}& \min_{\mathfrak{a}\in\mathsf{Act}(C_1,\sigma)} \sum_{(C_1;\,C_2,\sigma)\xrightarrow{\mathfrak{a},p}\mathfrak{c}''} \\
& \quad p\cdot\begin{cases}X(\tau) & \text{if } \mathfrak{c}''=(\Downarrow,\tau)\\ \mathsf{dop}[\![C'']\!]\,(X)\,(\sigma') & \text{if } \mathfrak{c}''=(C'',\sigma')\,.\end{cases} \qquad \text{(see below)} \\
={}& \mathsf{dop}[\![C_1;\,C_2]\!]\,(X)\,(\sigma)\,, \\
& \hfill \text{(for all } C \text{ and } \sigma,\ \mathsf{dop}[\![C]\!]\,(X)\,(\sigma) \ = \ (\mathsf{lfp}\ {}^{\min}_{\mathcal{O}}\Phi_X)((C,\sigma)))
\end{aligned}
$$

where the one but last step is an immediate consequence of the fact that by Figure 2.9 on page 48, we have

$$\mathfrak{c}'=(\Downarrow,\tau) \text{ iff } \mathfrak{c}''=(C_2,\tau) \qquad \text{and} \qquad \mathfrak{c}'=(C',\sigma') \text{ iff } \mathfrak{c}''=(C';\,C_2,\sigma')\,.$$

2 Appendix to Chapter 4

Lemma A.5 (Deciding Well-Definedness of Piecewise Linear Expectations). For all expressions f adhering to the grammar from Definition 4.4 on page 147, it is decidable whether it holds that

$$\text{for all } \sigma \in \mathsf{States}\colon\; [\![f]\!](\sigma) \geq 0\,,$$

i.e., whether $[\![f]\!]$ is a well-defined expectation in $\mathbb{E}$.

Proof. We construct a Boolean expression $\vartheta \in \mathsf{LBExpr}_{\mathbb{Q}}$ such that

$$\vartheta \text{ is unsatisfiable} \qquad \text{iff} \qquad [\![f]\!] \in \mathbb{E}\,.$$

This implies the claim since satisfiability of ϑ is decidable by Lemma 4.15 on page 148. For that, let

$$f \;=\; [\vartheta_1] \cdot \tilde{e}_1 + \ldots + [\vartheta_n] \cdot \tilde{e}_n\,.$$

The sought-after Boolean expression ϑ is given by

$$\bigvee_{\left((\rho_1,\tilde{a}_1),\ldots,(\rho_n,\tilde{a}_n)\right) \in \bigtimes_{i=1}^{n}\left\{(\vartheta_i,\tilde{e}_i),(\neg\vartheta_i,0)\right\}} \begin{cases} \mathsf{false} & \text{if } \tilde{a}_i = \infty \text{ for some } i \\ \left(\bigwedge_{i=1}^{n} \rho_i\right) \wedge \sum_{i=1}^{n} \tilde{a}_i < 0\,. & \text{otherwise} \end{cases}$$

■

		Storm		cegispro2												
					induct.-guided				static				dynamic			
Prog	$\lvert[\![\varphi]\!]\rvert$	sp	dd	best	$\lvert\text{States}'\rvert$	$\lvert I\rvert$	$t_s\%$	t	$\lvert\text{States}'\rvert$	$\lvert I\rvert$	$t_s\%$	t	$\lvert\text{States}'\rvert$	$\lvert I\rvert$	$t_s\%$	t
		MO	TO	3	33	10	40	3	–	–	–	TO	–	–	–	TO
boundedrwmultistep	$1 \cdot 10^5$	MO	TO	10	55	16	36	10	–	–	–	TO	–	–	–	TO
		MO	TO	–	–	–	–	TO	–	–	–	TO	–	–	–	TO
		MO	TO	11	56	23	40	11	70	10	35	18	–	–	–	TO
brp	$1 \cdot 10^{10}$	MO	TO	54	138	42	63	253	125	17	27	54	–	–	–	TO
		MO	TO	56	104	41	54	111	122	17	30	56	–	–	–	TO
		TO	TO	8	53	7	72	10	67	7	44	8	54	9	52	29
brpfinitefamily	$16 \cdot 10^{13}$	TO	TO	17	64	13	74	17	215	19	66	373	–	–	–	TO
		TO	TO	18	68	12	68	18	231	19	80	731	–	–	–	TO
		MO	TO	1	97	6	68	10	66	3	50	1	–	–	–	TO
chain	$1 \cdot 10^{12}$	MO	TO	24	–	–	–	TO	116	5	86	24	–	–	–	TO
		MO	TO	4933	–	–	–	TO	503	23	81	4933	–	–	–	TO
		MO	TO	9	156	7	71	29	156	7	71	29	81	7	49	9
chainselectstepsize	$3 \cdot 10^7$	MO	TO	96	–	–	–	TO	179	15	70	96	–	–	–	TO
		MO	TO	66	–	–	–	TO	164	15	58	66	–	–	–	TO
gridbig	$1 \cdot 10^6$	11	–	–	–	–	–	TO	–	–	–	TO	–	–	–	TO
gridsmall	$1 \cdot 10^2$	<1	32	1	15	7	36	1	46	10	37	3	20	5	39	2
		<1	32	2	26	11	35	2	77	17	32	10	71	10	82	59
		MO	TO	<1	7	3	22	<1	7	3	26	<1	7	3	31	<1
zeroconf	$1 \cdot 10^8$	MO	TO	<1	105	22	60	32	9	5	39	<1	156	7	92	215
		MO	TO	63	–	–	–	TO	173	23	59	63	–	–	–	TO
		TO	TO	2	48	3	49	2	48	3	49	2	59	3	52	3
zeroconffamily	$1 \cdot 10^{16}$	TO	TO	6	–	–	–	TO	77	9	56	6	252	13	81	854
		TO	TO	20	–	–	–	TO	99	9	70	20	164	13	73	265

Table 1: Storm vs. cegispro2. TO=2h, MO=8GB, time in seconds.

3 Appendix to Chapter 5

3.1 Details on the Comparison of Storm and cegispro2

Table 1 depicts the results for finite-state loops; the corresponding programs and specifications are given further below. We employ a *cooperative verifier* with $d = 2$. Column "Prog" depicts the name of the program and $\lvert[\![\varphi]\!]\rvert$ is the size of the relevant state space, i.e., the number of states satisfying the loop guard φ. "sp" and "dd" are the runtimes of Storm's sparse and decision diagram-based engines, respectively. "best" is the best runtime under all CEGIS configurations, which are then listed. We show the number $\lvert\text{States}'\rvert$ of counterexamples, the size $|I|$ of the inductive invariant (in terms of the number of summands), the fraction t_s% of time spent in the synthesizer, and the total time t.

Programs and Specifications. In the following,

$$[\vartheta] \implies \tilde{e} \quad \text{is a shorthand for} \quad [\vartheta]\cdot\tilde{e} + [\neg\vartheta]\cdot\infty\,.$$

boundedrwmultistep is the following program:

```
while($0 < x \wedge x < 200000 \wedge 1 \leq y \wedge y \leq 5$){
    {$x := x - 1$}
    [1/2]
    {
        if ($x = 1$){
            $y :=$ unif(1,5)
        }else{
            $x := x + y$
        }
    }
},
```

where $y :=$ `unif`$(1,5)$ is a shorthand for the program which samples uniformly from $\{1,\dots,5\}$ and assigns the result to y. The specifications are:

1. $\mathsf{wp}[\![C]\!]([x = 200000]) \sqsubseteq [x = 1] \implies 0.4$
2. $\mathsf{wp}[\![C]\!]([x = 200000]) \sqsubseteq [x = 1] \implies 0.3$
3. $\mathsf{wp}[\![C]\!]([x = 200000]) \sqsubseteq [x = 1] \implies 0.2$

brp is the following program:

```
while($sent < 8\,000\,000\,000 \wedge failed < 10$){
    {
        $failed := failed + 1$
    } [0.01] {
            $failed := 0; sent := sent + 1$
        }
    }
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 10]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.001$
2. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 10]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.0001$
3. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 10]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.00001$

brpfinitefamily is the following program:

```
    while($sent < N \land failed < 5 \land N < 8\,000\,000$){
        {
            $failed := failed + 1$
        } [0.01] {
            $failed := 0; sent := sent + 1$
        }
    }
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 5]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.05$
2. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 5]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.01$
3. $\mathsf{wp}[\![C]\!]([\mathit{failed} = 5]) \sqsubseteq [\mathit{failed} = 0 \land \mathit{sent} = 0] \implies 0.005$

chain is the following program:

```
    while($y = 0 \land x < 10^{16}$){
        {$y := 1$} [0.000000000001] {$x := x + 1$}
    }
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y = 1]) \sqsubseteq [y = 0 \land x = 0] \implies 0.8$

2. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [y=0 \land x=0] \implies 0.7$
3. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [y=0 \land x=0] \implies 0.641$

chainselectstepsize is the following program:

```
while$\left(y = 0 \land x < 10^7 \land z \leq 10\right)${
  if $(z = 0)${
    $z$ := unif$(1,10)$
  }else{
    if $(z \leq 2)${
        {$y$ := 1} [0.0000001] {$x$ := $x + z$}
    }else{
      if $(z \leq 4)${
        {$y$ := 1} [0.0000002] {$x$ := $x + z$}
      }else{
        if $(z \leq 6)${
          {$y$ := 1} [0.0000003] {$x$ := $x + z$}
        }else{
          if $(z \leq 8)${
            {$y$ := 1} [0.0000004] {$x$ := $x + z$}
          }else{
            {$y$ := 1} [0.0000005] {$x$ := $x + z$}
          }
        }
      }
    }
  }
}
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [y=0 \land x=0 \land z=0] \implies 0.7$
2. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [y=0 \land x=0 \land z=0] \implies 0.6$

3. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [y=0 \wedge x=0 \wedge z=0] \implies 0.55$

gridbig is the following program:

```
while($x < 1000 \wedge y < 1000$){
    {$x := x+1$} [$1/2$] {$y := y+1$}
}
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y=1000]) \sqsubseteq [x=0 \wedge y=0] \implies 0.99$

gridsmall is the following program:

```
while($x < 10 \wedge y < 10$){
    {$x := x+1$} [$1/2$] {$y := y+1$}
}
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y=10]) \sqsubseteq [x=0 \wedge y=0] \implies 0.8$
2. $\mathsf{wp}[\![C]\!]([y=10]) \sqsubseteq [x=0 \wedge y=0] \implies 0.7$

zeroconf is the following program:

```
while($\left(x < 10^8 \wedge y = 0 \wedge z \leq 1\right)${
    if ($z = 1$){
        {$z := 0$} [$1/2$] {$z := 0; y := 1$}
    }else{
        {$x := x+1$} [$0.999999999$] {$z := 1; x := 0$}
}
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.53$
2. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.526$

3. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.5251$

zeroconffamily is the following program:

```
while($x < N \wedge y = 0 \wedge z \leq 1 \wedge 10^8 \leq N \wedge N \leq 10^8$){
    if ($z = 1$){
        {$z := 0$} [$1/2$] {$z := 0; y := 1$}
    }else{
        {$x := x + 1$} [$0.999999999$] {$z := 1; x := 0$}
}
```

The specifications are:

1. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.555$
2. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.553$
3. $\mathsf{wp}[\![C]\!]([y=1]) \sqsubseteq [z=1 \wedge x=0 \wedge y=0] \implies 0.5251$

3.2 Details on the Comparison of Absynth and cegispro2

Table 2 depicts the results on the UPAST benchmarks with programs adapted from [NCH18]. We employ the baseline verifier from Theorem 5.5 on page 181. "Prog" is the name of the benchmark, t denotes the runtime in seconds required by the tools and "bound" indicates their respective synthesized bounds. |States′| depicts the number of counterexamples required by cegispro2.

3.3 Details on the Comparison of Exist and cegispro2

Tables 3 and 4 depicts the results on the wp-subinvariant benchmarks with programs and specifications adapted from [BTPH+22]. We employ the baseline verifier from Theorem 5.5 on page 181. "Prog" is the name of the benchmark, t denotes the runtime in seconds required by the tools, I denotes the wp-subinvariants computed by each tool, and |States′| depicts the number of counterexamples required by cegispro2.

	Absynth		cegispro2		
Prog	t	bound	$\|S'\|$	t	bound
bayesiannetwork	0.15	1 + 2 max(0, n)	1	2.99	(n * 2.0)
ber	≤0.01	1 + 2 max(0, n - x)	3	0.08	((x * -2.0) + (n * 3.0))
cowboyduel	≤0.01	1 + 1.2 max(0, flag)	1	0.06	11/5
C4Bt09	0.02	1 + max(0, -j + x)	15	0.37	max[((j * -1.0) + x + 1.0), ((j * -1.0) + x + 1.0)]
C4Bt13	0.02	1 + 2 max(0, x) + max(0, y)	8	0.25	((x * 2.0) + y)
C4Bt19	0.04	3 + max(0, 51 + i + k) + 2 max(0, i)	16	0.42	((i * 2.0) + k + -46.0)
C4Bt61	0.02	2 + max(0, l)	15	0.43	(l + -3.0)
condand	≤0.01	1 + 2 max(0, m)	4	0.09	((m * 2.0) + 1.0)
coupon	0.05	10 max(0, 5 - i)	5	0.25	max[149/12, 137/12, 61/6, 17/2, (i * 3/2)]
fcall	≤0.01	1 + 4 max(0, n - x)	3	0.09	((x * -3.0) + (n * 3.0))
fillingvol	0.09	1 + 0.666667 max(0, 10 - vM + vTF)	10	0.35	max[((vM * -1/36) + (vTF * 1/36) + 2.0), ((vM * -2/3) + (vTF * 2/3) + -583/180)]
geo	≤0.01	3	1	0.06	3.0
linear01	≤0.01	1 + 0.6 max(0, x)	1	0.06	x
trappedminer	0.03	1 + 11.5 max(0, -i + n)	72	3.58	((i * -3.0) + (n * 3.0) + 1.0)
noloop	≤0.01	2	1	0.06	2.0
prdwalk	0.05	1 + 0.571429 max(0, 4 + n - x)	6	0.27	((x * -4/7) + (n * 4/7) + 37/21)
prseq	0.04	0.65 max(0, x - y) + 0.35 max(0, y)	266	13.63	((x * 13/20) + (y * -1/2) + 47/20)
prspeed	0.04	1 + 2 max(0, m - y) + 0.666667 max(0, n - x)	18	0.41	max[((y * -2.0) + (m * 2.0) + (n * 2/3) + 1/3), ((x * -2/3) + (n * 2/3) + 1/3)]
race	0.17	1 + 0.666667 max(0, 9 - h + t)	32	1.95	((h * -2/3) + (t * 2/3) + 13/3)
rejectionsampling	4.03	1 + 5 max(0, n)	20	0.53	((n * 5.0) + 1.0)
rfindmc	≤0.01	1 + max(0, -i + k)	1	0.07	(k * 2.0)
rfindlv	≤0.01	1 + 2 max(0, flag)	1	0.05	3.0
rdseql	0.02	1 + 2.25 max(0, x) + max(0, y)	13	0.29	((x * 9/4) + y + -1/4)
rdspeed	0.03	1 + 2 max(0, m - y) + 0.666667 max(0, n - x)	18	0.44	max[((y * -2.0) + (m * 2.0) + (n * 2/3) + 1/3), ((x * -2/3) + (n * 2/3) + 1/3)]
sprdwalk	≤0.01	1 + 2 max(0, n - x)	3	0.08	((x * -3.0) + (n * 3.0))

Table 2: Absynth vs. cegispro2. TO=20min, MO=8GB, time in seconds.

	Exist		cegispro2		
Prog	t	I	|States′|	t	I
BiasDir1_0	78.31	$x+[x=y]\cdot(-0.5\cdot x+-0.5\cdot y+0.1)$	0	0.15	$[\varphi]\cdot 0+[\neg\varphi]\cdot x$
BiasDir1_1	97.29	$x+[x=y]\cdot(-0.5\cdot x+-0.5\cdot y+0.5)$	1	0.08	$[\varphi]\cdot 0.5+[\neg\varphi]\cdot x$
BiasDir2_0	66.42	$x+[x=y]\cdot(-0.5\cdot x+-0.5\cdot y+0.1)$	0	0.07	$[\varphi]\cdot 0+[\neg\varphi]\cdot x$
BiasDir2_1	171.83	$x+[x=y]\cdot(-0.5\cdot x+-0.5\cdot y+0.5)$	1	0.08	$[\varphi]\cdot 0.5+[\neg\varphi]\cdot x$
BiasDir3_0	66.55	$x+[x=y]\cdot(-0.3\cdot x+-0.3\cdot y)$	0	0.08	$[\varphi]\cdot 0+[\neg\varphi]\cdot x$
BiasDir3_1	99.23	$x+[x=y]\cdot(-0.5\cdot x+-0.5\cdot y+0.5)$	1	0.08	$[\varphi]\cdot 0.5+[\neg\varphi]\cdot x$
Bin01_0	53.19	$x+[n>0]\cdot 0$	1	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin02_0	83.04	$x+[n>0]\cdot 0$	1	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin03_0	53.13	$x+[n>0]\cdot 0$	1	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin11_0	32.52	$x+[n<M]\cdot 0$	1	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin11_1	TO	–	3	0.08	$[\varphi]\cdot(-0.5*n+x+1/2*M)+[\neg\varphi]\cdot x$
Bin12_0	78.19	$x+[n<M]\cdot 0$	1	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin12_1	218.17	$x+[n<M]\cdot(-0.1\cdot n+0.1\cdot M)$	3	0.09	$[\varphi]\cdot(-0.1*n+x+0.1*M)+[\neg\varphi]\cdot x$
Bin13_0	32.95	$x+[n<M]\cdot 0$	1	0.05	$[\varphi]\cdot x+[\neg\varphi]\cdot x$
Bin13_1	TO	–	3	0.08	$[\varphi]\cdot(-0.9*n+x+0.9*M)+[\neg\varphi]\cdot x$
Bin21_0	54.12	$x+[n>0]\cdot 0$	2	0.06	$[\varphi]\cdot x+[\neg\varphi]\cdot x$

Table 3: Exist vs. cegispro2. TO=5min, MO=8GB, time in seconds.

Prog	Exist		cegispro2		
	t	I	$\lvert\text{States}'\rvert$	t	I
Detm1_0	22.8	$count + [x <= 10] \cdot 0$	3	0.06	$[\varphi] \cdot count + [\neg\varphi] \cdot count$
Detm1_1	57.72	$count + [x <= 10] \cdot (1)$	3	0.08	$[\varphi] \cdot (count + 1) + [\neg\varphi] \cdot count$
Duel1_0	TO	–	5	0.12	$[\varphi] \cdot 10/19 + [\neg\varphi] \cdot t$
Duel2_0	TO	–	1	0.09	$[\varphi] \cdot 10/11 + [\neg\varphi] \cdot t$
Fair1_0	30.29	$count + [c1 + c2 = 0] \cdot 0$	2	0.08	$[\varphi] \cdot count + [\neg\varphi] \cdot count$
Fair1_1	37.05	$count + [c1 + c2 = 0] \cdot (1.0 \cdot 1)$	2	0.1	$[\varphi] \cdot (count + 1) + [\neg\varphi] \cdot count$
Gambler01_0	35.81	$z + [(x > 0)\&(y > x)] \cdot 0$	2	0.07	$[\varphi] \cdot z + [\neg\varphi] \cdot z$
Geo01_0	31.71	$z + [flip = 0] \cdot 0$	1	0.05	$[\varphi] \cdot z + [\neg\varphi] \cdot z$
Geo01_1	68.53	$z + [flip = 0] \cdot (0.5 \cdot 1)$	1	0.05	$[\varphi] \cdot (0.5 * z + 0.5) + [\neg\varphi] \cdot z$
Geo01_2	73.85	$z + [flip = 0] \cdot (1.0 \cdot 1)$	2	0.09	$[\varphi] \cdot (z + 0.5) + [\neg\varphi] \cdot z$
Geo11_0	42.13	$z + [flip = 0] \cdot 0$	1	0.05	$[\varphi] \cdot z + [\neg\varphi] \cdot z$
Geo21_0	37.1	$z + [flip = 0] \cdot 0$	1	0.05	$[\varphi] \cdot z + [\neg\varphi] \cdot z$
GeoAr01_0	122.83	$x + [z! = 0] \cdot (1.0 \cdot y)$	2	0.06	$[\varphi] \cdot x + [\neg\varphi] \cdot x$
GeoAr01_1	33.81	$x + [z! = 0] \cdot 0$	2	0.08	$[\varphi] \cdot (x + y) + [\neg\varphi] \cdot z$
LinExp1_0	189.11	$z + [n > 0] \cdot (n^{1.0} \cdot 2)$	7	0.53	$[\varphi] \cdot (16/21 * z + 2 * n) + [\neg\varphi] \cdot z$
LinExp1_1	196.32	$z + [n > 0] \cdot (1.0 \cdot n + 1.0 \cdot 1)$	3	0.52	$[\varphi] \cdot (z + 2) + [\neg\varphi] \cdot z$
PrinSys1_0	13.27	$[x = 1] \cdot 1 + [x = 0] \cdot 0$	0	0.06	$[x = 1] * 1 + [x \neq 1] * 0$
RevBin1_0	179.97	$z + [x > 0] \cdot (x^1 \cdot 2)$	2	0.09	$[\varphi] \cdot (x + z) + [\neg\varphi] \cdot z$
RevBin1_1	52.03	$z + [x > 0] \cdot 0$	1	0.05	$[\varphi] \cdot z + [\neg\varphi] \cdot z$
Sum01_0	TO	–	4	0.09	$[\varphi] \cdot (0.25 * n + x) + [\neg\varphi] \cdot x$
Mart1_0	84.39	$rounds + [b > 0] \cdot 0$	–	TO	–
Mart1_1	TO	–	–	TO	–

Table 4: Exist vs. cegispro2. TO=5min, MO=8GB, time in seconds.

4 A Note on Contributions of the Author

I am obliged to discuss my contributions to the publications I have co-authored as a doctoral researcher at RWTH Aachen University. It was my pleasure to collaborate with great researchers, some of which became good friends. Each of my collaborators took their part in the respective publications. I will not discuss their contributions in great detail. Rather, I will (i) mention who provided the initial idea and (ii) discuss *my* contributions to the respective publications.

[19] K. Batz, T. J. Biskup, J.-P. Katoen, and T. Winkler. "*Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs.*" *Proc. ACM Program. Lang.* 8.POPL (2024), pages 2792–2820

The initiative for conducting this research came from me. I have contributed significantly to the theory, the case studies, and the writing of the paper.

[18] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*A Deductive Verification Infrastructure for Probabilistic Programs.*" *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pages 2052–2082

