TU WIEN Informatics

Automated Analysis of Probabilistic Loops

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

DI Marcel Moosbrugger, BSc
Registration Number 01426526

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof Dr.techn. Laura Kovács, MSc
Second advisor: Univ.Prof. Dr. Ezio Bartocci

The dissertation has been reviewed by:

Christel Baier

Joël Ouaknine

Vienna, June 24, 2024

Marcel Moosbrugger

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Erklärung zur Verfassung der Arbeit

DI Marcel Moosbrugger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Juni 2024

Marcel Moosbrugger

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Acknowledgements

Completing this Ph.D. thesis has been an exciting and transformative journey, and I could not have reached this milestone without the support, guidance, and encouragement of many individuals. First and foremost I want to extend my deepest appreciation to my advisor Laura Kovács who went above and beyond her regular duties to guide me through the ups and downs of this journey. Her dedication, patience, and unwavering support have been invaluable to me. I would also like to extend my gratitude to my co-advisor Ezio Bartocci who was a constant source of inspiration, as well as to my colleague Mirolsav Stankovič whose research paved the way for the questions addressed in this thesis.

A special "thank you" goes to everyone in our research group and to all my collaborators for their support and friendship. Supervising Julian for his master's thesis was a pleasure. His brilliant ideas and technical knowledge made the process effortless, and our joint research sessions were a highlight of the final year. I also want to thank Benjamin Kaminski whose research, ideas, and feedback have always been sources of inspiration.

Moreover, I am deeply grateful to Joost-Pieter Katoen for his significant support and the opportunity to collaborate and work with his group. The scientific and general discussions with his students Christopher, Kevin, Lutz, and Tobias were always inspiring and helpful. Further, I want to thank Martina Maggio for graciously inviting me to Saarbrücken and introducing me to new potential applications of my research.

On a personal note, I am deeply indebted to my brothers, Daniel and Julian, and my parents, Melitta and Markus. Thank you for your unwavering support and encouragement throughout my life.

The research in this thesis was supported by the ERC Starting Grant SYMCAR 639270, the ERC Consolidator Grant ARTIST 101002685, ERC Advanced Grant FRAPPANT 787914, the Vienna Science and Technology Fund WWTF 10.47379/ICT19018 grant ProbInG, and the SecInt Doctoral College funded by TU Wien.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

This thesis addresses challenges and advancements in the automated analysis of probabilistic loops. It contributes to the theoretical foundations and computational techniques for analyzing the safety and liveness of probabilistic loops.

A core contribution is the development of a fully automated method for computing higher moments of program variables for a large class of probabilistic loops. This method leverages linear recurrences with constant coefficients to model higher moments of loop variables and compute their exact closed-form expressions. Introducing the theory of moment-computable loops, we show that our approach is complete for a class of programs supporting branching statements, polynomial arithmetic, and both discrete and continuous probability distributions. For probabilistic systems with unknown model parameters, we introduce a novel technique for automatic sensitivity analysis with respect to these parameters. By representing unknown parameters with symbolic constants and modeling sensitivities using recurrences, we show that our technique is applicable even to loops that are not moment-computable.

Furthermore, this thesis explores hardness bounds for computing the strongest polynomial invariant for classical polynomial loops, showing that this problem Skolem-hard. As an intermediary result of independent interest, we show that point-to-point reachability for polynomial dynamical systems is also Skolem-hard. Through the notion of moment invariant ideals, we extend these hardness results from classical to probabilistic program analysis. Despite the hardness results, we propose a method for computing polynomial invariants of bounded degree for (probabilistic) polynomial loops and a synthesis procedure to over-approximate polynomial loops with linearizable loops.

Additionally, the thesis introduces Polar, a tool implementing the developed techniques, demonstrating its capability to analyze benchmarks previously out of reach for state-of-the-art methods.

Regarding termination analysis, we propose a novel approach based on asymptotic bounds for polynomial probabilistic loops, leading to the development of Amber, the first tool to certify both probabilistic termination and non-termination.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Diese Dissertation beschreibt Herausforderungen und Fortschritte in der automatisierten Analyse von probabilistischen Schleifen, bezüglich ihrer Sicherheit und Lebendigkeit.

Ein Kernbeitrag ist die Entwicklung einer vollautomatischen Methode zur Berechnung höherer Momente von Programmvariablen für eine große Klasse von probabilistischen Schleifen. Diese Methode nutzt lineare Rekurrenzen mit konstanten Koeffizienten, um höhere Momente von Schleifenvariablen zu modellieren und exakte geschlossenen Ausdrücke für sie zu berechnen. Mit der Einführung der Theorie der moment-berechenbaren Schleifen zeigen wir, dass unser Ansatz vollständig für eine Klasse von Programmen ist, die Verzweigungsanweisungen, polynomielle Arithmetik und sowohl diskrete als auch kontinuierliche Wahrscheinlichkeitsverteilungen unterstützen. Für probabilistische Systeme mit unbekannten Modellparametern führen wir eine neue Technik für die automatische Sensitivitätsanalyse bezüglich dieser Parameter ein. Indem wir unbekannte Parameter mit symbolischen Konstanten darstellen und Sensitivitäten mit Rekurrenzen modellieren, zeigen wir, dass unsere Technik sogar auf Schleifen anwendbar ist, die nicht moment-berechenbar sind.

Darüber hinaus untersucht diese Dissertation Härtegrenzen für die Berechnung der stärksten polynomiellen Invarianten für klassische polynomielle Schleifen. Wir zeigen, dass dieses Problem Skolem-hart ist. Als ein Zwischenergebnis von eigenem Interesse zeigen wir, dass die Punkt-zu-Punkt-Erreichbarkeit für polynomielle dynamische Systeme ebenfalls Skolem-hart ist. Durch das Konzept der Moment-Invarianten-Ideale erweitern wir diese Härteergebnisse von der klassischen zur probabilistischen Programmanalyse. Trotz der Härteergebnisse entwickeln wir eine Methode zur Berechnung von polynomiellen Invarianten begrenzten Grades für (probabilistische) polynomielle Schleifen sowie ein Syntheseverfahren, um polynomielle Schleifen mit linearisierbaren Schleifen zu überapproximieren.

Zusätzlich führt die Dissertation Polar ein, ein Tool, das die entwickelten Techniken implementiert und Benchmarks analysieren kann, die bisher nicht gelöst werden konnten.

Bezüglich Terminierungsanalyse führen wir einen neuen Ansatz basierend auf asymptotischen Grenzen für polynomielle probabilistische Schleifen ein, was zur Entwicklung von Amber, dem ersten Tool führt, das sowohl probabilistische Terminierung als auch Nicht-Terminierung beweisen kann.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Contents

Abstract **vii**

Kurzfassung **ix**

Contents **xi**

1 Overview **1**
1.1 Probabilistic Programs 2
1.2 Algebraic Loop Analysis 4
1.3 Contributions & Synopsis 5
1.4 Impact of the Thesis 7

2 Computing Moments **11**
2.1 Problem Statement 11
2.2 Preliminaries 15
2.3 Probabilistic Program Model 16
2.4 Finite Types in Probabilistic Programs 23
2.5 Computing Higher Moments of Probabilistic Programs 25
2.6 Use-Cases of Higher Moments 34
2.7 Implementation and Evaluation 37
2.8 Related Work 41
2.9 Conclusion 42

3 Automated Sensitivity Analysis **45**
3.1 Problem Statement 45
3.2 Preliminaries 48
3.3 Sensitivity Analysis 51
3.4 Experiments and Evaluation 59
3.5 Related Work 62
3.6 Conclusion 63

4 Strong Invariants Are Hard **65**
4.1 Problem Statement 65
4.2 Preliminaries 69

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3 Hardness of Reachability in Polynomial Programs 74
4.4 Hardness of Computing the Strongest Polynomial Invariant 81
4.5 Strongest Invariant for Probabilistic Loops 84
4.6 Related Work 94
4.7 Conclusion 96

5 (Un)Solvable Loop Analysis 99
5.1 Problem Statement 99
5.2 Preliminaries 103
5.3 From Loops to Recurrences 106
5.4 Defective Variables 108
5.5 Synthesising Invariants 112
5.6 Synthesising Solvable Loops from Unsolvable Loops 115
5.7 Adjustments for Unsolvable Operators in Probabilistic Programs . . . 119
5.8 Applications of Unsolvable Operators towards Invariant Generation . . 121
5.9 Experiments 129
5.10 Conclusion 133

6 Automated Termination Analysis 135
6.1 Problem Statement 135
6.2 Preliminaries 139
6.3 Proof Rules for Probabilistic Termination 143
6.4 Relaxed Proof Rules for Probabilistic Termination 145
6.5 Algorithmic Termination Analysis through Asymptotic Bounds 148
6.6 Implementation and Evaluation 161
6.7 Related Work 166
6.8 Conclusion 168

7 The Probabilistic Termination Tool Amber 169
7.1 Problem Statement 169
7.2 Preliminaries 171
7.3 Amber: Programming Model 172
7.4 Proof Rules for Probabilistic Termination 173
7.5 Effective Termination Analysis through Asymptotic Bounds 176
7.6 Amber: Implementation and Components 184
7.7 Evaluation 187
7.8 Conclusion 190

8 Summary & Outlook 191

Bibliography 195

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 1

Overview

Software systems and programs are omnipresent in our modern world and deeply embedded in virtually every aspect of our daily lives. Ensuring the correctness and quality of programs is thus of utmost importance. This is especially true for programs employed in safety-critical environments. Manual inspection of program code is cumbersome and error-prone, rendering an automated approach inevitable. Program analysis is a term subsuming a plethora of methods and techniques for automatically inferring various properties of programs. The inferred properties can then enable different tasks such as finding bugs and security issues or reasoning about the correctness of programs.

Loops are fundamental program constructs that enable the definition of repetitive tasks in programs and as such constitute an integral part of almost every software system. However, the convenience and power of loops come at a price: loops are the main source of undecidability and uncomputability in program analysis. This fact has been known since the early days of computer science. Alan Turing showed in 1937 that the most fundamental liveness property, termination, is undecidable [Tur37]. In 1953, Rice proved that *every* non-trivial property of the behavior of a program is undecidable [Ric53]. Contrary to what these negative results might suggest, program analysis is an active area of research up to this day. The barriers posed by undecidability can be circumvented by resorting to semi-algorithms and approximation.

Allowing programs to draw from probability distributions or execute different parts of their code based on coin flips further adds to the difficulty of program analysis posed by loops. For instance, the termination problem is strictly harder for probabilistic loops [KKM19].

In this thesis, we confront the challenges posed by integrating loops with probabilities head-on. We develop automated techniques for the analysis of probabilistic loops, which are the crux of probabilistic program analysis.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
x, y = 0, 0
while ⋆:
    d = Bernoulli(1/2)
    if d == 0:
        x = x+1 {1/2} x−1
    else:
        y = y+1 {1/2} y−1
    end
end
```

(a)

```
stop = 0
count = 0
x = 1
while stop == 0:
    stop = Bernoulli(1/2)
    count = count+1
    x = 2∗x
end
```

(b)

Figure 1.1: Two examples of probabilistic programs.

1.1 Probabilistic Programs

Probabilistic programming languages enrich classical imperative or functional languages with native primitives to draw samples from random distributions, such as Bernoulli, Uniform, and Normal distributions. The resulting probabilistic programs (PPs) [Koz85, BKS20a] embed uncertain quantities, represented by random variables, within standard program control flows. As such, PPs offer a unifying framework to naturally encode probabilistic machine learning models [Gha15], for example, Bayesian networks [KKMO16], into programs. Moreover, PPs enable programmers to handle uncertainty resulting from sensor measurements and environmental perturbations in cyber-physical systems [SRB$^+$15, CYS20]. Other notable examples of PPs include the implementation of cryptographic [BGB12] and privacy [BKOB12] protocols, as well as randomized algorithms [MR95].

Fig. 1.1 shows two probabilistic loops. The infinite loop from Fig. 1.1a models a symmetric random walk in the XY-plane. In every iteration, a draw from a Bernoulli distribution with parameter $1/2$ determines whether the process moves in the x or y dimension. Depending on the outcome of the draw, either the variable x or the variable y is reassigned. The respective variable is incremented or decremented by 1, each with probability $1/2$. The probabilistic loop from Fig. 1.1b contains a non-trivial loop guard. It terminates whenever the draw from the Bernoulli distribution in its loop body is 0. The variable *count* counts the number of loop iterations until termination while x doubles in every iteration.

PPs can be viewed from two different perspectives: first as randomized algorithms and second as succinct descriptions of complex probability distributions.

PPs as randomized algorithms. In randomized algorithms, a source of randomness is used to influence their outcomes or runtimes [MR95]. For some problems, the use of randomization gives rise to algorithms whose worst-case expected runtimes are significantly faster compared to the worst-case runtimes of their deterministic versions. For instance,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Quicksort is a sorting algorithm with a worst-case runtime of $O(n^2)$. Randomly shuffling the inputs before sorting leads to a randomized version of Quicksort with a worst-case expected runtime of $O(n \log(n))$. A further notable use of randomness in algorithms is to break symmetries. In distributed systems, indistinguishable processes can self-stabilize by using randomness to break the symmetry between them [Sch93].

PPs as distributions. Program variables of PPs encode uncertain quantities. The outcome of a single run of a PP amounts to a draw from a particular probability distribution. Thus, PPs effectively describe complex probability distributions by integrating known distributions such as Bernoulli and Normal within standard program constructs. This integration facilitates building more intricate distributions in a structured and well-defined manner. In special cases, these distributions can be characterized by "simpler means". For example, the variable *count* in the program from Fig 1.1b counts the number of Bernoulli trials until the first success. Hence, upon termination of the program, the variable is fully described by a Geometric distribution. Generally, however, the distributions of PPs are considerably more complex. Distributions and even expected values of program variables upon termination are uncomputable for arbitrary PPs [KKM19].

Despite the challenge of uncomputability, the expressive power of PPs offers notable benefits. Historically, various graphical models such as Bayesian networks or Markov networks were used to model probabilistic systems [KF09]. PPs extend these models, providing a unified and structured framework that eliminates the need for distinct syntaxes, inference methods, and analysis algorithms [BKS20a]. As such, a wide range of systems can be modeled and analyzed in a unifying manner using PPs. In this context, PPs are created as models for analysis and query purposes, unlike classical programs designed for execution. Therefore, not only guarded loops but also infinite loops serve an important purpose in PPs. Infinite probabilistic loops can model stochastic processes. An example is the loop from Fig. 1.1a modeling a symmetric random walk in two dimensions. For infinite loops, distributions or moments of program variables can be given parameterized by the number of loop iterations n.

Fig. 1.2a presents closed-form expressions for the expected values and second moments of the variables for the PP from Fig. 1.1a. Guarded loops can be treated as infinite loops by assuming that the values of program variables remain constant after the loop guard is falsified. Fig. 1.2b shows closed-forms for specific moments of the guarded probabilistic loop from Fig. 1.1b. Moments after termination derive from analyzing the limiting behavior of their closed-forms (detailed in Chapter 2). For instance, the closed-form of the expected value of *count* in Fig. 1.2b implies that its expected value after termination and hence the expected runtime of the loop is 2. In contrast, the expected value of the variable x post-termination is infinite. The possibility of expected values of program variables being infinite after termination, even for loops with finite expected runtime, illustrates the limitations of sampling methods and the need for formal techniques in PP analysis. With sampling alone, it is impossible to determine that the expected value of x is infinite after termination because every concrete sample for x is a finite value.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Closed-forms:

$\mathbb{E}(x_n) = \mathbb{E}(y_n) = 0$
$\mathbb{E}(x_n^2) = \mathbb{E}(y_n^2) = n/2$

Invariants:

$\mathbb{E}(x) = \mathbb{E}(y) = 0$
$\mathbb{E}(x^2) - \mathbb{E}(y^2) = 0$

(a) Closed-forms and invariants for the PP from Fig. 1.1a.

Closed-forms:

$\mathbb{E}(stop_n) = 1 - 1/2^n$
$\mathbb{E}(count_n) = 2 - 2/2^n$
$\mathbb{E}(count_n^2) = (3 \cdot 2^n - 2n - 3) \cdot 2/2^n$
$\mathbb{E}(x_n) = n + 1$

Invariants:

$2\mathbb{E}(stop) - \mathbb{E}(count) = 0$

$$2\mathbb{E}(count)\mathbb{E}(x) + \mathbb{E}(count) - \mathbb{E}(count^2) - 4\mathbb{E}(x) + 4 = 0$$

(b) Closed-forms and invariants for the PP from Fig. 1.1b.

Figure 1.2: Closed-forms and invariants for probabilistic programs from Fig. 1.1.

Similar to classical programs, decidability issues arise with PPs containing loops. The two central classes of properties in the analysis of classical and probabilistic programs are *safety* and *liveness* properties. Intuitively, safety properties state that nothing bad happens and involve capturing the set of reachable program states by invariants. Liveness properties express something good happening eventually and are closely related to the termination problem. The specification of probabilistic models as programs allows their analysis using techniques from program languages and verification. *In this thesis, we develop program analysis techniques in the setting of liveness as well as safety for probabilistic loops, which are at the center of automated PP analysis.*

1.2 Algebraic Loop Analysis

In program loops, the values of program variables after any iteration are calculated based on their values in previous iterations. Similarly, *algebraic recurrence equations*, or just *recurrences*, relate elements of a sequence to preceding elements [EvdPSW03]. This parallel makes recurrences a natural tool for analyzing loops, enabling the application of recurrence theory for program analysis.

For example, ignoring the loop guard in the program from Fig. 1.1b, the values of the program variable x can be characterized by the linear recurrence equation $x(n+1) = 2 \cdot x(n)$ with initial value $x(0) = 1$. This recurrence admits the closed-form solution $x(n) = 2^n$ which allows for the computation of x after n iterations without executing the loop. Recurrences have been widely recognized as vital in classical program analysis [RcK04, RK07, Kov08, HJK18b, FK15, KCBR18, KBCR19, BCKR20]. However,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

applying algebraic recurrences to probabilistic loop analysis presents challenges, as program variables do not adhere to a predetermined numeric sequence. In Chapter 2, we introduce a novel method utilizing recurrences for the analysis of probabilistic loops, facilitating computing the closed-form expressions presented in Fig. 1.2 fully automatically

C-finite recurrences are linear recurrence equations with constant coefficients and constitute a class of recurrences that is particularly tractable [KP11]. C-finite recurrences obey strong closure properties and their solutions always exist and are computable, rendering them particularly suitable for program analysis. Indeed, many loops can be effectively analyzed using this class of recurrences [Kov08]. For instance, as we establish in Chapter 2, all moments of all variables for the PPs in Fig. 1.1 can be modeled by C-finite recurrences.

As previously mentioned in Section 1.1, invariants play an essential role in the safety analysis of programs. An important class of invariants are *polynomial invariants*. For classical loops, polynomial invariants are polynomials in program variables that evaluate to 0 after *every* loop iteration. Notably, the set of polynomial invariants associated with a given loop always forms an *ideal* – a concept from algebraic geometry [CLO97]. This relationship allows for the application of techniques from computational algebraic geometry in computing and studying polynomial invariants. We extend the concepts of polynomial invariants and invariant ideals to the probabilistic setting in Chapter 4. Figure 1.2 shows some polynomial invariants over moments of program variables for the PPs from Figure 1.1. With the theory and techniques presented in this thesis, all polynomials from Figure 1.2 can be computed fully automatically.

Integrating the theory of C-finite recurrences and computational algebraic geometry proves immensely useful in program analysis. For instance, invariant ideals can be derived from the closed-form expressions of specific C-finite recurrences using techniques from computational algebraic geometry. A recurring theme throughout this thesis is the adoption of approaches from both fields to develop and study techniques for the automated analysis of probabilistic loops.

1.3 Contributions & Synopsis

Each chapter of this thesis is based on one or two peer-reviewed publications, which are listed at the beginning of every chapter. Designed to be self-contained, every chapter introduces its notation, preliminaries, and related work, allowing for independent reading from the rest of the thesis. The contributions of this thesis are as follows.

Chapter 2 – Computing Moments. In this chapter, we develop a novel static analysis technique that utilizes recurrences to compute exact closed-form expressions for higher moments of program variables for a large class of probabilistic loops. Additionally, we present the concept of *moment-computable* PPs, a class of probabilistic loops for which our technique is complete. Moment-computable PPs adhere to specific restrictions concerning branching statements and the arithmetic within their loop bodies. As we show, removing

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the restriction on branching statements immediately leads to uncomputability. We present the new tool Polar implementing all techniques from this chapter and demonstrate applications of exact higher moment computation in computing tail probabilities and recovering probability distributions of loop variables. Furthermore, through experimental evaluation, we highlight the practicality of Polar, successfully solving several benchmarks previously beyond the reach of the state-of-the-art probabilistic program analysis.

Chapter 3 – Automated Sensitivity Analysis. When modeling stochastic processes using probabilistic loops, values or even distributions of model parameters are often unknown. To manage this lack of information, we can represent unknown model parameters with symbolic constants that stand for any real number. Sensitivity analysis of (higher) moments of program variables aims to quantify how small changes in these parameters affect the variable moments. In this chapter, we introduce an exact technique to compute sensitivities of (higher) moments of probabilistic loop variables on symbolic parameters. We introduce the notion of *sensitivity recurrence* enabling sensitivity analysis even for probabilistic loops that are not moment-computable.

Chapter 4 – Strong Invariants Are Hard. In addition to branching statements and probabilities, the complexity of automated loop analysis significantly depends on the type of arithmetic permitted within the loop bodies. For classical loops containing only linear assignments, the strongest polynomial invariant, or its invariant ideal, is known to be computable. However, even for classical loops with polynomial assignments, this problem remained wide open. In this chapter, we show that computing the strongest polynomial invariant for polynomial loops is at least as hard as the Skolem problem, a famous problem whose decidability has been open for almost a century. As an intermediate result of independent interest, we prove that point-to-point reachability for polynomial loops is Skolem-hard as well. Furthermore, we introduce the notion of *moment invariant ideals* and show that they generalize the strongest polynomial invariant for classical loops to probabilistic loops. This generalization allows us to transfer hardness results for classical loops to the probabilistic setting. While we show that moment invariant ideals are computable for the probabilistic loops supported by our static analyzer Polar, our hardness results justify that no restriction on Polar's input programs can be lifted without running into serious hardness boundaries.

Chapter 5 – (Un)Solvable Loop Analysis. Program loops with arbitrary polynomial assignments are termed *unsolvable*, whereas those that can be linearized are referred to as *solvable*. The results from Chapter 4 demonstrate that computing the strongest polynomial invariant for unsolvable loops is out of reach. Despite this, in this chapter, we introduce a novel method for synthesizing polynomial invariants of bounded degree for unsolvable loops. Our approach automatically partitions program variables and identifies so-called *defective* variables that characterize unsolvability. We present a technique that automatically synthesizes polynomials in defective variables that admit closed-form solutions and thus lead to polynomial loop invariants. Moreover, we propose a synthesis

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

procedure that creates an over-approximation of an unsolvable loop in terms of a solvable loop. While the theoretical foundation of this chapter primarily addresses classical loops, we also demonstrate its applicability to probabilistic loops, showcasing its broader relevance.

Chapter 6 – Automated Termination Analysis. In stark contrast to classical loops, probabilistic loops can terminate with probability 1 yet exhibit an infinite expected runtime. This distinctive feature necessitates a nuanced approach to termination analysis, differentiating between almost-sure termination (AST) – which denotes termination with probability 1 – and positive almost-sure termination (PAST), characterized by finite expected runtime. In this chapter, we present a fully automated approach to the termination analysis of probabilistic loops whose guards and assignments are polynomial expressions. Because proving (P)AST is undecidable in general, existing proof rules typically provide sufficient conditions. We refine and expand upon four proof rules from the literature. We effectively automate the extended proof rules by computing asymptotic bounds on polynomials in program variables using recurrences. The resulting algorithms and software tool Amber can check AST, PAST as well as their negations for a large class of polynomial probabilistic loops. Our experimental results underscore the advantages of our generalized proof rules and demonstrate that Amber can analyze probabilistic loops that are out of reach for other state-of-the-art tools.

Chapter 7 – The Probabilistic Termination Tool Amber. In this chapter, we provide a comprehensive overview of our probabilistic termination analysis tool Amber, as introduced in the preceding chapter. We delve into the methodologies employed by Amber and extend the method of asymptotic bounds to analyze probabilistic termination by further supporting symbolic constants and common probability distributions, which may be continuous, discrete, finitely- or infinitely supported. Furthermore, we expand upon the experimental evaluation of Amber. This includes an increased number of benchmarks and a comparison with other state-of-the-art tools, offering insights into Amber performance and applicability.

Chapter 8 – Summary & Outlook. In this chapter, we conclude the thesis and provide a discussion on potential avenues for future work that could build upon our findings.

1.4 Impact of the Thesis

Included publications. The results presented in this thesis stem from a series of peer-reviewed publications. The author of this thesis has served as either the main contributor or one of the main contributors in all listed works.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[MBKK21a] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. Automated Termination Analysis of Polynomial Probabilistic Programs. In Proc. of ESOP, 2021.

[MBKK21b] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. In Proc. of FM, 2021.

[MBKK22] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. Formal Methods Syst. Des., 2022.

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving Invariant Generation for Unsolvable Loops. In Proc. of SAS, 2022.

[MSBK22a] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This Is the Moment for Probabilistic Loops. Proc. ACM Program. Lang., (OOPSLA2), 2022.

[MMK23] Marcel Moosbrugger, Julian Müllner, and Laura Kovács. Automated Sensitivity Analysis for Probabilistic Loops. In Proc. of iFM, 2023.

[MMK24] Julian Müllner, Marcel Moosbrugger, and Laura Kovács. Strong Invariants Are Hard: On the Hardness of Strongest Polynomial Invariants for (Probabilistic) Programs. Proc. ACM Program. Lang., (POPL), 2024.

[ABK+24] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. (Un)Solvable Loop Analysis. Formal Methods Syst. Des., 2024. To appear.

Excluded publications. This thesis also influenced additional papers, though their content is not incorporated in this thesis. The author of this thesis served as one of the main contributors in all listed publications.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution Estimation for Probabilistic Loops. In Proc. of QEST, 2022.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-Based Invariants for Probabilistic Loops With Non-Polynomial Assignments. In Proc. of QEST, 2022.

[KMS+24] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovič, Ezio Bartocci, and Efstathia Bura. Exact and Approximate Moment Derivation for Probabilistic Loops With Non-Polynomial Assignments. ACM Trans. Model. Comput. Simul., 2024.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Awards & Honors. The research leading to the results presented in this thesis garnered recognition in various forms:

- The publication [KMS+22b] received the *Best Paper Award 2022* by the *International Conference on Quantitative Evaluation of SysTems*.
- The work on invariant synthesis for unsolvable loops [ABK+22] was honored with the *Radhia Cousot Best Young Researcher Paper Award*, awarded by the program committee of the *Static Analysis Symposium*.
- Three conference papers [MBKK21b, ABK+22, KMS+22b] were invited for publication as extended articles in journals [MBKK22, ABK+24, KMS+24].
- The Christina Hörbiger Prize from TU Wien was awarded in recognition of the efforts and contributions towards this thesis.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Computing Moments

This chapter is based on the following publication [MSBK22a]:

Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This Is the Moment for Probabilistic Loops. Proc. ACM Program. Lang., (OOPSLA2), 2022.

2.1 Problem Statement

The random nature of probabilistic programs (PPs) makes their functional analysis very challenging as one needs to reason about probability distributions of random variables instead of computing with single variable values [BKS20a]. A standard approach towards handling probability distributions associated with random variables is to estimate such distributions by sampling PPs using Monte Carlo simulation techniques [Has70]. While such approaches work well for statistical model checking [YS06], they are not suitable for the analysis of PPs with potentially infinite program loops as simulating infinite-state behavior is not always viable. Moreover, even for PPs with finitely many states, simulation-based analysis is inherently approximative.

With the aim of precisely, and not just approximately, handling random variables, probabilistic model checking [KNP11, DJKV17] became a prominent approach in the analysis of PPs with finite state spaces. For analyzing unbounded PPs, these techniques would however require non-trivial user guidance, in terms of assertion templates and/or invariants.

We address the challenge of precisely analyzing, and even recovering, probability distributions induced by PPs with both countably and uncountably infinite state spaces. We do so by extending both expressivity and automation of the state-of-the-art in PP analysis: We (i) focus on PPs with probabilistic infinite loops (see Figure 2.1) and (ii) fully automate

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

the analysis of such loops by computing exact higher-order statistical moments of program variables x parameterized by a loop counter n.

Functional representations $f(n)$ for a program variable x, with $f(n)$ characterizing the kth moment $\mathbb{E}(x_n^k)$ of x at iteration n, can be interpreted as a quantitative invariant $\mathbb{E}(x_n^k) - f(n) = 0$, as the equation is true for all loop iterations $n \in \mathbb{N}$. Inferring quantitative invariants is arguably not novel. On the contrary, it is one of the most challenging aspects of PP analysis, dating back to the seminal works of [MM05, KMMM10] introducing the weakest pre-expectations calculus. Template-based approaches to discover invariants or (super-)martingales emerged [BEFH16, KUH19] by translating the invariant generation problem into a constraint solving one. The derived quantitative invariants are generally provided in terms of expected values [CS14, KMMM10, MM05]. Nevertheless, the expected value alone — also referred to as the *first moment* — provides only partial information about the underlying probability distribution. This motivates the critical importance of higher moments for PP analysis [KUH19, BKS20c, WHR21, SBK22].

Higher Moments for PP Analysis. Using concentration-of-measure inequalities [BLM13], we can utilize higher moments $\mathbb{E}(X^k)$ to obtain upper and lower bounds on tail probabilities $\mathbb{P}(X > t)$, measuring the probability that a given random variable X, corresponding for example to our program variables x from Figure 2.1, surpasses some value t. *We also show that when a program variable x admits only $k < \infty$ many values, we can fully recover its probability mass function as a closed-form expression in the loop counter n using the first $k-1$ raw moments* (see Section 2.6). Furthermore, raw moments can be used to compute central moments $\mathbb{E}((X - \mathbb{E}(X))^k)$ and thus provide insights on other important characteristics of the distribution such as the *variance*, *skewness* and *kurtosis* [Dur19]. However, *computing exact higher statistical moments* for PPs is computationally expensive [KKM19], a challenge which we also address, as illustrated in Figures 2.1–2.2 and described next.

Computing Higher Moments. The theory we establish describes how to compute higher moments of program variables for a large class of probabilistic loops and how to utilize these moments to gain more insights into the analyzed programs. We call this theory the *theory of moment-computable probabilistic loops* (Section 2.5). Our approach is fully automatic, meaning it does *not* rely on externally provided invariants or templates. Unlike constraint solving over templates [BEFH16, KUH19], we employ algebraic techniques based on systems of *linear recurrences* with constant coefficients describing so-called *C-finite sequences* [KP11]. Different equivalence preserving program transformations (Section 2.3) and power reduction of finite valued variables (Section 2.4) allow us to simplify PPs and represent their higher moments as linear recurrence systems in the loop counter. Figure 2.1 shows a PP with many unique features supported by our technique towards PP analysis: it has an uncountable state-space, contains if-statements, symbolic constants, draws from continuous probability distributions with state-dependent parameters, and employs polynomial arithmetic as well as circular variable dependencies. *We are not aware of other works automating the reasoning about*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
toggle,sum,x,y,z = 0,s0,1,1,1
while ⋆:
  toggle = 1−toggle
  if toggle == 0:
    x = x+1 {1/2} x+2
    y = y+z+x**2 {1/3} y−z−x
    z = z+y {1/4} z−y
  end
  l = Laplace(x+y, 1)
  g = Normal(0,1)
  if g < 1/2: sum = sum+x end
end
```

$$\mathbb{E}(\text{toggle}_n) = \frac{1}{2} - \frac{(-1)^n}{2}$$

$$\mathbb{E}(x_n) = \frac{5}{8} + \frac{3n}{4} + \frac{3(-1)^n}{8}$$

$$\mathbb{E}(x_n^2) = \frac{15}{32} + \frac{17n}{16} + \frac{9n(-1)^n}{16} + \frac{17(-1)^n}{32} + \frac{9n^2}{16}$$

$$\mathbb{E}(l_n) = \frac{-17}{8} - \frac{15n}{4} + \frac{67 \cdot 2^{-n} \cdot 6^{\frac{n}{2}}}{10} + \frac{67 \cdot 2^{-n} 6^{\frac{1+n}{2}}}{30} - \frac{37 \cdot 3^{-n} 6^{\frac{n}{2}}}{10} - \frac{37 \cdot 3^{-n} 6^{\frac{1+n}{2}}}{20} + \frac{67 \cdot 6^{\frac{n}{2}} (-1)^n}{10 \cdot 2^n} - \frac{15(-1)^n}{8} - \frac{67 \cdot 6^{\frac{1+n}{2}} (-1)^n}{30 \cdot 2^n} + \frac{37 \cdot 6^{\frac{1+n}{2}} (-1)^n}{20 \cdot 3^n} - \frac{37 \cdot 6^{\frac{n}{2}} (-1)^n}{10 \cdot 3^n}$$

Figure 2.1: An example of a multi-path PP loop, with Laplace and Normal distributions parametrized by program variables. Our work fully automates the analysis of such and similar PP loops by computing higher moments. Several moments for program variables in the loop counter n are listed on the right. Each moment was automatically generated.

such and similar probabilistic loops, in particular for computing precise higher moments of variables. Figure 2.1 lists some of the variables' moments computed automatically by our technique. Further, these moments can be used to compute tail probability bounds or central moments, such as the variance, to characterize the distribution of the program variables as the loop progresses.

Thanks to our power reduction techniques (Section 2.4), our approach supports arbitrary polynomial dependencies among finite valued variables. Moreover, our technique can fully recover the value distributions of finite valued program variables, from finitely many higher moments. We illustrate this in Section 2.6 for Herman's self-stabilization protocol depicted in Figure 2.2, a randomized algorithm for recovering faults in a process token ring [Her90].

Theory and Practice in Computing Higher Moments. In theory, our approach can compute any higher moment for any variable and PP of our program model, under assumptions stated in Sections 2.3 and 2.5. We also establish the necessity of these assumtions in Section 2.5.3. In a nutshell, the completeness theorem (Theorem 6) holds for probabilistic loops for which non-finite program variables are not polynomially self-dependent and all branching conditions are over finite valued variables. We strengthen the theory of [BKS19] to support if-statements, circular variable dependencies, state-dependent distribution parameters, simultaneous assignments, and multiple assignments, and establish the necessity of our assumptions. Moreover, unlike [WHR21], our approach does not rely on templates and provides exact closed-form representations of higher moments parameterized by the loop counter.

In practice, our approach is implemented in the Polar tool and compared against exact as well as approximate methods [MSBK22b]. Our experiments (Section 2.7) show

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
x1, x2, x3 = 1, 1, 1
t1, t2, t3 = 1, 1, 1
p = 1/2; tokens = t1 + t2 + t3
while ⋆:
  x1o, x2o, x3o = x1, x2, x3
  if x1o == x3o: x1=Bernoulli(p) else: x1=x3o end
  if x2o == x1o: x2=Bernoulli(p) else: x2=x1o end
  if x3o == x2o: x3=Bernoulli(p) else: x3=x2o end

  if x1 == x3: t1 = 1 else: t1 = 0
  if x2 == x1: t2 = 1 else: t2 = 0
  if x3 == x2: t3 = 1 else: t3 = 0
  tokens = t1 + t2 + t3
end
```

$\mathbb{E}(tokens_n) = 1 + 2 \cdot 4^{-n}$ | $\mathbb{E}(tokens_n^2) = 1 + 8 \cdot 4^{-n}$ | $\mathbb{E}(tokens_n^3) = 1 + 26 \cdot 4^{-n}$

Figure 2.2: Herman's self stabilization algorithm with three nodes encoded as a probabilistic loop together with three moments of *tokens*.

that Polar outperforms the state-of-the-art of moment computation for probabilistic loops in terms of supported programs and efficiency. Furthermore, Polar is able to compute exact higher moments magnitudes faster than sampling can establish reasonable confidence intervals.

Contributions. Our main contributions are listed below:

- An automated approach for computing higher moments of program variables for a large class of probabilistic loops with potentially uncountable state spaces (Sections 2.3-2.5).
- We develop power reduction techniques to reduce the degrees of finite valued program variables in polynomials (Section 2.4).
- We prove completeness of our approach for computing higher moments (Section 2.5).
- We fully recover the distributions of finite valued program variables and approximate distributions for unbounded/continuous program variables from finitely many moments (Section 2.6).
- We provide an implementation and empirical evaluation of our approach, outperforming the state-of-the-art in PP analysis in terms of automation and expressivity (Section 2.7).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Preliminaries

We use the symbol $\mathbb{P}$ for probability measures and $\mathbb{E}$ for the expectation operator. The support of a random variable X is denoted by $\text{supp}(X)$.

2.2.1 Probability Theory

Operationally, a probabilistic program is a Markov chain with potentially uncountably many states. Let us recall some notions about Markov chains. For more details on Markov chains and probability theory in general we refer the reader to [Dur19].

For a fixed set S, a σ-algebra is a non-empty set of subsets of S closed under complementation and countable unions.

Definition 1 (Sequence Space)**.** *Let $(S, \mathcal{S})$ be a measurable space, that is, S is a set with a σ-algebra $\mathcal{S}$. Its* sequence space *is the measurable space $(S^\omega, \mathcal{S}^\omega)$ where $S^\omega := \{(s_1, s_2, \dots) : s_i \in S\}$ and $\mathcal{S}^\omega$ is the σ-algebra generated by the* cylinder sets *$Cyl[B_1, \dots, B_n] := \{\theta : \theta_i \in B_i, 1 \le i \le n\}$ for all prefixes $B_1, \dots, B_n \in \mathcal{S}$ and all $n \in \mathbb{N}$.*

A *Markov kernel* is, on a high level, a generalization of transition probabilities between states to uncountable state spaces and is required for the definition of a *Markov chain*.

Definition 2 (Markov Chain)**.** *Let $(S, \mathcal{S}, \mathbb{P})$ be a probability space and $p : S \times \mathcal{S} \to [0, 1]$ a Markov kernel. A stochastic process X_n is a* Markov chain *with Markov kernel p if*

$$\mathbb{P}(X_{n+1} \in B \mid X_0 = x_0, X_1 = x_1, \dots, X_n = x_n) = p(X_n, B). \tag{2.1}$$

Given a measurable space $(S, \mathcal{S})$, an *initial distribution* μ, a stochastic process X_n and a Markov kernel p, *Kolmogorov's Extension Theorem* says that there is a unique measure $\mathbb{P}$ such that X_n is a Markov chain in $(S^\omega, \mathcal{S}^\omega, \mathbb{P})$.

For a random variable X, central moments $\mathbb{E}((X - \mathbb{E}(X))^k)$ can be computed from raw moments $\mathbb{E}(X^k)$ and vice versa through the transformation of center:

$$\mathbb{E}\Big((X - b)^k\Big) = \mathbb{E}\Big(((X - a) + (a - b))^k\Big) = \sum_{i=0}^{k} \binom{k}{i} \mathbb{E}\Big((X - a)^i\Big)(a - b)^{k-i}. \tag{2.2}$$

2.2.2 Linear Recurrences

We briefly recall standard terminology on algebraic sequences and recurrences. For further details, we refer the reader to [KP11]. A sequence $(a_n)_{n=0}^\infty$ is called *C-finite* if it obeys a linear recurrence with constant coefficients, that is, $(a_n)_{n=0}^\infty$ satisfies an equation of the form

$$a_{n+l} = c_{l-1} \cdot a_{n-l-1} + c_{l-2} \cdot a_{n-l-2} + \dots + c_0 \cdot a_n,$$

for some *order* $l \in \mathbb{N}$, some constants $c_i \in \mathbb{R}$ and all $n \in \mathbb{N}$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

lop ∈ {*and*, *or*}, *cop* ∈ {=, ≠, <, >, ≥, ≤}, *Dist* ∈ {*Bernoulli*, *Normal*, *Uniform*, …}

⟨*sym*⟩ ::= `a` | `b` | … ⟨*var*⟩ ::= `x` | `y` | …

⟨*const*⟩ ::= $r \in \mathbb{R}$ | ⟨*sym*⟩ | ⟨*const*⟩ (`+` | `*` | `/`) ⟨*const*⟩

⟨*poly*⟩ ::= ⟨*const*⟩ | ⟨*var*⟩ | ⟨*poly*⟩ (`+` | `-` | `*`) ⟨*poly*⟩ | ⟨*poly*⟩`**n`

⟨*assign*⟩ ::= ⟨*var*⟩ `=` ⟨*assign_right*⟩ | ⟨*var*⟩ `,` ⟨*assign*⟩ `,` ⟨*assign_right*⟩

⟨*categorical*⟩ ::= ⟨*poly*⟩ (`{`⟨*const*⟩`}` ⟨*poly*⟩)* [`{`⟨*const*⟩`}`]

⟨*assign_right*⟩ ::= ⟨*categorical*⟩ | Dist`(`⟨*poly*⟩*`)` | `Exponential(`⟨*const*⟩`/`⟨*poly*⟩`)`

⟨*bexpr*⟩ ::= `true` (⋆) | `false` | ⟨*poly*⟩ ⟨*cop*⟩ ⟨*poly*⟩ | `not` ⟨*bexpr*⟩ | ⟨*bexpr*⟩ ⟨*lop*⟩ ⟨*bexpr*⟩

⟨*ifstmt*⟩ ::= `if` ⟨*bexpr*⟩`:` ⟨*statems*⟩ (`else if` ⟨*bexpr*⟩`:` ⟨*statems*⟩)* [`else:` ⟨*statems*⟩] `end`

⟨*statem*⟩ ::= ⟨*assign*⟩ | ⟨*ifstmt*⟩ ⟨*statems*⟩ ::= ⟨*statem*⟩$^+$

⟨*loop*⟩ ::= ⟨*statem*⟩* `while` ⟨*bexpr*⟩ `:` ⟨*statems*⟩ `end`

Figure 2.3: Grammar describing the syntax of probabilistic loops ⟨*loop*⟩.

Theorem 1 (Closed-form [KP11])**.** *Every C-finite sequence* $(a_n)_{n=0}^{\infty}$ *can be written as an* exponential polynomial*, that is* $a_n = \sum_{i=1}^{m} n^{d_i} u_i^n$ *for some natural numbers* $d_i \in \mathbb{N}$ *and complex numbers* $u_i \in \mathbb{C}$*. We refer to* $\sum_{i=1}^{m} n^{d_i} u_i^n$ *as the* closed-form *or the* solution *of the sequence* $(a_n)_{n=0}^{\infty}$ *or its recurrence.*

An important fact is that closed-forms of linear recurrences with constant coefficients of *any order* always exist and are computable. This also holds for all variables in *systems* of linear recurrences with constant coefficients.

2.3 Probabilistic Program Model

In this section we introduce our programming model (Section 2.3.1) and describe its semantics in terms of Markov chains (Section 2.3.2). Moreover, we introduce transformations (2.3.3) normalizing a probabilistic program to simplify its analysis.

2.3.1 Probabilistic Program Syntax

The syntax defining our program model is given by the grammar in Figure 2.3. Throughout the chapter, we will use the phrases *(probabilistic) loops* and *(probabilistic) programs* interchangeably for loops adhering to the syntax in Figure 2.3. We infer higher moments $\mathbb{E}(x_n^k)$ of program variables x parameterized by the loop counter n. We abstract from concrete loop guards by defining the guards of programs in our program model to be *true* (written as ⋆). Guarded loops **`while`** ϕ: … can be modeled as an infinite loops **`while`** ⋆: **`if`** ϕ: …, with the limit behaviour giving the moments after termination (cf. Section 2.5.1).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Our program model defined in Figure 2.3 contains non-nested while-loops which are preceded by a loop-free initialization part. The loop-body and initialization part allow for (nested) if-statements, polynomial arithmetic, drawing from common probability distributions, and symbolic constants. Symbolic constants can be used to represent arbitrary real numbers and are also used for uninitialized program variables. Categorical expressions (defined by the non-terminal $\langle categorical \rangle$ in Figure 2.3) are expressions of the form $v_1\{p_1\} \dots v_l\{p_l\}$ such that $\sum p_i = 1$. Their intended meaning is that they evaluate to v_i with probability p_i. The last parameter p_l can be omitted and in that case is set to $p_l := 1 - \sum_{i=1}^{l-1} p_i$. For a program $\mathcal{P}$ we denote with $\mathrm{Vars}(\mathcal{P})$ the set of $\mathcal{P}$'s variables appearing on the left-hand side of an assignment in $\mathcal{P}$'s loop-body. The programs of Figures 2.1-2.2 are examples of our program model defined in Figure 2.3. In comparison to the *probabilistic Guarded Command Language (pGCL)* [BKS20a], programs of our model contain exactly one while-loop, no non-determinism [1] but support continuous distributions and simultaneous assignments.

2.3.2 Program Semantics

In what follows we define the semantics of probabilistic programs in terms of Markov chains on a measurable space. We then introduce the notion of *normalized* probabilistic loops by means of so-called *$\mathcal{P}$-preserving program transformations* (Section 2.3.3).

Definition 3 (State & Run Space)**.** *Let $\mathcal{P}$ be a probabilistic program with m variables. We denote by* $\mathrm{ND}(\mathcal{P})$ the non-probabilistic program obtained from $\mathcal{P}$ *by replacing every probabilistic choice C in $\mathcal{P}$ by a non-deterministic choice over supp(C). Let $States_{\mathcal{P}} \subseteq \mathbb{R}^m$ be the set of program states of ND($\mathcal{P}$) reachable from any initial state. The* state space *of $\mathcal{P}$ is the measurable space $(States_{\mathcal{P}}, \mathcal{S}_{\mathcal{P}})$, where $\mathcal{S}_{\mathcal{P}}$ is the Borel σ-algebra on $\mathbb{R}^m$ restricted to $States_{\mathcal{P}}$. The* run space *of $\mathcal{P}$ is the sequence space $(States_{\mathcal{P}}^{\omega}, \mathcal{S}_{\mathcal{P}}^{\omega}) =: (Runs_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}})$.*

In what follows, we omit the subscript $\mathcal{P}$ whenever the program $\mathcal{P}$ is irrelevant or clear from the context. Executions/runs of a probabilistic program $\mathcal{P}$ define a stochastic process, as follows.

Definition 4 (Run Process)**.** *Let $\mathcal{P}$ be a probabilistic program with m variables. The* run process $\Phi_n : Runs \to States$ *is a stochastic process in the run space mapping a program run to its nth state, that means, $\Phi_n(run) := run_n$.*

For program variable x with index $i \geq 1$, we denote by x_n the projection of Φ_n to its ith component $\Phi_n(\cdot)(i)$*. Given an arithmetic expression A over $\mathcal{P}$'s variables, we write A_n for the stochastic process where every program variable x in A is replaced by x_n.*

Remark 1. *Given an initial distribution of program states μ and a Markov kernel p defined according to the* standard meaning *of the program statements, by* Kolmogorov's

[1] Non-determinism is different from probabilistic choice. Demonic (angelic) non-determinism is concerned with the worst-case (best-case) behavior. For instance, a variable can be assigned to 0 or 1 both with probability 1/2. This is different from assigning 0 or 1 non-deterministically, where the probability is not specified.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Extension Theorem *we conclude that there is a unique probability measure* $\mathbb{P}_\mathcal{P}$ *on* $(Runs_\mathcal{P}, \mathcal{R}_\mathcal{P})$ *such that the run process is a Markov chain.* $(Runs_\mathcal{P}, \mathcal{R}_\mathcal{P}, \mathbb{P}_\mathcal{P})$ *is the probability space associated to program* $\mathcal{P}$*. Distributions and (higher) moments of* $\mathcal{P}$*'s variables are to be understood with respect to this probability space.*

For probabilistic loops according to the syntax in Figure 2.3, the initial distribution μ of values of loop variables is the distribution of states after the statements $\langle statem \rangle^*$ just before the while-loop. Moreover, the loop body in Figure 2.3 is considered to be atomic, meaning the Markov kernel p describes the transition between full iterations in contrast to single statements.

2.3.3 $\mathcal{P}$-Preserving Transformations

Our probabilistic programs defined by the grammar in Figure 2.3 support rich arithmetic and complex probabilistic behavior/distributions. Such an expressivity of Figure 2.3 comes at the cost of turning the analysis of programs defined by Figure 2.3 cumbersome. In this section, we address this difficulty and introduce a number of program transformations that allow us to simplify our probabilistic programs to a so-called *normal form* while preserving the joint distribution of program variables. Normal forms allow us to extract recursive properties from the program, which we will later use to compute moments for program variables (Section 2.5).

Schemas and Unification. The program transformations we introduce in this section build on the notion of *schemas* and program parts. A *program part* is an empty word or any word resulting from any non-terminal of the grammar in Figure 2.3. For our purposes, a schema S is a program part with some subtrees in the program part's syntax tree being replaced by placeholder symbols $\dot{s}_1, \dots, \dot{s}_l$. A *substitution* is a finite mapping $\sigma = \{\dot{s}_1 \mapsto p_1, \dots, \dot{s}_l \mapsto p_l\}$ where $p_1, \dots, p_l$ are program parts. We denote by $S[\sigma]$ the program part resulting from S by replacing every $\dot{s}_i$ by p_i, assuming $S[\sigma]$ is well-formed. For two schemas S_1 and S_2 a substitution u such that $S_1[u] = S_2[u]$ is called a *unifier* (with respect to S_1 and S_2). In this case S_1 and S_2 are called *unifiable* (by u).

Transformations. In what follows, we consider $\mathcal{P}$ to be a fixed probabilistic program defined by Figure 2.3 and give all definitions relative to $\mathcal{P}$. A *transformation* T is a mapping from program parts to program parts with respect to a schema *Old*. T is *applicable* to a subprogram S of $\mathcal{P}$ if S and *Old* are unifiable by the unifier u. Then, the transformed subprogram is defined as $T(S) := New[u]$ where *New* is a schema depending on *Old* and u. A transformation is fully specified by defining how *New* results from *Old* and u. We write $T(\mathcal{P}, S)$ for the program resulting from $\mathcal{P}$ by replacing the subprogram S of $\mathcal{P}$ by $T(S)$.

The first transformation we consider removes simultaneous assignments from $\mathcal{P}$. For this, we store a copy of each assignment in an auxiliary variable to preserve the values

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

used for simultaneous assignments, in case an assigned variable appears in an assignment expression. Variables are then assigned their intended value.

Definition 5 (Simultaneous Assignment Transformation)**.** *A* simultaneous assignment transformation *is the transformation defined by*

$\dot{x}_1, \ldots, \dot{x}_l$ **=** $\dot{v}_1, \ldots, \dot{v}_l \;\mapsto\; t_1$**=**$\dot{v}_1; \ldots; t_l$**=**$\dot{v}_l; \dot{x}_1$**=**$t_1; \ldots; \dot{x}_l$**=**$t_l$,
where $t_1, \ldots, t_l$ are fresh variables.

In what follows, we assume that parameters of common distributions used in programs are constant. Nevertheless, the following transformation enables the use of some non-constant distribution parameters.

Definition 6 (Distribution Transformation)**.** *A* distribution transformation *is a transformation defined by either of the mappings*

- $\dot{x}$ **=** Normal($\dot{p}$,$\dot{v}$) $\mapsto$ t **=** Normal(0,$\dot{v}$); $\dot{x}$**=**$\dot{p}$+t
- $\dot{x}$ **=** Uniform($\dot{p}_1$,$\dot{p}_2$) $\mapsto$ t **=** Uniform(0,1); $\dot{x}$**=**$\dot{p}_1$+($\dot{p}_2$−$\dot{p}_1$)∗t
- $\dot{x}$ **=** Laplace($\dot{p}$,$\dot{b}$) $\mapsto$ t **=** Laplace(0,$\dot{b}$); $\dot{x}$**=**$\dot{p}$+t
- $\dot{x}$ **=** Exponential($\dot{c}/\dot{p}$) $\mapsto$ t **=** Exponential($\dot{c}$); $\dot{x}$**=**$\dot{p}$∗t

where, for every mapping, t is a fresh variable.

Example 1. *In Figure 2.1,* l,g **=** Laplace(x+y,1),Normal(0,1) *is a simultaneous assignment and can be transformed using the transformation rules from Definitions 5-6 as follows:*

$\overset{(sim)}{\mapsto}$
```
t1=Laplace(x+y,1)
t2=Normal(0,1)
x=t1
y=t2
```
$\overset{(dist)}{\mapsto}$
```
t3=Laplace(0,1)
t1=x+y+t3
t2=Normal(0,1)
x=t1
y=t2
```

To simplify the structure of probabilistic loops, we assume **else if** branches to be syntactic sugar for nested **if else** statements. We remove **else** by splitting it into if-statements (**if** C and **if not** C). Since variables in C could be changed within the first branch, we store their original values in auxiliary variables and use those for the condition C' of the second **if** statement. We capture this transformation in the following definition.

Definition 7 (Else Transformation)**.** *An* else transformation *is the transformation*

if $\dot{C}$: $\overset{\ldots}{Branch_1}$
else: $\overset{\ldots}{Branch_2}$ **end** , u $\mapsto$ t_1**=**$x_1; \ldots; t_l$**=**x_l
if $\dot{C}$: $\overset{\ldots}{Branch_1}$ **end**
if not C': $\overset{\ldots}{Branch_2}$ **end**

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

where $x_1, \ldots, x_l$ are all variables appearing in $\dot{C}[u]$ which are also being assigned in $\dddot{Branch}_1[u]$. Every t_i is a fresh variable and C' results from $\dot{C}[u]$ by substituting every x_i with t_i.

To further simplify the loop body into a flattened list of assignments, we equip every assignment a of form "x = value"with a condition C_a (initialized to true $\top$) and a default variable d_a (initialized to x), written as "x = value $[C_a]$ d_a". The semantics of the conditioned assignment is that x is assigned *value* if C_a holds just before the assignment and d_a otherwise. With conditioned assignments, the loop body's structure can be flattened using the following transformation.

Definition 8 (If Transformation)**.** *An* if transformation *is the transformation defined by*

$$\begin{array}{l} \texttt{if } \dot{C}_1: \\ \dot{x} \texttt{ = } \dot{v} \ [\dot{C}_2] \ \dot{x} \\ \dddot{Rest} \texttt{ end} \end{array} , u \mapsto \begin{array}{l} \texttt{t = } \dot{x} \\ \dot{x} \texttt{ = } \dot{v} \ [\dot{C}_1 \texttt{ and } \dot{C}_2] \ \dot{x} \\ \texttt{if } C: \ \dddot{Rest} \texttt{ end} \end{array}$$

where t is a fresh variable and C results from $\dot{C}_1[u]$ by substituting $\dot{x}[u]$ by t. If $\dddot{Rest}[u]$ is empty, the line **if** *C: $\dddot{Rest}$* **end** *is omitted from the result. If $\dot{x}[u]$ does not appear in $\dot{C}_1[u]$ the line* t = $\dot{x}$ *is dropped.*

Example 2. *The following program containing nested if-statements can be flattened as follows:*

```
if x == 1:
x = Bernoulli(1/2)
if x == 0: y = 1 end
end
```

$\overset{(if)}{\mapsto}$

```
if x == 1:
x = Bernoulli(1/2)
y = 1 [x == 0] y
end
```

$\overset{(if)}{\mapsto}$

```
t = x
x = Bernoulli(1/2) [t == 1] x
if t == 1:
y = 1 [x == 0] y
end
```

$\overset{(if)}{\mapsto}$

```
t = x
x = Bernoulli(1/2) [t == 1] x
y = 1 [t == 1 ∧ x == 0] y
```

To bring further simplicity to our program $\mathcal{P}$, we ensure for each variable to be modified only once within the loop body. To remove duplicate assignments we introduce new variables $x_1, \ldots, x_{l-1}$ to store intermediate states. Assignments to other variables, in between the updates of x, will be adjusted to refer to the latest x_i instead of x.

Definition 9 (Multi-Assignment Transformation)**.** *A* multi-assignment transformation *is the transformation defined by*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{array}{l} \dot{x} = \dot{v}_1 \ [\dot{C}_1] \ \dot{x} ; \overset{\dots}{Rest}_1 ; \\ \dot{x} = \dot{v}_2 \ [\dot{C}_2] \ \dot{x} ; \overset{\dots}{Rest}_2 ; \\ \dots ; \ \dot{x} = \dot{v}_l \ [\dot{C}_l] \ \dot{x} ; \end{array} , u \quad \mapsto \quad \begin{array}{l} x_1 = \dot{v}_1 \ [\dot{C}_1] \ \dot{x} ; Rest_1 ; \\ x_2 = v_2 \ [C_2] \ x_1 ; Rest_2 ; \\ \dots ; \ \dot{x} = v_l \ [C_l] \ x_{l-1} ; \end{array}$$

where $x_1, \dots, x_{l-1}$ are fresh variables. For $i \geq 2$, v_i, C_i and $Rest_i$ result from $\dot{v}_i[u]$, $\dot{C}_i[u]$ and $\overset{\dots}{Rest}_i[u]$, respectively, by replacing $\dot{x}[u]$ by x_{i-1}.

Example 3. *In the program of Figure 2.2, program line* **if** x1 **==** x3: t1 **=** 1 **else:** t1 **=** 0 *can be transformed using transformation rules from Definitions 7-9 as follows:*

```
(else)    if x1 == x3: t1 = 1
  ↦       if x1 != x3: t1 = 0

(if)      t1 = 1 [x1 == x3] t1
  ↦       t1 = 0 [x1 != x3] t1

(multi)   t11 = 1 [x1 == x3] t1
  ↦       t1 = 0 [x1 != x3] t11
```

With program transformations defined, we can turn our attention to program properties. In particular, we show that our transformations of $\mathcal{P}$ do not change the joint distribution of $\mathcal{P}$'s variables. Since our transformations may introduce new variables, we consider program equivalence with respect to program variables in order to ensure that the distribution of $\mathcal{P}$ is maintained/preserved by our transformations.

Definition 10 (Program Equivalence)**.** *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two probabilistic programs. We define $\mathcal{P}_1$ and $\mathcal{P}_2$ to be* equivalent with respect to a set of program variables X*, in symbols $\mathcal{P}_1 \equiv^X \mathcal{P}_2$, if:*

1. *$X \subseteq Vars(\mathcal{P}_1) \cap Vars(\mathcal{P}_2)$, and*
2. *the joint distributions of X arising from $\mathcal{P}_1$ and $\mathcal{P}_2$ are equal.*

To relate a program $\mathcal{P}$ to its transformed version $T(\mathcal{P}, S)$ we consider the distribution of variables of the original program $\mathcal{P}$. If $\mathcal{P}$ retains the joint distribution of its variables after applying transformation T, we say that T is $\mathcal{P}$-preserving.

Definition 11 ($\mathcal{P}$-Preserving Transformation)**.** *We say that a transformation T is* $\mathcal{P}$-preserving *if $\mathcal{P} \equiv^{Vars(\mathcal{P})} T(\mathcal{P}, S)$ for all subprograms S of $\mathcal{P}$ which are unifiable with Old.*

It is not hard to argue that the transformations defined above are $\mathcal{P}$-preserving, yielding the following result.

Lemma 2. *The transformations from Definitions 5-9 are $\mathcal{P}$-preserving.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

By exhaustively applying the $\mathcal{P}$-preserving transformations of Definitions 5-9 over $\mathcal{P}$, we obtain a so-called normalized program $\mathcal{P}_N$, as defined below. The normalized $\mathcal{P}_N$ will then further be used in computing higher moments of $\mathcal{P}$ in Sections 2.5, as the $\mathcal{P}_N$ preserves the moments of $\mathcal{P}$ (Theorem 3).

Definition 12 (Normal Form)**.** *A program $\mathcal{P}$ is in* normal form *or a* normalized *program if none of the transformations from Definitions 5-9 are applicable to $\mathcal{P}$.*

Theorem 3 (Normal Form)**.** *For every probabilistic program $\mathcal{P}$ there is a $\mathcal{P}_\mathcal{N}$ in normal form such that $\mathcal{P} \equiv^{Vars(\mathcal{P})} \mathcal{P}_\mathcal{N}$. Moreover, $\mathcal{P}_\mathcal{N}$ can be effectively computed from $\mathcal{P}$ by exhaustively applying transformations from Definitions 5-9.*

Proof. There are two claims in the theorem, which we need to address: (i) exhaustively applying transformation rules terminates (*termination*), and (ii) it preserves statistical properties of the (original) program variables (*correctness*).

For termination, we show that programs become smaller, in some sense, after every transformation. In particular, we consider the program size to be given by a tuple (*Sim*, *Dist*, *Else*, *If*, $Multi_B$, $Multi_I$), representing the number of simultaneous assignments, non-trivial distributions, else statements, assignments within if branches (weighted for nested ifs), and number of variables with multiple assignments in the loop body and initialization part, respectively. With respect to the lexicographic order, each transformation reduces the size of the program which is lower-bounded by 0.

Correctness can be shown by treating each transformation separately and showing that it does not alter the variables' distributions after a single application (Lemma 2). This is true for all transformations from Definitions 5-9. Auxiliary variables are used to store the original value to prevent intervening variable modifications. For Multi-Assignment Transformation (Definition 9), we also revise the rest of the assignments to reflect the change of the original variable. The Distribution Transformation (Definition 6) uses statistical properties of well-known distributions. □

Properties of Normalized Programs. Figure 2.4 shows a normal form for the program from Figure 2.1. Normalized programs have the following important properties: (1) all distribution parameters are constant; (2) the loop body is a sequence of guarded assignments; (3) every program variable is only assigned once in the loop body. Moreover for every guarded assignment v = $assign_{true}$ [C] $assign_{false}$, the guard C is a boolean condition and $assign_{false}$ is a single variable which is assigned to v if C evaluates to *false*. If C evaluates to *true*, the variable v is assigned $assign_{true}$. The expression $assign_{true}$ is either a distribution with constant parameters or a probabilistic choice of polynomials as illustrated in Figure 2.4.

Remark 2. *Based on the order in which transformations are applied to a program $\mathcal{P}$ and the names used for auxiliary variables, several different normalized programs can be achieved for $\mathcal{P}$. For our approach, only the existence of a normal form is relevant.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
toggle = 0; sum = s0
x = 1; y = 1; z = 1
while ⋆:
  toggle = 1−toggle
  x = 1+x {1/2} 2+x [toggle==0] x
  y = y+z+x**2 {1/3} −x+y−z [toggle==0] y
  z = z+y {1/4} z−y [toggle==0] z
  t1 = Laplace(0, 1)
  l = t1+x+y
  g = Normal(0, 1)
  sum = sum+x [g < 1/2] sum
end
```

Figure 2.4: A normal form for the program in Figure 2.1.

Moreover, from the definitions of the transformation, it is apparent that exhaustively applying them leads to a normal form whose size is linear in the size of the original program.

2.4 Finite Types in Probabilistic Programs

Given a probabilistic program $\mathcal{P}$ in our programming model, the transformations of Section 2.3.3 simplify $\mathcal{P}$ by computing its normalized form while maintaining the distribution (and hence also moments) of $\mathcal{P}$. Nevertheless, the normalized form of $\mathcal{P}$ contains computationally expensive polynomial arithmetic, potentially hindering the automated analysis of $\mathcal{P}$ in Section 2.5 due to a computational blowup. Therefore, we introduce further simplifications for $\mathcal{P}$ by means of power reduction techniques.

Example 4. *Consider Figure 2.1 and assume we are interested in the* k*th power of variable toggle and deriving the raw moment* $\mathbb{E}(toggle_n^k)$*. Our analysis relies on replacing variables with their assignments (see Section 2.5), leading to the expression* $(1-toggle_{n-1})^k$*. When expanded, this is a polynomial in* $toggle_{n-1}$ *with* $k+1$ *monomials:*

$$(1 - toggle_{n-1})^k = \sum_{i=0}^{k} \binom{k}{i} (-1)^i toggle_{n-1}^i$$

Higher moments, together with the aforementioned replacements, may lead to blowups of the number of monomials to consider. However, observing that the variable toggle is binary, we have $toggle_n^k = toggle_n = 1 - toggle_{n-1}$ *for any* $k \geq 0$*. Arbitrary powers of the finite variable toggle with* 2 *possible values can be written in terms of powers smaller than* 2*. In the rest of this section, we show that this phenomenon generalizes from binary variables to arbitrary finite valued variables, thus simplifying the analysis of higher moments of finite valued program variables.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As defined in Definition 4, for an arithmetic expression X over program variables, X_n denotes the stochastic process mapping a program run to the value of X after iteration n.

Definition 13 (Finite Expression)**.** *Let $\mathcal{P}$ be a probabilistic program and X an arithmetic expression over the variables of $\mathcal{P}$. We say that X is* finite *if there exist $a_1, \ldots, a_m \in \mathbb{R}$ such that for all $n \in \mathbb{N}$: $X_n \in \{a_1, \ldots, a_m\}$.*

2.4.1 Power Reduction for Finite Types

As established in [BKS20b, Lemma 1], high powers k of a random variable X over a finite set can be reduced. We adapt their result to our setting as follows.

Theorem 4 (Finite Power Reduction)**.** *Let $m, k \in \mathbb{Z}$ and X be a discrete random variable over $A = \{a_1, \ldots, a_m\}$. Then we can rewrite X^k as a linear combination of $1, X, X^2, \cdots, X^{m-1}$. Furthermore,*

$$X^k = \overline{a^k} M^{-1} \overline{X}, \tag{2.3}$$

where $\overline{a^k} = (a_1^k, \ldots, a_m^k)$, M is an $m \times m$ matrix with $M_{ij} = a_j^{i-1}$ (with $0^0 := 1$), and $\overline{X} = (X^0, \ldots, X^{m-1})^T$.

In other words, Theorem 4 implies that any higher moment of X, can be computed from just its first $m-1$ moments. Furthermore, we build on Theorem 4 and establish the inverse of matrix M explicitly (M is explicit in Theorem 4 but its inverse is implicit).

Theorem 5 (Reduction Formula)**.** *Recall that the kth elementary symmetric polynomial with respect to a set $V = \{v_1, \ldots, v_n\}$ is $e_k(V) = \sum_{1 \leq j_1 < \cdots < j_k \leq n} v_{j_1} \cdots v_{j_k}$ and let $A_{-j} = A \setminus \{a_j\}$). Then the inverse of M in (2.3) is given by*

$$M_{ij}^{-1} = -\frac{(-1)^j e_{m-j}(A_{-i})}{\prod_{a \in A_{-i}} (a - a_i)}. \tag{2.4}$$

Proof. Let $MN = B$ for M as of (2.3) and N as of (2.4). We show that $B = I$ by showing that $B_{ij} = 1$ if $i = j$ and $B_{ij} = 0$ otherwise. We have

$$\begin{aligned} B_{ij} = \sum_{1 \leq k \leq m} N_{ik} M_{kj} = \sum_{1 \leq k \leq m} -\frac{(-1)^k e_{m-k}(A_{-i})}{\prod_{a \in A_{-i}} (a - a_i)} a_j^{k-1} \\ = \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \sum_{1 \leq k \leq m} -(-1)^k a_j^{k-1} e_{m-k}(A_{-i}) \\ = \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \sum_{0 \leq k \leq m-1} (-a_j)^k e_{m-k-1}(A_{-i}) \\ = \frac{1}{\prod_{a \in A_{-i}} (a - a_i)} \prod_{a \in A_{-i}} (a - a_j), \end{aligned}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

where the last equation comes from the expansion of product $\prod_{a \in A_{-i}}(a - a_j)$ and grouping by the exponent of a_j. We can clearly see that the last expression is 1 if $i = j$ and 0 otherwise. □

Example 5. *Let X be a random variable over $A := \{-2, 0, 1, 3\}$. Using Theorem 4-5, we obtain the 10th power of X as:*

$$X^{10} = \begin{pmatrix}(-2)^{10} & 0^{10} & 1^{10} & 3^{10}\end{pmatrix} \begin{pmatrix} 0 & -1/10 & 2/5 & -1/30 \\ 1 & -5/6 & -1/3 & 1/6 \\ 0 & 1 & 1/6 & -1/6 \\ 0 & -1/15 & 1/30 & 1/30 \end{pmatrix} \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \end{pmatrix}$$
$$= 1934X^3 + 2105X^2 - 4038X.$$

2.5 Computing Higher Moments of Probabilistic Programs

We now bring together the results from Sections 2.3-2.4 to develop the *theory of moment-computability* for probabilistic loops. We establish the technical details leading to sufficient conditions that ensure moment-computability, culminating in the proof of Theorem 6. The main ideas of our method are illustrated on the probabilistic loop from Figure 2.1 in Example 6 at the end of this section.

Definition 14 (Moment-Computability)**.** *A probabilistic loop $\mathcal{P}$ is* moment-computable *if a closed-form (according to Theorem 1) of $\mathbb{E}(x_n^k)$ exists and is computable for all $x \in Vars(\mathcal{P})$ and $k \in \mathbb{N}$.*

We will describe the class of moment-computable probabilistic loops through the properties of the dependencies between program variables.

Definition 15 (Variable Dependency)**.** *Let $\mathcal{P}$ be a probabilistic loop and $x, y \in Vars(\mathcal{P})$. We define:*

- y depends conditionally on x, *if there is an assignment of y within an if-else-statement and x appears in the if-condition.*
- y depends finitely on x, *if x is finite and appears in an assignment of y.*
- y depends linearly on x, *if x appears only linearly in every assignment of y.*
- y depends polynomially on x, *if there is an assignment of y in which x appears non-linearly and x is not finite (motivated by Section 2.4.1).*
- y depends on x *if it depends on x conditionally, finitely, linearly, or polynomially.*

Furthermore, we consider the transitive closure for variable dependency as follows: If z depends on y and y depends on x, then z depends on x. If one of the two dependencies is polynomial, then z depends polynomially on x.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A crucial point to highlight in Definition 15 is that due to transitivity, variables can depend on themselves. For instance, if variable x depends on y and y on x, then x is self-dependent. Moreover, if either of the dependencies between x and y is non-linear, x depends, by Definition 15, *polynomially on itself.* The absence of such polynomial self-dependencies is a central condition for our notion of moment-computable loops.

Theorem 6 (Moment-Computability)**.** *A probabilistic loop $\mathcal{P}$ is moment-computable if* (1) none of its non-finite variables depends on itself polynomially, and (2) if the variables in all if-conditions are finite.

Note that none of the program transformations from Section 2.3.3 can introduce a polynomial (self-)dependence. We capture this in the following lemma:

Lemma 7 (Non-Dependency Preservation)**.** *If a variable $x \in Vars(\mathcal{P})$ does not depend on itself polynomially, neither does $x \in Vars(\mathcal{P}_\mathcal{N})$.*

Before we prove Theorem 6, let us first show its validity for programs in normal form (as defined in Section 2.3.3). Recall that a normalized program's loop body is a flat list of (guarded) assignments, one for every (possibly auxiliary) program variable.