The initiative for conducting this research came from Christoph Matheja. I have contributed significantly to the the theory, the case studies, and the writing of the paper. The tool has been developed by Philipp Schröer.

[17] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and L. Verscht. "*A Calculus for Amortized Expected Runtimes.*" *Proc. ACM Program. Lang.* 7.POPL (2023), pages 1957–1986

The initiative for conducting this research came from Benjamin Kaminski. I have contributed significantly to the theory, the case studies, and the writing of the paper.

[16] K. Batz, M. Chen, S. Junges, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants.*" *TACAS (2)*. Volume 13994. Lecture Notes in Computer Science. Springer, 2023, pages 410–429

The initiative to employ CEGIS for the automatic synthesis of quantitative loop invariants came from Sebastian Junges. I then contributed ideas on concrete algorithms. This includes the inner CEGIS loop as well as identifying the need for template refinement strategies and the corresponding algorithms. I have contributed significantly to the algorithms, the theory, the case studies, the implementation of the tool, and the writing of the paper.

[15] K. Batz, A. Gallus, B. L. Kaminski, J.-P. Katoen, and T. Winkler. "*Weighted Programming: A Programming Paradigm for Specifying Mathematical Models.*" *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pages 1–30

The initiative for conducting this research came for Benjamin Kaminski and Tobias Winkler. I have contributed significantly to the theory, the case studies, and the writing of the paper.

[14] K. Batz, I. Fesefeldt, M. Jansen, J.-P. Katoen, F. Keßler, C. Matheja, et al. "*Foundations for Entailment Checking in Quantitative Separation Logic.*" *ESOP*. Volume 13240. Lecture Notes in Computer Science. Springer, 2022, pages 57–84

The initiative for conducting this research came from Christoph Matheja and Thomas Noll. I have contributed significantly to the theory, the case studies, and the writing of the paper.

[12] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-based Reasoning.*" *Proc. ACM Program. Lang.* 5.POPL (2021), pages 1–30

The initiative for conducting this research came from Benjamin Kaminski. I have contributed significantly to the theory and the writing of the paper.

[11] K. Batz, M. Chen, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. "*Latticed k-Induction with an Application to Probabilistic Programs.*" *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 524–549

The initiative for conducting this research came from me. I have contributed significantly to the theory, the case studies, the implementation of the tool, and the writing of the paper.

[9] L. Klinkenberg, K. Batz, B. L. Kaminski, J.-P. Katoen, J. Moerman, and T. Winkler. "*Generating Functions for Probabilistic Programs.*" *LOPSTR*. Volume 12561. Lecture Notes in Computer Science. Springer, 2020, pages 231–248

The initiative for conducting this research came from Benjamin Kaminski. I helped to develop the theory by numerous discussions and contributed significantly to the writing of the paper.

[6] K. Batz, S. Junges, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. "*PrIC3: Property Directed Reachability for MDPs.*" *CAV (2)*. Volume 12225. Lecture Notes in Computer Science. Springer, 2020, pages 512–538

The initiative for conducting this research came from Sebastian Junges, Benjamin Kaminski, and Christoph Matheja, who supervised my master thesis [4] on a probabilistic generalization of IC3. However, the approach presented in [6; 7] and this thesis differs significantly from the approach given in [4]. The foundations of the fixpoint-theoretic perspective on IC3 have been laid in [4, Chapter 3]. I have contributed significantly to the algorithms, the theory, the case studies, the implementation of the tool, and the writing of the paper.

[5] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and T. Noll. "*Quantitative Separation logic: A Logic for Reasoning about Probabilistic Pointer Programs.*" *Proc. ACM Program. Lang.* 3.POPL (2019), 34:1–34:29

The initiative for conducting this research came from Benjamin Kaminski and Christoph Matheja. I was happy to support the development of QSL as a student assistant. I have contributed significantly to formal proofs and case studies. The paper was mostly authored by Joost-Pieter Katoen, Thomas Noll, Benjamin Kaminski, and Christoph Matheja.

[2] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times.*" *ESOP*. Volume 10801. Lecture Notes in Computer Science. Springer, 2018, pages 186–213

This work emerged from my bachelor thesis [1] which has been supervised by Benjamin Kaminski and Christoph Matheja. My task was to investigate classes of loops for which expected runtimes can be computed automatically. The idea to consider i.i.d. loops is due to me. The idea to apply the so-obtained techniques to Bayesian networks is due to Christoph Matheja. I have contributed significantly to the theory, the case studies, and the implementation of the tool.

Bibliography

References

[ABDF^{+}15] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. "*Syntax-Guided Synthesis.*" *Dependable Software Systems Engineering*. Volume 40. IOS Press, 2015, pages 1–25 (cited on pages 165, 197).

[ABFG^{+}22] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. "*The Prusti Project: Formal Verification for Rust.*" *NFM*. Volume 13260. Lecture Notes in Computer Science. Springer, 2022, pages 88–108 (cited on page 3).

[ACJK^{+}21] R. Andriushchenko, M. Ceska, S. Junges, J.-P. Katoen, and S. Stupinský. "*PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs.*" *CAV (1)*. Volume 12759. Lecture Notes in Computer Science. Springer, 2021, pages 856–869 (cited on page 200).

[ACJK21] R. Andriushchenko, M. Ceska, S. Junges, and J.-P. Katoen. "*Inductive Synthesis for Probabilistic Programs Reaches New Horizons.*" *TACAS (1)*. Volume 12651. Lecture Notes in Computer Science. Springer, 2021, pages 191–209 (cited on page 200).

[ACN18] S. Agrawal, K. Chatterjee, and P. Novotný. "*Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs.*" *Proc. ACM Program. Lang.* 2.POPL (2018), 34:1–34:32 (cited on page 198).

[ADH22] N. Amat, S. Dal-Zilio, and T. Hujsa. "*Property Directed Reachability for Generalized Petri Nets.*" *TACAS (1)*. Volume 13243. Lecture Notes in Computer Science. Springer, 2022, pages 505–523 (cited on page 255).

[AG00] T. Arts and J. Giesl. "*Termination of term rewriting using dependency pairs.*" *Theor. Comput. Sci.* 236.1-2 (2000), pages 133–178 (cited on page 198).

[AGR21] A. Abate, M. Giacobbe, and D. Roy. "*Learning Probabilistic Termination Proofs.*" *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 3–26 (cited on pages 118, 198, 199).

[AJ95] S. Abramsky and A. Jung. "*Domain Theory.*" *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. Oxford University Press, Inc., 1995, pages 1–168 (cited on page 11).

[AKNP$^+$00] L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. "*Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation.*" *TACAS*. Volume 1785. Lecture Notes in Computer Science. Springer, 2000, pages 395–410 (cited on page 200).

[AMS20] M. Avanzini, G. Moser, and M. Schaper. "*A modular cost analysis for probabilistic programs.*" *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 172:1–172:30 (cited on page 198).

[AO19] K. R. Apt and E. Olderog. "*Fifty years of Hoare's logic.*" *Formal Aspects Comput.* 31.6 (2019), pages 751–807 (cited on page 119).

[Apt81] K. R. Apt. "*Ten Years of Hoare's Logic: A Survey - Part 1.*" *ACM Trans. Program. Lang. Syst.* 3.4 (1981), pages 431–483 (cited on page 119).

[Bar17] H. Barbosa. "*New techniques for instantiation and proof production in SMT solving. (Nouvelles techniques pour l'instanciation et la production des preuves dans SMT).*" PhD thesis. University of Lorraine, Nancy, France, 2017 (cited on page 199).

[BBBK$^+$22] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. "*cvc5: A Versatile and Industrial-Strength SMT Solver.*" *TACAS (1)*. Volume 13243. Lecture Notes in Computer Science. Springer, 2022, pages 415–442 (cited on pages 3, 128).

[BBW14] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. "*Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR).*" *CAV*. Volume 8559. Lecture Notes in Computer Science. Springer, 2014, pages 831–848 (cited on page 255).

[BCHK+97] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Z. Kwiatkowska, and M. Ryan. "*Symbolic Model Checking for Probabilistic Processes.*" *ICALP*. Volume 1256. Lecture Notes in Computer Science. Springer, 1997, pages 430–440 (cited on page 200).

[BDW15] D. Beyer, M. Dangl, and P. Wendler. "*Boosting k-Induction with Continuously-Refined Invariants.*" *CAV (1)*. Volume 9206. Lecture Notes in Computer Science. Springer, 2015, pages 622–640 (cited on pages 127, 163).

[BDW18] D. Beyer, M. Dangl, and P. Wendler. "*A Unifying View on SMT-Based Software Verification.*" *J. Autom. Reason.* 60.3 (2018), pages 299–335 (cited on page 129).

[BE98] A. Borodin and R. El-Yaniv. "*Online computation and competitive analysis.*" Cambridge University Press, 1998 (cited on page 197).

[BEFH16] G. Barthe, T. Espitau, L. M. F. Fioriti, and J. Hsu. "*Synthesizing Probabilistic Invariants via Doob's Decomposition.*" *CAV (1)*. Volume 9779. Lecture Notes in Computer Science. Springer, 2016, pages 43–61 (cited on pages 118, 198).

[BEGG+18] G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, and P. Strub. "*An Assertion-Based Program Logic for Probabilistic Programs.*" *ESOP*. Volume 10801. Lecture Notes in Computer Science. Springer, 2018, pages 117–144 (cited on page 5).

[Bel57] R. Bellman. "*A Markovian Decision Process.*" *Indiana University Mathematics Journal* 6 (1957), pages 679–684 (cited on pages 31, 39).

[Ber49] J. Bertrand. "*Traité d'Arithmétique.*" Libraire de L. Hachette et Cie., 1849 (cited on pages 95, 104).

[BG15] N. S. Bjørner and A. Gurfinkel. "*Property Directed Polyhedral Abstraction.*" *VMCAI*. Volume 8931. Lecture Notes in Computer Science. Springer, 2015, pages 263–281 (cited on page 255).

[BHK19] C. Baier, H. Hermanns, and J.-P. Katoen. "*The 10, 000 Facets of MDP Model Checking.*" *Computing and Software Science*. Volume 10000. Lecture Notes in Computer Science. Springer, 2019, pages 420–451 (cited on page 4).

[BJ15] N. S. Bjørner and M. Janota. "*Playing with Quantified Satisfaction.*" *LPAR (short papers)*. Volume 35. EPiC Series in Computing. EasyChair, 2015, pages 15–27 (cited on page 199).

[BK08] C. Baier and J.-P. Katoen. "*Principles of model checking.*" MIT Press, 2008 (cited on pages 4, 12, 24, 26, 32, 35, 38, 43, 124, 193, 222, 241, 246, 263).

[BK11] D. Beyer and M. E. Keremoglu. "*CPAchecker: A Tool for Configurable Software Verification.*" *CAV*. Volume 6806. Lecture Notes in Computer Science. Springer, 2011, pages 184–190 (cited on pages 4, 127).

[BKLP+17] C. Baier, J. Klein, L. Leuschner, D. Parker, and S. Wunderlich. "*Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes.*" *CAV (1)*. Volume 10426. Lecture Notes in Computer Science. Springer, 2017, pages 160–180 (cited on pages 163, 164, 200).

[BKS19] E. Bartocci, L. Kovács, and M. Stankovic. "*Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops.*" *ATVA*. Volume 11781. Lecture Notes in Computer Science. Springer, 2019, pages 255–276 (cited on page 199).

[BKS20a] G. Barthe, J.-P. Katoen, and A. Silva. "*Foundations of Probabilistic Programming.*" Cambridge University Press, 2020 (cited on page 5).

[BKS20b] E. Bartocci, L. Kovács, and M. Stankovic. "*Mora - Automatic Generation of Moment-Based Invariants.*" *TACAS (1)*. Volume 12078. Lecture Notes in Computer Science. Springer, 2020, pages 492–498 (cited on page 199).

[Bla67] D. Blackwell. "*Positive dynamic programming.*" *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. Volume 1. University of California Press, 1967, pages 415–418 (cited on page 73).

[BLNR$^+$09] C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. "*Solving Non-linear Polynomial Arithmetic via SAT Modulo Linear Arithmetic.*" *CADE*. Volume 5663. Lecture Notes in Computer Science. Springer, 2009, pages 294–305 (cited on page 128).

[BPF15] N. S. Bjørner, A. Phan, and L. Fleckenstein. "*νZ - An Optimizing SMT Solver.*" *TACAS*. Volume 9035. Lecture Notes in Computer Science. Springer, 2015, pages 194–199 (cited on page 247).

[BPv15] V. Belle, A. Passerini, and van den Broeck, G. "*Probabilistic Inference in Hybrid Domains by Weighted Model Integration.*" *IJCAI*. AAAI Press, 2015, pages 2770–2776 (cited on page 200).

[Bra11a] A. R. Bradley. "*SAT-Based Model Checking without Unrolling.*" *VMCAI*. Volume 6538. Lecture Notes in Computer Science. Springer, 2011, pages 70–87 (cited on pages 7, 31, 201, 202, 216, 218, 219, 255).

[Bra11b] A. R. Bradley. "*SAT-Based Model Checking without Unrolling.*" *VMCAI*. Volume 6538. LNCS. Springer, 2011, pages 70–87 (cited on page 225).

[BTPH$^+$22] J. Bao, N. Trivedi, D. Pathak, J. Hsu, and S. Roy. "*Data-Driven Invariant Learning for Probabilistic Programs.*" *CAV (1)*. Volume 13371. Lecture Notes in Computer Science. Springer, 2022, pages 33–54 (cited on pages 118, 193, 195, 199, 281).

[CAG05] S. Cheshire, B. Aboba, and E. Guttman. "*Dynamic Configuration of IPv4 Link-Local Addresses.*" *RFC* 3927 (2005), pages 1–33 (cited on pages 193, 253).

[Car08] J. Carter. "*Categories for the working mathematician: making the impossible possible.*" *Synth.* 162.1 (2008), pages 1–13 (cited on page 256).

[CBRZ01] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. "*Bounded Model Checking Using Satisfiability Solving.*" *Formal Methods Syst. Des.* 19.1 (2001), pages 7–34 (cited on pages 6, 124, 125, 163).

[CC79] P. Cousot and R. Cousot. "*Constructive versions of Tarski's fixed point theorems.*" *Pacific Journal of Mathematics* 82.1 (1979), pages 43–57 (cited on pages 26, 27, 137, 140).

[CCMS07] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. "*Reasoning about probabilistic sequential programs.*" *Theor. Comput. Sci.* 379.1-2 (2007), pages 142–165 (cited on page 5).

[CDDG$^+$15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. "*Moving Fast with Software Verification.*" *NFM*. Volume 9058. Lecture Notes in Computer Science. Springer, 2015, pages 3–11 (cited on pages 4, 124).

[CDM17] D. Chistikov, R. Dimitrova, and R. Majumdar. "*Approximate counting in SMT and value estimation for probabilistic programs.*" *Acta Informatica* 54.8 (2017), pages 729–764 (cited on page 200).

[CFG16] K. Chatterjee, H. Fu, and A. K. Goharshady. "*Termination Analysis of Probabilistic Programs Through Positivstellensatz's.*" *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Edited by S. Chaudhuri and A. Farzan. Volume 9779. Lecture Notes in Computer Science. Springer, 2016, pages 3–22 (cited on pages 118, 198).

[CFMV15] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi. "*From Weighted to Unweighted Model Counting.*" *IJCAI*. AAAI Press, 2015, pages 689–695 (cited on page 200).

[CFNH16] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. "*Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs.*" *POPL*. ACM, 2016, pages 327–342 (cited on page 118).

[CFNH18] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. "*Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs.*" *ACM Trans. Program. Lang. Syst.* 40.2 (2018), 7:1–7:45 (cited on page 118).

[CG12] A. Cimatti and A. Griggio. "*Software Model Checking via IC3.*" *CAV*. Volume 7358. Lecture Notes in Computer Science. Springer, 2012, pages 277–293 (cited on page 255).

[CGMT14] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. "*IC3 Modulo Theories via Implicit Predicate Abstraction.*" *TACAS*. Volume 8413. Lecture Notes in Computer Science. Springer, 2014, pages 46–61 (cited on page 255).

[CGMT16] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. “*Infinite-state invariant checking with IC3 and predicate abstraction.*” *FMSD* 49.3 (2016), pages 190–218 (cited on pages 224, 255).

[CGSS13] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. “*The MathSAT5 SMT Solver.*” *TACAS*. Volume 7795. Lecture Notes in Computer Science. Springer, 2013, pages 93–107 (cited on pages 3, 128).

[CHJK21] M. Ceska, C. Hensel, S. Junges, and J.-P. Katoen. “*Counterexample-guided inductive synthesis for probabilistic systems.*” *Formal Aspects Comput.* 33.4-5 (2021), pages 637–667 (cited on page 200).

[CHLP10] H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. “*Inference with constrained hidden Markov models in PRISM.*” *Theory Pract. Log. Program.* 10.4-6 (2010), pages 449–464 (cited on page 4).

[CHWZ15] Y. Chen, C. Hong, B. Wang, and L. Zhang. “*Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation.*” *CAV (1)*. Volume 9206. Lecture Notes in Computer Science. Springer, 2015, pages 658–674 (cited on pages 198, 199).

[CKKW22] M. Chen, J.-P. Katoen, L. Klinkenberg, and T. Winkler. “*Does a Program Yield the Right Distribution? Verifying Probabilistic Programs via Generating Functions.*” *CAV (1)*. Volume 13371. Lecture Notes in Computer Science. Springer, 2022, pages 79–101 (cited on page 200).

[Cla76] E. M. Clarke. “*Completeness and Incompleteness Theorems for Hoare-like Axiom Systems.*” PhD thesis. Cornell University, USA, 1976 (cited on page 119).

[CMMV16] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi. “*Approximate Probabilistic Inference via Word-Level Counting.*” *AAAI*. AAAI Press, 2016, pages 3218–3224 (cited on page 200).

[CNZ17] K. Chatterjee, P. Novotný, and D. Zikelic. “*Stochastic invariants for probabilistic termination.*” *POPL*. ACM, 2017, pages 145–160 (cited on pages 118, 198).