Lemma 8 (Normal Moment-Computability)**.** *The Moment-Computability Theorem (Theorem 6) holds for loops in normal form.*

Proof. We have to show that for an arbitrary normalized program $\mathcal{P}$ satisfying the conditions of Theorem 6, all $x \in \text{Vars}(\mathcal{P})$ and all $k \in \mathbb{N}$, the kth moment of x (that is $\mathbb{E}(x_n^k)$) admits a closed-form as an exponential polynomial. $\mathbb{E}(x_n^k)$ admits a closed-form as an exponential polynomial in n if it satisfies a linear recurrence. We show a slightly more general statement. That is, we show that for *any monomial of program variables* M (and hence also for x^k) $\mathbb{E}(M)$ satisfies a linear recurrence. The main idea of the proof is to show that $\mathbb{E}(M)$ only depends on a finite set of monomials, each (in some sense) *not larger* than M itself. Intuitively, the finite set of monomials on which $\mathbb{E}(M)$ depends on are all monomials of program variables such that their expected values determine $\mathbb{E}(M)$. We will show that this set of monomials exists, is finite, and leads to a system of linear recurrences containing $\mathbb{E}(M)$, implying a computable exponential polynomial closed-form for $\mathbb{E}(M)$ by Theorem 1.

Let $\mathcal{P}$ be a normalized program satisfying the conditions of Theorem 6, $x \in \text{Vars}(\mathcal{P})$, $k, n \in \mathbb{N}$ arbitrary, and $\mathcal{M}$ the set of all monomials over $\text{Vars}(\mathcal{P})$ with the powers of every finite variable d bounded by the number of possible values of d (higher powers can be reduced as of Theorem 4).

Recurrences over Moments. Given the syntax of probabilistic programs and properties of expectation, for any monomial $M \in \mathcal{M}$ there is a natural way to express the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

expected value of M in iteration $n+1$, that is $\mathbb{E}(M_{n+1})$, as a linear combination of expectations of monomials in iteration n:

$$\mathbb{E}(M_{n+1}) = \sum_{N \in M^*} c_N \mathbb{E}(N_n), \tag{2.5}$$

for some finite set $M^* \subset \mathcal{M}$, and non-zero constants c_N. Equation (2.5) is called *the recurrence* of $\mathbb{E}(M)$. The set M^* is the set of monomials that appear in the recurrence of $\mathbb{E}(M)$. We define the * operator to give such a set for any monomial and extend the definition to sets by

$$S^* = \bigcup_{M \in S} M^*.$$

The exact recurrence can be computed from $\mathbb{E}(M_{n+1})$ by replacing variables appearing in M by their assignments and using the linearity of $\mathbb{E}$ to convert expected values of polynomials to linear combinations of expected monomials. Recall that for a program in normal form, every program variable is only assigned once, all distribution parameters are constant, and the loop body is a flat list of guarded assignments (Section 2.3.3). The guarded assignments are of the form

$$x = a_0\{p_0\}\dots\{p_{i-1}\}a_i[C_x]d_x,$$

for polynomials of program variables $a_0, \dots, a_i$ and constant probabilities $p_0, \dots, p_{i-1}$, or

$$x = Dist[C_x]d_x$$

for some admissible distribution $Dist$. The guard $[C_x]$ is a boolean condition and for normalized programs, d_x is always a program variable. Assume, that the variable x appears in the monomial M. Hence, $M = M' \cdot x_{n+1}^k$ for some monomial M' not containing x. If the single assignment of x is a (guarded) probabilistic choice of polynomials we rewrite $\mathbb{E}(M_{n+1})$ to

$$\mathbb{E}(M_{n+1}) = \mathbb{E}(M' \cdot x_{n+1}^k) = \mathbb{E}\Big(M'\Big(d_x[\neg C_x] + \sum p_i a_i^k [C_x]\Big)\Big). \tag{2.6}$$

If the single assignment of x is a (guarded) draw from a distribution we rewrite $\mathbb{E}(M_{n+1})$ to

$$\mathbb{E}(M_{n+1}) = \mathbb{E}(M' \cdot x_{n+1}^k) = \mathbb{E}(M' d_x[\neg C_x]) + \mathbb{E}(M'[C_x])\mathbb{E}\Big(Dist^k\Big). \tag{2.7}$$

Variables in conditions $[C_x]$ are all finite since they come from branch conditions. We further simplify the expressions of Equations (2.6)-(2.7) by replacing the logical conditions $[C_x]$ by polynomials that evaluate to 1 whenever variables satisfy the condition $[C_x]$ and to 0 otherwise. It is possible to write any logical condition over finitely valued variables as such a polynomial ([SBK22]), with $[x = c] := \prod_{d \in \omega(x) \setminus \{c\}} \frac{x-d}{c-d}$, $[\neg C] = 1 - [C]$, and

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

$[C_1 \wedge C_2] = [C_1] \cdot [C_2]$, where $\omega(x)$ is the set of possible values of x[2]. Converting conditions to polynomials in the equation above leads to polynomials over moments of program variables. We can compute the recurrence of $\mathbb{E}(M)$ in Equation (2.5) by replacing all variables in M of iteration $n+1$ from last to first (by their appearance in $\mathcal{P}$'s loop body). Throughout, the linearity of $\mathbb{E}$ is used to convert expected values of polynomials to linear combinations of expected monomials. In Example 6, we illustrate this computation on the program from Figure 2.1 with its normal form from Figure 2.4.

Ordering. Now that we can compute the recurrences, we need to introduce the order for monomials, such that the monomials appearing in the recurrence for M are not larger than M. We will need this to show that computing the recurrences as described above is, indeed, a finite process. Intuitively, a variable y is larger than (or equal to) x if it depends on x. Mutually dependent variables will form an equivalence class. We then extend the order to monomials based on their degrees with respect to the variables' equivalence classes.

More formally, let $\preceq$ be a smallest total preorder on variables such that $x \preceq y$ whenever y depends on x. We write $x \prec y$ iff $x \preceq y$ and $y \npreceq x$, and $x \sim y$ iff $x \preceq y$ and $y \preceq x$. Let $\widetilde{x}$ be the equivalence class of x induced by $\sim$. Note, that all variables in an equivalence class are mutually dependent. However, because of our assumptions on the program $\mathcal{P}$, the mutual dependencies among non-finite variables are all linear (as there are no polynomial self-dependencies).

We extend $\preceq$ to a preorder on the set of monomials $\mathcal{M}$. For every monomial M and non-finite variable x, we consider the degree $deg(\widetilde{x}, M)$ of M in the equivalence class of $\widetilde{x}$[3]. We associate M with the sequence of $deg(\widetilde{x}, M)$ for equivalence classes of all non-finite variables, ordered reverse-lexicographically with respect to $\preceq$. Then the relations $\prec$, $\sim$, and equivalence classes $\widetilde{(-)}$ follow naturally from $\preceq$.

By Theorem 4 and the definition of $\preceq$, the equivalence class $\widetilde{M}$ is finite for each $M \in \mathcal{M}$. Let $\mathcal{M}_\sim$ be the set of equivalence classes of $\sim$. The preorder $\preceq$ induces a partial order $\preceq_\sim$ on $\mathcal{M}_\sim$. Note that monomials only contain non-negative powers and a finite number of variables. These facts together with $\preceq$ being total imply that $\preceq_\sim$ is a well-order. We will write $\preceq$ instead of $\preceq_\sim$ when the meaning is clear from the context.

With these orders defined, we have $N \preceq M$ for any $N \in M^*$ and $M \in \mathcal{M}$. Intuitively, this means that for every monomial M, the monomials occurring in the recurrence of M are not larger than M itself. This is true because the order on variables is defined according to variable dependencies, the order on monomials is a reverse-lexicographic extension of the order on variables, and the fact that polynomial self-dependencies are not allowed by assumption.

[2]Because negation and conjunction are functionally complete for propositional logic, we can also handle disjunctions using De Morgan's laws: $[P \vee Q] = [\neg(\neg P \wedge \neg Q)] = 1-(1-[P]) \cdot (1-[Q])$. Inequalities can then be transformed into a disjunction of equalities since we assume that all variables appearing in if-conditions are finitely valued.

[3]$deg(\widetilde{x}, M) := \sum_{x \in \widetilde{x}} deg(x, M)$, where $deg(x, M)$ is the degree of x in M.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Finite $\mathcal{S}^{(Q)}$ We show, that for any monomial Q, there is a finite set of monomials $\mathcal{S}^{(Q)} \subset \mathcal{M}$ containing Q such that $M^* \subset \mathcal{S}^{(Q)}$ for any $M \in \mathcal{S}^{(Q)}$. Because M^* is the set of all monomials in the recurrence of M, this means that $\mathcal{S}^{(Q)}$ contains all monomials necessary to construct a system of linear recurrences containing $E(Q)$ (if it exists and is finite).

Let

$$\mathcal{S}^{(Q)} := \widetilde{Q} \cup \bigcup_{\substack{A \in \widetilde{Q}^* \\ A \prec Q}} \mathcal{S}^{(A)}. \tag{2.8}$$

Clearly $Q \in \mathcal{S}^{(Q)}$ and $M^* \subset \mathcal{S}^{(Q)}$ for any $M \in \mathcal{S}^{(Q)}$ by construction. We are left to show that $\mathcal{S}^{(Q)}$ is finite for all $Q \in \widetilde{Q} \in \mathcal{M}_\sim$, which we can do by transfinite induction over $\mathcal{M}_\sim$.

For the base case, we have to show that $\mathcal{S}^{(Q)}$ is finite for all Q in $\widetilde{1}$ (the trivial monomial). Since $\widetilde{1} = \{\prod_{d \in F} d^{\lambda_d} \mid \lambda_d \leq m_d\}$, where F is the set of finite program variables and m_d are upper bounds on their powers as of Theorem 4, $\mathcal{S}^{(Q)} = \widetilde{1}$ is finite for all $Q \in \widetilde{1}$. Suppose $\mathcal{S}^{(A)}$ is finite for all $A \prec Q$. Since $\widetilde{Q}^*$ is finite, so is the union in (2.8) and, as a result, $\mathcal{S}^{(Q)}$.

Moments. A system of linear recurrences with constant coefficients can be constructed for monomials in $\mathcal{S}^{(Q)}$. Therefore, the closed-form of any $E(M) \in \mathcal{M}$ exists and is computable. □

We now turn back to Theorem 6 and establish its validity. The crux of our proof below relies on the fact that our transformations computing normal forms of $\mathcal{P}$ (see Section 2.3.3) are $\mathcal{P}$-preserving.

Proof (of Theorem 6). By Theorem 3 (Normal Form Termination), $\mathcal{P}$ can be transformed to a normalized loop $\mathcal{P}_\mathcal{N}$. By Lemma 7 (Non-Dependency Preservation), $\mathcal{P}_\mathcal{N}$ satisfies all conditions from Lemma 8 (Normal Moment-Computability). Thus, $\mathcal{P}_\mathcal{N}$ is moment-computable and the moments are equivalent to those of $\mathcal{P}$ for $x \in \text{Vars}(\mathcal{P})$ by Theorem 3 (Normal Form Correctness). □

The proofs of Theorem 6 and Lemma 8 are constructive and describe a procedure to compute (higher) moments of program variables by (1) transforming a probabilistic loop into a normal form according to Theorem 3, (2) constructing a system of linear recurrences as in the proof of Lemma 8 and (3) solving the system of linear recurrences with constant coefficients. In the following example, we illustrate the whole procedure on the running example from Figure 2.1.

Example 6. *We return to the probabilistic loop $\mathcal{P}$ from Figure 2.1. A normal form $\mathcal{P}_\mathcal{N}$ for $\mathcal{P}$ was given in Figure 2.4. To compute a closed-form of the expected value of the program variable z, we will model $\mathbb{E}(z_n)$ as a system of linear recurrences according to*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the proof of Lemma 8. For a cleaner presentation we will refer to the variable toggle *by* t*. Note that* t *is binary. To construct the recurrence for* $\mathbb{E}(z_n)$*, we start with* $\mathbb{E}(z_{n+1})$ *at the assignment of* z *in* $\mathcal{P}_\mathcal{N}$ *and repeatedly replace variables by the right-hand side of their assignment starting from* z*'s assignment and stopping at the top of the loop body. Throughout the process we use the linearity of expectation to convert expected values of polynomials to linear combinations of expected monomials (as required by Equation 2.5):*

$$\mathbb{E}(z_{n+1})$$

$$\downarrow \textit{assignment of } z$$

$$\mathbb{E}(([t_{n+1}=0](z_n+y_{n+1})\ \{1/4\}\ [t_{n+1}=0](z_n-y_{n+1}))+[t_{n+1}\neq 0]z_n)$$

$$\downarrow \textit{replace (in)equalities by polynomials}$$

$$\mathbb{E}(((1-t_{n+1})(z_n+y_{n+1})\ \{1/4\}\ (1-t_{n+1})(z_n-y_{n+1}))+t_{n+1}z_n)$$

$$\downarrow \mathbb{E} \textit{ on categorical}$$

$$\frac{1}{4}\mathbb{E}((1-t_{n+1})(z_n+y_{n+1}))+\frac{3}{4}\mathbb{E}((1-t_{n+1})(z_n-y_{n+1}))+\mathbb{E}(t_{n+1}z_n)$$

$$\downarrow \textit{simplify; } \mathbb{E} \textit{ linearity}$$

$$\mathbb{E}(z_n)+\frac{1}{2}\mathbb{E}(t_{n+1}y_{n+1})-\frac{1}{2}\mathbb{E}(y_{n+1})$$

$$\downarrow_* \textit{similarly, replace } y_{n+1}, x_{n+1}, t_{n+1}$$

$$\mathbb{E}(z_n)-\frac{1}{6}\mathbb{E}(t_nx_n)-\frac{1}{2}\mathbb{E}(t_ny_n)-\frac{1}{6}\mathbb{E}(t_nx_n^2)+\frac{1}{12}\mathbb{E}(t_n)+\frac{1}{6}\mathbb{E}(t_nz_n).$$

The last line of the calculation represents the recurrence equation of the expected value of z*. For every monomial in the recurrence equation of* $\mathbb{E}(z_n)$ *(that is:* $z^* = \{tx, ty, tx^2, t, tz\}$*), we compute its respective recurrence equation and recursively repeat this procedure which eventually terminates according to Lemma 8.*

The ordering of variables and monomials, which is essential for proving termination in Lemma 8 comes from an ordering of variables t, x, y, z, l, g, sum*. Based on the program assignments, we need* $t \preceq x \prec y \preceq l; y \preceq z; t \preceq z \preceq y; x \preceq sum; g \preceq sum$*. Variables* y *and* z *form an equivalence class, since* $y \preceq z$ *and* $z \preceq y$*. Notice that some variables are not ordered, for example* t *and* g*. This gives some freedom in choosing the total preorder, any will work. Let us have* $t \prec g \prec x \prec sum \prec y \sim z \prec l$*. This gives an order on the equivalence classes* $\{t\} \prec \{g\} \prec \{x\} \prec \{sum\} \prec \{y,z\} \prec \{l\}$*. The ordering on monomials then considers the classes of non-finite program variables, i.e. all equivalence classes except* $\{t\}$*. A monomial is then assigned a sequence of degrees with respect to each equivalence class, e.g.* $z \to (0,0,0,1,0)$*,* $tz \to (0,0,0,1,0)$*, and* $x^2 \to (0,0,2,0,0)$*,*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

with monomials ordered reverse-lexicographically with respect to their sequence. For the monomials in this example: $1 \sim t \prec x \sim tx \prec x^2 \sim tx^2 \prec y \sim z \sim ty \sim tz$.

After computing all necessary recurrence equations, we are faced with a system of linear recurrences of the expected values of the monomials z, tx, ty, x, y, tx^2, t, tz, x^2 *and* 1 *with recurrence matrix*

$$\begin{bmatrix} 1 & -\frac{1}{6} & -\frac{1}{2} & 0 & 0 & -\frac{1}{6} & \frac{1}{12} & \frac{1}{6} & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{3}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & 1 & \frac{1}{3} & -\frac{1}{6} & -\frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & \frac{5}{2} & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Using any computer algebra system, such as `sympy`[4]*, we arrive at the closed-form solution for the expected value of* z, $\mathbb{E}(z_n)$*, parameterized by the loop counter* n*:*

$$\begin{aligned} \mathbb{E}(z_n) = \frac{883}{32} + \frac{29n}{16} - \frac{201}{20}2^{-n}6^{\frac{n}{2}} - \frac{67}{20}2^{-n}6^{\frac{1+n}{2}} - \frac{37}{10}3^{-n}6^{\frac{n}{2}} \\ - \frac{37}{20}3^{-n}6^{\frac{1+n}{2}} - \frac{201}{20}6^{\frac{n}{2}}\left(\frac{-1}{2}\right)^n - \frac{37}{10}6^{\frac{n}{2}}\left(\frac{-1}{3}\right)^n + \frac{67}{20}6^{\frac{1+n}{2}}\left(\frac{-1}{2}\right)^n \\ + \frac{37}{20}6^{\frac{1+n}{2}}\left(\frac{-1}{3}\right)^n + \frac{9}{16}n(-1)^n + \frac{29}{32}(-1)^n + \frac{9}{16}n^2. \end{aligned}$$

This example highlights the strength of algebraic techniques for probabilistic program analysis in comparison to constraint-based methods employing templates. We are not aware of any template-based method able to handle functions of the complexity of $\mathbb{E}(z_n)$*. Our tool* Polar *is able to find the closed-form of* $\mathbb{E}(z_n)$ *in under one second (see Section 2.7).*

2.5.1 Guarded Loops

When we model a guarded loop **while** ϕ: ... as an infinite loop **while** $\star$: **if** ϕ: ..., we impose the same restrictions on ϕ as on if-conditions (that means ϕ only contains finite variables) to guarantee computability and correctness. The kth moment of x after termination is then given by

$$\lim_{n\to\infty} \mathbb{E}(x_n^k \mid \neg\phi_n) = \lim_{n\to\infty} \frac{\mathbb{E}(x_n^k \cdot [\neg\phi_n])}{\mathbb{E}([\neg\phi_n])}. \tag{2.9}$$

[4] `https://www.sympy.org`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

If the limit exists, we can use standard methods from computer algebra to compute it, as the (higher) moments our approach computes are given as exponential polynomials [Gru96].

Example 7. *Consider the following loop, in which x after termination is geometrically distributed with parameter $1/2$:*

```
x, stop = 0, 0
while stop == 0:
    stop=Bernoulli(1/2)
    x=x+1
end
```

With Equation 2.9 and the techniques from this section we get:

$$\begin{aligned}\mathbb{E}(x) &= \lim_{n\to\infty} \mathbb{E}(x_n \mid stop_n = 1)\\ &= \lim_{n\to\infty} \frac{\mathbb{E}(x_n \cdot stop_n)}{\mathbb{E}(stop_n)}\\ &= \lim_{n\to\infty} \frac{-n2^{-n} - 2^{1-n} + 2}{1 - 2^{-n}} = 2.\end{aligned}$$

Moreover, whenever the variables in the loop guard are not probabilistic, traditional techniques can be applied to determine the number of loop iterations n which can then be plugged into the (higher) moments computed by our approach. Apart from the guarded loops, many systems show the type of infinite behavior naturally modeled with infinite loops, such as probabilistic protocols or dynamical systems.

2.5.2 Infinite If-Conditions

Theorem 6 on moment-computability requires the variables in all if-conditions to be finite. Nevertheless, in some cases, if-conditions containing infinite variables can be handled by our approach. Let $\mathcal{P}$ be a probabilistic loop containing an if-statement with condition F **and** I where F contains only finite variables and I contains infinite variables. Without loss of generality, no variable in I is assigned in or after the if-statement. Let the transformation removing I be defined by

```
                          t = Bernoulli(p)
if F and I: Branch   ↦    if F and t == 1:
                          Branch end
```

where t is a fresh variable and $p := \mathbb{P}(I)$ (potentially symbolic). Then, the transformation preserves the distributions of all $x \in \text{Vars}(\mathcal{P})$ under the following assumptions:

1. I is iteration independent, meaning for every variable x in I neither x nor any variable x depends on (as of Definition 15) has a self-dependency.

2. I is statistically independent from F and all conditions in *Branch*.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. For every assignment A in *Branch* and every variable x in A which has been assigned before A, it holds that I and x are statistically independent.

Assumption 1 ensures that $\mathbb{P}(I) = \mathbb{E}([I])$ is constant. Assumption 2 and 3 further ensure that $\mathbb{E}([I])$ can always be "pulled out" (that means $\mathbb{E}([I]x) = \mathbb{E}([I])\mathbb{E}(x)$) in the construction of the recurrences. Assumptions 1-3 can often be checked automatically.

Example 8. *Consider the statement* `if g < 1/2: sum=sum+x` *of the program from Figure 2.1, where the value of* g *is drawn from a standard normal distribution. In this case, the transformation's parameter* p *represents* $\mathbb{P}(Normal(0,1) < 1/2)$*, but is left symbolic for the moment computation. The integral* $\mathbb{P}(Normal(0,1) < 1/2)$ *can be solved separately and the result be substituted for* p*.*

2.5.3 On the Necessity of the Conditions Ensuring Moment-Computability

Theorem 6 states two conditions that are sufficient to ensure that the closed-forms of the program variables' higher moments always exist and are computable. *Condition 1 enforces that there is no variable with potentially infinite values with a polynomial self-dependency. Condition 2 demands that all variables appearing in if-conditions are finite.* Our approach for computing the moments of variables of probabilistic loops can handle precisely the programs that satisfy these two conditions. We argue that both conditions are necessary, in the sense that if either of the conditions does not hold, the existence or computability of the variable moments' closed-forms as exponential polynomials cannot be guaranteed for all programs from our program model when one or both conditions are removed from Theorem 6.

Condition 1. Relaxing condition 1 of Theorem 6 means that we allow for polynomial self-dependencies of non-finite variables. The *logistic map* [May76] is a quadratic first-order recurrence defined by $x_{n+1} = r \cdot x_n(1 - x_n)$ and well-known for its chaotic behavior. A famous fact about the logistic map is that it does not have an analytical solution for most values of r [Mar20]. By neglecting condition 1, we can easily devise a loop modeling the logistic map:

```
while ⋆:
    x = r·x(1−x)
end
```

The value of the program variable x after iteration n is equal to the nth term of the logistic map. This means, for most values of r and initial values of x, there does not exist an analytical closed-form solution for the program variable x. Moreover, our counterexample illustrates that condition 1 is necessary already for programs with a single variable and without stochasticity and if-statements.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Condition 2. Loosening condition 2 of Theorem 6 and allowing for non-finite variables in if-conditions renders our programming model Turing-complete. Intuitively, one can model a Turing machine's tape with two variables `l` and `r` such that the binary representation of `l` represents the tape's content left of the read-write-head. The binary representation of `r` represents the tape's content at the position of the read-write-head and towards the right. The least significant bit of `r` is the current symbol the Turing machine is reading. We can extract the least significant bit of `r` in our programming model (and neglecting condition 2) by introducing a variable `lsb` and using a single if-statement involving non-finite variables: whenever the loop changes the value of `r`, we set `lsb` := `r`. The while-loop's body is of the form "**if** lsb **>** 1: lsb=lsb−2 **else** *transitions* **end**". The Turing machine's transition table can be encoded using if-statements. Writing and shifting can be accommodated for by multiplying by 2 or 1/2 and using addition and subtraction. The Turing machine's state can be modelled by a single finite variable. Therefore, by dropping condition 2, being able to model the program variables by linear recurrences would give rise to a decision procedure for the Halting problem: assume we introduce a variable `terminated` which is initialized to 0 before the loop and set to 1 whenever the Turing-machine terminates. If `terminated` can be modelled by a linear recurrence of order k, it suffices to check the first k values of the recurrence to determine whether or not `terminated` is always 0 [KP11] and the Turing-machine does not terminate. As the Halting problem is well-known to be undecidable, condition 2 is necessary to guarantee that the program variables can be modelled by linear recurrences, even without stochasticity and polynomial arithmetic.

Remark 3 (Sequential & Nested Loops)**.** *Our program model consists of single non-nested loops. Sequential loops can be analyzed one by one with the same techniques as presented in this section. For nested loops, one could design a program transformation transforming a nested loop into a non-nested loop and then apply the techniques presented in this section. Alternatively, we conjecture that the approach presented for guarded loops (Section 2.5.1) could be used to first compute the moments of the most inner loops and then use the obtained information to compute the moments of the outer loops. The main challenge lies in ensuring the moment-computability conditions for the outer loops (Theorem 6) once the inner loops have been analyzed.*

2.6 Use-Cases of Higher Moments

For probabilistic loops, computing closed-forms of the variables' (higher) moments poses a technique for synthesizing quantitative invariants: Given a program variable x and a closed-form $f(n)$ of its kth moment, the equation $\mathbb{E}(x_n^k) - f(n) = 0$ is an invariant. Moreover, closed-forms of raw moments can be converted into closed-forms of *central* moments, such as variance, skewness or kurtosis (cf. Section 2.2.1). In addition, this section provides hints on two further use-cases of higher moments of probabilistic loops: (i) deriving tail probabilities (Section 2.6.1) and (ii) inferring distributions of random variables from their moments (Section 2.6.2).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

2.6.1 From Moments to Tail Probabilities

Tail probabilities measure the probability that a random variable surpasses some value. The mathematical literature contains several inequalities providing upper- or lower bounds on tail probabilities given (higher) moments [BLM13]. Two examples are *Markov's inequality* for upper and the *Paley-Zygmund inequality* for lower bounds.

Theorem 9 (Markov's Inequality)**.** *Let X be a non-negative random variable, and $t \geq 0$, then*

$$\mathbb{P}(X \geq t) \leq \frac{\mathbb{E}(X^k)}{t^k}.$$

Theorem 10 (Paley-Zygmund Inequality)**.** *Let X be a random variable with $X \geq t$ almost-surely. Then*

$$\mathbb{P}(X > t) \geq \frac{(\mathbb{E}(X) - t)^2}{\mathbb{E}(X^2) - 2t\mathbb{E}(X) + t^2}.$$

Example 9. *For* Herman's Self-Stabilization *program from Figure 2.2 almost-surely $tokens \in \{0, 1, 2, 3\}$. With the techniques from previous sections, we can compute the first two moments $\mathbb{E}(tokens_n) = 1 + 2 \cdot 4^{-n}$ and $\mathbb{E}(tokens_n^2) = 1 + 8 \cdot 4^{-n}$. Markov's inequality (Theorem 9) gives us the upper bound $\mathbb{P}(tokens_n \geq 2) \leq 1/2 + 4^{-n}$ using the first moment and $\mathbb{P}(tokens_n \geq 2) \leq 1/4 + 2 \cdot 4^{-n}$ utilizing the second moment.*

For the Paley-Zygmund inequality (Theorem 10) both the first and the second moment are required, yielding the lower bound $\mathbb{P}(tokens_n \geq 2) = \mathbb{P}(tokens_n > 1) \geq 4^{-n}$. The theorem's precondition that almost-surely $tokens \geq 1$ might not be apparent at first sight. We take this for granted for now and will clarify this fact in Example 10.

Markov's inequality and the Paley-Zygmund inequality are just two examples showing that our technique for moment computation can be leveraged for further program analysis using known results from probability theory. Our approach computes the *exact moments* of variables in probabilistic loops instead of just approximations or bounds on moments. This enables our technique to be readily combined with results from probability theory that require exact moments.

2.6.2 From Moments to Distributions

For finite random variables, their full distribution can be recovered from finitely many moments. More precisely, given a random variable X with m possible values, the distribution of X can be recovered from its first $m-1$ moments, as the following theorem states:

Theorem 11. *Let X be a random variable over $\{a_1, \ldots, a_m\}$ and $p_i := P(X = a_i)$. The values p_i are the solutions of the system of linear equations given by $\sum_{i=1}^{m} p_i a_i^j = \mathbb{E}(X^j)$ for $0 \leq j < m$.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

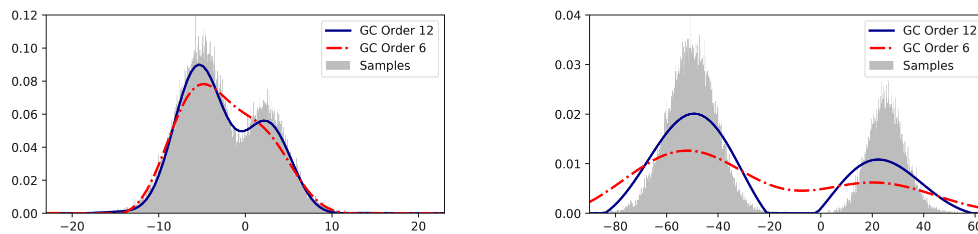


Figure 2.5: The empirical density of program variable x for the benchmark *Bimodal* (cf. Table 2.1) and loop iterations 10 (left) and 100 (right) obtained by 10^5 samples, together with two approximations using the Gram-Charlier A Series with 6 (red dashed lines) and 12 (blue solid lines) moments.

Proof. There are m unknowns p_i for $1 \leq p_i \leq m$. Note that all a_i are constant and that the first $m-1$ raw moments of X are fixed. Using the definition of raw moments, we get m linear equations $\sum_{i=1}^{m} p_i a_i^j = \mathbb{E}(X^j)$ for $0 \leq j < m$. The solutions of the system of m linear equations are the values p_i that determine the distribution of X. □

Example 10. *Consider* Herman's Self-Stabilization *program from Figure 2.2. In Example 9 we obtained upper and lower bounds for tail probabilities of the tokens variable using the first one or two moments. With the first* three *moments we can fully recover the distribution of the tokens variable. We have that tokens* $\in \{0, 1, 2, 3\}$*. Let* $p_i := \mathbb{P}(tokens_n = i)$ *for* $0 \leq i \leq 3$*. By Theorem 11, we get the following system:*

$$
\begin{aligned}
p_0 + p_1 + p_2 + p_3 &= \mathbb{E}(tokens_n^0) = 1, \\
p_1 + 2p_2 + 3p_3 &= \mathbb{E}(tokens_n) = 1 + 2 \cdot 4^{-n}, \\
p_1 + 4p_2 + 9p_3 &= \mathbb{E}(tokens_n^2) = 1 + 8 \cdot 4^{-n}, \\
p_1 + 8p_2 + 27p_3 &= \mathbb{E}(tokens_n^3) = 1 + 26 \cdot 4^{-n}.
\end{aligned}
$$

The solution can be obtained using standard techniques and tools, yielding $p_0 = 0; p_1 = 1 - 4^{-n}; p_2 = 0; p_3 = 4^{-n}$.

Note that probabilities are given as functions of the loop iteration n*. Moreover, the solution shows that almost-surely tokens* ≥ 1*, which we assumed to be true in Example 9.*

The distributions of program variables with potentially infinitely many values, including continuous variables, cannot be, in general, fully reconstructed from finitely many moments. However, expansions such as the *Gram–Charlier A Series* [Kol06] can be used to approximate a probability density function using finitely many moments. Figure 2.5 illustrates how exact moments computed by our approach can be used to approximate unknown probability density functions of program variables. While Figure 2.5 shows approximations for specific loop iterations, we emphasize that the symbolic nature of our approach allows for approximating the densities of program variables for all – potentially infinitely many – loop iterations simultaneously. Therefore, our technique is constant in

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the number of loop iterations, whereas sampling has linear complexity. We compute the approximations from Figure 2.5 for all infinitely many loop iterations in ∼22 seconds, with the experimental setup from Section 2.7. In comparison, sampling the loop 10^5 times takes ∼3.6 minutes for loop iteration 10 and ∼33 minutes for loop iteration 100.

2.7 Implementation and Evaluation

Implementation. The program transformations (Section 2.3) and (higher) moment computation (Section 2.5) are implemented in the new tool Polar [MSBK22b]. The experiments can be reproduced using the corresponding artifact[5]. For automatically inferring finiteness of program variables, we use a standard approach based on *abstract interpretation*. Polar is implemented in `python3`, consisting of ∼3300 LoC, and uses the packages `sympy`[6] and `symengine`[7] for symbolic manipulation of mathematical expressions. Together with all our benchmarks, Polar is publicly available at `https://github.com/probing-lab/polar`.

Experimental Setting and Evaluation. The evaluation of our method is split into three parts. First, we evaluate Polar on the ability of computing higher moments for 15 probabilistic programs exhibiting different characteristics (Section 2.7.1). Second, we compare Polar to the exact tool Mora [BKS20c] which computes so-called *moment-based invariants* for a subset of our programming model (Section 2.7.2). Third, we compare our tool to approximate methods estimating program variable moments by confidence intervals through sampling (Section 2.7.3). All experiments have been run on a machine with a 2.6 GHz Intel i7 (Gen 10) processor and 32 GB of RAM. Runtime measurements are averaged over 10 executions.

2.7.1 Experimental Results with Higher Moments

Table 2.1 shows the evaluation of Polar on the program from Figure 2.1 and 14 benchmarks which are either from the literature on probabilistic programming [KNP12] (*Herman-3*), [MM05] (*Duelling-Cowboys*), [GKM13] (*Martingale-Bet*), [CS14] (*Hawk-Dove-Symbolic*, *Variable-Swap*), [BEFH16] (*Gambler-Ruin-Momentum*), [BCK+21] (*Retransmission-Protocol*), differential privacy schemes [War65] (*Randomized-Response*), Dynamic Bayesian Networks (*DBN-Umbrella*, *DBN-Component-Health*) or well-known stochastic processes (*Las-Vegas-Search*, *Pi-Approximation*, *Bimodal*). The benchmarks *Retransmission-Protocol* and *Hawk-Dove-Symbolic* were further generalized from their original definition by replacing concrete numbers with symbolic constants. This makes these benchmarks only harder as solutions to the generalized versions are solutions for the concretizations. Table 2.1 illustrates that Polar can compute higher moments for various probabilistic programs exhibiting different features, like circular variable dependencies,

[5] `https://doi.org/10.5281/zenodo.7055030`
[6] `https://www.sympy.org`
[7] `https://github.com/symengine`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 2.1: Evaluation of Polar on 15 benchmarks. All times are in seconds. #V = number of variables in benchmark; C = benchmark contains circular dependencies; If = contains if-statements; S = contains symbolic constants; INF = state space is infinite; CONT = state space is continuous; Moment = Moment to compute; RT = Total runtime.

Benchmark	#V	C/If/S/INF/CONT	Moment	RT
Running-Example (Fig. 2.1)	7	✓/ ✓/ ✓/ ✓/ ✓	$\mathbb{E}(z)$	0.67
Herman-3	10	✓/ ✓/ ✗/ ✗/ ✗	$\mathbb{E}(\text{tokens}^3)$	0.58
Las-Vegas-Search	3	✗/ ✓/ ✗/ ✓/ ✗	$\mathbb{E}(\text{found}^{20})$	0.36
Pi-Approximation	4	✗/ ✓/ ✗/ ✓/ ✓	$\mathbb{E}(\text{count}^3)$	0.47
50-Coin-Flips	101	✗/ ✓/ ✗/ ✗/ ✗	$\mathbb{E}(\text{total})$	0.91
Gambler-Ruin-Momentum	4	✓/ ✗/ ✓/ ✓/ ✗	$\mathbb{E}(x^3)$	2.89
Hawk-Dove-Symbolic	5	✗/ ✓/ ✓/ ✓/ ✗	$\mathbb{E}(\text{p1bal}^4)$	2.00
Variable-Swap	4	✓/ ✗/ ✗/ ✓/ ✓	$\mathbb{E}(x^{30})$	2.42
Retransmission-Protocol	4	✗/ ✓/ ✓/ ✓/ ✗	$\mathbb{E}(\text{fail}^3)$	1.62
Randomized-Response	7	✗/ ✓/ ✓/ ✓/ ✗	$\mathbb{E}(\text{p1}^3)$	0.59
Duelling-Cowboys	4	✓/ ✓/ ✓/ ✗/ ✗	$\mathbb{E}(\text{ahit})$	1.14
Martingale-Bet	4	✗/ ✓/ ✓/ ✓/ ✗	$\mathbb{E}(\text{capital}^3)$	8.44
Bimodal	5	✗/ ✓/ ✗/ ✓/ ✓	$\mathbb{E}(\text{x}^{10})$	4.50
DBN-Umbrella	2	✗/ ✓/ ✓/ ✗/ ✗	$\mathbb{E}(\text{umbrella}^5)$	0.77
DBN-Component-Health	3	✗/ ✓/ ✗/ ✗/ ✗	$\mathbb{E}(\text{obs}^5)$	0.26

if-statements, and symbolic constants with finite, infinite, continuous, and discrete state spaces. Moreover, the table shows that the number of program variables is *not* the primary factor for the complexity of computing moments. For instance, the benchmarks *50-Coin-Flips* and *Duelling-Cowboys* have 101 and 4 program variables respectively. Nevertheless, the runtimes for computing first moments for the two benchmarks only differ by 0.23 s. The complexity of computing moments lies in the complexity of the resulting systems of recurrences which depend on the concrete features present in the benchmarks like specific variable dependencies, symbolic constants, or degrees of polynomials.

2.7.2 Experimental Comparison to Exact Methods

To the best of our knowledge, Mora is the only other tool capable of computing higher moments for variables of probabilistic loops without templates – as described in [BKS19]. Mora operates on so-called *Prob-solvable loops* which form a strict subset of our program model (Section 2.3). Prob-solvable loops do not admit circular variable dependencies, if-statements, or state-dependent distribution parameters. We compare Polar against Mora on the Mora benchmarks taken from [KUH19, CHWZ15, CS14, KMMM10]. Details can be found in Table 2.2. The experiments illustrate that Polar can handle all

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 2.2: Comparison of Polar to Mora. The runtimes are in seconds per tool, benchmark, and moment. For Polar the comparison contains in brackets the seconds spent on parsing, normalizing, and type inference.

Benchmark	Moment	Mora	Polar
COUPON	$\mathbb{E}(c)$	0.25	0.29
	$\mathbb{E}(c^2)$	0.27	0.29 (0.07)
	$\mathbb{E}(c^3)$	0.29	0.29
COUPON4	$\mathbb{E}(c)$	0.71	0.36
	$\mathbb{E}(c^2)$	0.87	0.36 (0.08)
	$\mathbb{E}(c^3)$	1.22	0.36
RANDOM_WALK_1D	$\mathbb{E}(x)$	0.07	0.12
	$\mathbb{E}(x^2)$	0.11	0.23 (0.07)
	$\mathbb{E}(x^3)$	0.10	0.24
SUM_RND_SERIES	$\mathbb{E}(x)$	0.27	0.27
	$\mathbb{E}(x^2)$	0.97	0.43 (0.07)
	$\mathbb{E}(x^3)$	2.48	0.79
PRODUCT_DEP_VAR	$\mathbb{E}(p)$	0.37	0.28
	$\mathbb{E}(p^2)$	1.41	0.46 (0.08)
	$\mathbb{E}(p^3)$	4.03	0.97
RANDOM_WALK_2D	$\mathbb{E}(x)$	0.10	0.12
	$\mathbb{E}(x^2)$	0.21	0.24 (0.08)
	$\mathbb{E}(x^3)$	0.17	0.24
BINOMIAL(p)	$\mathbb{E}(x)$	0.12	0.25
	$\mathbb{E}(x^2)$	0.32	0.29 (0.07)
	$\mathbb{E}(x^3)$	0.79	0.44
STUTTERING_A	$\mathbb{E}(s)$	0.29	0.26
	$\mathbb{E}(s^2)$	1.13	0.43 (0.07)
	$\mathbb{E}(s^3)$	3.32	0.97
STUTTERING_B	$\mathbb{E}(s)$	0.26	0.27
	$\mathbb{E}(s^2)$	0.94	0.39 (0.07)
	$\mathbb{E}(s^3)$	2.26	0.79
STUTTERING_C	$\mathbb{E}(s)$	0.76	0.40
	$\mathbb{E}(s^2)$	12.43	1.94 (0.08)
	$\mathbb{E}(s^3)$	74.83	8.19
STUTTERING_D	$\mathbb{E}(s)$	0.76	0.43
	$\mathbb{E}(s^2)$	8.19	1.34 (0.07)
	$\mathbb{E}(s^3)$	25.67	4.33
STUTTERING_P	$\mathbb{E}(s)$	0.26	0.28
	$\mathbb{E}(s^2)$	1.17	0.49 (0.07)
	$\mathbb{E}(s^3)$	3.53	1.17
SQUARE	$\mathbb{E}(y)$	0.30	0.29
	$\mathbb{E}(y^2)$	0.88	0.45 (0.07)
	$\mathbb{E}(y^3)$	1.98	0.65

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 2.3: Comparison of Polar to approximation through sampling. Polar = the tools runtime to compute the precise moment; CI N = an approximated 0.95-CI-interval from N samples; $T_{100.000}$ = the runtime for CI 100.000. The symbolic constant p in *Retransmission-Protocol* is set to 0.9 and for *Hawk-Dove-Symbolic* we set $v = 4$ and $c = 8$.

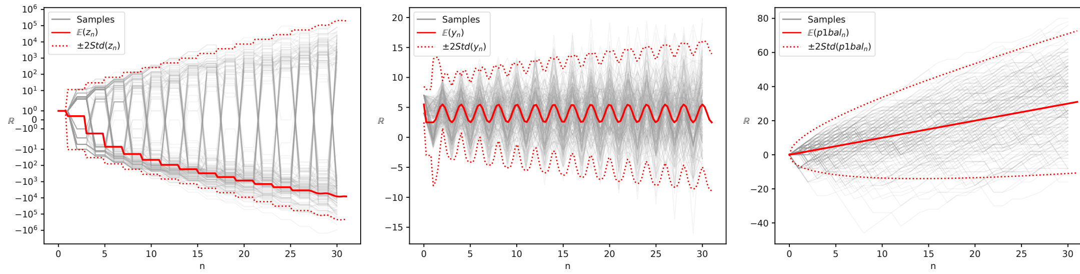
Benchmark	Moment	CI 100 CI 1.000 CI 100.000	$T_{100.000}$	Polar
Running-Example (Fig. 2.1)	$\mathbb{E}(z_{10})$	$(-55.6, 6.55)$ $(-58.5, -39.3)$ $(-45.6, -43.7)$	545.6s	0.67s
Retransmission-Protocol	$\mathbb{E}(\text{fail}_{10})$	$(0.09, 0.25)$ $(0.11, 0.16)$ $(0.109, 0.114)$	146.2s	0.30s
Variable-Swap	$\mathbb{E}(y_{10})$	$(4.47, 5.75)$ $(5.06, 5.45)$ $(5.50, 5.54)$	245.8s	0.13s
Hawk-Dove-Symbolic	$\mathbb{E}(p1bal_{10})$	$(8.28, 12.3)$ $(9.07, 10.5)$ $(9.86, 10.0)$	347.4s	0.27s

programs and moments that Mora can. Mora, however, cannot compute any moment for any program in Table 2.1. On simple benchmarks, Polar is slightly slower than Mora due to the constant overhead of the program transformations and type inference to identify finite valued variables. On complex benchmarks Polar provides a significant speedup compared to Mora. For instance, for the *STUTTERING_C* benchmark Polar computes the moment $\mathbb{E}(s^3)$ in about 8 seconds, whereas Mora needs over one minute.

2.7.3 Experimental Comparison with Sampling

For a probabilistic loop with program variable x the moment $\mathbb{E}(x_n^k)$ can be approximated for fixed k and n by sampling x_n^k and calculating the sample average or confidence intervals. Table 2.3 compares Polar to computing confidence intervals by sampling for $k = 1$ and $n = 10$. The table shows that our tool is able to compute precise moments in a fraction of the time needed to sample programs to achieve satisfactory confidence intervals. An advantage of sampling is that it is applicable for any probabilistic loop. However, by its nature, sampling fails to give any formal guarantees or hard bounds on the approximated moments. This is critical if the loop body contains branches that are executed with low probability. If applicable, Polar can provide *exact* moments for symbolic n (and involving other symbolic constants) faster than sampling can establish acceptable approximations. Moreover, even if the sampling of the loops is sped up by using a more efficient implementation, Polar enjoys complexity theoretical advantages. The complexity of sampling is linear in both the number of samples and the number of loop iterations. In contrast, Polar does not need to take multiple samples for higher

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Running-Example (Fig. 2.1) (b) Variable-Swap (c) Hawk-Dove-Symbolic

Figure 2.6: Samples obtained by simulation plotted together with precise moments computed by Polar. In each benchmark, the thin gray lines are 200 samples over 30 iterations. The thick red line is the precise expected value. The dotted red lines are the expected values $\pm$ twice the standard deviation given by the precise first two moments. Figure 2.6a is symmetric log scale.

precision as it symbolically computes the exact moments. Additionally, our method is *constant* in the number of loop iterations. With Polar, computing the moment for a specific loop iteration, say 10^5, just amounts to evaluate the closed-form at 10^5.

Figure 2.6 illustrates the importance of higher moments for probabilistic loops. The first moment provides a center of mass but contains no information on how the mass is distributed around this center. For this purpose higher moments are essential.

2.7.4 Evaluation Summary

Our experimental evaluation demonstrates that: (1) Polar can compute higher moments for a rich class of probabilistic loops with various characteristics, (2) Polar outperforms the state-of-the-art of moment computation for probabilistic loops in terms of supported programs and efficiency, and (3) Polar computes exact moments magnitudes faster than sampling can establish reasonable approximations.

2.8 Related Work

Using recurrence equations to extract closed-forms for variables and quantitative invariants of loops is a well-studied technique for non-probabilistic programs [FK15, BCKR20, KBCR19, KCBR18, dOBP16, HJK17, HJK18b, Kov08, RcK04]. Because a classical program is a special case of a probabilistic program, our technique presented in Section 2.5 is a generalization of the closed-form computation for classical programs to probabilistic programs. Moreover, the generalization to probabilistic programs is not trivial. One reason for this is that for classical programs the closed form for x^p is just the closed-form for x to the power p. However, this fails for moments, as in general $E(x^p)$ is not equal to $E(x)^p$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A common approach to quantitatively and exactly analyze probabilistic programs is to employ probabilistic model checking techniques [BK08, KNP11, DJKV17, KZH+11, HJVC+21].

Exact inference for computing precise posterior distributions for probabilistic programs has been studied in [GMV16, HdBM20, NCR+16, CRN+13, SRM21]. An interesting direction for future research is using our techniques to assist probabilistic inference in the presence of loops.

A different approach to characterize the distributions of program variables are statistical methods such as Monte Carlo and hypothesis testing [YS06]. Simulations are however performed on a chosen finite number of program steps and do not provide guarantees over a potentially infinite execution, such as unbounded loops, limiting thus their use (if at all) for invariant generation.

In [MM05], a deductive approach, the *weakest pre-expectation calculus*, for reasoning about PPs with discrete program variables is introduced. Based on the weakest pre-expectation calculus, [KMMM10] presents the first template-based approach for generating linear quantitative invariants for PPs. Other works [FZJ+17, CHWZ15] also address the synthesis of non-linear invariants or employ *martingale* expressions [BEFH16]. All of these works target a slightly different problem and, unlike our approach, rely on templates. The first data-driven technique for invariant generation for PPs is presented in [BTP+22].

Another line of related work comes with computing bounds over expected values [BGP+16, Kar94, CFGG20] and higher moments [KUH19, WHR21]. The approach in [BGP+16] can provide bounds for higher moments and can handle non-linear terms at the price of producing more conservative bounds. In contrast, our approach natively supports probabilistic polynomial assignments and provides a precise symbolic expression for higher moments.

The technique presented in [BKS19] automates the generation of so-called moment-based invariants for a subclass of PPs with polynomial probabilistic updates and sets the basis for fully automatic exact higher moment computation. Relative to our approach, [BKS19] supports neither if-statements (thus also no guarded loops), state-dependent distribution parameters, nor circular variable dependencies. Our work establishes stronger theoretical foundations.

2.9 Conclusion

We describe a fully automated approach for inferring exact higher moments for program variables of a large class of probabilistic loops with complex control flow, polynomial assignments, symbolic constants, circular dependencies among variables, and potentially uncountable state spaces. Our work uses program transformations to normalize and simplify probabilistic programs while preserving the joint distribution of program variables. We propose a power reduction technique for finite program variables to ease the complex polynomial arithmetic of probabilistic programs. We prove soundness and completeness

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of our approach, by establishing the theory of moment-computable probabilistic loops. We demonstrate use cases of exact higher moments in the context of computing tail probabilities and recovering distributions from moments. Our experimental evaluation illustrates the applicability of our technique, solving several examples whose automation so far was not yet supported by the state-of-the-art in probabilistic program analysis.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Automated Sensitivity Analysis

This chapter is based on the following publication [MMK23]:

Marcel Moosbrugger, Julian Müllner, and Laura Kovács. Automated Sensitivity Analysis for Probabilistic Loops. In Proc. of iFM, 2023.

3.1 Problem Statement

A challenging task in the analysis of probabilistic programs comes from the fact that values, or even value distributions, of symbolic parameters used within program expressions over probabilistic program variables are often unknown. Sensitivity analysis aims to quantify how small changes in such parameters influence computation results [ABH+21, BEG+18]. Sensitivity analysis thus provides additional information about the probabilistic program executions, even if some parameters are (partially) unknown. This sensitivity information can further be used, among others, in code optimization: sensitivity information quantifies the influence of parameters on the program variables, allowing to derive cost-effective estimates and optimize expected runtimes of probabilistic loops.

The sensitivity analysis of probabilistic programs is however hard due to their intrinsic randomness: program variables are no longer assigned single values but rather hold probability distributions [BKS20a]. Uncountably infinite state spaces and non-linear assignments are further obstacles to the formal analysis of probabilistic programs. In recent years, several frameworks to *manually* reason about the sensitivity of probabilistic programs were proposed [ABH+21, BEG+18, VVB22]. However, the state-of-the-art in *automated* sensitivity analysis mainly focuses on loop-free programs such as *Bayesian networks* [CD02, CD04, BKS20b, SBK22] and statically-bounded loops [HWM18]. The technique presented in [WFC+20] supports loops with variable-dependent termination times, but can only verify that the sensitivities obey certain bounds. To the best of our

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

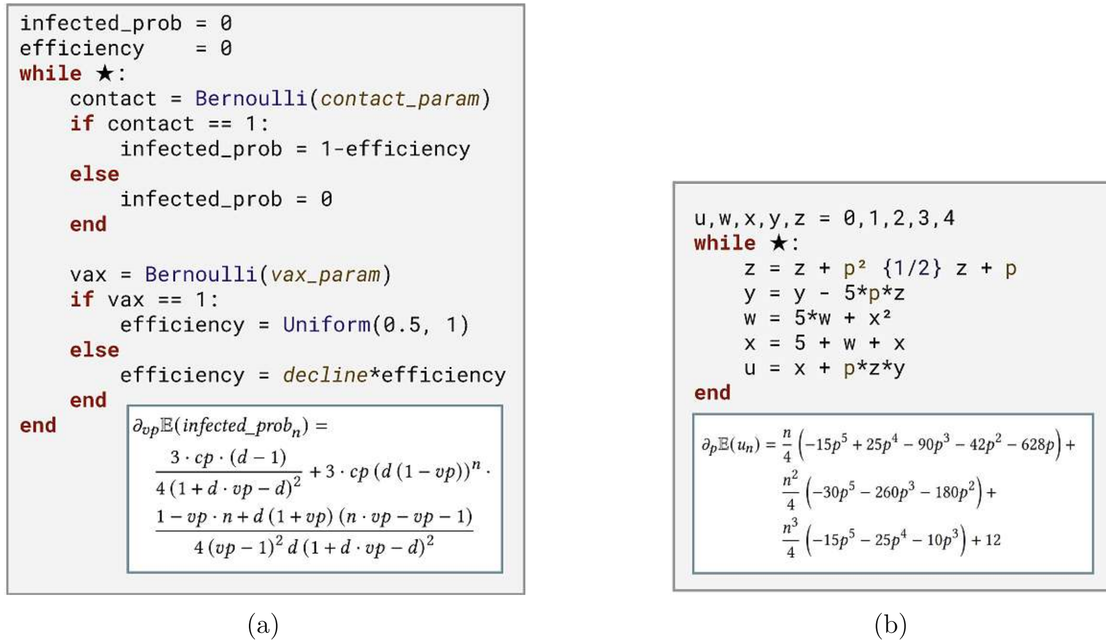


Figure 3.1: Two examples of parameterized probabilistic loops, where our approach automatically derives loop sensitivities ∂_p as polynomial expressions depending on the loop counter n and other parameters; for example `infected_prob` with respect to `vax_param` (Fig. 3.1a) or that of `u` with respect to `p` (Fig. 3.1b). Using these results, our approach shows that, when assuming `decline`=0.9, `contact_param`=0.7, after $n = 10$ time steps and currently having `vax_param`=0.1, then a small change ε in `vax_param` will decrease `infected_prob` by approximately 1.7ε in the next time step of Fig. 3.1a.

knowledge, up to now, there is no automated and exact method supporting the sensitivity analysis of (potentially) unbounded probabilistic loops.

We propose a fully automatic technique for the sensitivity analysis of unbounded probabilistic loops. The crux of our approach lies within the integration of methods from symbolic computation, probability theory and static analysis in order to automatically capture sensitivity information about probabilistic loops. Such an integrated framework allows us to also characterize a class of loops for which our technique is sound and complete.

Our framework for algebraic sensitivity analysis. We advocate the use of algebraic recurrences to model the behavior of probabilistic loops. We combine and adjust techniques from symbolic summation, partial derivatives, and probability theory to provide a step towards the exact and automated sensitivity analysis of probabilistic loops, even in the presence of uncountable state spaces and polynomial assignments. Figure 3.1 shows two probabilistic loops for which our approach automatically computes the sensitivities of program variables with respect to different parameters. For example, Fig. 3.1a depicts a probabilistic program, modelling the incidence of a disease within a population. More

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

precisely, it models the probability `infected_prob` that a single organism within the population is infected, in dependence on symbolic parameters that model the amount of social interaction (`contac_param`), the frequency of vaccinations (`vax_param`) and effect of a vaccination weakening over time (`decline`). Sensitivity analysis helps to reason about the influence of these parameters on the disease infection process, answering for example the question "How will an increase in the rate of vaccinations `vax_param` influence the probability `infected_prob` of an infection?". Our work provides an algebraic approach to answering such and similar questions.

In a nutshell, our technique computes exact closed-form solutions for the sensitivities of (higher) moments of program variables for all, possibly infinitely many, loop iterations. Higher moments are necessary to recover/estimate the value distributions of probabilistic loop variables and hence these moments help in inferring valuable sensitivity information for the variance or skewness. We utilize algebraic techniques in probabilistic loop analysis to model moments of program variables with linear recurrences, so-called *moment recurrences* [MSBK22a, BKS19]. However, moment recurrences do not support loops with intricate polynomial arithmetic, such as the loop in Figure 3.1b. To overcome this limitation, we propose the notion of *sensitivity recurrences*, which shortcut computing closed-forms for variable moments and directly model sensitivities via linear recurrence equations. In Figure 3.1b, the program variable w is independent of the parameter p. By exploiting the independence of program variables from parameters, *sensitivity recurrences* enable the exact sensitivity analysis for loops such as Figure 3.1b. We characterize a class of probabilistic loops for which we prove *sensitivity analysis via sensitivity recurrences* to be sound and complete.

Our contributions. We integrate symbolic computation, in particular symbolic summation and partial derivation, in combination with methods from probability theory into the landscape of probabilistic program reasoning. In particular, we argue that recurrence-based loop analysis yields a fully automated and precise way to derive sensitivity information over unknown symbolic parameters in probabilistic loops. As such, our contributions are as follows:

- We propose a fully automated approach for the sensitivity analysis of probabilistic loops based on *moment recurrences* (Section 3.3.1).
- We introduce *sensitivity recurrences* and an algorithm for sensitivity analysis going beyond *moment recurrences* (Section 3.3.2, Algorithm 1).
- We provide a precise characterization of the class of probabilistic loops for which *sensitivity recurrences* are provably sound and complete (Theorem 15).
- We describe an experimental evaluation demonstrating the feasibility of our techniques on many interesting probabilistic programs (Section 3.4).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

3.2 Preliminaries

We write $\mathbb{N}$ for the natural numbers, $\mathbb{R}$ for the reals, $\overline{\mathbb{Q}}$ for the algebraic numbers, and $\mathbb{K}[x_1, \ldots, x_k]$ for the polynomial ring with coefficients in the field $\mathbb{K}$. A polynomial consisting of a single monic term is a *monomial*. The expected value operator is denoted as $\mathbb{E}$.

3.2.1 Syntax and Semantics of Probabilistic Loops

Syntax. We focus on unbounded probabilistic while-loops, as illustrated by the two examples of Figure 3.1 and introduced in [MSBK22a]. Our programming model considers non-nested while-loops preceded by a variable initialization part, with the loop body being a sequence of (nested) if-statements and variable assignments. Unbounded probabilistic loops occur frequently when modeling dynamical systems. Guarded loops `while G: body` can be analyzed by considering the limiting behavior of unbounded loops of the form `while true:  if G: body`.

The right-hand side of every variable assignment is either a probability distribution with existing moments (e.g. Normal or Uniform) and constant parameters, or a probabilistic choice of polynomials in program variables, that is $\mathtt{x} = poly_1\{p_1\} \ldots poly_k\{p_k\}$, where $\mathtt{x}$ is assigned to $poly_i$ with probability p_i. Further, programs can be parameterized by symbolic constants which represent arbitrary real numbers.

Throughout this chapter, we refer to programs from our programming model simply by (probabilistic) loops or (probabilistic) programs. For a program $\mathcal{P}$ we denote the set of program variables by $\mathrm{Vars}(\mathcal{P})$ and the set of symbolic parameters by $\mathrm{Params}(\mathcal{P})$.

Dependencies between program variables is a syntactical notion introduced next, representing a central part in our technique.

Definition 16 (Variable Dependency)**.** *Let $\mathcal{P}$ be a probabilistic loop and $x, y \in Vars(\mathcal{P})$. We say that x* depends directly on *y, and write $x \to y$, if y appears in an assignment of x or an assignment of x occurs in an if-statement where y appears in the if-condition. Furthermore, we say that the dependency is* non-linear*, denoted as $x \xrightarrow{N} y$, if y appears non-linearly in an assignment of x.*

By $\twoheadrightarrow$ we denote the transitive closure of $\to$. Regarding non-linearity, we write $x \overset{N}{\twoheadrightarrow} y$, if at least one of the direct dependencies from x to y is non-linear.

Example 11. *In Figure 3.1b, we have (among others) $y \to z$, $w \xrightarrow{N} x$, $u \overset{N}{\twoheadrightarrow} w$, and $w \overset{N}{\twoheadrightarrow} u$. To illustrate the influence of if-conditions, in Figure 3.1a, note that efficiency $\to$ vax and infected_prob $\twoheadrightarrow$ vax.*

Semantics. Operationally, every probabilistic loop models an infinite-state Markov chain, which in turn induces a canonical probability space. Due to brevity, we omit the straightforward but rather technical construction of the Markov chains associated to

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

probabilistic loops. For more details, we refer the interested reader to [MSBK22a, Dur19]. For an arithmetic expression *Expr* in program variables, we denote by $Expr_n$ the stochastic process evaluating *Expr* after the nth loop iteration.

3.2.2 C-finite Recurrences

We recall relevant notions from algebraic recurrences [EvdPSW03, KP11].

A *sequence* of algebraic numbers is a function $u\colon \mathbb{N} \to \overline{\mathbb{Q}}$, succinctly denoted by $\langle u(n)\rangle_{n=0}^{\infty}$ or $\langle u(n)\rangle_n$. A *recurrence* for the sequence u of order $\ell \in \mathbb{N}$ is specified by a function $\mathrm{f} : \mathbb{R}^{\ell+1} \to \mathbb{R}$ and given by the equation $u(n{+}\ell) = \mathrm{f}(u(n{+}\ell{-}1), \dots, u(n{+}1), u(n), n)$. The *solutions* of a recurrence are the sequences satisfying the recurrence equation. Of particular relevance is the class of *linear recurrences with constant coefficients* or more shortly, *C-finite recurrences*. The sequence u satisfies a C-finite recurrence if $u(n{+}\ell) = c_{\ell-1}u(n{+}\ell{-}1) + c_{\ell-2}u(n{+}\ell{-}2) + \cdots + c_0 u(n)$ holds, where $c_0, \dots, c_{\ell-1} \in \overline{\mathbb{Q}}$ are constants and $c_0 \neq 0$. Every C-finite recurrence is associated with its *characteristic polynomial* $x^n - c_{\ell-1}x^{\ell-1} - \cdots - c_1 x - c_0$. The solutions of C-finite recurrences can always be computed [KP11] and written in closed-form as *exponential polynomials*. More precisely, if $\langle u(n)\rangle_n$ is the solution to a C-finite recurrence, then $u(n) = \sum_{k=1}^{r} P_k(n)\lambda_k^n$ where $P_k(n) \in \overline{\mathbb{Q}}[n]$ and $\lambda_1, \dots, \lambda_r$ are the roots of the characteristic polynomial. The properties of C-finite recurrences also hold for systems of C-finite recurrences (systems of linear recurrence equations with constant coefficients, specifying multiple sequences).

3.2.3 Higher Moment Analysis using Recurrences

For a random variable x, its higher moments are defined as $\mathbb{E}(x^k)$ for $k \in \mathbb{N}$. More generally, mixed moments for a set of random variables S are expected values of monomials in S. Recent works in probabilistic program analysis [MSBK22a, BKS20c] introduced techniques and tools based on C-finite recurrences to compute higher moments of program variables for probabilistic loops. For example, for a probabilistic loop, $k \in \mathbb{N}$ and a program variable x, a closed-form solution for the kth higher moment of x parameterized by the loop iteration n, that is $\mathbb{E}(x_n^k)$, is computed in [MSBK22a] using the Polar tool. This is achieved by first normalizing the program to eliminate if-statements and ensure every variable is only assigned once in the loop body. Then, a system of C-finite recurrences is constructed that models expected values of monomials in program variables. More precisely, for a monomial M in program variables, the work of [MSBK22a] constructs a linear recurrence equation, relating the expected value of M in iteration $n{+}1$ to the expected values of program variable monomials in iteration n. The linear recurrence for the expected value of M in iteration $n{+}1$ is constructed by starting with the expression $\mathbb{E}(M_{n+1})$ and replacing variables contained in the expression by their assignments bottom-up as they appear in the loop body. Throughout, the linearity of expectation is used to convert expected values of polynomials into expected values of monomials.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

We adopt the setting of [MSBK22a, BKS20c] and refer by *moment recurrences* to the recurrence equations these techniques construct for moments of program variables.

Definition 17 (Moment Recurrence)**.** *Let $\mathcal{P}$ be a probabilistic loop and M a monomial in Vars($\mathcal{P}$). A* moment recurrence *for M is an equation $\mathbb{E}(M_{n+1}) = \sum_{i=1}^{r} c_i \cdot \mathbb{E}(W_n^{(i)})$ where $c_i \in \overline{\mathbb{Q}}$ and all $W^{(i)}$ are monomials in Vars($\mathcal{P}$).*

In order to compute a closed-form solution for $\mathbb{E}(x_n^k)$, we employ [MSBK22a] to first compute a moment recurrence R for the monomial x^k. Next, we derive moment recurrences for all monomials $W^{(i)}$ in R (cf. Definition 17) to construct a system of C-finite recurrences.

Example 12. *Consider the program from Figure 3.1a. For a more succinct representation, we abbreviate the symbolic parameters as cp := contact_param; vp := vax_param and d := decline. The first moments of the program variables are modeled through the following system of C-finite recurrences [MSBK22a]:*

$$\mathbb{E}(\mathit{infected_prob}_{n+1}) = cp - cp \cdot \mathbb{E}(\mathit{efficiency}_n)$$
$$\mathbb{E}(\mathit{efficiency}_{n+1}) = (d - d \cdot vp) \cdot \mathbb{E}(\mathit{efficiency}_n) + \frac{3}{4} \cdot vp$$

The initial values of $\mathbb{E}(\mathit{infected_prob}_n)$ and $\mathbb{E}(\mathit{efficiency}_n)$ are both 0. The system can be automatically solved [KP11] to obtain closed-form solutions, which are, when expanded, exponential polynomials, e.g. for $\mathbb{E}(\mathit{infected_prob}_n)$:

$$\mathbb{E}(\mathit{infected_prob}_n) = cp + \frac{3 \cdot vp \cdot cp \cdot \left((d - d \cdot vp)^{n-1} - 1\right)}{4(d \cdot vp - d + 1)}$$

We note that moment recurrences do not always exist. Moreover, termination is not guaranteed when recursively inferring the moment recurrences for all monomials $W^{(i)}$ in Definition 17 in order to construct a C-finite system.

Example 13. *To illustrate that the approach based on moment recurrences does not work unconditionally, consider the loop from Figure 3.1b and construct the moment recurrence $\mathbb{E}(w_{n+1}) = 5 \cdot \mathbb{E}(w_n) + \mathbb{E}(x_n^2)$. Since the recurrence contains $\mathbb{E}(x_n^2)$, we require the moment recurrence $\mathbb{E}(x_{n+1}^2) = \mathbb{E}((5 + w_{n+1} + x_n)^2) = \mathbb{E}(w_{n+1}^2) + \dots$ which requires the recurrence for $\mathbb{E}(w_n^2)$. This in turn necessitates a recurrence for $\mathbb{E}(x_n^4)$, which necessitates the recurrence for $\mathbb{E}(w_n^4)$ and so on. This process will repeatedly require recurrences for increasing moments of x_n and w_n, implying that this process will not terminate.*

To circumvent variable dependencies and compute closed-forms of moment recurrences, we note that the following two conditions on the probabilistic loops ensure existence and computability of higher order moments.

Definition 18 (Admissible Loop)**.** *A loop is* admissible *if*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

1. *all variables in branching conditions only assume values in a finite set (i.e. they are* finite valued*), and*

2. *no variable x is non-linearly self-dependent ($x \overset{N}{\not\twoheadrightarrow} x$)* [1]*.*

Example 14. *The probabilistic loop in Figure 3.1a is admissible. However, the program in Figure 3.1b is not admissible. It does not satisfy condition 2: the variable x depends linearly on w and w depends quadratically on x; therefore, x is non-linearly self-dependent.*

Admissible probabilistic loops are *moment-computable* [MSBK22a], that is, higher moments of program variables admit computable closed-forms as exponential polynomials. The restriction on finite valued variables in branching conditions is necessary to guarantee computability and completeness: a single branching statement involving an unbounded variable renders the program model Turing-complete [MSBK22a].

3.3 Sensitivity Analysis

In this section, we study the sensitivity of program variable moments with respect to symbolic parameters. We present two exact and fully automatic methods to answer the question of how small changes in symbolic parameters influence the moments of program variables. As such, we exploit the fact that closed-forms for variable moments in admissible loops are computable (Section 3.3.1). We further go beyond the admissible loop setting (Section 3.3.2) and devise a sensitivity analysis technique applicable to some non-admissible loops, such as the program in Figure 3.1b.

Definition 19 (Sensitivity)**.** *Let $\mathcal{P}$ be a probabilistic loop, $x \in Vars(\mathcal{P})$ and $p \in Params(\mathcal{P})$. The* sensitivity *of the kth moment of x with respect to p, denoted as $\partial_p \mathbb{E}(x_n^k)$, is defined as the partial derivative of $\mathbb{E}(x_n^k)$ with respect to p, and parameterized by loop counter n. For monomials M of variables, the sensitivity $\partial_p \mathbb{E}(M_n)$ is defined analogously.*

Similar to *moment computability* [MSBK22a], we define a program to be *sensitivity computable* if the sensitivities of all the variables' expected values are expressible in closed-form.

Definition 20 (Sensitivity Computability)**.** *Let $\mathcal{P}$ be a probabilistic program and $p \in Params(\mathcal{P})$. $\mathcal{P}$ is* sensitivity computable *with respect to p, if for every variable $x \in Vars(\mathcal{P})$ the sensitivity $\partial_p \mathbb{E}(x_n)$ has an exponential polynomial closed-form that is computable.*

[1] While [MSBK22a] allows arbitrary dependencies among finite valued variables, we omit this generalization for simplicity. Nevertheless, our results also apply to admissible loops with arbitrary dependencies among finite valued variables.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3.1 Sensitivity Analysis for Admissible Loops

As mentioned in Section 3.2, for admissible loops, any moment of every program variable admits a closed-form solution as an exponential polynomial which is computable. That is, for a program variable x and $k \in \mathbb{N}$, the kth moment of x can be written as $\mathbb{E}(x_n^k) = \sum_{j=0}^{r} P_j(n)\lambda_j^n$, where $P_j \in \overline{\mathbb{Q}}[n]$ and $\lambda_j \in \overline{\mathbb{Q}}$ may contain symbolic parameters. We next show that based on the closed-forms of variable moments, we can compute exponential polynomials representing the sensitivities of moments on parameters.

Theorem 12 (Admissible Sensitivities)**.** *Let $\mathcal{P}$ be an admissible program, $x \in Vars(\mathcal{P})$, $p \in Params(\mathcal{P})$, and $k \in \mathbb{N}$. Then, the sensitivity $\partial_p\mathbb{E}(x_n^k)$ has an exponential polynomial closed-form that is computable.*

Proof. Because $\mathcal{P}$ is admissible, $\mathbb{E}(x_n^k)$ can be expressed as an exponential polynomial. We show that the sensitivity can be expressed as an exponential polynomial by expanding $\mathbb{E}(x_n^k)$ into a sum of exponential monomials: $\mathbb{E}(x_n^k) = \sum_{j=0}^{r} P_j(n)\lambda_j^n = \sum_{j=0}^{r}\sum_{i=0}^{m_j} M_{ij}(n)\lambda_j^n$, where m_j is the number of monomials in P_j and every M_{ij} is a monomial. Note that every M_{ij} and λ_j may depend on the symbolic constant p. The derivative of the exponential monomials can then be obtained by applying the product rule for derivatives:

$$\begin{aligned}\partial_p\mathbb{E}(x_n^k) &= \sum_{j=0}^{r}\sum_{i=0}^{m_j}(\partial_p M_{ij}(n))\lambda_j^n + M_{ij}(n)\cdot n\cdot(\partial_p\lambda_j)\cdot\lambda_j^{n-1}\\ &= \sum_{j=0}^{r}(\partial_p P_j(n) + P_j(n)\cdot n\cdot\partial_p\lambda_j\cdot\frac{1}{\lambda_j})\lambda_j^n\end{aligned}$$

It is left to show that the exponential polynomial $\partial_p\mathbb{E}(x_n^k)$ is computable. Because $\mathcal{P}$ is admissible, an exponential polynomial for $\mathbb{E}(x_n^k)$ is computable. Now, the second claim follows from the fact that exponential polynomials are elementary and that the derivative of any elementary function is computable. □

As a corollary, admissible loops are sensitivity computable. Although *sensitivity computability* only refers to first moments, Theorem 12 shows that for admissible loops, sensitivities of *all* higher moments of program variables admit a computable closed-form.

Example 15. *Consider Figure 3.1a. In Example 12 we stated the closed-form solutions of $\mathbb{E}(infected_prob_n)$. The sensitivities of the respective expected values can be computed by symbolic differentiation and, by Theorem 12, can be expanded to exponential polynomials. For example, the following expression describes the sensitivity of $\mathbb{E}(infected_prob_n)$ with respect to the parameter vp:*

$$\begin{aligned}\partial_{vp}\mathbb{E}(infected_prob_n) =& \frac{3\cdot cp(1-vp\cdot n + d(1+vp)(n\cdot vp - vp - 1))(d(1-vp))^n}{4(vp-1)^2 d(1+d\cdot vp - d)^2}\\ &+ \frac{3\cdot cp\cdot(d-1)}{4(1+d\cdot vp-d)^2}\end{aligned}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.3.2 Sensitivity Analysis for Non-Admissible Loops

In general, moments of program variables of non-admissible loops do not satisfy linear recurrences. Therefore, we cannot utilize closed-forms of the moments for sensitivity analysis. Nevertheless, sensitivity analysis is feasible even for some non-admissible loops. In this section, we propose a novel sensitivity analysis approach applicable to non-admissible loops. Moreover, we characterize the class of (non-admissible) loops for which our method is sound and complete.

For admissible loops, linear recurrences describing variable moments can be used as an intermediary step to compute sensitivities. The core of our approach towards handling non-admissible loops is to shortcut moment recurrences and devise recurrences directly for sensitivities. Due to independence with respect to the sensitivity parameter, sensitivities of program variables can follow a linear recurrence even though their moments do not. We illustrate the idea of our new method on the non-admissible loop from Figure 3.1b.

Example 16. *Consider the non-admissible program from Figure 3.1b. The moment recurrences for all program variables are:*

$$\begin{aligned} \mathbb{E}(z_{n+1}) &= \mathbb{E}(z_n) + 0.5 \cdot (p + p^2) & \mathbb{E}(y_{n+1}) &= \mathbb{E}(y_n) - 5p \cdot \mathbb{E}(z_{n+1}) \\ \mathbb{E}(w_{n+1}) &= 5 \cdot \mathbb{E}(w_n) + \mathbb{E}(x_n^2) & \mathbb{E}(x_{n+1}) &= 5 + \mathbb{E}(w_{n+1}) + \mathbb{E}(x_n) \\ \mathbb{E}(u_{n+1}) &= \mathbb{E}(x_{n+1}) + p \cdot \mathbb{E}(zy_{n+1}) \end{aligned}$$

As illustrated in Example 13, we cannot complete the recurrences to a C-finite system because both w and x are non-linearly self-dependent. Therefore, we cannot compute closed-form solutions for $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$. However, we can shortcut solving for $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$ by differentiating the moment recurrences with respect to p and establish recurrences directly for the sensitivities:

$$\begin{aligned} \partial_p\mathbb{E}(z_{n+1}) &= \partial_p\mathbb{E}(z_n) + 0.5 \cdot (1 + 2p) \\ \partial_p\mathbb{E}(y_{n+1}) &= \partial_p\mathbb{E}(y_n) - 5p \cdot \partial_p\mathbb{E}(z_{n+1}) - 5 \cdot \mathbb{E}(z_{n+1}) \\ \partial_p\mathbb{E}(w_{n+1}) &= 5 \cdot \partial_p\mathbb{E}(w_n) + \partial_p\mathbb{E}(x_n^2) \\ \partial_p\mathbb{E}(x_{n+1}) &= \partial_p\mathbb{E}(w_{n+1}) + \partial_p\mathbb{E}(x_n) \\ \partial_p\mathbb{E}(u_{n+1}) &= \partial_p\mathbb{E}(x_{n+1}) + \mathbb{E}(zy_{n+1}) + p \cdot \partial_p\mathbb{E}(zy_{n+1}) \end{aligned}$$

Now, because the variables w and x do not depend on the parameter p, we conclude that $\partial_p\mathbb{E}(w_n) \equiv \partial_p\mathbb{E}(x_n) \equiv 0$. The sensitivity recurrences thus simplify:

$$\begin{aligned} \partial_p\mathbb{E}(z_{n+1}) &= \partial_p\mathbb{E}(z_n) + \frac{1 + 2p}{2} \\ \partial_p\mathbb{E}(y_{n+1}) &= \partial_p\mathbb{E}(y_n) - 5p \cdot \partial_p\mathbb{E}(z_{n+1}) - 5 \cdot \mathbb{E}(z_{n+1}) \\ \partial_p\mathbb{E}(u_{n+1}) &= \mathbb{E}(zy_{n+1}) + p \cdot \partial_p\mathbb{E}(zy_{n+1}) \end{aligned}$$

We can interpret sensitivities such as $\partial_p\mathbb{E}(z_n)$ or $\partial_p\mathbb{E}(u_n)$ as atomic recurrence variables. In the resulting recurrences, all variables with non-linear self-dependencies vanished.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Therefore, the recurrences can be completed to a C-finite system and solved by existing techniques, even though $\mathbb{E}(w_n)$ *and* $\mathbb{E}(x_n)$ *are not C-finite. The resulting system of recurrences consists of all recurrences for sensitivities and moments that appear on the right-hand side of another recurrence. That is, the system of recurrences consists of the sensitivity recurrences for* $\partial_p\mathbb{E}(z)$, $\partial_p\mathbb{E}(y)$, $\partial_p\mathbb{E}(u)$, $\partial_p\mathbb{E}(yz)$, $\partial_p\mathbb{E}(z^2)$ *and the moment recurrences for* $\mathbb{E}(z)$, $\mathbb{E}(y)$, $\mathbb{E}(yz)$, $\mathbb{E}(z^2)$.

Motivated by Example 16, we introduce the notion of *sensitivity recurrences.*

Definition 21 (Sensitivity Recurrence)**.** *Let* $\mathcal{P}$ *be a program,* $p \in \mathit{Params}(\mathcal{P})$ *a symbolic parameter,* M *a monomial in* $\mathit{Vars}(\mathcal{P})$ *and let* $\mathbb{E}(M_{n+1}) = \sum_{i=1}^{r} c_i \cdot \mathbb{E}(W_n^{(i)})$ *be the moment-recurrence of* M*. Then the* sensitivity recurrence *of* M *with respect to* p *is defined as*

$$\boxed{\partial_p\mathbb{E}(M_{n+1})} := \frac{\partial\mathbb{E}(M_{n+1})}{\partial p} = \frac{\partial}{\partial p}\left(\sum_{i=1}^{r} c_i \cdot \mathbb{E}\Big(W_n^{(i)}\Big)\right)$$
$$\boxed{= \sum_{i=1}^{r}\left(\frac{\partial}{\partial p}c_i\right) \cdot \mathbb{E}\Big(W_n^{(i)}\Big) + c_i \cdot \partial_p\mathbb{E}\Big(W_n^{(i)}\Big)} \tag{3.1}$$

The sensitivity recurrence of M equates the sensitivity of M at iteration $n{+}1$ to moments *and sensitivities* at iteration n. Along the ideas in Example 16, we provide with Algorithm 1 a procedure for sensitivity analysis also applicable to non-admissible loops. The idea of Algorithm 1 is to determine $\partial_p\mathbb{E}(M_n)$ by constructing a C-finite system consisting of all necessary recurrence equations for the moments and sensitivities of program variables. As illustrated in Example 16, we can exploit the independence of variables from the sensitivity parameter p to simplify the problem: if a monomial W' is independent from p then $\partial_p\mathbb{E}(W'_n) \equiv 0$. Moreover, if p does not appear in the constant c_i of Equation (3.1), then $(\partial/\partial p)c_i = 0$, and hence the moment recurrence of W' does not need to be constructed (lines 8–9 of Algorithm 1). This is essential if the *expected value* of W' does not admit a closed-form. Algorithm 1 is sound by construction, however, termination is non-trivial. In the remainder of this section, we formalize the notion of parameter (in)dependence and give a characterization of the class of non-admissible loops for which Algorithm 1 terminates. As a consequence of Algorithm 1, we show that sensitivity recurrences yield an exact and complete technique for sensitivity analysis (Theorem 15).

Definition 22 (p-Dependent Variable)**.** *Let* $\mathcal{P}$ *be a program and* $p \in \mathit{Params}(\mathcal{P})$ *a parameter. A variable* $x \in \mathit{Vars}(\mathcal{P})$ *is* p-dependent*, if (1)* p *appears in an assignment of* x*, (2)* x *depends on some* $y \in \mathit{Vars}(\mathcal{P})$ $(x \twoheadrightarrow y)$ *and* y *is* p*-dependent or (3) an assignment of* x *occurs in an if-statement where* p *appears in the if-condition. A variable is* p-independent *if it is not* p*-dependent. A monomial* M *in program variables is* p*-dependent if* M *contains at least one* p*-dependent variable, otherwise it is* p*-independent.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 1 Computing Sensitivities via Sensitivity Recurrences

Input: program $\mathcal{P}$, monomial M in $\text{Vars}(\mathcal{P})$, $p \in \text{Params}(\mathcal{P})$
Output: closed-form for $\partial_p \mathbb{E}(M_n)$

1: **if** M is p-independent **then**
2:     **return** 0
3: **end if**
4: $\mathit{Eqs} \leftarrow \emptyset$; $\mathit{Mom} \leftarrow \emptyset$; $\mathit{Sens} \leftarrow \{M\}$
5: **while** $\mathit{Sens} \neq \emptyset$ **do** ▷ *Add all necessary sensitivity recurrences*
6:     pick $W \in \mathit{Sens}$; $\mathit{Sens} \leftarrow \mathit{Sens} \setminus \{W\}$
7:     $\mathit{SRec} \leftarrow$ sensitivity recurrence of W
8:     Replace every $\partial_p \mathbb{E}(W'_n)$ in SRec by 0 if W' is p-independent
9:     Replace every $(\partial/\partial pc)\mathbb{E}(W'_n)$ in SRec by 0 if $(\partial/\partial pc) = 0$
10:     $\mathit{Eqs} \leftarrow \mathit{Eqs} \cup \{\mathit{SRec}\}$
11:     Add to Sens all monomials W' s.t. $\partial_p \mathbb{E}(W'_n)$ in SRec
12:     $\hookrightarrow$ and the sensitivity recurrence of $W' \notin \mathit{Eqs}$
13:     Add to Mom all monomials W' s.t. $\mathbb{E}(W'_n)$ in SRec
14: **end while**
15: **while** $\mathit{Mom} \neq \emptyset$ **do** ▷ *Add all necessary moment recurrences*
16:     pick $W \in \mathit{Mom}$; $\mathit{Mom} \leftarrow \mathit{Mom} \setminus \{W\}$
17:     $\mathit{MRec} \leftarrow$ moment recurrence of W
18:     $\mathit{Eqs} \leftarrow \mathit{Eqs} \cup \{\mathit{MRec}\}$
19:     Add to Mom all monomials W' s.t. $\mathbb{E}(W'_n)$ in MRec
20:     $\hookrightarrow$ and the moment recurrence of $W' \notin \mathit{Eqs}$
21: **end while**
22: $S \leftarrow$ solve system of C-finite recurrences Eqs
23: **return** closed-form of $\partial_p \mathbb{E}(M_n)$ from S

For any p-independent monomial M in program variables, the corresponding sensitivity $\partial_p \mathbb{E}(M_n)$ is zero (by using induction on n and applying Definition 22).

Lemma 13. *Let $\mathcal{P}$ be a program, $p \in \mathit{Params}(\mathcal{P})$ a symbolic parameter and M a p-independent monomial in $\mathit{Vars}(\mathcal{P})$, then it holds that the sensitivity variable of M is zero, i.e., $\forall n \geq 0 : \partial_p \mathbb{E}(M_n) = 0$.*

In Example 16, the moments $\mathbb{E}(w_n)$ and $\mathbb{E}(x_n)$ do not admit closed-forms. We resolved this issue by differentiating all moment recurrences and working directly with the sensitivity recurrences, where the moment recurrences for w and x vanished. Crucial for this phenomenon is the fact that the variables w and x are independent of the sensitivity parameter p.

However, a second fact is necessary to guarantee that the moment recurrences of w and x do not appear in the resulting system of recurrences: Assume some new variable v depends on x and has the moment recurrence $\mathbb{E}(v_{n+1}) = \mathbb{E}(v_n) + p \cdot \mathbb{E}(x_n)$. Then the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

sensitivity recurrence for v is given by $\partial_p \mathbb{E}(v_{n+1}) = \partial_p \mathbb{E}(v_n) + \mathbb{E}(x_n) + p \cdot \partial_p \mathbb{E}(x_n)$. Even though x itself is p-independent, $\mathbb{E}(x_n)$ remains in the sensitivity recurrence of v because the coefficient of $\mathbb{E}(x_n)$ contains the parameter p. A similar effect occurs if the moment recurrence for v was $\mathbb{E}(v_{n+1}) = \mathbb{E}(v_n) + \mathbb{E}(z_n x_n)$, because z is p-dependent.

Our goal is to characterize the class of probabilistic loops for which sensitivity recurrences yield a sound and complete method for sensitivity analysis. Hence, we need to capture the notion that some dependencies between variables are free of multiplicative factors involving the sensitivity parameter. We do this in the following definition by refining our dependency relation $\twoheadrightarrow$.

Definition 23 (p-Influenced Dependency)**.** *Let $\mathcal{P}$ be a program with parameter $p \in Params(\mathcal{P})$ and $x, y \in Vars(\mathcal{P})$ with $x \rightarrow y$. Then, the direct dependency between x and y is* p-influenced*, written as $x \rightarrow_p y$, if at least one of the following conditions hold:*

- *An assignment of x contains y and occurs in an if-statement with the if-condition involving p or a p-dependent variable.*
- *An assignment of x contains y and is a probabilistic choice with some probability of the choice depending on p.*
- *An assignment of x contains a term $c \cdot M \cdot y$ where c is constant and M is a monomial in program variables (possibly containing y). Moreover, either c contains p or M contains a p-dependent variable.*

If $x \twoheadrightarrow y$, we write $x \twoheadrightarrow_p y$ if some dependency from x to y is p-influenced. If $x \twoheadrightarrow y$ and $x \not\twoheadrightarrow_p y$ we call the dependency between x and y p-free*.*

Definition 23 covers all cases in the construction of moment recurrences that introduce multiplicative factors depending on the sensitivity parameter p [MSBK22a].

More concretely, assume $\mathcal{P}$ to be a program and $x \in \text{Vars}(\mathcal{P})$. The moment recurrence of x contains expected values of monomials M of program variables. Additionally, the moment recurrences of any M will again contain expected values of monomials of program variables and so on. We capture all of these monomials with the notion of *descendant monomials* in Definition 24. Intuitively, to construct a system of moment recurrences for $\mathbb{E}(x_n)$ one needs to include the moment recurrences of all descendants of x.

Definition 24 (Descendant Monomial)**.** *Let $\mathcal{P}$ be a program, $x \in Vars(\mathcal{P})$, and M a monomial in program variables. The monomial M is a* descendant *of the variable x if (1) $M = x$, or (2) M occurs in the moment recurrence of a monomial W and W is a descendant of x. The variable x is an* ancestor *of M.*

There is a dependency between x and any variable of any descendant of x, which means $x \twoheadrightarrow y$ for every descendant M of x and every variable y in M. Our dependency relation

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

from Definition 23 allows us to pinpoint the variables in the moment recurrence of any descendant of x (Definition 24) with a multiplicative factor involving the sensitivity parameter. Definitions 23 and 24 together with the procedure constructing moment recurrences yield:

Lemma 14 (p-Influenced Moment Recurrence)**.** *Let $\mathcal{P}$ be a program, $x \in Vars(\mathcal{P})$, and $p \in Params(\mathcal{P})$. Assume M is a monomial in program variables descending from x. Let W be a monomial in M's moment recurrence with non-zero coefficient c. If the parameter p occurs in c, then for all variables y in W we have $x \twoheadrightarrow_p y$. Moreover, if some variable z in W is p-dependent, then for all variables y in W different from z we have $x \twoheadrightarrow_p y$.*

We now state our main result (Theorem 15) describing the class of probabilistic loops for which Algorithm 1 terminates and, hence, sensitivity recurrences are sound and complete. We characterize the class of loops in terms of our dependency relations as well as variables with non-linear self-dependencies, which we refer to as *defective* variables.

Definition 25 (Defective Variables)**.** *Let $\mathcal{P}$ be a program and $x \in Vars(\mathcal{P})$, then x is* defective *if $x \overset{N}{\twoheadrightarrow} x$. Otherwise, x is* effective*.*

Theorem 15 (Non-Admissible Sensitivities)**.** *Let $\mathcal{P}$ be a probabilistic program, $p \in Params(\mathcal{P})$, $x \in Vars(\mathcal{P})$, and assume all the following conditions:*

1. *All variables occuring in branching conditions are finite.*
2. *All defective variables are p-independent.*
3. *All dependencies on defective variables are p-free.*

Then, for every monomial M in program variables descending from x, Algorithm 1 terminates on input $\mathcal{P}$, M and p.

Proof. First, we cover the case where the monomial M contains a defective variable. If M contains a defective variable y, then y must be p-independent by condition 2. As M is a descendant of x, we have $x \twoheadrightarrow y$. Moreover, if there exists a p-dependent variable z in M different from y, Lemma 14 gives us $x \twoheadrightarrow_p y$. However, $x \twoheadrightarrow_p y$ contradicts our condition 3 that all dependencies on defective variables must be p-free. Hence, the monomial M is p-independent and Algorithm 1 terminates on line 2.

For the second, more involved case, assume all variables in M are effective and possibly p-dependent. Algorithm 1 does not terminate if and only if the algorithm adds infinitely many monomials W' to the set *Sens* one line 11 or to the set *Mom* on lines 13 or 19. Every monomial W' added to the set *Mom* occurs in the moment recurrence of W from line 16. Moreover, every monomial W' added to the set *Sens* occurs in the sensitivity recurrence of W from line 6 and hence also occurs in the moment recurrence of W. That

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is because the sensitivity recurrence and the moment recurrence of W share the same monomials (Definition 21).

In [MSBK22a], the authors showed that every monomial W' occurring in the moment recurrence of a monomial W decreases with respect to a well-founded ordering if (A) all variables in branching conditions are finite, and (B) all variables in W and W' are effective. Premise (A) matches our condition 1. Therefore, to show that only finitely many monomials are added to *Sens* and *Mom* (and hence establish termination of Algorithm 1), *it suffices to show that all monomials W' added to Sens and Mom only contain effective variables.*

First, note that every monomial W' added to *Sens* or *Mom* is a descendant of the variable x. This holds because the algorithm starts with $\mathit{Sens} = \{M\}$, $\mathit{Mom} = \emptyset$, the monomial M is a descendant of x, and W' occurs in the moment recurrence of some $W \in \mathit{Sens} \cup \mathit{Mom}$.

Claim: All monomials W' added to Sens on line 11 only contain effective variables. Towards a contradiction, assume some monomial W' is added to *Sens* on line 11 and W' contains a defective variable y. By condition 2, y is p-independent. By Lemma 14 and condition 3, all variables in W' are p-independent. Hence, the monomial W' is p-independent and $\partial_p \mathbb{E}(W'_n)$ was replaced by 0 on line 8. Therefore, W' could not have been added to *Sens* on line 11.

Claim: All monomials W' added to Mom on line 13 only contain effective variables. Towards a contradiction, assume some monomial W' is added to *Mom* on line 13 and W' contains a defective variable y. The monomial W' occurs in the sensitivity recurrence of W (fixed at line 6) with coefficient $(\partial/\partial p c)$. Therefore, W' occurs in the moment recurrence of W with coefficient c. By Lemma 14 and condition 3, the constant c does not contain the parameter p. Hence, $(\partial/\partial p c) = 0$ and $\mathbb{E}(W'_n)$ was replaced by 0 on line 9. Therefore, W' could not have been added to *Mom* on line 13.

Claim: All monomials W' added to Mom on line 19 only contain effective variables. First, note that for all monomials W' added to *Mom* on line 13, the corresponding coefficient $(\partial/\partial p c) \neq 0$ and hence c must contain the parameter p. Therefore, by Lemma 14, for all variables y in all monomials W' added to *Mom* in the first while-loop, we have $x \twoheadrightarrow_p y$. By transitivity of $\twoheadrightarrow_p$, we get $x \twoheadrightarrow_p y$ for all variables y in all monomials W' added to *Mom* on line 19. Therefore, all W' added to *Mom* on line 19 cannot contain defective variables by condition 3. □

Theorem 15 characterizes the class of probabilistic loops for which sensitivity recurrences provide a sound and complete method for sensitivity analysis. As an immediate corollary, this class of loops is sensitivity computable because every variable is a descendant of itself. Note that all conditions of Theorem 15 are statically checkable: the concepts of defective variables, p-independent variables, and p-free dependencies are purely syntactic notions. Moreover, program variables occurring in branching conditions only admitting finitely many values can be verified using standard techniques based on *abstract interpretation*.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Theorem 15 also applies to sensitivity analysis for higher moments: let $v \in \mathrm{Vars}(\mathcal{P})$ and $k \in \mathbb{N}$, then Theorem 15 covers the sensitivity of v's kth moment if v^k is a descendant of some variable. Otherwise, v^k can be dealt with by introducing a fresh variable w and appending the assignment $w := v^k$ to $\mathcal{P}$'s loop body.

The proof of Theorem 15 provides an alternative argument for admissible loops being sensitivity computable (Theorem 12); as admissible loops do not contain defective variables by definition (Definition 18), the class of loops characterized by Theorem 15 subsumes the class of admissible loops.

3.4 Experiments and Evaluation

We evaluate our methods for sensitivity analysis for admissible loops (Section 3.3.1) and non-admissible loops (Section 3.3.2). Our techniques for sensitivity analysis extend the Polar framework [MSBK22a], which is publicly available at `https://github.com/probing-lab/polar`. For admissible loops, we use the existing functionality of the Polar framework to compute closed-forms for the moments of program variables.

Experimental Setup. We split our evaluation into two parts. First, we compute the sensitivities of (higher) moments of program variables for admissible loops by automatically differentiating the closed-forms of the variables' moments (Table 3.1). In the second part, we consider our method using sensitivity recurrences, which is also applicable to non-admissible loops (Table 3.2). To the best of our knowledge, our method provides the first exact and fully automatic tool to compute the sensitivities of (higher) moments of program variables for probabilistic loops. All our experiments have been executed on a machine with a 2.6 GHz Intel i7 (Gen 10) processor and 32 GB of RAM with a timeout (TO) of 120 s.

Differentiating Closed-Forms. Table 3.1 shows the evaluation of our sensitivity analysis technique for admissible loops (Section 3.3) on 11 benchmarks. The benchmarks consist of the running example from Figure 3.1a and parameterized probabilistic loops from the benchmarks in [MSBK22a], coming from literature on probabilistic program analysis [BEFH16, CS14, GKM13, BKS20b]. All the benchmarks contain at least one symbolic parameter with respect to which the sensitivities are computed. Table 3.1 shows that our approach is capable of computing the sensitivities of higher moments of program variables for challenging loops with various characteristics, such as discrete and continuous state spaces as well as drawing from common distributions.

Sensitivity Recurrences. Table 3.2 shows the evaluation of our sensitivity analysis technique from Algorithm 1 using *sensitivity recurrences*. The benchmarks consist of four non-admissible loops (Fig. 3.2) and six admissible loops from Table 3.1. Non-admissible loops are known to be notoriously hard to analyze automatically [ABK+22]. Table 3.2

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 3.1: Evaluation of the sensitivity computation for 11 admissible loops by differentiating closed-forms of variable moments. Rec: size of the recurrence system to compute the variables' moments; RT: runtime in seconds; TO: timeout.

Benchmark	Sensitivity	Rec, RT	Sensitivity	Rec, RT
50-Coin-Flips	$\partial_p\mathbb{E}(\mathrm{total})$	51, 1.56	$\partial_p\mathbb{E}(\mathrm{total}^2)$	TO, TO
Bimodal	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{x})$	3, 0.40	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{x}^2)$	5, 0.72
Component-Health	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{obs})$,	2, 0.61	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{obs}^2)$,	2, 0.62
Umbrella	$\partial_{\mathrm{u1}}\mathbb{E}(\mathrm{umbrella})$	2, 0.97	$\partial_{\mathrm{u1}}\mathbb{E}(\mathrm{umbrella}^2)$	2, 0.98
Gambler's Ruin	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{money})$	4, 11.2	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{money}^2)$	10, 64.6
Hawk-Dove	$\partial_{\mathrm{v}}\mathbb{E}(\mathrm{p1bal})$	1, 0.34	$\partial_{\mathrm{v}}\mathbb{E}(\mathrm{p1bal}^2)$	2, 0.67
Las-Vegas-Search	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{attempts})$	2, 0.57	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{attempts}^2)$	4, 7.31
1D-Random-Walk	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{x})$	1, 0.27	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{x}^2)$	2, 0.39
2D-Random-Walk	$\partial_{\mathrm{p_right}}\mathbb{E}(\mathrm{x})$	1, 0.28	$\partial_{\mathrm{p_right}}\mathbb{E}(\mathrm{x}^2)$	2, 0.41
Randomized-Response	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{p1})$	1, 0.29	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{p1}^2)$	2, 0.42
Vaccination (Fig. 3.1a)	$\partial_{\mathrm{vp}}\mathbb{E}(\mathrm{infected})$	2, 1.25	$\partial_{\mathrm{vp}}\mathbb{E}(\mathrm{infected}^2)$	2, 1.19

Table 3.2: Evaluation of the sensitivity computation for 10 loops (4 are non-admissible) using sensitivity recurrences. Rec: size of the recurrence system to compute the variables' sensitivities; RT: runtime in seconds; TO: timeout.

Benchmark	Sensitivity	Rec, RT	Sensitivity	Rec, RT
Non-Admissible (Fig. 3.1b)	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{u})$	9, 1.40	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{y}^2)$	9, 1.75
Non-Admissible-2	$\partial_{\mathrm{par}}\mathbb{E}(\mathrm{y})$	5, 6.56	$\partial_{\mathrm{par}}\mathbb{E}(\mathrm{xz})$	4, 3.67
Non-Admissible-3	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{total})$	6, 12.6	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{z1}^2)$	12, 56.5
Non-Admissible-4	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{z})$	4, 0.48	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{cnt}^2)$	3, 0.39
Bimodal	$\partial_{\mathrm{var}}\mathbb{E}(\mathrm{x})$	3, 0.28	$\partial_{\mathrm{var}}\mathbb{E}(\mathrm{x}^2)$	5, 0.42
Component-Health	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{obs})$	3, 0.74	$\partial_{\mathrm{p1}}\mathbb{E}(\mathrm{obs}^2)$	3, 0.73
Gambler's Ruin	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{money})$	7, 66.9	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{money}^2)$	TO, TO
Las-Vegas-Search	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{attempts})$	3, 0.81	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{attempts}^2)$	7, 13.3
Randomized-Response	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{p1})$	1, 0.30	$\partial_{\mathrm{p}}\mathbb{E}(\mathrm{p1}^2)$	3, 0.40
Vaccination (Fig. 3.1a)	$\partial_{\mathrm{vp}}\mathbb{E}(\mathrm{infected})$	3, 8.26	$\partial_{\mathrm{vp}}\mathbb{E}(\mathrm{infected}^2)$	3, 7.85

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
u,w,x,y,z = 0,1,2,3,4
while ⋆:
  z = z+p² {1/2} z+p
  y = y - 5⋆p⋆z
  w = 5⋆w + x²
  x = 5 + w+x
  u = x + p⋆z⋆y
end
```

(a) Non-Admissible (Fig. 3.1b)

```
x,y,z,var = 1,2,a,0
d1,d2 = 5,3
run = -1
while ⋆:
  run = 2⋆run + z²
  z = z+1
  d1,d2 = d1⋆d2+3, d1+z
  x = 3⋆x + d2 + par²⋆z + run⋆z
  y = 3⋆(x-y) + par²⋆run
end
```

(b) Non-Admissible-2

```
cnt,total = 0,0
x1,x2 = 1,2
y1,y2 = 0,3
z1,z2 = 1,5
while ⋆:
  cnt = cnt + 1
  x1 = x1² + q⋆x2
  x2 = y1 + cnt + q
  y1 = r⋆(y1-cnt) + y2⋆cnt
  y2 = r⋆y1 + 5
  z1 = cnt² - cnt + p⋆z1
  z2 = z1⋆3 - 5⋆(z2-p)
  total = x2 + y2 + z2
end
```

(c) Non-Admissible-3