[Coo78] S. A. Cook. “*Soundness and Completeness of an Axiom System for Program Verification.*” *SIAM J. Comput.* 7 (1978), pages 70–90 (cited on pages 77, 96, 115, 119).

[Cou21] P. Cousot. “*Principles of Abstract Interpretation.*” MIT Press, 2021 (cited on page 256).

[CS13] A. Chakarov and S. Sankaranarayanan. “*Probabilistic Program Analysis with Martingales.*” *CAV*. Volume 8044. Lecture Notes in Computer Science. Springer, 2013, pages 511–526 (cited on pages 118, 198).

[CSS03] M. Colón, S. Sankaranarayanan, and H. Sipma. “*Linear Invariant Generation Using Non-linear Constraint Solving.*” *CAV*. Volume 2725. Lecture Notes in Computer Science. Springer, 2003, pages 420–432 (cited on pages 145, 197).

[CV10] R. Chadha and M. Viswanathan. “*A counterexample-guided abstraction-refinement framework for Markov decision processes.*” *ACM Trans. Comput. Log.* 12.1 (2010), 1:1–1:49 (cited on pages 224, 241).

[DHKR11] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. “*Software Verification Using k-Induction.*” *SAS*. Volume 6887. Lecture Notes in Computer Science. Springer, 2011, pages 351–368 (cited on pages 125, 127, 163).

[Dij75] E. W. Dijkstra. “*Guarded Commands, Nondeterminacy and Formal Derivation of Programs.*” *Commun. ACM* 18.8 (1975), pages 453–457 (cited on pages 6, 53, 117).

[Dij76] E. W. Dijkstra. “*A Discipline of Programming.*” Prentice-Hall, 1976 (cited on pages 26, 45, 53, 117).

[DJJL01] P. R. D’Argenio, B. Jeannet, H. E. Jensen, and K. G. Larsen. “*Reachability Analysis of Probabilistic Systems by Successive Refinements.*” *PAPM-PROBMIV*. Volume 2165. Lecture Notes in Computer Science. Springer, 2001, pages 39–56 (cited on pages 56, 121, 169, 253).

[DJKV17] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. “*A Storm is Coming: A Modern Probabilistic Model Checker.*” *CAV (2)*. Volume 10427. Lecture Notes in Computer Science. Springer, 2017, pages 592–600 (cited on pages 4, 122, 172, 193).

[DKR10] Donaldson, Kroening, and Rümmer. “*Automatic Analysis of Scratch-Pad Memory Code for Heterogeneous Multicore Processors.*” *TACAS*. 2010 (cited on pages 163, 255).

[DKR11] Donaldson, Kroening, and Rümmer. "*Automatic analysis of DMA races using model checking and* k*-induction.*" *Formal Methods Syst. Des.* 39.1 (2011) (cited on page 163).

[DKV09] M. Droste, W. Kuich, and H. Vogler. "*Handbook of Weighted Automata.*" 1st. Springer Publishing Company, Incorporated, 2009 (cited on page 41).

[Ech05] F. Echenique. "*A short and constructive proof of Tarski's fixed-point theorem.*" *Int. J. Game Theory* 33.2 (2005), pages 215–218 (cited on page 27).

[EFT94] H. Ebbinghaus, J. Flum, and W. Thomas. "*Mathematical logic (2. ed.)*" Undergraduate texts in mathematics. Springer, 1994 (cited on page 85).

[EMB11] N. Eén, A. Mishchenko, and R. K. Brayton. "*Efficient implementation of property directed reachability.*" *FMCAD*. FMCAD Inc., 2011, pages 125–134 (cited on pages 201, 202, 205, 255).

[FB18] G. Fedyukovich and R. Bodík. "*Accelerating Syntax-Guided Invariant Synthesis.*" *TACAS (1)*. Volume 10805. Lecture Notes in Computer Science. Springer, 2018, pages 251–269 (cited on page 197).

[FC19] H. Fu and K. Chatterjee. "*Termination of Nondeterministic Probabilistic Programs.*" *VMCAI*. Volume 11388. Lecture Notes in Computer Science. Springer, 2019, pages 468–490 (cited on pages 118, 198).

[FH15] L. M. F. Fioriti and H. Hermanns. "*Probabilistic Termination: Soundness, Completeness, and Compositionality.*" *POPL*. ACM, 2015, pages 489–501 (cited on pages 118, 198).

[Flo67] R. W. Floyd. "*Assigning Meanings to Programs.*" *Mathematical Aspects of Computer Science* 19.19-32 (1967), page 1 (cited on page 1).

[Fon08] G. Fontaine. "*Continuous Fragment of the mu-Calculus.*" *CSL*. Volume 5213. Lecture Notes in Computer Science. Springer, 2008, pages 139–153 (cited on page 23).

[Fou25] J. Fourier. "*Analyse des travaux de l'Académie royale des sciences pendant l'année 1824: rapport lu dans la séance publique de l'Institut le 24 avril 1825. Partie mathématique.*" Institut (Paris). Institut royal de France, 1825 (cited on page 176).

[FPS11] P. Flajolet, M. Pelletier, and M. Soria. "*On Buffon Machines and Numbers.*" *SODA*. SIAM, 2011, pages 172–183 (cited on page 78).

[FZJZ$^+$17] Y. Feng, L. Zhang, D. N. Jansen, N. Zhan, and B. Xia. "*Finding Polynomial Loop Invariants for Probabilistic Programs.*" *ATVA*. Volume 10482. Lecture Notes in Computer Science. Springer, 2017, pages 400–416 (cited on pages 118, 198).

[GABE$^+$17] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. "*Analyzing Program Termination and Complexity Automatically with AProVE.*" *J. Autom. Reason.* 58.1 (2017), pages 3–31 (cited on page 198).

[Gha15] Z. Ghahramani. "*Probabilistic machine learning and artificial intelligence.*" *Nat.* 521.7553 (2015), pages 452–459 (cited on page 4).

[GHNR14] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. "*Probabilistic programming.*" *FOSE*. ACM, 2014, pages 167–181 (cited on page 4).

[GI15] A. Gurfinkel and A. Ivrii. "*Pushing to the Top.*" *FMCAD*. IEEE, 2015, pages 65–72 (cited on page 255).

[GI17] A. Gurfinkel and A. Ivrii. "*K-induction without unrolling.*" *FMCAD*. IEEE, 2017, pages 148–155 (cited on pages 125, 163).

[GKKN15] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. "*The SeaHorn Verification Framework.*" *CAV (1)*. Volume 9206. Lecture Notes in Computer Science. Springer, 2015, pages 343–361 (cited on page 4).

[GKM13] F. Gretz, J.-P. Katoen, and A. McIver. "*Prinsys - On a Quest for Probabilistic Loop Invariants.*" *QEST*. Volume 8054. Lecture Notes in Computer Science. Springer, 2013, pages 193–208 (cited on page 197).

[GKM14] F. Gretz, J.-P. Katoen, and A. McIver. "*Operational versus weakest pre-expectation semantics for the probabilistic guarded command language.*" *Perform. Evaluation* 73 (2014), pages 110–132 (cited on pages 69, 73).

[GLMN14] P. Garg, C. Löding, P. Madhusudan, and D. Neider. "*ICE: A Robust Framework for Learning Invariants.*" *CAV*. Volume 8559. Lecture Notes in Computer Science. Springer, 2014, pages 69–87 (cited on pages 197, 199).

[GM09] Y. Ge and L. M. de Moura. "*Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories.*" *CAV*. Volume 5643. Lecture Notes in Computer Science. Springer, 2009, pages 306–320 (cited on page 199).

[GM15] Gario and Micheli. "*PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms.*" *SMT Workshop*. 2015 (cited on pages 155, 193).

[GMV16] T. Gehr, S. Misailovic, and M. T. Vechev. "*PSI: Exact Symbolic Inference for Probabilistic Programs.*" *CAV (1)*. Volume 9779. Lecture Notes in Computer Science. Springer, 2016, pages 62–83 (cited on page 200).

[Göd31] K. Gödel. "*Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.*" *Monatshefte für Mathematik und Physik* 38.1 (1931), pages 173–198 (cited on pages 95, 96, 98, 100).

[GR16] A. Griggio and M. Roveri. "*Comparing Different Variants of the ic3 Algorithm for Hardware Model Checking.*" *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35.6 (2016), pages 1026–1039 (cited on page 255).

[Gre16] F. Gretz. "*Semantics and loop invariant synthesis for probabilistic programs.*" PhD thesis. RWTH Aachen University, Germany, 2016 (cited on page 197).

[GSV14] O. Grumberg, S. Shoham, and Y. Vizel. "*SAT-based Model Checking: Interpolation, IC3, and Beyond.*" *Software Systems Safety*. Volume 36. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2014, pages 17–41 (cited on page 255).

[GTC16] N. D. Goodman, J. B. Tenenbaum, and T. P. Contributors. "*Probabilistic Models of Cognition.*" `http://probmods.org/v2`. Accessed: 2024-4-29. 2016 (cited on page 4).

[Har21] M. Hark. “*Towards complete methods for automatic complexity and termination analysis of (probabilistic) programs.*” PhD thesis. RWTH Aachen University, Germany, 2021 (cited on page 192).

[Har99] J. den Hartog. “*Verifying Probabilistic Programs Using a Hoare like Logic.*” *ASIAN*. Volume 1742. Lecture Notes in Computer Science. Springer, 1999, pages 113–125 (cited on page 5).

[HB12] K. Hoder and N. Bjørner. “*Generalized Property Directed Reachability.*” *SAT*. Volume 7317. LNCS. Springer, 2012, pages 157–171 (cited on pages 224, 255).

[HBS13] Z. Hassan, A. R. Bradley, and F. Somenzi. “*Better generalization in IC3.*” *FMCAD*. IEEE, 2013, pages 157–164 (cited on pages 249, 255).

[Her90] T. Herman. “*Probabilistic Self-Stabilization.*” *Inf. Process. Lett.* 35.2 (1990), pages 63–67 (cited on page 4).

[HFC18] M. Huang, H. Fu, and K. Chatterjee. “*New Approaches for Almost-Sure Termination of Probabilistic Programs.*” *APLAS*. Edited by S. Ryu. Volume 11275. Lecture Notes in Computer Science. Springer, 2018, pages 181–201 (cited on page 118).