```
y,x,z,cnt = 0,0,0,0
while ⋆:
  x = DiscreteUniform(1,5)
  if x < 3:
    inc = Bernoulli(p1)
    cnt = cnt + inc
  else:
    inc = Bernoulli(p2)
    cnt = cnt - inc
  end
  f = DiscreteUniform(0,10)
  y = y² + x ⋆ f
  z = cnt² - 3⋆y² + x³
end
```

(d) Non-Admissible-4

Figure 3.2: Four parameterized non-admissible loops used for our experiments (Table 3.2).

shows that *sensitivity recurrences* are capable of computing the sensitivities for admissible as well as non-admissible loops.

Experimental Summary. When comparing both approaches on admissible loops, the differentiation-based approach typically performs better, e.g., on the benchmarks "Gambler's Ruin" or "Vaccination". This is not surprising, as the main complexity in both approaches lies in solving the system of recurrences and when using sensitivity recurrences, the number of recurrences tends to be higher. However, the exact number of recurrences depends on the program structure, and as such, there are cases where the approach using sensitivity recurrences performs equally well, such as in the "Randomized-Response" benchmark. Nevertheless, for the class of loops characterized in Section 3.3.2, the differentiation-based approach fails, whereas sensitivity recurrences still deliver exact results in a fully automated manner.

Our experiments demonstrate that our novel techniques for sensitivity analysis can compute the sensitivities for a rich class of probabilistic loops with discrete and continuous

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

state spaces, drawing from probability distributions, and including polynomial arithmetic. Moreover, the technique based on our new notion of *sensitivity recurrences* can compute sensitivities for probabilistic loops for which closed-forms of the variables' moments do not exist.

3.5 Related Work

Sensitivity & Probabilistic Programs. Bayesian networks can be seen as special loop-free probabilistic programs. The sensitivity of Bayesian networks with discrete probability distribution was studied in [CD02, CD04]. The works of [BKS20b, SBK22] provide a framework to analyze properties (sensitivity among others) of *Prob-solvable Bayesian networks*. In contrast, our technique focuses on probabilistic loops with more complex control flow and supports continuous distributions. In recent years, techniques emerged to manually reason about sensitivities of probabilistic programs, such as program calculi [ABH+21], custom logics [BEG+18], or type systems [VVB22]. Although applicable to general probabilistic programs, these techniques require manual reasoning or user guidance, while our approach focuses on full automation.

A fully-automatic and exact sensitivity analyzer for probabilistic programs with statically bounded loops was proposed in [HWM18]. In comparison, our approach focuses on potentially unbounded loops. The authors of [WFC+20] introduce an automatable approach for expected sensitivity based on martingales. Their technique proves that a given program is Lipschitz-continuous for *some* Lipschitz constant. In contrast, our technique produces *exact* sensitivities for unbounded loops and we characterize a class of loops for which our technique is complete.

Recurrences in Program Analysis. Recurrence equations are a common tool in program analysis. The work of [RcK04, RK07] first introduced the idea of using linear recurrences and Gröbner basis computation to synthesize loop invariants. This line of work has been further generalized in [Kov08, HJK18b] to support more general recurrences. In [FK15, KCBR18] the authors apply linear recurrences to more complex programs and combine it with over-approximation techniques.

The work [BCKR20] combines recurrence techniques with template-based methods to analyze recursive procedures. Recurrence equations were first used for the analysis of probabilistic loops in [BKS19] to synthesize so-called *moment-based invariants*. This approach was further generalized by [MSBK22a]. Our technique of *sensitivity recurrences* is applicable to loops whose variables' moments do not satisfy linear recurrences. The recent work [ABK+22] studies the synthesis of invariants for such loops, but does not address sensitivity analysis.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

3.6 Conclusion

We establish a fully automatic and exact technique to compute the sensitivities of higher moments of program variables for probabilistic loops. Our method is applicable to probabilistic loops with potentially uncountable state spaces, complex control flow, polynomial assignments, and drawing from common probability distributions. For admissible loops, we utilize closed-forms of the variables' moments obtained through linear recurrences. Moreover, we propose the notion of *sensitivity recurrences* enabling the sensitivity analysis for probabilistic loops whose moments do not admit closed-forms. We characterize a class of loops for which we prove *sensitivity recurrences* to be sound and complete. Our experiments demonstrate the feasibility of our techniques on challenging benchmarks.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Strong Invariants Are Hard

This chapter is based on the following publication [MMK24]:

Julian Müllner, Marcel Moosbrugger, and Laura Kovács. Strong Invariants Are Hard: On the Hardness of Strongest Polynomial Invariants for (Probabilistic) Programs. Proc. ACM Program. Lang., (POPL), 2024.

4.1 Problem Statement

Loop invariants describe valid program properties that hold before and after every loop iteration. Intuitively, invariants provide correctness information that may prevent programmers from introducing errors while making changes to the loop. As such, invariants are fundamental to formalizing program semantics as well as to automate the formal analysis and verification of programs. While automatically synthesizing loop invariants is, in general, an uncomputable problem, when considering only single-path loops with linear updates (linear loops), the strongest polynomial invariant is in fact computable [Kar76, MS04a, Kov08]. The computability remains intact for linear loops with non-deterministic branching [HOPW18]. Yet, already for single-path loops with "only" polynomial updates, computing the strongest invariant has been an open challenge since 2004 [MS04b]. We bridge the gap between the computability result for linear loops and the uncomputability result for general loops by providing, to the best of our knowledge, the *first hardness result for computing the strongest polynomial invariant of single-path polynomial loops*.

Problem setting. Let us motivate our hardness results using the two loops in Figure 4.1, showcasing that very small changes in loop arithmetic may significantly increase the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{bmatrix} f & u & v & w \end{bmatrix} \leftarrow \begin{bmatrix} 1 & -1 & 2 & 0 \end{bmatrix}$$

while $\star$ **do**

$t \leftarrow 3t + 2u - 5w$

$u \leftarrow u + 3w$

$v \leftarrow 4u + 3v + w$

$w \leftarrow t + u + 2v$

end while

(a) An *affine* loop from [KLO+22].

$$\begin{bmatrix} x & y \end{bmatrix} \leftarrow \begin{bmatrix} x_0 & y_0 \end{bmatrix}$$

while $\star$ **do**

$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} x + y \cdot \Delta_t \\ y + (y \cdot (1 - x^2) - x) \cdot \Delta_t \end{bmatrix}$$

end while

(b) A *polynomial* loop, modelling the discrete-time Van der Pol oscillator [DDP17] for some constant sampling time Δ_t.

Figure 4.1: Two examples of deterministic programs.

difficulty of computing the strongest invariants. Figure 4.1a depicts an affine loop, that is, a loop where all updates are affine combinations of program variables. On the other hand, Figure 4.1b shows a polynomial loop whose updates are polynomials in program variables.

An affine (polynomial) invariant is a conjunction of affine (polynomial) equalities holding before and after every loop iteration. The computability of both the strongest affine and polynomial invariant has been studied extensively. For affine loops, the seminal paper [Kar76] shows that the strongest affine invariant is computable, whereas [Kov08] proves computability of the strongest polynomial invariant for single-path affine loops. Regarding polynomial programs, for example the one in Figure 4.1b, [MS04a] gives an algorithm to compute all polynomial invariants of *bounded degree*.

Based on these results, the strongest polynomial invariant of Figure 4.1a is thus computable. Yet, the more general problem of computing the strongest polynomial invariant for *polynomial loops* without any restriction on the degree remained an open challenge since 2004 [MS04b]. We address this challenge, which we coin as the SPInv problem and define below.

The SPInv Problem: Given a single-path loop with polynomial updates, compute the strongest polynomial invariant.

In Section 4.4, we prove that SPInv is *very hard*, essentially "defending" the state-of-the-art that so far failed to derive computational bounds on computing the strongest polynomial invariants of polynomial loops. The crux of our results is based on the Skolem problem, a prominent algebraic problem in the theory of linear recurrences [EvdPSW03, Tao08], which we briefly recall below and refer to Section 4.2.3 for details.

The Skolem Problem [EvdPSW03, Tao08]: Does a given linear recurrence sequence with constant coefficients have a zero?

The decidability of the Skolem problem has been open for almost a century, and its decidability is connected to far-fetching conjectures in number theory [BLN+22, LLN+22].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

In Section 4.4, we show that SPInv is at least as hard as the Skolem problem, providing thus a computational lower bound showcasing the hardness of SPInv.

To the best of our knowledge, our results from Section 4.4 are the first lower bounds for SPInv and provide an answer to the open challenge posed by [MS04a]. While [HOPW23] proved that the strongest polynomial invariant is uncomputable for multi-path polynomial programs, the computability of SPInv has been left open for future work. With our results proving that SPInv is Skolem-hard (Theorem 19), we show that the missing computability proof of SPInv is not surprising: solving SPInv is really hard.

Connecting invariant synthesis and reachability. A computational gap also exists in the realm of model-checking between affine and polynomial programs, similar to the computability of SPInv. Point-to-point reachability is arguably the simplest model-checking property; it asks whether a program can reach a given target state from a given initial state. For example, one may start the Van der Pol oscillator from Figure 4.1b in some initial configuration (x_0, y_0) and certify that it will eventually reach a certain target configuration (x_t, y_t). Reachability, and even more involved model-checking properties, are known to be decidable for affine loops [KLO+22]. However, the decidability or mere reachability of *polynomial loops* remains unknown without any existing non-trivial lower bounds. We refer to this reachability quest via the P2P problem.

The Point-To-Point Reachability Problem (P2P): Given a single-path loop with polynomial updates, is a given target state reachable starting from a given initial state?

In Section 4.3, we resolve the lack of computational results on reachability in polynomial loops. In particular, we show that P2P is Skolem-hard (Theorem 18) as well. To reduce Skolem to P2P, we construct a polynomial loop from a given linear recurrence sequence, such that the loop reaches the all-zero state if and only if the linear recurrence sequence has a zero. For our reduction, a linear recurrence sequence of order k is encoded as a loop with k variables. The crux of the reduction in Section 4.3 is that every variable is a shifted “non-linear variant”of the original sequence such that, once any variable becomes 0, it remains 0 forever. Then, the resulting loop reaches the all-zero state if and only if the original sequence has a zero. To the best of our knowledge, this yields the first non-trivial hardness result for P2P.

In Section 4.4, we further show that P2P and SPInv are connected in the sense that P2P reduces to SPInv. To reduce P2P to SPInv, we show how to decide whether a given loop reaches a given target state only using polynomial invariants. For the reduction, we add an auxiliary variable to the loop that becomes and remains 0 as soon as the original loop reaches the given target state. Intuitively, the auxiliary variable is *eventually* invariant if and only if the original loop reaches the target state. Utilizing techniques from computational algebraic geometry, we show how to decide whether the auxiliary variable is eventually invariant given the strongest polynomial invariant. Hence, we show that SPInv is at least as hard as P2P.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Therefore, our reduction chain Skolem $\leq$ P2P $\leq$ SPInv implies that the decidability of P2P and/or SPInv would immediately solve the Skolem problem a longstanding conjecture in number theory.

Beyond (non)deterministic loops and invariants. In addition to computational limits within standard, (non)deterministic programs, we further establish computational (hardness) bounds in probabilistic loops. Probabilistic programs model stochastic processes and encode uncertainty information in standard control flow, used for example in cryptography [BGB12], privacy [BKOB12], cyber-physical systems [KMS+22b], and machine learning [Gha15].

Because classical invariants, as in SPInv, do not account for probabilistic information, we provide a proper generalization of the strongest polynomial invariant for probabilistic loops in Section 4.5 (Lemma 20). With this generalization, we transfer the SPInv problem to the probabilistic setting. We hence consider the probabilistic version of SPInv as being the Prob-SPInv problem.

The Prob-SPInv Problem: Given a probabilistic loop with polynomial updates, compute the "probabilistic analog" of the strongest polynomial invariant.

In Section 4.5 we prove that Prob-SPInv inherits Skolem-hardness from its classical SPInv analog (Theorem 23). We also show that enriching the probabilistic program model with guards or branching statements renders the strongest polynomial (probabilistic) invariant uncomputable, even in the affine case (Theorems 21). We nevertheless provide a decision procedure when considering Prob-SPInv for a restricted class of polynomial loops: we define the class of *moment-computable* (polynomial) loops and show that Prob-SPInv is computable for such loops (Algorithm 2). Despite being restrictive, our moment-computable loops subsume affine loops with constant probabilistic choice. As such, Section 4.5 shows the limits of computability in deriving the strongest polynomial (probabilistic) invariants for probabilistic polynomial loops.

Our contributions. In conclusion, the main contributions are as follows:

- In Section 4.3, we provide a reduction from Skolem to point-to-point reachability for polynomial loops, proving that P2P is Skolem-hard (Theorem 18).
- Section 4.4 gives a reduction from P2P to the problem of computing the strongest polynomial invariant of polynomial loops, establishing the connection between P2P and SPInv. As such, we prove that SPInv is Skolem-hard (Theorem 19).
- In Section 4.5, we generalize the concept of strongest polynomial invariants to the probabilistic setting (Lemma 20). We show that Prob-SPInv is Skolem-hard (Theorem 23) and uncomputable for general polynomial probabilistic programs (Theorem 21), but it becomes computable for moment-computable polynomial probabilistic programs (Algorithm 2).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

4.2 Preliminaries

We write $\mathbb{N}$ for the natural numbers, $\mathbb{Q}$ for the rationals, $\mathbb{R}$ for the reals, and $\overline{\mathbb{Q}}$ for the algebraic numbers. We denote by $\mathbb{K}[x_1, \dots, x_k]$ the polynomial ring over k variables with coefficients in some field $\mathbb{K}$. Further, we use the symbol $\mathbb{P}$ for probability measures and $\mathbb{E}$ for the expected value operator.

4.2.1 Program Models

In accordance with [HOPW23, KV23], we consider *polynomial programs* $\mathcal{P} = (Q, E, q_0)$ over k variables, where Q is a set of locations, $q_0 \in Q$ is an initial location, and $E \subseteq Q \times \mathbb{Q}[x_1, \dots, x_k]^k \times Q$ is a set of transitions. The vector of *variable valuations* is denoted as $\vec{x} = (x_1, \dots, x_k)$, where each transition $(q, f, q') \in E$ maps a (program) configuration $(q, \vec{x})$ to some configuration $(q', f(\vec{x}))$. A transition $(q, f, q') \in E$ is *affine* if the function f is affine. In case all program transitions $(q, f, q') \in E$ are affine, we say that the polynomial program $\mathcal{P}$ is an *affine program*.

A *loop* is a program $\mathcal{L} = (Q, E, q_0)$ with exactly two locations $Q = \{q_0, q_1\}$, such that the initial state q_0 has exactly one outgoing transition to q_1 and all outgoing transitions of q_1 are self-loops, that is, $E = \{(q_0, f_1, q_1), (q_1, f_2, q_1), \dots, (q_1, f_n, q_1)\}$.

In a *guarded program*, each transition is additionally guarded by an equality/inequality predicate among variables of the state vector $\vec{x}$. If in some configuration the guard of an outgoing transition holds, we say that the transition is *enabled*, otherwise the transition is *disabled*.

(Non)Deterministic programs. If for any location $q \in Q$ in a program $\mathcal{P}$ there is exactly one outgoing transition (q, f, q'), then $\mathcal{P}$ is *deterministic*; otherwise $\mathcal{P}$ is *nondeterministic*. A deterministic guarded program may have multiple outgoing transitions from each location, but for any configuration, exactly one outgoing transition must be enabled. For a guarded nondeterministic program, we require that each configuration has at least one enabled outgoing transition. Deterministic, unguarded programs are called *single-path* programs.

To capture the concept of a loop invariant, we consider the collecting semantics of $\mathcal{P}$, associating each location $q \in Q$ with a set of vectors $\mathcal{S}_q$ that are reachable from the initial state $(q_0, \vec{0})$. More formally, the sets $\{\mathcal{S}_q \mid q \in Q\}$ are the least solution of the inclusion system

$$\mathcal{S}_{q_0} \supseteq \{\vec{0}\} \qquad \text{and} \qquad \mathcal{S}_{q'} \supseteq f(\mathcal{S}_q) \quad \text{for all } (q, f, q') \in E.$$

Definition 26 (Invariant). *A polynomial $p \in \overline{\mathbb{Q}}[x_1, \dots, x_k]$ is an* invariant *with respect to program location $q \in Q$, if for all reachable configurations $\vec{x} \in \mathcal{S}_q$ the polynomial vanishes, that is $p(\vec{x}) = 0$. Moreover, for a loop $\mathcal{L}$, the polynomial p is an* invariant of $\mathcal{L}$, *if p is an invariant with respect to the looping state q_1.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Probabilistic programs. In probabilistic programs, a probability pr is added to each program transition. That is, $E \subseteq Q \times \mathbb{Q}[x_1, \ldots, x_k]^k \times (0,1] \times Q$, where we require that each location has countably many outgoing transitions and that their probabilities pr sum up to 1. Under the intended semantics, a transition (q, f, pr, q') then maps a configuration $(q, \vec{x})$ to configuration $(q', f(\vec{x}))$ with probability pr. Again, for guarded probabilistic programs, we require that each configuration has at least one enabled outgoing transition and that the probabilities of the enabled transition sum up to 1.

For probabilistic programs $\mathcal{P}$, we consider moment invariants over higher-order statistical moments of the probability distributions induced by $\mathcal{P}$ (see Section 4.5). In this respect, it is necessary to count the number of executed transitions in the semantics of $\mathcal{P}$. Formally, the sets $\{\mathcal{S}_q^n \mid q \in Q, n \in \mathbb{N}_0\}$ are defined as

$$\mathcal{S}_{q_0}^0 := \{\vec{0}\} \qquad \text{and} \qquad \mathcal{S}_{q'}^{n+1} := f\left(\mathcal{S}_q^n\right) \quad \text{for all } (q, f, pr, q') \in E \text{ and } n \in \mathbb{N}_0.$$

In addition, the probability of a configuration $\vec{x}$ in location q after n iterations, in symbols $\mathbb{P}(\vec{x} \mid \mathcal{S}_q^n)$, can be defined inductively: (i) in the initial state, the configuration $\vec{0}$ after 0 executed transitions has probability 1; (ii) for any other state, the probability of reaching a specific configuration is defined by summing up the probabilities of all incoming paths. More formally, the probability $\mathbb{P}(\vec{x} \mid \mathcal{S}_q^n)$ is defined by

$$\mathbb{P}\left(\vec{x} \mid \mathcal{S}_q^0\right) := \begin{cases} 1 & q = q_0 \wedge \vec{x} = \vec{0} \\ 0 & \text{otherwise} \end{cases} \qquad \text{and}$$
$$\mathbb{P}\left(\vec{x} \mid \mathcal{S}_{q'}^{n+1}\right) := \sum_{(q,f,pr,q') \in E} \; \sum_{\vec{y} \in f^{-1}(\vec{x})} pr \cdot \mathbb{P}(\vec{y} \mid \mathcal{S}_q^n).$$

We then define the nth higher-order statistical moment of a monomial M in program variables as the expected value of M after n loop iterations. Namely,

$$\mathbb{E}[M_n] := \sum_{q \in Q} \sum_{\vec{x} \in \mathcal{S}_q^n} M(\vec{x}) \cdot \mathbb{P}(\vec{x} \mid \mathcal{S}_q^n), \tag{4.1}$$

where $M(\vec{x})$ evaluates the monomial M in a specific configuration $\vec{x}$.

Example 17. *The following loop encodes a symmetric 1-dimensional random walk starting at* 0*. In every step, the random walk moves left or right with probability* $1/2$*. The loop is given in code:*

$x \leftarrow 0$
while $\star$ ***do***
  $x \leftarrow x + 1$ $[1/2]$ $x - 1$
end while

Replacing the probabilistic choice in the loop body with non-deterministic choice, results in a non-deterministic program.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Universality of loops. In this chapter, we focus on polynomial loops. This is justified by the universality of loops [HOPW23, Section 4], as every polynomial program can be transformed into a polynomial loop that preserves the collecting semantics. Intuitively, this is done by merging all program states into the looping state and by introducing additional variables that keep track of which state is actually active while invalidating infeasible traces. It is then possible to recover the sets $\mathcal{S}_q^{(n)}$ of the original program from the sets $\mathcal{S}_q^{(n)}$ of the loop.

4.2.2 Computational Algebraic Geometry & Strongest Invariants

We study polynomial invariants $p(\vec{x})$ of polynomial programs; here, $p(\vec{x})$ are multivariate polynomials in program variables $\vec{x}$. We therefore recap necessary terminology from algebraic geometry [CLO97], to support us in reasoning whether $p(\vec{x}) = 0$ is a loop invariant. In the following $\mathbb{K}$ denotes a field, such as $\mathbb{R}$, $\mathbb{Q}$ or $\overline{\mathbb{Q}}$.

Definition 27 (Ideal)**.** *A subset of polynomials $I \subseteq \mathbb{K}[x_1, \ldots, x_k]$ is an* ideal *if (i) $0 \in I$; (ii) for all $x, y \in I$: $x + y \in I$; and (iii) for all $x \in I$ and $y \in \mathbb{K}[x_1, \ldots, x_k]$: $xy \in I$. For polynomials $p_1, \ldots, p_l \in \mathbb{K}[x_1, \ldots, x_k]$ we denote by $\langle p_1, \ldots, p_l \rangle$ the ideal generated by these polynomials, that is*

$$\langle p_1, \ldots, p_l \rangle := \left\{ \sum_{i=1}^{l} q_i p_i \;\middle|\; q_1, \ldots q_k \in \mathbb{K}[x_1, \ldots, x_k] \right\}$$

The set $I = \langle p_1, \ldots, p_l \rangle$ is an ideal, with the polynomials $p_1, \ldots, p_l$ being a basis *of I.*

Of particular importance is the set of all polynomial invariants of a program location. It is easy to check that this set forms an ideal.

Definition 28 (Invariant Ideal)**.** *Let $\mathcal{P}$ be a program with location q. The set $\mathcal{I}$ of all invariants with respect to the location q is called the* invariant ideal *of q. If $\mathcal{P}$ is a loop and $\mathcal{I}$ is the invariant ideal with respect to the looping state q_1, we call $\mathcal{I}$ the invariant ideal of the loop $\mathcal{P}$* [1]*.*

As the invariant ideal $\mathcal{I}$ of a loop $\mathcal{L}$ contains *all* polynomial invariants, a basis for $\mathcal{I}$ is the strongest polynomial invariant of $\mathcal{L}$. This is further justified by the following key result, establishing that every ideal has a basis.

Theorem 16 (Hilbert's Basis Theorem)**.** *Every ideal $I \subseteq \mathbb{K}[x_1, \ldots, x_k]$ has a basis. That is, $I = \langle p_1, \ldots, p_l \rangle$ for some $p_1, \ldots, p_l \in I$.*

[1] Computing bases for invariant ideals is equivalent to computing the *Zariski closure* of the loop: the Zariski closure is the smallest algebraic set containing the set of reachable states [HOPW18].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

While an ideal I may have infinitely many bases, the work of [Buc06] proved that every ideal I has a unique (reduced) *Gröbner basis*, where uniqueness is guaranteed modulo some *monomial order*. A monomial order $<$ is a total order on all monomials such that for all monomials m_1, m_2, m_3, if $m_1 < m_2$ then $m_1 m_3 < m_2 m_3$. For instance, assume our polynomial ring is $\mathbb{K}[x, y, z]$, that is, over three variables x, y, and z. A total order $z < y < x$ over variables can be extended to a lexicographic ordering on monomials, denoted also by $<$ for simplicity. In this case, for example, $xyz^3 < xy^2$ and $y^2 z < x$. For a given monomial order, one can consider the leading term of a polynomial p which we denote by $LT(p)$. For a set of polynomials S we write $LT(S)$ for the set of all leading terms of all polynomials. Continuing the example mentioned before, we have $LT(xyz^3 + xy^2) = xy^2$ and $LT(\{y + z, y^2 z + x\}) = \{y, x\}$.

Definition 29 (Gröbner Basis)**.** *Let $I \subseteq \mathbb{K}[x_1, \ldots, x_k]$ be an ideal and fix a monomial order. A basis $G = \{g_1, \ldots, g_k\}$ of I is a* Gröbner basis*, if $\langle LT(g_1), \ldots, LT(g_l)\rangle = \langle LT(I)\rangle$. Further, G is a* reduced Gröbner basis *if every g_i has leading coefficient 1 and for all $g, h \in G$ with $g \neq h$, no monomial in g is a multiple of $LT(h)$.*

Gröbner bases provide the workhorses to compute and implement algebraic operations over (infinite) ideals, including ideal intersections/unions, variable eliminations, and polynomial memberships. Given *any* basis for an ideal I, a unique reduced Gröbner basis with respect to any monomial ordering $<$ is computable using Buchberger's algorithm [Buc06]. A central property of Gröbner basis computation is that repeated division of a polynomial p by elements of a Gröbner basis results in a unique remainder, regardless of the order in which the divisions are performed. Hence, to decide if a polynomial p is an element of an ideal I, that is deciding polynomial membership, it suffices to divide p by a Gröbner basis of I and check if the remainder is 0. Moreover, eliminating a variable y from an ideal $I \subseteq \mathbb{K}[x, y]$ is performed by computing the Gröber basis of the elimination ideal $I \cap \mathbb{K}[x]$ only over x.

4.2.3 Recurrence Equations

Recurrence equations relate elements of a sequence to previous elements. There is a strong connection between recurrence equations and program loops: assignments in program loops relate values of program variables in the current iteration to the values in the next iteration. It is therefore handy to interpret a (polynomial) program loop as a recurrence. We briefly introduce linear and polynomial recurrence systems and refer to [KP11] for details.

We say that a sequence $u(n) : \mathbb{N}_0 \to \mathbb{Q}$ is a *linear recurrence sequence (LRS)* of order k, if there are coefficients $a_0, \ldots, a_{k-1} \in \mathbb{Q}$, where $a_0 \neq 0$ and for all $n \in \mathbb{N}_0$ we have

$$u(n + k) = a_{k-1} u(n + k-1) + \ldots + a_1 u(n + 1) + a_0 u(n) \tag{4.2}$$

The recurrence equation (4.2) is called a *linear recurrence equation*, with the coefficients $a_0, \ldots, a_{k-1}$ and the initial values $u(0), \ldots, u(k-1)$ uniquely specifying the sequence $u(n)$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Any LRS $u(n)$ of order k as defined via (4.2) can be specified by a system of k linear recurrence sequences $u_1(n), \ldots, u_k(n)$, such that each $u_i(n)$ is of order 1 and, for all $n \in \mathbb{N}_0$, we have $u(n) = u_1(n)$ and

$$\begin{aligned} u_1(n+1) &= \sum_{i=1}^{k} a_i^{(1)} u_i(n) = a_1^{(1)} u_1(n) + \ldots + a_k^{(1)} u_k(n) \\ &\vdots \\ u_k(n+1) &= \sum_{i=1}^{k} a_i^{(k)} u_i(n) = a_1^{(k)} u_1(n) + \ldots + a_k^{(k)} u_k(n) \end{aligned} \tag{4.3}$$

Again, the LRS $u(n)$ is uniquely defined by the coefficients $a_i^{(j)}$ and the initial values $u_1(0), \ldots, u_k(0)$.

Polynomial recursive sequences are natural generalizations of linear recurrence sequences and allow not only linear combinations of sequence elements but also polynomial combinations [CMP+20]. More formally, a sequence $u(n)$ is *polynomial recursive*, if there exists $k \in \mathbb{N}$ sequences $u^1(n), \ldots, u^k(n) : \mathbb{N}_0 \to \mathbb{Q}$ such that $u(n) = u_1(n)$ and there are polynomials $p_1, \ldots, p_k \in \mathbb{Q}[u_1, \ldots, u_k]$ such that, for all $n \in \mathbb{N}_0$, we have

$$\begin{aligned} u_1(n+1) &= p_1(u_1(n), \ldots, u_k(n)) \\ &\vdots \\ u_k(n+1) &= p_k(u_1(n), \ldots, u_k(n)) \end{aligned} \tag{4.4}$$

The sequence $u(n)$ from (4.4) is uniquely defined by the polynomials $p_1, \ldots, p_k$ and the initial values $u_1(0), \ldots, u_k(0)$. In contrast to linear recurrence sequences (4.2), polynomial recursive sequences (4.4) *cannot* be in general modeled using a single polynomial recurrence [CMP+20]. Systems of recurrences are widely used to model the evolution of dynamical systems in discrete time.

We conclude this section by recalling the Skolem problem [BLN+22, LLN+22] related to linear recurrence sequences, whose decidability is an open question since the 1930s. We formally revise the definition from Section 4.1 as:

The Skolem Problem [EvdPSW03, Tao08]: Given an LRS $u(n), n \in \mathbb{N}_0$, does there exist some $m \in \mathbb{N}_0$ such that $u(m) = 0$?

In the upcoming sections, we show that the Skolem problem is reducible to the decidability of three fundamental problems in programming languages, namely P2P, SPInv and Prob-SPInv from Section 4.1. As such, we prove that the Skolem problem gives us intrinsically hard computational lower bounds for P2P, SPInv, and Prob-SPInv.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

4.3 Hardness of Reachability in Polynomial Programs

We first address the computational limitations of reachability analysis within polynomial programs. It is decidable whether a loop with *affine* assignments reaches a target state from a given initial state [KL80]. Additionally, even problems generalizing reachability are known to be decidable for linear loops, such as various model-checking problems [KLO+22]. However, reachability for loops with polynomial assignments, or equivalently discrete-time polynomial dynamical systems, has been an open challenge. In this section, we address this reachability challenge via our P2P problem, showing that reachability in polynomial program loops is at least as hard as the Skolem problem (Theorem 18). To this end, let us revisit and formally define our P2P problem from Section 4.1, as follows.

The Point-To-Point Reachability Problem (P2P): Given a system of k polynomial recursive sequences $u_1(n), \ldots, u_k(n), n \in \mathbb{N}_0$ and a target vector $\vec{t} = (t_1, \ldots, t_k)$, does there exist some $m \in \mathbb{N}_0$ such that for all $1 \leq i \leq k$, it holds that $u_i(m) = t_i$?

To the best of our knowledge, nothing is known about the hardness of P2P for polynomial recursive sequences[2], and hence for loops with arbitrary polynomial assignments, apart from the trivial lower bounds provided by the linear/affine cases [KL80, KLO+22].

In the sequel, in Theorem 18 we prove that the P2P problem for polynomial recursive sequences is *at least as hard* as Skolem. Doing so, we show that solving Skolem can be solved by *reducing* it to inputs for P2P, written in symbols as Skolem $\leq$ P2P. We thus establish a computational lower bound for P2P in the sense that providing a decision procedure for P2P for polynomial recursive sequences would prove the decidability of the long-lasting open decision problem given by Skolem.

Our reduction for Skolem $\leq$ P2P. In a nutshell, we fix an arbitrary Skolem instance, that is, a linear recurrence sequence $u(n)$ of order k. We say that the instance $u(n)$ is *positive*, if there exists some $m \in \mathbb{N}_0$ such that $u(m) = 0$, otherwise we call the instance *negative*. Our reduction Skolem $\leq$ P2P constructs an instance of P2P that reaches the all-zero vector $\vec{0}$ if and only if the Skolem instance is positive. Hence, a decision procedure for P2P would directly lead to a decision procedure for Skolem.

Following (4.2), let our Skolem instance of order k to be the LRS $u(n) : \mathbb{N}_0 \to \mathbb{Q}$ specified by coefficients $a_0, \ldots a_{k-1} \in \mathbb{Q}$ such that $a_0 \neq 0$ and, for all $n \in \mathbb{N}_0$, we have

$$u(n+k) = a_{k-1} \cdot u(n+k-1) + \ldots + a_1 \cdot u(n+1) + a_0 \cdot u(n) = \sum_{i=0}^{k-1} a_i \cdot u(n+i). \tag{4.5}$$

From our Skolem instance (4.5), we construct a system of k polynomial recursive sequences $x_0, \ldots, x_{k-1}$, as given in (4.4). Namely, the initial sequence values are defined

[2] For linear systems, the Point-To-Point Reachability problem (P2P) is also referred to as the *Orbit problem* in [KL80].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

inductively as

$$\boxed{x_0(0) \coloneqq u(0)} \qquad \boxed{x_i(0) \coloneqq u(i) \cdot \prod_{\ell=0}^{i-1} x_\ell(0) \qquad (1 \leq i < k)}$$

With the initial values defined, the sequences $x_0, \ldots, x_{k-1}$ are uniquely defined via the following system of recurrence equations:

$$\begin{array}{l} \boxed{x_i(n+1) \coloneqq x_{i+1}(n) \qquad (1 \leq i < k-1)} \\ \boxed{x_{k-1}(n+1) \coloneqq \sum_{i=0}^{k-1} a_i \cdot x_i(n) \cdot \prod_{\ell=i}^{k-1} x_\ell(n)} \end{array} \tag{4.6}$$

Intuitively, the x_i sequences are “non-linear variants” of the Skolem instance $u(n)$ such that, once any x_i reaches 0, x_i remains 0 forever. The target vector for our P2P instance is therefore $\vec{t} = \vec{0}$.

Let us illustrate the main idea of our construction with the following example.

Example 18. *Assume our Skolem instance from* (4.5) *is given by the recurrence* $u(n+3) = 2u(n+2) - 2u(n+1) - 12u(n)$ *and initial values* $u(0) = 2, u(1) = -3, u(2) = 3$. *Following our reduction* (4.6)*, we construct a system of polynomial recursive sequences* $x_i(n)$ *given by the initial values*

$$\begin{aligned} x_0(0) &= u(0) = 2, \\ x_1(0) &= u(1)x_0(0) = -6, \\ x_2(0) &= u(2)x_0(0)x_1(0) = -36, \end{aligned}$$

and the following system of recurrences.

$$\begin{aligned} x_0(n+1) &= x_1(n) \\ x_1(n+1) &= x_2(n) \\ x_2(n+1) &= 2x_2(n)^2 - 2x_1(n)^2 x_2(n) - 12x_0(n)^2 x_1(n) x_2(n) \end{aligned}$$

The first few sequence elements of $u(n)$ *and* $x_0(n)$ *are shown in Figure 4.2 and illustrate the key property of our reduction:*

(i) $x_0(n)$ *is non-zero as long as* $u(n)$ *is non-zero, which we prove in Lemma 17;*

(ii) if there is an N *such that* $u(N) = 0$*, it holds that for all* $n \geq N : x_0(n) = 0$*. The other sequences* x_1 *and* x_2 *in the system are “shifted” variants of* x_0*. Hence, the constructed sequences all eventually reach the all-zero configuration and remain there. In Theorem 18, we prove that this is the case if and only if the Skolem instance* $u(n)$ *is positive.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

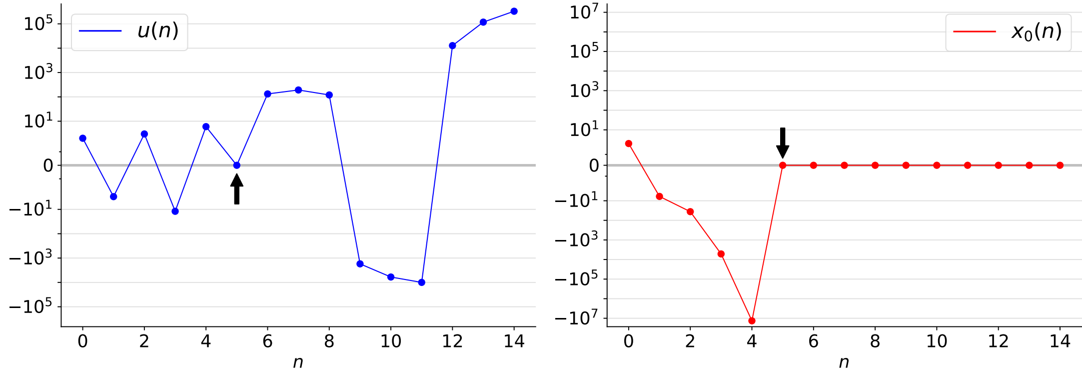


Figure 4.2: The first 15 sequence elements of $u(n)$ and $x_0(n)$ in Example 18.

Correctness of Skolem $\leq$ P2P. To prove the correctness of our reduction Skolem $\leq$ P2P and to assert the properties (i)-(ii) of Example 18 among $u(n)$ and $x_i(n)$, we introduce k auxiliary variables $s_0, \ldots, s_{k-1}$ defined as

$$s_i(0) := \begin{cases} 1 & (i = 0) \\ \prod_{\ell=0}^{i-1} x_\ell(0) & (1 \leq i < k) \end{cases} \qquad s_i(n+1) := \begin{cases} s_{i+1}(n) & (i \neq k-1) \\ s_{k-1}(n) \cdot x_{k-1}(n) & (i = k-1) \end{cases}$$

As illustrated in Example 18, the high-level idea of our reduction is that the x_i sequences are "non-linear variants" of the Skolem instance $u(n)$ such that, once any x_i reaches 0, x_i remains 0 forever. With the next lemma, we make the connections between the sequences $x_i(n)$ and $u(n)$ precise, using the auxiliary sequences $s_i(n)$. The central connection is $x_0(n) = s_0(n) \cdot u(n)$ and $s_0(n) = \prod_{l=0}^{n-1} x_0(l)$, which we utilize in the correctness proof in Theorem 18. The main idea behind the construction of the P2P instance is to ensure that this connection, and similar connections for the other sequences x_i and s_i, do hold. We formally prove these properties by induction.

Lemma 17. *For the system of polynomial recursive sequences in* (4.6)*, it holds that* $\forall n \geq 0$ *and* $0 \leq i < k$

$$x_i(n) = s_i(n) \cdot u(n+i), \text{ and} \tag{4.7}$$

$$s_i(n) = \prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{i-1} x_\ell(n). \tag{4.8}$$

Proof. We prove the two properties by well-founded induction on the lexicographic order (n, i), where $n \geq 0$ and $0 \leq i < k$. Here, $(n, i) \leq (n', i')$ if and only if $n < n'$ or $n = n' \wedge i < i'$. The order has the unique least element $(0, 0)$.

Base case: $n = 0$. If $i = 0$, then properties (4.7) and (4.8) hold by definition of $s_0(0) := 1 = \prod_{\ell=0}^{-1} x_0(\ell) \cdot \prod_{\ell=0}^{-1} x_\ell(0)$ and $x_0(0) := u(0) = s_0(0) \cdot u(0)$. Also, if $0 < i < k$,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

then properties (4.7) and (4.8) are trivially satisfied by the definition of the initial values: $s_i(0) \coloneqq \prod_{\ell=0}^{i-1} x_\ell(0)$ and $x_i(0) \coloneqq u(i) \cdot \prod_{\ell=0}^{i-1} x_\ell(0) = u(i) \cdot s_i(0)$.

Induction step – Case 1: $n > 0 \ \wedge\ 0 \leq i < k{-}1$. By the lexicographical ordering, it holds that $(n, i{+}1) < (n{+}1, i)$. Hence, we can assume that properties (4.7) and (4.8) hold for $(n, i{+}1)$. Thus, we have the induction hypothesis

$$x_{i+1}(n) = s_{i+1}(n) \cdot u(n+i+1), \text{ and} \tag{4.9}$$

$$s_{i+1}(n) = \prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{i} x_\ell(n). \tag{4.10}$$

To prove property (4.7) for $(n{+}1, i)$ means to show that

$$x_i(n+1) = s_i(n+1) \cdot u(n+i+1).$$

The sequences x_i and s_i are defined by $x_i(n{+}1) = x_{i+1}(n)$ and $s_i(n{+}1) = s_{i+1}(n)$ and hence property (4.7) follows from the induction hypothesis (4.9).

To prove property (4.8) for $(n{+}1, i)$ means to show that

$$s_i(n+1) = \prod_{\ell=0}^{n} x_0(\ell) \cdot \prod_{\ell=0}^{i-1} x_\ell(n+1).$$

We prove the equation by using the induction hypothesis (4.10), the definitions $x_i(n{+}1) = x_{i+1}(n)$ and $s_i(n{+}1) = s_{i+1}(n)$, and index manipulation:

$$\begin{aligned} s_i(n+1) = s_{i+1}(n) &= \prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{i} x_\ell(n) \\ &= \prod_{\ell=0}^{n-1} x_0(\ell) \cdot x_0(n) \cdot \prod_{\ell=0}^{i-1} x_{\ell+1}(n) \\ &= \prod_{\ell=0}^{n} x_0(\ell) \cdot \prod_{\ell=0}^{i-1} x_\ell(n+1) \end{aligned}$$

Induction step – Case 2: $n > 0$ and $i = k{-}1$. We show that property (4.7) holds for $(n{+}1, k{-}1)$ by proving it to be equivalent to the definition of $x_{k-1}(n{+}1)$. To do so, we first instantiate property (4.7) and replace both $s_{k-1}(n{+}1)$ and $u(n{+}k)$ by their defining recurrence:

$$\begin{aligned} x_{k-1}(n+1) &= s_{k-1}(n+1) \cdot u(n+k) \\ &= s_{k-1}(n) \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n+i) \right) \end{aligned}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Next, we rearrange and apply the induction hypothesis (4.8) for $(n, k-1)$ and (n, i) and obtain:

$$
\begin{aligned}
x_{k-1}(n+1) &= x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n+i) \cdot s_{k-1}(n) \right) \\
&= x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n+i) \cdot \underbrace{\prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{k-2} x_\ell(n)}_{s_{k-1}(n) \text{ by I.H. (4.8)}} \right) \\
&= x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n+i) \cdot \underbrace{\prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{i-1} x_\ell(n)}_{=s_i(n) \text{ by I.H. (4.8)}} \cdot \prod_{\ell=i}^{k-2} x_\ell(n) \right) \\
&= x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n+i) \cdot s_i(n) \cdot \prod_{\ell=i}^{k-2} x_\ell(n) \right) \\
&= \sum_{i=0}^{k-1} a_i \cdot u(n+i) \cdot s_i(n) \cdot \prod_{\ell=i}^{k-1} x_\ell(n)
\end{aligned}
$$

Now, we can apply the induction hypothesis (4.7) to replace $u(n+i) \cdot s_i(n)$ by $x_i(n)$ and arrive at the relation:

$$
x_{k-1}(n+1) = \sum_{i=0}^{k-1} a_i \cdot x_i(n) \cdot \prod_{\ell=i}^{k-1} x_\ell(n)
$$

However, this is exactly the defining recurrence equation from (4.6). Hence, property (4.8) necessarily holds for $(n, k-1)$.

To prove property (4.8) for $(n+1, k-1)$ we use the defining equation of $s_{k-1}(n+1)$ and the induction hypothesis for $(n, k-1)$:

$$
\begin{aligned}
s_{k-1}(n+1) = s_{k-1}(n) \cdot x_{k-1}(n) = x_{k-1}(n) \cdot \prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{k-2} x_\ell(n) = \prod_{\ell=0}^{n-1} x_0(\ell) \cdot \prod_{\ell=0}^{k-1} x_\ell(n) \\
= \prod_{\ell=0}^{n-1} x_0(\ell) \cdot x_0(n) \cdot \prod_{\ell=0}^{k-2} x_{\ell+1}(n) = \prod_{\ell=0}^{n} x_0(\ell) \cdot \prod_{\ell=0}^{k-2} x_\ell(n+1)
\end{aligned}
$$

As we have covered all possible cases, we conclude the proof. □

Lemma 17 establishes two central properties of our reduction. We now use these properties to show that P2P is at least as hard as Skolem.

Theorem 18 (Hardness of P2P)**.** *P2P is Skolem-hard. That is, Skolem $\leq$ P2P.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Proof. We show that our polynomial recursive system constructed in (4.6) reaches the all-zero vector from the initial value if and only if the original Skolem instance is positive.

($\Rightarrow$) : Assume the Skolem instance is positive, then there is some smallest $N \in \mathbb{N}_0$ such that $u(N) = 0$. Property (4.7) of Lemma 17 implies

$$x_0(N) = s_0(N) \cdot u(N) = 0.$$

Using this equation and property (4.8) of Lemma 17, we deduce that for all $n > N$, each $s_i(n)$ contains $x_0(N)$ as a factor and hence $s_i(n) = 0$. Additionally, as $x_i(n) = s_i(n) \cdot u(n+i)$ by property (4.7), we conclude that for all $n > N$ also $x_i(n) = 0$. Hence, the polynomial recursive system reaches the all-zero vector.

($\Leftarrow$) Assume that the Skolem instance is negative, meaning that the linear recurrence sequence $u(n)$ does not have a 0. In particular, $u(i) \neq 0$ for all $0 \leq i < k$. Therefore, by definition of the polynomial recursive system (4.6), $x_i(0) \neq 0$ for all $0 \leq i < k$. Towards a contradiction, assume that the polynomial recursive system still reaches the all-zero vector. Hence, there is a smallest $N \in \mathbb{N}_0$ such that $x_i(N) = 0$ for all $0 \leq i < k$. In particular, $x_0(N) = 0$. Moreover, x_0 is the last sequence to reach 0, because of the recurrence equation $x_i(n+1) = x_{i+1}(n)$ for $0 \leq i < k$. Therefore, N is also the smallest number such that $x_0(N) = 0$. By property (4.7) of Lemma 17, we have

$$x_0(N) = s_0(N) \cdot u(N) = 0.$$

However, $s_0(N)$ must be non-zero, because

$$s_0(N) = \prod_{\ell=0}^{N-1} x_0(\ell),$$

by property (4.8) of Lemma 17, and the fact that N is the smallest number such that $x_0(N) = 0$. Then we necessarily have $u(N) = 0$, yielding a contradiction. □

Theorem 18 shows that P2P for polynomial recursive sequences is at least as hard as the Skolem problem. Thus, reachability and model-checking of loops with polynomial assignments is Skolem-hard. A decision procedure establishing decidability for P2P would lead to a major breakthrough in number theory, as by Theorem 18 this would imply the decidability of the Skolem problem.

Remark 4. *In [HOPW23] the authors show that the the strongest polynomial invariant is uncomputable for polynomial programs* with nondeterminism. *The proof reduces from an undecidable problem to finding the strongest polynomial invariant for nondeterministic polynomial programs. A similarity between our reduction from this section and the reduction in [HOPW23] is the idea of projecting specific states to the zero vector. Nevertheless, the setting and reasons for using such a projection differ significantly between the two reductions. The reduction in [HOPW23] maps invalid program traces to the zero state to argue about the dimension of an algebraic set. In contrast, our reduction maps the single program trace to the zero state if and only if the original Skolem instance is positive.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

On polynomial recurrences. The original Skolem problem is defined for linear recurrence sequences. The extensive use of polynomial arithmetic in the reduction of Skolem to P2P might suggest that it is possible to generalize our reduction to reduce from the equivalent of the Skolem problem for polynomial recurrences. Such a generalization would undoubtably strengthen our hardness result. However, our reduction critically depends on the linearity of the Skolem instance. The resulting P2P instance $x_i(n)$ does not account for the polynomial degrees of the recurrence. Therefore, without further modifications, the reduction generates the same P2P instance for all original recurrences that only differ in their polynomial degrees. For instance, in Example 18, the reduction would produce the same P2P instance $x_i(n)$ for the linear recurrence $u(n{+}3) = 2u(n{+}2) - 2u(n{+}1) - 12u(n)$ and the polynomial recurrence $v(n{+}3) = 2v(n{+}2) - 2v(n{+}1) - 12v^2(n)$, assuming they have the same initial values. Since the instances $x_i(n)$ and sequences $s_i(n)$ are identical for both recurrences, but the two original sequences $u(n)$ and $v(n)$ are not equivalent, the central Lemma 17 cannot hold for polynomial recurrences.

Furthermore, the reduction cannot be easily adapted to account for polynomial recurrences. The core idea of the reduction is to construct a polynomial recursive sequence $x(n)$ from a given recursive sequence $u(n)$ such that

$$x(n) = u(n)s(n) \tag{4.11}$$

$$\text{and } s(n) = \prod_{i=0}^{n-1} u(i). \tag{4.12}$$

If $u(n)$ is fixed, for instance by the polynomial recurrence $u(n+1) = u^2(n) + u(n)$ with $u(0) = 1$, we can attempt to solve for a recurrence for $x(n)$:

$$\begin{aligned}
x(n+1) &= u(n+1)s(n+1) && \text{(by 4.11)}\\
&= s(n+1)\Big(u^2(n) + u(n)\Big) && \text{(by the recurrence for } u(n))\\
&= s(n+1)\left(\frac{x^2(n)}{s^2(n)} + \frac{x(n)}{s(n)}\right) && \text{(by 4.11)}\\
&= x^2(n)\frac{s(n+1)}{s^2(n)} + x(n)\frac{s(n+1)}{s(n)} && \text{(by reordering)}\\
&= x^2(n)\frac{x(n)}{s(n)} + x(n)x(n) && \text{(by 4.12)}
\end{aligned}$$

For sequences $u(n)$ defined by linear recurrences, the last expression in the derivation above is always free of $s(n)$. This results in a single polynomial recurrence defining $x(n)$ and corresponds to the P2P instance from our reduction. However, for polynomial recurrences, occurences of $s(n)$ cannot be eliminated in general, hindering the construction of a polynomial recurrence for $x(n)$. Hence, a potential reduction from the equivalent of the Skolem problem for polynomial recurrences to P2P would require a fundamentally different approach than the one presented in this chapter.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4 Hardness of Computing the Strongest Polynomial Invariant

This section goes beyond reachability analysis and focuses on inferring the strongest polynomial invariants of polynomial loops. As such, we turn our attention to solving the SPInv problem of Section 4.1, which is formally defined as given below.

The SPInv Problem: Given an unguarded, deterministic loop with polynomial updates, compute a basis of its polynomial invariant ideal.

We prove that finding the strongest polynomial invariant for deterministic loops with polynomial updates, that is, solving SPInv, is at least as hard as P2P (Theorem 19). Hence, P2P $\leq$ SPInv.

Then, by the Skolem $\leq$ P2P hardness result of Theorem 18, we conclude the Skolem-hardness of SPInv, that is Skolem $\leq$ P2P $\leq$ SPInv. To the best of our knowledge, our Theorem 18 together with Theorem 19 provide the first computational lower bound on SPInv, when focusing on loops with arbitrary polynomial updates (see Table 4.1).

Our reduction for P2P $\leq$ SPInv. We fix an arbitrary P2P instance of order k, given by a system of polynomial recursive sequences $u_1, \dots, u_k : \mathbb{N}_0 \to \mathbb{Q}$ and a target vector $\vec{t} = (t_1, \dots, t_k) \in \mathbb{Q}^k$. This P2P instance is positive if and only if there exists an $N \in \mathbb{N}_0$ such that $(u_1(N), \dots, u_k(N)) = \vec{t}$. For reducing P2P to SPInv, we construct the following deterministic loop with polynomial updates over k+2 variables:

$$\begin{aligned}
&\begin{bmatrix} f & g & x_1 & \dots & x_k \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & u_1(0) & \dots & u_k(0) \end{bmatrix} \\
&\textbf{while} \star \textbf{ do} \\
&\quad \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ f \\ g \end{bmatrix} \leftarrow \begin{bmatrix} p_1(x_1, \dots, x_k) \\ \vdots \\ p_k(x_1, \dots, x_k) \\ f \cdot ((x_1 - t_1)^2 + \dots + (x_k - t_k)^2) \\ g + 1 \end{bmatrix} \\
&\textbf{end while}
\end{aligned} \tag{4.13}$$

The polynomial recursive sequences $u_1, \dots, u_k$ are fully determined by their initial values and the polynomials $p_1, \dots, p_k \in \mathbb{Q}[u_1, \dots, u_k]$ defining the respective recurrence equations $u_i(n{+}1) = p_i(u_1(n), \dots, u_k(n))$. Hence, by the construction of the SPInv instance (4.13), every program variable x_i models the sequence u_i. As such, for any number of loop iterations $n \in \mathbb{N}_0$, we have $x_i(n) = u_i(n)$. Moreover, the variable g models the loop counter n, meaning $g(n) = n$ for all $n \in \mathbb{N}_0$. The motivation behind using the program variable f is that f becomes 0 as soon as all sequences u_i reach their target t_i; moreover, f remains 0 afterward. More precisely, for $n \in \mathbb{N}_0$, $f(n) = 0$ if and only if there is some $N \leq n$ such that $x_1(N) = t_1 \wedge \dots \wedge x_k(N) = t_k$. Hence, the sequence f

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

has a 0 value, and subsequently, all its values are 0, if and only if the original instance of P2P is positive.

Let us illustrate the main idea of our P2P $\leq$ SPInv reduction via the following example.

Example 19. *Consider the recursive sequences $x(n+1) = x(n)+2$ and $y(n+1) = y(n)+3$, with initial values $x(0) = y(0) = 0$. It is easy to see that the system $S = (x(n), y(n))$ reaches the target $\vec{t_1} = (4, 6)$ but does not reach the target $\vec{t_2} = (5, 7)$. Following are the two SPInv instances produced by our reduction for the P2P instances $(S, \vec{t_1})$ and $(S, \vec{t_2})$.*

SPInv ***instance for*** $(\mathbf{S}, \vec{\mathbf{t_1}})$:

$$\begin{bmatrix} f & g & x & y \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

while $\star$ ***do***

$$\begin{bmatrix} x \\ y \\ f \\ g \end{bmatrix} \leftarrow \begin{bmatrix} x+2 \\ y+3 \\ f \cdot \left((x-\mathbf{4})^2 + (y-\mathbf{6})^2\right) \\ g+1 \end{bmatrix}$$

end while

Invariant ideal: $\langle x-2g, y-3g, g(g-1)f \rangle$

SPInv ***instance for*** $(\mathbf{S}, \vec{\mathbf{t_2}})$:

$$\begin{bmatrix} f & g & x & y \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

while $\star$ ***do***

$$\begin{bmatrix} x \\ y \\ f \\ g \end{bmatrix} \leftarrow \begin{bmatrix} x+2 \\ y+3 \\ f \cdot \left((x-\mathbf{5})^2 + (y-\mathbf{7})^2\right) \\ g+1 \end{bmatrix}$$

end while

Invariant ideal: $\langle x-2g, y-3g \rangle$

The invariant ideals for both instances are given in terms of Gröbner bases with respect to the lexicographic order for the variable order $g < f < y < x$.

For the instance with the reachable target $\vec{t_1}$, we have $f(n) = 0$ for $n \geq 2$. Hence, $g(g-1)f$ is a polynomial invariant and must be in the invariant ideal of this SPInv instance; in fact, $g(g-1)f$ is not only in the invariant ideal but even a basis element for the Gröbner basis with the chosen order. However, $g(g-1)f$ is not in the ideal of the SPInv instance with the unreachable target $\vec{t_2}$. These two SPInv instances illustrate thus how a basis of the invariant ideal can be used to decide P2P.

While, for simplicity, our recursive sequences $x(n)$ and $y(n)$ are linear, our approach to reducing P2P to SPInv also applies to polynomial recursive sequences. In Theorem 19, we show that a polynomial such as $g(g-1)f$ is an element of the basis of the invariant ideal (with respect to a specific monomial order) if and only if the original P2P instance is positive.

Correctness of P2P $\leq$ SPInv. To show that it is decidable whether $f(n)$ has a 0 given a basis of the invariant ideal, we employ Gröbner bases and an argument introduced in [Kau05] for recursive sequences defined by rational functions, adjusted to our setting using recursive sequences defined by polynomials.

Theorem 19 (Hardness of SPInv)**.** *SPInv is at least as hard as P2P. That is, P2P $\leq$ SPInv.*

Proof. Assume we are given an oracle for SPInv, computing a basis B of the polynomial invariant ideal $\mathcal{I} = \langle B \rangle$ of our loop (4.13). We show that given such a basis B, it is

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

decidable whether $f(n)$ has a root, which is equivalent to the fixed P2P instance being positive.

Note that by the construction of the loop (4.13), if $f(N) = 0$ for some $N \in \mathbb{N}_0$, then $\forall n \geq N : f(n) = 0$. Moreover, such an N exists if and only if the P2P instance is positive. This is true if and only if there exists an $N \in \mathbb{N}_0$ such that the sequence

$$n \mapsto f(n) \cdot n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-N+1)$$

is 0 for all $n \in \mathbb{N}_0$. Consequently, the polynomial invariant ideal $\mathcal{I}$ contains a polynomial

$$P \coloneqq f \cdot g \cdot (g-1) \ldots \cdot (g-N+1) \tag{4.14}$$

for some $N \in \mathbb{N}_0$ only if the P2P instance (4.13) is positive. It is left to show that, given a basis B of $\mathcal{I}$, it is decidable whether $\mathcal{I}$ contains a polynomial (4.14). Using Buchberger's algorithm [Buc06], B can be transformed into a Gröbner basis with respect to any monomial order. We choose a total order among program variables such that $g < f < x_1, \ldots, x_k$. Without loss of generality, we assume that B is a Gröbner basis with respect to the lexicographic order extending the variable order.

In what follows, we argue that if a polynomial P as in (4.14) is an element of $\mathcal{I}$, then P must be an element of the basis B. As the leading term of P is $g^N \cdot f$, there must be some polynomial Q in B with a leading term that divides $g^N \cdot f$. By the choice of the lexicographic order, this polynomial must be of the form $Q = Q_1(g) \cdot f - Q_2(g)$, since if any other term would occur in Q, it would necessarily be in the leading term. As both $P \in \mathcal{I}$ and $Q \in \mathcal{I}$, it holds that

$$P \cdot Q_1 - g \cdot (g-1) \ldots \cdot (g-N+1) \cdot Q \in \mathcal{I}.$$

By expanding P and Q, we see that the above polynomial is equivalent to

$$Q_2 \cdot g \cdot (g-1) \ldots \cdot (g-N+1).$$

As this polynomial is in the ideal $\mathcal{I}$, it follows that for all $n \in \mathbb{N}_0$:

$$Q_2(n) \cdot n \cdot (n-1) \ldots \cdot (n-N+1) = 0.$$

However, this implies that $Q_2(n)$ has infinitely many zeros, a property that is unique to the zero polynomial. Therefore, we conclude that $Q_2 \equiv 0$. Hence, if the original P2P instance is positive, there necessarily exists a basis polynomial of the form $Q_1(g) \cdot f$.

We show that this basis polynomial $Q_1(g) \cdot f$ actually has the form (4.14): choose the basis polynomial of the form $Q_1(g) \cdot f$ such that Q_1 has minimal degree. Assume $Q_1(g)$ is not of the form $g \cdot (g-1) \ldots \cdot (g-N+1)$. Then, at least one factor $(g-m)$ is not a factor of Q_1, or equivalently $Q_1(m) \neq 0$. Then, necessarily $f(m) = 0$ and $g \cdot (g-1) \cdot \ldots \cdot (g-m+1) \cdot f$ must be in the ideal $\mathcal{I}$, contradicting the minimality of the degree of Q_1.

Therefore, we conclude that the P2P instance is positive if and only if the Gröbner basis contains a polynomial of the form (4.14). As the basis B is finite, this property can be checked by enumeration of the basis elements of B. Hence, given an oracle for SPInv, we can decide if the P2P instance is positive or negative. □

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Theorem 19 shows that SPInv is at least as hard as the P2P problem. Together with Theorem 18, we conclude that SPInv is Skolem-hard.

An improved direct reduction from Skolem to SPInv. Theorem 19 together with Theorem 18 yields the chain of reductions

$$\text{Skolem} \leq \text{P2P} \leq \text{SPInv}.$$

Within these reductions, a Skolem instance of order k yields a P2P instance with k sequences, which in turn reduces to a SPInv instance over $k{+}2$ variables.

We conclude this section by noting that, if the linear recurrence sequence of the Skolem-instance is an *integer sequence*, then a reduction directly from Skolem to SPInv can be established by using only $k{+}1$ variables. A slight modification of Skolem $\leq$ P2P reduction of Section 4.3 results in a reduction from Skolem instances of order k directly to SPInv instances with $k{+}1$ variables. Any system of polynomial recursive sequences can be encoded in a loop with polynomial updates. Hence, the instance produced by the Skolem $\leq$ P2P reduction can be interpreted as a loop. It is sufficient to modify the resulting loop in the following way:

$$\boxed{\begin{array}{l} x_{k-1} \leftarrow \sum_{i=0}^{k-1} a_i \cdot x_i \cdot \prod_{\ell=i}^{k-1} x_\ell \\ s_{k-1} \leftarrow x_{k-1} \cdot s_{k-1} \end{array}} \quad \rightarrow \quad \boxed{\begin{array}{l} x_{k-1} \leftarrow \sum_{i=0}^{k-1} a_i \cdot x_i \cdot \prod_{\ell=i}^{k-1} \mathbf{2} \cdot x_\ell \\ s_{k-1} \leftarrow \mathbf{2} \cdot x_{k-1} \cdot s_{k-1} \end{array}}$$

As in the reduction in Section 4.3, the equation $u_0(n) = \frac{x_0(n)}{s_0(n)}$ still holds and the resulting loop reaches the all-zero configuration if and only if the original Skolem-instance is positive (the integer sequence has a 0). Additionally, the resulting loop has infinitely many *different* configurations if and only if the Skolem instance is positive, as the additional factor in the updates forces a strict increase in $|s_{k-1}|$. Assuming a solution to SPInv for the constructed loop, that is a basis of the polynomial invariant ideal, it is decidable whether the number of reachable program locations (and its algebraic closure) is finite or not [CLO97]. Therefore, an oracle for SPInv implies the decidability of Skolem for *integer sequences*, while the chain of reductions Skolem $\leq$ P2P $\leq$ SPInv is also valid for rational sequences. For more details, we refer to [Mül23].

Summary of computability results in polynomial (non)determinstic loops. We conclude this section by overviewing our computability results in Table 4.1, focusing on the strongest polynomial invariants of (non)deterministic loops and in relation to the state-of-the-art.

4.5 Strongest Invariant for Probabilistic Loops

In this section, we finally go beyond (non-)deterministic programs and address computational challenges in probabilistic programming, in particular loops. Unlike the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 4.1: Summary of computability results for strongest invariants of *nonprobabilistic* polynomial loops, including our own results (Theorems 18 & 19). With '✓' we denote decidable problems, while '✗' denotes undecidable problems.

Program Model			Strongest Affine Invariant		Strongest Polynomial Invariant	
Det.	Unguarded	Affine	✓	[Kar76]	✓	[Kov08]
		Poly.	✓	[MS04a]	SKOLEM-hard	Theorems 18 & 19
	Guarded $(=, <)$	Affine	✗ (Halting Problem)			
		Poly.				
Nondet.	Unguarded	Affine	✓	[Kar76]	✓	[HOPW23]
		Poly.	✓	[MS04a]	✗	[HOPW23]
	Guarded $(=, <)$	Affine	✗ [MS04b]			
		Poly.				

programming models of Section 4.3–4.4, probabilistic loops follow different transitions with different probabilities (cf. Example 17).

Recall that the standard definition of an invariant I, as given in Definition 26, demands that I holds in *every reachable* configuration and location. As such, when using Definition 26 to define an invariant I of a probabilistic loop, the information provided by the probabilities of reaching a configuration within the respective loop is omitted in I. However, Definition 26 captures an invariant I of a probabilistic loop when every probabilistic loop transition is replaced by a nondeterministic transition.

Nevertheless, for incorporating probability-based information in loop invariants, Definition 26 needs to be revised to consider expected values and higher (statistical) moments describing the value distributions of probabilistic loop variables [Koz83, MM05]. For instance, the symmetric 1-dimensional random walk from Example 17 does not have any non-trivial polynomial invariants. However, considering expected values of program variables, $\mathbb{E}[x] = 0$ is an invariant property of Example 17. Therefore, in Definition 31 we introduce *polynomial moment invariants* to reason about value distributions of probabilistic loops. We do so by utilizing higher moments of the probability distributions induced by the value distributions of loop variables during the execution (Section 4.5.1). The notion of polynomial moment invariants is the main contribution of this section as it allows us to transfer specific (un)computability results for classical invariants to the probabilistic case. We prove that polynomial moment invariants generalize classical invariants (Lemma 20) and show that the strongest moment invariants up to moment order ℓ are computable for the class of so-called moment-computable polynomial loops (Section 4.5.2). In this respect, in Algorithm 2 we give a complete procedure for computing the strongest moment invariants of moment-computable polynomial loops. When considering *arbitrary* polynomial probabilistic loops, we prove that the strongest moment invariants are (i) not computable for guarded probabilistic loops (Section 4.5.3) and (ii) SKOLEM-hard to compute for unguarded probabilistic loops (Section 4.5.4).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.5.1 Polynomial Moment Invariants

Higher moments capture expected values of monomials over loop variables, for example, $\mathbb{E}[x^2]$ and $\mathbb{E}[xy]$ respectively yield the second-order moment of x and a second-order mixed moment. Such higher moments are necessary to characterize, and potentially recover, the value distribution of probabilistic loop variables, allowing us to reason about statistical properties, such as variance or skewness, over probabilistic value distributions.

When reasoning about moments of probabilistic program variables, note that in general neither $\mathbb{E}[x^\ell] = \mathbb{E}[x]^\ell$ nor $\mathbb{E}[xy] = \mathbb{E}[x]\mathbb{E}[y]$ hold, due to potential dependencies among the (random) loop variables x and y. Therefore, describing all polynomial invariants among all higher moments by finitely many polynomials is futile. A natural restriction is to consider polynomials over finitely many moments, which we do as follows.

Definition 30 (Moments of Bounded Degree)**.** *Let ℓ be a positive integer. Then the* set of program variable moments of order at most ℓ *is given by*

$$\mathbb{E}^{\leq \ell} := \{\mathbb{E}[x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_k^{\alpha_k}] \mid \alpha_1 + \ldots + \alpha_k \leq \ell\}.$$

Classical invariants are defined over a finite set of program variables. In the probabilistic setting, the elements of $\mathbb{E}^{\leq \ell}$ serve as the formal variables over which moment invariants are defined. As such, bounding the degrees of the moments is different from bounding the degrees of the invariants, which is a common technique for classical programs [MS04b]. Although the moments in $\mathbb{E}^{\leq \ell}$ are bounded, in this section, we study unbounded polynomial invariants involving these moments. While Definition 30 uses a bound ℓ to define the set of moments of bounded degree, our subsequent results apply to any *finite* set of moments of program variables.

Recall that Section 4.2.1 defines the semantics $\mathcal{S}_q^n$ of a probabilistic loop with respect to the location $q \in Q$ and the number of executed transitions $n \geq 0$. The set $\mathcal{S}_q^n$ in combination with the probability of each configuration allows us to define the moments of program variables after n transitions. Further, for a monomial M in program variables, we defined $\mathbb{E}[M_n]$ in (4.1) to be the expected value of M after n transitions. For example, $\mathbb{E}[x_n]$ denotes the expected value of the program variable x after n transitions. With this, we define the set of polynomial invariants among moments of program variables, as follows.

Definition 31 (Moment Invariant Ideal)**.** *Let $\mathbb{E}^{\leq \ell} = \{\mathbb{E}[M^{(1)}], \ldots, \mathbb{E}[M^{(k)}]\}$ be the set of program variable moments of order less than or equal to ℓ. The* moment invariant ideal $\mathbb{I}^{\leq \ell}$ *is defined as*

$$\mathbb{I}^{\leq \ell} = \Big\{ p\Big(\mathbb{E}[M^{(1)}], \ldots, \mathbb{E}[M^{(k)}]\Big) \in \overline{\mathbb{Q}}\Big[\mathbb{E}^{\leq \ell}\Big] \;\Big|\; \forall n \in \mathbb{N}_0 \colon p\Big(\mathbb{E}[M_n^{(1)}], \ldots, \mathbb{E}[M_n^{(k)}]\Big) = 0 \Big\}.$$

We refer to elements of $\mathbb{I}^{\leq \ell}$ as polynomial moment invariants.

Intuitively, the moment invariant ideal $\mathbb{I}^{\leq \ell}$ is the set of *all* polynomials in the moments $\mathbb{E}^{\leq \ell}$ that vanish after any number of executed transitions. For example, using Definition 31,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a polynomial $p(\mathbb{E}[x], \mathbb{E}[y])$ in the expected values of the variables x and y is a *polynomial moment invariant*, if $p(\mathbb{E}[x_n], \mathbb{E}[y_n]) = 0$ for all number of transitions $n \in \mathbb{N}_0$. Note that, although $\mathbb{E}^{\leq \ell}$ is a finite set, the moment invariant ideal $\mathbb{I}^{\leq \ell}$ is, in general, an infinite set.

Example 20. *Consider two asymmetric random walks x_n and y_n that both start at the origin. Both random walks increase or decrease with probability $1/2$, respectively. The random walk x_n either decreases by 2 or increases by 1, while y_n behaves conversely, which means y_n either decreases by 1 or increases by 2. Following is a probabilistic loop encoding this process together with the moment invariant ideal $\mathbb{I}^{\leq 2}$. The loop is given as program code. The intended meaning of the expression $e_1[pr]e_2$ is that it evaluates to e_1 with probability pr and to e_2 with probability $1{-}pr$.*

$$\begin{bmatrix} x & y \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 0 \end{bmatrix}$$
while $\star$ ***do***
$$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{bmatrix} x+2 \; [1/2] \; x-1 \\ y+1 \; [1/2] \; y-2 \end{bmatrix}$$
end while

Basis of the moment invariant ideal $\mathbb{I}^{\leq 2}$:

$$\mathbb{E}\left[x^2\right] - \mathbb{E}\left[y^2\right]$$
$$9 \cdot \mathbb{E}[x] - 2 \cdot \mathbb{E}[xy] - 2 \cdot \mathbb{E}\left[y^2\right]$$
$$\mathbb{E}[xy]^2 + 2 \cdot \mathbb{E}[xy] \cdot \mathbb{E}\left[y^2\right] + 81/4 \cdot \mathbb{E}[xy] + \mathbb{E}\left[y^2\right]^2$$
$$2 \cdot \mathbb{E}[xy] + 9 \cdot \mathbb{E}[y] + 2 \cdot \mathbb{E}\left[y^2\right]$$

This ideal $\mathbb{I}^{\leq 2}$ contains all algebraic relations that hold among $\mathbb{E}[x_n]$, $\mathbb{E}[y_n]$, $\mathbb{E}[x_n^2]$, $\mathbb{E}[y_n^2]$ and $\mathbb{E}[(xy)_n]$ after all number of iterations $n \in \mathbb{N}_0$. The ideal provides information about the stochastic process encoded by the loop. For instance, using the basis, it can be automatically checked that $\mathbb{E}[xy] - \mathbb{E}[x]\mathbb{E}[y]$ is an element of $\mathbb{I}^{\leq 2}$. Hence, $\mathbb{E}[xy] = \mathbb{E}[x]\mathbb{E}[y]$ is an invariant, witnessing x and y being uncorrelated.

Moment invariant ideals of Definition 31 generalize the notion of classical invariant ideals of Definition 28 for nonprobabilistic loops. For a program variable x of a nonprobabilistic loop, the expected value of x after n transitions is just the value of x after n iterations, that is $\mathbb{E}[x_n] = x_n$. Furthermore, $\mathbb{E}[x_n \cdot y_n] = x_n \cdot y_n$ for all program variables x and y. Hence, a moment invariant such as $\mathbb{E}[x^2]^3 - \mathbb{E}[y]\mathbb{E}[y^2]$ corresponds to the classical invariant $x^6 - y^3$. To formalize this observation, we introduce a function ψ mapping invariants involving moments to classical invariants.

Definition 32 (From Moment Invariants to Invariants)**.** *Let $\mathcal{P}$ be a program with variables $x_1, \dots, x_k$. We define the natural* ring homomorphism $\psi\colon \overline{\mathbb{Q}}[\mathbb{E}^{\leq \ell}] \to \overline{\mathbb{Q}}[x_1, \dots, x_k]$ *extending $\psi(\mathbb{E}[M]) := M$. That means, for all $p, q \in \overline{\mathbb{Q}}[\mathbb{E}^{\leq \ell}]$ and $c \in \overline{\mathbb{Q}}$ the function ψ satisfies the properties (i) $\psi(p+q) = \psi(p) + \psi(q)$; (ii) $\psi(p \cdot q) = \psi(p) \cdot \psi(q)$; and (iii) $\psi(c \cdot p) = c \cdot \psi(p)$.*

The function ψ maps polynomials over moments to polynomials over program variables, for example, $\psi(\mathbb{E}[x^2]^3 - \mathbb{E}[y]\mathbb{E}[y^2]) = \psi(\mathbb{E}[x^2])^3 - \psi(\mathbb{E}[y])\psi(\mathbb{E}[y^2]) = x^6 - y^3$. If p is a

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

polynomial moment invariant of a *probabilistic* program, $\psi(p)$ is in general *not* a classical invariant. However, for nonprobabilistic programs, $\psi(p)$ is necessarily an invariant for every moment invariant p, as we show in the next lemma.

Lemma 20 (Moment Invariant Ideal Generalization)**.** *Let $\mathcal{L}$ be a* nonprobabilistic *loop. Let $\mathcal{I}$ be the classical invariant ideal and $\mathbb{I}^{\leq \ell}$ the moment invariant ideal of order ℓ. Then, $\mathbb{I}^{\leq \ell}$ and $\mathcal{I}$ are identical under ψ, that is*

$$\psi\left(\mathbb{I}^{\leq \ell}\right) := \left\{\psi(p) \mid p \in \mathbb{I}^{\leq \ell}\right\} = \mathcal{I}.$$

Proof. We show that $\psi(\mathbb{I}^{\leq \ell}) \subseteq \mathcal{I}$. The reasoning for $\mathcal{I} \subseteq \psi(\mathbb{I}^{\leq \ell})$ is analogous.

Let $q \in \psi(\mathbb{I}^{\leq \ell})$. Then, there is a $p(\mathbb{E}[M^{(1)}], \dots, \mathbb{E}[M^{(m)}]) \in \mathbb{I}^{\leq \ell}$ for some monomials in program variables $M^{(i)}$ such that $\psi(p) = p(M^{(1)}, \dots, M^{(m)}) = q$. The polynomial p in moments of program variables is an invariant because it is an element of $\mathbb{I}^{\leq \ell}$. Moreover, because the loop $\mathcal{L}$ is nonprobabilistic, we have $\mathbb{E}[M_n] = M_n$ for all number of transitions $n \in \mathbb{N}_0$ and all monomials M in program variables [3]. Hence, $q = p(M^{(1)}, \dots, M^{(m)})$ necessarily is a classical invariant as in Definition 26 and therefore $q \in \mathcal{I}$. □

Lemma 20 hence proves that Definition 31 generalizes the notion of invariant ideals of nonprobabilistic loops.

4.5.2 Computability of Moment Invariant Ideals

We next consider a special class of probabilistic loops, called *moment-computable polynomial loops*. For such loops, we prove that the bases for moment invariant ideals $\mathbb{I}^{\leq \ell}$ are computable for any order ℓ. Moreover, in Algorithm 2 we give a decision procedure computing moment invariant ideals of moment-computable polynomial loops.

Let us recall the semantical notion of *moment-computable loops* [MSBK22a], which we adjusted to our setting of polynomial probabilistic loops.

Definition 33 (Moment-Computable Polynomial Loops)**.** *A polynomial probabilistic loop $\mathcal{L}$ is* moment-computable *if, for any monomial M in loop variables of $\mathcal{L}$, we have that $\mathbb{E}[M_n]$ exists and is computable as $\mathbb{E}[M_n] = f(n)$, where $f(n)$ is an exponential polynomial in n, describing sums of polynomials multiplied by exponential terms in n. That is, $f(n) = \sum_{i=0}^{k} p_i(n) \cdot \lambda^n$ where all $p_i \in \overline{\mathbb{Q}}[n]$ are polynomials and $\lambda \in \overline{\mathbb{Q}}$.*

As stated in [KP11], we note that any LRS (4.2) has an exponential polynomial as closed form. As proven in [MSBK22a], when considering loops with affine assignments, probabilistic choice with constant probabilities, and drawing from probability distributions with constant parameters and existing moments, all moments of program variables follow

[3] If the loop contains nondeterministic choice, this property holds with respect to every scheduler resolving nondeterminism. For readability and simplicity, we omit the treatment of schedulers and refer to [BKS20a] for details on schedulers.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 2 Computing moment invariant ideals

Input: A moment-computable polynomial loop $\mathcal{L}$ and an order $\ell \in \mathbb{N}$
Output: A basis B for the moment invariant ideal $\mathbb{I}^{\leq \ell}$
▷ *Closed forms of moments as exponential polynomials*
$C \leftarrow$ `compute_closed_forms`$(\mathcal{L}, \mathbb{E}^{\leq \ell})$
▷ *A basis for the ideal of all algebraic relations among sequences in C*
$B \leftarrow$ `compute_algebraic_relations`(C)
return B

linear recurrence sequences. Moreover, one may also consider polynomial (and not just affine) loop updates such that non-linear dependencies among variables are acyclic. If-statements can also be supported if the loop guards contain only program variables with a finite domain. Under such structural considerations, the resulting probabilistic loops are moment-computable loops [MSBK22a]: expected values $\mathbb{E}[M_n]$ for monomials M over loop variables are exponential polynomials in n. Furthermore, a basis for the polynomial relations among exponential polynomials is computable [KZ08]. We thus obtain a decision procedure computing the bases of moment invariant ideals of moment-computable polynomial loops, as given in Algorithm 2 and discussed next.

The procedure `compute_closed_form`$(\mathcal{L}, S)$ in Algorithm 2 takes as inputs a moment-computable polynomial loop $\mathcal{L}$ and a set S of moments of loop variables and computes exponential polynomial closed forms of the moments in S; here, we adjust results of [MSBK22a] to implement `compute_closed_form`$(\mathcal{L}, S)$. Moreover, in Algorithm 2, `compute_algebraic_relations`(C) denotes a procedure that takes a set C of exponential polynomial closed forms as input and computes a basis for all algebraic relations among them; the procedure, `compute_algebraic_relations`(C) is implemented using [KZ08]. Soundness of Algorithm 2 follows from the soundness arguments of [MSBK22a, KZ08]. We implemented Algorithm 2 in our tool called `Polar`[4], allowing us to automatically derive the strongest polynomial moment invariants of moment-computable polynomial loops.

Example 21. *Using Algorithm 2 for the probabilistic loop of Example 20, we compute a basis for the moment invariant ideal* $\mathbb{I}^{\leq 2}$ *in approximately* 0.4 *seconds and for* $\mathbb{I}^{\leq 3}$ *in roughly* 0.8 *seconds, on a machine with a* 2.6 GHz *Intel i7 processor and* 32 GB *of RAM.*

4.5.3 Hardness for Guarded Probabilistic Loops

As Algorithm 2 provides a decision procedure for moment-computable polynomial loops, a natural question is whether the moment invariant ideals remain computable if we relax

(C1) the restrictions on the guards,

(C2) the structural requirements on the polynomial assignments

[4] `https://github.com/probing-lab/polar`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

of moment-computable polynomial loops.

We first focus on **(C1)**, that is, lifting the restriction on guards and show that in this case a basis for the moment invariant ideal of any order becomes uncomputable (Theorem 21).

We recall the seminal result of [MS04b] proving that the strongest polynomial invariant for *nonprobabilistic* loops with affine updates, nondeterministic choice, and guarded transitions is uncomputable. Interestingly, nondeterministic choice can be replaced by uniform probabilistic choice, allowing us to also establish the uncomputability of the strongest polynomial moment invariants, which means a basis for the ideal $\mathbb{I}^{\leq \ell}$, for any order ℓ.

Theorem 21 (Uncomputability of Moment Invariant Ideal)**.** *For the class of guarded probabilistic loops with affine updates, a basis for the moment invariant ideal $\mathbb{I}^{\leq \ell}$ is uncomputable for any order ℓ.*

Proof. The proof is by reduction from Post's correspondence problem (PCP), which is undecidable [Pos46]. A PCP instance consists of a finite alphabet Σ and a finite set of tuples $\{(x_i, y_i) \mid 1 \leq i \leq N, x_i, y_i \in \Sigma^*\}$. A solution is a sequence of indices (i_k), $1 \leq k \leq K$ where $i_k \in \{1, \ldots, N\}$ and the concatenations of the substrings indexed by the sequence are identical, written in symbols as

$$x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_K} = y_{i_1} \cdot y_{i_2} \cdot \ldots \cdot y_{i_K}$$

Note that the tuple elements may be of different lengths. Moreover, any instance of the PCP over a finite alphabet Σ can be equivalently represented over the alphabet $\{0, 1\}$ by a binary encoding.

Now, given an instance of the (binary) PCP, we construct the guarded probabilistic loop with affine updates shown in Figure 4.3. We encode the binary strings as integers and denote a transition with probability pr, guard g and updates f as $[pr] : g : \vec{x} \leftarrow f(\vec{x})$.

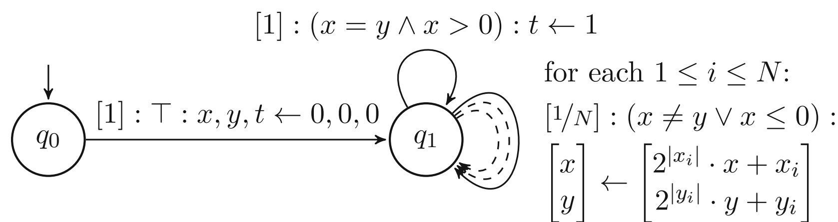


Figure 4.3: A guarded probabilistic loop with affine updates simulating the PCP.

The idea is to pick a pair of integer-encoded strings uniformly at random and append them to the string built so far. This is done by left-shifting the existing bits of the string (by multiplying by a power of 2) and adding the randomly selected string.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

If the PCP instance does not have a solution, we have $t = 0$ after every transition. Hence, $\mathbb{E}[t] = 0$ must be an invariant. Therefore, $\mathbb{E}[t]$ is necessarily an element of $\mathbb{I}^{\leq \ell}$ for any order ℓ.

If the PCP instance does have a solution $(i_k), 1 \leq k \leq K$, then after exactly $n = K + 2$ transitions it holds that $\mathbb{P}(x_n = y_n) \geq \left(\frac{1}{N}\right)^K$, as this is the probability of choosing the correct sequence uniformly at random. Because t is an indicator variable, $\mathbb{E}[t_n] = \mathbb{P}(t_n = 1) = \mathbb{P}(x_n = y_n) \geq \left(\frac{1}{N}\right)^K > 0$. Hence, $\mathbb{E}[t_n] \neq 0$ after n transitions and $\mathbb{E}[t]$ cannot be an element of $\mathbb{I}^{\leq \ell}$ for any order ℓ.

Consequently, for all orders ℓ, the PCP instance has a solution if and only if $\mathbb{E}[t]$ is an element of $\mathbb{I}^{\leq \ell}$. However, given a basis, checking for ideal membership is decidable (cf. Section 4.2.2). Hence, a basis for the moment invariant ideal $\mathbb{I}^{\leq \ell}$ must be uncomputable for any order ℓ. □

Note that the PCP reduction within the proof of Theorem 21 requires only affine updates and affine invariants. Therefore, allowing loop guards renders even the problem of finding the strongest affine invariant for a finite set of moments uncomputable for probabilistic loops with affine updates.

4.5.4 Hardness for Unguarded Polynomial Probabilistic Loops

In this section we address challenge **(C2)**, that is, study computational lower bounds for computing a basis of moment invariant ideals for probabilistic loops that lack guards and nondeterminism, but feature arbitrary polynomial updates. We show that addressing **(C2)** boils down to solving the Prob-SPInv problem of Section 4.1, which in turn we prove to be Skolem-hard (Theorem 23). As such, computing the moment invariant ideals of probabilistic loops with arbitrary polynomial updates as stated in **(C2)** is Skolem-hard.

We restrict our attention to moment invariant ideals of order 1. Intuitively, a basis for $\mathbb{I}^{\leq 1}$ is easier to compute than $\mathbb{I}^{\leq \ell}$ for $\ell > 1$. A formal justification in this respect is given by the following lemma.

Lemma 22 (Moment Invariant Ideal of Order 1)**.** *Given a basis for the moment invariant ideal $\mathbb{I}^{\leq \ell}$ for any order $\ell \in \mathbb{N}$, a basis for $\mathbb{I}^{\leq 1}$ is computable.*

Proof. The moment invariant ideal $\mathbb{I}^{\leq \ell}$ is an ideal in the polynomial ring with variables $\mathbb{E}^{\leq \ell}$. Moreover, $\mathbb{E}^{\leq 1} \subseteq \mathbb{E}^{\leq \ell}$. Hence, $\mathbb{I}^{\leq 1} = \mathbb{I}^{\leq \ell} \cap \overline{\mathbb{Q}}[\mathbb{E}^{\leq 1}]$, meaning $\mathbb{I}^{\leq 1}$ is an elimination ideal of $\mathbb{I}^{\leq \ell}$. Given a basis for a polynomial ideal, bases for elimination ideals are computable [CLO97]. □

Using Lemma 22, we translate challenge **(C2)** into the Prob-SPInv problem of Section 4.1, formally defined as follows.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The Prob-SPInv Problem: Given an unguarded, probabilistic loop with polynomial updates and without nondeterministic choice, compute a basis of the moment invariant ideal of order 1.

Recall that computing a basis for the classical invariant ideal for nonprobabilistic programs with arbitrary polynomial updates, that is, deciding SPInv, is Skolem-hard (Theorem 18 and Theorem 19). We next show that SPInv reduces to Prob-SPInv, thus implying Skolem-hardness of Prob-SPInv as a direct consequence of Lemma 20.

Theorem 23 (Hardness of Prob-SPInv)**.** *Prob-SPInv is at least as hard as SPInv, in symbols SPInv $\leq$ Prob-SPInv.*

Proof. Assume $\mathcal{L}$ is an instance of SPInv. That is, $\mathcal{L}$ is a deterministic loop with polynomial updates. Let $x_1, \ldots, x_k$ be the program variables and $\mathcal{I}$ the classical invariant ideal of $\mathcal{L}$. Note that $\mathcal{L}$ is also an instance of Prob-SPInv and assume B is a basis for the moment invariant ideal $\mathbb{I}^{\leq 1}$. From Lemma 20 we know that $\psi(\mathbb{I}^{\leq 1}) = \mathcal{I}$. For order 1, the function ψ is a ring isomorphism between the polynomial rings $\overline{\mathbb{Q}}[x_1, \ldots, x_k]$ and $\overline{\mathbb{Q}}[\mathbb{E}[x_1], \ldots, \mathbb{E}[x_k]]$. Hence, the set $\{\psi(b) \mid b \in B\}$ is a basis for $\mathcal{I}$. Therefore, given a basis for $\mathbb{I}^{\leq 1}$, a basis for $\mathcal{I}$ is computable. □

Theorem 23 shows that Prob-SPInv is at least as hard as the SPInv problem. Together with Theorem 18 and Theorem 19, we conclude the following chain of reductions:

$$\textsc{Skolem} \leq \text{P2P} \leq \text{SPInv} \leq \text{Prob-SPInv}$$

On attempting to prove uncomputability of Prob-SPInv– A remaining open challenge. While Theorem 23 asserts that Prob-SPInv is Skolem-hard, it could be that Prob-SPInv is uncomputable.

Recall that for proving the uncomputability of moment invariant ideals for guarded probabilistic programs in Theorem 21, we replaced nondeterministic choice with probabilistic choice. The "nondeterministic version" of Prob-SPInv refers to computing the strongest polynomial invariant for nondeterministic polynomial programs, which has been recently established as uncomputable [HOPW23]. Therefore, it is natural to consider transferring the uncomputability results of [HOPW23] to Prob-SPInv by replacing nondeterministic choice with probabilistic choice. However, such a generalization of [HOPW23] to the probabilistic setting poses considerable problems and ultimately fails to establish the potential uncomputability of Prob-SPInv, for the reasons discussed next.

The proof in [HOPW23] reduces the Boundedness problem for Reset Vector Addition System with State (VASS) to the problem of finding the strongest polynomial invariant for nondeterministic polynomial programs. A Reset VASS is a nondeterministic program where any transition may increment, decrement, or reset a vector of unbounded, non-negative variables. Importantly, a transition can *only be executed if no zero-valued variable is decremented*. The *Boundedness Problem for Reset VASS* asks, given a Reset VASS

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

and a specific program location, whether the set of reachable program configurations is finite. The Boundedness Problem for Reset VASS is undecidable [DFS98] and therefore instrumental in the reduction of [HOPW23].

Namely, in the reduction of [HOPW23] to prove uncomputability of the strongest polynomial invariant for nondeterministic polynomial programs, an arbitrary Reset VASS $\mathcal{V}$ with n variables $a_1, \ldots, a_n$ is simulated by a nondeterministic polynomial program $\mathcal{P}$ with $n+1$ variables $b_0, \ldots b_n$. Note that the programming model is purely nondeterministic, that is, without equality guards, since introducing guards would render the problem immediately undecidable [MS04b]. To avoid zero-testing the variables before executing a transition, the crucial point in the reduction of [HOPW23] is to map invalid traces to the vector $\vec{0}$ and faithfully simulate valid executions. By properties of the reduction, it holds that the configuration $(b_0, \ldots, b_n)$ is reachable in $\mathcal{P}$, if and only if there exists a corresponding configuration $1/b_0 \cdot (b_1, \ldots, b_n)$ in $\mathcal{V}$. Essential to the reduction of [HOPW23] is, that even though there may be multiple configurations in $\mathcal{P}$ for each configuration in $\mathcal{V}$, all these configurations are only scaled by the factor b_0 and hence collinear. By collinearity, the variety of the invariant ideal can be covered by a finite set of lines if and only if the set of reachable VASS configurations is finite. Testing this property is decidable, and hence finding the invariant ideal must be undecidable.

Transferring the reduction of [HOPW23] to the probabilistic setting of Prob-SPInv by replacing nondeterministic choice with probabilistic choice poses the following problem: in the nondeterministic setting, any path is independent of all other paths. However, this does not hold in the probabilistic setting of Prob-SPInv. The expected value operator $\mathbb{E}[x_n]$ aggregates all possible valuations of x in iteration n across all possible paths through the program. Specifically, the expected value is a linear combination of the possible configurations of $\mathcal{V}$, which is not necessarily limited to a collection of lines but may span a higher-dimensional subspace. This is the step where a reduction similar to [HOPW23] fails for Prob-SPInv.

Example 22. *Consider a Reset VASS $\mathcal{V}$ with variable x initialized to 0, initial state q_0, and additional state q_1. Assume a single transition from q_0 to q_1 incrementing x and two transitions from q_1 to q_1. One transition from q_1 to q_1 decrements x, whereas the other leaves x unchanged. In a Reset VASS, it is forbidden to decrement a zero-valued variable. Therefore, the set of reachable configurations in q_1 is $\{0, 1\}$ and hence finite. The reduction in [HOPW23] constructs from $\mathcal{V}$ a nondeterministic polynomial program $\mathcal{P}$ with two variables y and z. Similar to $\mathcal{V}$, the program $\mathcal{P}$ has two states $\hat{q}_0$ and $\hat{q}_1$, one transition from $\hat{q}_0$ to $\hat{q}_1$ and two transitions from $\hat{q}_1$ to itself. In contrast to $\mathcal{V}$, the transitions in $\mathcal{P}$ model polynomial assignments for the variables y and z. For more details on the reduction, we refer to [HOPW23]. Important are the reachable configurations of $\mathcal{P}$ depicted in the computation tree in Figure 4.4. For every reachable configuration $(y, z) \neq (0, 0)$ we have $z/y \in \{0, 1\}$ Hence, all reachable configurations lie on finitely many lines. Replacing nondeterministic choice in the state $\hat{q}_1$ by uniform probabilistic choice and considering expected values breaks this central property of the reduction. The sequence of expected values for y and z can be obtained by averaging the variable values for every*

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

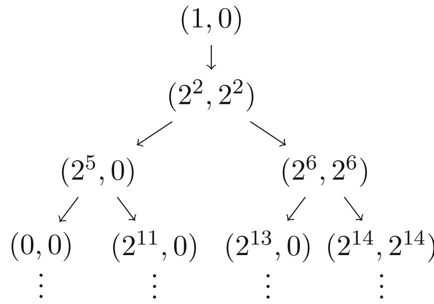


Figure 4.4: Computation tree of the program $\mathcal{P}$ from Example 22.

level in the computation tree in Figure 4.4 and is $(1,0),(4,4),(48,32),(4096,6656),\ldots$. *The ratios of the expected values after* $n \geq 1$ *transitions are can be calculated to be*

$$\frac{\mathbb{E}[z_n]}{\mathbb{E}[y_n]} = \frac{1}{\sum_{i=0}^{n-1} \frac{1}{2^{2^i - 1}}},$$

and hence the points $\{(\mathbb{E}[y_n], \mathbb{E}[z_n]) \mid n \in \mathbb{N}\}$ *cannot be covered with finitely many lines.*

It is however worth noting how well-suited the Boundedness Problem for Reset VASS is for proving the undecidability of problems for unguarded programs. A Reset VASS is not powerful enough to determine if a variable is zero, yet the Boundedness Problem is still undecidable. The vast majority of other undecidable problems that may be used in a reduction are formulated in terms of counter-machines, Turing machines, or other automata that rely on explicitly determining if a given variable is zero, hindering a straightforward simulation as unguarded programs. Therefore, we conjecture that any attempt towards proving (un)computability of Prob-SPInv would require a new methodology, unrelated to [HOPW23]. We leave this task as an open challenge for future work.

4.5.5 Summary of Computability Results for Probabilistic Polynomial Loop Invariants

We finally conclude this section by summarizing our computability results on the strongest polynomial (moment) invariants of probabilistic loops. We overview our results in Table 4.2.

4.6 Related Work

We discuss our work in relation to the state-of-the-art in computing strongest (probabilistic) invariants and analyzing point-to-point reachability.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 4.2: Our computability results for strongest polynomial (moment) invariants of polynomial *probabilistic* loops. The symbol '✓' denotes computable problems, '?' shows open problems, and '✗' marks uncomputable problems.

<table>
<tr><td colspan="3">Program Model</td><td colspan="2">Strongest Affine Invariant</td><td colspan="2">Strongest Polynomial Invariant</td></tr>
<tr><td rowspan="4">Prob.</td><td rowspan="2">Unguarded & Guarded (finite)</td><td>Affine</td><td>✓</td><td>Algorithm 2</td><td>✓</td><td>Algorithm 2</td></tr>
<tr><td>Poly.</td><td>?</td><td></td><td>SKOLEM-hard</td><td>Theorem 23</td></tr>
<tr><td rowspan="2">Guarded (=, <)</td><td>Affine</td><td colspan="4" rowspan="2">✗ Theorem 21</td></tr>
<tr><td>Poly.</td></tr>
</table>

Strongest Invariants. Algebraic invariants were first considered for unguarded deterministic programs with affine updates [Kar76]. Here, a basis for both the ideal of affine invariants and for the ideal of polynomial invariants is computable [Kar76, Kov08].

For unguarded deterministic programs with polynomial updates, all invariants of *bounded degree* are computable [MS04a], while the more general task of computing a basis for the ideal of *all* polynomial invariants, that is solving our SPINV problem, was stated as an open problem. In Section 4.4 we proved that SPINV is at least as hard as SKOLEM and P2P. Strengthening these results by proving computability for SPINV would result in a major breakthrough in number theory, as this would imply the decidability of the SKOLEM problem.

For guarded deterministic programs, the strongest affine invariant is uncomputable, even for programs with only affine updates. This is a direct consequence of the fact that this model is sufficient to encode Turing machines and allows us to encode the Halting problem [HU69]. Nevertheless, there exists a multitude of incomplete methods capable of extracting useful invariants even for non-linear programs, for example, based on abstract domains [KCBR18], over-approximation in combination with recurrences [FK15, KBCR19] or using consequence finding in tractable logical theories of non-linear arithmetic [KKZ23].

For nondeterministic programs with affine updates, a basis for the invariant ideal is computable [Kar76]. Furthermore, the set of invariants of bounded degree is computable for nondeterministic programs with polynomial updates, while bases for the ideal of all invariants are uncomputable [MS04a, HOPW23]. Additionally, even a single transition guarded by an equality or inequality predicate renders the problem uncomputable, already for affine updates [MS04b].

Point-To-Point Reachability. The Point-To-Point reachability problem formalized by our P2P problem appears in various areas dealing with discrete systems, such as dynamical systems, discrete mathematics, and program analysis. For linear dynamical systems, P2P is known as the *Orbit problem* [COW13], with a significant amount of work on analyzing and proving decidability of P2P for linear systems [KL80, COW13, COW15, BFJ+21]. In contrast, for polynomial systems, the P2P problem remained open regarding decidability or computational lower bounds. Existing techniques in this

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

respect resorted to approximate techniques [DDP17, DT12]. Contrarily to these works, in Section 4.3 we rigorously proved that P2P for polynomial systems is at least as hard as the Skolem problem. The P2P problem is essentially undecidable already for affine systems that additionally include nondeterministic choice [FGH13, KNP18].

Probabilistic Invariants. Invariants for probabilistic loops can be defined in various incomparable ways, depending on the context and use case. Dijkstra's weakest-precondition calculus for classical programs was generalized to the weakest-preexpectation (wp) calculus in the seminal works [Koz83, Koz85, MM05]. In the wp-calculus, the semantics of a loop can be described as the least fixed point of the *characteristic* function of the loop in the lattice of so-called *expectations* [KKM19]. Invariants are expectations that over- or under-approximate this fixed point and are called super- or sub-invariants, respectively. One line of research is to synthesize such invariants using templates and constraint-solving methods [GKM13, BCK+21, BCJ+23]. A calculus, analogous to the wp-calculus, has been introduced for expected runtime analysis [KKMO18] and amortized expected runtime analysis [BKK+23]. The work of [CNZ17] introduces the notion of *stochastic invariants*, that is, expressions that are violated with bounded probability. Other notions of probabilistic invariants involve martingale theory [BEFH16] or utilize bounds on the expected value of program variable expressions [CS14]. The techniques presented in [MSBK22a, BKS19] compute closed forms for moments of program variables parameterized by the loop counter.

The different notions of probabilistic invariants, in general, do not form ideals or are relative to some other expression. Furthermore, the existing procedures to compute invariants are heuristics-driven and hence incomplete. Contrarily to these, our *polynomial moment invariants* presented in Section 4.5 form ideals and relate all variables. Moreover, our Algorithm 2 computes a basis for *all* moment invariants and is complete for the class of moment-computable polynomial loops. Going beyond such loops, we showed that Prob-SPInv is Skolem-hard and/or uncomputable (Theorem 23 and Theorem 21).

4.7 Conclusion

We prove that computing the strongest polynomial invariant for single-path loops with polynomial assignments (SPInv) is at least as hard as the Skolem problem, a famous problem whose decidability has been open for almost a century. As such, we provide the first non-trivial lower bound for computing the strongest polynomial invariant for deterministic polynomial loops, a quest introduced in [MS04b]. As an intermediate result, we show that point-to-point reachability in deterministic polynomial loops (P2P), or equivalently in discrete-time polynomial dynamical systems, is Skolem-hard. Further, we devise a reduction from P2P to SPInv. We generalize the notion of invariant ideals from classical programs to the probabilistic setting, by introducing *moment invariant ideals* and addressing the Prob-SPInv problem. We show that the strongest polynomial moment invariant, and hence Prob-SPInv, is (i) computable for the class of *moment-*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

computable probabilistic loops, but becomes (ii) uncomputable for probabilistic loops with branching statements and (iii) Skolem-hard for polynomial probabilistic loops without branching statements. Going beyond Skolem-hardness of Prob-SPInv and SPInv are open challenges we aim to further study.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

(Un)Solvable Loop Analysis

This chapter is based on the following article [ABK+24]:

Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. (Un)Solvable Loop Analysis. Formal Methods Syst. Des., 2024. To appear.

The article is an extended version of the conference paper [ABK+22]:

Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving Invariant Generation for Unsolvable Loops. In Proc. of SAS, 2022.

5.1 Problem Statement

With substantial progress in computer-aided program analysis and automated reasoning, several techniques have emerged to automatically synthesise loop invariants, thus advancing a central challenge in the computer-aided verification of programs with loops. We address the problem of automatically generating loop invariants in the presence of polynomial arithmetic, which is still unsolved. This problem remains unsolved even when we restrict consideration to loops that are non-nested, without conditionals, and/or without exit conditions. Our work improves the state of the art under such and similar considerations.

Loop invariants, in the sequel simply *invariants*, are properties that hold before and after every iteration of a loop. Invariants therefore provide the key inductive arguments for automating the verification of programs; for example, proving correctness of deterministic loops [RcK04, Kov08, dOBP16, KCBR18, HJK18b] and correctness of hybrid and probabilistic loops [HFM+14, KKMO16, BKS19], or data flow analysis

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

and compiler optimisation [MS04a]. One challenging aspect in invariant synthesis is the derivation of *polynomial invariants* for arithmetic loops. Such invariants are defined by polynomial relations $P(x_1, \dots, x_k) = 0$ among the program variables $x_1, \dots, x_k$. While deriving polynomial invariants is, in general, undecidable [HOPW23], efficient invariant synthesis techniques emerge when considering restricted classes of polynomial arithmetic in so-called *solvable loops* [RcK04], such as loops with (blocks of) affine assignments [Kov08, dOBP16, HJK18b, KCBR18].

A common approach for constructing polynomial invariants, first pioneered in [EGLW72, KM76], is to (i) map a loop to a system of recurrence equations modelling the behaviour of program variables; (ii) derive closed-forms for program variables by solving the recurrences; and (iii) compute polynomial invariants by eliminating the loop counter n from the closed-forms. The polynomial invariants resulting from step (iii) over-approximate the fixed point of the loop. The central components in this setting follow. In step (i) a *recurrence operator* is employed to map loops to recurrences, which leads to closed-forms for the program variables as *exponential polynomials* in step (ii); that is, each program variable is written as a finite sum of the form $\sum_j P_j(n)\lambda_j^n$ parameterised by the nth loop iteration for polynomials P_j and algebraic numbers λ_j. From the theory of algebraic recurrences, this is the case if and only if the behaviour of each variable obeys a linear recurrence equation with constant coefficients [EvdPSW03, KP11]. Exploiting this result, the class of recurrence operators that can be linearised are called *solvable* [RcK04]. Intuitively, a loop with a recurrence operator is solvable only if the non-linear dependencies in the resulting system of polynomial recurrences are acyclic (see Section 5.3). However, even simple loops may fall outside the category of solvable operators, but still admit polynomial invariants and closed-forms for combinations of variables. This phenomenon is illustrated in Figure 5.1 whose recurrence operators are not solvable (i.e. unsolvable). In other works, the generation of polynomial invariants is usually limited to those variables that admit closed-forms. With our approach, specifically in step (iii), we can generate polynomial invariants from *combinations of program variables* that admit closed-forms (where individual variables may fail to do so). This analysis can lead to a tighter over-approximation of a loop's fixed point. In general, the main obstacle in the setting of unsolvable recurrence operators is the absence of "well-behaved" closed-forms for the resulting recurrences.

5.1.1 Related Work

To the best of our knowledge, the study of invariant synthesis from the viewpoint of recurrence operators is mostly limited to the setting of solvable operators (or minor generalisations thereof). In [RcK04, RK07] the authors introduce solvable loops and mappings to model loops with (blocks of) affine assignments and propose solutions for steps (i)–(iii) for this class of loops: all polynomial invariants are derived by first solving linear recurrence equations and then eliminating variables based on Gröbner basis computation. These results have further been generalised in [Kov08, HJK18b] to handle more generic recurrences; in particular, deriving arbitrary exponential polynomials as closed-forms

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
z ← 0
while ⋆ do
    z ← 1 − z
    x ← 2x + y^2 + z
    y ← 2y − y^2 + 2z
end while
```

Closed-form of $x + y$:
$x(n)+y(n) = 2^n(x(0)+y(0)+2) - (-1)^n/2 - 3/2$

(a) The program $\mathcal{P}_{\square}$.

```
x, y ← 1, 1
while ⋆ do
    w ← x + y
    x ← w^2
    y ← w^3
end while
```

Polynomial Invariant:
$y^2(n) - x^3(n) = 0$

(b) The program $\mathcal{P}_{\mathrm{SC}}$.

Figure 5.1: Two running examples with unsolvable recurrence operators. Nevertheless, $\mathcal{P}_{\square}$ admits a closed-form for combinations of variables and $\mathcal{P}_{\mathrm{SC}}$ admits a polynomial invariant. Herein we use $\star$ (rather than a loop guard or `true`) as loop termination is not our focus. For the avoidance of doubt: we consider standard mathematical arithmetic (e.g. mathematical integers) rather than machine floating-point and finite precision arithmetic.

of loop variables and allowing restricted multiplication among recursively updated loop variables. The authors of [FK15, KCBR18] generalise the setting: they consider more complex programs and devise abstract (wedge) domains to map the invariant generation problem to the problem of solving *C-finite recurrences*. (We give further details of this class of recurrences in Section 5.2). All the aforementioned approaches are mainly restricted to C-finite recurrences for which closed-forms always exist, thus yielding loop invariants. In [BKS19, BKS20b] the authors establish techniques to apply invariant synthesis techniques developed for deterministic loops to probabilistic programs. Instead of devising recurrences describing the precise value of variables in step (i), their approach produces C-finite recurrences describing (higher) moments of program variables, yielding moment-based invariants after step (iii).

Pushing the boundaries in analyzing unsolvable loops is addressed in [KCBR18, FHG20]. The approach of [KCBR18] extracts C-finite recurrences over linear combinations of loops variables from unsolvable loops. For example, the method presented in [KCBR18] can also synthesise the closed-forms identified by our approach for Figure 5.1a. However, unlike [KCBR18], our method is not limited to linear combinations (we can extract C-finite recurrences over *polynomial* relations in the loop variables). As such, the technique of [KCBR18] cannot synthesise the polynomial loop invariant in Figure 5.1b, whereas our technique can. A further related approach to our method is given in [FHG20], yet in the setting of loop termination. However, our method is not restricted to solvable loops that are triangular, but can handle mutual dependencies among (unsolvable) loop variables, as evidenced in Figure 5.1.

Related work in the literature introduces techniques from the theory of martingales in

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

order to synthesise invariants in the setting of probabilistic programs [CS13]. Therein, the programming model is represented by a class of loop programs where all updates are linear and the synthesised invariants are given by linear templates. By contrast, our method allows us to handle polynomial arithmetic; in particular, we automatically generate invariants given by monomials in the program variables. On the other hand, the approach of [CS13] can also synthesise supermartingales whereas our work is restricted to invariants defined by equalities.

5.1.2 Our Contributions

We consider the sister problems of invariant generation and solvable loop synthesis in the setting of unsolvable recurrence operators. We introduce the notions of *effective* and *defective* program variables where, figuratively speaking, the defective variables are those "responsible" for unsolvability. Our main contributions are summarised below.

1. Crucial for our synthesis technique is our novel characterisation of unsolvable recurrence operators in terms of defective variables (Theorem 27). Our approach complements existing techniques in loop analysis, by extending these methods to the setting of 'unsolvable' loops.

2. On the one hand, defective variables do not generally admit closed-forms. On the other hand, some polynomial combinations of such variables are well-behaved (see e.g., Figure 5.1). We show how to compute the set of defective variables in polynomial time (Algorithm 3).

3. We introduce a new technique to synthesise valid linear relations in defective monomials such that these relations admit closed-forms, from which polynomial loop invariants follow (Section 5.5).

4. Given an unsolvable loop, we introduce an algorithmic approach (Algorithm 4) that synthesises a solvable loop with the following property: every polynomial invariant of the solvable loop is also an invariant of the given unsolvable loop (Section 5.6).

5. We generalise our technique to the analysis of probabilistic program loops (Section 5.7) and showcase further applications of unsolvable operators in such programs (Section 5.8).

6. We provide a fully automated approach in the tool Polar[1]. For evaluating our method, we compiled an extensive lists of challenging loops from the literature, including applications of mathematical and physical modelling. Our experiments demonstrate the feasibility of invariant synthesis for 'unsolvable' loops and the applicability of our approach to deterministic loops, probabilistic models, and biological systems (Section 5.9).

[1] https://github.com/probing-lab/polar

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

5.1.3 Beyond Invariant Generation

We believe our approach can provide new solutions towards compiler optimisation challenges. *Scalar evolution*[2] is a technique to detect general induction variables. Scalar evolution and general induction variables are used for a multitude of compiler optimisations, for example inside the LLVM toolchain [LA04]. On a high-level, general induction variables are loop variables that satisfy linear recurrences. As we show, defective variables do not satisfy linear recurrences in general; hence, scalar evolution optimisations cannot be applied upon them. However, some linear combinations of defective monomials *do* satisfy linear recurrences, which opens avenues where we can apply scalar evolution techniques over such monomials. In particular, our method automatically computes polynomial combinations of some defective loop variables, which potentially enlarges the class of loops that, for example, LLVM can optimise.

5.1.4 Structure and Summary of Results

We briefly recall preliminary material in Section 5.2. Section 5.3 abstracts from concrete recurrence-based approaches to invariant synthesis via recurrence operators. Section 5.4 introduces effective and defective variables, presents Algorithm 3 that computes the set of defective program variables in polynomial time, and characterises unsolvable loops in terms of defective variables (Theorem 27). In Section 5.5 we present our new technique that synthesises linear relations in defective monomials that admit well-behaved closed-forms. In Section 5.6 we introduce Algorithm 4 to synthesise solvable loops. That is, given an unsolvable loop, Algorithm 4 outputs a solvable loop, if it exists such that each polynomial invariant of the solvable loop is also an invariant of the unsolvable loop. In Section 5.7 we detail the necessary changes to the algorithms in Sections 5.5 and 5.6 for probabilistic programs. We illustrate our approach with several case-studies in Section 5.8, and describe a fully-automated tool support of our method in Section 5.9. We also report on accompanying experimental evaluation in Sections 5.8–5.9, and conclude in Section 5.10.

5.2 Preliminaries

5.2.1 Notation

We write $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ to respectively denote the sets of natural, rational, and real numbers. We write $\overline{\mathbb{Q}}$, the algebraic closure of $\mathbb{Q}$, to denote the field of algebraic numbers. We write $\mathbb{R}[x_1, \dots, x_k]$ and $\overline{\mathbb{Q}}[x_1, \dots, x_k]$ for the polynomial rings of all polynomials $P(x_1, \dots, x_k)$ in k variables $x_1, \dots, x_k$ with coefficients in $\mathbb{R}$ and $\overline{\mathbb{Q}}$, respectively (with $k \in \mathbb{N}$ and $k \neq 0$). A *monomial* is a monic polynomial with a single term.

For a program $\mathcal{P}$, $\text{Vars}(\mathcal{P})$ denotes the set of program variables. We adopt the following syntax in our examples. Sequential assignments in while loops are listed on separate

[2]`https://llvm.org/docs/Passes.html`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

lines (as demonstrated in Figure 5.1). In programs where simultaneous assignments are performed, we employ vector notation (as demonstrated by the assignments to the variables x and y in program $\mathcal{P}_{\mathrm{MC}}$ in Example 25).

We refer to a directed graph with G, whose edge and vertex (node) sets are respectively denoted via $A(G)$ and $V(G)$. We endow each element of $A(G)$ with a label according to a labelling function $\mathcal{L}$. A *path* in G is a finite sequence of contiguous edges of G, whereas a *cycle* in G is a path whose initial and terminal vertices coincide. A graph that contains no cycles is *acyclic*. In a graph G, if there exists a path from vertex u to vertex v, then we say that v is *reachable* from vertex u and say that u is a *predecessor* of v.

5.2.2 C-finite recurrences

We recall relevant results on (algebraic) recurrences and refer to [EvdPSW03, KP11] for further details. A *sequence* in $\overline{\mathbb{Q}}$ is a function $u\colon \mathbb{N} \to \overline{\mathbb{Q}}$, shortly written also as $\langle u(n)\rangle_{n=0}^{\infty}$ or simply just $\langle u(n)\rangle_n$. A *recurrence* for a sequence $\langle u(n)\rangle_n$ is an equation $u(n{+}\ell) = \mathrm{Rec}(u(n{+}\ell{-}1), \dots, u(n{+}1), u(n), n)$, for some function $\mathrm{Rec}\colon \mathbb{R}^{\ell+1} \to \mathbb{R}$. The number $\ell \in \mathbb{N}$ is the *order* of the recurrence.

A special class of recurrences we consider are the *linear recurrences with constant coefficients*, in short *C-finite recurrences*. A C-finite recurrence for a sequence $\langle u(n)\rangle_n$ is an equation of the form

$$u(n{+}\ell) = a_{\ell-1}u(n{+}\ell{-}1) + a_{\ell-2}u(n{+}\ell{-}2) + \cdots + a_0u(n) \tag{5.1}$$

where $a_0, \dots, a_{\ell-1} \in \overline{\mathbb{Q}}$ are constants and $a_0 \neq 0$. A sequence $\langle u(n)\rangle_n$ satisfying a C-finite recurrence (5.1) is a *C-finite sequence* and is uniquely determined by its initial values $u_0 = u(0), \dots, u_{\ell-1} = u(\ell{-}1)$. The *characteristic polynomial* associated with the C-finite recurrence relation (5.1) is

$$x^{n+\ell} - a_{\ell-1}x^{n+\ell-1} - a_{\ell-2}x^{n+\ell-2} - \cdots - a_0x^n.$$

The terms of a C-finite sequence can be written in a closed-form as exponential polynomials, depending only on n and the initial values of the sequence. That is, if $\langle u(n)\rangle_n$ is determined by a C-finite recurrence (5.1), then $u(n) = \sum_{k=1}^{r} P_k(n)\lambda_k^n$ where $P_k(n) \in \overline{\mathbb{Q}}[n]$ and $\lambda_1, \dots, \lambda_r$ are the roots of the associated characteristic polynomial. Importantly, closed-forms of (systems of) C-finite sequences always exist and are computable [EvdPSW03, KP11].

5.2.3 Invariants

A loop invariant is a loop property that holds before and after each loop iteration [Hoa69]. We focus on *polynomial invariants*, the class of invariants given by Boolean combinations of polynomial equations among loop variables. There is a minor caveat to our characterisation of (polynomial) loop invariants. We assume that a (polynomial) invariant consists of a finite number of initial values together with a closed-form expression of a monomial

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

in the loop variables. Thus the closed-form of a loop invariant must eventually hold after a (computable) finite number of loop iterations. Let us illustrate this caveat with the following loop example.

Example 23.

$x, y, z \leftarrow 0, 1, 0$
while $\star$ ***do***
  $x \leftarrow 1$
  $y \leftarrow y + x$
  $z \leftarrow z + 1$
end while

The loop admits the polynomial invariant $y - z - 1 = 0$ *given by the initial values* $x(0) = 0$, $y(0) = 1$, $z(0) = 0$ *and the closed-forms* $x(n) = 1$, $y(n) = n + 1$, *and* $z(n) = n$. *For each* $n \geq 1$, *we denote by* $v(n)$ *the value of a loop variable* v *at loop iteration* n.

Herein, we synthesise invariants that satisfy inhomogeneous first-order recurrence relations and it is straightforward to show that each associated closed-form holds for $n \geq 1$.

5.2.4 Polynomial Invariants and Invariant Ideals

A polynomial *ideal* is a subset $I \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_k]$ with the following properties: I contains 0; I is closed under addition; and if $P \in \overline{\mathbb{Q}}[x_1, \dots, x_k]$ and $Q \in I$, then $PQ \in I$. For a set of polynomials $S \subseteq \overline{\mathbb{Q}}[x_1, \dots, x_k]$, one can define the *ideal generated by* S by

$$I(S) := \{s_1 q_1 + \cdots + s_\ell q_\ell \mid s_i \in S, q_i \in \overline{\mathbb{Q}}[x_1, \dots, x_k], \ell \in \mathbb{N}\}.$$

Let $\mathcal{P}$ be a program as before. For $x_j \in \mathrm{Vars}(\mathcal{P})$, let $\langle x_j(n) \rangle_n$ denote the sequence whose nth term is given by the value of x_j in the nth loop iteration. The set of polynomial invariants of $\mathcal{P}$ form an ideal, the *invariant ideal* of $\mathcal{P}$ [RK07]. If for each program variable x_j the sequence $\langle x_j(n) \rangle_n$ is C-finite, then a basis for the invariant ideal can be computed as follows. Let $f_j(n)$ be the exponential polynomial closed-form of variable x_j. The exponential terms $\lambda_1^n, \dots, \lambda_s^n$ in each of the $f_j(n)$ are replaced by fresh symbols, yielding the polynomials $g_j(n)$. Next, with techniques from [KZ08], the set R of all polynomial relations among $\lambda_1^n, \dots, \lambda_s^n$ (that hold for each $n \in \mathbb{N}$) is computed. Then we express the polynomial relations in terms of the fresh constants, so that we can interpret R as a set of polynomials. Thus

$$I(\{x_j - g_j(n) \mid 1 \leq i \leq k\} \cup R) \cap \overline{\mathbb{Q}}[x_1, \dots, x_k]$$

is precisely the invariant ideal of $\mathcal{P}$. Finally, we can compute a finite basis for the invariant ideal with techniques from Gröbner bases and elimination theory [KZ08].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.3 From Loops to Recurrences

5.3.1 Recurrence Operators

Modelling properties of loop variables by algebraic recurrences and solving the resulting recurrences is an established approach in program analysis. Multiple works [Kov08, FK15, KCBR18, HJK17, HJK18b] associate a loop variable x with a sequence $\langle x(n) \rangle_n$ whose nth term is given by the value of x in the nth loop iteration. These works are primarily concerned with the problem of representing such sequences via recurrence equations whose closed-forms can be computed automatically, as in the case of C-finite sequences. A closely connected question to this line of research focuses on identifying classes of loops that can be modelled by solvable recurrences, as advocated in [RcK04]. To this end, over-approximation methods for general loops are proposed in [FK15, KCBR18] such that solvable recurrences can be obtained from (over-approximated) loops.

In order to formalise the above and similar efforts in associating loop variables with recurrences, herein we introduce the concept of a *recurrence operator*, and the characterisation of both *solvable* and *unsolvable operators*. Intuitively, a recurrence operator maps program variables to recurrence equations describing some properties of the variables; for instance, the exact values at the nth loop iteration [RcK04, Kov08, FK15] or statistical moments in probabilistic loops [BKS19].

Definition 34 (Recurrence Operator)**.** *A recurrence operator $\mathcal{R}$ maps the program variables $\text{Vars}(\mathcal{P})$ to the polynomial ring $\mathbb{R}[\text{Vars}_n(\mathcal{P})]$. The set of equations*

$$\{x(n{+}1) = \mathcal{R}[x] \mid x \in \text{Vars}(\mathcal{P})\}$$

constitutes a polynomial first-order system of recurrences. We call $\mathcal{R}$ linear *if $\mathcal{R}[x]$ is linear for all $x \in \text{Vars}(\mathcal{P})$.*

One can extend the operator $\mathcal{R}$ to $\mathbb{R}[\text{Vars}(\mathcal{P})]$. Then, with a slight abuse of notation, for $P(x_1, \dots, x_j) \in \mathbb{R}[\text{Vars}(\mathcal{P})]$ we define $\mathcal{R}(P)$ by $P(\mathcal{R}[x_1], \dots, \mathcal{R}[x_j])$.

Example 24. *Consider the program $\mathcal{P}_{SC}$ in Figure 5.1b. One can employ a recurrence operator $\mathcal{R}$ in order to capture the values of the program variables in the nth iteration. For $v \in \text{Vars}(\mathcal{P}_{SC})$, $\mathcal{R}[v]$ is obtained by bottom-up substitution in the polynomial updates starting with v. As a result, we obtain the following system of recurrences:*

$$\begin{aligned} w(n{+}1) &= \mathcal{R}[w] = x(n) + y(n) \\ x(n{+}1) &= \mathcal{R}[x] = x(n)^2 + 2x(n)y(n) + y(n)^2 \\ y(n{+}1) &= \mathcal{R}[y] = x(n)^3 + 3x(n)^2y(n) + 3x(n)y(n)^2 + y(n)^3. \end{aligned}$$

Similarly, for the program $\mathcal{P}_\square$ of Figure 5.1a, we obtain the following system of recurrences:

$$\begin{aligned} z(n{+}1) &= \mathcal{R}[z] = 1 - z(n) \\ x(n{+}1) &= \mathcal{R}[x] = 2x(n) + y(n)^2 - z(n) + 1 \\ y(n{+}1) &= \mathcal{R}[y] = 2y(n) - y(n)^2 - 2z(n) + 2. \end{aligned}$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

5.3.2 Solvable Operators

Systems of linear recurrences with constant coefficients admit computable closed-form solutions as exponential polynomials [EvdPSW03, KP11]. This property holds for a larger class of recurrences with polynomial updates, which leads to the notion of *solvability* introduced in [RcK04]. We adjust the notion of solvability to our setting by using recurrence operators. In the following definition, we make a slight abuse of notation and order the program variables so that we can transform program variables by a matrix operator.

Definition 35 (Solvable Operators [RcK04, dOBP16])**.** *The recurrence operator* $\mathcal{R}$ *is* solvable *if there exists a partition of* Vars_n*; that is,* $\text{Vars}_n = W_1 \uplus \cdots \uplus W_k$ *such that for* $x(n) \in W_j$*,*

$$\mathcal{R}[x] = M_j \cdot W_j^\top + P_j(W_1, \ldots, W_{j-1})$$

for some matrices M_j *and polynomials* P_j*. A recurrence operator that is not solvable is said to be* unsolvable*.*

This definition captures the notion of solvability in [RcK04] (see discussion in [dOBP16]).

We conclude this section by emphasising the use of (solvable) recurrence operators beyond deterministic loops, in particular relating its use to probabilistic program loops. As evidenced in [BKS19], recurrence operators model statistical moments of program variables by essentially focusing on solvable recurrence operators extended with an expectation operator $\mathbb{E}(\,\cdot\,)$ to derive closed-forms of (higher) moments of program variables, as illustrated below.

Example 25. *Consider the probabilistic program* $\mathcal{P}_{MC}$ *of [SO19, CVS16] modelling a non-linear Markov chain, where* Bernoulli(p) *refers to a Bernoulli distribution with parameter* p*. Here the updates to the program variables* x *and* y *occur simultaneously.*

while $\star$ ***do***
  $s \leftarrow \text{Bernoulli}(1/2)$
  if $s = 0$ ***then***
    $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + xy \\ \frac{1}{3}x + \frac{2}{3}y + xy \end{pmatrix}$
  else
    $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + y + \frac{2}{3}xy \\ 2y + \frac{2}{3}xy \end{pmatrix}$
  end if
end while

We can construct recurrence equations, in terms of the expectation operator $\mathbb{E}(\,\cdot\,)$*, for*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

this program as follows:

$$\begin{aligned}
\mathbb{E}(s_{n+1}) &= \tfrac{1}{2} \\
\mathbb{E}(x_{n+1}) &= \mathbb{E}(x_n) + \tfrac{1}{2}\mathbb{E}(y_n) + \tfrac{5}{6}\mathbb{E}(x_n y_n) \\
\mathbb{E}(y_{n+1}) &= \tfrac{1}{6}\mathbb{E}(x_n) + \tfrac{4}{3}\mathbb{E}(y_n) + \tfrac{5}{6}\mathbb{E}(x_n y_n).
\end{aligned}$$

5.4 Defective Variables

To the best of our knowledge, existing approaches in loop analysis and invariant synthesis are restricted to solvable recurrence operators. In this section, we establish a new characterisation of unsolvable recurrence operators. Our characterisation pinpoints the program variables responsible for unsolvability, the *defective variables* (see Definition 38). Moreover, we provide a polynomial time algorithm to compute the set of defective variables (Algorithm 3), in order to exploit our new characterisation for synthesising invariants in the presence of unsolvable operators in Section 5.5.

For simplicity, we limit the discussion in this section to deterministic programs. We note however that the results presented herein can also be applied to probabilistic programs. The details of the necessary changes in this respect are given in Section 5.7.

In what follows, we write $\mathcal{M}_n(\mathcal{P})$ to denote the set of non-trivial *monomials in* $\text{Vars}(\mathcal{P})$ *evaluated at the nth loop iteration* so that

$$\mathcal{M}_n(\mathcal{P}) := \left\{ \textstyle\prod_{x \in \text{Vars}(\mathcal{P})} x^{\alpha_x}(n) \mid \exists x \in \text{Vars}(\mathcal{P}) \text{ with } \alpha_x \neq 0 \right\}.$$

We next introduce the notions of variable dependency and dependency graph, needed to further characterise defective variables.

Definition 36 (Variable Dependency)**.** *Let $\mathcal{P}$ be a loop with recurrence operator $\mathcal{R}$ and $x, y \in \text{Vars}(\mathcal{P})$. We say x* depends on *y if y appears in a monomial in $\mathcal{R}[x]$ with non-zero coefficient. Moreover, x* depends linearly on *y if all monomials with non-zero coefficients in $\mathcal{R}[x]$ containing y are linear. Analogously, x* depends non-linearly on *y if there is a non-linear monomial with non-zero coefficient in $\mathcal{R}[x]$ containing y.*

Furthermore, we consider the transitive closure for variable dependency. If z depends on y and y depends on x, then z depends on x and, if in addition, one of these two dependencies is non-linear, then z depends non-linearly on x. We otherwise say the dependency is linear.

For each program with polynomial updates, we further define a *dependency graph* with respect to a recurrence operator.

Definition 37 (Dependency Graph)**.** *Let $\mathcal{P}$ be a program with recurrence operator $\mathcal{R}$. The* dependency graph *of $\mathcal{P}$ with respect to $\mathcal{R}$ is the labelled directed graph $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

with vertex set Vars($\mathcal{P}$)*, edge set* $A := \{(x, y) \mid x, y \in \text{Vars}(\mathcal{P}) \wedge x$ *depends on* $y\}$*, and a function* $\mathcal{L}\colon A \to \{L, N\}$ *that assigns a unique* label *to each edge such that*

$$\mathcal{L}(x, y) = \begin{cases} L & \textit{if } x \textit{ depends } \text{linearly } \textit{on } y, \textit{ and} \\ N & \textit{if } x \textit{ depends } \text{non-linearly } \textit{on } y. \end{cases}$$

Given a program and a recurrence operator, its dependency graph can be constructed automatically with standard techniques. In our approach, we partition the variables Vars($\mathcal{P}$) of the program $\mathcal{P}$ into two sets: *effective*- and *defective variables*, denoted by $E(\mathcal{P})$ and $D(\mathcal{P})$ respectively. Our partition builds on the definition of the dependency graph of $\mathcal{P}$, as follows.

Definition 38 (Effective-Defective)**.** *A variable* $x \in \text{Vars}(\mathcal{P})$ *is* effective *if:*

1. *x appears in no directed cycle with at least one edge with an N label, and*
2. *x cannot reach a vertex of an aforementioned cycle (as in 1).*

A variable is defective *if it is not effective.*

Example 26. *From the recurrence equations of Example 24 for the program $\mathcal{P}_{SC}$ (see Figure 5.1b), one obtains the dependencies between the program variables of $\mathcal{P}_{SC}$: the program variable w depends linearly on both x and y, whilst x and y depend non-linearly on each other and on w. By definition, the partition into effective and defective variables is $E(\mathcal{P}_{SC}) = \emptyset$ and $D(\mathcal{P}_{SC}) = \{w, x, y\}$.*

Similarly, we can construct the dependency graph for the program $\mathcal{P}_{\square}$ from Figure 5.1a, as illustrated in Figure 5.2. We derive that $E(\mathcal{P}_{\square}) = \{z\}$ and $D(\mathcal{P}_{\square}) = \{x, y\}$.

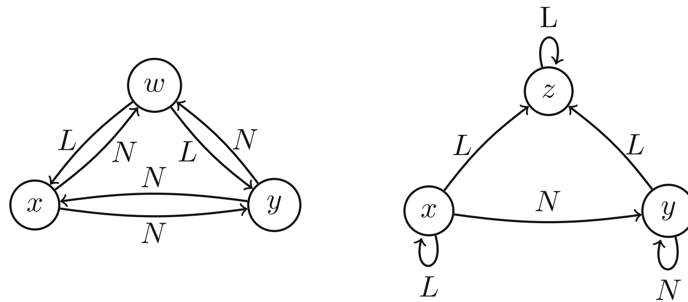


Figure 5.2: The dependency graphs for $\mathcal{P}_{\text{SC}}$ and $\mathcal{P}_{\square}$ from Figure 5.1.

We give the following straightforward corollary of Definition 38.

Corollary 23.1. *Given any effective variable $x \in E(\mathcal{P})$, the recurrence relation $\mathcal{R}[x]$ is a polynomial in effective variables.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The concept of effective, and, especially, defective variables allows us to establish a new characterisation of programs with unsolvable recurrence operators: *a recurrence operator is unsolvable if and only if there exists a defective variable* (as stated in Theorem 27 and automated in Algorithm 3). We formalise and prove this results via the following three lemmas.

Lemma 24. *Let $\mathcal{P}$ be a program with recurrence operator $\mathcal{R}$. If $D(\mathcal{P})$ is non-empty, so that there is at least one defective variable, then $\mathcal{R}$ is unsolvable.*

Proof. Let $x \in \text{Vars}(\mathcal{P})$ be a defective variable and $G = (\text{Vars}(\mathcal{P}), A, \mathcal{L})$ the dependency graph of $\mathcal{P}$ with respect to a recurrence operator $\mathcal{R}$. Following Definition 38, there exists a cycle C such that x is a vertex visited by or can reach said cycle and, in addition, there is an edge in C labelled by N.

Assume, for a contradiction, that $\mathcal{R}$ is solvable. Then there exists a partition $W_1, \ldots, W_k$ of $\text{Vars}_n(\mathcal{P})$ as described in Definition 35. Moreover, since C is a cycle, there exists $j \in \{1, \ldots, k\}$ such that each variable visited by C lies in W_j. Let $(w, y) \in C$ be an edge labelled with N. Since w depends on y non-linearly, and $\mathcal{R}[w] = M_j \cdot W_j^\top + P_j(W_1, \ldots, W_{j-1})$ (by Definition 35), it is clear that $y(n) \in W_\ell$ for some $\ell \neq j$. We also have that $y(n) \in W_j$ since C visits y. Thus we arrive at a contradiction as $W_1, \ldots, W_k$ is a partition of $\text{Vars}_n(\mathcal{P})$. Hence $\mathcal{R}$ is unsolvable. □

Given a program $\mathcal{P}$ whose variables are all effective, it is immediate that a pair of distinct mutually dependent variables are necessarily linearly dependent and, similarly, a self-dependent variable is necessarily linearly dependent on itself. Consider the following binary relation $\sim$ on program variables:

$$x \sim y \iff x = y \vee (x \text{ depends on } y \wedge y \text{ depends on } x).$$

Thus, any two mutually dependent variables are related by $\sim$. Under the assumption that all variables of a program $\mathcal{P}$ are effective, it is easily seen that $\sim$ defines an equivalence relation on $\text{Vars}(\mathcal{P})$. The partition of the equivalence classes Π of $\text{Vars}(\mathcal{P})$ under $\sim$ admits the following notion of dependence between equivalence classes: for $\pi, \hat{\pi} \in \Pi$ we say that π *depends on* $\hat{\pi}$ if there exist variables $x \in \pi$ and $y \in \hat{\pi}$ such that variable x depends on variable y.

Lemma 25. *Suppose that all variables of a program $\mathcal{P}$ are effective. Consider the graph $\mathcal{G}$ with vertex set given by the set of equivalence classes Π and edge set $A' := \{(\pi, \hat{\pi}) \mid (\pi \neq \hat{\pi}) \wedge (\pi$ depends on $\hat{\pi})\}$. Then $\mathcal{G}$ is acyclic.*

Proof. From the definition of $\mathcal{G}$, it is clear that the graph is directed and has no self-loops. Now assume, for a contradiction, that $\mathcal{G}$ contains a cycle. Since the relation $\sim$ is transitive, there exists a cycle C in $\mathcal{G}$ of length two. Moreover, the variables in a given equivalence class are mutually dependent. Thus the elements of the two classes in C are equivalent under the relation $\sim$, which contradicts the partition into distinct equivalence classes. Therefore the graph $\mathcal{G}$ is acyclic, as required. □

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Lemma 26. *Let $\mathcal{P}$ be a program with recurrence operator $\mathcal{R}$. If each of the program variables of $\mathcal{P}$ is effective then $\mathcal{R}$ is solvable.*

Proof. By Lemma 25, the associated graph $\mathcal{G} = (\Pi, A')$ on the equivalence classes of $\text{Vars}(\mathcal{P})$ is directed and acyclic. Thus there exists a topological ordering of $\Pi = \{\pi_1, \ldots, \pi_{|\Pi|}\}$ such that for every $(\pi_i, \pi_j) \in A'$ we have $i > j$. Thus if $x \in \pi_i$ then x does not depend on any variables in class π_j for $j > i$. Moreover, for each $\pi_i \in \Pi$, if $x, y \in \pi_i$ then x cannot depend on y non-linearly because every variable is effective (and all the variables in π_i are mutually dependent). Thus Π evaluated at loop iteration n partitions $\text{Vars}_n(\mathcal{P})$ and satisfies the criteria in Definition 35. We thus conclude that $\mathcal{R}$ is solvable. □

Together, Lemmas 24–26 yield a new characterisation of unsolvable operators.

Theorem 27 (Defective Characterisation)**.** *Let $\mathcal{P}$ be a program with recurrence operator $\mathcal{R}$, then $\mathcal{R}$ is unsolvable if and only if $D(\mathcal{P})$ is non-empty.*

In Algorithm 3 we provide a polynomial time algorithm that constructs both $E(\mathcal{P})$ and $D(\mathcal{P})$ given a program and a recurrence operator. We use the initialism "DFS" for the *depth-first search* procedure. Algorithm 3 terminates in polynomial time as both the construction of the dependency graph and depth-first search exhibit polynomial time complexity. The procedure searches for cycles in the dependency graph with at least one non-linear edge (labelled by N). All variables that reach such cycles are, by definition, defective.

Algorithm 3 Construct $E(\mathcal{P})$ and $D(\mathcal{P})$ from program $\mathcal{P}$ with operator $\mathcal{R}$.

```
1:  Construct the dependency graph G = (Vars(P), A, L) of P with respect to R.
2:  D(P) ← ∅
3:  for (x, y) ∈ A where L(x, y) = N do
4:      if x = y then
5:          predecessor ← ∅
6:          DFS(x, predecessor)
7:          D(P) ← D(P) ∪ predecessor
8:      end if
9:      if x ≠ y then
10:         predecessor ← ∅
11:         DFS(y, predecessor)
12:         if x ∈ predecessor then
13:             D(P) ← D(P) ∪ predecessor
14:         end if
15:     end if
16: end for
17: E(P) ← Vars(P) \ D(P)
```


Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

In what follows, we focus on programs with unsolvable recurrence operators, or equivalently by Theorem 27, the case where $\mathcal{D}(\mathcal{P}) \neq \emptyset$. The characterisation of unsolvable operators in terms of defective variables and our polynomial algorithm to construct the set of defective variables is the foundation for our approach synthesising invariants in the presence of unsolvable recurrence operators in Section 5.5.

Remark 5. *The recurrence operator $\mathcal{R}[x]$ for an effective variable x will admit a closed-form solution for every initial value x_0. For the avoidance of doubt, the same cannot be said for the recurrence operator of a defective variable. However, it is possible that a set of initial values will lead to a closed-form expression as a C-finite sequence: consider a loop with defective variable x and update $x \leftarrow x^2$ and initialisation $x_0 \leftarrow 0$ or $x_0 \leftarrow \pm 1$.*

5.5 Synthesising Invariants

In this section we propose a new technique to *synthesise invariants for programs with unsolvable recurrence operators*. The approach is based on our new characterisation of unsolvable operators in terms of defective monomials (Section 5.4).

For the remainder of this section we fix a program $\mathcal{P}$ with an unsolvable recurrence operator $\mathcal{R}$, or equivalently with $D(\mathcal{P}) \neq \emptyset$. We start by extending the notions of *effective* and *defective* from program variables to monomials of program variables. Let $\mathcal{E}$ be the set of *effective monomials* given by

$$\mathcal{E}(\mathcal{P}) = \left\{ \prod_{x \in E(\mathcal{P})} x^{\alpha_x} \mid \alpha_x \in \mathbb{N} \right\}.$$

The complement, the *defective monomials*, is given by $\mathcal{D}(\mathcal{P}) := \mathcal{M}(\mathcal{P}) \setminus \mathcal{E}(\mathcal{P})$. The difficulty with defective variables is that in general they do not admit closed-forms. However, linear combinations of defective monomials may allow for closed-forms as illustrated in previous examples. The main idea of our technique for invariant synthesis in the presence of defective variables is to find such polynomials. We fix a *candidate polynomial* called $S(n)$ based on an arbitrary degree $d \in \mathbb{N}$:

$$S(n) = \sum_{W \in \mathcal{D}_n(\mathcal{P}) \restriction_d} c_W W, \tag{5.2}$$

where the coefficients $c_W \in \mathbb{R}$ are unknown real constants. We use $\mathcal{D}_n(\mathcal{P}) \restriction_d$ to indicate the set of *defective monomials of degree at most d*.

Example 27. *For $\mathcal{P}_\square$ in Figure 5.1a we have $\mathcal{D}_n(\mathcal{P}_\square) \restriction_1 = \{x, y\}$, and $\mathcal{D}_n(\mathcal{P}_\square) \restriction_2 = \{x, y, x^2, y^2, xy, xz, yz\}$.*

On the one hand, all variables in $S(n)$ are defective; however, $S(n)$ may admit a closed-form. This occurs if $S(n)$ obeys a "well-behaved" recurrence equation; that is to say, an

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

inhomogeneous recurrence equation where the inhomogeneous component is given by a linear combination of effective monomials. In such instances the recurrence takes the form

$$S(n+1) = \kappa S(n) + \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M \tag{5.3}$$

where the coefficients c_M are unknown. Thus an intermediate step towards our goal of synthesising invariants is to determine whether there are constants $c_M, c_W, \kappa \in \mathbb{R}$ that satisfy the above equations. If such constants exist then we come to our final step: solving a first-order inhomogeneous recurrence relation. There are standard methods available to solve first-order inhomogeneous recurrences of the form $S(n+1) = \kappa S(n) + h(n)$, where $h(n)$ is the closed-form of $\sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M$, see e.g., [KP11]. We note $h(n)$ is computable and an exponential polynomial since it is determined by a linear sum of effective monomials. Thus $\langle S(n) \rangle_n$ is a C-finite sequence.

Remark 6. *Observe that the sum on the right-hand side of equation* (5.3) *is finite, since all but finitely many of the coefficients c_M are zero. Further, the coefficient c_M of monomial M is non-zero only if M appears in $\mathcal{R}[S]$.*

Going further, in equation (5.3) we express $S(n+1)$ in terms of a polynomial in $\mathrm{Vars}_n(\mathcal{P})$ with unknown coefficients c_M, c_W, and κ. An alternative expression for $S(n+1)$ in $\mathrm{Vars}_n(\mathcal{P})$ is given by the recurrence operator $S(n+1) = \mathcal{R}[S]$. Taken in combination, we arrive at the following formula

$$\mathcal{R}[S] - \kappa S(n) - \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M = 0,$$

yielding a polynomial in $\mathrm{Vars}_n(\mathcal{P})$. Thus all the coefficients in the above formula are necessarily zero as the polynomial is identically zero. Therefore *all* solutions to the unknowns c_M, c_W, and κ are computed by solving a (quadratic) system of equations. The main complexity of our invariant synthesis technique lies in solving the quadratic system. In the candidate polynomial, every monomial in defective variables (of degree at most d) is associated with a unique unknown coefficient. Hence, the size of the quadratic system can be polynomial in d and exponential in the number of defective variables.

Example 28. *Consider the following illustration of our invariant synthesis procedure. Recall program $\mathcal{P}_\square$ from Figure 5.1a:*

$z \leftarrow 0$
while $\star$ ***do***
  $z \leftarrow 1 - z$
  $x \leftarrow 2x + y^2 + z$
  $y \leftarrow 2y - y^2 + 2z$
end while

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

From Algorithm 3 we obtain $E(\mathcal{P}_\square) = \{z\}$ *and* $D(\mathcal{P}_\square) = \{x, y\}$*. Because* $D(\mathcal{P}_\square) \neq \emptyset$*, we deduce using Theorem 27 that the associated operator* $\mathcal{R}$ *is unsolvable. Consider the candidate* $S(n) = ax(n) + by(n)$ *with unknowns* $a, b \in \mathbb{R}$*. The recurrence for* $S(n)$ *given by* $\mathcal{R}$ *is*

$$\begin{aligned} S(n{+}1) = \mathcal{R}[S] = a\mathcal{R}[x] + b\mathcal{R}[y] \\ = a + 2b + 2ax(n) + 2by(n) - (a + 2b)z(n) + (a - b)y^2(n). \end{aligned}$$

We next express $S(n{+}1)$ *in terms of an inhomogeneous recurrence equation (cf. equation* (5.3)*). When we substitute for* $S(n)$*, we obtain*

$$S(n{+}1) = \kappa(ax(n) + by(n)) + (cz(n) + d)$$

where the coefficients in the inhomogeneous component are unknown. We then combine the preceding two equations (for brevity we suppress the loop counter n *in the program variables* x, y, z*) and derive*

$$(a + 2b - d) + (-a - c - 2b)z + (2a - \kappa a)x + (2b - \kappa b)y + (a - b)y^2 = 0.$$

Thus we have a polynomial in the program variables that is identically zero. Therefore, all the coefficients in the above equation are necessarily zero. We then solve the resulting system of quadratic equations, which leads to the non-trivial solution $a = b$*,* $\kappa = 2$*,* $d = 3a$*, and* $c = -3a$*. We substitute this solution back into the recurrence for* $\mathcal{R}[S]$ *and find*

$$S(n{+}1) = 2S(n) + 3a(1 - z(n)) = 2S(n) + 3a\frac{1 + (-1)^n}{2}.$$

Here, we have used the closed-form solution $z(n) = 1/2 - (-1)^n/2$ *of the effective variable* z*. We can compute the solution of this inhomogeneous first-order recurrence equation. In the case that* $a = 1$*, we have* $S(n) = 2^n(S(0) + 2) - (-1)^n/2 - 3/2$*. Therefore, the following identity holds for each* $n \in \mathbb{N}$*:*

$$x(n) + y(n) = 2^n(x(0) + y(0) + 2) - (-1)^n/2 - 3/2$$

and so we have synthesised the closed-form of $x + y$ *for program* $\mathcal{P}_\square$ *of Figure 5.1a.*

5.5.1 Solution Space of Invariants for Unsolvable Operators

Given a program and a recurrence operator, our invariant synthesis technique is relative-complete with respect to the degree d of the candidate $S(n)$. This means, for a fixed degree $d \in \mathbb{N}$, our approach is in theory able to compute *all* polynomials of defective variables with maximum degree d that satisfy a "well-behaved" recurrence; that is, a first-order recurrence equation of the form (5.3). This holds because of our reduction of the problem to a system of quadratic equations for which all solutions are computable. It is not guaranteed that a solution does exist. In that case, our technique can rule out the existence of well-behaved polynomials of defective variables of degree at most d if the resulting system has no (non-trivial) solutions.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Example 29. *The following loop models the* logistic map *[May76] which is well-known for its chaotic behaviour.*

while $\star$ ***do***
$x \leftarrow rx(1-x)$
end while

The single variable x is defective due to its non-linear self-dependency. For most values of r and initial values of x the logistic map does not admit an analytical solution and well-behaved polynomials in x do not exist. [Mar20]. Hence, our invariant synthesis technique provides no solution for candidates of fixed degrees.

Let $\mathcal{P}$ be a program with program variables $\text{Vars}(\mathcal{P}) = \{x_1, \ldots, x_k\}$. The set of polynomials P with $P(x_1(n), \ldots, x_k(n))=0$ for all $n \in \mathbb{N}$ form an ideal, the *invariant ideal* of $\mathcal{P}$. The requirement of closed-forms is the main obstacle for computing a basis for the invariant ideal in the presence of defective variables. Our work introduces a method that includes defective variables in the computation of invariant ideals, via the following steps of deriving the *polynomial invariant ideal of an unsolvable loop:*

- For every effective variable x_i, let $f_i(n)$ be its closed-form and assume $h(n)$ is the closed-form for some candidate S given by a polynomial in defective variables.
- Let $\lambda_1^n, \ldots, \lambda_s^n$ be the exponential terms in all $f_i(n)$ and $h(n)$. Replace the exponential terms in all $f_i(n)$ as well as $h(n)$ by fresh constants to construct the polynomials $g_i(n)$ and $l(n)$ respectively.
- Next, construct the set R of polynomial relations among all exponential terms, as explained in Section 5.2. Then, the ideal

  $$I(\{x_i - g_i(n) \mid x_i \in E(\mathcal{P})\} \cup \{S - l(n)\} \cup R) \cap \overline{\mathbb{Q}}[x_1, \ldots, x_k]$$

  contains precisely all polynomial relations among program variables implied by the equations $\{x_i = f_i(n)\} \cup \{S = g(n)\}$ in the theory of polynomial arithmetic.
- A finite basis for this ideal is computed using techniques from Gröbner bases and elimination theory. This step is similar to the case of the invariant ideal for solvable loops, see e.g., [RcK04, Kov08].

In conclusion, we infer a *finite representation of the ideal of polynomial invariants for loops with unsolvable recurrence operators.*

5.6 Synthesising Solvable Loops from Unsolvable Loops

In previous sections, we introduced a new technique to compute invariants for unsolvable loops; that is, loops containing defective variables. An orthogonal challenge is to

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

synthesise a solvable loop from an unsolvable loop that preserves or over-approximates given specifications.

In this section we establish, with Algorithm 4, a new method to synthesise a solvable loop $\mathcal{P}'$ from an unsolvable loop $\mathcal{P}$. The solvable loop $\mathcal{P}'$ over-approximates the behaviour of $\mathcal{P}$ in the sense that every polynomial invariant of $\mathcal{P}'$ is an invariant of $\mathcal{P}$. Moreover, we show the invariants among effective variables of $\mathcal{P}$ and $\mathcal{P}'$ coincide. The following example illustrates the main idea leading to Algorithm 4.

Algorithm 4 Solvable Loop Synthesis

Input: Unsolvable loop $\mathcal{P}$ with recurrence operator $\mathcal{R}$, degree $d \in \mathbb{N}$
Output: Solvable loop $\mathcal{P}'$

1: Compute $E(\mathcal{P})$ and $D(\mathcal{P})$ using Algorithm 3.
2: Fix candidate polynomial $S(n)$ of degree d (as in Section 5.5 (5.2)).
3: Solve for coefficients in
4: $\quad S(n{+}1) = \kappa S(n) + \sum_{M \in \mathcal{E}_n(\mathcal{P})} c_M M$ (as in Section 5.5 (5.3)).
5: `initial` = []
6: `body_left` = []
7: `body_right` = []
8: ▷ *Add all effective variables to new loop*
9: **for** $x \in E(\mathcal{P})$ **do**
10: $\quad$ `initial.append(`$x \leftarrow x_0$`)`
11: $\quad$ `body_left.append(`x`)`
12: $\quad$ `body_right.append(`$\mathcal{R}[x]$`)`
13: **end for**
14: ▷ *Add well-behaved combination of defective monomials*
15: **if** $S \neq 0$ **then**
16: $\quad$ Choose a fresh symbol `s`.
17: $\quad$ `initial.append(s` $\leftarrow S(0)$`)`
18: $\quad$ `body_left.append(s)`
19: $\quad$ `body_right.append(`κ`s` $+ \sum_{M \in \mathcal{E}(\mathcal{P})} c_M M$`)`
20: **end if**
21: $\mathcal{P}' \leftarrow$ `initial` "while $\star$ do" `body_left` $\leftarrow$ `body_right` "end while"**return** $\mathcal{P}'$

Example 30. *Example 28 showed how to synthesise the polynomial $S = x+y$ of defective variables x and y for the loop in Figure 5.1a such that S admits a closed-form. In this case, the polynomial of program variables S satisfies the linear inhomogeneous recurrence $S(n+1) = 2S(n) - 3z(n) + 3$. We can use this recurrence to construct a solvable loop from the unsolvable from Figure 5.1a that captures the dynamics of the only effective variable z as well as S:*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

$z \leftarrow 0$
while $\star$ ***do***
$\quad z \leftarrow 1 - z$
$\quad x \leftarrow 2x + y^2 + z$
$\quad y \leftarrow 2y - y^2 + 2z$
end while
	Algorithm 4
	$\longmapsto$
		$z \leftarrow 0$
		$s \leftarrow x_0 + y_0$
		while $\star$ ***do***
		$\quad \begin{pmatrix} z \\ s \end{pmatrix} \leftarrow \begin{pmatrix} 1 - z \\ 2s - 3z + 3 \end{pmatrix}$
		end while

Our algorithmic approach, as given in Algorithm 4, synthesises the solvable loop from the unsolvable loop on the left. Such a synthesis step is implemented within our tool (Section 5.9), allowing us to synthesize the above solvable loop in about 1 *second.*

The inputs for Algorithm 4 are an unsolvable loop $\mathcal{P}$ with recurrence operator $\mathcal{R}$ and a fixed degree $d \in \mathbb{N}$; the algorithm outputs a solvable loop $\mathcal{P}'$, if it exist. Briefly, the algorithm invokes the invariant synthesis procedure from Section 4 (for degree d) and constructs the loop $\mathcal{P}'$ from $\mathcal{P}$ by removing all the defective variables and then introducing a new variable s that models an invariant among defective monomials, if such an invariant exists. The recurrence operator $\mathcal{R}'$ associated with the synthesised loop $\mathcal{P}'$ is the canonical recurrence operator: $\mathcal{R}'$ maps every program variable to its assignment.

Lemma 28 (Soundness)**.** *The loop $\mathcal{P}'$ returned by Algorithm 4 is solvable. Moreover, we have $E(\mathcal{P}) \subseteq \text{Vars}(\mathcal{P}')$.*

Proof. Using our characterisation of solvable and unsolvable loops in terms of effective and defective variables (Theorem 27), we show that all variables in $\mathcal{P}'$ are effective. The variables of $\mathcal{P}'$ consist of the effective variables of $\mathcal{P}$ as well as the fresh variable s: $\text{Vars}(\mathcal{P}') = E(\mathcal{P}) \cup \{s\}$, or $\text{Vars}(\mathcal{P}') = E(\mathcal{P})$ if no invariant among defective monomials with degree d exists.

First, every $x \in E(\mathcal{P})$ necessarily remains effective for the synthesised program $\mathcal{P}'$: in the dependency graph of $\mathcal{P}'$, the variable x cannot occur in a cycle containing s, because s is a fresh variable and hence cannot appear in the recurrence $\mathcal{R}[y]$ for any $y \in E(\mathcal{P})$ (Algorithm 4 line 16). Hence, if $z \in E(\mathcal{P}) \cap D(\mathcal{P}')$, then there must exist a cycle in the dependency graph of $\mathcal{P}'$ with a non-linear edge (x, y) (i.e., $\mathcal{L}(x, y) = N$) such that every vertex in the cycle is in $E(\mathcal{P})$. Consequently, this cycle is also present in the dependency graph of the original program $\mathcal{P}$. This means that not all variables in $E(\mathcal{P})$ are effective, which is a contradiction.

Second, the variable s is effective. Because s is a fresh variable, the only incoming edge of s in the dependency graph of $\mathcal{P}'$ is a linear self-loop (Algorithm 4 line 19). Hence s cannot occur in a cycle with a non-linear edge. Moreover, all outgoing edges point to effective variables. Thus $s \in E(\mathcal{P}')$ is effective. □

With the next two lemmas, we prove that the loop $\mathcal{P}'$ synthesised by Algorithm 4 over-approximates the invariants of the unsolvable loop $\mathcal{P}$. Let $\text{Inv}(\mathcal{P})$ and $\text{Inv}(\mathcal{P}')$ denote the invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$ respectively. The next lemma establishes that the synthesised loop $\mathcal{P}'$ is *complete with respect to invariants among effective variables.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Lemma 29 (Completeness with respect to Effective Variables)**.** *The invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$ coincide when restricted on the effective variables of $\mathcal{P}$. That is,*

$$\operatorname{Inv}(\mathcal{P}) \cap \overline{\mathbb{Q}}[E(\mathcal{P})] = \operatorname{Inv}(\mathcal{P}') \cap \overline{\mathbb{Q}}[E(\mathcal{P})].$$

Proof. By the construction of $\mathcal{P}'$, every $x \in E(\mathcal{P})$ is also a variable of $\mathcal{P}'$. To distinguish the variable x in $\mathcal{P}$ from the variable x in $\mathcal{P}'$ we refer to the latter by x'. We will show that for every $x \in E(\mathcal{P})$ the sequences $\langle x(n) \rangle_n$ and $\langle x'(n) \rangle_n$ coincide. If this is the case, the polynomial invariants for the programs $\mathcal{P}$ and $\mathcal{P}'$ among the variables $E(\mathcal{P})$ necessarily coincide as well.

Let $\mathcal{R}$ and $\mathcal{R}'$ be the recurrence operators associated to $\mathcal{P}$ and $\mathcal{P}'$, respectively. For every $x \in E(\mathcal{P})$ we have $\mathcal{R}[x] = \mathcal{R}'[x']$ and $x_0 = x_0'$ by the construction of $\mathcal{P}'$. Furthermore, by Corollary 23.1, defective variables cannot occur in $\mathcal{R}[x]$ for any $x \in E(\mathcal{P})$. Moreover, the fresh variable s cannot occur in $\mathcal{R}'[x']$ by the construction of $\mathcal{P}'$. Hence the two systems of first-order recurrences $\{x(n{+}1) = \mathcal{R}[x] \mid x \in E(\mathcal{P})\}$ and $\{x'(n{+}1) = \mathcal{R}'[x'] \mid x \in E(\mathcal{P})\}$ together with the initial values $\{x_0 \mid E(\mathcal{P})\}$ and $\{x_0' \mid E(\mathcal{P})\}$ induce the same sequences $\langle x(n) \rangle_n$ and $\langle x'(n) \rangle_n$ for all $x \in E(\mathcal{P})$. □

We note that when an invariant among defective monomials of degree d does not exist, the variables of the synthesised loop $\mathcal{P}'$ are precisely the effective variables of $\mathcal{P}$. In this case, Lemma 29 fully characterises the relationship between the invariants of $\mathcal{P}$ and $\mathcal{P}'$.

In the following, let us consider the complementary case, namely when an invariant among the defective monomials of $\mathcal{P}$ exists. Hence, the synthesised loop $\mathcal{P}'$ contains the additional fresh variable s modelling the behaviour of this invariant. With the next lemma, we confirm that the synthesised loop $\mathcal{P}'$ is indeed a *sound over-approximation of the unsolvable loop* $\mathcal{P}$. We show that every invariant of $\mathcal{P}'$ is also an invariant of $\mathcal{P}$. The program variable $\mathsf{s} \in \operatorname{Vars}(\mathcal{P}')$ introduced by Algorithm 4 is however not a program variable of $\mathcal{P}$; nevertheless, s models the polynomial S of defective variables in $\mathcal{P}$. Hence, to compare the invariant ideals of $\mathcal{P}$ and $\mathcal{P}'$, we need to "substitute" s by the polynomial of defective variables it models. This can be done by adding the equation $\mathsf{s} = S$ ($\mathsf{s} - S = 0$) to the invariant ideal of $\mathcal{P}'$ and restricting the resulting ideal to $\operatorname{Vars}(\mathcal{P})$:

$$I(\operatorname{Inv}(\mathcal{P}') \cup \{\mathsf{s} - S\}) \cap \overline{\mathbb{Q}}[\operatorname{Vars}(\mathcal{P})]. \tag{5.4}$$

Lemma 30 (Over-Approximation)**.** *Let J be the ideal in* (5.4) *constructed from the invariant ideal of $\mathcal{P}'$ by replacing the program variable s by the polynomial of defective variables it models. Then, $J \subseteq \operatorname{Inv}(\mathcal{P})$.*

Proof. As argued in the proof of Lemma 29, for every $x \in E(\mathcal{P})$, the sequences corresponding to the variable x in both $\mathcal{P}$ and $\mathcal{P}'$ coincide; that is, $\langle x(n) \rangle_n \equiv \langle x'(n) \rangle_n$. Furthermore, we have $\operatorname{Vars}(\mathcal{P}') = E(\mathcal{P}) \cup \{\mathsf{s}\}$. The fresh variable s in $\mathcal{P}'$ models the

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

polynomial $S \in \overline{\mathbb{Q}}[\text{Vars}(\mathcal{P})]$. Let $\langle \mathtt{s}(n) \rangle_n$ be the sequence induced by the program variable $\mathtt{s}$ in $\mathcal{P}'$ and, likewise, $\langle S(n) \rangle_n$ the sequence induced by the polynomial $S \in \overline{\mathbb{Q}}[\text{Vars}(\mathcal{P})]$. Then, by the construction of $\mathcal{P}'$, we have $\langle \mathtt{s}(n) \rangle_n \equiv \langle S(n) \rangle_n$.

Let $Q \in \text{Inv}(P')$. By the definition of the ideal J in (5.4), it holds that

$$Q \in \text{Inv}(P') \iff Q\{\mathtt{s} \mapsto S\} \in J$$

(where the notation indicates that S is substituted for $\mathtt{s}$).

Now, Q is a polynomial relation among the sequences induced by the variables in $\text{Vars}(\mathcal{P}')$. Because $\langle x(n) \rangle_n \equiv \langle x'(n) \rangle_n$ for every $x \in E(\mathcal{P})$ and $\langle \mathtt{s}(n) \rangle_n \equiv \langle S(n) \rangle_n$, the polynomial $Q\{\mathtt{s} \mapsto S\}$ represents a relation among the sequences induced by the variables in $\text{Vars}(\mathcal{P})$. Hence we have $Q \in \text{Inv}(\mathcal{P})$. □

Remark 7. *Our loop synthesis procedure given in Algorithm 4 computes a single invariant among defective monomials (if such an invariant exists) of an unsolvable loop $\mathcal{P}$. The results in this section naturally generalise to multiple invariants among defective monomials, as follows: for every invariant I, add a fresh variable s_I modelling the behaviour of I to the synthesised program $\mathcal{P}'$. Note that with each additional invariant I added, the dynamics of the synthesised loop $\mathcal{P}'$ more closely resembles that of the unsolvable loop $\mathcal{P}$.*

5.7 Adjustments for Unsolvable Operators in Probabilistic Programs

5.7.1 Defective Variables in Probabilistic Loop Models

The works [MSBK22a, BKS19] defined recurrence operators for probabilistic loops. Specifically, a recurrence operator is defined for loops with polynomial assignments, probabilistic choice, and drawing from common probability distributions with constant parameters. Recurrences for deterministic loops model the precise values of program variables. For probabilistic loops, this approach is not viable, due to the stochastic nature of the program variables. Thus a recurrence operator for a probabilistic loop models *(higher) moments* of program variables. As illustrated in Example 25, the recurrences of a probabilistic loop are taken over expected values of program variable monomials.

The authors of [MSBK22a, BKS19] explicitly excluded the case of circular non-linear dependencies to guarantee computability. However, in contrast to our notions in Sections 5.3, they defined variable dependence not on the level of recurrences but on the level of assignments in the loop body. To use the notions of effective and defective variables for probabilistic loops, we follow the same approach and base the dependency graph on assignments rather then recurrences. We illustrate the necessity of this adaptation in the following example.

Example 31. *A probabilistic assignment $x \leftarrow a\ \{p\}\ b$ intuitively means that x is assigned a with probability p and b with probability $1-p$. Consider the following probabilistic loop*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

and associated set of first-order recurrence relations in terms of the expectation operator $\mathbb{E}(\,\cdot\,)$.

while $\star$ **do**
  $y \leftarrow 4y(1-y)$
  $x \leftarrow x - y \ \{1/2\}\ x + y$
end while

$$
\begin{aligned}
\mathbb{E}(y_{n+1}) &= 4\mathbb{E}(y_n) - 4\mathbb{E}(y_n^2)\\
\mathbb{E}(x_{n+1}) &= \mathbb{E}(x_n)\\
\mathbb{E}(x_{n+1}^2) &= \frac{1}{2}\mathbb{E}((x_n - y_{n+1})^2) + \frac{1}{2}\mathbb{E}((x_n + y_{n+1})^2)\\
&= \mathbb{E}(x_n^2) + \mathbb{E}(y_{n+1}^2)
\end{aligned}
$$

It is straightforward to see that variable y is defective from the deterministic update $y \leftarrow 4y(1-y)$ with its characteristic non-linear self-dependence. Moreover, y appears in the probabilistic assignment of x: However, due to the particular form of the assignment, the recurrence of $\mathbb{E}(x_n)$ does not contain y. Nevertheless, y appears in the recurrence of $\mathbb{E}(x_n^2)$. This phenomenon is specific to the probabilistic setting. For deterministic loops, it is always the case that if the values of a program variable w do not depend on defective variables, then neither do the values of any power of w.

In light of the phenomenon exhibited in Example 31, we adapt our notion of *variable dependency*, for probabilistic loops. Without loss of generality, we assume that every program variable has exactly one assignment in the loop body. Let $\mathcal{P}$ be a probabilistic loop and $x, y \in \mathrm{Vars}(\mathcal{P})$. We say *$x$ depends on y*, if y appears in the assignment of x. Additionally, the dependency is *linear* if all occurrences of y in the assignment of x are linear, else the dependency is *non-linear*. Further, we consider the transitive closure of variable dependency analogous to deterministic loops and Definition 36.

With variable dependency thus defined, the dependency graph and the notions of effective and defective variables follow immediately. Analogous to our characterisation of unsolvable recurrence operators in terms of defective variables for deterministic loops, *all (higher) moments* of effective variables of probabilistic loops can be described by a system of linear recurrences [MSBK22a, BKS19]. For defective variables this property will generally fail For instance, in Example 31, the variable x is now classified as defective and $\mathbb{E}(x_n^2)$ cannot be modelled by linear recurrences for some initial values.

The only necessary change to the invariant synthesis algorithm from Section 5.5 is as follows: instead of program variable monomials, we consider expected values of program variable monomials. Now, our invariant synthesis technique from Section 5.5 can also be applied to probabilistic loops to synthesise combinations of expected values of defective monomials that do satisfy a linear recurrence.

5.7.2 Synthesising Solvable Probabilistic Loops

In Algorithm 4 we introduced a procedure, utilising our new invariant synthesis technique from Section 5.5, to over-approximate an unsolvable loop by a solvable loop. The inputs to Algorithm 4 are an unsolvable loop with a recurrence operator and a natural

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

number specifying a fixed degree. As mentioned, our invariant synthesis procedure is also applicable to probabilistic loops using the recurrence operator modelling moments of program variables introduced in [MSBK22a, BKS19]. Hence, Algorithm 4 can also be used to synthesise solvable loops from unsolvable probabilistic loops. In the probabilistic case, however, the invariants computed by our approach are over *moments of program variables*. Therefore, the invariant ideal of probabilistic loops describes polynomial relations among a given set of moments of program variables, such as the expected values. Consequently, the loop synthesised by Algorithm 4 for a given probabilistic loop will be deterministic and model the dynamics of moments of program variables of the probabilistic loop.

Example 32. *Recall the program $\mathcal{P}_{MC}$ of Example 25. An invariant synthesised by our approach in Section 5.5 with degree* 1 *is* $\mathbb{E}(x_n - y_n) = \frac{5^n}{6^n}(x_0 - y_0)$*. Hence the solvable loop synthesised by Algorithm 4 for $\mathcal{P}_{MC}$ with input degree* 1 *is*

$\mathtt{f} \leftarrow x_0 + y_0$
while $\star$ ***do***
  $s \leftarrow \frac{1}{2}$
  $\mathtt{f} \leftarrow \frac{5}{6}\mathtt{f}$
end while

where $\mathtt{f}$ *is the fresh variable introduced by Algorithm 4 modelling* $\mathbb{E}(x_n - y_n)$*. Our approach from Algorithm 4 synthesises this solvable loop from $\mathcal{P}_{MC}$, using less than* 0.5 *second within our implementation (Section 5.9).*

5.8 Applications of Unsolvable Operators towards Invariant Generation

Our approach automatically generates invariants for programs with defective variables (Section 5.5), and pushes the boundaries of both theory and practice of invariant generation: we introduce and incorporate defective variable analysis into the state-of-the-art methodology for reasoning about solvable loops, complementing thus existing methods, see e.g., [RcK04, Kov08, KCBR18, HJK18b], in the area. As such, the class of unsolvable loops that can be handled by our approach extends (aforementioned) existing approaches on polynomial invariant generation. The experimental results of our approach (see Section 5.9) demonstrate the efficiency and scalability of our technique in deriving invariants for unsolvable loops. Since our approach to loops via recurrences is generic, we can deal with emerging applications of programming paradigms such as: transitions systems and statistical moments in probabilistic programs; and reasoning about biological systems. We showcase these applications in this section and also exemplify the limitations of our approach. In the sequel, we write $\mathbb{E}(t)$ to refer to the expected value of an expression t, and denote by $\mathbb{E}(t_n)$ (or $\mathbb{E}(t(n))$) the expected value of t at loop iteration n.

Example 33 (Moments of Probabilistic Programs [SO19])**.** *As mentioned in Example 32,* $\mathbb{E}(x_n - y_n) = \frac{5^n}{6^n}(x_0 - y_0)$ *is an invariant for the program $\mathcal{P}_{MC}$ introduced in Example 25.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Closed-form solutions for higher order expressions are also available; for example,

$$\mathbb{E}((x_n - y_n)^d) = \frac{(2^d + 3^d)^n}{2^n \cdot 3^{dn}}(x_0 - y_0)^d$$

refers to the dth moment of $x(n) - y(n)$. While the work in [SO19] uses martingale theory to synthesise the above invariant (of degree 1), our approach automatically generates such invariants over higher-order moments (see Table 5.2). We note to this end that the defective variables in $\mathcal{P}_{MC}$ are precisely x and y as can be seen from their mutual non-linear interdependence. Namely, we have $D(\mathcal{P}_{MC}) = \{x, y\}$ and $E(\mathcal{P}_{MC}) = \{s\}$.

Example 34 (`non-lin-markov-2`)**.** *We give a second example of a non-linear Markov chain. We analyse the moments of this probabilistic program in the next section.*

$x, y \leftarrow 0, 1$
while $\star$ ***do***
  $s \leftarrow \text{Bernoulli}(1/2)$
  if $s = 0$ ***then***
    $$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \frac{4}{10}(x + xy) \\ \frac{4}{10}(13x + \frac{2}{3}y + xy) \end{pmatrix}$$
  else
    $$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \frac{4}{10}(x + y + \frac{2}{3}xy) \\ \frac{4}{10}(2y + \frac{2}{3}xy) \end{pmatrix}$$
  end if
end while

Example 35 (Biological Systems [BFPS02])**.** *A model for the decision-making process of swarming bees choosing one nest-site from a selection of two is introduced in [BFPS02] and further studied in [DDP16, SCGP20]. Previous works have computed probability distributions for this model [SCGP20]. The (unsolvable) loop is a discrete-time model with five classes of bees (each represented by a program variable). The coefficient Δ is the length of the time-step in the model and the remaining coefficients parameterise the rates of change. All coefficients here are symbolic.*

$$\begin{pmatrix} x \\ y_1 \\ y_2 \\ z_1 \\ z_2 \end{pmatrix} \leftarrow \begin{pmatrix} \text{Normal}(475, 5) \\ \text{Uniform}(350, 400) \\ \text{Uniform}(100, 150) \\ \text{Normal}(35, 1.5) \\ \text{Normal}(35, 1.5) \end{pmatrix}$$
while $\star$ ***do***
$$\begin{pmatrix} x \\ y_1 \\ y_2 \\ z_1 \\ z_2 \end{pmatrix} \leftarrow \begin{pmatrix} x - \Delta(\beta_1 x y_1 + \beta_2 x y_2) \\ y_1 + \Delta(\beta_1 x y_1 - \gamma y_1 + \delta\beta_1 y_1 z_1 + \alpha\beta_1 y_1 z_2) \\ y_2 + \Delta(\beta_2 x y_2 - \gamma y_2 + \delta\beta_2 y_2 z_2 + \alpha\beta_2 y_2 z_1) \\ z_1 + \Delta(\gamma y_1 - \delta\beta_1 y_1 z_1 - \alpha\beta_2 y_2 z_1) \\ z_2 + \Delta(\gamma y_2 - \delta\beta_2 y_2 z_2 - \alpha\beta_1 y_1 z_2) \end{pmatrix}$$
end while

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

We note that the model in [SCGP20] uses truncated Normal distributions. Contrary to our approach, their technique is limited to finite supports for the program variables.

In the loop above, each of the variables exhibits non-linear self-dependence, and so the variables are partitioned into $D(\mathcal{P}) = \{x, y_1, y_2, z_1, x_2\}$ *and* $E(\mathcal{P}) = \emptyset$*. While the recurrence operator of the loop above is unsolvable, our approach infers polynomial loop invariants using defective variable reasoning (Section 5.5). Namely, we generate the following closed-form solutions over expected values of program variables:*

$$\begin{aligned}\mathbb{E}(x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n)) &= 1045,\\ \mathbb{E}((x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))^2) &= 3277349/3, \quad \textit{and}\\ \mathbb{E}((x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))^3) &= 1142497455.\end{aligned}$$

One can interpret such invariants in terms of the biological assumptions in the model. Take, for example, the fact that $\mathbb{E}(x(n) + y_1(n) + y_2(n) + z_1(n) + z_2(n))$ *is constant. This invariant is in line with the assumption in the model that the total population of the swarm is constant. In fact, our invariants reflect the behaviour of the system in the original* continuous-time *model proposed in [BFPS02], because our approach is able to process all coefficients (most importantly* Δ*) as symbolic constants.*

Example 36 (Probabilistic Transition Systems [SO19])**.** *Consider the following probabilistic loop modelling a* probabilistic transition system *from [SO19]:*

while $\star$ ***do***

$$\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} \text{Normal}(0,1) \\ \text{Normal}(0,1) \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} x + axy \\ y + bxy \end{pmatrix}$$

end while

While [SO19] uses martingale theory to synthesise a degree one invariant of the form $a\mathbb{E}(x_k) + b\mathbb{E}(y_k) = a\mathbb{E}(x_0) + b\mathbb{E}(y_0)$*, our technique automatically generates invariants over higher-order moments involving the defective variables* x *and* y*, as presented in Table 5.2.*

The next example demonstrates an unsolvable loop whose recurrence operator cannot (yet) be handled by our approach.

Example 37 (Trigonometric Updates)**.** *As our approach is limited to polynomial updates of the program variables, the loop below cannot be handled by our technique:*

while $\star$ ***do***

$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \cos(x) \\ \sin(x) \end{pmatrix}$$

end while

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Note the trigonometric functions are transcendental, from which it follows that one cannot generally obtain closed-form solutions for the program variables. Nevertheless, this program does admit polynomial invariants in the program variables; for example, $x^2 + y^2 = 1$. Although our definition of a defective variables does not apply here, we could say the variable x here is somehow defective*: while the exact value of* $\sin(x)$ *cannot be computed, it could be approximated using power series. Extending our approach with more general notions of defective variables is an interesting line for future work.*

Examples 38–41 (below) are custom-made benchmarks. We have tailored these benchmarks to demonstrate the flexibility and applicability of our method to the current state of the art. Our experimental analysis is delayed to Section 5.9.

Example 38 (`squares+`)**.**

$s, x, y, z \leftarrow 0, 2, 1, 0$
while $\star$ ***do***
  $s \leftarrow \text{Bernoulli}(1/2)$
  $z \leftarrow z - 1 \ \{1/2\} \ z + 2$
  $x \leftarrow 2x + y^2 + s + z$
  $y \leftarrow 2y - y^2 + 2s$
end while

Example 39 (`prob-squares`)**.**

$g \leftarrow 1$
while $\star$ ***do***
  $g \leftarrow \text{Uniform}(g, 2g)$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \leftarrow \begin{pmatrix} a^2 + 2bc - df + b \\ df - a^2 + 2bd + 2c \\ g - bc - bd + \frac{1}{2}a \end{pmatrix}$$

end while

Example 40 (`squares-squared`)**.**

while $\star$ ***do***

$$\begin{pmatrix} x \\ y \\ z \\ m \end{pmatrix} \leftarrow \begin{pmatrix} xyz + x^2 \\ 2y + z - x^2 + 3ymz^2 \\ \frac{3}{2}x + \frac{3}{2}z + \frac{1}{2}y + \frac{1}{2}x^2 \\ \frac{2}{3}z + 3m - \frac{1}{3}x^2 - \frac{1}{3}xyz - ymz^2 \end{pmatrix}$$

end while

Example 41 (`deg-d`)**.** *The benchmarks* `deg-5`*,* `deg-6`*,* `deg-7`*,* `deg-8`*,* `deg-9`*, and* `deg-500` *are parameterised by the degree d in the following program.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$x, y \leftarrow 1, 1$
while $\star$ ***do***
$z \leftarrow \text{Normal}(0, 1)$
$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 2x^d + z + z^2 \\ 3x^d + z + z^2 + z^3 \end{pmatrix}$$
end while

The following set of examples are taken from the literature on the theory of trace maps. Arguably the most famous example is the classical *Fibonacci Trace Map* $f\colon \mathbb{R}^3 \to \mathbb{R}^3$ given by $f(x, y, z) = (y, z, 2yz - x)$ (Example 42 below); said map has garnered the attention of researchers in fields as diverse as invariant analysis, representation theory, geometry, and mathematical physics (cf. the survey papers [BGJ93, RB94]). From a computational viewpoint, trace maps arise from substitution rules on matrices (see, again, the aforementioned survey papers). Given two matrices $A, B \in \text{SL}(2, \mathbb{R})$ (the group of 2×2 matrices with unit determinant), consider the following substitution rule on strings of matrices: $A \mapsto B$ and $B \mapsto AB$. The classical Fibonacci Trace Map is determined by the action of this substitution on the traces of the matrices; i.e.,

$$f(\text{tr}(A), \text{tr}(B), \text{tr}(AB)) = (\text{tr}(B), \text{tr}(AB), -\text{tr}(A) + 2\,\text{tr}(B)\,\text{tr}(AB)).$$

Further examples of trace maps (Example 43 and Example 44 below) are constructed from similar substitution rules on strings of matrices. For Examples 42–44, our method generates the cubic polynomial invariant

$$x^2 + y^2 + z^2 - 2xyz = x_0^2 + y_0^2 + z_0^2 - 2x_0y_0z_0,$$

from the well-studied class of *Fricke–Vogt invariants* as well as higher-degree polynomial invariants.

Example 42 (`fib1`)**.**
while $\star$ ***do***
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ z \\ 2yz - x \end{pmatrix}$$
end while

Example 43 (`fib2`)**.** *A Generalised Fibonacci Trace Map*
while $\star$ ***do***
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 2xz - y \\ 4xyz - 2x^2 - 2y^2 + 1 \end{pmatrix}$$
end while

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Example 44 (`fib3`)**.** *A second Generalised Fibonacci Trace Map*

while $\star$ ***do***

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} 1 + x + y + xy - z \\ x \\ y \end{pmatrix}$$

end while

Examples 45–46 are while loops that generate *Markov triples* [Cas72, Chapter II.3]; that is, at every iteration each loop variable takes an integer value that appears in a Diophantine solution of the Markov equation $x^2 + y^2 + z^2 = 3xyz$.

Example 45 (`markov-triples-toggle`)**.** *A while loop that generates an infinite sequence of nodes on the Markov tree (the walk alternates between 'upper' and 'lower' branches).*

$x,y,z = 1,1,2;$

$branch = 0;$

while $\star$ ***do***

  if $branch = 0$ ***then***

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x \\ 3xy - z \\ y \end{pmatrix};$$

    $branch = 1;$

  else

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 3yz - x \\ z \end{pmatrix};$$

    $branch = 0;$

  end if

end while

For this example, our approach generates the polynomial invariant, the Markov equation, *given by* $x^2 + y^2 + z^2 - 3xyz = 0$.

Example 46 (`markov-triples-random`)**.** *A while loop that simulates a Bernoulli walk on the Markov tree.*

$x,y,z = 1,1,2$

while $\star$ ***do***

  $p \leftarrow \text{Bernoulli}(1/2)$

  if $p = 1$ ***then***

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x \\ 3xy - z \\ y \end{pmatrix}$$

  else

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} y \\ 3yz - x \\ z \end{pmatrix}$$

end if
end while

For this benchmark, our technique generates both closed-forms and invariants in (higher) moments of program variables such as $\mathbb{E}(x_n^2+y_n^2+z_n^2-3x_ny_nz_n) = 0$ *and* $\mathbb{E}(x_n-z_n) = 2^{-n}$.

The next two benchmarks concern a special class of polynomial automorphisms, the *Yagzhev* maps, (sometimes *cubic-homogeneous* maps) introduced (independently) by Yagzhev [Jag80] and Bass, Connell, and Wright [BCW82] in the context of the Jacobian conjecture. Recall that Yagzhev maps $f\colon \mathbb{C}^n \to \mathbb{C}^n$ take the form $f(x) = x - g(x)$ such that $\det f'(x) = 1$ for all x, and $g\colon \mathbb{C}^n \to \mathbb{C}^n$ is a homogeneous polynomial mapping of degree 3. De Bondt exhibited a Yagzhev mapping in 10 dimensions that has no linear invariants (providing a counterexample to the "linear dependence conjecture" for the class of maps) [dB06]. Zampieri demonstrated that De Bondt's example has quadratic and cubic invariants [Zam08] (see Example 47 for such a Yagzhev map in 9 dimensions). Work by Santos Freire, Gorni, and Zampieri [dSFGZ08] exhibited a Yagzhev mapping in 11 dimensions that has neither linear invariants nor quadratic invariants (see Example 48).

Example 47 (`yagzhev9` [Zam08])**.**

while $\star$ ***do***

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + x_1x_7x_9 + x_2x_9^2 \\ x_2 - x_1x_7^2 - x_2x_7x_9 \\ x_3 + x_3x_7x_9 + x_4x_9^2 \\ x_4 - x_3x_7^2 - x_4x_7x_9 \\ x_5 + x_5x_7x_9 + x_6x_9^2 \\ x_6 - x_5x_7^2 - x_6x_7x_9 \\ x_7 + (x_1x_4 - x_2x_3)x_9 \\ x_8 + (x_3x_6 - x_4x_5)x_9 \\ x_9 + (x_1x_4 - x_2x_3)x_8 - (x_3x_6 + x_4x_5)x_7 \end{pmatrix}$$

end while

For this example, our method generates an invariant quadratic homogeneous polynomial in six variables and three symbolic constants, which we can interpret in terms of determinants (as given below):

$$a\begin{vmatrix} x_1(n) & x_2(n) \\ x_3(n) & x_4(n) \end{vmatrix} + b\begin{vmatrix} x_3(n) & x_4(n) \\ x_5(n) & x_6(n) \end{vmatrix} + c\begin{vmatrix} x_1(n) & x_2(n) \\ x_5(n) & x_6(n) \end{vmatrix}$$
$$= a\begin{vmatrix} x_1(0) & x_2(0) \\ x_3(0) & x_4(0) \end{vmatrix} + b\begin{vmatrix} x_3(0) & x_4(0) \\ x_5(0) & x_6(0) \end{vmatrix} + c\begin{vmatrix} x_1(0) & x_2(0) \\ x_5(0) & x_6(0) \end{vmatrix}.$$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Our computation confirms previous work by the authors of [Zam08]. Those authors demonstrated that this example has no linear invariants, but does admit the above quadratic invariant.

Example 48 (`yagzhev11` [dSFGZ08])**.**

while $\star$ **do**

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{pmatrix} \leftarrow \begin{pmatrix} x_1 - x_3 x_{10}^2 \\ x_2 - x_2 x_{11}^2 \\ x_3 + x_1 x_{11}^2 - x_2 x_{10}^2 \\ x_4 - x_6 x_{10}^2 \\ x_5 - x_6 x_{11}^2 \\ x_6 + x_4 x_{11}^2 - x_5 x_{10}^2 \\ x_7 - x_9 x_{10}^2 \\ x_8 - x_9 x_{11}^2 \\ x_9 + x_7 x_{11}^2 - x_8 x_{10}^2 \\ x_{10} + (x_3 x_5 - x_2 x_6) x_7 + (x_1 x_6 - x_3 x_4) x_8 + (x_2 x_4 - x_1 x_5) x_9 \\ x_{11} - x_{10}^3 \end{pmatrix}$$

end while

For this example, our approach generates an invariant cubic homogeneous polynomial, which we can interpret as the determinant of a matrix in the program variables:

$$\begin{vmatrix} x_1(n) & x_2(n) & x_3(n) \\ x_4(n) & x_5(n) & x_6(n) \\ x_7(n) & x_8(n) & x_9(n) \end{vmatrix} = \begin{vmatrix} x_1(0) & x_2(0) & x_3(0) \\ x_4(0) & x_5(0) & x_6(0) \\ x_7(0) & x_8(0) & x_9(0) \end{vmatrix}.$$

For the avoidance of doubt, the above polynomial was previously found by the authors of [dSFGZ08]. Indeed, our implementation confirms previous results: there are neither linear nor quadratic invariants for this example.

Example 49 (`nagata` [Nag72])**.** *Let us now consider the classical* Nagata automorphism *introduced by Nagata [Nag72, pg. 41] (see also [vP03]).*

while $\star$ **do**

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \leftarrow \begin{pmatrix} x - 2(xz + y^2)y - (xz + y^2)^2 z \\ y + (xz + y^2) z \\ z \end{pmatrix}$$

end while

For the Nagata Automorphism, it is easy to see that the variable z is effective and so contributes a linear invariant. When used to search for quadratic closed-forms, our method generates the polynomial

$$c(x(n)z(n) + y(n)^2) = az(0) + bz(0)^2 - az(n) - bz(n)^2 + c(x(0)z(0) + y(0)^2)$$

where a, b, and c are symbolic constants. Here we note that $x(n)z(n)$ and $y(n)^2$ are defective monomials. Our computation confirms the invariants and closed-forms established in [Nag72].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.9 Experiments

In this section, we report on our implementation towards fully automating the analysis of unsolvable loops and describe our experimental setting and results.

5.9.1 Implementation

Algorithm 3, our method for synthesising invariants involving defective variables, and Algorithm 4, for the synthesis of solvable loops from unsolvable loops, are implemented in the POLAR tool[3]. We use `python3` and the `sympy` package [MSP+17] for symbolic manipulations of algebraic expressions.

5.9.2 Benchmark Selection

While previous works [RcK04, RK07, dOBP16, HJK18a, BKS19, KCBR18] consider invariant synthesis, their techniques are only applicable in a restricted setting: the analysed loops are, for the most part, solvable; or, for unsolvable loops, the search for polynomial invariants is template-driven or employs heuristics. In contrast, the work herein complements and extends the techniques presented for solvable loops in [RcK04, RK07, dOBP16, HJK18a, BKS19, KCBR18]. Indeed, our automated approach turns the problem of polynomial invariant synthesis into a decidable problem for a larger class of unsolvable loops.

While solvable loops can clearly be analysed by our approach, the main benefit of our method comes with handling unsolvable loops by translating them into solvable ones. For this reason, in our experimentation we are not interested in examples of solvable loops and so only focus on unsolvable loop benchmarks. There is therefore no sensible baseline that we can compare against, as state-of-the-art techniques cannot routinely synthesise invariants for unsolvable loops in the generality we present.

We present a set of 23 examples of unsolvable loops, as listed in Table 5.1[4]. Common to all 23 benchmarks from Table 5.1 is the exhibition of circular non-linear dependencies within the variable assignments. We display features of our benchmarks in Table 5.1 (for example, column 3 of Table 5.1 counts the number of defective variables for each benchmark).