[HFCG19] M. Huang, H. Fu, K. Chatterjee, and A. K. Goharshady. “*Modular verification for almost-sure termination of probabilistic programs.*” *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 129:1–129:29 (cited on page 118).

[HJKQ$^+$22] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk. “*The probabilistic model checker Storm.*” *Int. J. Softw. Tools Technol. Transf.* 24.4 (2022), pages 589–610 (cited on pages 4, 122, 172, 193).

[HJVM$^+$21] S. Holtzen, S. Junges, M. Vazquez-Chanlatte, T. D. Millstein, S. A. Seshia, and van den Broeck, G. “*Model Checking Finite-Horizon Markov Chains with Probabilistic Inference.*” *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 577–601 (cited on pages 200, 253).

[HK20] A. Hartmanns and B. L. Kaminski. “*Optimistic Value Iteration.*” *CAV (2)*. Volume 12225. Lecture Notes in Computer Science. Springer, 2020, pages 488–511 (cited on pages 163, 200).

[HKD09] T. Han, J.-P. Katoen, and B. Damman. "*Counterexample Generation in Probabilistic Model Checking.*" *IEEE Trans. Software Eng.* 35.2 (2009), pages 241–257 (cited on page 224).

[HKGK20] M. Hark, B. L. Kaminski, J. Giesl, and J.-P. Katoen. "*Aiming low is harder: induction for lower bounds in probabilistic program verification.*" *Proc. ACM Program. Lang.* 4.POPL (2020), 37:1–37:28 (cited on pages 68, 191).

[HKNP06] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. "*PRISM: A Tool for Automatic Verification of Probabilistic Systems.*" *TACAS*. Volume 3920. Lecture Notes in Computer Science. Springer, 2006, pages 441–444 (cited on page 4).

[HMM05] J. Hurd, A. McIver, and C. Morgan. "*Probabilistic guarded commands mechanized in HOL.*" *Theoretical Computer Science* 346.1 (2005). Quantitative Aspects of Programming Languages (QAPL 2004), pages 96–112 (cited on page 159).

[Hoa62] C. A. R. Hoare. "*Quicksort.*" *The Computer Journal* 5.1 (1962), pages 10–16 (cited on page 4).

[Hoa69] C. A. R. Hoare. "*An Axiomatic Basis for Computer Programming.*" *Commun. ACM* 12.10 (1969), pages 576–580 (cited on pages 1, 31).

[HSP82] S. Hart, M. Sharir, and A. Pnueli. "*Termination of Probabilistic Concurrent Programs.*" *POPL*. ACM Press, 1982, pages 1–6 (cited on page 5).

[HSV93] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. "*Proof-Checking a Data Link Protocol.*" *TYPES*. Volume 806. Lecture Notes in Computer Science. Springer, 1993, pages 127–165 (cited on pages 56, 121, 169, 253).

[Hüt10] H. Hüttel. "*Transitions and Trees - An Introduction to Structural Operational Semantics.*" Cambridge University Press, 2010 (cited on page 15).

[HV02] J. den Hartog and E. P. de Vink. "*Verifying Probabilistic Programs Using a Hoare Like Logic.*" *Int. J. Found. Comput. Sci.* 13.3 (2002), pages 315–340 (cited on pages 5, 119).

[HvM20] S. Holtzen, van den Broeck, G., and T. D. Millstein. "*Scaling exact inference for discrete probabilistic programs.*" *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 140:1–140:31 (cited on page 200).

[JÁZW$^+$12] N. Jansen, E. Ábrahám, B. Zajzon, R. Wimmer, J. Schuster, J.-P. Katoen, and B. Becker. "*Symbolic Counterexample Generation for Discrete-Time Markov Chains.*" *FACS*. Volume 7684. Lecture Notes in Computer Science. Springer, 2012, pages 134–151 (cited on page 163).

[JD16] D. Jovanovic and B. Dutertre. "*Property-directed k-induction.*" *FMCAD*. IEEE, 2016, pages 85–92 (cited on pages 125, 163).

[JDKK$^+$16] N. Jansen, C. Dehnert, B. L. Kaminski, J.-P. Katoen, and L. Westhofen. "*Bounded Model Checking for Probabilistic Programs.*" *ATVA*. Volume 9938. Lecture Notes in Computer Science. 2016, pages 68–85 (cited on pages 163, 241).

[Jon90] C. Jones. "*Probabilistic non-determinism.*" PhD thesis. University of Edinburgh, UK, 1990 (cited on page 255).

[KABB$^+$23] M. Kori, F. Ascari, F. Bonchi, R. Bruni, R. Gori, and I. Hasuo. "*Exploiting Adjoints in Property Directed Reachability Analysis.*" *CAV (2)*. Volume 13965. Lecture Notes in Computer Science. Springer, 2023, pages 41–63 (cited on pages 254–256).

[Kam19] B. L. Kaminski. "*Advanced Weakest Precondition Calculi for Probabilistic Programs.*" PhD thesis. RWTH Aachen University, Germany, 2019 (cited on pages 5, 6, 16, 31, 50, 53, 60, 64, 66, 68, 77, 92, 190, 195).

[Kat16] J.-P. Katoen. "*The Probabilistic Model Checking Landscape.*" *LICS*. ACM, 2016, pages 31–45 (cited on page 4).

[KBGM15] A. Komuravelli, N. S. Bjørner, A. Gurfinkel, and K. L. McMillan. "*Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays.*" *FMCAD*. IEEE, 2015, pages 89–96 (cited on page 255).

[KBT14] T. King, C. W. Barrett, and C. Tinelli. "*Leveraging Linear and Mixed Integer Programming for SMT.*" *SMT*. Volume 1163. CEUR Workshop Proceedings. CEUR-WS.org, 2014, page 65 (cited on page 148).

[KCÁ16] G. Kremer, F. Corzilius, and E. Ábrahám. "*A Generalised Branch-and-Bound Approach and Its Application in SAT Modulo Nonlinear Integer Arithmetic.*" *CASC*. Volume 9890. Lecture Notes in Computer Science. Springer, 2016, pages 315–335 (cited on page 128).

[KG23] J. Kassing and J. Giesl. "*Proving Almost-Sure Innermost Termination of Probabilistic Term Rewriting Using Dependency Pairs.*" *CADE*. Volume 14132. Lecture Notes in Computer Science. Springer, 2023, pages 344–364 (cited on page 198).

[KGC14] A. Komuravelli, A. Gurfinkel, and S. Chaki. "*SMT-Based Model Checking for Recursive Programs.*" *CAV*. Volume 8559. Lecture Notes in Computer Science. Springer, 2014, pages 17–34 (cited on page 255).

[KK15] B. L. Kaminski and J.-P. Katoen. "*On the Hardness of Almost-Sure Termination.*" *MFCS (1)*. Volume 9234. Lecture Notes in Computer Science. Springer, 2015, pages 307–318 (cited on pages 5, 115, 117).

[KK17] B. L. Kaminski and J.-P. Katoen. "*A weakest pre-expectation semantics for mixed-sign expectations.*" *LICS*. IEEE Computer Society, 2017, pages 1–12 (cited on page 59).

[KKM19] B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*On the hardness of analyzing probabilistic programs.*" *Acta Informatica* 56.3 (2019), pages 255–285 (cited on pages 5, 66, 115, 117).

[KKMO16] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. "*Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs.*" *ESOP*. Volume 9632. Lecture Notes in Computer Science. Springer, 2016, pages 364–389 (cited on pages 118, 190).

[KKMO18] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. "*Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms.*" *J. ACM* 65.5 (2018), 30:1–30:68 (cited on pages 118, 190).

[KMMM10] J.-P. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. "*Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods.*" *SAS*. Volume 6337. Lecture Notes in Computer Science. Springer, 2010, pages 390–406 (cited on pages 118, 145, 147, 149, 151, 153, 169, 173, 197).

[KNP02] M. Z. Kwiatkowska, G. Norman, and D. Parker. "*PRISM: Probabilistic Symbolic Model Checker.*" *Computer Performance Evaluation / TOOLS*. Volume 2324. Lecture Notes in Computer Science. Springer, 2002, pages 200–204 (cited on page 4).

[Koz82] D. Kozen. “*Results on the Propositional μ-Calculus.*” *ICALP*. Volume 140. Lecture Notes in Computer Science. Springer, 1982, pages 348–359 (cited on pages 11, 21, 23, 25, 26).

[Koz83] D. Kozen. “*A Probabilistic PDL.*” *STOC*. ACM, 1983, pages 291–297 (cited on pages 5, 6, 53, 72).

[Koz85] D. Kozen. “*A Probabilistic PDL.*” *J. Comput. Syst. Sci.* 30.2 (1985), pages 162–178 (cited on pages 5, 6, 53, 72).

[KR92] E. Kushilevitz and M. O. Rabin. “*Randomized Mutual Exclusion Algorithms Revisited.*” *PODC*. ACM, 1992, pages 275–283 (cited on pages 157, 159).

[Kri63] S. Kripke. “*Semantical Considerations on Modal Logic.*” *Acta Philosophica Fennica* 16 (1963), pages 83–94 (cited on page 12).

[KUH21] S. Kura, H. Unno, and I. Hasuo. “*Decision Tree Learning in CEGIS-Based Termination Analysis.*” *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 75–98 (cited on pages 197, 199).

[KUKS$^+$22] M. Kori, N. Urabe, S. Katsumata, K. Suenaga, and I. Hasuo. “*The Lattice-Theoretic Essence of Property Directed Reachability Analysis.*” *CAV (1)*. Volume 13371. Lecture Notes in Computer Science. Springer, 2022, pages 235–256 (cited on pages 254, 255).

[KVGG19] H. G. V. Krishnan, Y. Vizel, V. Ganesh, and A. Gurfinkel. “*Interpolating Strong Induction.*” *CAV (2)*. Volume 11562. Lecture Notes in Computer Science. Springer, 2019, pages 367–385 (cited on pages 6, 163, 254, 255, 258).

[LAH23] J. M. Li, A. Ahmed, and S. Holtzen. “*Lilac: A Modal Separation Logic for Conditional Probability.*” *Proc. ACM Program. Lang.* 7.PLDI (2023), pages 148–171 (cited on page 5).

[Lan18] T. F. Lange. “*IC3 software model checking.*” PhD thesis. RWTH Aachen University, Germany, 2018 (cited on pages 125, 224, 255).

[Lea00] C. League. “*Lambda Calculi: A Guide for Computer Scientists by Chris Hankin.*” *SIGACT News* 31.1 (2000), pages 8–13 (cited on page 87).

[Lei10] K. R. M. Leino. "*Dafny: An Automatic Program Verifier for Functional Correctness.*" *LPAR (Dakar)*. Volume 6355. Lecture Notes in Computer Science. Springer, 2010, pages 348–370 (cited on page 3).

[LMSS56] K. de Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. "*Computability by Probabilistic Machines.*" *Automata Studies. (AM-34), Volume 34*. Edited by C. E. Shannon and J. McCarthy. Princeton University Press, 1956, pages 183–212 (cited on page 4).

[LNN15] T. Lange, M. R. Neuhäußer, and T. Noll. "*IC3 Software Model Checking on Control Flow Automata.*" *FMCAD*. IEEE, 2015, pages 97–104 (cited on page 255).

[LNNK20a] T. Lange, M. R. Neuhäußer, T. Noll, and J.-P. Katoen. "*IC3 Software Model Checking.*" *STTT* (2020) (cited on page 224).

[LNNK20b] T. Lange, M. R. Neuhäußer, T. Noll, and J.-P. Katoen. "*IC3 software model checking.*" *Int. J. Softw. Tools Technol. Transf.* 22.2 (2020), pages 135–161 (cited on page 255).

[LNS82] J. Lassez, V. L. Nguyen, and L. Sonenberg. "*Fixed Point Theorems and Semantics: A Folk Tale.*" *Inf. Process. Lett.* 14.3 (1982), pages 112–116 (cited on page 140).

[LS16] X. Li and K. Schneider. "*Control-flow guided property directed reachability for imperative synchronous programs.*" *MEMOCODE*. IEEE, 2016, pages 23–33 (cited on page 255).

[LS87] J. Loeckx and K. Sieber. "*The Foundations of Program Verification, 2nd ed.*" Wiley-Teubner, 1987 (cited on page 94).

[LSS84] J. Loeckx, K. Sieber, and R. Stansifer. "*The Foundations of Program Verification.*" Wiley Teubner Series on Applicable Theory in Computer Science Series. John Wiley, 1984 (cited on pages 79, 96, 119).

[Lum13] J. O. Lumbroso. "*Optimal Discrete Uniform Generation from Coin Flips, and Applications.*" *CoRR* abs/1304.1916 (2013) (cited on page 160).

[MB08] L. M. de Moura and N. S. Bjørner. "*Z3: An Efficient SMT Solver.*" *TACAS*. Volume 4963. Lecture Notes in Computer Science. Springer, 2008, pages 337–340 (cited on pages 3, 128, 148, 155, 176, 193).

[MBKK21] M. Moosbrugger, E. Bartocci, J.-P. Katoen, and L. Kovács. "*Automated Termination Analysis of Polynomial Probabilistic Programs.*" *ESOP*. Volume 12648. Lecture Notes in Computer Science. Springer, 2021, pages 491–518 (cited on page 198).

[MHG21] F. Meyer, M. Hark, and J. Giesl. "*Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes.*" *TACAS (1)*. Volume 12651. Lecture Notes in Computer Science. Springer, 2021, pages 250–269 (cited on page 198).

[Mil89] R. Milner. "*Communication and concurrency.*" PHI Series in computer science. Prentice Hall, 1989 (cited on page 164).

[MM05] A. McIver and C. Morgan. "*Abstraction, Refinement and Proof for Probabilistic Systems.*" Monographs in Computer Science. Springer, 2005 (cited on pages 5, 6, 16, 31, 45, 53).

[MM99] C. Morgan and A. McIver. "*PGCL: Formal reasoning for random algorithms.*" *South African Computer Journal* 22 (Nov. 1999) (cited on page 45).

[MMS96] C. Morgan, A. McIver, and K. Seidel. "*Probabilistic Predicate Transformers.*" *ACM Trans. Program. Lang. Syst.* 18.3 (1996), pages 325–353 (cited on pages 5, 6, 53).

[Mot36] T. Motzkin. "*Beitraege zur Theorie der linearen Ungleichungen.*" Universitaet Basel, 1936 (cited on pages 176, 197).

[MP95] Z. Manna and A. Pnueli. "*Temporal verification of reactive systems - safety.*" Springer, 1995 (cited on pages 31, 124).

[NCH18] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. "*Bounded expectations: resource analysis for probabilistic programs.*" *PLDI*. ACM, 2018, pages 496–512 (cited on pages 118, 193, 195, 198, 281).

[NN07] H. R. Nielson and F. Nielson. "*Semantics with Applications: An Appetizer.*" Undergraduate Topics in Computer Science. Springer, 2007 (cited on page 75).

[NPRB$^+$21] A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli. "*Syntax-Guided Quantifier Instantiation.*" *TACAS (2)*. Volume 12652. Lecture Notes in Computer Science. Springer, 2021, pages 145–163 (cited on page 199).

[OGJK$^+$18] F. Olmedo, F. Gretz, N. Jansen, B. L. Kaminski, J. Katoen, and A. McIver. "*Conditioning in Probabilistic Programming.*" *ACM Trans. Program. Lang. Syst.* 40.1 (2018), 4:1–4:50 (cited on page 258).

[OKKM16] F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "*Reasoning about Recursive Probabilistic Programs.*" *LICS*. ACM, 2016, pages 672–681 (cited on page 79).

[Old80] E. Olderog. "*General Equivalence of Expressivity Definitions Using Strongest Postconditions Resp. Weakest Preconditions.*" Universität Kiel. Institut für Informatik und Praktische Mathematik, 1980 (cited on page 119).

[Orn69] D. Ornstein. "*On the existence of stationary optimal strategies.*" *Proceedings of the American Mathematical Society* 20.2 (1969), pages 563–569 (cited on page 44).

[Par69] D. Park. "*Fixpoint Induction and Proofs of Program Properties.*" *Machine Intelligence* 5 (1969) (cited on pages 28, 66).

[PBFA19] E. Polgreen, M. Brain, M. Fränzle, and A. Abate. "*Verifying Reachability Properties in Markov Chains via Incremental Induction.*" *CoRR* abs/1909.08017 (2019) (cited on page 255).

[Plo04] G. D. Plotkin. "*The origins of structural operational semantics.*" *J. Log. Algebraic Methods Program.* 60-61 (2004), pages 3–15 (cited on page 47).

[PNB17] M. Preiner, A. Niemetz, and A. Biere. "*Counterexample-Guided Model Synthesis.*" *TACAS (1)*. Volume 10205. Lecture Notes in Computer Science. 2017, pages 264–280 (cited on page 199).

[Pou07] D. Pous. "*Complete Lattices and Up-To Techniques.*" *APLAS*. Volume 4807. Lecture Notes in Computer Science. Springer, 2007, pages 351–366 (cited on page 164).

[PS12] D. Pous and D. Sangiorgi. "*Enhancements of the bisimulation proof method.*" *Advanced Topics in Bisimulation and Coinduction*. Volume 52. Cambridge tracts in theoretical computer science. Cambridge University Press, 2012, pages 233–289 (cited on page 164).

[Put94] M. L. Puterman. "*Markov Decision Processes: Discrete Stochastic Dynamic Programming.*" Wiley Series in Probability and Statistics. Wiley, 1994 (cited on pages 31, 44).

[QK18] T. Quatmann and J. Katoen. “*Sound Value Iteration*.” *CAV (1)*. Volume 10981. Lecture Notes in Computer Science. Springer, 2018, pages 643–661 (cited on pages 163, 200).

[Ram79] L. Ramshaw. “*Formalizing the analysis of algorithms*.” PhD thesis. Stanford University, USA, 1979 (cited on page 5).

[RBF18] A. Reynolds, H. Barbosa, and P. Fontaine. “*Revisiting Enumerative Instantiation*.” *TACAS (2)*. Volume 10806. Lecture Notes in Computer Science. Springer, 2018, pages 112–131 (cited on page 199).

[RDKB$^+$15] A. Reynolds, M. Deters, V. Kuncak, C. W. Barrett, and C. Tinelli. “*On Counterexample Guided Quantifier Instantiation for Synthesis in CVC4*.” *CoRR* abs/1502.04464 (2015) (cited on page 199).

[Rey02] J. C. Reynolds. “*Separation Logic: A Logic for Shared Mutable Data Structures*.” *LICS*. IEEE Computer Society, 2002, pages 55–74 (cited on pages 118, 119).

[RICB15] H. Rocha, H. Ismail, L. C. Cordeiro, and R. S. Barreto. “*Model Checking Embedded C Software Using k-Induction and Invariants*.” *SBESC*. IEEE Computer Society, 2015, pages 90–95 (cited on page 163).

[RKSB$^+$21] F. Ronquist, J. Kudlicka, V. Senderov, J. Borgström, N. Lartillot, D. Lundén, L. Murray, T. B. Schön, and D. Broman. “*Universal probabilistic programming offers a powerful approach to statistical phylogenetics*.” *Communications Biology* 4.1, 244 (2021) (cited on page 4).

[Rob49] J. Robinson. “*Definability and Decision Problems in Arithmetic*.” *J. Symb. Log.* 14.2 (1949), pages 98–114 (cited on page 97).

[RS09] Rabehaja and Sanders. “*Refinement Algebra with Explicit Probabilism*.” *TASE*. 2009 (cited on page 143).

[RTGK$^+$13] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. W. Barrett. “*Quantifier Instantiation Techniques for Finite Model Finding in SMT*.” *CADE*. Volume 7898. Lecture Notes in Computer Science. Springer, 2013, pages 377–391 (cited on page 199).

[RTM14] A. Reynolds, C. Tinelli, and L. M. de Moura. “*Finding conflicting instances of quantified formulas in SMT*.” *FMCAD*. IEEE, 2014, pages 195–202 (cited on page 199).

[RWKY$^+$14] M. N. Rabe, C. M. Wintersteiger, H. Kugler, B. Yordanov, and Y. Hamadi. “*Symbolic Approximation of the Bounded Reachability Probability in Large Markov Chains.*” *QEST*. Volume 8657. Lecture Notes in Computer Science. Springer, 2014, pages 388–403 (cited on page 200).

[RZ15] R. Rand and S. Zdancewic. “*VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs.*” *MFPS*. Volume 319. Electronic Notes in Theoretical Computer Science. Elsevier, 2015, pages 351–367 (cited on page 5).

[SI20] K. Suenaga and T. Ishizawa. “*Generalized Property-Directed Reachability for Hybrid Systems.*” *VMCAI*. Volume 11990. LNCS. Springer, 2020, pages 293–313 (cited on pages 224, 255).

[SO19] A. Schreuder and C. L. Ong. “*Polynomial Probabilistic Invariants and the Optional Stopping Theorem.*” *CoRR* abs/1910.12634 (2019) (cited on page 118).

[SRBE05] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. “*Programming by sketching for bit-streaming programs.*” *PLDI*. ACM, 2005, pages 281–294 (cited on page 165).

[SS17] T. Seufert and C. Scholl. “*Sequential Verification Using Reverse PDR.*” *MBMV*. Shaker Verlag, 2017, pages 79–90 (cited on page 202).

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. “*Checking Safety Properties Using Induction and a SAT-Solver.*” *FMCAD*. Volume 1954. Lecture Notes in Computer Science. Springer, 2000, pages 108–125 (cited on pages 6, 125–127, 163).

[STBS$^+$06] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. “*Combinatorial sketching for finite programs.*” *ASPLOS*. ACM, 2006, pages 404–415 (cited on page 165).

[Stu22] B. Sturgis. “*Automatic Verification of Loop Invariants in Weighted Programs.*” RWTH Aachen University, Germany, 2022 (cited on page 163).

[Tar48] A. Tarski. “*A Decision Method for Elementary Algebra and Geometry.*” Project rand. Rand Corporation, 1948 (cited on page 197).

[Tar55] A. Tarski. "*A LATTICE-THEORETICAL FIXPOINT THEOREM AND ITS APPLICATIONS.*" *Pacific Journal of Mathematics* 5 (1955), pages 285–309 (cited on pages 20, 136).

[TCA09] M. Tatsuta, W. Chin, and M. F. A. Ameen. "*Completeness of Pointer Program Verification by Separation Logic.*" *Software Engineering and Formal Methods*. IEEE Computer Society, 2009, pages 179–188 (cited on pages 94, 119).

[TCA19] M. Tatsuta, W. Chin, and M. F. A. Ameen. "*Completeness and expressiveness of pointer program verification by separation logic.*" *Inf. Comput.* 267 (2019), pages 1–27 (cited on pages 94, 119).

[TOUH18] T. Takisaka, Y. Oyabu, N. Urabe, and I. Hasuo. "*Ranking and Repulsing Supermartingales for Reachability in Probabilistic Programs.*" *ATVA*. Volume 11138. Lecture Notes in Computer Science. Springer, 2018, pages 476–493 (cited on pages 118, 198).

[TOUH21] T. Takisaka, Y. Oyabu, N. Urabe, and I. Hasuo. "*Ranking and Repulsing Supermartingales for Reachability in Randomized Programs.*" *ACM Trans. Program. Lang. Syst.* 43.2 (2021), 5:1–5:46 (cited on pages 118, 198).

[Tur37] A. M. Turing. "*On computable numbers, with an application to the Entscheidungsproblem.*" 2.1 (1937), pages 230–265 (cited on page 2).

[UTK21] H. Unno, T. Terauchi, and E. Koskinen. "*Constraint-Based Relational Verification.*" *CAV (1)*. Volume 12759. Lecture Notes in Computer Science. Springer, 2021, pages 742–766 (cited on pages 197, 199).

[VG14] Y. Vizel and A. Gurfinkel. "*Interpolating Property Directed Reachability.*" *CAV*. Volume 8559. Lecture Notes in Computer Science. Springer, 2014, pages 260–276 (cited on page 255).

[WBB09] R. Wimmer, B. Braitling, and B. Becker. "*Counterexample Generation for Discrete-Time Markov Chains Using Bounded Model Checking.*" *VMCAI*. Volume 5403. Lecture Notes in Computer Science. Springer, 2009, pages 366–380 (cited on page 163).

[Wel13] T. Welp. "*Program Verification with Property Directed Reachability.*" PhD thesis. University of California, Berkeley, USA, 2013 (cited on page 255).

[WFGC$^+$19] P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi. "*Cost analysis of nondeterministic probabilistic programs.*" *PLDI*. ACM, 2019, pages 204–220 (cited on page 198).

[Wil07] D. R. Wilkins. "*Course 221: Hilary Term 2007. Section 6: The Extended Real Number System.*" Lecture Notes. Accessed online November 20, 2023. 2007 (cited on page 35).

[Win93] G. Winskel. "*The formal semantics of programming languages - an introduction.*" Foundation of computing series. MIT Press, 1993 (cited on pages 21, 22, 77, 79, 82, 93, 94, 96, 102, 110, 119).

[WK23a] T. Winkler and J.-P. Katoen. "*Certificates for Probabilistic Pushdown Automata via Optimistic Value Iteration.*" *TACAS (2)*. Volume 13994. Lecture Notes in Computer Science. Springer, 2023, pages 391–409 (cited on pages 7, 123, 163).

[WKH20] D. Wang, D. M. Kahn, and J. Hoffmann. "*Raising expectations: automating expected cost analysis with types.*" *Proc. ACM Program. Lang.* 4.ICFP (2020), 110:1–110:31 (cited on page 198).

[YFKZ$^+$24] T. Yang, H. Fu, J. Ke, N. Zhan, and S. Wu. "*Piecewise Linear Expectation Analysis via k-Induction for Probabilistic Programs.*" *CoRR* abs/2403.17567 (2024) (cited on pages 7, 123).

Co-authored Publications

Cited publications

[1] K. Batz. *“Proof Rules for Expected Runtimes of Probabilistic Programs.”* RWTH Aachen University, Germany, 2017. DOI: https://doi.org/10.5281/zenodo.11576700 (cited on page 287).

[2] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. *“How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times.” ESOP*. Volume 10801. Lecture Notes in Computer Science. Springer, 2018, pages 186–213 (cited on pages 9, 287).

[3] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and T. Noll. *“Quantitative Separation Logic.” CoRR* abs/1802.10467 (2018) (cited on pages 69, 73).

[4] K. Batz. *“IC3 for Probabilistic Systems.”* RWTH Aachen University, Germany, 2019. DOI: https://doi.org/10.5281/zenodo.11576982 (cited on pages 202, 287).

[5] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and T. Noll. *“Quantitative Separation logic: A Logic for Reasoning about Probabilistic Pointer Programs.” Proc. ACM Program. Lang.* 3.POPL (2019), 34:1–34:29 (cited on pages 9, 47, 69, 73, 118, 287).

[6] K. Batz, S. Junges, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. *“PrIC3: Property Directed Reachability for MDPs.” CAV (2)*. Volume 12225. Lecture Notes in Computer Science. Springer, 2020, pages 512–538 (cited on pages 7, 8, 201, 287).

[7] K. Batz, S. Junges, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. *“PrIC3: Property Directed Reachability for MDPs.” CoRR* abs/2004.14835 (2020) (cited on pages 201, 287).

[8] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. *“Relatively Complete Verification of Probabilistic Programs.” CoRR* abs/2010.14548 (2020) (cited on pages 75, 101, 111).

[9] L. Klinkenberg, K. Batz, B. L. Kaminski, J.-P. Katoen, J. Moerman, and T. Winkler. *“Generating Functions for Probabilistic Programs.” LOPSTR*. Volume 12561. Lecture Notes in Computer Science. Springer, 2020, pages 231–248 (cited on pages 9, 286).

[10] K. Batz, M. Chen, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. “*Latticed k-Induction with an Application to Probabilistic Programs.*” *CoRR* abs/2105.14100 (2021) (cited on pages 121, 137, 138, 155).

[11] K. Batz, M. Chen, B. L. Kaminski, J.-P. Katoen, C. Matheja, and P. Schröer. “*Latticed k-Induction with an Application to Probabilistic Programs.*” *CAV (2)*. Volume 12760. Lecture Notes in Computer Science. Springer, 2021, pages 524–549 (cited on pages 6, 8, 121, 155, 286).

[12] K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. “*Relatively Complete Verification of Probabilistic Programs: An Expressive Language for Expectation-based Reasoning.*” *Proc. ACM Program. Lang.* 5.POPL (2021), pages 1–30 (cited on pages 6, 8, 60, 75, 83, 84, 286).

[13] K. Batz, M. Chen, S. Junges, B. L. Kaminski, J.-P. Katoen, and C. Matheja. “*Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants.*” *CoRR* abs/2205.06152 (2022) (cited on page 165).

[14] K. Batz, I. Fesefeldt, M. Jansen, J.-P. Katoen, F. Keßler, C. Matheja, and T. Noll. “*Foundations for Entailment Checking in Quantitative Separation Logic.*” *ESOP*. Volume 13240. Lecture Notes in Computer Science. Springer, 2022, pages 57–84 (cited on pages 8, 286).

[15] K. Batz, A. Gallus, B. L. Kaminski, J.-P. Katoen, and T. Winkler. “*Weighted Programming: A Programming Paradigm for Specifying Mathematical Models.*” *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pages 1–30 (cited on pages 8, 55, 163, 197, 286).

[16] K. Batz, M. Chen, S. Junges, B. L. Kaminski, J.-P. Katoen, and C. Matheja. “*Probabilistic Program Verification via Inductive Synthesis of Inductive Invariants.*” *TACAS (2)*. Volume 13994. Lecture Notes in Computer Science. Springer, 2023, pages 410–429 (cited on pages 7, 8, 165, 285).

[17] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and L. Verscht. “*A Calculus for Amortized Expected Runtimes.*” *Proc. ACM Program. Lang.* 7.POPL (2023), pages 1957–1986 (cited on pages 8, 69, 73, 285).

[18] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. “*A Deductive Verification Infrastructure for Probabilistic Programs.*” *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pages 2052–2082 (cited on pages 6, 8, 79, 145, 157, 257, 285).

[19] K. Batz, T. J. Biskup, J.-P. Katoen, and T. Winkler. “*Programmatic Strategy Synthesis: Resolving Nondeterminism in Probabilistic Programs.*” *Proc. ACM Program. Lang.* 8.POPL (2024), pages 2792–2820 (cited on pages 8, 47, 258, 263, 285).

Index