Three examples from Table 5.1 are challenging benchmarks taken from the invariant generation literature [CVS16, DDP16, SO19, SCGP20]; full automation in analysing these examples was not yet possible. These examples are listed as `non-lin-markov-1`, `pts`, and `bees` in Table 5.1, respectively corresponding to Example 25 (and hence Example 33), Example 36, and Example 35 from Section 5.8. Eight further benchmarks, as described in Examples 42–49, are drawn from the theoretical physics and pure mathematics literature (references are given in Section 5.8).

[3] `https://github.com/probing-lab/polar`

[4] each benchmark in Table 5.1 references, in parentheses, the respective example

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.1: Features of the benchmarks. Var = Total number of loop variables; Def = Number of defective variables; Term = Total number of terms in assignments; Deg = Maximum degree in assignments; Cand-7 = Number of monomials in candidate with degree 7; Eqn-7 = Size of the system of equations associated with a candidate of degree 7; - = Timeout (60 seconds).

Benchmark	Var	Def	Term	Deg	Cand-7	Eqn-7
`squares (Fig. 5.1a)`	3	2	8	2	35	113
`squares+ (Ex.38)`	4	2	12	2	35	204
`non-lin-markov-1 (Ex. 25)`	2	2	11	2	35	64
`non-lin-markov-2 (Ex. 34)`	2	2	11	2	35	64
`prob-squares (Ex. 5.1b)`	3	3	4	3	119	337
`pts (Ex. 36)`	4	2	6	3	35	57
`squares-squared (Ex. 28)`	4	4	15	4	329	-
`bees (Ex. 35)`	5	5	21	5	791	-
`deg-5 (Ex. 41)`	3	2	8	5	35	42
`deg-6 (Ex. 41)`	3	2	8	6	35	42
`deg-7 (Ex. 41)`	3	2	8	7	35	42
`deg-8 (Ex. 41)`	3	2	8	8	35	43
`deg-9 (Ex. 41)`	3	2	8	9	35	43
`deg-500 (Ex. 41)`	3	2	8	500	35	43
`fib1 (Ex. 42)`	3	3	4	2	119	204
`fib2 (Ex. 43)`	3	3	7	3	119	368
`fib3 (Ex. 44)`	3	3	7	2	119	204
`markov-triples-toggle (Ex. 45)`	3	3	10	2	119	454
`markov-triples-random (Ex. 46)`	3	3	9	2	119	254
`yagzhev9 (Ex. 47)`	9	9	29	3	11439	-
`yagzhev11 (Ex. 48)`	11	11	30	3	31823	-
`nagata (Ex. 49)`	3	2	9	4	35	1313

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The remaining 12 examples of Table 5.1 are self-constructed benchmarks to highlight the key ingredients of our method in synthesising invariants associated with unsolvable recurrence operators.

Experimental Setup

We evaluate our approach in POLAR on the examples from Table 5.1. All our experiments were performed on a machine with a 1.80GHz Intel i7 processor and 16 GB of RAM.

Evaluation Setting

The landscape of benchmarks in the invariant synthesis literature for solvable loops can appear complex with: high numbers of variables, high degrees in polynomial updates, and multiple update options. However, we do not intend to compete on these metrics for solvable loops. The power of our invariant synthesis technique lies in its ability to handle 'unsolvable' loop programs: those with cyclic inter-dependencies and non-linear self-dependencies in the loop body. Regarding Algorithm 4, specifying our method for synthesising solvable from unsolvable loops, the main complexity lies in constructing an invariant involving defective variables. Once, such an invariant has been found, the remaining complexity of Algorithm 4 is linear in the number of program variables. Hence, the experimental results for our invariant synthesis technique also show the feasibility of our new method synthesising solvable from unsolvable loops. While the benchmarks of Table 5.1 may be considered simple, the fact that previous works cannot systematically handle such *simple models* crystallises that even simple loops can be unsolvable, limiting the applicability of state-of-the-art methods, as illustrated in the example below.

Example 50. *Consider the question:* does the unsolvable loop program `deg-9` in Table 5.1 (i.e. Example 41) possess a cubic invariant? *The program variables for* `deg-9` *are x, y, and z. The variables x and y are defective. Using* POLAR, *we derive that the cubic, non-trivial polynomial $p(x_n, y_n, z_n)$ given by*

$$12(ay_n + by_n^2 + cy_n^3 + dx_n + ex_ny_n + fx_ny_n^2) - (3a + 24b + 117c + 2d + 17e + 26f)x_n^2 - (6a - 6b + 315c + 4d - 2e + 88f)x_n^2y_n + 3(3a - 3b + 144c + 2d - e + 35f)x_n^3$$

yields a cubic polynomial loop invariant, where a, b, c, d, e, and f are symbolic constants. Moreover, for $n \geq 1$, the expectation of this polynomial (`deg-9` *is a probabilistic loop) in the nth iteration is given by*

$$\mathbb{E}(p(x_n, y_n, z_n)) = -108a + 312b - 1962c - 68d + 52e - 68f.$$

Experimental Results

Our experiments using POLAR to synthesise invariants are summarised in Table 5.2, using the examples of Table 5.1. Patterns in Table 5.2 show that, if time considerations

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.2: The time taken to search for polynomial candidates with closed-forms (results in seconds); - = Timeout (75 seconds); * = Found invariant of the corresponding degree.

	Candidate Degree						
Benchmark	1	2	3	4	5	6	7
`squares (Fig. 5.1a)`	*0.95	1.09	1.66	2.72	4.98	11.67	19.87
`squares+ (Ex. 38)`	*0.67	1.02	1.92	3.77	7.39	15.66	28.78
`non-lin-markov-1 (Ex. 25)`	*0.42	*0.68	*1.04	*2.35	*4.02	*9.81	*13.58
`non-lin-markov-2 (Ex. 34)`	*0.38	*0.59	*1.15	*2.40	*3.19	*5.91	*14.91
`prob-squares (Ex. 39)`	*0.76	1.50	4.69	20.85	-	-	-
`squares-and-cube (Fig. 5.1b)`	0.32	*0.51	*1.20	*4.21	*19.49	-	-
`pts (Ex. 36)`	*0.35	*0.49	*0.71	*1.13	*1.90	*3.42	*5.98
`squares-squared (Ex. 40)`		*0.48	*1.55	*8.92	-	-	-
`bees (Ex. 35)`	*0.69	*3.27	*42.16	-	-	-	-
`deg-5 (Ex. 41)`	*0.46	*0.65	*1.01	*1.94	*4.04	*8.20	*19.28
`deg-6 (Ex. 41)`	*0.54	*0.69	*1.62	*1.91	*4.32	*8.24	*19.75
`deg-7 (Ex. 41)`	*0.49	*0.98	*1.06	*1.84	*4.42	*8.98	*19.39
`deg-8 (Ex. 41)`	*0.45	*0.62	*1.08	*2.04	*4.07	*8.93	*20.97
`deg-9 (Ex. 41)`	*0.47	*0.65	*1.11	*1.84	*4.31	*7.85	*19.67
`deg-500 (Ex. 41)`	*0.47	*0.65	*1.08	*1.96	*4.29	*8.58	*21.05
`fib1 (Ex. 42)`	0.31	0.41	*0.91	2.02	2.97	*5.47	14.66
`fib2 (Ex. 43)`	0.31	0.49	*1.34	3.06	8.64	*17.24	-
`fib3 (Ex. 44)`	0.31	0.48	*1.73	3.46	16.92	-	-
`markov-triples-toggle (Ex. 45)`	0.39	0.61	*1.41	2.63	5.84	*9.45	22.71
`markov-triples-random (Ex. 46)`	*0.44	*0.62	*1.65	*3.52	*7.93	*21.39	*71.23
`yagzhev9 (Ex. 47)`	0.47	*2.44	*24.31	-	-	-	-
`yagzhev11 (Ex. 48)`	0.59	7.12	*39.27	-	-	-	-
`nagata (Ex. 49)`	0.33	*1.04	*3.49	*8.19	*40.03	-	-

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

are the limiting factor, then the greatest impact cannot be attributed to the number of program variables nor the maximum degree in the program assignments (Table 5.1). Three of the examples in Table 5.1 exhibit timeouts (60 secs) in the final column. The property common to each of these examples is the high number of monomial terms in any polynomial candidate of degree 7. In turn, this property feeds into a large system of simultaneous equations, which we solve to test for invariants. Indeed, time elapsed is not so strongly correlated with either of these program features. As supporting evidence we note the specific attributes of benchmark `deg-500` whose assignments include polynomial updates of large degree and yet returns synthesised invariants with relatively low time elapsed in Table 5.2. We note the significantly longer running times associated with the benchmark `bees` (Example 35). This suggests that mutual dependencies between program variables in the loop assignment explain this phenomenon: such inter-relations lead to the construction of larger systems of equations, which itself feeds into the problem of resolving the recurrence equation associated with a candidate.

Experimental Summary

Our experiments illustrate the feasibility of synthesising invariants and solvable loops using our approach for programs with unsolvable recurrence operators from various domains such as biological systems, probabilistic loops, and classical programs (see Section 5.5). This further motivates the theoretical characterisation of unsolvable operators in terms of defective variables (Section 5.4).

5.10 Conclusion

We establish a new technique that synthesises invariants for loops with unsolvable recurrence operators and show its applicability for deterministic and probabilistic programs. Our work is further extended to translate unsolvable loops into solvable ones, by ensuring that the polynomial loop invariants of the solvable loop are invariants of the given unsolvable loop. Our work is based on a new characterisation of unsolvable loops in terms of effective and defective variables: the presence of defective variables is equivalent to unsolvability. In order to generate invariants, we provide an algorithm to isolate the defective program variables and a new method to compute polynomial combinations of defective variables admitting exponential polynomial closed-forms. The implementation of our approach in the tool Polar and our experimental evaluation demonstrate the usefulness of our alternative characterisation of unsolvable loops and the applicability of our invariant synthesis technique to systems from various domains.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Automated Termination Analysis

This chapter is based on the following publication [MBKK21a]:

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. Automated Termination Analysis of Polynomial Probabilistic Programs. In Proc. of ESOP, 2021.

6.1 Problem Statement

Classical program termination. Termination is a key property in program analysis [CPR11]. The question whether a program terminates on all possible inputs – the universal halting problem – is undecidable. Proof rules based on ranking functions have been developed that impose sufficient conditions implying (non-)termination. Automated termination checking has given rise to powerful software tools such as AProVE [GAB+17] and NaTT [YKS14] (using term rewriting), and UltimateAutomizer [HCD+18] (using automata theory). These tools have shown to be able to determine the termination of several intricate programs. The industrial tool Terminator [CPR06] has taken termination proving into practice and is able to prove termination – or even more general liveness properties – of e.g., device driver software. Rather than seeking a single ranking function, it takes a disjunctive termination argument using sets of ranking functions. Other results include termination proving methods for specific program classes such as linear and polynomial programs, see, e.g., [BMS05, HFG20].

Termination of probabilistic program. Probabilistic programs extend sequential programs with the ability to draw samples from probability distributions. They are used e.g. for, encoding randomized algorithms, planning in AI, security mechanisms, and in cognitive science. In this chapter, we consider probabilistic while-programs with discrete probabilistic choices, in the vein of the seminal works [Koz81] and [MM05].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
x ← 10
while x > 0 do
    x ← x + 1 {1/2} x − 1
end while
```
(a)

```
x ← 10
while x > 0 do
    x ← x − 1 {1/2} x + 2
end while
```
(b)

```
x, y ← 0, 0
while x^2 + y^2 < 100 do
    x ← x + 1 {1/2} x − 1
    y ← y + x {1/2} y − x
end while
```
(c)

```
x, y ← 10, 0
while x > 0 do
    y ← y + 1
    x ← x + 4y {1/2} x − y^2
end while
```
(d)

Figure 6.1: Examples of probabilistic programs in our probabilistic language. Program 6.1a is a symmetric 1D random walk. The program is AST but not PAST. Program 6.1b is not AST. Programs 6.1c and 6.1d contain dependent variable updates with polynomial guards and both programs are PAST.

Termination of probabilistic programs differs from the classical halting problem in several respects, e.g., probabilistic programs may exhibit diverging runs that have probability mass zero in total. Such programs do not always terminate, but terminate with probability one – they are *almost surely terminating* (AST). An example of such a program is given in Figure 6.1a where variable x is incremented by 1 with probability $1/2$, and otherwise decremented with this amount. This program encodes a one-dimensional (1D) left-bounded random walk starting at position 10.

Another important difference to classical termination is that the expected number of program steps until termination may be infinite, even if the program almost surely terminates. Thus, almost sure termination (AST) does not imply that the expected number of steps until termination is finite. Programs that have a finite expected runtime are referred to as *positively almost surely terminating* (PAST). Figure 6.1c is a sample program that is PAST. While PAST implies AST, the converse does not hold, as evidenced by Figure 6.1a: the program of Figure 6.1a terminates with probability one but needs infinitely many steps on average to reach $x{=}0$, hence is not PAST. (The terminology AST and PAST was coined in [BG05] and has its roots in the theory of Markov processes.)

Proof rules for AST and PAST. Proving termination of probabilistic programs is hard: AST for a single input is as hard as the universal halting problem, whereas PAST is even harder [KK15]. Termination analysis of probabilistic programs is currently attracting quite some attention. It is not just of theoretical interest. For instance, a popular way to analyze probabilistic programs in machine learning is by using some advanced form of simulation. If, however, a program is not PAST, the simulation may take forever. In addition, the use of probabilistic programs in safety-critical environments [ARS13, BBR+15, FDG+19] necessitates providing formal guarantees on termination.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Different techniques are considered for probabilistic program termination ranging from probabilistic term rewriting [ALY20], sized types [DG19], and Büchi automata theory [CH20], to weakest pre-condition calculi for checking PAST [KKMO18]. A large body of works considers *proof rules* that provide sufficient conditions for proving AST, PAST, or their negations. These rules are based on martingale theory, in particular supermartingales. They are stochastic processes that can be (phrased in a simplified manner) viewed as the probabilistic analog of ranking functions: the value of a random variable represents the "value" of the function at the beginning of a loop iteration. Successive random variables model the evolution of the program loop.

Being a supermartingale means that the expected value of the random variables at the end of a loop does not exceed its value at the start of the loop. Constraints on supermartingales form the essential part of proof rules. For example, the AST proof rule in [MMKK18] requires the existence of a supermartingale whose value decreases at least with a certain amount by at least a certain probability on each loop iteration. Intuitively speaking, the closer the supermartingales comes to zero – indicating termination – the more probable it is that it increases more. The AST proof rule in [MMKK18] is applicable to prove AST for the program in Figure 6.1a; yet, it cannot be used to prove PAST of Figures 6.1c-6.1d. On the other hand, the PAST proof rule in [CS13, FFH15] requires that the expected decrease of the supermartingale on each loop iteration is at least some positive constant ϵ and on loop termination needs to be at most zero – very similar to the usual constraint on ranking functions. While [CS13, FFH15] can be used to prove the program in Figure 6.1c to be PAST, these works cannot be used for Figure 6.1a. They cannot be used for proving Figure 6.1d to be PAST either.

The rule for showing non-AST [CNZ17] requires the supermartingale to be repulsing. This intuitively means that the supermartingale decreases on average with at least ε and is positive on termination. Figuratively speaking, it repulses terminating states. It can be used to prove the program in Figure 6.1b to be not AST. In summary, while existing works for proving AST, PAST, and their negations are generic in nature, they are also restricted for classes of probabilistic programs. *We propose relaxed versions of existing proof rules for probabilistic termination that turn out to treat quite a number of programs that could not be proven otherwise (Section 6.4).* In particular, (non-)termination of all four programs of Figure 6.1 can be proven using our proof rules.

Automated termination checking of AST and PAST. Whereas there is a large body of techniques and proof rules, software tool support to automate checking termination of probabilistic programs is still in its infancy. *We present novel algorithms to automate various proof rules for probabilistic programs:* the three aforementioned proof rules [CS13, FFH15, MMKK18, CNZ17] and a variant of the non-AST proof rule to prove non-PAST [CNZ17][1]. We also present relaxed versions of each of the proof rules, going beyond the state-of-the-art in the termination analysis of probabilistic programs.

[1]For automation, the proof rule of [MMKK18] is considered for constant decrease and probability functions.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We focus on so-called Prob-solvable loops, extending [BKS19]. Namely, we define Prob-solvable loops as probabilistic while-programs whose guards compare two polynomials (over program variables) and whose body is a sequence of random assignments with polynomials as right-hand side such that a variable x, say, only depends on variables preceding x in the loop body. While restrictive, Prob-solvable loops cover a vast set of interesting probabilistic programs (see Remark 8).

An essential property of our programs is that the statistical moments of program variables can be obtained as closed-form formulas [BKS19]. *The key of our algorithmic approach is a procedure for computing asymptotic lower, upper and absolute bounds on polynomial expressions over program variables in our programs (Section 6.5).* This enables a novel method for automating probabilistic termination and non-termination proof rules based on (super)martingales, going beyond the state-of-the-art in probabilistic termination. Our relaxed proof rules allow us to fully automate (P)AST analysis by using only polynomial witnesses. Our experiments provide practical evidence that polynomial witnesses within Prob-solvable loops are sufficient to certify most examples from the literature and even beyond (Section 6.6).

Our termination tool Amber. We have implemented our algorithmic approach in the publicly available tool Amber. It exploits asymptotic bounds over polynomial martingales and uses the tool mora [BKS19] for computing the first-order moments of program variables and the computer algebra system package `diofant`. It employs over- and under-approximations realized by a simple static analysis. *Amber establishes probabilistic termination in a fully automated manner* and has the following unique characteristics:

- it includes the first implementation of the AST proof rule of [MMKK18], and
- it is the first tool capable of certifying AST for programs that are not PAST and cannot be split into PAST subprograms, and
- it is the first tool that brings the various proof rules under a single umbrella: AST, PAST, non-AST and non-PAST.

An experimental evaluation on various benchmarks shows that: (1) Amber is superior to existing tools for automating PAST [NCH18] and AST [CS13], (2) the relaxed proof rules enable proving substantially more programs, and (3) Amber is able to automate the termination checking of intricate probabilistic programs (within the class of programs considered) that could not be automatically handled so far (Section 6.6). For example, *Amber solves 23 termination benchmarks that no other automated approach could so far handle.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Main contributions. To summarize, the main contributions are:

1. Relaxed proof rules for (non-)termination, enabling treating a wider class of programs (Section 6.4).
2. Efficient algorithms to compute asymptotic bounds on polynomial expressions of program variables (Section 6.5).
3. Automation: a realisation of our algorithms in the tool Amber (Section 6.6).
4. Experiments showing the superiority of Amber over existing tools for proving (P)AST (Section 6.6).

6.2 Preliminaries

We denote by $\mathbb{N}$ and $\mathbb{R}$ the set of natural and real numbers, respectively. Further, let $\overline{\mathbb{R}}$ denote $\mathbb{R} \cup \{+\infty, -\infty\}$, $\mathbb{R}_0^+$ the non-negative reals and $\mathbb{R}[x_1, \dots, x_m]$ the polynomial ring in $x_1, \dots, x_m$ over $\mathbb{R}$. We write $x \leftarrow E_{(1)}\ \{p_1\}\ E_{(2)}\ \{p_2\} \dots \{p_{m-1}\}\ E_{(m)}$ for the probabilistic update of program variable x, denoting the execution of $x \leftarrow E_{(j)}$ with probability p_j, for $j = 1, \dots, m-1$, and the execution of $x \leftarrow E_{(m)}$ with probability $1 - \sum_{j=1}^{m-1} p_j$, where $m \in \mathbb{N}$. We write indices of expressions over program variables in round brackets and use E_i for the stochastic process induced by expression E. This section introduces our programming language extending *Prob-solvable loops* [BKS19] and defines the probability space introduced by such programs. We assume the reader to be familiar with probability theory [KSK76].

6.2.1 Programming Model: Prob-Solvable Loops

Prob-solvable loops [BKS19] are syntactically restricted probabilistic programs with polynomial expressions over program variables. The statistical higher-order moments of program variables, like expectation and variance of such loops, can always be computed as functions of the loop counter. We extend Prob-solvable loops with polynomial loop guards in order to study their termination behavior, as follows.

Definition 39 (Prob-solvable loop $\mathcal{L}$)**.** *A* Prob-solvable loop $\mathcal{L}$ *with real-valued variables* $x_{(1)}, \dots, x_{(m)}$*, where* $m \in \mathbb{N}$*, is a program of the form:* $\mathcal{I}_\mathcal{L}$ *while* $\mathcal{G}_\mathcal{L}$ *do* $\mathcal{U}_\mathcal{L}$ *end, with*

- *(Init)* $\mathcal{I}_\mathcal{L}$ *is a sequence* $x_{(1)} \leftarrow r_{(1)}, \dots, x_{(m)} \leftarrow r_{(m)}$ *of* m *assignments, with* $r_{(j)} \in \mathbb{R}$
- *(Guard)* $\mathcal{G}_\mathcal{L}$ *is a strict inequality* $P > Q$*, where* $P,\ Q \in \mathbb{R}[x_{(1)}, \dots, x_{(m)}]$
- *(Update)* $\mathcal{U}_\mathcal{L}$ *is a sequence of* m *probabilistic updates of the form*

  $$x_{(j)} \leftarrow a_{(j1)}x_{(j)} + P_{(j1)}\ \{p_{j1}\}\ a_{(j2)}x_{(j)} + P_{(j2)}\ \{p_{j2}\}\ \dots\ \{p_{j(l_j-1)}\}\ a_{(jl_j)}x_{(j)} + P_{(jl_j)},$$

  where $a_{(jk)} \in \mathbb{R}_0^+$ *are constants,* $P_{(jk)} \in \mathbb{R}[x_{(1)}, \dots, x_{(j-1)}]$ *are polynomials,* $p_{(jk)} \in [0,1]$ *and* $\sum_k\ p_{jk} < 1$*.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

If $\mathcal{L}$ is clear from the context, the subscript $\mathcal{L}$ is omitted from $\mathcal{I}_{\mathcal{L}}$, $\mathcal{G}_{\mathcal{L}}$, and $\mathcal{U}_{\mathcal{L}}$. Figure 6.1 gives four example Prob-solvable loops.

Remark 8 (Prob-solvable expressiveness)**.** *The enforced order of assignments in the loop body of Prob-solvable loops seems restrictive. Notwithstanding these syntactic restrictions, many non-trivial probabilistic programs can be naturally modeled as succinct Prob-solvable loops. These include complex stochastic processes such as 2D random walks and dynamic Bayesian networks [BKS20b]. Almost all existing benchmarks on automated probabilistic termination analysis fall within the scope of Prob-solvable loops (cf. Section 6.6).*

In the sequel, we consider an arbitrary Prob-solvable loop $\mathcal{L}$ and provide all definitions relative to $\mathcal{L}$. The semantics of $\mathcal{L}$ is defined next, by associating $\mathcal{L}$ with a probability space.

6.2.2 Canonical Probability Space

A probabilistic program, and thus a Prob-solvable loop, can be semantically described as a probabilistic transition system [CS13] or as a probabilistic control flow graph [CNZ17], which in turn induce an infinite Markov chain (MC) [2]. An MC is associated with a *sequence space* [KSK76], a special probability space. In the sequel, we associate $\mathcal{L}$ with the sequence space of its corresponding MC, similarly as in [HKGK20]. To this end, we first define the notions *state* and *run* for a Prob-solvable loop.

Definition 40 (State, Run of $\mathcal{L}$)**.** *The* state *of Prob-solvable loop $\mathcal{L}$ over m variables, is a vector $s \in \mathbb{R}^m$. Let $s[j]$ or $s[x_{(j)}]$ denote the j-th component of s representing the value of the variable $x_{(j)}$ in state s. A* run *ϑ of $\mathcal{L}$ is an infinite sequence of states.*

Note that any infinite sequence of states is a run. Infeasible runs will however be assigned measure 0. We write $s \models B$ to denote that the logical formula B holds in state s. A probability space $(\Omega, \Sigma, \mathbb{P})$ consists of a measurable space (Ω, Σ) and a probability measure $\mathbb{P}$ for this space. First, we define a measurable space for $\mathcal{L}$ and later equip it with a probability measure.

Definition 41 (Loop Space of $\mathcal{L}$)**.** *The Prob-solvable loop $\mathcal{L}$ induces a canonical measurable space $(\Omega^{\mathcal{L}}, \Sigma^{\mathcal{L}})$, called* loop space*, where*

- *the sample space $\Omega^{\mathcal{L}} := (\mathbb{R}^m)^\omega$ is the set of all program runs,*
- *the σ-algebra $\Sigma^{\mathcal{L}}$ is the smallest σ-algebra containing all cylinder sets $Cyl(\pi) := \{\pi\vartheta \mid \vartheta \in (\mathbb{R}^m)^\omega\}$ for all finite prefixes $\pi \in (\mathbb{R}^m)^+$, that is $\Sigma^L := \langle\{Cyl(\pi) \mid \pi \in (\mathbb{R}^m)^+\}\rangle_\sigma$.*

[2] In fact, [CNZ17] consider Markov decision processes, but in absence of non-determinism in Prob-solvable loops, Markov chains suffice for our purpose.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

To turn the loop space of $\mathcal{L}$ into a proper probability space, we introduce a probability measure. To this end, we define the probability $p(\pi)$ of a finite non-empty prefix π of a program run. Let $\mu_{\mathcal{I}}(s)$ denote the probability that, after initialization $\mathcal{I}_{\mathcal{L}}$, the loop $\mathcal{L}$ is in state s. Because probabilistic constructs are not allowed in $\mathcal{I}_{\mathcal{L}}$, $\mu_{\mathcal{I}}(s)$ is a Dirac-distribution, such that $\mu_{\mathcal{I}}(s) = 1$ for the unique state s defined by $\mathcal{I}_{\mathcal{L}}$ and $\mu_{\mathcal{I}}(s') = 0$ for $s' \neq s$. Moreover, $\mu_{\mathcal{U}}(s, s')$ denotes the probability that, after one loop iteration starting in state s, the resulting program state is s'. Note that $\mu_{\mathcal{I}}(s)$ and $\mu_{\mathcal{U}}(s, s')$ are solely determined by $\mathcal{I}_{\mathcal{L}}$ and $\mathcal{U}_{\mathcal{L}}$. The probability $p(\pi)$ of a finite non-empty prefix π of a program run is then defined as

$$p(s) := \mu_{\mathcal{I}}(s), \quad p(\pi s s') := \begin{cases} p(\pi s) \cdot [s' = s], & \text{if } s \vDash \neg \mathcal{G}_{\mathcal{L}} \\ p(\pi s) \cdot \mu_{\mathcal{U}}(s, s'), & \text{if } s \vDash \mathcal{G}_{\mathcal{L}} \end{cases}$$

where $[\dots]$ denote the Iverson brackets, i.e. $[s' = s]$ is 1 iff $s' = s$. Intuitively, $p(\pi)$ is the probability that prefix π is the sequence of the first $|\pi|$ program states when executing $\mathcal{L}$. We note that the effect of the loop body $\mathcal{U}$ is considered as atomic.

Definition 42 (Loop Measure of $\mathcal{L}$)**.** *The* loop measure *of a Prob-solvable loop $\mathcal{L}$ is a canonical probability measure $\mathbb{P}^{\mathcal{L}} : \Sigma^{\mathcal{L}} \to [0, 1]$ on the loop space of $\mathcal{L}$, with $\mathbb{P}^{\mathcal{L}}(Cyl(\pi)) := p(\pi)$.*

The loop space and the loop measure of $\mathcal{L}$ form the probability space $(\Omega^{\mathcal{L}}, \Sigma^{\mathcal{L}}, \mathbb{P}^{\mathcal{L}})$.

6.2.3 Probabilistic Termination

In order to formalize termination properties of a Prob-solvable loop $\mathcal{L}$, we define the *looping time* of $\mathcal{L}$ to be a random variable in $\mathcal{L}$'s loop space. A random variable X in a probability space $(\Omega, \Sigma, \mathbb{P})$ is a (Σ-)measurable function $X : \Omega \to \overline{\mathbb{R}}$, i.e. for every open interval $U \subseteq \overline{\mathbb{R}}$ it holds that $X^{-1}(U) \in \Sigma$. The expected value of a random variable X, denoted by $\mathbb{E}(X)$, is defined as the Lebesgue integral of X over the probability space, i.e. $\mathbb{E}(X) := \int_{\Omega} X d\mathbb{P}$. In the special case that X takes only countably many values, we have $\mathbb{E}(X) = \int_{\Omega} X d\mathbb{P} = \sum_{r \in X(\Omega)} \mathbb{P}(X = r) \cdot r$. We now define the *looping time* of a Prob-solvable loop $\mathcal{L}$, as follows.

Definition 43 (Looping Time of $\mathcal{L}$)**.** *The* looping time *of $\mathcal{L}$ is the random variable $T^{\neg\mathcal{G}} : \Omega \to \mathbb{N} \cup \{\infty\}$, where $T^{\neg\mathcal{G}}(\vartheta) := \inf\{i \in \mathbb{N} \mid \vartheta_i \vDash \neg\mathcal{G}\}$.*

Intuitively, the looping time $T^{\neg\mathcal{G}}$ maps a program run of $\mathcal{L}$ to the index of the first state falsifying the loop guard $\mathcal{G}$ of $\mathcal{L}$ or to ∞ if no such state exists. We now formalize termination properties of $\mathcal{L}$ using the looping time $T^{\neg\mathcal{G}}$.

Definition 44 (Termination of $\mathcal{L}$)**.** *The Prob-solvable loop $\mathcal{L}$ is* AST *if $\mathbb{P}(T^{\neg\mathcal{G}} < \infty) = 1$. $\mathcal{L}$ is PAST if $\mathbb{E}(T^{\neg\mathcal{G}}) < \infty$.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2.4 Filtrations and Martingales

For a thorough analysis of the hardness of deciding AST and PAST we refer to [KK15]. While for arbitrary probabilistic programs, answering $\mathbb{P}(T^{\neg\mathcal{G}} < \infty)$ and $\mathbb{E}(T^{\neg\mathcal{G}} < \infty)$ is undecidable, sufficient conditions for AST, PAST and their negations have been developed [CS13, FFH15, MMKK18, CNZ17]. These works use (super)martingales which are special stochastic processes. In this section, we adopt the general setting of martingale theory to a Prob-solvable loop $\mathcal{L}$ and then formalize sufficient termination conditions for $\mathcal{L}$ in Section 6.3.

Definition 45 (Stochastic Process of $\mathcal{L}$)**.** *A stochastic process $(X_i)_{i\in\mathbb{N}}$ is a sequence of random variables. Every arithmetic expression E over the program variables of $\mathcal{L}$ induces the stochastic process $(E_i)_{i\in\mathbb{N}}$, $E_i : \Omega \to \mathbb{R}$ with $E_i(\vartheta) := E(\vartheta_i)$. For a run ϑ of $\mathcal{L}$, $E_i(\vartheta)$ is the evaluation of E in the i-th state of ϑ.*

In the sequel, for a boolean condition B over program variables x of $\mathcal{L}$, we write B_i to refer to the result of substituting x by x_i in B. In Figure 6.1a, the stochastic process $(x_i)_{i\in\mathbb{N}}$ is such that every x_i maps a given program run ϑ to the value of the variable x in the i-th state of ϑ. Note that the σ-algebra $\Sigma^{\mathcal{L}}$ contains the cylinder sets for finite program run prefixes of *arbitrary* length. This does not capture the gradual information gain when executing $\mathcal{L}$ iteration by iteration. In probability theory, *filtrations* are a standard notion to formalize the information available at a specific point in time.

Definition 46 (Filtration [KSK76])**.** *For a probability space $(\Omega, \Sigma, \mathbb{P})$, a* filtration *is a sequence $(\mathcal{F}_i)_{i\in\mathbb{N}}$ such that (1) every $\mathcal{F}_i$ is a sub-σ-algebra and (2) $\mathcal{F}_i \subseteq \mathcal{F}_{i+1}$. Further, $(\Omega, \Sigma, (\mathcal{F}_i)_{i\in\mathbb{N}}, \mathbb{P})$ is called a* filtered probability space*.*

We adopt filtrations to Prob-solvable loops and enrich the loop space of $\mathcal{L}$ to a filtered probability space, as follows.

Definition 47 (Loop Filtration of $\mathcal{L}$)**.** *The* loop filtration *$(\mathcal{F}_i^{\mathcal{L}})_{i\in\mathbb{N}}$ of $\Sigma^{\mathcal{L}}$ is defined by $\mathcal{F}_i^{\mathcal{L}} = \langle\{Cyl(\pi) \mid \pi \in (\mathbb{R}^m)^+,\ |\pi| = i{+}1\}\rangle_\sigma$. $(\Omega^{\mathcal{L}}, \Sigma^{\mathcal{L}}, (\mathcal{F}^{\mathcal{L}})_{i\in\mathbb{N}}, \mathbb{P}^{\mathcal{L}})$ is a* filtered probability space *of $\mathcal{L}$.*

Based on Definition 47, note that $\mathcal{F}_0^{\mathcal{L}}$ is the smallest σ-algebra containing the cylinder sets of finite prefixes of program runs of length 1. That is, the cylinder sets of finite prefixes of program runs of length greater than or equal to 2 are not present in $\mathcal{F}_0^{\mathcal{L}}$. Hence, $\mathcal{F}_0^{\mathcal{L}}$ captures exactly the information available about the program run after executing just the initialization $\mathcal{I}_{\mathcal{L}}$. Similarly, $\mathcal{F}_i^{\mathcal{L}}$ captures the information about the program run after the loop body $\mathcal{U}_{\mathcal{L}}$ has been executed i times. In Figure 6.1a, for example, the event $\{\vartheta \in \Omega \mid x_i(\vartheta) = r\}$ denoted by $\{x_i = r\}$ is $\mathcal{F}_i^{\mathcal{L}}$-measurable for every $i \in \mathbb{N}$ and every $r \in \mathbb{R}$, as the value of x_i depends only on information available up to the i-th iteration of the loop body of Figure 6.1a. The following definition formalizes this observation.

Definition 48 (Adapted Process [KSK76])**.** *A stochastic process $(X_i)_{i\in\mathbb{N}}$ is said to be* adapted *to a filtration $(\mathcal{F}_i)_{i\in\mathbb{N}}$ if X_i is $\mathcal{F}_i$-measurable for every $i \in \mathbb{N}$.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

It is not hard to argue that, for any arithmetic expression E over the variables of $\mathcal{L}$, the induced stochastic process $(E_i)_{i\in\mathbb{N}}$ is adapted to the loop filtration $\mathcal{F}_i^{\mathcal{L}}$ of $\mathcal{L}$: the value of E_i only depends on the information available up to the i-th loop iteration of $\mathcal{L}$.

The concept of (super)martingales builds upon the notion of *conditional expected values* which is defined as follows.

Definition 49 (Conditional Expected Value [KSK76])**.** *For a probability space* $(\Omega, \Sigma, \mathbb{P})$*, an integrable random variable* X *and a sub-*σ*-algebra* $\Delta \subseteq \Sigma$*, the* expected value of X conditioned on Δ, $\mathbb{E}(X \mid \Delta)$*, is any* Δ*-measurable function such that for every* $D \in \Delta$ *we have* $\int_D \mathbb{E}(X \mid \Delta) d\mathbb{P} = \int_D X d\mathbb{P}$*. The random variable* $\mathbb{E}(X \mid \Delta)$ *is almost surely unique.*

We now introduce (super)martingales as special stochastic processes. In Section 6.3 these notions are used to define sufficient conditions for PAST, AST and their negations.

Definition 50 (Martingales)**.** *Let* $(\Omega, \Sigma, (\mathcal{F}_i)_{i\in\mathbb{N}}, \mathbb{P})$ *be a filtered probability space and* $(M_i)_{i\in\mathbb{N}}$ *be an integrable stochastic process adapted to* $(\mathcal{F}_i)_{i\in\mathbb{N}}$*. Then* $(M_i)_{i\in\mathbb{N}}$ *is a* martingale *if* $\mathbb{E}(M_{i+1} \mid \mathcal{F}_i) = M_i$ *(or equivalently* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = 0$*). Moreover,* $(M_i)_{i\in\mathbb{N}}$ *is called a* supermartingale (SM) *if* $\mathbb{E}(M_{i+1} \mid \mathcal{F}_i) \leq M_i$ *(or equivalently* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq 0$*). For an arithmetic expression* E *over the program variables of* $\mathcal{L}$*, the conditional expected value* $\mathbb{E}(E_{i+1} - E_i \mid \mathcal{F}_i)$ *is called* the martingale expression of E.

6.3 Proof Rules for Probabilistic Termination

While AST and PAST are undecidable in general [KK15], sufficient conditions, called *proof rules*, for AST and PAST have been introduced, see e.g. [CS13, FFH15, MMKK18, CNZ17]. In this section, we survey four proof rules, adapted to Prob-solvable loops. In the sequel, a *pure invariant* is a loop invariant in the classical deterministic sense [Hoa69]. Based on the probability space corresponding to $\mathcal{L}$, a pure invariant holds before and after every iteration of $\mathcal{L}$.

6.3.1 Positive Almost Sure Termination (PAST)

The proof rule for PAST introduced in [CS13] relies on the notion of ranking supermartingales (RSMs), which is a SM that decreases by a fixed positive ϵ on average at every loop iteration. Intuitively, RSMs resemble ranking functions for deterministic programs, yet for probabilistic programs.

Theorem 31 (Ranking-Supermartingale-Rule (RSM-Rule) [CS13], [FFH15])**.** *Let* $M : \mathbb{R}^m \to \mathbb{R}$ *be an expression over the program variables of* $\mathcal{L}$ *and* I *a pure invariant of* $\mathcal{L}$*. Assume the following conditions hold for all* $i \in \mathbb{N}$*:*

1. *(Termination)* $\mathcal{G} \land I \implies M > 0$
2. *(RSM Condition)* $\mathcal{G}_i \land I_i \implies \mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq -\epsilon$*, for some* $\epsilon > 0$*.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Then, $\mathcal{L}$ is PAST. Further, M is called an ϵ-ranking supermartingale.

Example 51. *Consider Figure 6.1c, set $M := 100 - x^2 - y^2$ and $\epsilon := 2$ and let I be true. Condition (1) of Theorem 31 trivially holds. Further, M is also an ϵ-ranking supermartingale, as $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = 100 - \mathbb{E}(x_{i+1}^2 \mid \mathcal{F}_i) - \mathbb{E}(y_{i+1}^2 \mid \mathcal{F}_i) - 100 + x_i^2 + y_i^2 = -2 - x_i^2 \leq -2$. That is because $\mathbb{E}(x_{i+1}^2 \mid \mathcal{F}_i) = x_i^2 + 1$ and $\mathbb{E}(y_{i+1}^2 \mid \mathcal{F}_i) = y_i^2 + x_i^2 + 1$. Figure 6.1c is thus proved PAST using the RSM-Rule.*

6.3.2 Almost Sure Termination (AST)

Recall that Figure 6.1a is AST but not PAST, and hence the RSM-rule cannot be used for Figure 6.1a. By relaxing the ranking conditions, the proof rule in [MMKK18] uses general supermartingales to prove AST of programs that are not necessarily PAST.

Theorem 32 (Supermartingale-Rule (SM-Rule) [MMKK18])**.** *Let $M : \mathbb{R}^m \to \mathbb{R}_{\geq 0}$ be an expression over the program variables of $\mathcal{L}$ and I a pure invariant of $\mathcal{L}$. Let $p : \mathbb{R}_{\geq 0} \to (0, 1]$ (for* probability*) and $d : \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$ (for* decrease*) be antitone (i.e. monotonically decreasing) functions. Assume the following conditions hold for all $i \in \mathbb{N}$:*

1. *(Termination) $\mathcal{G} \wedge I \implies M > 0$*
2. *(Decrease) $\mathcal{G}_i \wedge I_i \implies \mathbb{P}(M_{i+1} - M_i \leq -d(M_i) \mid \mathcal{F}_i) \geq p(M_i)$*
3. *(SM Condition) $\mathcal{G}_i \wedge I_i \implies \mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq 0$.*

Then, $\mathcal{L}$ is AST.

Intuitively, the requirement of d and p being antitone forbids that the "execution progress" of $\mathcal{L}$ towards termination becomes infinitely small while still being positive.

Example 52. *The SM-Rule can be used to prove AST for Figure 6.1a. Consider $M := x$, $p := 1/2$, $d := 1$ and $I := true$. Clearly, p and d are antitone. The remaining conditions of Theorem 32 also hold as* (1) $x > 0 \implies x > 0$; (2) *x decreases by d with probability p in every iteration; and* (3) $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = x_i - x_i \leq 0$.

6.3.3 Non-Termination

While Theorems 31 and 32 can be used for proving AST and PAST, respectively, they are not applicable to the analysis of non-terminating Prob-solvable loops. Two sufficient conditions for certifying the negations of AST and PAST have been introduced in [CNZ17] using so-called *repulsing-supermartingales*. Intuitively, a *repulsing-supermartingale M* on average decreases in every iteration of $\mathcal{L}$ and on termination is non-negative. Figuratively, M repulses terminating states.

Theorem 33 (Repulsing-AST-Rule (R-AST-Rule) [CNZ17])**.** *Let $M : \mathbb{R}^m \to \mathbb{R}$ be an expression over the program variables of $\mathcal{L}$ and I a pure invariant of $\mathcal{L}$. Assume the following conditions hold for all $i \in \mathbb{N}$:*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. *(Negative)* $M_0 < 0$
2. *(Non-Termination)* $\neg\mathcal{G} \wedge I \implies M \geq 0$
3. *(RSM Condition)* $\mathcal{G}_i \wedge I_i \implies \mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq -\epsilon$*, for some* $\epsilon > 0$
4. *(c-Bounded Differences)* $|M_{i+1} - M_i| < c$*, for some* $c > 0$*.*

Then, $\mathcal{L}$ *is* not *AST.* M *is called an* ϵ-repulsing supermartingale with c-bounded differences*.*

Example 53. *Consider Figure 6.1b and let* $M := -x$*,* $c := 3$*,* $\epsilon := 1/2$ *and* $I := true$*. All four above conditions hold: (1)* $-x_0 = -10 < 0$*; (2)* $x \leq 0 \implies -x \geq 0$*; (3)* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = -x_i - 1/2 + x_i = -1/2 \leq -\epsilon$*; and (4)* $|x_i - x_{i+1}| < 3$*. Thus, Figure 6.1b is not AST.*

While Theorem 33 can prove programs not to be AST, and thus also not PAST, it cannot be used to prove programs not to be PAST when they are AST. For example, Theorem 33 cannot be used to prove that Figure 6.1a is not PAST. To address such cases, a variation of the R-AST-Rule [CNZ17] for certifying programs not to be PAST arises by relaxing the condition $\epsilon > 0$ of the R-AST-Rule to $\epsilon \geq 0$. We refer to this variation by *Repulsing-PAST-Rule (R-PAST-Rule).*

Example 54. *Consider Figure 6.1a. We set* $M := -x$*,* $c := 1$ *and* $\epsilon := 0$*. Note that* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = -x_i + x_i \leq 0$ *and it is easy to see that all four conditions of Theorem 33 hold (with* $\epsilon \geq 0$*). Thus, the R-PAST-Rule proves that Figure 6.1a is not PAST.*

6.4 Relaxed Proof Rules for Probabilistic Termination

While Theorems 31-33 provide sufficient conditions proving PAST, AST and their negations, the applicability to Prob-solvable loops is somewhat restricted. For example, the RSM-Rule cannot be used to prove Figure 6.1d to be PAST using the simple expression $M := x$, as explained in detail with Example 55, but may require more complex witnesses for certifying PAST, complicating automation. In this section, we relax the conditions of Theorems 31-33 by requiring these conditions to only hold "eventually". A property $P(i)$ parameterized by a natural number $i \in \mathbb{N}$ *holds eventually* if there is an $i_0 \in \mathbb{N}$ such that $P(i)$ holds for all $i \geq i_0$. Our relaxations of probabilistic termination proof rules can intuitively be described as follows: If $\mathcal{L}$, after a fixed number of steps, almost surely reaches a state from which the program is PAST or AST, then the program is PAST or AST, respectively. Let us first illustrate the benefits of reasoning with "eventually" holding properties for probabilistic termination in the following example.

Example 55 (Limits of the RSM-Rule and SM-Rule)**.** *Consider Figure 6.1d. Setting* $M := x$*, we have the martingale expression* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = -y_i^2/2 + y_i + 3/2 =$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$x, y \leftarrow x_0, 0$
while $x > 0$ **do**
$y \leftarrow y + 1$
$x \leftarrow x + (y - 5)\ \{1/2\}\ x - (y - 5)$
end while

(a)

$x, y \leftarrow 1, 2$
while $x > 0$ **do**
$y \leftarrow 1/2 \cdot y$
$x \leftarrow x + 1 - y\ \{2/3\}\ x - 1 + y$
end while

(b)

Figure 6.2: Prob-solvable loops which require our relaxed proof rules for termination analysis.

$-i^2/2 + i + 3/2$. *Since* $\mathbb{E}(x_{i+1} - x_i \mid \mathcal{F}_i)$ *is non-negative for* $i \in \{0, 1, 2, 3\}$*, we conclude that* M *is not an RSM. However, Figure 6.1d either terminates within the first three iterations or, after three loop iterations, is in a state such that the RSM-Rule is applicable. Therefore, Figure 6.1d is PAST but the RSM-Rule cannot directly prove using* $M := x$*. A similar restriction of the SM-Rule can be observed for Figure 6.2a. By considering* $M := x$*, we derive the martingale expression* $\mathbb{E}(x_{i+1} - x_i \mid \mathcal{F}_i) = 0$*, implying that* M *is a martingale for Figure 6.2a. However, the decrease function* d *for the SM-Rule cannot be defined because, for example, in the fifth loop iteration of Figure 6.2a, there is no progress as* x *is almost surely updated with its previous value. However, after the fifth iteration of Figure 6.2a,* x *always decreases by at least* 1 *with probability* $1/2$ *and all conditions of the SM-Rule are satisfied. Thus, Figure 6.2a either terminates within the first five iterations or reaches a state from which it terminates almost surely. Consequently, Figure 6.2a is AST but the SM-Rule cannot directly prove it using* $M := x$*.*

We therefore relax the RSM-Rule and SM-Rule of Theorems 31 and 32 as follows.

Theorem 34 (Relaxed Termination Proof Rules)**.** *For the RSM-Rule to certify PAST of* $\mathcal{L}$*, it is sufficient that conditions* (1)-(2) *of Theorem 31 hold eventually (instead of for all* $i \in \mathbb{N}$*). Similarly, for the SM-Rule to certify AST of* $\mathcal{L}$*, it is sufficient that conditions* (1)-(3) *of Theorem 32 hold eventually.*

Proof. We prove the relaxation of the RSM-Rule. The proof of the relaxed SM-Rule is analogous. Let $\mathcal{L} := \mathcal{I}$ *while* $\mathcal{G}$ *do* $\mathcal{U}$ *end* be as in Definition 39. Assume $\mathcal{L}$ satisfies the conditions (1)-(2) of Theorem 31 after some $i_0 \in \mathbb{N}$. We construct the following probabilistic program $\mathcal{P}$, where i is a new variable not appearing in $\mathcal{L}$:

$$\begin{aligned}&\mathcal{I}; i \leftarrow 0\\&\textit{while } i < i_0 \textit{ do } \mathcal{U}; i \leftarrow i + 1 \textit{ end}\\&\textit{while } \mathcal{G} \textit{ do } \mathcal{U} \textit{ end}\end{aligned} \tag{6.1}$$

We first argue that if $\mathcal{P}$ is PAST, then so is $\mathcal{L}$. Assume $\mathcal{P}$ to be PAST. Then, the looping time of $\mathcal{L}$ is either bounded by i_0 or it is PAST, by the definition of $\mathcal{P}$. In both cases, $\mathcal{L}$ is PAST. Finally, observe that $\mathcal{P}$ is PAST if and only if its second while-loop is PAST. However, the second while-loop of $\mathcal{P}$ can be certified to be PAST using the RSM-Rule and additionally using $i \geq i_0$ as an invariant. □

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Remark 9. *The central point of our proof rule relaxations is that they allow for simpler witnesses. While for Example 55 it can be checked that $M := x + 2^{y+5}$ is an RSM, the example illustrates that the relaxed proof rule allows for a much simpler PAST witness (linear instead of exponential). This simplicity is key for automation.*

Similar to Theorem 34, we relax the R-AST-Rule and the R-PAST-Rule. However, compared to Theorem 34, it is not enough for a non-termination proof rule to certify non-AST from some state onward, because $\mathcal{L}$ may never reach this state as it might terminate earlier. Therefore, a necessary assumption when relaxing non-termination proof rules comes with ensuring that $\mathcal{L}$ has a positive probability of reaching the state after which a proof rule witnesses non-termination. This is illustrated in the following example.

Example 56 (Limits of the R-AST-Rule)**.** *Consider Figure 6.2b and set $M := -x$. As a result, we get $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = {}^{y_i}/_6 - {}^1/_3 = {}^{2^{-i}}/_3 - {}^1/_3$. Thus, $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = 0$ for $i = 0$, implying that M cannot be an ϵ-repulsing supermartingale with $\epsilon > 0$ for all $i \in \mathbb{N}$. However, after the first iteration of $\mathcal{L}$, M satisfies all requirements of the R-AST-Rule. Moreover, $\mathcal{L}$ always reaches the second iteration because in the first iteration x almost surely does not change. From this follows that Figure 6.2b is not AST.*

The following theorem formalizes the observation of Example 56 relaxing the R-AST-Rule and R-PAST-Rule of Theorem 33.

Theorem 35 (Relaxed Non-Termination Proof Rules for)**.** *For the R-AST-Rule to certify non-AST for $\mathcal{L}$ (Theorem 33), as well as for the R-PAST-Rule to certify non-PAST for $\mathcal{L}$ (Theorem 33), if $\mathbb{P}(M_{i_0} < 0) > 0$ for some $i_0 \geq 0$, it suffices that conditions* (2)-(4) *hold for all $i \geq i_0$ (instead of for all $i \in \mathbb{N}$).*

Proof. We prove the relaxation of the R-AST-Rule. The proof for the R-PAST-Rule is analogous. Let $\mathcal{L} := \mathcal{I}$ *while* $\mathcal{G}$ *do* $\mathcal{U}$ *end* be as in Definition 39. Assume $\mathcal{L}$ satisfies conditions (2)-(4) of the R-AST-Rule for all $i \geq i_0$ for some fixed $i_0 \in \mathbb{N}$. Moreover, assume $\mathbb{P}(M_{i_0} < 0) > 0$.

We construct again a probabilistic program $\mathcal{P}$ as in (6.1). Observe that for the second while-loop of $\mathcal{P}$, we have $i \geq i_0$. By assumption, the second while-loop of $\mathcal{P}$ satisfies conditions (2)-(4) of the R-AST-Rule. By the R-AST-Rule, we conclude $\mathcal{P}$ being not AST, if there is a $Cyl(\pi) \in \mathcal{F}^{\mathcal{P}}_{i_0}$, such that $\mathbb{P}^{\mathcal{P}}(Cyl(\pi)) > 0$ and $M_{i_0}(\vartheta) < 0$ for all $\vartheta \in Cyl(\pi)$.

By the definition of $\mathcal{P}$, it then follows for $\mathcal{L}$ that if there is a $Cyl(\pi) \in \mathcal{F}^{\mathcal{L}}_{i_0}$, such that $\mathbb{P}^{\mathcal{L}}(Cyl(\pi)) > 0$ and $M_{i_0}(\vartheta) < 0$ for all $\vartheta \in Cyl(\pi)$, then $\mathcal{L}$ is not AST. As $\mathbb{P}^{\mathcal{L}}(M_{i_0} < 0) > 0$, we conclude that such a $Cyl(\pi)$ exists and derive that $\mathcal{L}$ is not AST. □

Note that for a repulsing supermartingale M, the condition $\mathbb{P}(M_{i_0} < 0) > 0$ implies that there is a positive probability of reaching iteration i_0, because M would have to be almost surely non-negative upon termination.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

In what follows, whenever we write RSM-Rule, SM-Rule, R-AST-Rule or R-PAST-Rule we refer to our relaxed versions of the proof rules.

6.5 Algorithmic Termination Analysis through Asymptotic Bounds

The *two major challenges when automating reasoning* with the proof rules of Sections 6.3 and 6.4 are (i) constructing expressions M over the program variables and (ii) proving inequalities involving $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i)$. In this section, we address these two challenges for Prob-solvable loops. For the loop guard $\mathcal{G}_{\mathcal{L}} = P > Q$, let $G_{\mathcal{L}}$ denote the polynomial $P - Q$. As before, if $\mathcal{L}$ is clear from the context, we omit the subscript $\mathcal{L}$. It holds that $G > 0$ is equivalent to $\mathcal{G}$.

(i) Constructing (super)martingales M. For a Prob-solvable loop $\mathcal{L}$, the polynomial G is a natural candidate for the expression M in termination proof rules (RSM-Rule, SM-Rule) and $-G$ in the non-termination proof rules (R-AST-Rule, R-PAST-Rule). Hence, we construct potential (super)martingales M by setting $M := G$ for the RSM-Rule and the SM-Rule, and $M := -G$ for the R-AST-Rule and the R-PAST-Rule. The property $\mathcal{G} \implies G > 0$, a condition of the RSM-Rule and the SM-Rule, trivially holds. Moreover, for the R-AST-Rule and R-PAST-Rule the condition $\neg\mathcal{G} \implies -G \geq 0$ is satisfied. The remaining conditions of the proof rules are:

- RSM-Rule: (a) $\mathcal{G}_i \implies \mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) \leq -\epsilon$ for some $\epsilon > 0$
- SM-Rule: (a) $\mathcal{G}_i \implies \mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) \leq 0$ and (b) $\mathcal{G}_i \implies \mathbb{P}(G_{i+1} - G_i \leq -d \mid \mathcal{F}_i) \geq p$ for some $p \in (0, 1]$ and $d \in \mathbb{R}^+$ (for the purpose of efficient automation, we restrict the functions $d(r)$ and $p(r)$ to be constant)
- R-AST-Rule: (a) $\mathcal{G}_i \implies \mathbb{E}(-G_{i+1}+G_i \mid \mathcal{F}_i) \leq -\epsilon$ for some $\epsilon > 0$ and (b) $|G_{i+1} - G_i| \leq c$, for some $c > 0$.

All these conditions express bounds over G_i. Choosing G as the potential witness may seem simplistic. However, Example 55 already illustrated how our relaxed proof rules can mitigate the need for more complex witnesses (even exponential ones). *The computational effort in our approach does not lie in synthesizing a complex witness but in constructing asymptotic bounds for the loop guard.* Our approach can therefore be seen as complementary to approaches synthesizing more complex witnesses [CS13, CFG16, CNZ17]. The martingale expression $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$ is an expression over program variables, whereas $G_{i+1} - G_i$ cannot be interpreted as a single expression but through a distribution of expressions.

Definition 51 (One-step Distribution)**.** *For expression H over the program variables of Prob-solvable loop $\mathcal{L}$, let the* one-step distribution $\mathcal{U}_{\mathcal{L}}^{H}$ *be defined by $E \mapsto \mathbb{P}(H_{i+1} = E \mid \mathcal{F}_i)$*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

with support set $supp(\mathcal{U}_{\mathcal{L}}^{H}) := \{B \mid \mathcal{U}_{\mathcal{L}}^{H}(B) > 0\}$*. We refer to expressions* $B \in supp(\mathcal{U}_{\mathcal{L}}^{H})$ *by* branches of H*.*

The notation $\mathcal{U}_{\mathcal{L}}^{H}$ is chosen to suggest that the loop body $\mathcal{U}_{\mathcal{L}}$ is "applied" to the expression H, leading to a distribution over expressions. Intuitively, the support $supp(\mathcal{U}_{\mathcal{L}}^{H})$ of an expression H contains all possible updates of H after executing a single iteration of $\mathcal{U}_{\mathcal{L}}$.

Example 57 (One-step Distribution)**.** *Consider the following Prob-solvable loop:*

$x, y \leftarrow 1, 1$
while $x > 0$ ***do***
  $y \leftarrow y + 1 \ \{1/2\} \ y + 2$
  $x \leftarrow x + y \ \{1/3\} \ x - y$
end while

For the expression $H := x^2$*, the one-step distribution* $\mathcal{U}_{\mathcal{L}}^{H}$ *is as follows:*

Expression E	$\mathcal{U}_{\mathcal{L}}^{H}(E)$
$x_i^2 + 2x_iy_i + 2x_i + y_i^2 + 2y_i + 1$	$1/6$
$x_i^2 + 2x_iy_i + 4x_i + y_i^2 + 4y_i + 4$	$1/6$
$x_i^2 - 2x_iy_i - 2x_i + y_i^2 + 2y_i + 1$	$1/3$
$x_i^2 - 2x_iy_i - 4x_i + y_i^2 + 4y_i + 4$	$1/3$
Any other E	0

The first entry in the table can be derived like:

$$\begin{aligned}
x_{i+1}^2 = (x_i + y_{i+1})^2 &= x_i^2 + 2x_iy_{i+1} + y_{i+1}^2 && \textit{(with probability } 1/3\textit{)} \\
&= x_i^2 + 2x_i(y_i + 1) + (y_i + 1)^2 && \textit{(with probability } 1/2 \cdot 1/3\textit{)} \\
&= x_i^2 + 2x_iy_i + 2x_i + y_i^2 + 2y_i + 1 && \textit{(with probability } 1/6\textit{)}
\end{aligned}$$

(ii) Proving inequalities involving $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i)$**.** To automate the termination analysis of $\mathcal{L}$ with the proof rules from Section 6.3, we need to compute bounds for the expression $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$ as well as for the branches of G. In addition, our relaxed proof rules from Section 6.4 only need asymptotic bounds, i.e. bounds which hold eventually. In Section 6.5.2, we propose Algorithm 5 for computing *asymptotic lower and upper bounds* for any polynomial expression over program variables of $\mathcal{L}$. Our procedure allows us to derive bounds for $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$ and the branches of G. Before formalizing our method, let us first illustrate how reasoning with asymptotic bounds helps to apply termination proof rules to $\mathcal{L}$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Example 58 (Asymptotic Bounds for the RSM-Rule). *Consider the following program:*

$x, y \leftarrow 1, 0$
while $x < 100$ ***do***
  $y \leftarrow y + 1$
  $x \leftarrow 2x + y^2 \; \{1/2\} \; 1/2 \cdot x$
end while

Observe $y_i = i$*. The martingale expression for* $G = 100 - x$ *is* $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) = 1/2(100 - 2x_i - (i+1)^2) + 1/2(100 - x_i/2) - (100 - x_i) = -x_i/4 - i^2/2 - i - 1/2$*. Note that if the term* $-x_i/4$ *would not be present in* $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$*, we could certify the program to be PAST using the RSM-Rule because* $-i^2/2 - i - 1/2 \leq -1/2$ *for all* $i \geq 0$*. However, by taking a closer look at the variable* x*, we observe that it is* eventually *and almost surely lower bounded by the function* $\alpha \cdot 2^{-i}$ *for some* $\alpha \in \mathbb{R}^+$*. Therefore,* eventually $-x_i/4 \leq -\beta \cdot 2^{-i}$ *for some* $\beta \in \mathbb{R}^+$*. Thus,* eventually $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) \leq -\gamma \cdot i^2$ *for some* $\gamma \in \mathbb{R}^+$*. By our RSM-Rule, the program is PAST.*

Now, the question arises how the asymptotic lower bound $\alpha \cdot 2^{-i}$ *for* x *can be computed automatically. In every iteration,* x *is either updated with* $2x + y^2$ *or* $1/2 \cdot x$*. Considering the updates as recurrences, we have the inhomogeneous parts* y^2 *and* 0*. Asymptotic lower bounds for these parts are* i^2 *and* 0*, respectively, where* 0 *is the "asymptotically smallest one". Taking* 0 *as the inhomogeneous part, we construct two recurrences: (1)* $l_0 = \alpha$*,* $l_{i+1} = 2l_i + 0$ *and (2)* $l_0 = \alpha$*,* $l_{i+1} = 1/2 \cdot l_i + 0$*, for some* $\alpha \in \mathbb{R}^+$*. Solutions to these recurrences are* $\alpha \cdot 2^i$ *and* $\alpha \cdot 2^{-i}$*, where the last one is the desired lower bound because it is "asymptotically smaller". We will formalize this idea of computing asymptotic bounds in Algorithm 5.*

Example 59 (Bounds & R-AST-Rule). *Consider the following Prob-solvable loop:*

$x, y \leftarrow 1, 2$
while $x > 0$ ***do***
  $y \leftarrow 1/2 \cdot y$
  $x \leftarrow x + 1 - y \; \{2/3\} \; x - 1 + y$
end while

The martingale expression for $-G = -x$ *is* $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) = \frac{y_i}{6} - \frac{1}{3} = \frac{2^{-i}}{3} - \frac{1}{3}$*, using* $y_i = 2^{-i}$*. We observe that almost surely* x_i *is eventually lower bounded (and upper bounded) by a function* $\alpha \cdot i$ *for some* $\alpha \in \mathbb{R}^+$*. Therefore, eventually* $\mathbb{E}(G_i - G_{i+1} \mid \mathcal{F}_i) \leq -\beta$ *holds for some* $\beta \in \mathbb{R}^+$*. Consequently,* $-x$ *is eventually an* ϵ*-repulsing supermartingale. A bound* c *on the differences* $|x_{i+1} - x_i|$ *can be established by a similar style of reasoning to arrive at* $|x_{i+1} - x_i| < \gamma$*, for some* $\gamma \in \mathbb{R}^+$*. Because all the conditions of the R-AST-Rule are satisfied and there is a positive probability of reaching any iteration (necessary for the relaxation of the R-AST-Rule), the program is not AST.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example 60 (Bounds & SM-Rule). *Consider the following Prob-solvable loop:*

$x, y \leftarrow 10, 0$
while $x > 0$ ***do***
  $y \leftarrow y + 1 \; \{1/3\} \; y + 2 \; \{1/3\} \; y + 3$
  $x \leftarrow x + y^2 - 1 \; \{1/2\} \; x - y^2 + 1$
end while

The martingale expression for $G = x$ *is* $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) = 0$*, i.e.* x *is a martingale. To apply the SM-Rule, we need to provide a* $p \in (0, 1]$ *and a* $d \in \mathbb{R}^+$ *such that eventually* x *decreases by at least* d *with a probability of at least* p*. To that end, we consider all branches* $B \in supp(\mathcal{U}^G_{\mathcal{L}})$*:* $x_i + y_i^2 + 2y_i, x_i + y_i^2 + 4y_i + 3$*,* $x_i + y_i^2 + 6y_i + 8, x_i - y_i^2 - 2y_i$*,* $x_i - y_i^2 - 4y_i - 3$ *and* $x_i - y_i^2 - 6y_i - 8$*. All branches occur with probability* $\frac{1}{6}$*.*

For branch $x_i - y_i^2 - 2y_i$*, we get that* x *changes by* $-y_i^2 - 2y_i$*. Moreover,* y_i *is eventually lower bounded (and upper bounded) by some function* $\alpha \cdot i$ *for some* $\alpha \in \mathbb{R}^+$*. This implies that eventually* $-y_i^2 - 2y_i \leq -\beta \cdot i^2$ *for some* $\beta \in \mathbb{R}^+$*. Hence, it holds that eventually* x *decreases by at least* 1 *with probability at least* $\frac{1}{6}$*. Therefore, the program is AST.*

We next present our method for computing asymptotic bounds over martingale expressions in Sections 6.5.1-6.5.2. Based on these asymptotic bounds, in Section 6.5.3 we introduce algorithmic approaches for our proof rules from Section 6.4, solving our aforementioned challenges (i)-(ii) in a fully automated manner (Section 6.5.4).

6.5.1 Prob-solvable Loops and Monomials

Algorithm 5 computes asymptotic bounds on monomials over program variables in a recursive manner. To ensure termination of Algorithm 5, it is important that there are no circular dependencies among monomials. By the definition of Prob-solvable loops, this indeed holds for program variables (monomials of order 1). Every Prob-solvable loop $\mathcal{L}$ comes with an ordering on its variables and every variable is restricted to only depend linearly on itself and polynomially on previous variables. Acyclic dependencies naturally extend from single variables to monomials.

Definition 52 (Monomial Ordering). *Let* $\mathcal{L}$ *be a Prob-solvable loop with variables* $x_{(1)}, ..., x_{(m)}$*. Let* $y_1 = \prod_{j=1}^m x_{(j)}^{p_j}$ *and* $y_2 = \prod_{j=1}^m x_{(j)}^{q_j}$*, where* $p_j, q_j \in \mathbb{N}$*, be two monomials over the program variables. The* order $\preceq$ on monomials *over the program variables of* $\mathcal{L}$ *is defined by* $y_1 \preceq y_2 \iff (p_m, ..., p_1) \leq_{lex} (q_m, ..., q_1)$*, where* $\leq_{lex}$ *is the lexicographic order on* $\mathbb{N}^m$*. The order* $\preceq$ *is total because* $\leq_{lex}$ *is total. With* $y_1 \prec y_2$ *we denote* $y_1 \preceq y_2 \land y_1 \neq y_2$*.*

Example 61 (Monomials). *Let* $\mathcal{L}$ *be a Prob-solvable loop with variables* $x_{(1)}, ..., x_{(m)}$*. The following statements hold for the monomial order* $\preceq$*:*

$1 \prec x_{(1)} \prec x_{(2)} \prec ... \prec x_{(m-1)} \prec x_{(m)}$*,* $x_{(1)}^k \prec x_{(2)}$ *for any* $k \in \mathbb{N}$

$x_{(1)}^2 \prec x_{(1)}^3$ *and* $x_{(3)}^4 x_{(2)}^{100} x_{(1)}^{99} \prec x_{(3)}^5 x_{(2)}^2 x_{(1)}^3$*.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

To prove acyclic dependencies for monomials we exploit the following fact.

Lemma 36. *Let y_1, y_2, z_1, z_2 be monomials. If $y_1 \preceq z_1$ and $y_2 \preceq z_2$ then $y_1 \cdot y_2 \preceq z_1 \cdot z_2$.*

By structural induction over monomials and Lemma 36, we establish:

Lemma 37 (Monomial Acyclic Dependency)**.** *Let x be a monomial over the program variables of $\mathcal{L}$. For every branch $B \in supp(\mathcal{U}^x_{\mathcal{L}})$ and monomial y in B, $y \preceq x$ holds.*

Proof. We use structural induction over monomials. The base case for which x is a single variable holds by the definition of $\mathcal{L}$ being a Prob-solvable loop. Let $x := s \cdot t$ where s and t are monomials over the variables of $\mathcal{L}$ and

- for every $B_s \in supp(\mathcal{U}^s_{\mathcal{L}})$ and every monomial u in B_s it holds that $u \preceq s$,
- for every $B_t \in supp(\mathcal{U}^t_{\mathcal{L}})$ and every monomial w in B_t it holds that $w \preceq t$,

Let $B \in supp(\mathcal{U}^x_{\mathcal{L}})$ be an arbitrary branch of x. By definition of $\mathcal{U}^x_{\mathcal{L}}$, we get $B = B_s \cdot B_t$, where B_s is a branch of s and B_t is a branch of t. Note that B_s and B_t are polynomials over program variables or equivalently linear combinations of monomials. Therefore, for every monomial y in B we have $y = u \cdot w$ where u is a monomial in B_s and w a monomial in B_t. By the induction hypothesis, $u \preceq s$ and $w \preceq t$. Using Lemma 36, we get $u \cdot w \preceq s \cdot t$ which means $y \preceq x$. □

Lemma 37 states that the value of a monomial x over the program variables of $\mathcal{L}$ only depends on the value of monomials y which precede x in the monomial ordering $\preceq$. This ensures the dependencies among monomials over the program variables of $\mathcal{L}$ to be acyclic.

6.5.2 Computing Asymptotic Bounds for Prob-solvable Loops

The structural result on monomial dependencies from Lemma 37 allows for recursive procedures over monomials. This is exploited in Algorithm 5 for computing asymptotic bounds for monomials. The standard Big-O notation does not differentiate between positive and negative functions, as it considers the absolute value of functions. We, however, need to differentiate between functions like 2^i and -2^i. Therefore, we introduce the notions of *Domination* and *Bounding Functions.*

Definition 53 (Domination)**.** *Let F be a finite set of functions from $\mathbb{N}$ to $\mathbb{R}$. A function $g : \mathbb{N} \to \mathbb{R}$ is* dominating *F if eventually $\alpha \cdot g(i) \geq f(i)$ for all $f \in F$ and some $\alpha \in \mathbb{R}^+$. A function $g : \mathbb{N} \to \mathbb{R}$ is* dominated by *F if all $f \in F$ dominate $\{g\}$.*

Intuitively, a function f dominates a function g if f eventually surpasses g modulo a positive constant factor. *Exponential polynomials* are sums of products of polynomials with exponential functions, i.e. $\sum_j p_j(x) \cdot c_j^x$, where $c_j \in \mathbb{R}^+_0$. All functions arising in

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithms 5–9 are exponential polynomials. For a finite set F of exponential polynomials, a function dominating F and a function dominated by F are easily computable with standard techniques, by analyzing the terms of the functions in the finite set F. With *dominating*(F) we denote an algorithm computing an exponential polynomial dominating F. With *dominated*(F) we denote an algorithm computing an exponential polynomial dominated by F. We assume the functions returned by the algorithms *dominating*(F) and *dominated*(F) to be monotone and either non-negative or non-positive.

Example 62 (Domination)**.** *The following statements are true:* 0 *dominates* $\{-i^3 + i^2 + 5\}$, i^2 *dominates* $\{2i^2\}$, $i^2 \cdot 2^i$ *dominates* $\{i^2 \cdot 2^i + i^9, i^5 + i^3, 2^{-i}\}$, i *is dominated by* $\{i^2 - 2i + 1, \frac{1}{2}i - 5\}$ *and* -2^i *is dominated by* $\{2^i - i^2, -10 \cdot 2^{-i}\}$.

Definition 54 (Bounding Function for $\mathcal{L}$)**.** *Let E be an arithmetic expression over the program variables of $\mathcal{L}$. Let $l, u : \mathbb{N} \to \mathbb{R}$ be monotone and non-negative or non-positive.*

1. *l is a* lower bounding function *for E if eventually $\mathbb{P}(\alpha \cdot l(i) \leq E_i \mid T^{\neg\mathcal{G}} > i) = 1$ for some $\alpha \in \mathbb{R}^+$.*
2. *u is an* upper bounding function *for E if eventually $\mathbb{P}(E_i \leq \alpha \cdot u(i) \mid T^{\neg\mathcal{G}} > i) = 1$ for some $\alpha \in \mathbb{R}^+$.*
3. *An* absolute bounding function *for E is an upper bounding function for $|E|$.*

A bounding function imposes a bound on an expression E over the program variables holding eventually, almost surely, and modulo a positive constant factor. Moreover, bounds on E only need to hold as long as the program has not yet terminated.

Given a Prob-solvable loop $\mathcal{L}$ and a monomial x over the program variables of $\mathcal{L}$, Algorithm 5 computes a lower and upper bounding function for x. Because every polynomial expression is a linear combination of monomials, the procedure can be used to compute lower and upper bounding functions for any polynomial expression over $\mathcal{L}$'s program variables by substituting every monomial with its lower or upper bounding function depending on the sign of the monomial's coefficient. Once a lower bounding function l and an upper bounding function u are computed, an absolute bounding function can be computed by *dominating*$(\{u, -l\})$.

In Algorithm 5, candidates for bounding functions are modeled using recurrence relations. Solutions $s(i)$ of these recurrences are closed-form candidates for bounding functions parameterized by loop iteration i. Algorithm 5 relies on the existence of closed-form solutions of recurrences. While closed-forms of general recurrences do not always exist, a property of *C-finite recurrences*, linear recurrences with constant coefficients, is that their closed-forms always exist and are computable [KP11]. In all occurring recurrences, we consider a monomial over program variables as a single function. Therefore, throughout this section, all recurrences arising from a Prob-solvable loop $\mathcal{L}$ in Algorithm 5 are C-finite or can be turned into C-finite recurrences. Moreover, closed-forms $s(i)$ of C-finite

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

recurrences are given by exponential polynomials. Therefore, for any solution $s(i)$ to a C-finite recurrence and any constant $r \in \mathbb{R}$, the following holds:

$$\exists \alpha, \beta \in \mathbb{R}^+, \exists i_0 \in \mathbb{N} : \forall i \geq i_0 : \alpha \cdot s(i) \leq s(i + r) \leq \beta \cdot s(i). \quad (6.2)$$

Intuitively, the property states that constant shifts do not change the asymptotic behavior of s. We use this property at various proof steps in this section. Moreover, we recall that limits of exponential polynomials are computable [Gru96].

For every monomial x, every branch $B \in \mathit{supp}(\mathcal{U}_{\mathcal{L}}^{x})$ is a polynomial over the program variables. Let $\mathit{Rec}(x) := \{\text{coefficient of } x \text{ in } B \mid B \in \mathit{supp}(\mathcal{U}_{\mathcal{L}}^{x})\}$ denote the set of coefficients of the monomial x in all branches of $\mathcal{L}$. Let $\mathit{Inhom}(x) := \{B - c \cdot x \mid B \in \mathit{supp}(\mathcal{U}_{\mathcal{L}}^{x}) \text{ and } c = \text{coefficient of } x \text{ in } B\}$ denote all the branches of the monomial x without x and its coefficient. The symbolic constants c_1 and c_2 in Algorithm 5 represent arbitrary initial values of the monomial x for which bounding functions are computed. The fact that they are symbolic ensures that all potential initial values are accounted for. c_1 represents positive initial values and $-c_2$ negative initial values. The symbolic constant d is used in the recurrences to account for the fact that the bounding functions only hold modulo a constant. Intuitively, if we use the bounding function in a recurrence we need to restore the lost constant. $\mathit{Sign}(x)$ is an over-approximation of the sign of the monomial x, i.e., if $\exists i : \mathbb{P}(x_i > 0) > 0$, then $+ \in \mathit{Sign}(x)$ and if $\exists i : \mathbb{P}(x_i < 0) > 0$, then $- \in \mathit{Sign}(x)$.

Algorithm 5 Computing bounding functions for monomials

Input: A Prob-solvable loop $\mathcal{L}$ and a monomial x over $\mathcal{L}$'s variables
Output: Lower and upper bounding functions $l(i)$, $u(i)$ for x

1: $\mathit{inhomBoundsUpper} \leftarrow \{\text{upper bounding function of } P \mid P \in \mathit{Inhom}(x)\}$ (recursion)
2: $\mathit{inhomBoundsLower} \leftarrow \{\text{lower bounding function of } P \mid P \in \mathit{Inhom}(x)\}$ (recursion)
3: $U(i) \leftarrow \mathit{dominating}(\mathit{inhomBoundsUpper})$
4: $L(i) \leftarrow \mathit{dominated}(\mathit{inhomBoundsLower})$
5: $\mathit{maxRec} \leftarrow \max \mathit{Rec}(x)$
6: $\mathit{minRec} \leftarrow \min \mathit{Rec}(x)$
7: $I \leftarrow \emptyset$
8: **if** $+ \in \mathit{Sign}(x)$ **then**
9: $\quad I \leftarrow I \cup \{c_1\}$
10: **end if**
11: **if** $- \in \mathit{Sign}(x)$ **then**
12: $\quad I \leftarrow I \cup \{-c_2\}$
13: **end if**
14: $\mathit{uCand} \leftarrow$ closed-forms of $\{y_{i+1} = r \cdot y_i + d \cdot U(i) \mid r \in \{\mathit{minRec}, \mathit{maxRec}\}, y_0 \in I\}$
15: $\mathit{lCand} \leftarrow$ closed-forms of $\{y_{i+1} = r \cdot y_i + d \cdot L(i) \mid r \in \{\mathit{minRec}, \mathit{maxRec}\}, y_0 \in I\}$
16: $u(i) \leftarrow \mathit{dominating}(\mathit{uCand})$
17: $l(i) \leftarrow \mathit{dominated}(\mathit{lCand})$
18: **return** $l(i), u(i)$

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Lemma 37, the computability of closed-forms of C-finite recurrences and the fact that within a Prob-solvable loop only finitely many monomials can occur, implies the termination of Algorithm 5. Its correctness is stated in the next theorem.

Theorem 38 (Correctness of Algorithm 5)**.** *The functions $l(i), u(i)$ returned by Algorithm 5 on input $\mathcal{L}$ and x are a lower- and an upper bounding function for x, respectively.*

Proof. Intuitively, it has to be shown that regardless of the paths through the loop body taken by any program run, the value of x is always eventually upper bounded by some function in *uCand* and eventually lower bounded by some function in *lCand* (almost surely and modulo positive constant factors). We show that x is always eventually upper bounded by some function in *uCand*. The proof for the lower bounding function is analogous.

Let $\vartheta \in \Sigma$ be a *possible* program run, i.e. $\mathbb{P}(Cyl(\pi)) > 0$ for all finite prefixes π of ϑ. Then, for every $i \in \mathbb{N}$, if $T^{\neg\mathcal{G}}(\vartheta) > i$, the following holds:

$$x_{i+1}(\vartheta) = a_{(1)} \cdot x_i(\vartheta) + P_{(1)i}(\vartheta) \quad \text{or} \quad x_{i+1}(\vartheta) = a_{(2)} \cdot x_i(\vartheta) + P_{(2)i}(\vartheta)$$
$$\text{or ... or} \quad x_{i+1}(\vartheta) = a_{(k)} \cdot x_i(\vartheta) + P_{(k)i}(\vartheta),$$

where $a_{(j)} \in Rec(x)$ and $P_{(j)} \in Inhom(x)$ are polynomials over program variables. Let $u_1(i), ..., u_k(i)$ be upper bounding functions of $P_{(1)}, ..., P_{(k)}$, which are computed recursively at line 14. Moreover, let $U(i) := dominating(\{u_1(i), ..., u_k(i)\})$, $minRec = \min Rec(x)$ and $maxRec = \max Rec(x)$. Let $l_0 \in \mathbb{N}$ be the smallest number such that for all $j \in \{1, ..., k\}$ and $i \geq l_0$:

$$\mathbb{P}(P_{(j)i} \leq \alpha_j \cdot u_j(i) \mid T^{\neg\mathcal{G}} > i) = 1 \text{ for some } \alpha_j \in \mathbb{R}^+, \text{ and} \tag{6.3}$$
$$u_j(i) \leq \beta \cdot U(i) \text{ for some } \beta \in \mathbb{R}^+ \tag{6.4}$$

Thus, all inequalities from the bounding functions u_j and the dominating function U hold from l_0 onward. Because U is a dominating function, it is by definition either non-negative or non-positive. Assume $U(i)$ to be non-negative, the case for which $U(i)$ is non-positive is symmetric. Using the facts (6.3) and (6.4), we establish: For the constant $\gamma := \beta \cdot \max_{j=1..k} \alpha_j$, it holds that $\mathbb{P}(P_{(j)i} \leq \gamma \cdot U(i) \mid T^{\neg\mathcal{G}} > i) = 1$ for all $j \in \{1, ..., k\}$ and all $i \geq l_0$. Let l_1 be the smallest number such that $l_1 \geq l_0$ and $U(i + l_0) \leq \delta \cdot U(i)$ for all $i \geq l_1$ and some $\delta \in \mathbb{R}^+$.

Case 1, x_i is almost surely negative for all $i \geq l_1$. Consider the recurrence relation $y_0 = m$, $y_{i+1} = minRec \cdot y_i + \eta \cdot U(i)$, where $\eta := \max(\gamma, \delta)$ and m is the maximum value of $x_{l_1}(\vartheta)$ among all possible program runs ϑ. Note that m exists because there are only finitely many values $x_{l_1}(\vartheta)$ for possible program runs ϑ. Moreover, m is negative by our case assumption. By induction, we get $\mathbb{P}(x_i \leq y_{i-l_1} \mid T^{\neg\mathcal{G}} > i) = 1$ for all $i \geq l_1$. Therefore, for a closed-form solution $s(i)$ of the recurrence relation y_i, we get $\mathbb{P}(x_i \leq s(i - l_1) \mid T^{\neg\mathcal{G}} > i) = 1$ for all $i \geq l_1$. We emphasize that s exists and can

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

effectively be computed because y_i is C-finite. Moreover, $s(i - l_1) \leq \theta \cdot s(i)$ for all $i \geq l_2$ for some $l_2 \geq l_1$ and some $\theta \in \mathbb{R}^+$. Therefore, s satisfies the bound condition of an upper bounding function. Also, s is present in *uCand* by choosing the symbolic constants c_2 and d to represent $-m$ and η respectively. The function $u(i) := \mathit{dominating}(\mathit{uCand})$, at line 16, is dominating *uCand* (hence also s), is monotone and either non-positive or non-negative. Therefore, $u(i)$ is an upper bounding function for x.

Case 2, x_i is not almost surely negative for all $i \geq l_1$. Thus, there is a possible program run ϑ' such that $x_i(\vartheta') \geq 0$ for some $i \geq l_1$. Let $l_2 \geq l_1$ be the smallest number such that $x_{l_2}(\hat{\vartheta}) \geq 0$ for some possible program run $\hat{\vartheta}$. This number certainly exists, as $x_i(\vartheta')$ is non-negative for some $i \geq l_1$. Consider the recurrence relation $y_0 = m$, $y_{i+1} = \mathit{maxRec} \cdot y_i + \eta \cdot U(i)$, where $\eta := \max(\gamma, \delta)$ and m is the maximum value of $x_{l_2}(\vartheta)$ among all possible program runs ϑ. Note that m exists because there are only finitely many values $x_{l_2}(\vartheta)$ for possible program runs ϑ. Moreover, m is non-negative because $m \geq x_{l_2}(\hat{\vartheta}) \geq 0$. By induction, we get $\mathbb{P}(x_i \leq y_{i-l_2} \mid T^{\neg\mathcal{G}} > i) = 1$ for all $i \geq l_2$. Therefore, for a solution $s(i)$ of the recurrence relation y_i, we get $\mathbb{P}(x_i \leq s(i - l_2) \mid T^{\neg\mathcal{G}} > i) = 1$ for all $i \geq l_2$. As above, s exists and can effectively be computed because y_i is C-finite. Moreover, $s(i - l_2) \leq \theta \cdot s(i)$ for all $i \geq l_3$ for some $l_3 \geq l_2$ and some $\theta \in \mathbb{R}^+$. Therefore, s satisfies the bound condition of an upper bounding function Also, s is present in *uCand* by choosing the symbolic constants c_1 and d to represent m and η respectively. The function $u(i) := \mathit{dominating}(\mathit{uCand})$, at line 16, is dominating *uCand* (hence also s), is monotone and either non-positive or non-negative. Therefore, $u(i)$ is an upper bounding function for x. □

Example 63 (Bounding functions)**.** *We illustrate Algorithm 5 by computing bounding functions for x and the Prob-solvable loop from Example 58: We have $\mathit{Rec}(x) := \{2, \frac{1}{2}\}$ and $\mathit{Inhom}(x) = \{y^2, 0\}$. Computing bounding functions recursively for $P \in \mathit{Inhom}(x) = \{y^2, 0\}$ is simple, as we can give exact bounds leading to $\mathit{inhomBoundsUpper} = \{i^2, 0\}$ and $\mathit{inhomBoundsLower} = \{i^2, 0\}$. Consequently, we get $U(i) = i^2$, $L(i) = 0$, $\mathit{maxRec} = 2$ and $\mathit{minRec} = \frac{1}{2}$. With a rudimentary static analysis of the loop, we determine the (exact) over-approximation $\mathit{Sign}(x) := \{+\}$ by observing that $x_0 > 0$ and all $P \in \mathit{Inhom}(x)$ are strictly positive. Therefore, uCand is the set of closed-form solutions of the recurrences $y_0 := c_1$, $y_{i+1} := 2y_i + d \cdot i^2$ and $y_0 := c_1$, $y_{i+1} := \frac{1}{2}y_i + d \cdot i^2$. Similarly, lCand is the set of closed-form solutions of the recurrences $y_0 := c_1$, $y_{i+1} := 2y_i$ and $y_0 := c_1$, $y_{i+1} := \frac{1}{2}y_i$. Using any algorithm for computing closed-forms of C-finite recurrences, we obtain $\mathit{uCand} = \{c_1 2^i - di^2 - 2di + 3d2^i - 3d,\ c_1 2^{-i} + 2di^2 - 8di - 12d2^{-i} + 12d\}$ and $\mathit{lCand} = \{c_1 2^i,\ c_1 2^{-i}\}$. This leads to the upper bounding function $u(i) = 2^i$ and the lower bounding function $l(i) = 2^{-i}$. The bounding functions $l(i)$ and $u(i)$ can be used to compute bounding functions for expressions containing x linearly by replacing x by $l(i)$ or $u(i)$ depending on the sign of the coefficient of x. For instance, eventually and almost surely the following inequality holds: $-\frac{x_i}{4} - \frac{i^2}{2} - i - \frac{1}{2} \leq -\frac{1}{4} \cdot \alpha \cdot 2^{-i} - \frac{i^2}{2} - i - \frac{1}{2}$ for some $\alpha \in \mathbb{R}^+$. The inequality results from replacing x_i by $l(i)$. Therefore, eventually and almost surely $-\frac{x_i}{4} - \frac{i^2}{2} - i - \frac{1}{2} \leq -\beta \cdot i^2$ for some $\beta \in \mathbb{R}^+$. Thus, $-i^2$ is an upper*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

bounding function for the expression $-\frac{x_i}{4} - \frac{i^2}{2} - i - \frac{1}{2}$.

Remark 10. *Algorithm 5 describes a general procedure computing bounding functions for special sequences. Figuratively, that is for sequences s such that $s_{i+1} = f(s_i, i)$ but in every step the function f is chosen non-deterministically among a fixed set of special functions (corresponding to branches in our case). We reserve the investigation of applications of bounding functions for such sequences beyond the probabilistic setting for future work.*

6.5.3 Algorithms for Termination Analysis of Prob-solvable Loops

Using Algorithm 5 to compute bounding functions for polynomial expressions over program variables at hand, we are now able to formalize our algorithmic approaches automating the termination analysis of Prob-solvable loops using the proof rules from Section 6.4. Given a Prob-solvable loop $\mathcal{L}$ and a polynomial expression E over $\mathcal{L}$'s variables, we denote with $lbf(E)$, $ubf(E)$ and $abf(E)$ functions computing a lower, upper and absolute bounding function for E respectively. Our algorithmic approach for proving PAST using the RSM-Rule is given in Algorithm 6.

Algorithm 6 Ranking-Supermartingale-Rule for proving PAST

Input: Prob-solvable loop $\mathcal{L}$
Output: If *true* then $\mathcal{L}$ with G satisfies the RSM-Rule; hence $\mathcal{L}$ is PAST

1: $E \leftarrow \mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$
2: $u(i) \leftarrow ubf(E)$
3: $limit \leftarrow \lim_{i\to\infty} u(i)$
4: **return** $limit < 0$

Example 64 (Algorithm 6)**.** *Let us illustrate Algorithm 6 with the Prob-solvable loop from Examples 58 and 63. Applying Algorithm 6 on $\mathcal{L}$ leads to $E = -\frac{x_i}{4} - \frac{i^2}{2} - i - \frac{1}{2}$. We obtain the upper bounding function $u(i) := -i^2$ for E. Because $\lim_{i\to\infty} u(i) < 0$, Algorithm 6 returns true. This is valid because $u(i)$ having a negative limit witnesses that E is eventually bounded by a negative constant and therefore is eventually an RSM.*

We recall that all functions arising from $\mathcal{L}$ are exponential polynomials (see Section 6.5.2) and that limits of exponential polynomials are computable [Gru96]. Therefore, the termination of Algorithm 6 is guaranteed and its correctness is stated next.

Theorem 39 (Correctness of Algorithm 6)**.** *If Algorithm 6 returns* true *on input $\mathcal{L}$, then $\mathcal{L}$ with $G_{\mathcal{L}}$ satisfies the RSM-Rule.*

Proof. When returning *true* at line 4 we have $\mathbb{P}(E_i \leq \alpha \cdot u(i) \mid T^{\neg\mathcal{G}} > i) = 1$ for all $i \geq i_0$ and some $i_0 \in \mathbb{N}$, $\alpha \in \mathbb{R}^+$. Moreover, $u(i) < -\epsilon$ for all $i \geq i_1$ for some $i_1 \in \mathbb{N}$, by the definition of lim. From this follows that $\forall i \geq \max(i_0, i_1)$ almost surely $\mathcal{G}_i \implies \mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) \leq -\alpha \cdot \epsilon$, which means G is eventually an RSM. □

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, Your knowledge hub, WIEN

Our approach proving AST using the SM-Rule is captured with Algorithm 7.

Algorithm 7 Supermartingale-Rule for proving AST

Input: Prob-solvable loop $\mathcal{L}$

Output: If *true*, $\mathcal{L}$ with G satisfies the SM-Rule with constant d and p; hence $\mathcal{L}$ is AST

1: $E \leftarrow \mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i)$
2: $u(i) \leftarrow ubf(E)$
3: **if** not eventually $u(i) \leq 0$ **then**
4:     **return** false
5: **end if**
6: **for** $B \in supp(\mathcal{U}_{\mathcal{L}}^{G})$ **do**
7:     $d(i) \leftarrow ubf(B - G)$
8:     $limit \leftarrow \lim_{i\to\infty} d(i)$
9:     **if** $limit < 0$ **then**
10:         **return** true
11:     **end if**
12: **end for**
13: **return** false

Example 65 (Algorithm 7)**.** *Let us illustrate Algorithm 7 for the Prob-solvable loop $\mathcal{L}$ from Figure 6.2a: Applying Algorithm 7 on $\mathcal{L}$ yields $E \equiv 0$ and $u(i) = 0$. The expression G (= x) has two branches. One of them is $x_i - y_i + 4$, which occurs with probability $1/2$. When the for-loop of Algorithm 7 reaches this branch $B = x_i - y_i + 4$ on line 6, it computes the difference $B - G = -y_i + 4$. An upper bounding function for $B - G$ is given by $d(i) = -i$. Because $\lim_{i\to\infty} d(i) < 0$, Algorithm 7 returns true. This is valid because of the branch B witnessing that G eventually decreases by at least a constant with probability $1/2$. Therefore, all conditions of the SM-Rule are satisfied and $\mathcal{L}$ is AST.*

Theorem 40 (Correctness of Algorithm 7)**.** *If Algorithm 7 returns* true *on input $\mathcal{L}$, then $\mathcal{L}$ with $G_{\mathcal{L}}$ satisfies the SM-Rule with constant d and p.*

Proof. Similarly as for the correctness of Algorithm 6, G is a supermartingale if Algorithm 7 returns *true*. Moreover, there is a branch $B \in supp(\mathcal{U}_{\mathcal{L}}^{G})$ such that G changes eventually and almost surely by at most $\alpha \cdot d(i)$, for some $\alpha \in \mathbb{R}^+$. In addition, because $\lim_{i\to\infty} d(i) < 0$, it follows that $d(i) \leq -\epsilon$ for all $i \geq i_0$ for some $i_0 \in \mathbb{N}$, $\epsilon \in \mathbb{R}^+$. Therefore, eventually G decreases by at least $\alpha \cdot \epsilon$ with probability at least $\mathcal{U}_{\mathcal{L}}^{G}(B) > 0$. Hence, all conditions of the SM-Rule are satisfied. □

As established in Section 6.4, the relaxation of the R-AST-Rule requires that there is a positive probability of reaching the iteration i_0 after which the conditions of the proof rule hold. Regarding automation, we strengthen this condition by ensuring that there is a positive probability of reaching any iteration, i.e. $\forall i \in \mathbb{N} : \mathbb{P}(\mathcal{G}_i) > 0$. Obviously, this implies $\mathbb{P}(\mathcal{G}_{i_0}) > 0$. Furthermore, with *CanReachAnyIteration*($\mathcal{L}$) we denote a computable

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

under-approximation of $\forall i \in \mathbb{N} : \mathbb{P}(\mathcal{G}_i) > 0$. That means, *CanReachAnyIteration*($\mathcal{L}$) implies $\forall i \in \mathbb{N} : \mathbb{P}(\mathcal{G}_i) > 0$. Our approach proving non-AST is summarized in Algorithm 8.

Algorithm 8 Repulsing-AST-Rule for proving non-AST

Input: Prob-solvable loop $\mathcal{L}$

Output: if *true*, $\mathcal{L}$ with $-G$ satisfies the R-AST-Rule; hence $\mathcal{L}$ is not AST

1: $E \leftarrow \mathbb{E}(-G_{i+1} + G_i \mid \mathcal{F}_i)$
2: $u(i) \leftarrow ubf(E)$
3: **if** not eventually $u(i) \leq 0$ **then**
4:     **return** false
5: **end if**
6: **if** $\neg$*CanReachAnyIteration*($\mathcal{L}$) **then**
7:     **return** false
8: **end if**
9: $\epsilon(i) \leftarrow -u(i)$
10: **if** $\epsilon(i) \notin \Omega(1)$ **then**
11:     **return** false
12: **end if**
13: $differences \leftarrow \{B + G \mid B \in supp(\mathcal{U}_{\mathcal{L}}^{-G})\}$
14: $diffBounds \leftarrow \{abf(d) \mid d \in differences\}$
15: $c(i) \leftarrow dominating(diffBounds)$
16: **return** $c(i) \in O(1)$

Example 66 (Algorithm 8). *Let us illustrate Algorithm 8 for the Prob-solvable loop $\mathcal{L}$ from Figure 6.2a: Applying Algorithm 8 on $\mathcal{L}$ leads to $E = \frac{y_i}{6} - \frac{1}{3} = \frac{2^{-i}}{3} - \frac{1}{3}$ and to the upper bounding function $u(i) = -1$ for E on line 2. Therefore, the if-statement on line 3 is not executed, which means $-G$ is eventually a ϵ-repulsing supermartingale. Moreover, with a simple static analysis of the loop, we establish CanReachAnyIteration($\mathcal{L}$) to be true, as there is a positive probability that the loop guard does not decrease. Thus, the if-statement on line 6 is not executed. Also, the if-statement on line 10 is not executed, because $\epsilon(i) = -u(i) = 1$ is constant and therefore in $\Omega(1)$. E eventually decreases by $\epsilon = 1$ (modulo a positive constant factor), because $u(i) = -1$ is an upper bounding function for E. We have differences $= \{1 - \frac{y_i}{2}, 1 + \frac{y_i}{2}\}$. Both expressions in differences have an absolute bounding function of 1. Therefore, diffBounds $= \{1\}$. As a result on line 15 we have $c(i) = 1$, which eventually and almost surely is an upper bound on $|-G_{i+1} + G_i|$ (modulo a positive constant factor). Therefore, the algorithm returns true. This is correct, as all the preconditions of the R-AST-Rule are satisfied (and therefore $\mathcal{L}$ is not AST).*

Theorem 41 (Correctness of Algorithm 8). *If Algorithm 8 returns* true *on input $\mathcal{L}$, then $\mathcal{L}$ with $-G_{\mathcal{L}}$ satisfies the R-AST-Rule.*

Proof. With the same reasoning as for the correctness of Algorithm 7, $-G$ is a supermartingale if Algorithm 8 returns *true*. Moreover, the condition $\mathbb{P}(-G_{i_0} < 0) > 0$ of

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the R-AST-Rule is satisfied, due to the under-approximation $\mathit{CanReachAnyIteration}(\mathcal{L})$ and the if-statement on line 6. The function $u(i)$ is an upper bounding function for $\mathbb{E}(-G_{i+1} + G_i \mid \mathcal{F}_i)$. Hence, eventually and almost surely $\mathbb{E}(-G_{i+1} + G_i \mid \mathcal{F}_i) \leq -\alpha \cdot \epsilon(i)$ for $\epsilon(i) := -u(i)$ and some $\alpha \in \mathbb{R}^+$. The if-statement at line 10 ensures that $\epsilon(i)$ is lower bounded by a constant. Therefore, $-G$ eventually is an $(\alpha \cdot \epsilon)$-repulsing supermartingale. The function $c(i)$, assigned to $\mathit{dominating}(\mathit{diffBounds})$, is a function dominating absolute bounding functions of all branches of $-G_{i+1} + G_i$. Consequently, $c(i)$ is a bound on the differences of G, i.e. eventually and almost surely $|-G_{i+1} + G_i| \leq \beta \cdot c(i)$ for some $\beta \in \mathbb{R}^+$. Algorithm 8 returns true only if $c(i)$ can be bounded by a constant which in turn means G has $(\beta \cdot c)$-bounded differences. Thus, if Algorithm 8 returns *true*, all preconditions of the R-AST-Rule are satisfied. □

We finally provide Algorithm 9 for the R-PAST-Rule. The algorithm is a variation of Algorithm 8 (for the R-AST-Rule). The if-statement on line 2 forces $-G$ to be a martingale. Therefore, after the if-statement $-G$ is an ϵ-repulsing supermartingale with $\epsilon = 0$.

Algorithm 9 Repulsing-PAST-Rule for proving non-PAST

Input: Prob-solvable loop $\mathcal{L}$
Output: If *true*, $\mathcal{L}$ with $-G$ satisfies the R-PAST-Rule; hence $\mathcal{L}$ is not PAST

1: $E \leftarrow \mathbb{E}(-G_{i+1} + G_i \mid \mathcal{F}_i)$
2: **if** $E \not\equiv 0$ **then**
3:     **return** false
4: **end if**
5: **if** $\neg\mathit{CanReachAnyIteration}(\mathcal{L})$ **then**
6:     **return** false
7: **end if**
8: $\mathit{differences} \leftarrow \{B + G \mid B \in \mathit{supp}(\mathcal{U}_{\mathcal{L}}^{-G})\}$
9: $\mathit{diffBounds} \leftarrow \{\mathit{abf}(d) \mid d \in \mathit{differences}\}$
10: $c(i) \leftarrow \mathit{dominating}(\mathit{diffBounds})$
11: **return** $c(i) \in O(1)$

6.5.4 Ruling out Proof Rules for Prob-Solvable Loops

A question arising when combining our algorithmic approaches from Section 6.5.3 into a unifying framework is that, given a Prob-solvable loop $\mathcal{L}$, what algorithm to apply first for determining $\mathcal{L}$'s termination behavior? In [BKS19] the authors provide an algorithm for computing an algebraically closed-form of $\mathbb{E}(M_i)$, where M is a polynomial over $\mathcal{L}$'s variables. The following lemma explains how the expression $\mathbb{E}(M_{i+1} - M_i)$ relates to the expression $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i)$.

Lemma 42 (Rule out Rules for $\mathcal{L}$)**.** *Let $(M_i)_{i\in\mathbb{N}}$ be a stochastic process. If $\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq -\epsilon$ then $\mathbb{E}(M_{i+1} - M_i) \leq -\epsilon$, for any $\epsilon \in \mathbb{R}^+$.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Proof.

$$
\begin{aligned}
&\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i) \leq -\epsilon && \Longrightarrow && \text{(Monotonicity of } \mathbb{E}\text{)} \\
&\mathbb{E}(\mathbb{E}(M_{i+1} - M_i \mid \mathcal{F}_i)) \leq \mathbb{E}(-\epsilon) && \Longleftrightarrow && \text{(Property of } \mathbb{E}(\cdot \mid \mathcal{F}_i)\text{)} \\
&\mathbb{E}(M_{i+1} - M_i) \leq \mathbb{E}(-\epsilon) && \Longleftrightarrow && (-\epsilon \text{ is constant}) \\
&\mathbb{E}(M_{i+1} - M_i) \leq -\epsilon && && \square
\end{aligned}
$$

The contrapositive of Lemma 42 provides a criterion to rule out the viability of a given proof rule. For a Prob-solvable loop $\mathcal{L}$, if $\mathbb{E}(G_{i+1} - G_i) \not\leq 0$ then $\mathbb{E}(G_{i+1} - G_i \mid \mathcal{F}_i) \not\leq 0$, meaning G is not a supermartingale. The expression $\mathbb{E}(G_{i+1} - G_i)$ depends only on i and can be computed by $\mathbb{E}(G_{i+1} - G_i) = \mathbb{E}(G_{i+1}) - \mathbb{E}(G_i)$, where the expected value $\mathbb{E}(G_i)$ is computed as in [BKS19]. Therefore, in some cases, proof rules can automatically be deemed nonviable, without the need to compute bounding functions.

6.6 Implementation and Evaluation

6.6.1 Implementation

We implemented and combined our algorithmic approaches from Section 6.5 in the new software tool Amber to stand for *Asymptotic Martingale Bounds*. Amber and all benchmarks are available at `https://github.com/probing-lab/amber`. Amber uses Mora [BKS19][BKS20c] for computing the first-order moments of program variables and the diofant package[3] as its computer algebra system.

Computing *dominating* and *dominated*. The *dominating* and *dominated* procedures used in Algorithms 5 and 8 are implemented by combining standard algorithms for Big-O analysis and bookkeeping of the asymptotic polarity of the input functions. Let us illustrate this. Consider the following two input-output-pairs which our implementation would produce: (a) $\mathit{dominating}(\{i^2 + 10, 10 \cdot i^5 - i^3\}) = i^5$ and (b) $\mathit{dominating}(\{-i + 50, -i^8 + i^2 - 3 \cdot i^3\}) = -i$. For (a) i^5 is eventually greater than all functions in the input set modulo a constant factor because all functions in the input set are $O(i^5)$. Therefore, i^5 dominates the input set. For (b), the first function is $O(i)$ and the second is $O(i^8)$. In this case, however, both functions are eventually negative. Therefore, $-i$ is a function dominating the input set. Important is the fact that an exponential polynomial $\sum_j p_j(i) \cdot c_j^i$, where $c_j \in \mathbb{R}_0^+$ will always be eventually either only positive or only negative (or 0 if identical to 0).

Sign Over-Approximation. The over-approximation $\mathit{Sign}(x)$ of the signs of a monomial x used in Algorithm 5 is implemented by a simple static analysis: For a monomial x consisting solely of even powers, $\mathit{Sign}(x) = \{+\}$. For a general monomial x, if $x_0 \geq 0$ and all monomials on which x depends, together with their associated coefficients are always

[3] `https://github.com/diofant/diofant`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

positive, then $- \notin Sign(x)$. For example, if $supp(\mathcal{U}_{\mathcal{L}}^{x}) = \{x_i + 2y_i - 3z_i, x_i + u_i\}$, then $- \notin Sign(x)$ if $x_0 \geq 0$ as well as $- \notin Sign(y)$, $+ \notin Sign(z)$ and $- \notin Sign(u)$. Otherwise, $- \in Sign(x)$. The over-approximation for $+ \notin Sign(x)$ is analogous.

Reachability Under-Approximation. The procedure $CanReachAnyIteration(\mathcal{L})$, used in Algorithm 8, needs to satisfy the property that if it returns true, then loop $\mathcal{L}$ reaches any iteration with positive probability. In Amber, we implement this under-approximation as follows: $CanReachAnyIteration(\mathcal{L})$ is true if there is a branch B of the loop guard polynomial $G_{\mathcal{L}}$ such that $B - G_{\mathcal{L}i}$ is non-negative for all $i \in \mathbb{N}$. Otherwise, $CanReachAnyIteration(\mathcal{L})$ is false. In other words, if $CanReachAnyIteration(\mathcal{L})$ is true, then in any iteration there is a positive probability of $G_{\mathcal{L}}$ not decreasing.

Bound Computation Improvements. In addition to Algorithm 5 computing bounding functions for monomials of program variables, Amber implements the following refinements:

1. A monomial x is deterministic, which means it is independent of probabilistic choices, if x has a single branch and only depends on monomials having single branches. In this case, the exact value of x in any iteration is given by its first-order moments and bounding functions can be obtained by using these exact representations.

2. Bounding functions for an odd power p of a monomial x can be computed by $u(i)^p$ and $l(i)^p$, where $u(i)$ is an upper- and $l(i)$ a lower bounding function for x.

Whenever the above enhancements are applicable, Amber prefers them over Algorithm 5.

6.6.2 Experimental Setting and Results

Experimental Setting and Comparisons. Regarding programs which are PAST, we compare Amber against the tool Absynth [NCH18] and the tool in [CS13] which we refer to as Mgen. Absynth uses a system of inference rules over the syntax of probabilistic programs to derive bounds on the expected resource consumption of a program and can, therefore, be used to certify PAST. In comparison to Amber, Absynth requires the degree of the bound to be provided upfront. Moreover, Absynth cannot refute the existence of a bound and therefore cannot handle programs that are not PAST. Mgen uses linear programming to synthesize linear martingales and supermartingales for probabilistic transition systems with linear variable updates. To certify PAST, we extended Mgen [CS13] with the SMT solver Z3 [dMB08] in order to find or refute the existence of conical combinations of the (super)martingales derived by Mgen which yield RSMs.

With Amber-Light we refer to a variant of Amber without the relaxations of the proof rules introduced in Section 6.4. That is, with Amber-Light the conditions of the proof rules need to hold for all $i \in \mathbb{N}$, whereas with Amber the conditions are allowed to only

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

hold eventually. For all benchmarks, we compare AMBER against AMBER-LIGHT to show the effectiveness of the respective relaxations. For each experimental table (Tables 6.1-6.3), ✓ symbolizes that the respective tool successfully certified PAST/AST/non-AST for the given program; ✗ means it failed to certify PAST/AST/non-AST. Further, **NA** indicates the respective tool failed to certify PAST/AST/non-AST because the given program is out-of-scope of the tool's capabilities. Every benchmark has been run on a machine with a 2.2 GHz Intel i7 (Gen 6) processor and 16 GB of RAM and finished within a timeout of 50 seconds, where most benchmarks terminated within a few seconds.

Benchmarks. We evaluated AMBER against 38 probabilistic programs. We present our experimental results by separating our benchmarks within three categories: (i) 21 programs which are PAST (Table 6.1), (ii) 11 programs which are AST (Table 6.2) but not necessarily PAST, and (iii) 6 programs which are not AST (Table 6.3). The benchmarks have either been introduced in the literature on probabilistic programming [NCH18, CS13, BKS19, GGH19, MMKK18], are adaptations of well-known stochastic processes or have been designed specifically to test unique features of AMBER, like the ability to handle polynomial real arithmetic.

The 21 PAST benchmarks consist of 10 programs representing the original benchmarks of MGEN [CS13] and ABSYNTH [NCH18] augmented with 11 additional probabilistic programs. Not all benchmarks of MGEN and ABSYNTH could be used for our comparison as MGEN and ABSYNTH target related but different computation tasks than certifying PAST. Namely, MGEN aims to synthesize (super)martingales, but not ranking ones, whereas ABSYNTH focuses on computing bounds on the expected runtime. Therefore, we adopted *all* (50) benchmarks from [CS13] (11) and [NCH18] (39) for which the termination behavior is non-trivial. A benchmark is trivial regarding PAST if either (i) there is no loop, (ii) the loop is bounded by a constant, or (iii) the program is meant to run forever. Moreover, we cleansed the benchmarks of programs for which the witness for PAST is just a trivial combination of witnesses for already included programs. For instance, the benchmarks of [NCH18] contain multiple programs that are concatenated constant biased-random-walks. These are relevant benchmarks when evaluating ABSYNTH for discovering bounds, but would blur the picture when comparing against AMBER for PAST certification. With these criteria, 10 out of the 50 original benchmarks of [CS13] and [NCH18] remain. We add 11 additional benchmarks which have either been introduced in the literature on probabilistic programming [BKS19, GGH19, MMKK18], are adaptations of well-known stochastic processes or have been designed specifically to test unique features of AMBER. Notably, out of the 50 original benchmarks from [NCH18] and [CS13], only 2 remain which are included in our benchmarks and which AMBER cannot prove PAST (because they are not Prob-solvable). All our benchmarks are available at `https://github.com/probing-lab/amber`.

Experiments with PAST – Table 6.1. Out of the 21 PAST benchmarks, AMBER certifies 18 programs. AMBER cannot handle the benchmarks *nested_loops* and *sequential_loops*, as these examples use nested or sequential loops and thus are not expressible

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.1: 21 programs which are PAST.

Program	Amber	Amber-Light	Absynth	MgenZ3
2d_bounded_random_walk	✓	✓	✗	NA
biased_random_walk_constant	✓	✓	✓	✓
biased_random_walk_exp	✓	✓	✗	✓
biased_random_walk_poly	✓	✗	✗	✗
binomial_past	✓	✓	✓	✓
complex_past	✓	✗	✗	NA
consecutive_bernoulli_trails	✓	✓	✓	✓
coupon_collector_4	✓	✗	✗	✓
coupon_collector_5	✓	✗	✗	✓
dueling_cowboys	✓	✓	✓	✓
exponential_past_1	✓	✓	NA	NA
exponential_past_2	✓	✓	NA	NA
geometric	✓	✓	✓	✓
geometric_exponential	✗	✗	✗	✗
linear_past_1	✓	✓	✗	✗
linear_past_2	✓	✓	✗	NA
nested_loops	NA	NA	✓	✗
polynomial_past_1	✓	✗	✗	NA
polynomial_past_2	✓	✗	✗	NA
sequential_loops	NA	NA	✓	✗
tortoise_hare_race	✓	✓	✓	✓
Total ✓	18	12	8	9

as Prob-solvable loops. The benchmarks *exponential_past_1* and *exponential_past_2* are out of scope of Absynth because they require real numbers, while Absynth can only handle integers. MgenZ3 cannot handle benchmarks containing non-linear variable updates or non-linear guards. Table 6.1 shows that Amber outperforms both Absynth and MgenZ3 for Prob-solvable loops, even when our relaxed proof rules from Section 6.4 are not used. Yet, our experiments show that our relaxed proof rules enable Amber to certify 6 examples to be PAST, which could not be proved without these relaxations by Amber-Light.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Table 6.2: 11 programs which are AST and not necessarily PAST.

Program	Amber	Amber-Light
fair_in_limit_random_walk	NA	NA
gambling	✓	✓
symmetric_2d_random_walk	✗	✗
symmetric_random_walk_constant_1	✓	✓
symmetric_random_walk_constant_2	✓	✓
symmetric_random_walk_exp_1	✓	✗
symmetric_random_walk_exp_2	✓	✗
symmetric_random_walk_linear_1	✓	✗
symmetric_random_walk_linear_2	✓	✓
symmetric_random_walk_poly_1	✓	✗
symmetric_random_walk_poly_2	✓	✗
Total ✓	9	4

Experiments with AST – Table 6.2. We compare Amber against Amber-Light on 11 benchmarks which are AST but not necessarily PAST and also cannot be split into PAST subprograms. Therefore, the SM-Rule is needed to certify AST. To the best of our knowledge, Amber is the first tool able to certify AST for such programs. Existing approaches like [ACN18] and [CH20] can only witness AST for non-PAST programs, if - intuitively speaking - the programs contain subprograms which are PAST. Therefore, we compared Amber only against Amber-Light on this set of examples. The benchmark *symmetric_2d_random_walk*, which Amber fails to certify as AST, models the symmetric random walk in $\mathbb{R}^2$ and is still out of reach of current automation techniques. In [MMKK18] the authors mention that a closed-form expression M and functions p and d satisfying the conditions of the SM-Rule have not been discovered yet. The benchmark *fair_in_limit_random_walk* involves non-constant probabilities and can therefore not be modeled as a Prob-solvable loop.

Experiments with non-AST – Table 6.3. We compare Amber against Amber-Light on 6 benchmarks which are not AST. To the best of our knowledge, Amber is the first tool able to certify non-AST for such programs, and thus we compared Amber only against Amber-Light. In [CNZ17], where the notion of repulsing supermartingales and the R-AST-Rule are introduced, the authors also propose automation techniques. However, the authors of [CNZ17] claim that their "experimental results are basic" and their computational methods are evaluated on only 3 examples, without having any available tool support. For the benchmarks in Table 6.3, the outcomes of Amber

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.3: 6 programs which are not AST.

Program	Amber	Amber-Light
biased_random_walk_nast_1	✓	✓
biased_random_walk_nast_2	✓	✓
biased_random_walk_nast_3	✓	✓
biased_random_walk_nast_4	✓	✓
binomial_nast	✓	✓
polynomial_nast	✗	✗
Total ✓	5	5

and Amber-Light coincide. The reason for this is R-AST-Rule's condition that the martingale expression has to have c-bounded differences. This condition forces a suitable martingale expression to be bounded by a linear function, which is also the reason why Amber cannot certify the benchmark *polynomial_nast*.

Experimental Summary. Our results from Tables 6.1-6.3 demonstrate that:

- Amber outperforms the state-of-the-art in automating PAST certification for Prob-solvable loops (Table 6.1).
- Complex probabilistic programs which are AST and not PAST as well as programs which are not AST can automatically be certified as such by Amber (Tables 6.2, 6.3).
- The relaxations of the proof rules introduced in Section 6.4 are helpful in automating the termination analysis of probabilistic programs, as evidenced by the performance of Amber against Amber-Light (Tables 6.1-6.3).

6.7 Related Work

Proof Rules for Probabilistic Termination. Several proof rules have been proposed in the literature to provide sufficient conditions for the termination behavior of probabilistic programs. The work of [CS13] uses martingale theory to characterize *positive almost sure termination (PAST)*. In particular, the notion of a ranking supermartingale (RSM) is introduced together with a proof rule (RSM-Rule) to certify PAST, as discussed in Section 6.3.1. The approach of [FFH15] extended this method to include (demonic) non-determinism and continuous probability distributions, showing the completeness of the RSM-Rule for this program class. The compositional approach proposed in [FFH15]

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

was further strengthened in [HFCG19] to a sound approach using the notion of *descent supermartingale map*. In [ACN18], the authors introduced *lexicographic* RSMs.

The SM-Rule discussed in Section 6.3.2 was introduced in [MMKK18]. It is worth mentioning that this proof rule is also applicable to non-deterministic probabilistic programs. The work of [HFC18] presented an independent proof rule based on supermartingales with lower bounds on conditional absolute differences. Both proof rules are based on supermartingales and can certify AST for programs that are not necessarily PAST. The approach of [TOUH18] examined martingale-based techniques for obtaining bounds on reachability probabilities — and thus termination probabilities— from an order-theoretic viewpoint. The notions of *nonnegative repulsing supermartingales* and *γ-scaled submartingales*, accompanied by sound and complete proof rules, have also been introduced. The R-AST-Rule from Section 6.3.3 was proposed in [CNZ17] mainly for obtaining bounds on the probability of stochastic invariants.

An alternative approach is to exploit weakest precondition techniques for probabilistic programs, as presented in the seminal works [Koz81, Koz85] that can be used to certify AST. The work of [MM05] extended this approach to programs with non-determinism and provided several proof rules for termination. These techniques are purely syntax-based. In [KKMO18] a weakest precondition calculus for obtaining bounds on expected termination times was proposed. This calculus comes with proof rules to reason about loops.

Automation of Martingale Techniques. The work of [CS13] proposed an automated procedure — by using Farkas' lemma — to synthesize *linear* (super)martingales for probabilistic programs with linear variable updates. This technique was considered in our experimental evaluation, cf. Section 6.6. The algorithmic construction of supermartingales was extended to treat (demonic) non-determinism in [CFNH18] and to polynomial supermartingales in [CFG16] using semi-definite programming. The recent work of [CH20] uses ω-regular decomposition to certify AST. They exploit so-called *localized* ranking supermartingales, which can be synthesized efficiently but must be linear.

Other Approaches. Abstract interpretation is used in [Mon01] to prove the probabilistic termination of programs for which the probability of taking a loop k times decreases at least exponentially with k. In [EGK12], a sound and complete procedure deciding AST is given for probabilistic programs with a finite number of reachable states from any initial state. The work of [NCH18] gave an algorithmic approach based on potential functions for computing bounds on the expected resource consumption of probabilistic programs. In [LLMR17], model checking is exploited to automatically verify whether a parameterized family of probabilistic concurrent systems is AST.

Finally, the class of Prob-solvable loops considered in this chapter extends [BKS19] to a wider class of loops. While [BKS19] focused on computing statistical higher-order moments, we address the termination behavior of probabilistic programs. The related approach of [GGH19] computes exact expected runtimes of constant probability

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

programs and provides a decision procedure for AST and PAST for such programs. Our programming model strictly generalizes the constant probability programs of [GGH19], by supporting polynomial loop guards, updates and martingale expressions.

6.8 Conclusion

This chapter reported on the automation of termination analysis of probabilistic while-programs whose guards and expressions are polynomial expressions. To this end, we introduced mild relaxations of existing proof rules for AST, PAST, and their negations, by requiring their sufficient conditions to hold only eventually. The key to our approach is that the structural constraints of Prob-solvable loops allow for automatically computing almost sure asymptotic bounds on polynomials over program variables. Prob-solvable loops cover a vast set of complex and relevant probabilistic processes including random walks and dynamic Bayesian networks [BKS20b]. Only two out of 50 benchmarks in [CS13, NCH18] are outside the scope of Prob-solvable loops regarding PAST certification. The almost sure asymptotic bounds were used to formalize algorithmic approaches for proving AST, PAST, and their negations. Moreover, for Prob-solvable loops four different proof rules from the literature uniformly come together in our approach.

Our approach is implemented in the tool Amber (github.com/probing-lab/amber), offering a fully automated approach to probabilistic termination. Our experimental results show that our relaxed proof rules enable proving probabilistic (non-)termination of more programs than could be treated before. A comparison to the state-of-art in automated analysis of probabilistic termination reveals that Amber significantly outperforms related approaches. To the best of our knowledge, Amber is the first tool to automate AST, PAST, non-AST and non-PAST in a single tool-chain.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 7

The Probabilistic Termination Tool Amber

This chapter is based on the following article [MBKK22]:

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. Formal Methods Syst. Des., 2022.

The article is an extended version of the conference paper [MBKK21b]:

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. In Proc. of FM, 2021.

7.1 Problem Statement

Probabilistic programming obviates the need to manually provide inference methods for different stochastic models and enables rapid prototyping [Gha15, BKS20a]. Automated formal verification of probabilistic programs, however, is still in its infancy. In this chapter, we provide a step towards closing this gap when it comes to automating the termination analysis of probabilistic programs, which is an active research topic [EGK12, CS13, FFH15, CFG16, ACN18, CNZ17, MMKK18, HFC18, CH20, CGMZ22]. Probabilistic programs are almost-surely terminating (AST) if they terminate with probability 1 on all inputs. They are positively AST (PAST) if their expected runtime is finite [BG05].

Addressing the challenge of (P)AST analysis, we describe Amber, a fully automated software artifact to prove/disprove (P)AST. Amber supports the analysis of a class of polynomial probabilistic programs. Probabilistic programs supported in our programming model consist of single loops whose body is a sequence of random assignments

with acyclic variable dependencies. Moreover, Amber's programming model supports programs parametrized by symbolic constants and drawing from common probability distributions, such as *Uniform* or *Normal* (Section 7.3). To automate termination analysis, Amber automates relaxations of various existing martingale-based proof rules ensuring (non-)(P)AST [CFN20] and combines symbolic computation with asymptotic bounding functions (Sections 7.4-7.5). Amber certifies (non-)(P)AST without relying on user-provided templates/bounds over termination conditions. Our experiments demonstrate Amber outperforming the state-of-the-art in automated termination analysis of probabilistic programs (Section 7.7). Our tool Amber is available at

`https://github.com/probing-lab/amber`.

Related Work. While probabilistic termination is an actively studied research challenge, tool support for probabilistic termination is limited. We compare Amber with computer-aided verification approaches proving probabilistic termination. The tools `MGen` [CS13] and `LexRSM` [ACN18] use linear programming techniques to certify PAST and AST, respectively. A modular approach verifying AST was recently proposed in [CH20]. Automated techniques for refuting (P)AST were proposed in [CNZ17] and techniques for synthesizing polynomial ranking supermartingales using semi-definite programming in [CFG16]. The work [CGMZ22] introduced a sound and relatively complete algorithm to prove lower bounds on termination probabilities. However, the works of [CH20, CFG16, CNZ17, CGMZ22] lack full tool support. The recent tools `Absynth` [NCH18], `KoAT2` [MHG21] and `ecoimp` [AMS20] can establish upper bounds on expected costs, therefore also on expected runtimes, and thus certify PAST. While powerful on respective AST/PAST domains, we note that none of the aforementioned tools support both proving and disproving AST or PAST. Our tool Amber is the first to prove and/or disprove (P)AST in a unifying manner. Our recent work [MBKK21a] introduced relaxations of existing proof rules for probabilistic (non-)termination together with automation techniques based on *asymptotic bounding functions*. We utilize these proof rule relaxations in Amber and extend the technique of asymptotic bounds to programs drawing from various probability distributions and including symbolic constants.

Contributions. This chapter describes the tool Amber, a fully automatic open-source software artifact for certifying probabilistic (non-)termination.

- We provide techniques to extend the method of asymptotic bounds for probabilistic termination to support symbolic constants and drawing from common probability distributions which can be continuous, discrete, finitely- or infinitely supported (Section 7.3 and Section 7.5).
- We describe the various components and give an overview of the implementation principles of Amber (Section 7.6).

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

- We extensively compare Amber to related tools and report on our experimental findings (Section 7.7).
- We provide a benchmark suite of 50 probabilistic programs as a publicly available repository of probabilistic program examples (Section 7.7).

Extending [MBKK21b], we provide the theoretical prerequisites in Section 7.2. Sections 7.4-7.5 complement [MBKK21b] with new material introducing the supported termination proof rules and illustrating Amber's algorithmic approach towards termination analysis. Moreover, Section 7.5 describes extensions of the asymptotic bound algorithm [MBKK21a] to programs drawing from common probability distributions and containing symbolic constants. Section 7.6 goes beyond the details of [MBKK21b] in describing the different components of Amber and their interplay.

7.2 Preliminaries

By $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$ we denote the set of natural, rational, and real numbers, respectively. We write $\overline{\mathbb{Q}}$, the real algebraic closure of $\mathbb{Q}$, to denote the field of real algebraic numbers. We write $\overline{\mathbb{Q}}[x_1, \dots, x_k]$ for the polynomial ring of all polynomials $P(x_1, \dots, x_k)$ in k variables $x_1, \dots, x_k$ with coefficients in $\overline{\mathbb{Q}}$ (with $k \in \mathbb{N}$ and $k \neq 0$). We assume the reader to be familiar with Markov chains and probability theory in general. For more details we refer to [KSK76, Dur19].

7.2.1 C-finite Recurrences

We recall some relevant notions and results from algebraic recurrences. For more details we refer to [EvdPSW03, KP11]. A sequence in $\overline{\mathbb{Q}}$ is a function $f\colon \mathbb{N} \to \overline{\mathbb{Q}}$. A recurrence of order r for a sequence is an equation $f(i{+}r) = \mathcal{R}(f(i{+}r{-}1), \dots, f(i{+}1), f(i), i)$, for some function $\mathcal{R}\colon \mathbb{R}^{r+1} \to \mathbb{R}$. A special class of recurrences relevant to our approach are *linear recurrences with constant coefficients*, or *C-finite recurrences* in short. A C-finite recurrence for a sequence $f(i)$ is an equation of the form

$$f(i{+}r) = a_{r-1}{\cdot}f(i{+}r{-}1) + a_{r-2}{\cdot}f(i{+}r{-}2) + \dots + a_0{\cdot}f(i) \tag{7.1}$$

where $a_0, \dots, a_{r-1} \in \overline{\mathbb{Q}}$ are constants and $a_0 \neq 0$. A sequence satisfying a C-finite recurrence (7.1) is a *C-finite sequence* and is uniquely determined by its initial values $f(0), \dots, f(r{-}1) \in \overline{\mathbb{Q}}$. The terms of a C-finite sequence can be written in closed-form as exponential polynomials (i.e. as a linear combination of exponential sequences and polynomials), depending only on i and the initial values of the sequence. That is, if $f(i)$ is C-finite, then $f(i) = \sum_{n=0}^{k} P_n(i){\cdot}\lambda_n^i$ where all $P_n(i) \in \overline{\mathbb{Q}}[i]$ and all $\lambda_n \in \overline{\mathbb{Q}}$; we refer to $\sum_{n=0}^{k} P_n(i)\lambda_n^i$ as an exponential polynomial. Moreover, every polynomial exponential over $i \in \mathbb{N}$ is the solution of some C-finite recurrence. Importantly, closed-forms of C-finite sequences always exist and are computable [KP11].

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

bop $\in \{+, -, *, /\}$, *cop* $\in \{>, <\}$ *dist* $\in$ {uniform, gauss, laplace, bernoulli, binomial, geometric, hypergeometric, exponential, beta, chi-squared, rayleigh}

$\langle program \rangle$ ::= $\langle i_assign \rangle^*$ while $\langle poly \rangle$ $\langle cop \rangle$ $\langle poly \rangle$: $\langle body \rangle$

$\langle body \rangle$ ::= $\langle rv_assign \rangle^*$ ($\langle rv_assign \rangle$ | $\langle v_assign \rangle$) $\langle v_assign \rangle^*$

$\langle i_assign \rangle$ ::= $\langle var \rangle$ = $\langle const \rangle$ | $\langle var \rangle$ = $\langle rv_expr \rangle$

$\langle rv_assign \rangle$ ::= $\langle var \rangle$ = $\langle rv_expr \rangle$

$\langle v_assign \rangle$ ::= $\langle var \rangle$ = $\langle branches \rangle$

$\langle rv_expr \rangle$ ::= RV($\langle dist \rangle$ [, $\langle const \rangle$]*)

$\langle branches \rangle$ ::= $\langle poly \rangle$ | $\langle poly \rangle$ @ $\langle const \rangle$; $\langle branches \rangle$

$\langle poly \rangle$ ::= $p \in C[V]$

$\langle sym \rangle$ ::= [a-zA-Z][a-zA-Z0-9]*

$\langle var \rangle$ ::= [a-zA-Z][a-zA-Z0-9]*

$\langle const \rangle$ C ::= $n \in \mathbb{N}$ | $\langle sym \rangle$ | - $\langle const \rangle$ | $\langle const \rangle$ $\langle bop \rangle$ $\langle const \rangle$

Figure 7.1: The input syntax of Amber, where $C[V]$ denotes the set of polynomials in V (program variables) with coefficients from C (constants); ** is used as the power operator to express polynomials in $\langle poly \rangle$.

Special recurrences relevant for the internals of Amber are inhomogeneous linear recurrences with exponential polynomials as inhomogeneous parts:

$$f(i{+}1) = a{\cdot}f(i) + \sum_{n=0}^{k} P_n(i){\cdot}\lambda_n^i, \tag{7.2}$$

where $a \in \overline{\mathbb{Q}}$. Every sequence satisfying a recurrence of form (7.2) is C-finite, because the inhomogeneous part in (7.2) is C-finite and all components of systems of C-finite sequences are C-finite. Moreover, if $a \geq 0$ and all $\lambda_n \geq 0$, then the exponential polynomial closed-form for $f(i)$ only contains positive exponential terms. For such exponential polynomials the limit $l \in \mathbb{R} \cup \{-\infty, \infty\}$ as $i \to \infty$ can always be computed [Gru96].

7.3 Amber: Programming Model

Amber analyzes the probabilistic termination behavior of a class of probabilistic programs involving polynomial arithmetic and random drawings from common probability distributions, parameterized by symbolic constants. The grammar in Figure 7.1 defines the input programs to Amber. Inputs to Amber consist of an initialization part and

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

a while-loop, whose guard is a polynomial inequality over program variables. The initialization part is a sequence of assignments either assigning (symbolic) constants or values drawn from probability distributions. Within the loop body, program variables are updated with either (i) a value drawn from a distribution or (ii) one of multiple polynomials over program variables with some probability. Additional to the structure imposed by the grammar in Figure 7.1, input programs are required to satisfy the following *structural constraint: Each variable updated in the loop body only depends linearly and non-negatively on itself and in a polynomial way on variables preceding it in the loop body.* On a high level, this structural constraint is what enables the use of algebraic recurrence relations in probabilistic termination analysis. More concretely, the restriction to linear self-dependencies is necessary to ensure that the resulting recurrence relations (cf. Section 7.5) are C-finite and guaranteed to have computable closed-forms. Even seemingly simple first-order *quadratic* recurrences are problematic: the recurrence $f(n{+}1) = r \cdot f(n)^2 - r \cdot f(n)$ does not have known analytical closed-form solutions for most values of $r \in \mathbb{R}$ [Mar20]. Furthermore, coefficients in linear self-dependencies are required to be non-negative to prevent oscillating dynamics. For instance, the sequence defined by the recurrence $f(n{+}1) = -1 \cdot f(n)$ oscillates between 1 and -1 for $f(0) = 1$. Amber computes asymptotic bounds for monomials in program variables using recurrences. A central requirement of the termination analysis technique implemented in Amber (cf. Section 7.5) is that the asymptotic bounds are eventually monotone and non-negative or non-positive. Restricting coefficients in linear self-dependencies to be non-negative ensures this necessary property. Moreover, the algorithm computing asymptotic bounds for a program variable x first recursively computes the asymptotic bounds for all (monomials in) program variables on which x depends. Hence, to ensure termination, the dependencies among variables must be acyclic. This is guaranteed by restricting variable dependencies to preceding variables.

Despite the syntactical restrictions, most existing benchmarks on automated probabilistic termination analysis [MBKK21a] and dynamic Bayesian networks [BKS20b] can be encoded in our programming language. Figure 7.2 shows three example input programs to Amber. For each of these examples, Amber automatically infers the respective termination behavior, by relying on its workflow described in Section 7.6. Our programming model extends *Prob-solvable loops* [BKS20c] with polynomial inequalities as loop guards. For a loop with loop guard $\mathcal{G}$ of the form $P > Q$ we write G for the expression $P{-}Q$. In the sequel, we refer to programs of our programming model simply by *loops* or *programs*.

7.4 Proof Rules for Probabilistic Termination

We now describe the theoretical foundations of existing proof rules for establishing probabilistic (non-)termination, which are used and further refined in Amber (Section 7.5).

Loop Space. Operationally, every program loop represents a Markov chain (MC) with state space $\mathbb{R}^m$ if the loop has m program variables. This MC in turn induces a canonical

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
x = RV(gauss, 0, 1)
y = RV(gauss, 0, 1)
while x**2+y**2 < c:
    s = RV(uniform, 1, 2)
    t = RV(gauss, 0, 1)
    x = x+s @1/2; x+2*s
    y = y+x+t**2 @1/2; y-x-t**2
```
(a)

```
x = x0
while x > 0:
    x = x+c @1/2; x-c
```
(b)

```
x = x0
while x > 0:
    x = x+c @1/2+e; x-c
```
(c)

Figure 7.2: Examples of programs supported by Amber, with symbolic constants $c, x0, e \in \mathbb{R}^+$; program 7.2a is PAST; program 7.2b is AST but not PAST; program 7.2c is not AST

probability space. In this way, every loop $\mathcal{L}$ is associated with a (filtered) probability space $(\Omega^{\mathcal{L}}, \Sigma^{\mathcal{L}}, (Run_i^{\mathcal{L}}), \mathbb{P}^{\mathcal{L}})$. We omit the superscripts if $\mathcal{L}$ is clear from context. The sample space Ω is the set of all infinite program runs. More precisely, if $\mathcal{L}$ has m program variables, then $\Omega = (\mathbb{R}^m)^\omega$. Σ is the σ-algebra constructed from all finite program run prefixes. The purpose of the loop filtration (Run_i) is to capture the information gain as the loop is executed. Every σ-algebra Run_i of the filtration is constructed from all finite program run prefixes of length $i+1$. In this way, Run_i allows measuring events concerned with the first i loop iterations. Finally, $\mathbb{P}$ is the probability measure defined according to the intended semantics of the program statements. For a formal definition of $\mathbb{P}$ and more details regarding the semantics of probabilistic loops we refer to [MBKK21a]. For an expression E over the program variables, E_i denotes the random variable mapping a program run to the value of E after the i-th iteration. With the loop space at hand, the notions of AST and PAST (originally considered in [Sah78]) can be defined in terms of a random variable capturing the termination time.

Definition 55 ((Positive) Almost-sure Termination)**.** *The* termination time *of a loop $\mathcal{L}$ with guard $\mathcal{G}$ is the random variable $T^{\neg\mathcal{G}}$:*

$$T^{\neg\mathcal{G}} : \Omega \to \mathbb{N} \cup \{\infty\} \quad \textit{with} \quad T^{\neg\mathcal{G}}(\vartheta) := \inf\{i \in \mathbb{N} \mid \vartheta_i \models \neg\mathcal{G}\}$$

$\mathcal{L}$ is said to be almost-surely terminating (AST) *if $\mathbb{P}(T^{\neg\mathcal{G}} < \infty) = 1$ and* positively almost-surely terminating (PAST) *if $\mathbb{E}(T^{\neg\mathcal{G}}) < \infty$.*

7.4.1 Termination Proof Rules

Despite the fact that the problems of AST and PAST are undecidable in general [KK15], several proof rules – sufficient conditions – have been developed to certify PAST, AST and their negations. On a high level, many proof rules require a witness in the form of an arithmetic expression over program variables that satisfies some conditions based on martingale theory. Amber utilizes three martingale-based proof rules from the literature,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

one for PAST [CS13, FFH15], one for AST [MMKK18] and one rule able to certify non-AST and non-PAST [CNZ17]. In [MBKK21a], the authors relaxed these three proof rules such that their conditions only need to hold eventually rather than always. A property $P(i)$ holds eventually, if $P(i)$ is true for all $i \geq i_0$ for some $i_0 \in \mathbb{N}$. These relaxations enable using *asymptotic reasoning* when automating the respective proof rules. Amber implements the relaxed versions of these proof rules by choosing the loop guard expression G (defined as $P - Q$ for loop guard $\mathcal{G} = P > Q$ for polynomials P and Q) as the potential witness and checking the proof rule conditions using asymptotic bounds (cf. Section 7.5). To certify PAST, Amber uses the *Ranking SM-Rule*.

Theorem 43 (Ranking SM-Rule [CS13, FFH15, MBKK21a])**.** *Let $\mathcal{L}$ be a probabilistic loop with guard $\mathcal{G}$. Assume the following condition holds eventually:*

$$\mathbb{E}(G_{i+1} - G_i \mid Run_i) \leq -\epsilon, \text{ for some } \epsilon > 0$$

Then, $\mathcal{L}$ is PAST. In this case, G is called a ranking supermartingale.

Probabilistic programs with an infinite expected runtime can still terminate with probability one. The symmetric one-dimensional random walk (Figure 7.2b) is a well-known example that is AST but not PAST. For such programs, the *SM-Rule* provides a solution to certify AST.

Theorem 44 (SM-Rule [MMKK18, MBKK21a])**.** *Let $\mathcal{L}$ be a probabilistic loop with guard $\mathcal{G}$, $d > 0$ and $p \in (0, 1]$. Assume the following conditions hold eventually:*

1. $\mathbb{E}(G_{i+1} - G_i \mid Run_i) \leq 0$
2. $\mathbb{P}(G_{i+1} - G_i \leq -d \mid Run_i) \geq p$

Then, $\mathcal{L}$ is AST. If G satisfies condition 1, it is called a supermartingale.

For non-terminating programs, the *Repulsing SM-Rule* can certify their divergence. It is capable of certifying non-AST as well as non-PAST.

Theorem 45 (Repulsing SM-Rule [CNZ17, MBKK21a])**.** *Let $\mathcal{L}$ be a probabilistic loop with guard $\mathcal{G}$. Assume $\forall i : \mathbb{P}(\mathcal{G}_i) > 0$ and that the following conditions hold eventually:*

1. $\mathbb{E}(G_i - G_{i+1} \mid Run_i) \leq -\epsilon$, *for some* $\epsilon > 0$
2. $|G_i - G_{i+1}| < c$, *for some* $c > 0$.

Then, $\mathcal{L}$ is not *AST. If all conditions are true with the domain of ϵ in condition 1 relaxed to include 0 (i.e. $\epsilon \geq 0$), then $\mathcal{L}$ is* not *PAST.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The *Ranking SM-Rule* as well as the *SM-Rule* require G, and the *Repulsing SM-Rule* $-G$, to be a supermartingale. An expression E cannot be a supermartingale if $\mathbb{E}(E_{i+1}-E_i) > 0$ [MBKK21a]. The tool `Mora` [BKS19, BKS20c] can compute an exponential polynomial closed-form of $\mathbb{E}(E_{i+1}-E_i)$ for Amber's input programs. In Amber, we utilize the functionality of `Mora` to compute a closed-form of $\mathbb{E}(G_{i+1}-G_i)$. Amber uses this closed-form in trying to rule-out the applicability of some of the proof rules.

7.5 Effective Termination Analysis through Asymptotic Bounds

The conditions in the proof rules from Section 7.4.1 contain three *types of inequalities*:

- **Type 1:** Inequalities over conditional expected values, as $\mathbb{E}(G_{i+1}-G_i \mid Run_i) \leq -\epsilon$ (≤ 0) in the *Ranking SM-Rule* (*SM-Rule*) for proving PAST (AST).
- **Type 2:** Inequalities over conditional probabilities, as $\mathbb{P}(G_{i+1}-G_i \leq -d \mid Run_i) \geq p$ in the *SM-Rule* for establishing AST.
- **Type 3:** Inequalities over absolute values, as $|G_i-G_{i+1}| < c$ in the *Repulsing SM-Rule* for disproving AST.

In the sequel we detail how these three type of inequalities are handled in Amber for proving/disproving (P)AST .

Type 1. For Amber's programming model, the expression $\mathbb{E}(G_{i+1}-G_i \mid Run_i)$ is a polynomial in the program variables. For the program in Figure 7.2a we have $G = c-x^2-y^2$. The expression $\mathbb{E}(G_{i+1}-G_i \mid Run_i) = \mathbb{E}(G_{i+1} \mid Run_i)-G_i$ can be computed by starting with G, substituting left-hand sides of assignments by right-hand sides in a bottom-up fashion, averaging over probabilistic statements and finally subtracting G. For Figure 7.2a, this leads to the polynomial $\mathbb{E}(G_{i+1}-G_i \mid Run_i) = -x_i^2-11x_i-{}^{115}/_{6}$. Thus, the expected change of the loop guard from an arbitrary iteration i to iteration $i+1$ is $-x_i^2-11x_i-{}^{115}/_{6}$, where x_i is the value of program variable x after iteration i. For an input program and a polynomial *poly*, $\mathbb{E}(poly_{i+1} \mid Run_i)$ itself is always a polynomial. That is because all expressions in probabilistic branching statements are polynomials, all branching probabilities are constants and all distributions input programs can draw from have constant parameters and thus also constant moments. Crucially, all inequalities in the termination proof rules only need to hold eventually. Therefore, knowing the asymptotic behavior of the polynomial $\mathbb{E}(G_{i+1}-G_i \mid Run_i)$ can be helpful in answering the respective inequalities: for instance, an asymptotic upper bound to $\mathbb{E}(G_{i+1}-G_i \mid Run_i)$ that tends to a negative number witnesses that eventually $\mathbb{E}(G_{i+1}-G_i \mid Run_i) \leq -\epsilon$ for some $\epsilon > 0$.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Type 2. After fixing the values drawn from distributions in the loop body at iteration i, every expression, and in particular G, can only progress to finitely many expressions in iteration $i{+}1$. We refer to these possible follow-up expressions, as *branches*. For the program in Figure 7.2b, the expression G $(= x)$ is either $x{+}c$ or $x{-}c$ after one iteration. If for at least one of these branches B of G, we have that eventually $B_i{-}G_i \leq -d$ for some $d > 0$ for any choice of values drawn from distributions, then it holds that $\mathbb{P}(G_{i+1}{-}G_i \leq -d \mid Run_i) \geq p$ for some $p > 0$. This holds, due to the fact that all probabilities in probabilistic branching statements are constant and non-zero. Similar to the inequalities of type 1, asymptotic bounds provide a method to answer inequalities of type 2: if for some branch B of G and any choice of values drawn from probability distributions, the polynomial $B_i{-}G_i$ obeys an asymptotic upper bound tending to a negative number, then it holds that eventually $\mathbb{P}(G_{i+1}{-}G_i \leq -d \mid Run_i) \geq p$ for some $d > 0$ and $p \in (0, 1]$.

Type 3. In contrast to the inequalities of type 2 that have to hold with at least some non-zero probability, inequalities of type 3 have to hold almost-surely, that means with probability one. Nevertheless, type 3 inequalities can be approached similarly as type 2 inequalities: if for *every* (in contrast to *some* as for type 2 inequalities) branch B of G and any choice of values drawn from probability distributions, eventually $|G_i{-}B_i| < c$ for some $c > 0$, then eventually and almost-surely $|G_i{-}G_{i+1}| < c$. In contrast to type 1 and 2 inequalities, tackling type 3 inequalities with asymptotic bounds requires one extra step. Due to the presence of the absolute value function, asymptotic upper bounds for the polynomials $G_i{-}B_i$ do not suffice. Additional to upper bounds, asymptotic lower bounds are needed. Given an asymptotic upper bound $u(i)$ and an asymptotic lower bound $l(i)$ for the polynomial $G_i{-}B_i$, $\max(-l(i), u(i))$ is an asymptotic upper bound for $|G_i{-}B_i|$.

7.5.1 Computing Asymptotic Bounds

We argued that all main conditions of the termination proof rules from Section 7.4.1 reduce to the task of finding asymptotic lower- and upper bounds for polynomials in program variables. For this purpose, Amber utilizes a recently introduced *bound algorithm* [MBKK21a]. The algorithm builds on the notion of dominant functions.

Definition 56 (Domination)**.** *Let f and g be two functions from $\mathbb{N}$ to $\mathbb{R}$. We say f* dominates *g if eventually $c{\cdot}f(i) \geq g(i)$ for some $c \in \mathbb{R}^+$. Let F be a finite set of functions from $\mathbb{N}$ to $\mathbb{R}$. A function $f \in F$ is* most dominant *with respect to F, if f dominates all functions in F. Similarly, f is* least dominant *with respect to F, if every $g \in F$ dominates f.*

The bound algorithm described in this section produces bounds in the form of exponential polynomials with positive exponential terms as mentioned in Section 7.2. For every such function f, we can always construct a monotonic and non-positive or non-negative function g with the same asymptotic behavior, meaning that g dominates f and f

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

dominates g. The function g can be established by simplifying f to its fastest increasing or decreasing term, its *leading term*. For instance, if $f(i) = i2^i - 2^i - i^2$, then $g(i) = i2^i$ is monotonic, non-negative, and has the same asymptotic behavior as f. In the remainder, we assume that every asymptotic lower- and upper bound is simplified to its leading term. Moreover, for two exponential polynomials with positive exponential terms, we can always decide which dominates the other by comparing their leading terms. We illustrate the algorithm for computing asymptotic bounds in the following example. For the algorithm's pseudo-code and further details, we refer to [MBKK21a].

Example 67. *Consider the following program:*

```
x = x0
y = y0
while y > 0:
    x = 2x+1 @1/2; x-1
    y = y + x**2 - x
```

Assume, we want to compute an asymptotic lower bound and asymptotic upper bound for the program variable y. This means that we are trying to find functions $l(i)$ and $u(i)$ such that eventually and almost-surely $c_1{\cdot}l(i) \leq y_i \leq c_2{\cdot}u(i)$ for some positive constants c_1 and c_2. For every iteration i, y_{i+1} is either equal to $y_i{+}4x_i^2{+}2x_i$ or equal to $y_i{+}x_i^2{-}3x_i{+}2$, both with probability $1/2$. These polynomials are the branches of y. The algorithm in [MBKK21a] first recursively computes asymptotic lower- and upper bounds for the monomials x and x^2 in order to construct bounds for y.

Asymptotic bounds for x: *The branches of x are $2x_i{+}1$ and $x_i{-}1$ with inhomogeneous parts 1 and -1 respectively. The bound algorithm first computes bounds for the inhomogeneous parts. Because the inhomogeneous parts are both constants, both their lower- and upper bounds are just given by the inhomogeneous parts themselves. This is the base case of the algorithm. The base case will always be reached, because of the constraint of Amber's programming model that the dependencies among program variables in the loop body are acyclic (cf. Section 7.3). The* recurrence coefficients *of x are 2 and 1 respectively. These are the constant coefficients of x_i in the branches of x. The results of [MBKK21a] establish that an upper bound for x is given by the solution of one of the following four recurrence relations:*

$$f(i+1) = a{\cdot}f(i) + 1, \;\; \textit{for } f(0) \in \{d, -d\}, a \in \{2, 1\} \tag{7.3}$$

The recurrences (7.3) are inhomogeneous first-order recurrences. Their recurrence coefficients are given by the minimum and maximum recurrence coefficients of x. The inhomogeneous term in (7.3) is the most dominant upper bound of the inhomogeneous parts of x. The initial values of the recurrences (7.3) are d or $-d$ for a positive symbolic constant d. With a simple static analysis, Amber establishes that the program variable x can become positive as well as negative. Because x can be positive, d is among the initial values, and because x can be negative $-d$ is also required as an initial value. The solutions (closed-forms) to the four recurrences of $f(i)$ are respectively given by

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- $(d{+}1)2^i{-}1$;
- $(1{-}d)2^i{-}1$;
- $i{+}d$;
- $i{-}d$.

According to [MBKK21a], one of these four solutions is an upper bound to x. The closed form $(d{+}1)2^i{-}1$ of $f(i)$ dominates all other solutions. As we are only interested in asymptotic bounds modulo a constant factor, an asymptotic upper bound for x is therefore given by the leading term of $(d{+}1)2^i{-}1$; that is, x is asymptotically upper bounded by 2^i.

An asymptotic lower bound for x is computed analogously as the least *dominant solution to one of the recurrences*

$$f(i+1) = a{\cdot}f(i) - 1, \text{ for } f(0) \in \{d, -d\}, a \in \{2, 1\}. \tag{7.4}$$

In contrast to the upper bound computation, the inhomogeneous term in (7.4) *is given by the* least dominant lower bound *of the inhomogeneous parts of x, i.e. -1. The leading term in the least dominant solution of the recurrences* (7.4) *is -2^i and provides an asymptotic lower bound for x. Consequently, we established that eventually and almost surely*

$$c_1{\cdot}(-2^i) \leq x_i \leq c_2{\cdot}2^i \text{ for some } c_1, c_2 \in \mathbb{R}^+.$$

An absolute bounding function of x is an asymptotic bound for $|x_i|$ and is given by the most dominant function of $u(i) = 2^i$ and $-l(i) = -(-2^i)$. Note, that all recurrences in (7.3)–(7.4) *are first-order inhomogeneous linear recurrences with non-negative coefficients and exponential polynomials as inhomogeneous parts such that all exponential terms are positive. As argued in Section 7.2, recurrences of this type can always be solved automatically and lead to solutions for which their limits can be computed.*

Asymptotic bounds for x^2: *The branches of the monomial x^2 are $4x^2{+}4x{+}1$ and $x^2{-}2x{+}1$. Therefore the recurrence coefficients are given by 4 and 1. The inhomogeneous parts are $4x{+}1$ and $-2x{+}1$. Utilizing the already computed bounds for x, we get that $4x{+}1$ as well as $-2x + 1$ are asymptotically upper bounded by 2^i and lower bounded by -2^i. Hence, the most dominant upper bound of the inhomogeneous parts is 2^i and the least dominant lower bound is -2^i. Following the bound algorithm of [MBKK21a], we get that an asymptotic upper bound is given by the most dominant solution of the following recurrences:*

$$f(i+1) = a{\cdot}f(i) + 2^i, \text{ for } f(0) \in \{d\}, a \in \{4, 1\}$$

Computing the solutions of these recurrences and taking the leading term of the most dominant solution leads to the asymptotic upper bound 4^i for x^2. Note that the possible initial values are restricted to the positive constant d, as x^2 can never be negative. An

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

asymptotic lower of -2^i can be computed analogously. However, due to the non-negativity of x^2, the constant 0 is a tighter lower bound which is taken into account by the bound algorithm.

Asymptotic bounds for y: *Finally, we can compute asymptotic bounds for y using the bounds for x and x^2. The branches of y are $y+4x^2+2x$ and $y+x^2-3x+2$ with inhomogeneous parts $4x^2+2x$ and x^2-3x+2, respectively. Moreover, y has a single recurrence coefficient of 1. An asymptotic upper bound for the inhomogeneous part x^2-3x+2 can be established by substituting the bounds for the individual monomials. For x^2 we substitute its upper bound and for x its lower bound, due to the negative coefficient of x in the respective branch. For x^2-3x+2 this leads to an asymptotic upper bound of 4^i and an asymptotic lower bound of -2^i. Likewise, for the inhomogeneous part $4x^2+2x$, we get an asymptotic upper bound of 4^i and an asymptotic lower bound of -2^i. Therefore, the most dominant upper bound of the inhomogeneous parts is 4^i, and the least dominant lower bound is -2^i. Similar to the bounds computations for x and x^2, an asymptotic upper bound is given by the most dominant solution of*

$$f(i+1) = a \cdot f(i) + 4^i, \text{ for } f(0) \in \{d, -d\}, a \in \{1\}.$$

The leading term of the most dominant solution is 4^i and represents an asymptotic upper bound for y. An asymptotic lower bound for y of -2^i can be computed analogously.

The bound algorithm introduced in [MBKK21a] only supports programs of Amber's programming model, where every assignment in the loop body is a probabilistic branching statement over polynomials. In the remainder of this section, we describe how the techniques of [MBKK21a] can be extended to support symbolic constants and drawing from common probability distributions with constant parameters.

7.5.2 Supporting Symbolic Constants

A symbolic constant represents an arbitrary number from an infinite set of real numbers. For example, the program in Figure 7.2b encodes a symmetric one-dimensional random walk with symbolic step size c. For our purposes, defining a symbolic constant c to semantically represent *any* arbitrary real number $c \in \mathbb{R}$ is problematic, as illustrated in the following example.

Example 68. *Consider the following program with symbolic constant c:*

```
x = 1
while x > 0:
    x = x+c @1/2; x
```

Following the bound algorithm of [MBKK21a] for x would result in the lower bound of x being the least dominant of $c \cdot i$ and 1. Now, if c semantically represents an arbitrary *real number, we cannot conclusively decide whether $c \cdot i$ or 1 is more dominant: if $c > 0$, then $c \cdot i$ dominates 1 and if $c \leq 0$, then 1 dominates $c \cdot i$.*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To remedy the problem illustrated in the previous example, Amber adopts the semantic that symbolic constants represent an arbitrary *positive* real number. Negative constants can be modeled with the explicit use of "$-$". Still, the bound algorithm is incomplete for input programs with positive symbolic constants. A counter-example can be constructed from Example 68 by replacing c with $c{-}d$ where both c and d are symbolic constants. Now, the lower bound for the variable x is the least dominant of $(c{-}d){\cdot}i$ and 1 which cannot be answered without a case distinction involving the symbolic constants c and d. Nevertheless, experiments show that adopting the semantic of positive symbolic constants is useful and provides a solution to many challenging benchmarks (cf. Section 7.7).

7.5.3 Supporting Common Probability Distributions

Amber supports programs drawing from various common probability distributions with constant parameters (cf. Figure 7.1). The first key property of every supported distribution $\mathcal{D}$ is that $\mathbb{E}(\mathcal{D}^p)$ *exists and is computable for every* $p \in \mathbb{N}$. This ensures that for any polynomial *poly* in program variables, $\mathbb{E}(poly_{i+1} \mid Run_i)$ remains a polynomial.

The second key property is that *$\mathcal{D}$'s support is an interval.* More precisely, if $\mathcal{D}$ is continuous, then $supp(\mathcal{D}) = (a, b)$ (or $[a, b]$) for $a, b \in \mathbb{R} \cup \{-\infty, \infty\}$ and if $\mathcal{D}$ is discrete, then $supp(\mathcal{D}) = \{a, a{+}1, \dots, b{-}1, b\}$ for $a, b \in \mathbb{R} \cup \{-\infty, \infty\}$. Because the support of $\mathcal{D}$ is an interval, tight bounds for the support of $\mathcal{D}^p$ for $p \in \mathbb{N}$ can be computed using interval arithmetic.

Amber extends the main bound algorithm to support programs drawing from such distributions. Let x be a program variable drawing from a probability distribution, M a monomial of program variables not containing x, and $p \in \mathbb{N}$. Then Amber computes the asymptotic bounds for the monomial $x^p{\cdot}M$ in the following way: First, an upper bound $u(i)$ and lower bound $l(i)$ for M are computed recursively. Second, the boundaries a and b (with $a \leq b$) of the support of x^p are computed using interval arithmetic. Finally, an upper bound (lower bound) of $x^p \cdot M$ is given by the most dominant function (least dominant function) of $a{\cdot}u(i)$, $b{\cdot}u(i)$, $a{\cdot}l(i)$ and $b{\cdot}l(i)$. Due to Amber supporting unbounded distributions, a, b, $l(i)$ and $u(i)$ can be $\pm\infty$. The handle calculations involving infinities, we use the usual arithmetic rules for $\pm\infty$: $x{+}\infty{=}\infty$; $x{-}\infty{=}{-}\infty$; if $x > 0$ then $x{\cdot}\infty{=}\infty$; if $x < 0$ then $x{\cdot}\infty{=}{-}\infty$. Note that, because the asymptotic bounds $b(i)$ are always monotonic and non-positive or non-negative, Amber can always decide whether $\infty \cdot b(i)$ is ∞ or $-\infty$ (if $b(i)$ itself is not $\pm\infty$ or 0). In case of indeterminate forms ($\infty{-}\infty$ and $0{\cdot}\infty$), Amber aborts the bound computation and resorts to the loosest possible bounds of $-\infty$ and ∞.

Example 69. *Let x be a program variable drawing from a continuous uniform distribution between -1 and 2 and M a monomial of program variables not containing x. Assume M obeys an asymptotic lower bound $l(i) = i$ and an asymptotic upper bound $u(i) = i^2$. Asymptotic bounds for $x^3{\cdot}M$ are computed as follows. We have $supp(x^3) = (-1, 8)$. Let $F = \{-1{\cdot}i^2, 8{\cdot}i^2, -1{\cdot}i, 8{\cdot}i\}$. The most dominant function in F is $8{\cdot}i^2$ and because positive*

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

constant factors of asymptotic bounds can be absorbed, i^2 is an asymptotic upper bound for $x^3 \cdot M$.

Although Amber requires the parameters of distributions to be constant, some state-dependent parameters can be modeled through distribution transformations. For instance, $\text{Normal}(poly, c)$ is equivalent to $poly + \text{Normal}(0, c)$. Likewise, $\text{Uniform}(poly_1, poly_2)$ is equivalent to $poly_1 + (poly_2 - poly_1) \cdot \text{Uniform}(0, 1)$ for the continuous uniform distribution. Similar transformations exist for other distributions.

With the generalized bound algorithm, Amber can compute asymptotic upper- and lower bounds for polynomials of program variables, even if the programs draw from probability distributions. However, this generalization alone is not sufficient, in particular for the *SM-Rule*.

Example 70. *The following program models a symmetric 1-dimensional random walk:*

```
x = 1
while x > 0:
    s = RV(uniform, -1, 1)
    x = x+s
```

The program can be proven to be AST using the SM-Rule*. We have $G = x$. Amber computes $\mathbb{E}(x_{i+1} - x_i \mid Run_i) = 0$ and hence establishes condition 1 of the* SM-Rule*. However, condition 2 poses a problem. Amber extracts the only branch of x, that is $x+s$, and computes the asymptotic bounds for $x+s-x = s$, resulting in the lower bound -1 and the upper bound 1. Because the upper bound is always positive, without further information Amber cannot conclude that x decreases by some constant with constant probability. For this example, the problem can be mitigated by constructing an equivalent program in which the variable s is split into three different parts:*

```
x = 1
while x > 0:
    s1 = RV(uniform, -1, -1/2)
    s2 = RV(uniform, -1/2, 1/2)
    s3 = RV(uniform, 1/2, 1)
    x  = x+s1 @1/4; x+s2 @1/2; x+s3 @1/4;
```

Now, for the branch $x+s1$, Amber established the bounds for $x+s1-x = s1$ to be -1 and $-1/2$. As the asymptotic upper bound is negative, Amber concludes that eventually x decreases by at least some constant with at least some constant probability. Thus, condition 2 of the SM-Rule *is verified and Amber certifies the program to be AST.*

In Example 70, the program variable s is drawn from a uniform distribution whose support contains positive and negative values. Without further information, Amber can only establish that s is lower bounded by -1 and upper bounded by 1 but is oblivious to the fact that there is a constant probability such that s is negative. In Example 70, this

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

fact is made explicit to Amber through constructing an equivalent program by splitting the uniform distribution into three different parts such that two parts are bounded away from 0. In general, this exact approach is not feasible when drawing from more complex distributions. However, note that neither the exact probability of $1/4$ of branch $x+s1$, nor the exact distribution of $s1$ are necessary to answer condition 2 of the *SM-Rule*. It suffices that the branch is associated with *some* constant positive probability and that the support of $s1$ is strictly negative and bounded away from 0. In this sense, the only relevant information about the distribution of s is its support.

Following this observation, Amber implements an over-approximation when considering the branches of expressions, abstracting from concrete distributions: let B be a branch containing a variable s drawn from a probability distribution $\mathcal{D}$ with support boundaries $a < 0$ and $b > 0$. With $Tr(\mathcal{D}; \alpha, \beta)$ we denote the *truncated distribution* of $\mathcal{D}$ with lower bound α and upper bound β. Assume $\mathcal{D}$ is a continuous distribution. For discrete distributions, the following process is analogous. Let $\epsilon > 0$ with $|a| > \epsilon$ and $b > \epsilon$ and define

$$p_1 := \int_a^{-\epsilon} d\mathcal{D} \qquad p_2 := \int_{-\epsilon}^{\epsilon} d\mathcal{D} \qquad p_3 := \int_{\epsilon}^{b} d\mathcal{D}.$$

We have $p_1, p_2, p_3 > 0$ and can split $\mathcal{D}$ into three different parts such that one part has strictly negative, one part strictly positive support, and both supports are bounded away from 0. With $C \sim Categorical(3, p_1, p_2, p_3)$ we have

$$\begin{aligned} s \sim\ & [C = 1] \cdot Tr(\mathcal{D}, a, -\epsilon) + \\ & [C = 2] \cdot Tr(\mathcal{D}, -\epsilon, \epsilon) + \\ & [C = 3] \cdot Tr(\mathcal{D}, \epsilon, b). \end{aligned}$$

$[P]$ denotes the Iverson bracket which equals 1 if P is true and 0 otherwise. Now, the goal of Amber is to split the branch B containing s into three branches, where s is replaced by $s_1 \sim Tr(\mathcal{D}, a, -\epsilon)$, $s_2 \sim Tr(\mathcal{D}, -\epsilon, \epsilon)$ and $s_3 \sim Tr(\mathcal{D}, \epsilon, b)$ respectively. However, the distributions of s_1, s_2, and s_3 are potentially more complex than the original distribution $\mathcal{D}$, and the constants p_1, p_2, and p_3 each require solving an integral. Amber overcomes these issues with over-approximation. As previously argued, the precise values of p_1, p_2, and p_3 are not needed and only required to be positive, which is guaranteed. Moreover, the only relevant information about the distributions of s_1, s_2, and s_3 are their supports. Therefore, Amber over-approximates $Tr(\mathcal{D}, \alpha, \beta)$ by $Symb(\alpha, \beta)$, where $Symb(\alpha, \beta)$ represents any distribution $\mathcal{D}'$ with $supp(\mathcal{D}') = [\alpha, \beta]$. With $v \sim Symb(\alpha, \beta)$ we denote that $v \sim \mathcal{D}'$ for some $\mathcal{D}' \in Symb(\alpha, \beta)$. Consequently, for condition 2 of the *SM-Rule* and for condition 2 of the *Repulsing SM-Rule*, Amber splits every branch B containing a variable s drawn from a probability distributions $\mathcal{D}$ with mixed-sign support into three new branches $B[s/s_1]$, $B[s/s_2]$, and $B[s/s_3]$. The substituted variables are such that $s_1 \sim Symb(a, -\epsilon)$, $s_2 \sim Symb(-\epsilon, \epsilon)$, and $s_3 \sim Symb(\epsilon, b)$ where a and b are the boundaries of the support of $\mathcal{D}$ and ϵ is a fresh positive symbolic constant. This process

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

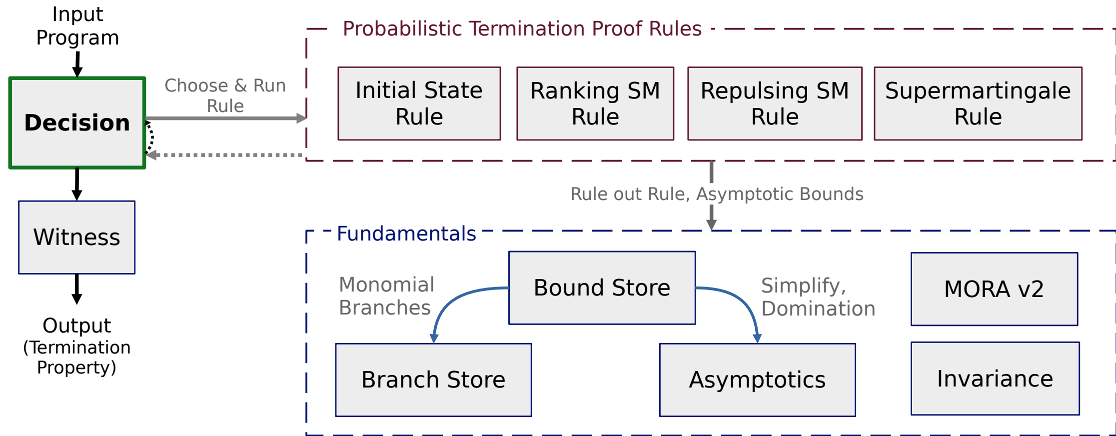


Figure 7.3: Main components of Amber and interactions between them

is repeated until all such variables s have been eliminated and the only distributions with mixed-sign supports left are over-approximations.

Example 71. *Consider the following program:*

```
x = 1
while x > 0:
    s = RV(normal, 0, 1)
    x = x+s
```

For $G = x$*, Amber computes the expression* $\mathbb{E}(x_{i+1} - x_i \mid Run_i) = 0$*. Therefore, condition 1 of the* SM-Rule *is satisfied. Regarding condition 2, Amber starts with the only branch of* x *which is* $x{+}s$*. The branch* $x{+}s$ *contains the variable* s *whose distribution has the mixed-sign support* $(-\infty, \infty)$*. Hence, Amber splits the branch* $x{+}s$ *into the three branches (1)* $x{+}s_1$*, (2)* $x{+}s_2$*, and (3)* $x{+}s_3$*, where* $s_1 \sim Symb(-\infty, -\epsilon)$*,* $s_2 \sim Symb(-\epsilon, \epsilon)$*,* $s_3 \sim Symb(\epsilon, \infty)$ *and* ϵ *is a fresh positive symbolic constant. For the new branch (1)* $x{+}s_1$*, an upper bound for* $x{+}s_1 - x = s_1$ *is given by* $-\epsilon$*. Therefore,* x *decreases by at least* ϵ *with some constant positive probability, confirming that also condition 2 of the* SM-Rule *is satisfied. Consequently, Amber certifies AST for this example.*

7.6 Amber: Implementation and Components

Implementation. Amber is implemented in `python3` and relies on the `lark-parser`[1] package to parse its input programs. Further, Amber uses the `diofant`[2] package as its computer-algebra system to (i) construct and manipulate mathematical expressions symbolically; (ii) solve algebraic recurrence relations, and (iii) compute function limits. To compute closed-form expressions for statistical moments of monomials over program

[1] `https://github.com/lark-parser/lark`
[2] `https://github.com/diofant/diofant`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

variables only depending on the loop counter, AMBER uses the tool `Mora` [BKS20c]. However, for efficient integration within AMBER, we reimplemented and adapted the `Mora` functionalities exploited by AMBER (`Mora v2`), in particular by deploying dynamic programming to avoid redundant computations. Altogether, AMBER consists of ~ 2000 lines of code. In what follows we discuss the main components of AMBER, as illustrated in Figure 7.3.

7.6.1 Decision in Amber

After parsing the input program, the *decision* module of AMBER is executed to initialize and call the probabilistic termination proof rules to be used on the input program. In order to initialize the proof rules, AMBER's *decision* module first constructs three expressions: (1) $\mathbb{E}(G_{i+1}-G_i \mid Run_i)$ (martingale expression); (2) $\mathbb{E}(G_i-G_{i+1} \mid Run_i)$ (negated martingale expression); and (3) $\mathbb{E}(G_{i+1}-G_i)$ (expected loop guard change). For Figure 7.2a with loop guard $x^2+y^2 < c$, we get the following expressions: (1) $\mathbb{E}(G_{i+1}-G_i \mid Run_i) = -x_i^2-11x_i-{}^{115}/_6$; (2) $\mathbb{E}(G_i-G_{i+1} \mid Run_i) = x_i^2+11x_i+{}^{115}/_6$; and (3) $\mathbb{E}(G_{i+1}-G_i) = -({}^{81}/_{16})i^2-({}^{1225}/_{48})i-{}^{121}/_6$. AMBER utilizes the relaxed proof rules from Section 7.4 and automates them using asymptotic bounds (cf. Section 7.5). As such, the *decision* module of AMBER initializes relaxed proof rules with the expressions above, applies the respective proof rules to the input program, and reports the analysis result containing potential witnesses for (non-)PAST or (non-)AST.

7.6.2 Probabilistic Termination Proof Rules in Amber

Initial State Rule. The *Initial State Rule* checks whether or not the initial state, given by the assignments preceding the loop, already falsifies the loop guard. More precisely, the rule returns a witness for PAST if the initial state falsifies the loop guard with probability one. The rule considers all possible combinations of lower and upper bounds of the initial assignments to the variables given by the support of the respective distributions.

Example 72. *In Figure 7.2a, the symbolic constant c in the loop guard represents an arbitrary positive constant. Therefore, for Figure 7.2a the probability of the initial state falsifying the loop guard is not* 1 *and the* Initial State Rule *does not return a witness for PAST.*

Ranking SM-Rule. The *Ranking SM-Rule* checks whether the polynomial G is eventually a ranking supermartingale (i.e. $\mathbb{E}(G_{i+1}-G_i \mid Run_i) \leq -\epsilon$) to conclude the input program to be PAST. If $\mathbb{E}(G_{i+1}-G_i) > 0$, G cannot be a (ranking) supermartingale. The rule determines its own applicability using `diofant` and the expected loop guard change $\mathbb{E}(G_{i+1}-G_i)$ to check $\mathbb{E}(G_{i+1}-G_i) > 0$. If the rule is applicable, the *Bound Store* module of AMBER is called to compute an asymptotic upper bound $u(i)$ for the martingale expression $\mathbb{E}(G_{i+1}-G_i \mid Run_i)$ (see Section 7.6.3). If $\lim_{i\to\infty} u(i) < 0$, then

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

G is eventually a ranking supermartingale and the input program is PAST. The *Ranking SM-Rule* uses `diofant` to verify $\lim_{i\to\infty} u(i) < 0$. If the condition holds, the *Ranking SM-Rule* constructs and returns a witness for PAST.

Example 73. *For Figure 7.2a, we have* $\mathbb{E}(G_{i+1}-G_i) = -(81/16)i^2 - (1225/48)i - 121/6 \not> 0$. *Thus the* Ranking SM-Rule *is applicable. For the martingale expression* $\mathbb{E}(G_{i+1}-G_i \mid Run_i) = -x_i^2 - 11x_i - 115/6$*, the* Bound Store *module computes an upper bounding function* $u(i) = -i^2$*. Because* $\lim_{i\to\infty} u(i) = -\infty < 0$*, the* Ranking SM-Rule *returns the martingale expression together with* $u(i)$ *as a witness for Figure 7.2a being PAST.*

SM-Rule. If the *Ranking SM-Rule* fails, the *SM-Rule* attempts to certify AST. The rule checks whether G is eventually a supermartingale (i.e. $\mathbb{E}(G_{i+1}-G_i \mid Run_i) \leq 0$) and whether G eventually decreases at least by some fixed constant with positive probability. The applicability criterion for the proof rule is the same as for the *Ranking SM-Rule* ($\mathbb{E}(G_{i+1}-G_i) \not> 0$), implemented in the same way. Moreover, Amber concludes G to be a supermartingale similarly to concluding it to be a ranking supermartingale. The only difference is that for the martingale expression's upper bound $u(i)$, its limit is allowed to be 0 (instead of negative). Amber automates the decrease condition by looping through all branches of G, splitting them as described in Section 7.5.3, and checking whether for one of the resulting branches B, the polynomial $B - G$ has an upper bounding function with a negative limit. This entails that eventually G decreases in any iteration with positive probability.

Example 74. *For Figure 7.2b, we have* $G = x$*. The martingale expression is* $\mathbb{E}(G_{i+1}-G_i \mid Run_i) = 0$*, which has limit* 0 *implying that* G *is a supermartingale.* Amber *retrieves the two branches of* G*, namely* $x+c$ *and* $x-c$*, where* c *is a positive symbolic constant. For the second branch* Amber *computes* $x-c-x = -c$ *which it determines to have the negative limit* $-c$*. Therefore,* Amber *concludes that* G *(eventually) decreases by (at least)* $-c$ *with positive probability and returns the martingale expression, the eventually decreasing branch and its asymptotic bound as a witness for AST.*

Repulsing SM-Rule. The *Repulsing SM-Rule* can potentially certify non-AST and non-PAST. It is applied in Amber whenever either the status of AST or PAST of the input program is not yet known after applying the *Ranking SM-Rule* and the *SM-Rule*. Moreover, $\mathbb{E}(G_{i+1}-G_i) \not< 0$ has to hold in order for the rule to be applicable because $-G$ needs to be a (ranking) supermartingale to certify non-PAST (non-AST). The applicability criterion as well as checking $-G$ to be a (ranking) supermartingale is realized with the same techniques as for the aforementioned proof rules. Additionally, Amber has to verify two more properties: (i) Eventually $|G_i-G_{i+1}| < c$ for some $c \in \mathbb{R}^+$; and (ii) in every iteration, there is a positive probability of G not decreasing. The first property (i) is realized with retrieving an absolute bounding function $a(i)$ from the *Bound Store* module and checking whether $a(i)$ is dominated by 1. Amber verifies the property (ii) by looping through all branches of G, splitting them as described in Section 7.5.3, and checking whether for one of the resulting branches B, the expression $G-B$ is always non-negative,

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

with a simple static analysis. This entails that there is always a positive probability that G does not decrease. AMBER returns a witness for non-PAST (non-AST) if all properties are satisfied and $-G$ is a (ranking) supermartingale.

Example 75. *For Figure 7.2c with $-G = -x$ we have the negative martingale expression $\mathbb{E}(G_i - G_{i+1} \mid Run_i) = -2c{\cdot}e$, where c and e are positive symbolic constants. Therefore, $-G$ is a ranking supermartingale. The two branches of $-x$ are (1) $-x-c$ and (2) $-x+c$. For both branches B we have $|B-(-x)| = c$ and a corresponding absolute bounding function $a(i) = c$. Hence, property (i) is satisfied. Property (ii) holds, because there is always a possibility of G not decreasing through branch (2). Thus, for Figure 7.2c as input, the* Repulsing SM-Rule *returns a witness for non-AST.*

7.6.3 Fundamentals in Amber

Bound Store. AMBER's *Bound Store* component derives lower, upper and absolute bounds for polynomials over program variables. These bounds are used by AMBER's termination proof rules (cf. Section 7.6.2). Asymptotic bounding functions only depend on the loop counter i and asymptotically bound the value of a program variable polynomials (modulo a positive constant factor). Asymptotic bounding functions for polynomials arise from combining bounding functions of its monomials. For monomials, asymptotic bounding functions are computed using the bound algorithm introduced in Section 7.5.

Other Fundamentals. The *Branch Store* module provides the functionality for extracting the branches of a given expression for the input program. The *Asymptotics* component of AMBER reasons about asymptotic properties of functions and simplifies expressions while preserving their asymptotic behavior. Multiple termination proof rules require the capability of checking whether some property over program variables is eventually invariant. This common requirement is implemented in AMBER's *Invariance* module.

7.7 Evaluation

Experimental Setup. AMBER and our benchmarks are publicly available at `https://github.com/probing-lab/amber`. The output of AMBER includes the martingale expression and an answer ("Yes", "No" or "Maybe") to PAST and AST for the input program. If the answer to (P)AST is definite ("Yes" or "No"), the output additionally contains a witness of the answer. We took all 39 benchmarks from [MBKK21a] and extended them by 11 new programs to test AMBER's capability to handle symbolic constants and drawing from probability distributions. The 11 new benchmarks are constructed from the 39 original programs, by adding noise drawn from common probability distributions and replacing concrete constants with symbolic ones. As such, we conduct experiments using a total of 50 challenging benchmarks. Further, we compare AMBER not only against `Absynth` and `MGen`, but also evaluate AMBER in comparison to the recent tools `LexRSM` [ACN18], `KoAT2` [MHG21] and `ecoimp` [AMS20]. Note that `MGen`

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 7.1: 27 programs which are PAST

Program	Amber	Absynth	MGen	LexRSM	KoAT2	ecoimp
2d_bounded_random_walk	✓	✗	NA	NA	✗	✗
biased_random_walk_const	✓	✓	✓	✓	✓	✓
biased_random_walk_exp	✓	✗	✓	✗	✗	✗
biased_random_walk_poly	✓	✗	✗	NA	✗	✗
binomial_past	✓	✓	✓	✓	✓	✓
complex_past	✓	✗	NA	NA	✗	✗
consecutive_bernoulli_trails	✓	✓	✓	✓	✓	✓
coupon_collector_4	✓	✗	✓	✓	✓	✓
coupon_collector_5	✓	✗	✓	✓	✓	✓
dueling_cowboys	✓	✓	✓	✓	✓	✓
exponential_past_1	✓	NA	NA	NA	✗	NA
exponential_past_2	✓	NA	NA	NA	✗	NA
geometric	✓	✓	✓	✓	✓	✓
geometric_exp	✗	✗	✗	✗	✗	✗
linear_past_1	✓	✗	✗	✗	✗	✗
linear_past_2	✓	✗	NA	✗	✗	✗
nested_loops	NA	✓	✗	✓	✓	✓
polynomial_past_1	✓	✗	NA	NA	✗	✗
polynomial_past_2	✓	✗	NA	NA	✗	✗
sequential_loops	NA	✓	✗	✓	✓	✓
tortoise_hare_race	✓	✓	✓	✓	✓	✓
dependent_dist*	NA	NA	NA	NA	✗	✓
exp_rw_gauss_noise*	✓	NA	NA	NA	NA	NA
gemoetric_gaussian*	✓	NA	NA	NA	NA	NA
race_uniform_noise*	✓	✗	✓	✓	✗	✓
symb_2d_rw*	✓	✗	NA	NA	✗	✗
uniform_rw_walk*	✓	✓	✓	✓	✓	✓
Total ✓	23	9	11	12	11	13

can only certify PAST and LexRSM only AST. Moreover, the tools Absynth, KoAT2 and ecoimp mainly aim to find upper bounds on expected costs. Tables 7.1-7.3 summarize our experimental results, with benchmarks separated into *PAST* (Table 7.1), *AST* (Table 7.2), and *not AST* (Table 7.3). Benchmarks marked with * are part of our 11 new examples. In every table, ✓ (✗) marks a tool (not) being able to certify the respective termination property. Moreover, **NA** symbolizes that a benchmark is out-of-scope for a tool, for instance, due to not supporting some distributions or polynomial arithmetic. All benchmarks have been run on a machine with a 2.6 GHz Intel i7 (Gen 10) processor and 32 GB of RAM and finished within a timeout of 50 seconds, where most experiments terminated within a few seconds.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 7.2: 14 programs which are AST and not necessarily PAST

Program	Amber	LexRSM
fair_in_limit_random_walk	NA	NA
gambling	✓	✗
symmetric_2d_random_walk	✗	NA
symmetric_random_walk_constant_1	✓	✗
symmetric_random_walk_constant_2	✓	✗
symmetric_random_walk_exp_1	✓	✗
symmetric_random_walk_exp_2	✓	NA
symmetric_random_walk_linear_1	✓	✗
symmetric_random_walk_linear_2	✓	✗
symmetric_random_walk_poly_1	✓	NA
symmetric_random_walk_poly_2	✓	NA
gaussian_rw_walk*	✓	NA
laplacian_noise*	✓	NA
symb_1d_rw*	✓	NA
Total ✓	12	0

Table 7.3: 9 programs which are not AST

Program	Amber
biased_random_walk_nast_1	✓
biased_random_walk_nast_2	✓
biased_random_walk_nast_3	✓
biased_random_walk_nast_4	✓
binomial_nast	✓
polynomial_nast	✗
binomial_nast_noise*	✓
symb_nast_1d_rw*	✓
hypergeo_nast*	✓
Total ✓	8

Experimental Analysis. Amber successfully certifies 23 out of the 27 PAST benchmarks (Table 7.1). Although Absynth, KoAT2 and ecosimp can find expected cost upper bounds for large programs [NCH18, MHG21, AMS20], they struggle on small programs whose termination is not known a priori. For instance, they struggle when a benchmark probabilistically "chooses" between two polynomials working against each other (one moving the program state away from a termination criterion and one towards it). Our experiments show that Amber handles such cases successfully. MGen supports the continuous uniform distribution and KoAT2 the geometric distribution whose support

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

is infinite. With these two exceptions, Amber is the only tool supporting continuous distributions and distributions with infinite support. To the best of our knowledge, Amber is the first tool certifying PAST supporting both discrete and continuous distributions as well as distributions with finite and infinite support. Amber successfully certifies 12 benchmarks to be AST which are potentially not PAST (Table 7.2). Whereas the `LexRSM` tool can certify non-PAST programs to be AST, such programs need to contain subprograms that are PAST [ACN18]. The well-known example of `symmetric_1D_random_walk`, contained in our benchmarks, does not have a PAST subprogram. Therefore, the `LexRSM` tool cannot establish AST for it. In contrast, Amber using the *SM-Rule* can handle such programs. To the best of our knowledge, Amber is the first tool capable of certifying non-AST for polynomial probabilistic programs involving drawing from distributions and symbolic constants. Amber is also the first tool automating (non-)AST and (non-)PAST analysis in a unifying manner for such programs.

Experimental Summary. Tables 7.1-7.3 demonstrate that (i) Amber outperforms the state-of-the-art in certifying (P)AST, and (ii) amber determines (non-)(P)AST for programs with various distributions and symbolic constants.

7.8 Conclusion

We described Amber, an open-source tool analyzing the termination behavior for polynomial probabilistic programs, in a fully automatic way. Amber computes asymptotic bounding functions and martingale expressions and is the first tool to prove and/or disprove (P)AST in a unifying manner. Amber can analyze continuous, discrete, finitely- and infinitely supported distributions in polynomial probabilistic programs parameterized by symbolic constants. Our experimental comparisons give practical evidence that Amber can prove and disprove (P)AST for a substantially larger class of programs than state-of-the-art tools.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 8

Summary & Outlook

In this thesis, we described various novel techniques for the automated analysis of probabilistic loops, relevant to safety as well as liveness of stochastic systems. Beyond algorithms and tools, this thesis contributes to and expands the foundational theory of invariant synthesis, relevant to both classical and probabilistic programs.

As a core contribution, we developed a fully automated method for computing exact closed-form expressions for higher moments of program variables for a large class of probabilistic loops. Our method models higher moments of loop variables using linear recurrences with constant coefficients. We established the theory of *moment-computable* loops and syntactically characterized a class of programs for which our approach is complete. Moment-computable loops allow for complex control flow, polynomial assignments, symbolic constants, as well as discrete and continuous probability distributions. The main restrictions on the supported loops are that variables within branching conditions must be finite, and non-linear dependencies must be acyclic.

For probabilistic systems with unknown model parameters, we introduced a novel approach to automatically compute the sensitivity of the system with respect to these unknown parameters. This method models parameters using symbolic constants and computes closed-form expressions for sensitivities of higher moments of program variables. By utilizing recurrences to model sensitivities directly – rather than moments – we demonstrated that sensitivity analysis is feasible for loops that are *not* moment-computable.

The new techniques for moment computation and sensitivity analysis have been implemented in the newly developed tool Polar. Our experimental evaluation showcases the applicability of these techniques, solving benchmarks that previously could not be automatically analyzed by the state-of-the-art.

The moment computation method introduced in this thesis imposes restrictions on the arithmetic permissible within loop bodies. Already for classical loops, the type of arithmetic used is known to be a major dimension of complexity. Although it is established

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

that the strongest polynomial invariant can be computed for linear loops, this problem has remained unresolved for polynomial loops. These loops are also called *unsolvable loops* and are equivalent to polynomial dynamical systems. We provided the first non-trivial lower bound for computing the strongest polynomial invariant for polynomial loops, by demonstrating that the problem is Skolem-hard. Additionally, as an intermediary result of independent interest, we showed that point-to-point reachability for polynomial dynamical systems is Skolem-hard as well. Furthermore, we generalized the notion of invariant ideals from classical programs to probabilistic programs, introducing the notion of *moment invariant ideals*. As a consequence, we were able to transfer various hardness results for classical program analysis to the probabilistic setting and justify that no restriction on the loops supported by Polar can be lifted without encountering significant hardness boundaries.

Despite the challenges presented by the hardness results, we introduced a novel approach for computing polynomial invariants of *bounded degree* for unsolvable loops. At the heart of our methodology is a new characterization of unsolvability in terms of *defective variables*. Leveraging the results of our invariant computation technique, we developed a synthesis procedure to over-approximate unsolvable loops with solvable loops.

Regarding termination analysis, we relaxed four proof rules from the literature that provide sufficient conditions for almost-sure termination, positive almost-sure termination and their negations. These modifications to the proof rules, alongside structural constraints on probabilistic loops, enabled us to algorithmically compute almost-sure asymptotic bounds on polynomials in program variables. Leveraging these bounds, we successfully automated all proof rules, leading to the new tool Amber, the first tool to certify both probabilistic termination and non-termination. We further extended the approach of asymptotic bounds for termination analysis to support symbolic constants and various common discrete and continuous probability distributions. Through experimental evaluation, we demonstrated that Amber can prove and refute probabilistic termination for a substantially larger class of programs than other state-of-the-art tools.

Building upon the foundations laid out in this thesis, several avenues for future research emerge. A particularily promising direction is the exploration of Polar's application in modeling control systems. Control systems with uncertainty in their sensor measurements or potential computational delays are inherently probabilistic and are frequently modeled using *Markov jump linear systems*. Preliminary investigations suggested a close connection between the loops analyzable by Polar and Markov jump linear systems, with findings indicating Polar's capability to certify mean-square stability of various closed-loop control systems. Advancing this line of inquiry could unveil new applications for Polar and enhance the field of probabilistic loop analysis.

Another compelling avenue involves adopting approximation techniques to extend the capabilities of our techniques and handle a wider array of probabilistic models. The hardness results established in this thesis make it difficult the expand the class of loops we can analyze using purely exact methods. However, these obstacles might be surmountable – without undermining the usefulness of the results – by integrating exact methods with

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

approximation techniques. For instance, in a publication not part of this thesis, we used approximation techniques to substitute trigonometric functions with polynomial expressions. As such, we enabled Polar to analyze various movement models subject to uncertainty. Despite utilizing these approximation methods, the resulting estimators were often unbiased, ensuring that the computed expected values remained exact. Similar hybrid approaches could offer a practical path forward, circumventing the inherent hardness barriers while retaining the utility and relevance of your analyses.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.

Bibliography

[ABH+21] Alejandro Aguirre, Gilles Barthe, Justin Hsu, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. A pre-expectation calculus for probabilistic sensitivity. *Proc. ACM Program. Lang.*, (POPL), 2021. doi:10.1145/3434333.

[ABK+22] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. Solving Invariant Generation for Unsolvable Loops. In *Proc. of SAS*, 2022. doi:10.1007/978-3-031-22308-2_3.

[ABK+24] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic. (Un)Solvable Loop Analysis. *Formal Methods Syst. Des.*, 2024. To appear.

[ACN18] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. *Proc. ACM Program. Lang.*, (POPL), 2018. doi:10.1145/3158122.

[ALY20] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. *Sci. Comput. Program.*, 2020. doi:10.1016/j.scico.2019.102338.

[AMS20] Martin Avanzini, Georg Moser, and Michael Schaper. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.*, (OOPSLA), 2020. doi:10.1145/3428240.

[ARS13] Nimar S. Arora, Stuart J. Russell, and Erik B. Sudderth. NET-VISA: Network Processing Vertically Integrated Seismic Analysis. *Seismol. Soc. Am., Bull.*, 2013. doi:10.1785/0120120107.

[BBR+15] John E. Bistline, David M. Blum, Chris Rinaldi, Gabriel Shields-Estrada, Siegfried S. Hecker, and Marie-Elisabeth Paté-Cornell. A Bayesian Model to Assess the Size of North Korea's Uranium Enrichment Program. *Sci. Global Secur.*, 2015. doi:10.1080/08929882.2015.1039431.

[BCJ+23] Kevin Batz, Mingshuai Chen, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Probabilistic program

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

verification via inductive synthesis of inductive invariants. In *Proc. of TACAS*, 2023. doi:10.1007/978-3-031-30820-8_25.

[BCK+21] Kevin Batz, Mingshuai Chen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. Latticed k-induction with an application to probabilistic programs. In *Proc. of CAV*, 2021. doi:10.1007/978-3-030-81688-9_25.

[BCKR20] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas W. Reps. Templates and recurrences: Better together. In *Proc. of PLDI*, 2020. doi:10.1145/3385412.3386035.

[BCW82] Hyman Bass, Edwin H. Connell, and David Wright. The Jacobian conjecture: reduction of degree and formal expansion of the inverse. *Bull. Amer. Math. Soc. (N.S.)*, 1982. doi:10.1090/S0273-0979-1982-15032-7.

[BEFH16] Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. Synthesizing probabilistic invariants via doob's decomposition. In *Proc. of CAV*, 2016. doi:10.1007/978-3-319-41528-4_3.

[BEG+18] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.*, (POPL), 2018. doi:10.1145/3158145.

[BFJ+21] Christel Baier, Florian Funke, Simon Jantsch, Toghrul Karimov, Engel Lefaucheux, Florian Luca, Joël Ouaknine, David Purser, Markus A. Whiteland, and James Worrell. The orbit problem for parametric linear dynamical systems. In *Proc. of CONCUR*, 2021. doi:10.4230/LIPIcs.CONCUR.2021.28.

[BFPS02] Nick F Britton, Nigel R Franks, Stephen C Pratt, and Thomas D Seeley. Deciding on a new home: how do honeybees agree? *Proc. R. Soc. Lond. Series B: Biological Sciences*, 2002. doi:10.1098/rspb.2002.2001.

[BG05] Olivier Bournez and Florent Garnier. Proving positive almost-sure termination. In *Proc. of RTA*, 2005. doi:10.1007/978-3-540-32033-3_24.

[BGB12] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *Proc. of MPC*, 2012. doi:10.1007/978-3-642-31113-0.

[BGJ93] M. Baake, U. Grimm, and D. Joseph. Trace maps, invariants, and some of their applications. *Internat. J. Modern Phys. B*, 1993. doi:10.1142/S021797929300247X.

[BGP+16] Olivier Bouissou, Eric Goubault, Sylvie Putot, Aleksandar Chakarov, and Sriram Sankaranarayanan. Uncertainty propagation using probabilistic

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

affine forms and concentration of measure inequalities. In *Proc. of TACAS*, 2016. doi:10.1007/978-3-662-49674-9_13.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN 978-0-262-02649-9.

[BKK+23] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. A calculus for amortized expected runtimes. *Proc. ACM Program. Lang.*, (POPL), 2023. doi:10.1145/3571260.

[BKOB12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. of POPL*, 2012. doi:10.1145/2103656.2103670.

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In *Proc. of ATVA*, 2019. doi:10.1007/978-3-030-31784-3_15.

[BKS20a] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020. doi:10.1017/9781108770750.

[BKS20b] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Analysis of Bayesian networks via prob-solvable loops. In *Proc. of ICTAC*, 2020. doi:10.1007/978-3-030-64276-1.

[BKS20c] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Mora - automatic generation of moment-based invariants. In *Proc. of TACAS*, 2020. doi:10.1007/978-3-030-45190-5_28.

[BLM13] Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration Inequalities - A Nonasymptotic Theory of Independence*. Oxford University Press, 2013. doi:10.1093/acprof:oso/9780199535255.001.0001.

[BLN+22] Yuri Bilu, Florian Luca, Joris Nieuwveld, Joël Ouaknine, David Purser, and James Worrell. Skolem meets schanuel. In *Proc. of MFCS*, 2022. doi:10.4230/LIPIcs.MFCS.2022.20.

[BMS05] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of Polynomial Programs. In *Proc. of VMCAI*, 2005. doi:10.1007/b105073.

[BTP+22] Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. Data-driven invariant learning for probabilistic programs. In *Proc. of CAV*, 2022. doi:10.1007/978-3-031-13185-1_3.

[Buc06] Bruno Buchberger. Bruno buchberger's phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *J. Symb. Comput.*, 2006. doi:10.1016/j.jsc.2005.09.007.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Cas72] J. W. S. Cassels. *An introduction to Diophantine approximation*. Cambridge Tracts in Mathematics and Mathematical Physics, No. 45. Hafner Publishing Co., New York, 1972. Facsimile reprint of the 1957 edition.

[CD02] Hei Chan and Adnan Darwiche. When do numbers really matter? *J. Artif. Intell. Res.*, 2002. doi:10.1613/jair.967.

[CD04] Hei Chan and Adnan Darwiche. Sensitivity Analysis in Bayesian Networks: From Single to Multiple Parameters. In *Proc. of UAI*, 2004.

[CFG16] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Proc. of CAV*, 2016. doi:10.1007/978-3-319-41528-4_1.

[CFGG20] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *Proc. of PLDI*, 2020. doi:10.1145/3385412.

[CFN20] Krishnendu Chatterjee, Hongfei Fu, and Petr Novotný. *Foundations of Probabilistic Programming*, chapter Termination Analysis of Probabilistic Programs with Martingales, page 221–258. Cambridge University Press, Cambridge, 2020. doi:10.1017/9781108770750.008.

[CFNH18] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. *ACM Trans. Program. Lang. Syst.*, 2018. doi:10.1145/3174800.

[CGMZ22] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic. Sound and complete certificates for quantitative termination analysis of probabilistic programs. In *Proc. of CAV*, 2022. doi:10.1007/978-3-031-13185-1_4.

[CH20] Jianhui Chen and Fei He. Proving almost-sure termination by omega-regular decomposition. In *Proc. of PLDI*, 2020. doi:10.1145/3385412.3386002.

[CHWZ15] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In *Proc. of CAV*, 2015. doi:10.1007/978-3-319-21690-4.

[CLO97] David A. Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms - an introduction to computational algebraic geometry and commutative algebra (2. ed.)*. Springer, 1997. doi:10.1137/1035171.

[CMP+20] Michaël Cadilhac, Filip Mazowiecki, Charles Paperman, Michal Pilipczuk, and Géraud Sénizergues. On polynomial recursive sequences. In *Proc. of ICALP*, 2020. doi:10.4230/LIPIcs.ICALP.2020.117.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

[CNZ17] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. Stochastic invariants for probabilistic termination. In *Proc. of POPL*, 2017. doi:10.1145/3009837.3009873.

[COW13] Ventsislav Chonev, Joël Ouaknine, and James Worrell. The orbit problem in higher dimensions. In *Proc. of STOC*, 2013. doi:10.1145/2488608.2488728.

[COW15] Ventsislav Chonev, Joël Ouaknine, and James Worrell. The polyhedron-hitting problem. In *Proc. of SODA*, 2015. doi:10.1137/1.9781611973730.64.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV*, 2006. doi:10.1007/11817963_37.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commun. ACM*, 2011. doi:10.1145/1941487.1941509.

[CRN+13] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In *Proc. of ESEC/FSE*, 2013. doi:10.1145/2491411.2491423.

[CS13] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *Proc. of CAV*, 2013. doi:10.1007/978-3-642-39799-8_34.

[CS14] Aleksandar Chakarov and Sriram Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In *Proc. of SAS*, 2014. doi:10.1007/978-3-319-10936-7_6.

[CVS16] Aleksandar Chakarov, Yuen-Lam Voronin, and Sriram Sankaranarayanan. Deductive proofs of almost sure persistence and recurrence properties. In *Proc. of TACAS*, 2016. doi:10.1007/978-3-662-49674-9_15.

[CYS20] Yi Chou, Hansol Yoon, and Sriram Sankaranarayanan. Predictive runtime monitoring of vehicle models using bayesian estimation and reachability analysis. In *Proc. of IROS*, 2020. doi:10.1109/IROS45743.2020.9340755.

[dB06] Michiel de Bondt. Quasi-translations and counterexamples to the homogeneous dependence problem. *Proc. Amer. Math. Soc.*, 2006. doi:10.1090/S0002-9939-06-08335-3.

[DDP16] Tommaso Dreossi, Thao Dang, and Carla Piazza. Parallelotope bundles for polynomial reachability. In *Proc. of HSCC*, 2016. doi:10.1145/2883817.2883838.

[DDP17] Tommaso Dreossi, Thao Dang, and Carla Piazza. Reachability computation for polynomial dynamical systems. *Formal Methods Syst. Des.*, 2017. doi:10.1007/s10703-016-0266-3.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[DFS98] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP*, 1998. doi:10.1007/BFb0055044.

[DG19] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Trans. Program. Lang. Syst.*, 2019. doi:10.1145/3293605.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *Proc. of CAV*, 2017. doi:10.1007/978-3-319-63390-9.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, 2008. doi:10.1007/978-3-540-78800-3.

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Proc. of ATVA*, 2016. doi:10.1007/978-3-319-46520-3_30.

[dSFGZ08] Ricardo dos Santos Freire, Jr., Gianluca Gorni, and Gaetano Zampieri. Search for homogeneous polynomial invariants and a cubic-homogeneous mapping without quadratic invariants. *Univ. Iagel. Acta Math.*, 2008.

[DT12] Thao Dang and Romain Testylier. Reachability analysis for polynomial dynamical systems using the bernstein expansion. *Reliab. Comput.*, 2012.

[Dur19] Rick Durrett. *Probability: Theory and Examples*. Cambridge University Press, 2019. doi:10.1017/9781108591034.

[EGK12] Javier Esparza, Andreas Gaiser, and Stefan Kiefer. Proving Termination of Probabilistic Programs Using Patterns. In *Proc. of CAV*, 2012. doi:10.1007/978-3-642-31424-7_14.

[EGLW72] B. Elspas, M. Green, K. Levitt, and R. Waldinger. Research in Interactive Program-Proving Techniques. Technical report, SRI, 1972.

[EvdPSW03] Graham Everest, Alfred J. van der Poorten, Igor E. Shparlinski, and Thomas Ward. *Recurrence Sequences*. Math. Surveys Monogr. Amer. Math. Soc., Providence, 2003. ISBN 978-0-8218-3387-2.

[FDG+19] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: a language for scenario specification and scene generation. In *Proc. of PLDI*, 2019. doi:10.1145/3314221.3314633.

[FFH15] Luis Luis María Ferrer Fioriti and Holger Hermanns. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proc. of POPL*, 2015. doi:10.1145/2676726.2677001.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[FGH13] Alain Finkel, Stefan Göller, and Christoph Haase. Reachability in register machines with polynomial updates. In *Proc. of MFCS*, 2013. doi:10.1007/978-3-642-40313-2_37.

[FHG20] Florian Frohn, Marcel Hark, and Jürgen Giesl. Termination of polynomial loops. In *Proc. of SAS*, 2020. doi:10.1007/978-3-030-65474-0_5.

[FK15] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *Proc. of FMCAD*, 2015. doi:10.1109/FMCAD.2015.7542253.

[FZJ+17] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In *Proc. of ATVA*, 2017. doi:10.1007/978-3-319-68167-2_26.

[GAB+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reasoning*, 2017. doi:10.1007/s10817-016-9388-y.

[GGH19] Jürgen Giesl, Peter Giesl, and Marcel Hark. Computing expected runtimes for constant probability programs. In *Proc. of CADE*, 2019. doi:10.1007/978-3-030-29436-6_16.

[Gha15] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 2015. doi:10.1038/nature14541.

[GKM13] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In *Proc. of QEST*, 2013. doi:10.1007/978-3-642-40196-1_17.

[GMV16] Timon Gehr, Sasa Misailovic, and Martin T. Vechev. PSI: Exact symbolic inference for probabilistic programs. In *Proc. of CAV*, 2016. doi:10.1007/978-3-319-41528-4_4.

[Gru96] Dominik Gruntz. *On Computing Limits in a Symbolic Manipulation System.* PhD thesis, ETH Zürich, 1996. doi:10.3929/ETHZ-A-001631582.

[Has70] Wilfred K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 1970. doi:10.2307/2334940.

[HCD+18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Proc. of TACAS*, 2018. doi:10.1007/978-3-319-89963-3_30.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[HdBM20] Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. Scaling exact inference for discrete probabilistic programs. In *Proc. of OOPSLA*, 2020. doi:10.1145/3428208.

[Her90] Ted Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 1990. doi:10.1016/0020-0190(90)90107-9.

[HFC18] Mingzhang Huang, Hongfei Fu, and Krishnendu Chatterjee. New Approaches for Almost-Sure Termination of Probabilistic Programs. In *Proc. of APLAS*, 2018. doi:10.1007/978-3-030-02768-1_11.

[HFCG19] Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.*, 2019. doi:10.1145/3360555.

[HFG20] Marcel Hark, Florian Frohn, and Jürgen Giesl. Polynomial loops: Beyond termination. In *Proc. of LPAR*, 2020. doi:10.29007/nxv1.

[HFM+14] Zhenqi Huang, Chuchu Fan, Alexandru Mereacre, Sayan Mitra, and Marta Z. Kwiatkowska. Invariant verification of nonlinear hybrid automata networks of cardiac cells. In *Proc. of CAV*, 2014. doi:10.1007/978-3-319-08867-9_25.

[HJK17] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In *Proc. of ISSAC*, 2017. doi:10.1145/3087604.3087623.

[HJK18a] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Aligator.jl - A julia package for loop invariant generation. In *Proc. of CICM*, 2018. doi:10.1007/978-3-319-96812-4_10.

[HJK18b] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. Invariant generation for multi-path loops with polynomial assignments. In *Proc. of VMCAI*, 2018. doi:10.1007/978-3-319-73721-8_11.

[HJVC+21] Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd D. Millstein, Sanjit A. Seshia, and Guy Van den Broeck. Model checking finite-horizon markov chains with probabilistic inference. In *Proc. of CAV*, 2021. doi:10.1007/978-3-030-81688-9_27.

[HKGK20] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. Aiming low is harder: induction for lower bounds in probabilistic program verification. *Proc. ACM Program. Lang.*, (POPL), 2020. doi:10.1145/3371105.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[HOPW18] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. Polynomial Invariants for Affine Programs. In *Proc. of LICS*, 2018. doi:10.1145/3209108.3209142.

[HOPW23] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. On strongest algebraic program invariants. *J. ACM*, 2023. doi:10.1145/3614319.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, 1969.

[HWM18] Zixin Huang, Zhenbang Wang, and Sasa Misailovic. PSense: Automatic Sensitivity Analysis for Probabilistic Programs. In *Proc. of ATVA*, 2018. doi:10.1007/978-3-030-01090-4_23.

[Jag80] A. V. Jagžev. On a problem of O.-H. Keller. *Sibirsk. Mat. Zh.*, 1980.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inform.*, 1976. doi:10.1007/BF00268497.

[Kar94] Richard M. Karp. Probabilistic recurrence relations. *J. ACM*, 1994. doi:10.1145/195613.195632.

[Kau05] Manuel Kauers. *Algorithms for Nonlinear Higher Order Difference Equations*. PhD thesis, RISC, Johannes Kepler University, Linz, 2005.

[KBCR19] Zachary Kincaid, Jason Breck, John Cyphert, and Thomas W. Reps. Closed forms for numerical loops. *Proc. ACM Program. Lang.*, (POPL), 2019. doi:10.1145/3290368.

[KCBR18] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, (POPL), 2018. doi:10.1145/3158142.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

[KK15] Benjamin Lucien Kaminski and Joost-Pieter Katoen. On the hardness of almost-sure termination. In *Proc. of MFCS*, 2015. doi:10.1007/978-3-662-48057-1_24.

[KKM19] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. On the hardness of analyzing probabilistic programs. *Acta Inform.*, 2019. doi:10.1007/s00236-018-0321-1.

[KKMO16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *Proc. of ESOP*, 2016. doi:10.1007/978-3-662-49498-1.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KKMO18] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of randomized algorithms. *J. ACM*, 2018. doi:10.1145/3208102.

[KKZ23] Zachary Kincaid, Nicolas Koh, and Shaowei Zhu. When less is more: Consequence-finding in a weak theory of arithmetic. *Proc. ACM Program. Lang.*, (POPL), 2023. doi:10.1145/3571237.

[KL80] Ravindran Kannan and Richard J. Lipton. The orbit problem is decidable. In *Proc. of STOC*, 1980. doi:10.1145/800141.804673.

[KLO+22] Toghrul Karimov, Engel Lefaucheux, Joël Ouaknine, David Purser, Anton Varonka, Markus A. Whiteland, and James Worrell. What's decidable about linear loops? *Proc. ACM Program. Lang.*, (POPL), 2022. doi:10.1145/3498727.

[KM76] Shmuel Katz and Zohar Manna. Logical analysis of programs. *Commun. ACM*, 1976. doi:10.1145/360032.360048.

[KMMM10] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *Proc. of SAS*, 2010. doi:10.1007/978-3-642-15769-1_24.

[KMS+22a] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *Proc. of QEST*, 2022. doi:10.1007/978-3-031-16336-4_2.

[KMS+22b] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura. Moment-based invariants for probabilistic loops with non-polynomial assignments. In *Proc. of QEST*, 2022. doi:10.1007/978-3-031-16336-4_1.

[KMS+24] Andrey Kofnov, Marcel Moosbrugger, Miroslav Stankovič, Ezio Bartocci, and Efstathia Bura. Exact and approximate moment derivation for probabilistic loops with non-polynomial assignments. *ACM Trans. Model. Comput. Simul.*, 2024. doi:10.1145/3641545. Just Accepted.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV*, 2011. doi:10.1007/978-3-642-22110-1.

[KNP12] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Form. Asp. Comput.*, 2012. doi:10.1007/s00165-012-0227-6.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KNP18] Sang-Ki Ko, Reino Niskanen, and Igor Potapov. Reachability problems in nondeterministic polynomial maps on the integers. In *Proc. of DLT*, 2018. doi:10.1007/978-3-319-98654-8_38.

[Kol06] John E. Kolassa. *Series Approximation Methods in Statistics*. Springer, 2006. doi:10.1007/0-387-32227-2.

[Kov08] Laura Kovács. Reasoning algebraically about p-solvable loops. In *Proc. of TACAS*, 2008. doi:10.1007/978-3-540-78800-3_18.

[Koz81] Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 1981. doi:10.1016/0022-0000(81)90036-2.

[Koz83] Dexter Kozen. A probabilistic PDL. In *Proc. of STOC*, 1983. doi:10.1145/800061.808758.

[Koz85] Dexter Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 1985. doi:10.1016/0022-0000(85)90012-1.

[KP11] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. Springer, 2011. doi:10.1007/978-3-7091-0445-3.

[KSK76] John G. Kemeny, J. Laurie Snell, and Anthony W. Knapp. *Denumerable Markov Chains: with a chapter of Markov Random Fields by David Griffeath*. Springer, 2 edition, 1976.

[KUH19] Satoshi Kura, Natsuki Urabe, and Ichiro Hasuo. Tail probabilities for randomized program runtimes via martingales for higher moments. In *Proc. of TACAS*, 2019. doi:10.1007/978-3-030-17465-1_8.

[KV23] Laura Kovács and Anton Varonka. What else is undecidable about loops? In *Proc. of RAMiCS*, 2023. doi:10.1007/978-3-031-28083-2_11.

[KZ08] Manuel Kauers and Burkhard Zimmermann. Computing the algebraic relations of c-finite sequences and multisequences. *J. Symb. Comput.*, 2008. doi:10.1016/J.JSC.2008.03.002.

[KZH+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.*, 2011. doi:10.1016/j.peva.2010.04.001.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of CGO*, 2004. doi:10.1109/CGO.2004.1281665.

[LLMR17] Ondrej Lengál, Anthony Widjaja Lin, Rupak Majumdar, and Philipp Rümmer. Fair termination for parameterized probabilistic concurrent systems. In *Proc. of TACAS*, 2017. doi:10.1007/978-3-662-54577-5_29.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[LLN+22] Richard Lipton, Florian Luca, Joris Nieuwveld, Joël Ouaknine, David Purser, and James Worrell. On the skolem problem and the skolem conjecture. In *Proc. of LICS*, 2022. doi:10.1145/3531130.3533328.

[Mar20] Milton F. Maritz. A note on exact solutions of the logistic map. *Chaos*, 2020. doi:10.1063/1.5125097.

[May76] Robert M. May. Simple mathematical models with very complicated dynamics. *Nature*, 1976. doi:10.1038/261459a0.

[MBKK21a] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. Automated Termination Analysis of Polynomial Probabilistic Programs. In *Proc. of ESOP*, 2021. doi:10.1007/978-3-030-72019-3_18.

[MBKK21b] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. In *Proc. of FM*, 2021. doi:10.1007/978-3-030-90870-6_36.

[MBKK22] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. The Probabilistic Termination Tool Amber. *Formal Methods Syst. Des.*, 2022. doi:10.1007/S10703-023-00424-Z.

[MHG21] Fabian Meyer, Marcel Hark, and Jürgen Giesl. Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes. In *Proc. of TACAS*, 2021. doi:10.1007/978-3-030-72016-2_14.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005. doi:10.1007/b138392.

[MMK23] Marcel Moosbrugger, Julian Müllner, and Laura Kovács. Automated Sensitivity Analysis for Probabilistic Loops. In *Proc. of iFM*, 2023. doi:10.1007/978-3-031-47705-8_2.

[MMK24] Julian Müllner, Marcel Moosbrugger, and Laura Kovács. Strong Invariants Are Hard: On the Hardness of Strongest Polynomial Invariants for (Probabilistic) Programs. *Proc. ACM Program. Lang.*, (POPL), 2024. doi:10.1145/3632872.

[MMKK18] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A New Proof Rule for Almost-sure Termination. *Proc. ACM Program. Lang.*, 2018. doi:10.1145/3158121.

[Mon01] David Monniaux. An abstract analysis of the probabilistic termination of programs. In *Proc. of SAS*, 2001. doi:10.1007/3-540-47764-0.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/cbo9780511814075.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[MS04a] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 2004. doi:10.1016/j.ipl.2004.05.004.

[MS04b] Markus Müller-Olm and Helmut Seidl. A note on karr's algorithm. In *Proc. of ICALP*, 2004. doi:10.1007/978-3-540-27836-8_85.

[MSBK22a] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This Is The Moment for Probabilistic Loops. *Proc. ACM Program. Lang.*, (OOPSLA2), 2022. doi:10.1145/3563341.

[MSBK22b] Marcel Moosbrugger, Miroslav Stankovič, Ezio Bartocci, and Laura Kovács. This is the Moment for Probabilistic Loops - Artifact (Polar), 2022. doi:10.5281/zenodo.7055030.

[MSP+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondrej Certík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepán Roucka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony M. Scopatz. Sympy: symbolic computing in python. *PeerJ Comput. Sci.*, 2017. doi:10.7717/PEERJ-CS.103.

[Mül23] Julian Müllner. Exact inference for probabilistic loops. Master's thesis, Technische Universität Wien, 2023.

[Nag72] Masayoshi Nagata. *On automorphism group of* $k[x, y]$. Kinokuniya Book Store Co., Ltd., Tokyo, 1972. Department of Mathematics, Kyoto University, Lectures in Mathematics, No. 5.

[NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *Proc. of PLDI*, 2018. doi:10.1145/3192366.3192394.

[NCR+16] Praveena Narayanan, Jacques Carette, Wren Romano, Chung chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *Proc. of FLOPS*, 2016. doi:10.1007/978-3-319-29604-3_5.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bull. Am. Math. Soc.*, 1946.

[RB94] John A. G. Roberts and Michael Baake. Trace maps as 3D reversible dynamical systems with an invariant. *J. Statist. Phys.*, 1994. doi:10.1007/BF02188581.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[RcK04] Enric Rodríguez-carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proc. of ISSAC*, 2004. doi:10.1145/1005285.1005324.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.*, 1953. doi:10.1090/s0002-9947-1953-0053041-6.

[RK07] Enric Rodríguez-Carbonell and Deepak Kapur. Generating All Polynomial Invariants in Simple Loops. *J. Symb. Comput.*, 2007. doi:10.1016/j.jsc.2007.01.002.

[Sah78] N. Saheb-Djahromi. Probabilistic LCF. In *Proc. of MFCS*, 1978. doi:10.1007/3-540-08921-7_92.

[SBK22] Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. Moment-based analysis of bayesian network properties. *Theor. Comput. Sci.*, 2022. doi:10.1016/j.tcs.2021.12.021.

[SCGP20] Sriram Sankaranarayanan, Yi Chou, Eric Goubault, and Sylvie Putot. Reasoning about uncertainties in discrete-time dynamical systems using polynomial forms. In *Proc. of NeurIPS*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/ca886eb9edb61a42256192745c72cd79-Abstract.html`.

[Sch93] Marco Schneider. Self-stabilization. *ACM Comput. Surv.*, 1993. doi:10.1145/151254.151256.

[SO19] Anne Schreuder and C.-H. Luke Ong. Polynomial probabilistic invariants and the optional stopping theorem. *CoRR*, abs/1910.12634, 2019.

[SRB+15] Konstantin Selyunin, Denise Ratasich, Ezio Bartocci, Md. Ariful Islam, Scott A. Smolka, and Radu Grosu. Neural programming: Towards adaptive control in cyber-physical systems. In *Proc. of CDC*, 2015. doi:10.1109/CDC.2015.7403319.

[SRM21] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. Sppl: Probabilistic programming with fast exact symbolic inference. In *Proc. of PLDI*, 2021. doi:10.1145/3453483.3454078.

[Tao08] Terrence Tao. *Structure and Randomness*. American Mathematical Society, 2008. ISBN 0-8218-4695-7.

[TOUH18] Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. Ranking and repulsing supermartingales for reachability in probabilistic programs. In *Proc. of ATVA*, 2018. doi:10.1007/978-3-030-01090-4_28.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 1937. doi:10.1112/PLMS/S2-42.1.230.

[vP03] Arno van den Essen and Ronen Peretz. Polynomial automorphisms and invariants. *Journal of Algebra*, 2003. doi:10.1016/S0021-8693(03)00424-1.

[VVB22] Elizaveta Vasilenko, Niki Vazou, and Gilles Barthe. Safe couplings: Coupled refinement types. In *Proc. of ICFP*, 2022. doi:10.1145/3547643.

[War65] Stanley L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *J. Am. Stat. Assoc.*, 1965. doi:10.1080/01621459.1965.10480775.

[WFC+20] Peixin Wang, Hongfei Fu, Krishnendu Chatterjee, Yuxin Deng, and Ming Xu. Proving expected sensitivity of probabilistic programs with randomized variable-dependent termination time. *Proc. ACM Program. Lang.*, (POPL), 2020. doi:10.1145/3371093.

[WHR21] Di Wang, Jan Hoffmann, and Thomas Reps. Central moment analysis for cost accumulators in probabilistic programs. In *Proc. of PLDI*, 2021. doi:10.1145/3453483.3454062.

[YKS14] Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. Nagoya termination tool. In *Proc. of RTA-TLCA*, 2014. doi:10.1007/978-3-319-08918-8_32.

[YS06] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 2006. doi:10.1016/j.ic.2006.05.002.

[Zam08] Gaetano Zampieri. Homogeneous polynomial invariants for cubic-homogeneous functions. *Univ. Iagel. Acta Math.*, 2008.

Die approbierte gedruckte Originalversion dieser Dissertation ist an der TU Wien Bibliothek verfügbar.
The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub