Informatics

Exakte Inferenz für Probabilistische Programme

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Julian Müllner, BSc
Matrikelnummer 11809619

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in techn. Laura Kovács, MSc
Mitwirkung: Univ.Ass. Dipl.-Ing. Marcel Moosbrugger, BSc

Wien, 4. Mai 2023

Julian Müllner

Laura Kovács

TU WIEN Informatics

Exact Inference for Probabilistic Loops

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Julian Müllner, BSc
Registration Number 11809619

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in techn. Laura Kovács, MSc
Assistance: Univ.Ass. Dipl.-Ing. Marcel Moosbrugger, BSc

Vienna, 4th May, 2023

Julian Müllner

Laura Kovács

Erklärung zur Verfassung der Arbeit

Julian Müllner, 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, 4. Mai 2023

Julian Müllner

Acknowledgements

Even though this thesis has only one author, many have part in it.

My deepest gratitude goes to my supervisors Laura Kovács and Marcel Moosbrugger, who supported my far beyond that what can be hoped for, even of excellent supervisors. Laura's feedback and extensive experience has considerably improved the quality of this thesis, while Marcel directly had part in its creation by devoting much of his time to fruitful discussions and constant ideas for improvement. Their advice and mentoring has been invaluable to me, both professionally and as a person.

But apart from the people that directly influenced this thesis, I would like to thank the people who initially got me excited about computer engineering and computer science. These are Andreas Steininger and Ulrich Schmid, both of whom had a fair share in me developing the skills necessary to conduct this research, through their exceptional teaching and engagement in the Bachelor with Honors project.

Additionally, the ERC Consolidator Grant 2020 ARTIST 101002685 and the ProbInG grant of the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018] financially supported the research that is partially presented in this thesis.

Apart from academic life, the years of my studies have been enriched by a lot of interesting people, first and foremost my girlfriend Selina, but also my siblings, friends, and colleagues. Nevertheless, all the outstanding efforts of the people mentioned previously would have been in vain, had it not been for my parents and their partners. Thank you!

Kurzfassung

Die probabilistische Programmierung erleichtert die Entwicklung und Analyse von statistischen Modellen erheblich. Lediglich mithilfe einer Beschreibung des statistischen Prozesses können algorithmische Ansätze oft vollautomatisch tiefgehende Einblicke in den modellierten Vorgang geben und ermöglichen es somit dem Benutzer Inferenzen durchzuführen, d. h. Schlussfolgerungen aus beobachteten Daten zu ziehen. Allerdings ist exakte Programmanalyse in der Gegenwart von Schleifen schon bei herkömmlichen deterministischen Programmen eine notorisch schwierige Aufgabe. Um dennoch rigorose Analyseergebnisse zu gewährleisten, ist es notwendig, maßgeschneiderte Ansätze für strukturell eingeschränkte Programme und Schleifen zu entwickeln. In dieser Arbeit entwickeln wir mehrere Techniken, für die automatischen Analyse von probabilistischen Schleifen.

Um Inferenz in probabilistischen Programmen durchzuführen, ist es notwendig, Informationen über die Wahrscheinlichkeitsverteilung, die durch das Programm spezifiziert wird, zu extrahieren. Um die modellierte Verteilung zu berechnen, stellen wir eine korrekte und vollständige Methode für Programme und Schleifen über endlichen Zustandsräumen vor und ermöglichen eine vollständige Automatisierung in unserem neuen Tool Blizzard. Für probabilistische Programme über unendlichen Zustandsräumen stellen wir eine Methode vor, mit der eine beschränkte Klasse von Programmen durch äquivalente, schleifenfreie Programme ersetzt werden kann. Darüber hinaus stellen wir maßgeschneiderte Techniken vor, die auf erzeugenden Funktionen basieren, und zeigen, dass Typsysteme geeignet sind, um die Verteilungen von Variablen in probabilistischen Schleifen zu identifizieren.

Dennoch sind einige Probleme bei der Analyse probabilistischer Programme nachweislich nicht entscheidbar. In dieser Arbeit untersuchen wir die Grenze der Entscheidbarkeit, indem wir das Problem der Invarianten-Synthese für probabilistische Programme untersuchen. Wir stellen bestehende, nicht-probabilistische Ergebnisse vor und geben korrespondierende Beweise für probabilistische Programme. Unter anderem präsentieren wir eine bisher unbekannte Beziehung zwischen dem Problem der Invarianten-Synthese und dem Skolem-Problem, einem schwierigen ungelösten Problem der Zahlentheorie.

Mit den vorgestellten Techniken bringen wir den Stand der Technik in der probabilistischen Schleifenanalyse voran und ermöglichen exakte Analyse und Inferenz für eine neue Klasse von probabilistischen Programmen und Schleifen.

Abstract

Probabilistic programming greatly facilitates the development and analysis of statistical models. By providing merely a description of the process, algorithmic approaches can often fully automatically derive deep insights into the modelled process and allow the user to perform inference, that is, to draw conclusions from observed data. However, performing exact program analysis in the presence of loops is a notoriously hard task, already for traditional deterministic programs. To still provide rigorous analysis results, it is necessary to provide tailored approaches for structurally restricted programs and loops. In this thesis, we develop several techniques that tackle the problem of automatically inferring properties of probabilistic loops.

To perform inference on probabilistic programs, it is necessary to obtain information about the probability distribution that is specified by the program. To extract the modelled distribution, we present a sound and complete method for programs and loops over a finite state-space and provide full automation in our new tool Blizzard. For infinite-state probabilistic programs, we present an approach that is able to replace a restricted class of programs with equivalent, loop-free programs. Moreover, we present tailored techniques based on generating functions and show that type systems are well-suited to identify the distributions of variables in probabilistic loops.

Nevertheless, some problems in the analysis of probabilistic programs provably cannot be answered. In this thesis, we investigate the boundary of decidability by studying the problem of invariant synthesis for probabilistic programs. We present existing, non-probabilistic results and give corresponding proofs for probabilistic programs. Among others, we present a hitherto unknown relation between the problem of invariant synthesis and the Skolem problem, a difficult open problem in number theory.

With the presented techniques, we advance the state-of-the-art in probabilistic loop analysis and enable exact analysis and inference for new classes of probabilistic programs and loops.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Due to the probabilistic and uncertain nature of numerous real-world processes such as financial markets, robotic environments and randomized algorithms, it is often necessary to reason in the presence of uncertainty and infer properties of the system from observed data. To achieve this, statistical modeling and machine learning are two widely applied tools. A statistical model can be used twofold: (a) to describe the data generation process and evaluate the probability of possible outcomes, and (b) to perform (Bayesian) inference on the model using observed output data and automatically learn/adapt the model parameters.

Traditionally, such models have been created manually and evaluated by tailored mathematical approaches based on e.g., statistics, logic and algebra, which are tedious and require significant mathematical expertise. To overcome these limitations, *Probabilistic Programming Languages* allow the user to specify the probabilistic model in a designated programming language and provide means to run fully-automated inference procedures, significantly facilitating the deployment of such models [GHNR14, vdMPYW18]. Moreover, probabilistic programming can be viewed as an attempt to provide machine learning procedures with domain knowledge by giving insight to how data is generated.

Example 1. To illustrate a first use-case (a), consider the biased, one-dimensional random walk shown in Figure 1.1a. Estimating the distribution modelled by the program is non-trivial, especially since the step width is not symmetrical and the direction is chosen with some bias. A possible query to an inference system would be “How is the probability distributed over possible values of x?”.

A machine-learning view of probabilistic programming (b) can be illustrated by the problem of estimating the average life expectancy in a country based on a number of

```
$x \leftarrow 0$
for $i$ in $1 \dots 10$ do
    $direction \leftarrow$ Bernoulli$(0.35)$
    if $direction = 0$ then
        $x \leftarrow x +$ Uniform$(1; 5)$
    else
        $x \leftarrow x -$ Uniform$(2; 7)$
    end if
end for
```

(a) A biased random walk

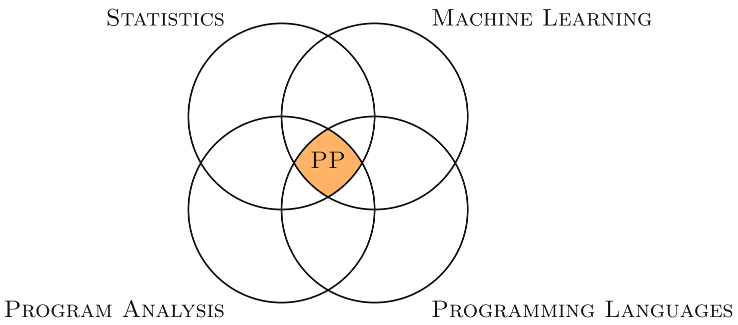
```
▷ lt: Expected Lifetime
▷ pSmokers: smoking prevalence
▷ pDisease: prevalence of severe diseases
lt $\leftarrow$ Normal$(M; V)$
if developed_country then
    lt $\leftarrow$ lt $+ D$
end if
lt $\leftarrow$ lt $- pSmokers \cdot S$
lt $\leftarrow$ lt $- pDisease \cdot I$
```

(b) A life-expectancy model

Figure 1.1: Two probabilistic programs

factors and some given training data[1]. One could simply train some standard model such as a neural network, but this hardly allows the user to include domain knowledge. In contrast, in Figure 1.1b, we show how one may model the problem in a probabilistic programming language. By presenting evidence, the inference engine can perform Bayesian inference on the parameters M, V, D, S and I and update the belief on what the 'real' values are. After performing inference, the task of average life expectancy estimation (even in the presence of incomplete data) can be reduced to evaluating the program on some input data, e.g., "What is the average life expectancy in a developed country where 20 percent of the population are smokers?".

Note that the preceding programs are not necessarily executed to answer these questions, **a probabilistic program is a merely a way to formulate the model.** In this manner, probabilistic programming ties together several research areas, since it is used to build statistical models in a programming language, uses program analysis techniques to infer the distribution and can be used to perform machine learning. As such, probabilistic programming (PP) lies at the intersection of statistics, machine learning, program analysis and programming language theory, as illustrated below.



[1] https://www.kaggle.com/datasets/kumarajarshi/life-expectancy-who

1.2 Problem Statement

It is evident that full-featured probabilistic programming languages are strictly more powerful than traditional programming languages, and thus it does not come as a surprise that certain properties are undecidable (such as termination [KKM19]) or very hard to answer (such as inference [HVM20, DL93, CDM14]). Therefore, most probabilistic programming languages have been carefully designed to syntactically only allow programs that actually can be evaluated within the chosen model (e.g., to only allow discrete variables, restriction to linear arithmetic or restrictions on distributions). Unsurprisingly, non-statically bounded loops are notoriously hard, as these cannot be unrolled to create a loop-free program and are thus absent in virtually all probabilistic programming languages.

To perform (Bayesian) inference on probabilistic programs, it is invaluable to have complete knowledge of the distribution described by the probabilistic model, as this allows to us to directly perform inference. *In this thesis, we will explore methods and programming models that allow an exact analysis of distributions modelled by probabilistic programs with a special focus on probabilistic loops.*

1.3 Contributions

The results presented in this thesis are manifold and range from advanced contributions, such as the hardness proofs in Chapter 6, to concrete, immediately applicable research outputs, such as the program analysis tool Blizzard presented in Chapter 3.

Before listing our contributions to the state-of-the-art, we stress again the hardness of probabilistic program analysis, especially in the presence of loops. It is well-known, that even reasoning about traditional programs in the presence of loops is an undecidable problem, and probabilistic programs are a strict generalization [KKM19, HOPW19]. Therefore, it is apparent that the problem is undecidable in full generality.

This is why virtually all existing exact methods are restricted to statically bounded loops, i.e., loops that can be unrolled to obtain a loop-free program. This is impossible in general, as some loops may be executed until a stopping criterion is met or the number of loop iterations may be unbounded. To address those problems, the main contributions of this thesis are summarized as follows:

1. For finite-state programs, and especially finite-state loops, we present a translation from probabilistic programs to Markov chains. The resulting Markov chain admits powerful analysis techniques by utilizing *probabilistic model checkers*. In our new tool Blizzard, we provide *fully automated analysis and distribution computation of finite-state probabilistic programs, even in the presence of loops* (Chapter 3).

2. For infinite-state programs, we identify a class of probabilistic loops that can be replaced by a loop-free fragment, *eliminating one major obstacle to automated*

program analysis. The identified class contains classes of loops previously considered in literature [Kov08, GGH19]. Additionally, this thesis presents a type system that is able to identify the distribution of program variables, based on the properties of probability distributions (Chapters 4-5).

3. It has already been stated that probabilistic program analysis and inference are hard problems. To formally prove the hardness and to pinpoint the border of undecidability, we investigate the problem of invariant synthesis. Invariant techniques have been successfully used to characterize and analyze traditional loops [Kov08, RK07, HOPW19, VK23], and we investigate corresponding techniques in the probabilistic setting. Additionally, *this thesis relates an open problem in invariant synthesis to a famous problem in mathematics*, proving a hitherto unknown hardness result for invariant synthesis (Chapter 6).

1.4 Structure of the Thesis

This thesis first familiarizes the reader with concepts of probability theory, necessary to present our programming model and the formal semantics in Chapter 2.

We present the fully automated analysis of finite-state probabilistic programs in Chapter 3.

After providing treatment of finite-state programs, we turn our attention to programs with potentially infinite state-space. In Chapter 4, we provide an elegant formulation to recover distributions from moments over program variables with finite support. Subsequently, in Chapter 5, we identify a class of probabilistic loops that can be replaced by a loop-free fragment and provide a type system for distribution identification.

Invariant techniques are introduced in Chapter 6, first for deterministic and nondeterministic programs, followed by corresponding definitions for the probabilistic case. Additionally, we provide a novel proof that established hardness for a problem in invariant synthesis.

We survey existing methods to analyze probabilistic programs in Chapter 7. Finally, Chapter 8 recalls the individual results, concludes and examines possible future work.

CHAPTER 2

Preliminaries

In order to provide formal treatment of probabilistic programs and especially probabilistic loops, it is necessary to first introduce our programming model and provide its semantics. As the operational semantics of probabilistic programming languages can be based on Markov chains and their nondeterministic variants [Koz79, MSBK22, CS13, GKM14], it will be necessary to first introduce fundamental concepts from probability theory. Afterward, we will introduce our programming model and provide probabilistic program semantics in terms of Markov chains.

2.1 Notation and Probability Theory

Throughout the thesis, 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. The notation $\mathbb{K}[x_1, \ldots, x_m]$ denotes the polynomial ring over m variables with coefficients in some field $\mathbb{K}$. Further, we will use the symbol $\mathbb{P}$ for probability measures and $\mathbb{E}$ for the expected value operator, as defined in the following.

As we will later see, the operational semantics of a probabilistic program can be in general given by an uncountable, infinite-state Markov chain. To formally introduce Markov chains, we provide some foundations from probability theory, following [MSBK22, Dur10].

Definition 1 (σ-Algebra)**.** Let S be some set and 2^S its power set, then a *σ-algebra* $\mathcal{S}$ over S is a non-empty subset of 2^S that is closed under complementation, countable union and countable intersection. The smallest σ-algebra containing some specific set A is called the σ-algebra *generated by* A and is well-defined.

One notable instance of a generated σ-algebra is the *Borel σ-algebra*, which is generated by the open sets of $\mathbb{R}$.

Definition 2 (Measurable Space)**.** Let S be some set and $\mathcal{S} \subseteq 2^S$ a σ-algebra over S, then the pair $(S, \mathcal{S})$ is called a *measurable space.*

Typically, S represents the sample (test data) space and $\mathcal{S}$ represents all possible events. Now, to assign probabilities to events, we additionally require a *probability measure* $\mathbb{P}$.

Definition 3 (Probability Measure & Probability Space)**.** Let $(S, \mathcal{S})$ be a measurable space, then a *probability measure* $\mathbb{P}$ is a countably additive function $\mathcal{S} \to [0, 1]$ where additionally $\mathbb{P}(S) = 1$.

Together, the measurable space and the probability measure form a *probability space* $(S, \mathcal{S}, \mathbb{P})$.

Given a probability space $(S, \mathcal{S}, \mathbb{P})$, a random variable X is a measurable function $X : S \to \mathbb{R} \cup \{\infty, -\infty\}$. A function is measurable, if its preimage is measurable for every subset of the target space, in symbols if for any open set $B \subseteq \mathbb{R} \cup \{\infty, -\infty\}$ it holds that $X^{-1}(B) = \{\omega : X(\omega) \in B\} \in \mathcal{S}$. We denote indicator random variables for events $\omega \in \mathcal{S}$ as $\mathbb{I}(\omega)$, that is, $I(\omega) = 1$ if ω occurs and 0 otherwise.

Definition 4 (Expected Value)**.** The *expected value* $\mathbb{E}[X]$ of a *non-negative* random variable X on a probability space $(S, \mathcal{S}, \mathbb{P})$ is defined in terms of the Lebesgue integral, in symbols

$$\mathbb{E}[X] = \int_S X d\mathbb{P}.$$

If the image of the random variable is countable or even finite, the Lebesgue integral simplifies to the following and shows its intuitive meaning as a weighted average

$$\mathbb{E}[X] = \sum_{x \in X(S)} x\mathbb{P}(X = x).$$

This construction is generalized to signed random variables by using the positive part $X^+ := \max(0, X)$ and the negative part $X^- := \max(0, -X)$ and defining $\mathbb{E}[X] = \mathbb{E}[X^+] - \mathbb{E}[X^-]$. We say that the expected value of X *exists* if not both $\mathbb{E}[X^+]$ and $\mathbb{E}[X^-]$ are infinite. The kth *moment* of a random variable X is defined as $\mathbb{E}[X^k]$.

To model the evolution of a random variable throughout discrete time, we introduce the notion of a *stochastic process.*

Definition 5 (Stochastic Process)**.** A *stochastic process* $\langle X_i \rangle_{i \in \mathbb{N}}$ is a collection of random variables over the same probability space. For a given stochastic process, the random variable X_n represents the possible values in the nth step of the process.

This gives us the final piece to formally define Markov chains as a stochastic process that is only dependent on its predecessor in time.

Definition 6 (Measurable Markov Chain)**.** Given a probability space $(S, \mathcal{S}, \mathbb{P})$, the stochastic process $\langle X_i \rangle_{i \in \mathbb{N}}$ is a *Markov chain*, if there exists a Markov kernel[1] $p : S \times \mathcal{S} \to [0, 1]$ such that for all $n \in \mathbb{N}$ and $B \in \mathcal{S}$

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

If the set of possible states S is countable, or even finite, then the preceding definition may be reformulated, as follows.

Definition 7 (Countable Markov Chain)**.** Given a probability space $(S, \mathcal{S}, \mathbb{P})$, with countable sample space S, the stochastic process $\langle X_i \rangle_{i \in \mathbb{N}}$ is a *countable Markov chain*, if for all $n \in \mathbb{N}$

$$\mathbb{P}(X_{n+1} = x_{n+1} \mid X_0 = x_0, X_1 = x_1, \dots, X_n = x_n) = \mathbb{P}(X_{n+1} = x_{n+1} \mid X_n = x_n)$$

The essential property of a Markov chain is, that the next "state", i.e., the random variable X_{n+1} conditionally depends *exclusively* on the previous state X_n. Further, given a concrete state, the probability of transitioning between two states remains the same for the infinite duration of the stochastic process. Therefore, if the state space of the Markov chain is not only countable, but even finite, then it is possible to represent the process as a set of states and transitions. By this observation, we present the following, further specialization of the concept of a Markov chain.

Definition 8 (Finite Markov Chain)**.** A *finite Markov chain* is a three-tuple (Σ, σ_I, T), where Σ is the set of states, $\sigma_I \in \Sigma$ is a designated initial state and $T : \Sigma \times \Sigma \to \mathbb{R}_{\geq 0}$ is the transition relation that assigns each pair of states a non-negative probability such that the outgoing transition probabilities for each state sum up to 1.

Remark. The restriction to a single initial state is without loss of generality, as any initial distribution may be encoded by adding a fresh initial state and adding outgoing transitions according to the required initial distribution.

Finally, we briefly introduce the concept of *Bayesian statistics*, to the extent necessary for this thesis and following [CL08].

In Bayesian statistics, probabilities are considered as a *belief* about some event, where the key concepts are the *prior* $\mathbb{P}(X)$, the *posterior* $\mathbb{P}(X \mid Y)$ and the *likelihood* $\mathbb{P}(Y \mid X)$.

Here, the prior $\mathbb{P}(X)$ describes the initial belief about some event X. Then, new evidence in the form of an event Y is observed, leading the observer to adapt the belief about X, i.e., under the evidence Y, the observer adapts its initial belief $\mathbb{P}(X)$ about the event X and arrives at the new belief, the posterior $\mathbb{P}(X \mid Y)$. The actual update of the belief is done according to the following result of probability theory, known as Bayes' law, that states that the posterior distribution is proportional to the product of the prior and the likelihood of the observation.

[1] A *Markov kernel* generalizes the concept of a transition matrix for uncountable state spaces. For a formal definition, we refer to [Dur10]

Theorem 2.1 (Bayes' Theorem)**.** *Let X and Y be two events, further let $\mathbb{P}(Y) \neq 0$, then the following relation holds*

$$\mathbb{P}(X \mid Y) = \frac{\mathbb{P}(Y \mid X)\, \mathbb{P}(X)}{\mathbb{P}(Y)}$$

2.2 Probabilistic Programs

As mentioned in the introduction, probabilistic programs extend traditional programming languages with the ability to sample from probability distributions and to condition on observed events. We present our programming model here in full generality, and will restrict it during the thesis whenever necessary. The probabilistic programs in this thesis are based on a simple iterative programming language without functions and are purely probabilistic, i.e., there is no construct modelling demonic or angelic nondeterminism.

However, the language allows symbolic constants, i.e., arbitrary real constants and unrestricted real arithmetic. It allows the most common control-flow statements in the form of conditional branching and usual looping constructs. A detailed grammar describing the structure of a program is given in Figure 2.1, however the following points deserve special consideration as they may be new to the reader and at the heart of probabilistic programming.

Probabilistic Choice The language allows statements of the form

$$x \leftarrow x_1\ [p_1]\ x_2\ [p_2] \ldots x_m\ [p_m],$$

where $\sum_i p_i = 1$ and where all x_i are real numbers. The intended semantic of these statements is that after the instruction, it holds that $\mathbb{P}(x = x_i) = p_i$; that is, x is updated by x_i with probability p_i. As syntactic sugar, the language allows omitting the last probability, as it is fixed anyway.

Observe Statement The language allows conditioning on certain events, which intuitively means that all executions that do not conform to the observation are rejected. This corresponds to pruning the tree of possible executions. We illustrate our programming model with the next example.

Example 2. The following program models a simple random walk that starts at the origin and in each iteration randomly chooses a direction with probability p and $1 - p$, until it leaves the interval $(-10, 10)$. We refer to the update $x \leftarrow x + 1$ with probability p as the "left" direction; similarly, by the "right" direction, we mean the update by $x - 2$ with probability $1 - p$. Note that the step size is not symmetrical and the probability p is unknown. However, assuming we observed that in a concrete run, the random walk terminated after reaching 10, then this is possibly encoded using an observe-statement.

```
$x \leftarrow 0$
while $-10 < x$ and $x < 10$ do
```

```
    $x \leftarrow x + 1\ [p]\ x - 2$
end while
observe($x = 10$)
```

The intuition behind the observe-statements is, that even though p is unknown, our observation provides us with some information that will cause us to update our prior belief in a Bayesian manner. For example, if we repeat the experiment and continue to observe that the random walk tends to the right, then we will infer that $p > 0.5$ with high probability, when assuming a reasonably chosen prior.

```
⟨program⟩ ::= ⟨stmts⟩          ⟨stmts⟩ ::= ⟨stmt⟩+
⟨stmt⟩ ::= skip | ⟨assign⟩ | ⟨ifstmt⟩ | ⟨for-loop⟩ | ⟨while-loop⟩ | ⟨observe⟩
⟨assign⟩ ::= ⟨var⟩ ← ⟨aexpr⟩ | ⟨var⟩,⟨assign⟩,⟨aexpr⟩
⟨ifstmt⟩ ::= if ⟨bexpr⟩ then ⟨stmts⟩ (else if ⟨bexpr⟩ then ⟨stmts⟩)* [else ⟨stmts⟩] end if
⟨for-loop⟩ ::= for ⟨var⟩ in ⟨const⟩...⟨const⟩ do ⟨stmts⟩ end for
⟨while-loop⟩ ::= while ⟨bexpr⟩ do ⟨stmts⟩ end while
⟨observe⟩ ::= observe( ⟨bexpr⟩ )
⟨aexpr⟩ ::= ⟨arith⟩ | ⟨dist⟩ | ⟨choice⟩
⟨arith⟩ ::= ⟨const⟩ | ⟨var⟩ | ⟨arith⟩ (+ | - | * | /) ⟨arith⟩ | ⟨arith⟩**⟨arith⟩
⟨dist⟩ ::= ( Bernoulli | Normal | ... ) ( ⟨aexpr⟩+ )
⟨categorical⟩ ::= ⟨aexpr⟩ ([⟨const⟩] ⟨aexpr⟩)* [[⟨const⟩]]
⟨bexpr⟩ ::= true (⋆) | false | ⟨aexpr⟩ ⟨cop⟩ ⟨aexpr⟩ | not ⟨bexpr⟩ | ⟨bexpr⟩ (and | or) ⟨bexpr⟩
⟨const⟩ ::= r ∈ ℝ | ⟨sym⟩ | ⟨const⟩ ( + | - | * | / ) ⟨const⟩
⟨sym⟩ ::= a | b | ...          ⟨var⟩ ::= x | y | ...          ⟨cop⟩ ::= = | ≠ | < | > | ≤ | ≥
```

Figure 2.1: Context-free grammar, describing the syntax of our imperative probabilistic programming language.

It is apparent that the execution of a probabilistic program describes a stochastic process, i.e., the state of the program variables after n steps is a random vector. We will now formalize this intuition and give the semantics of the probabilistic programs, following [MSBK22]. To do so, we first construct the state space of the program and then construct the measure spaces necessary to describe the stochastic process modelled by the program. Finally, we will show how to construct a probability measure that describes the distribution of program variables after a fixed number of steps.

Let us first define the set of reachable states, necessary to construct a measurable space.

Definition 9 (State Space)**.** Let $\mathcal{P}$ be a probabilistic program with m variables, and let $\hat{\mathcal{P}}$ be the nondeterministic program where every instance of probabilistic branching

C is replaced by nondeterministic choice over the possible values of C. Then the *state set* S of $\mathcal{P}$ is the set of states reachable in $\hat{\mathcal{P}}$ from any initial state. By taking the Borel σ-algebra $\mathcal{S}$ over S we obtain the measurable space $(S, \mathcal{S})$, which will be denoted as the *state space*.

To capture the evolution of the program in time, we will need to define a measurable space not only for the possible program states, but also for all possible *sequences of program states*. This is known as the *sequence space* and constructed as follows.

Definition 10 (Cylinder Set)**.** Let π be a finite sequence of states, then the *cylinder set* $\mathit{Cyl}[\pi]$ is the set of infinite continuations of π, in symbols

$$\mathit{Cyl}[\pi] \coloneqq \{\pi\rho \mid \rho \in \Omega^\omega\}, \text{ where } \pi \in \Omega^+.$$

Definition 11 (Sequence Space)**.** Let (Ω, Σ) be a measurable space, then the corresponding *sequence space* is the measurable space $(\Omega^\omega, \Sigma^\omega)$, where Ω^ω is the set of infinite sequences $\{s_1 s_2 s_3 \ldots \mid \forall i \geq 1 : s_i \in \Omega\}$ and Σ^ω is the σ-algebra generated by the cylinder sets $\mathit{Cyl}[\pi]$ for all finite prefixes $\pi \in \Omega^+$.

Now we are able to define the set of runs through the probabilistic programs:

Definition 12 (Run Space)**.** Let $\mathcal{P}$ be a probabilistic program with m variables, and let $(S, \mathcal{S})$, be the corresponding *state space*. Then the *run space* $(R, \mathcal{R})$ of $\mathcal{P}$ is the sequence space $(S^\omega, \mathcal{S}^\omega)$ of the state space.

This allows us to define the stochastic process related to the execution of the probabilistic program.

Definition 13 (Run Process)**.** Let $\mathcal{P}$ be a probabilistic program with m variables, then the *run process* $\langle \Phi_n \rangle_{n \in \mathbb{N}}$ is a stochastic process in the run space. It maps a particular run $\rho \in R$ to the corresponding state after n steps, that is $\Phi_n(\rho) = \rho_n$. In accordance, every arithmetic expression e over the program variables, induces some stochastic process $\langle e_n \rangle$ that evaluates to the expression where every variable x is replaced by the value of the variable in Φ_n.

After defining the measurable space for our probabilistic programs, we now aim to construct a probability measure $\mathbb{P}$ to form a measurable space that provides the semantics of our programming model. Clearly, the next state in a program only depends on the current state and the instruction to be executed, hence we can define a Markov kernel so that our stochastic process actually is a measurable Markov chain. Doing so is rather technical, but it is straightforward to see how to construct these transition rules based on the expected semantics of the program statements.

After defining an initial distribution over program states μ, Kolmogorov's extension theorem guarantees the existence of a *unique* probability measure $\mathbb{P}$ on the run space

$(R, \mathcal{R})$ such that the run process $\langle \Phi_n \rangle$ is a Markov chain [Dur10]. By these means, it is possible to construct the *canonical probability* space $(R, \mathcal{R}, \mathbb{P})$ that provides semantics for our class of probabilistic programs. *During the remainder of this thesis, all constructions are to be understood with respect to this canonical probability space.*

CHAPTER 3

Automated Analysis of Finite-State Probabilistic Programs

In the introductory chapter, we already highlighted the omnipresent lurking of undecidability when doing program analysis. Therefore, to extract the distribution described by a probabilistic model, it is unconditionally necessary to specialize the class of programs that can be analyzed. In this chapter, we will show that for programs that have only a finite number of possible states, a *full characterization* of the program distribution is efficiently computable and can be done fully automated.

In Chapter 2, we showed that it is possible to define the semantics of our probabilistic programs in terms of uncountable Markov chains. In this chapter, the main idea is to transform the *finite-state* probabilistic program into an equivalent *finite-state* Markov chain that can be analyzed using established techniques, such as computer algebra systems or dedicated probabilistic model checkers. Similar ideas have been used to analyze Bayesian networks [SK20] and dynamic Bayesian networks [DDM16], however, to the best of our knowledge, there does not exist a treatment of a full-featured probabilistic programming language. Interestingly, the “inverse” direction has been investigated, where Holtzen *et al.* [HJV^{+}21] reduced the model-checking problem to probabilistic inference, showing that inference and model checking can complement each other.

By specializing the class of programs to those over a finite state space, we are able to support the whole programming model presented in the preliminaries, including symbolic parameters, assignments featuring arbitrary arithmetical expressions and observations. Additionally, when analyzing probabilistic programs, it is often unavoidable to have nonterminating paths through the program. While this may represent a problem for approaches that are simulation-based or perform path exploration, in the presence of a finite state-space, this does not hinder a complete treatment of program analysis and inference in our case.

In the following, we will present a translation of finite-state probabilistic programs into finite-state Markov chains and provide full automation of this process with our new tool Blizzard. However, before formally considering the translation rules, let us motivate the idea by an example.

Example 3 (Bounded Random Walk.)**.** Consider the following bounded random walk:

```
$x \leftarrow 4$
while $0 < x < 10$ do
    $f \leftarrow$ Bernoulli($p$)                          ▷ symbolic parameter $p$
    if $f = 1$ then
        $x \leftarrow x +$ DiscreteUniform($1, 3$)
    else
        $x \leftarrow x - 2$
    end if
end while
```

By standard techniques such as abstract interpretation [CC77], we can determine that the domain of x is $dom(x) = \{-1, \dots 12\}$. This allows us to construct the finite-state Markov chain shown in Figure 3.1 with state set $\Sigma = \{\langle -1 \rangle, \dots, \langle 12 \rangle\}$, initial state $\langle 4 \rangle$ and absorbing states $\{\langle -1 \rangle, \langle 0 \rangle, \langle 10 \rangle, \langle 11 \rangle, \langle 12 \rangle\}$.

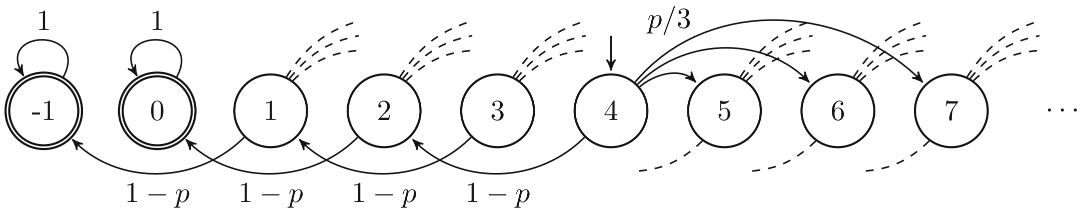


Figure 3.1: The Markov chain corresponding to the program in Example 3.

Typically, one would be interested in the distribution of x after program termination. For the corresponding Markov chain, this is equivalent to the reachability probability of all the reachable states that violate the loop guard. By the means presented in the remainder of this chapter, we can analyze the Markov chain of Figure 3.1 and obtain the results presented in Figure 3.2.

In Section 3.1, we provide full details on how to translate finite-state probabilistic programs into finite-state Markov chains. Following, in Section 3.2, we provide mathematical foundations from Markov chain theory that allow us to formally capture the distribution of a Markov chain (Section 3.2.1) and present *probabilistic model checkers*, specialized tools that can be used to analyze Markov chains (Section 3.2.2).

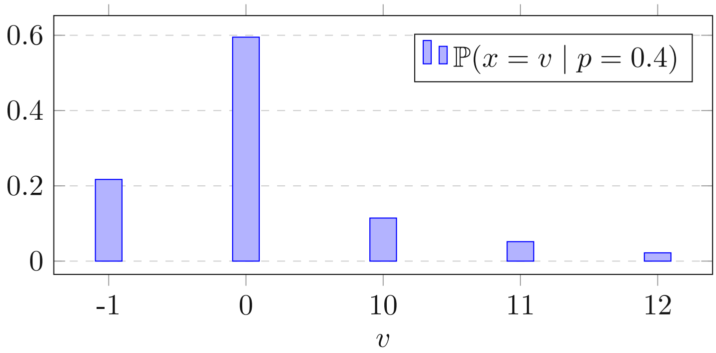


Figure 3.2: The probability distribution of x after termination in the previous example, assuming $p = 0.4$.

To enable fully automated analysis of probabilistic programs, we present our new tool Blizzard in Section 3.3. We compare our approach to existing approaches on benchmarks from literature, and highlight our strengths and investigate weaknesses (Sections 3.3.1-3.3.2). Finally, we show that our approach is not only useful on its own, but can also be utilized complementary with other tools (Section 3.3.3), followed by a brief conclusion in Section 3.4.

3.1 Translating Probabilistic Programs to Markov Chains

In this section, we formalize the intuition given in Example 3 and present a translation system from probabilistic programs to equivalent Markov chains. We consider a program to be a mapping from program counters to statements, more formally, let PC be the set of valid program counters and let Stmt be the set of program statements. Then a probabilistic program $\mathcal{P}$ is a function $\mathcal{P} : \mathsf{PC} \to \mathsf{Stmt}$ and, for ease of presentation, we assume that the last program statement is a skip-statement with special program counter $\circlearrowleft$, i.e., $\mathsf{PC} = \{0, 1, \dots, m, \circlearrowleft\}$.

The set Stmt comprises the statements presented in the programming model of Chapter 2, where we briefly recall the most important ones:

- Deterministic assignments of the form $x \leftarrow e$, where $x \in \mathsf{Vars}(\mathcal{P})$ and e is an expression over program variables.
- Probabilistic assignments of the form $x \leftarrow D(e_1, \dots, e_j)$, where $x \in \mathsf{Vars}(\mathcal{P})$, and D is a probability distribution with finite support and j parameters, e.g., $x \leftarrow \textsc{DiscreteUniform}(1, 5)$.
- Conditional branching by statements of the form **if** e **then** s_1 **else** s_2 **end**, where e is a boolean expression and s_1, s_2 are (lists of) statements.

- While-loops by statements of the form **while** e **do** s **end** , where e is a boolean expression and s is a list of sequential statements. Here, we encode **for**-loops as **while**-loops with a fresh loop counter variable.
- Observe statements of the form **observe**(e), where e is a boolean expression.

For a finite-state probabilistic program $\mathcal{P}$, let $\mathsf{Vars}(\mathcal{P}) = (v_1, \ldots, v_n)$ denote the (ordered) set of variables and note that by the finite-state property, every variable v_i in $\mathsf{Vars}(\mathcal{P})$ must have finite domain $dom(v_i)$. Without loss of generality, we assume that each variable has a symbol $\star$ that is only used to represent uninitialized values.

The preliminary chapter presented a general way of encoding the semantics of probabilistic programs in terms of an uncountably large Markov chain. As claimed above, for finite-state programs, we can explicitly construct this Markov chain and this chain directly encodes the semantic of the probabilistic program. We now present the translation $\mathsf{MC}(\mathcal{P})$, which maps a probabilistic program $\mathcal{P}$ to a Markov chain (Σ, σ_I, T). The set of states is given by the variable valuations and the current program counter. In addition, we introduce a special state $\langle \lightning \rangle$, that is entered only in case of an observation-violation. Hence, $\Sigma = \{dom(v_i) \times \ldots \times dom(v_n) \times \mathsf{PC}\} \cup \{\langle \lightning \rangle\}$ and $\sigma_I = (\star, \ldots, \star, 0)$.

Remark. To keep the presentation as simple as possible, we associate with each control-flow statement special program counters that can be used when presenting the translation rules. More specifically, for if-then-else-statements we assume pc_{if}, pc_{else} and pc_{end} representing the first statement of the if-branch, the else-branch and the subsequent statement, respectively. For while-statements, we assume pc_{head}, pc_{body} and pc_{end}, representing again the loop condition, the first statement of the loop body and the subsequent statement, respectively.

To describe the transition relation T, we use the expression pc' to denote the successor instruction of a statement. This is defined as:

$$pc' = \begin{cases} pc_{end} & \text{if } \mathcal{P}(pc) \text{ is the last statement of an if/else-branch} \\ pc_{head} & \text{if } \mathcal{P}(pc) \text{ is the last statement in the body of a loop} \\ pc + 1 & \text{otherwise} \end{cases}$$

We further define $\sigma \to^p \sigma'$ as shorthand for $(\sigma, \sigma', p) \in T$, and given an expression e and a state σ, $\sigma(e)$ denotes the evaluation of e in σ. If e contains variables that have state $\star$ in σ, $\sigma(e) = \star$ as well.

Now we define the transition relation by rules in Figure 3.3, where e is a deterministic expression and D is some discrete distribution over finite support.

By using these relations of T, we can translate a program $\mathcal{P}$ into a corresponding Markov chain. The correctness of the translation follows from the definition of probabilistic programs, since the underlying Markov chain constructed here *defines the semantic* of the program.

$$\textsc{Ass}\ \frac{\mathcal{P}(pc) = v_i \leftarrow e \qquad \sigma(e) = x}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_{i-1}, x, d_{i+1}, \dots, d_n, pc')}$$

$$\textsc{Dist}\ \frac{\mathcal{P}(pc) = v_i \leftarrow D(e_1, \dots e_j) \qquad \mathbb{P}(D(\sigma(e_1), \dots, \sigma(e_j)) = x) = p}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^p (d_1, \dots, d_{i-1}, x, d_{i+1}, \dots, d_n, pc')}$$

$$\textsc{If-}\top\ \frac{\mathcal{P}(pc) = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end} \qquad \sigma(e) \neq 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_n, pc_{\mathit{if}})}$$

$$\textsc{If-}\bot\ \frac{\mathcal{P}(pc) = \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end} \qquad \sigma(e) = 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_n, pc_{\mathit{else}})}$$

$$\textsc{While-}\top\ \frac{\mathcal{P}(pc) = \textbf{while}\ e\ \textbf{do}\ s\ \textbf{end} \qquad \sigma(e) \neq 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_n, pc_{\mathit{body}})}$$

$$\textsc{While-}\bot\ \frac{\mathcal{P}(pc) = \textbf{while}\ e\ \textbf{do}\ s\ \textbf{end} \qquad \sigma(e) = 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_n, pc_{\mathit{end}})}$$

$$\textsc{Observe-}\top\ \frac{\mathcal{P}(pc) = \textbf{observe}(e) \qquad \sigma(e) \neq 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 (d_1, \dots, d_n, pc')}$$

$$\textsc{Observe-}\bot\ \frac{\mathcal{P}(pc) = \textbf{observe}(e) \qquad \sigma(e) = 0}{\sigma = (d_1, \dots, d_n, pc) \rightarrow^1 \langle \lightning \rangle}$$

$$\textsc{Terminal-State}\ \frac{}{\sigma = (d_1, \dots, d_n, \circlearrowleft) \rightarrow^1 \sigma}$$

$$\textsc{Violation-State}\ \frac{}{\langle \lightning \rangle \rightarrow^1 \langle \lightning \rangle}$$

Figure 3.3: Transition relation T of a finite-state probabilistic program $\mathcal{P}$, with statements as in Figure 2.1.

3.2 Analyzing Markov Chains

Figure 3.3 presented a translation from probabilistic programs to Markov chains, it is now of immediate interest to extract the distribution of a finite-state Markov chain. In this section, we first provide the necessary foundations from Markov chain theory and then examine practical approaches to analyze the distribution that is implicitly encoded by a Markov chain.

3.2.1 Mathematical Characterization

As presented in the preliminary section, Definition 8, a finite-state Markov chain (Σ, σ_I, T) represents a set of states and probabilistic transitions between them. We consider discrete-time Markov chains, where time is quantized into discrete *steps* and in each time step exactly one transition happens. It is convenient to view the set of states as a vector $\mathbf{s}$ and the transition relation as a matrix $\mathbf{P}$. To relate this with the previous notion, we require that the matrix reflects the transition relation, i.e., that for all $i, j \in \Sigma$

$$T(s_i, s_j) = \mathbf{P}_{ij}$$

Then, the fundamental Markov chain theory guarantees that the probability distribution over all states after n steps is given by $\sigma_I \cdot \mathbf{P}^n$. To analyze probabilistic programs, we are interested in the distribution over the set of states *after program termination*. This is closely related to the steady-state distribution $\lim_{n\to\infty} \mathbf{P}^n$, if it exists.

Some state σ of the Markov chain is said to be *absorbing*, if it has a self-loop with probability one, i.e., $T(\sigma, \sigma) = 1$. It can be shown, that if every state of the Markov chain can reach some absorbing state with positive probability and in a finite number of steps, then the rows of $\mathbf{P}^n$ converge as n goes towards infinite [GS06].

For the finite-state Markov chains constructed by the relation from Figure 3.3, all terminal states with program counter $\circlearrowleft$ and the violation state $\langle ↯ \rangle$ are absorbing. However, there remains the possibility that some execution may reach a state in Figure 3.3 that does not terminate at all, i.e., no terminal state is reachable. To handle this scenario, we merge all the states in the Markov chain that cannot reach a terminal state into the violation state $\langle ↯ \rangle$. This does not change the properties of terminating runs, since no terminating run can reach such a state by definition, as these do not have a path to a terminating state by assumption. By this construction, all states in the chain can reach an absorbing state and hence $\mathbf{P}^\infty := \lim_{n\to\infty} \mathbf{P}^n$ is well-defined. Moreover, the probability that the program halts in some configuration is exactly $\sigma_I \cdot \mathbf{P}^\infty$, since the terminal states are absorbing and will never be left.

To compute the distribution over the final states, it is necessary to compute $\mathbf{P}^\infty$. We do so by partitioning the set of states into two sets, absorbing states r and the remaining states, the transient states t. By the definition of absorbing states, these have self-loops, hence there cannot be any transitions from an absorbing state to some other state. Therefore, the transition matrix has the structure of (3.1), where $\mathbf{Q}$ and $\mathbf{R}$ describe the transitions within the transient (resp. absorbing) states and $\mathbf{E}$ is the unit matrix:

$$\begin{bmatrix} t \\ r \end{bmatrix}_{n+1} = \underbrace{\begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{E} \end{bmatrix}}_{\mathbf{P}} \cdot \begin{bmatrix} t \\ r \end{bmatrix}_{n} \tag{3.1}$$

Given this formulation, the stationary distribution can be computed as $\mathbf{P}^\infty = (\mathbf{E} - \mathbf{Q})^{-1} \cdot \mathbf{R}$, as shown in [GS06]. Given these results from Markov chain theory, we conclude that we can *efficiently* compute the distribution over all states after termination by straightforward linear algebra over Figure 3.3. The following example continues Example 3 and illustrates the concepts presented in this section.

Example 4. We recall that the bounded random walk from Example 3 has the semantics given by the Markov chain, in Figure 3.1. Note that every state can reach some terminal state within a finite number of steps, hence no state is merged into $\langle \lightning \rangle$. As presented above, we partition the set of states into transient and recurrent states to define the state vector

$$\begin{bmatrix} t \\ - \\ r \end{bmatrix} = \left[\begin{array}{ccc|ccccc} \langle 1 \rangle & \cdots & \langle 9 \rangle & \langle 10 \rangle & \langle 11 \rangle & \langle 12 \rangle & \langle 0 \rangle & \langle -1 \rangle \end{array}\right]^T$$

We then extract the nontrivial part of the transition matrix, where we use $q = 1 - p$ for brevity. Note that the bottom half of the matrix scheme presented in (3.1) has been omitted for reasons of space.

$$\left[\begin{array}{c|c} \mathbf{Q} & \mathbf{R} \end{array}\right] = \left[\begin{array}{ccccccccc|ccccc}
0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & q \\
0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & q & 0 \\
q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & q & 0 & 0 & \frac{p}{3} & \frac{p}{3} & \frac{p}{3} & 0 & 0
\end{array}\right]$$

By [GS06], we are able to compute $\mathbf{P}^\infty$ as $(\mathbf{E} - \mathbf{Q})^{-1} \cdot \mathbf{R}$ using standard computer algebra systems (CASs). As initial state, we had $\langle 4 \rangle$ and therefore use the corresponding initial distribution vector s over the initial states. This gives us the following result, completely characterizing the terminal states of the probabilistic program from Example 3:

$$s \cdot (\mathbf{E} - \mathbf{Q})^{-1} \cdot \mathbf{R} = \begin{bmatrix}
-\frac{p^2\left(13\,p^7-73\,p^6-282\,p^5+1701\,p^4-2511\,p^3+1863\,p^2+1944\,p+729\right)}{9\,(5\,p^9-57\,p^8+51\,p^7+643\,p^6-2883\,p^5+4536\,p^4-3510\,p^3-486\,p^2+1701\,p-729)} \\
-\frac{p^3\left(16\,p^6-106\,p^5+9\,p^4+684\,p^3-1026\,p^2+810\,p+1701\right)}{9\,(5\,p^9-57\,p^8+51\,p^7+643\,p^6-2883\,p^5+4536\,p^4-3510\,p^3-486\,p^2+1701\,p-729)} \\
-\frac{p^3\left(7\,p^6-37\,p^5-24\,p^4+495\,p^3-891\,p^2+1053\,p+486\right)}{9\,(5\,p^9-57\,p^8+51\,p^7+643\,p^6-2883\,p^5+4536\,p^4-3510\,p^3-486\,p^2+1701\,p-729)} \\
-\frac{9\,(p-1)\left(p^8-12\,p^7+63\,p^6-118\,p^5+99\,p^4+93\,p^3-261\,p^2+216\,p-81\right)}{5\,p^9-57\,p^8+51\,p^7+643\,p^6-2883\,p^5+4536\,p^4-3510\,p^3-486\,p^2+1701\,p-729} \\
\frac{9\,p\,(p-1)^2\left(2\,p^6-18\,p^5+39\,p^4+22\,p^3-153\,p^2+216\,p-108\right)}{5\,p^9-57\,p^8+51\,p^7+643\,p^6-2883\,p^5+4536\,p^4-3510\,p^3-486\,p^2+1701\,p-729}
\end{bmatrix}$$

These symbolic results can be evaluated for arbitrary valuations of p to characterize the behavior of the probabilistic program.

In this section, we showed that Markov chains can be described by standard linear algebra and hence can be analyzed by standard CAS tools. However, due to the prevalence of Markov chains in various engineering disciplines, there has been the need to model and verify large-scale Markov chains. This originated the need for specialized implementations, beyond the capabilities of standard CAS tools. Such *probabilistic model checkers* offer the functionality to analyze Markov chains and after a brief introduction, we show how to utilize these tools to analyze probabilistic programs.

3.2.2 Probabilistic Model Checkers

We start by providing some intuition about the approaches taken by contemporary probabilistic model checkers and how these tools internally represent the model, i.e., the Markov chain.

The two most widely used probabilistic model checkers are Storm [HJK$^+$20] and PRISM [KNP11], each of them offering various functionalities, such as the analysis of (parameterized) discrete-time and continuous-time Markov chains or nondeterministic versions thereof (Markov decision process). For our purposes, we will utilize these tools to analyze discrete-time, finite-state Markov chains, potentially featuring symbolic parameters.

Like in traditional model checking, the state-space explosion represents the main obstacle when verifying non-trivial models. This necessitates an efficient encoding of the internal model, while allowing efficient computations at the same time. Since the structure of the model itself has a major impact on the size of an encoding, most available probabilistic model checkers allow the user to choose from multiple encoding strategies [Par03, HJK$^+$20]. We will briefly present three common strategies supported by both PRISM and Storm.

Explicit Encoding. When performing an explicit encoding, the Markov chain is explicitly instantiated, represented by vectors and matrices, which allows fast computations at the price of increased memory requirements. Since typically many entries in the resulting matrices are zero, model checkers utilize specialized representations for sparse matrices that reduce memory pressure.

Symbolic Encoding. Due to symmetry within the model itself, symbolic encodings may be used to provide a compact representation of the model in memory, similar to symbolic encodings in traditional model checking. When using such an encoding, a generalization of binary decision diagrams (BDDs) is used to store the matrices during the computation. This representation allows substructure-sharing and efficient representation of sparse matrices. The immediate drawback of these encodings is that not all necessary operations, e.g., random element access, can be performed efficiently on them.

Hybrid Encoding. To avoid the limitations mentioned for the previous approaches, the hybrid encoding (tries to) combine the best of both worlds. Here the model is stored using a symbolic encoding, but when needed, relevant fragments are converted into an explicit representation to facilitate efficient computations.

As briefly hinted before, both Storm and PRISM allow the user to utilize *symbolic transition probabilities*. Here, a transition probability in the Markov chain is replaced by a symbolic parameter and the result of the computation is in dependence of the symbolic parameter. Depending on the model checker used, the usage of symbolic parameters may influence the encoding strategy. This is due to the fact that parametric analysis is mostly done by state elimination and therefore most easily performed in explicit encodings.

To pass a Markov chain model to a tool, it needs to be appropriately encoded. There exist several file formats that are supported by various model checkers, where one especially widely supported modeling language is the PRISM language. Here, the model is encoded by defining variables and guarded transitions between states, where each transition is assigned a probability. When encoding the Markov chain from Example 3, one may model it as follows:

```
dtmc //discrete-time markov chain

const double p; // symbolic parameter

module walk
    x : [-1..12] init 4; // range constraint and initial value
    [] 0 < x & x < 10   -> p/3: (x' = x+1) +
                           p/3: (x' = x+2) +
                           p/3: (x' = x+3) +
                           (1-p): (x' = x-2);
    [] x <= 0 | x >= 10 -> true; // self-loop
endmodule
```

To query about properties of a model, the PCTL query language is widely supported. Using the query language, we can, e.g., query the reachability probability of the possible final states:

```
P=? [ F x = -1 ]; // probability that finally variable x is equal -1
P=? [ F x =  0 ];
P=? [ F x = 10 ];
P=? [ F x = 11 ];
P=? [ F x = 12 ];
```

Finally, the model and the queries can be jointly passed to a model checker, which dutifully computes the results and outputs them. Depending on the way the queries are formulated, the model checker may not be able to re-use the result of previous computations. However, for non-parametric models, both Storm and PRISM offer the possibility to compute all reachability probabilities at once, which is especially useful in our use-case.

3.3 Automated Probabilistic Program Analysis

In the previous sections, we introduced probabilistic model checkers (Section 3.2) and a translation of finite-state probabilistic programs into equivalent Markov chains (Section 3.1). To follow up, this section will show the feasibility of *automated probabilistic program analysis* by a translation to Markov chains and a subsequent analysis using probabilistic model checkers. We have implemented the approach in our new tool Blizzard which is freely available at `https://github.com/julian-muellner/blizzard`. To guarantee an easy setup, we also provide a Docker container that has Storm, PRISM and Blizzard pre-installed at `https://hub.docker.com/repository/docker/jmuellner/blizzard`.

The main steps of Blizzard are:

1. Convert a finite-state probabilistic program into an equivalent Markov chain, represented in PRISM format.

2. End-to-end analyze a finite-state probabilistic program using a configurable probabilistic model checkers as back-end, e.g., Storm or PRISM

3. Compute (joint) probability distributions that are modelled by a (parameterized) probabilistic program.

Steps 1. and 3. are the main contributions of our work, and complement probabilistic model checkers with novel features to compute distributions of finite-state probabilistic programs. Blizzard also allows the user to customize the used tool-chain, allowing for a comparison of the probabilistic model checkers used in the back-end. We now present two examples that illustrate the feasibility of the Blizzard approach.

Example 5. The following program represents a slight variation of the random walk in Example 3, where, as a non-trivial modification, it is observed that the random walk never visited a state where $x = 8$.

```
$x \leftarrow 4$
while $0 < x < 10$ do
    $f \leftarrow \textsc{Bernoulli}(p)$
    if $f = 1$ then
        $x \leftarrow x + \textsc{DiscreteUniform}(1, 3)$
    else
        $x \leftarrow x - 2$
    end if
    observe $(x \neq 8)$
end while
```

The program can be passed directly to Blizzard to obtain the probability of the various outcomes, including the probability of a violated assertion. Within seconds, Blizzard outputs a fully symbolic characterization of the probability distribution of x after termination. As an example, it holds that the (unnormalized) probability that after termination $x = 12$, without ever violating the observation, with respect to the symbolic constant p is

$$\mathbb{P}(x = 12) = \frac{p^2(5p^6 + 144p^3 - 120p^4 + 16p^5 - 135p^2 - 162p)}{9(8p^8 - 48p^7 + 159p^6 - 476p^5 + 762p^4 - 675p^3 - 216p^2 + 486p - 243)}$$

Example 6. To illustrate the use-case of probabilistic program analysis for verification, consider the following program from [CRN+13]. It models sampling from a uniform distribution that ranges over $[0, N)$, given only access to one fair random bit, i.e., a Bernoulli distribution with $p = 0.5$.

```
$g \leftarrow N$
while $g \geq N$ do
    $g, n \leftarrow 0, 1$
    while $n < N$ do
        $n \leftarrow 2n$
        if Bernoulli(0.5) then
            $g \leftarrow 2g$
        else
            $g \leftarrow 2g + 1$
        end if
    end while
end while
```

To verify that the program for $N = 100$ indeed satisfies its specification, we invoke Blizzard on the program:

```
> python3 blizzard.py benchmarks/uniform.pp --analyze g
Resulting program has 742 states
Storm: Time for model input parsing: 0.107s.
Storm: Time for model construction: 0.135s.
Storm: Time for model checking: 0.000s.

(g: 99): 0.009999999999999999
/* snip */
(g:  2): 0.009999999999999999
(g:  3): 0.009999999999999999
(g:  1): 0.009999999999999999
(g:  0): 0.009999999999999999
Elapsed time: 0.5021121501922607 s
```

We conclude, that the distribution over g is indeed uniform, even though there are some numerical inaccuracies within the model checker[1] Note that even though the resulting

[1] These can be resolved by using the `--exact` switch, at the price of increased runtime.

Markov chain has 742 states and Blizzard is a proof-of-concept implemented in the interpreted language python, the end-to-end runtime is only half a second, showing the practical potential of our approach.

3.3.1 Comparison to Other Tools

Considering the complexity and variety of the probabilistic programming landscape, there cannot be a one-size-fits-all approach that is suited best for every use-case and program class. In this section, we will present existing tools for probabilistic program analysis and compare them to our approach and our tool Blizzard, both qualitatively and quantitatively.

PSI. As one of the first tools to provide *exact* inference over probabilistic programs, PSI is a tool widely used in benchmarking comparisons [GMV16]. The main line of thought here is to step over the program line-by-line and maintain a probability distribution over each variable, while discharging symbolic computations via a configurable CAS back-end, such as Maple or Mathematica. It supports complex arithmetic and infinite state-spaces, but is restricted to statically bounded loops without symbolic constants, which is not the case in Blizzard. Another drawback is that PSI is heavily dependent on the back-end CAS and may have to report unevaluated integrals that could not be solved.

Polar. The Polar tool is able to compute *moments* of variable monomials in certain probabilistic loops [MSBK22]. The key idea here is to build a system of recurrences over expected values, instead of tracing the whole distribution. The resulting recurrences can then be discharged, where for the class of supported programs, these recurrences can always be solved fully automatically, i.e., the technique is complete for its programming model. Polar supports infinite state-spaces, polynomial arithmetic and potentially unbounded runs through the program. Compared to the other tools, including Blizzard, the main limitations of Polar are the lack of observe-statements and the restriction of branching conditions to use only variables over a finite domain. One additional restriction is that any variable may not be self-dependent, except linearly, i.e., the assignment $x \leftarrow 3x + 2y$ is fine (assuming y does not depend on x), while the assignment $x \leftarrow x^2$ is not.

DICE. As one of the more recent tools in the field, DICE is aimed at discrete, finite-state programs [HVM20]. It is able to provide fast inference by utilizing a symbolic approach based on BDDs. However, this comes at the price of quite heavy restrictions, such as the restrictions of variables to single bits (with integers as syntactic sugar) and coin flips as the only source of randomness. The tool does not support symbolic parameters and handles loops by unrolling, i.e., the loop must be statically bounded, which is not the case in Blizzard.

To conclude the preceding comparison, Table 3.1 provides a brief visual comparison between the tools. In comparison to other tools, Table 3.1 shows that the greatest strength of our approach in BLIZZARD is flexibility over finite state-spaces, since we can handle arbitrary discrete programs. The tool which is most closely related with respect to our envisaged use-case is DICE. Here, the biggest advantage of our approach in BLIZZARD is the ability to handle potentially unbounded loops that feature symbolic parameters, a class of programs outside the reach of DICE. However, this comes at the price of increased complexity when performing inference.

Feature	BLIZZARD	Polar	DICE	PSI
Finite-State Programs	✓	✓	✓	✓
Infinite-State Programs	✗	✓	✗	✓
Symbolic Parameters	✓[2]	✓	✗	✗
Unbounded Loops	✓	✓	✗	✗
Statically Bounded Loops	✓	✓	✓	✓
Observe-Statements	✓	✗	✓	✓
Continuous Distributions	✗	✓	✗	✓

Table 3.1: Qualitative Comparison of BLIZZARD, Polar, Dice and PSI.

Quantitative comparison. To provide also a quantitative comparison, we ran the previously mentioned tools (except Polar) on a set of benchmark instances. The benchmarks are from literature on probabilistic programming and Bayesian networks [MM05, CRN$^+$13, GMV16, MSBK22, BGHS17] in addition to the bounded random walk from Example 3. The benchmarking results were obtained on a machine with 8GB of RAM, and an Intel Core i5-8250U CPU, using the `hyperfine` benchmarking utility, running each benchmarking instance 20 times. We used the most recent PSI version at commit `f31dc38`, DICE version `1.0` available via Docker and ran BLIZZARD via the provided Docker container with Storm as backend. We also tested the version of PSI specialized for discrete programs, deploying dynamic programming for program analysis (DP).

The tool Polar is omitted from the comparison, since it actually characterizes moments over program variables, does not support observes and in general cannot recover distributions. To allow a fair comparison between the respective ideas, we note that both DICE and PSI are written in compiled languages (OCAML and D respectively), while BLIZZARD is a proof-of-concept implementation written in Python (even though the back-end tools are written in compiled languages). Nevertheless, BLIZZARD is able to compete with aforementioned tools, as shown in Table 3.2.

Table 3.2 shows, that our approach can compete with PSI on the loop-free benchmarking instances, where DICE outperforms both tools. When moving to programs that feature unbounded probabilistic loops, for the vast majority of benchmarks, only BLIZZARD can provide complete results. When truncating the execution of the program and hence accepting results that are only correct when neglecting longer runs, then the results show

[2] distributions only

Class	Benchmark	Blizzard	DICE	PSI	PSI (DP)
Bayesian Networks	Burglar Alarm	214	18	212	141
	Evidence 1	252	18	137	134
	Evidence 2	260	18	147	138
	Grass	285	18	212	146
	Murder Mystery	264	19	153	142
	Noisy-Or	702	18	664	198
Prob. Loops	Bounded Random Walk	299	$268^\dagger$	$11.531^\dagger$	$1.648^\dagger$
	Bounded Random Walk Param.	4.923	$\star$	$\star$	$\star$
	Duelling Cowboys	267	$22^\dagger$	$928^\dagger$	$226^\dagger$
	Duelling Cowboys Param.	271	$\star$	$\star$	$\star$
	Kruskal Card Trick	23.221	$\star$	$\star$	52.211
	Loopy 1	255	$20^\dagger$	$1.065^\dagger$	$328^\dagger$
	Loopy 2	250	$19^\dagger$	$758^\dagger$	$225^\dagger$
	Uniform (N = 20)	309	$\star$	$\star$	$3.361^\dagger$
	Uniform (N = 100)	548	$\star$	$\star$	$15.701^\dagger$
	Uniform (N = 500)	2.507	$\star$	$\star$	Timeout

Table 3.2: Quantitative Comparison of Blizzard, PSI and DICE, all results in milliseconds, timeout one minute. Entries with $\star$ and † indicate that the program cannot be statically bounded and hence PSI/DICE are not powerful enough to analyze those directly. Runtimes annotated with † indicate that all unbounded loops have been bounded to at most 100 iterations, yielding incomplete, best-effort treatment.

that Blizzard outperforms PSI, while DICE is faster than both of them. However, we stress again that Blizzard provides *safe and exact* results, for runs of arbitrary length.

3.3.2 Limitations of Our Approach

In the previous comparison, we noticed the advantage that the symbolic nature of our analysis allows us to re-use states that have been encountered before. As an example, re-consider the random walk from Example 3. This program has infinite traces, which may present a problem for simulation-based approaches and which cannot be directly encoded in PSI and DICE. Nevertheless, our presented here can solve the problem very efficiently, since when constructing the Markov chain only the number of reachable states are relevant, not the exponentially many paths through the program.

However, whenever loop counters are used (either as for-loop or explicitly), this will prohibit state re-use across loop iterations and force an exploration of all the paths in the program, as in the following program:

```
$x \leftarrow 0$
for $i$ in $1 \ldots 100$ do
    $x \leftarrow x + \textsc{DiscreteUniform}(1, 3)$
end for
```

When translating this program into a Markov chain, we arrive at a chain with 60.204 states. The probabilistic model checker Storm takes roughly 15 minutes to analyze the model. This worst-case example shows where the presented approach performs unfavorably. We hence conclude that our approach, and Blizzard, is best suited for programs over a finite-state space, where the same state may be entered multiple times, i.e., while-loops that may have runs of unbounded length.

3.3.3 Complementing Other Inferences Engines

By design, Blizzard is specialized to finite state-spaces, while some use-cases inherently are over an infinite state-space. Even though Blizzard is not designed to analyze infinite state programs, it provides highly performant inference for a restricted subclass, where existing, more general tools, may perform significantly worse. In this section, we show how Blizzard can be used to perform inference in combination with other inference engines, which may have different strengths.

As briefly mentioned in Section 7.3 and Section 3.3.1, existing tools mostly support two classes of loops: *statically bounded* for-loops and *non-statically bounded, terminating* while-loops. The latter class does not require a static bound on the number of loop iterations, but existing tools fail in case there are infinitely long runs through a program. As an example, the bounded random walk in Example 3 did admit infinite runs, prohibiting an analysis with existing tools. As such, Blizzard offers itself as a powerful primitive for finite-state fragments that existing approaches cannot analyze.

To further illustrate the power of our approach in this context, consider the following variant of the stochastic turtle-hare game [CS14] in Figure 3.4. The program models the race of a turtle and a hare, as in Aesop's fable, where the hare is moving whimsically with some probability p, but if it does, it moves forward a distance uniformly chosen between 1 and 3. The turtle, in contrast, has a head-start of 2 units and constantly moves one unit each time-step. If the turtle can outrun the hare by a margin of 5 units, it wins, but loses if the hare eventually catches up. After performing the race five times, we observe that the hare did win four out of those five races and while initially believing that all values for p are equally likely, we would like to update our belief of p, according to this observation.

The program of Figure 3.4 can neither be analyzed by existing tools like PSI [GMV16], nor by Blizzard. The unbounded loop in the procedure race, prohibits analysis by existing tools and the continuous prior on p impedes the methodology employed by Blizzard. However, it is possible to transform the procedure Race into an equivalent, finite-state loop and employ Blizzard to extract the resulting distribution with respect to the symbolic parameter p. To do so, we introduce a new variable $\Delta := \mathit{turtle} - \mathit{hare}$ and arrive in the equivalent finite-state program of Figure 3.5.

This finite-state program Race can be analyzed by our approach, and we obtain the

```
procedure Race(p)
    $hare, turtle \leftarrow 0, 2$
    while $hare < turtle \land turtle < hare + 5$ do
        $turtle \leftarrow turtle + 1$
        if Bernoulli($p$) then
            $hare \leftarrow hare +$ DiscreteUniform$(1, 3)$
        end if
    end while
    return $hare \geq turtle$
end procedure

procedure Main
    $p \leftarrow$ Uniform$(0, 1)$
    $hareWins \leftarrow 0$
    for $i$ in $1 \ldots 5$ do
        if Race(p) then
            $hareWins \leftarrow hareWins + 1$
        end if
    end for
    observe($hareWins = 4$)
end procedure
```

Figure 3.4: A Probabilistic Variant of the Turtle-Hare Race [CS14].

```
procedure Race(p)
    $\Delta \leftarrow 2$
    while $\Delta > 0 \land \Delta < 5$ do
        $\Delta \leftarrow \Delta + 1$
        if Bernoulli($p$) then
            $\Delta \leftarrow \Delta -$ DiscreteUniform$(1, 3)$
        end if
    end while
    return $\Delta \leq 0$
end procedure
```

Figure 3.5: A finite-state equivalent of the **Race** procedure in Figure 3.4 .

symbolic, closed-form solution for $p_h := \mathbb{P}(\textsc{race}(p) = \top)$ which is

$$p_h = \frac{10p^4 - 21p^3 + 27p}{37p^4 - 183p^3 + 324p^2 - 243p + 81}$$

With the goal of performing inference, we can replace the subprocedure call to Race by a simple coin-flip with probability p_h. The resulting program fits the programming model of existing tools, as the only loop is statically bounded and hence the program can

```
procedure MAIN
    $p \leftarrow \textsc{Uniform}(0, 1)$
    $p_h \leftarrow \mathbb{P}(\textsc{race}(p) = \top)$                ▷ Obtained with BLIZZARD
    $hareWins \leftarrow 0$
    for $i$ in $1 \ldots 5$ do
        if $\textsc{Bernoulli}(p_h)$ then
            $hareWins \leftarrow hareWins + 1$
        end if
    end for
    observe($hareWins = 4$)
end procedure
```

be analyzed using such approaches. This extensive example shows the envisaged use-case for BLIZZARD, besides being used as a stand-alone *exact* inference tool for probabilistic programs over finite state-spaces.

3.4 Conclusion

In this chapter, we investigated the fully-automated translation of finite-state probabilistic programs to Markov chains and the applicability of probabilistic model checkers to analyze the resulting Markov chains. Finally, we compared our approach to existing tools and benchmarked the performance.

We have shown that for finite-state probabilistic programs, especially while-loops, the presented methodology is a promising approach and extends the class of probabilistic programs that can be (efficiently) analyzed.

CHAPTER 4

Recovering Probability Distributions from Moments

In Chapter 3, we investigated the automated analysis of finite-state loops by means of a translation to Markov chains and applied probabilistic model checkers to analyze those. This approach is very well suited for certain classes of programs, while suffering from limitations due to the state-space explosion for other classes of programs. However, it may be asked what can be done for programs where the approach presented in Chapter 3 performs unfavorably or in programs with infinite state-spaces. To perform inference, knowledge of the distribution modelled by the program is still of great importance.

Previous work in this area investigated the characterization of *moments* of program variables, i.e., expected values of the form $\mathbb{E}[x^k]$ for program variable x and $k \in \mathbb{N}$ [BKS19, MSBK22]. For a restricted class of infinite-state loops, this approach is sound and complete, that is, it is able to compute *arbitrary moments* of program variables. The main limitations here are the restriction to programs that can be described by linear recurrences over their moments and the lack of observe-statements. Additionally, this technique does not attempt to recover the distribution of the program variables from the information encoded in the moments.

The problem of recovering distributions from given moments, known as *the moment problem*, is of surprising mathematical difficulty and has been under investigation for over a hundred years [Sch17]. More specifically, assuming *all moments exist*, that is $\int X^k d\mathbb{P}$ exists for each $k \in \mathbb{N}$, *under which conditions does the infinite sequence of moments uniquely specify a distribution?*

The answer is very subtle, as witnessed by the various variations of the moment problem [Dur10]. It can be shown, that if the distribution is allowed to have positive measure on the whole real line (Hamburger moment problem) or on the interval $[0, \infty)$ (Stieltjes moment problem), then the answer to the aforementioned question is negative, that is, there are *either no solutions or infinitely many distributions* with these moments. Only under stronger conditions on the limit of the moment sequence, such as Carleman's condition, the uniqueness of the distribution can be asserted. This inherent complexity is the reason why existing work often resorts to approximate solutions that try to find some distribution that fits the given moments as closely as possible [KMS+22]. However, *on bounded intervals* of the real line (Hausdorff moment problem), the answer is positive and if the support[1] of the random variable is finite, then even finitely many moments are sufficient to *uniquely* characterize the distribution [Dur10, Sch17].

In this chapter, we will investigate how to extend the approaches of [BKS19, MSBK22] to allow the recovery of distribution for program variables with finite support. *Note that the programs we consider in this chapter do not necessarily have finitely many states as in Chapter 3*, as the programs under consideration may have an infinitely large state space. It is only necessary that the *result*, i.e., some random variable of interest, has finite support. Thus, the methodology presented here allows us to extract the distribution and perform inference over a different class of probabilistic programs than in Chapter 3.

In the following sections, we will first examine the univariate case (Section 4.1), followed by an elegant generalization to the multivariate case (Section 4.2), allowing us also to recover *joint probability distributions* over multiple random variables. But first, to provide the reader with an intuition, let us illustrate the key idea with a simple example.

Example 7. Consider the following program, where the program variables X and Y are clearly *not* stochastically independent and have finite support $\{0, 1\}$.

```
$X, Y, Z \leftarrow 0, 0, 1$
while $Z = 1$ do
    $Z \leftarrow \text{Bernoulli}(0.5)$
    $X \leftarrow \text{Bernoulli}(p)$
    $Y \leftarrow 1 - X$
end while
```

For now, assume the moments over program variables and monomials are given, in this case as $\mathbb{E}[X] = p$, $\mathbb{E}[Y] = 1 - p$ and $\mathbb{E}[XY] = 0$. Note, that due to the stochastic dependence of X and Y, it does not hold that $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$.

In combination with the defining equations of the expected value operator, we can write

[1] The support of a discrete random variable X is the set of values that may be assumed with positive probability, i.e., $\{x \mid P(X = x) > 0\}$

down the following system of equations:

$$\begin{aligned}
1 &= \sum_{x,y} \mathbb{P}(X = x, Y = y) \\
p = \mathbb{E}[X] &= \sum_{x} x\mathbb{P}(X = x) = \mathbb{P}(X = 1) = \mathbb{P}(X = 1, Y = 0) + \mathbb{P}(X = 1, Y = 1) \\
1 - p = \mathbb{E}[Y] &= \sum_{y} y\mathbb{P}(Y = y) = \mathbb{P}(Y = 1) = \mathbb{P}(X = 0, Y = 1) + \mathbb{P}(X = 1, Y = 1) \\
0 = \mathbb{E}[XY] &= \sum_{x,y} xy\mathbb{P}(X = x, Y = y) = \mathbb{P}(X = 1, Y = 1)
\end{aligned}$$

Note that the previous system consists of four linearly independent equations in four unknowns, namely the four combinations of X and Y to define $\mathbb{P}(X, Y)$. Therefore, the system defines a unique probability distribution, which in this case is given by

$$\mathbb{P}(0,0) = 0 \qquad \mathbb{P}(0,1) = 1 - p \qquad \mathbb{P}(1,0) = p \qquad \mathbb{P}(1,1) = 0$$

4.1 Reconstruction of Univariate Probability Distributions

It has been shown in [MSBK22], that for a discrete random variable with finite support, finitely many moments *uniquely* define the distribution of the random variable. In this section, we will show how to extract the distribution information from the moments using linear algebra.

More formally, let X be a random variable with distribution μ and finite support $S = \{x_1, x_2, \dots, x_m\}$. Then, μ can be recovered from the first $m-1$ moments, since these specify $m-1$ linearly independent equations $\mathbb{E}\left[X^k\right] = \sum_{i=1}^m x_i^k \mathbb{P}(X = x_i)$ over $\mathbb{P}(X)$, where $1 \leq k < m$. In combination with the second axiom of probability theory, that is $\sum_{i=1}^m \mathbb{P}(X = x_i) = 1$, this results in m linearly independent equations that completely specify $\mathbb{P}(X)$ over all points in S.

When writing $\mathbb{E}[X^0] = 1$ in the last equation, we obtain a common formulation for all equations of the form

$$\mathbb{E}\left[X^k\right] = \sum_{i=1}^{m} x_i^k \; \mathbb{P}(X = x_i), \text{ where } 0 \leq k < m.$$

Since the support of X is finite by assumption, we can write out the previous equation as a vector multiplication of the form

$$\mathbb{E}\left[X^k\right] = \begin{bmatrix} \mathbb{P}(X = x_1) & \mathbb{P}(X = x_2) & \dots & \mathbb{P}(X = x_m) \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \\ \vdots \\ x_m^k \end{bmatrix}$$

Now note, that the vector of probabilities is the same for all moments and equations. Therefore, we can write the entire defining system of m equations as follows, where $\mathbf{M}$ is the *moment vector*, $\mathbf{P}$ the *probability vector* and $\mathbf{A}$ is the *support matrix*.

$$\underbrace{\begin{bmatrix}\mathbb{E}[X^0] & \dots & \mathbb{E}[X^{m-1}]\end{bmatrix}}_{\mathbf{M}} = \underbrace{\begin{bmatrix}\mathbb{P}(X = x_1) & \dots & \mathbb{P}(X = x_m)\end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix}1 & x_1 & x_1^2 & \dots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{m-1} \\ \vdots & & & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{m-1}\end{bmatrix}}_{\mathbf{A}} \tag{4.1}$$

By utilizing the formulation (4.1), it is easy to see that the distribution μ of X is completely characterized by the vector $\mathbf{P}$. Hence, the problem of obtaining μ is equivalent to solving the matrix equation $\mathbf{M} = \mathbf{P} \cdot \mathbf{A}$ for $\mathbf{P}$.

Now, note that the matrix $\mathbf{A}$ has a special shape, and this type of matrix is known as a *Vandermonde matrix*. It is well known, that a Vandermonde matrix is invertible if and only if all entries x_i of the second column are pairwise different. Since the support of our distribution is a set, this certainly holds and hence we conclude that (a) the m equations are indeed linearly independent and (b) the equation $\mathbf{M} = \mathbf{P} \cdot \mathbf{A}$ for $\mathbf{P}$ can always be solved by inversion of $\mathbf{A}$.

We summarize this section by the following lemma and illustrate it with a brief example.

Lemma 4.1. *Let X be a random variable with finite support set S, where $|S| = m$. Then the distribution of X is* uniquely *defined by the first $m-1$ moments and can be computed as*

$$\mathbf{P} = \mathbf{M} \cdot \mathbf{A}^{-1}$$

with $\mathbf{P}$, $\mathbf{M}$ and $\mathbf{A}$ as in (4.1).

Example 8. Let X be a random variable with support $S = \{-2, -1, 0\}$ and that the first two moments are given as $\mathbb{E}[X] = -0.9$ and $\mathbb{E}[X^2] = 1.1$. Then we can write the defining equation as in (4.1) as

$$\begin{bmatrix}\mathbb{E}[X^0] & \mathbb{E}[X^1] & \mathbb{E}[X^2]\end{bmatrix} = \begin{bmatrix}\mathbb{P}(X = -2) & \mathbb{P}(X = -1) & \mathbb{P}(X = 0)\end{bmatrix} \begin{bmatrix}1 & -2 & (-2)^2 \\ 1 & -1 & (-1)^2 \\ 1 & 0 & 0\end{bmatrix}$$

Using elementary linear algebra, we compute the inverse of $\mathbf{A}$ as

$$\mathbf{A}^{-1} = \begin{bmatrix}1 & -2 & (-2)^2 \\ 1 & -1 & (-1)^2 \\ 1 & 0 & 0\end{bmatrix}^{-1} = \begin{bmatrix}0 & 0 & 1 \\ 0.5 & -2 & 1.5 \\ 0.5 & -1 & 0.5\end{bmatrix}$$

Now it is an easy task to specify the distribution of X by computing $\mathbf{P}$:

$$\mathbf{P} = \mathbf{M} \cdot \mathbf{A}^{-1} = \begin{bmatrix}1 & -0.9 & 1.1\end{bmatrix} \begin{bmatrix}0 & 0 & 1 \\ 0.5 & -2 & 1.5 \\ 0.5 & -1 & 0.5\end{bmatrix} = \begin{bmatrix}0.1 & 0.7 & 0.2\end{bmatrix}$$

4.2 Reconstruction of Joint Probability Distributions

Encouraged by the previous section, we now seek to generalize Lemma 4.1 to multivariate distributions over multiple random variables. We recall the ultimate goal to reconstruct the stochastic dependence between random variables that allows us to compute the joint probability distribution and perform inference over probabilistic programs, even if we are only given access to a finite number of moments, e.g., obtained by the approach in [MSBK22].

In the following, we show that the *joint distribution* over multiple variables is *uniquely* characterized by the *mixed moments*. Further, in an elegant generalization of Lemma 4.1, we show that the support matrix $\mathbf{A}$ in the multivariate case can be computed by the Kronecker product of the univariate support matrices (Theorem 4.2). To get there, we first consider the bivariate case, as it will show that this is the crucial generalization sufficient for generalizing to an arbitrary number of variables.

Assume two random variables X and Y with finite support $S_X = \{x_1, \ldots, x_m\}$ and $S_Y = \{y_1, \ldots, y_n\}$ that are jointly distributed with a distribution μ that is characterized by the probability measure $\mathbb{P}(X, Y)$. In a natural generalization, we now assume that we are given access to an oracle providing us with the first $n \times m$ *mixed moments* $\mathbb{E}\left[X^k Y^\ell\right]$, where $0 \leq k < m$ and $0 \leq \ell < n$.

It will be convenient to write the vectors $\mathbf{M}$ and $\mathbf{P}$ both as lexicographically ordered with $X < Y$. For ease of presentation, we introduce the subvectors $\mathbf{M_{X^k}}$ of the moment vector.

$$\begin{aligned}
\mathbf{M_{X^k}} &\coloneqq \begin{bmatrix} \mathbb{E}\left[X^k Y^0\right] & \mathbb{E}\left[X^k Y^1\right] & \ldots & \mathbb{E}\left[X^k Y^{n-1}\right] \end{bmatrix} \\
\mathbf{M} &\coloneqq \begin{bmatrix} \mathbf{M_{X^0}} & \mathbf{M_{X^1}} & \ldots & \mathbf{M_{X^{m-1}}} \end{bmatrix} \\
\mathbf{P} &\coloneqq \begin{bmatrix} \mathbb{P}(X = x_1, Y = y_1) & \mathbb{P}(X = x_1, Y = y_2) & \ldots & \mathbb{P}(X = x_m, Y = y_n) \end{bmatrix}
\end{aligned}$$

When considering the defining equation for the mixed moments

$$\mathbb{E}\left[X^k Y^\ell\right] = \sum_x \sum_y x^k y^\ell \mathbb{P}(X = x, Y = y),$$

we observe that we can unpack this computation into a matrix multiplication by using a coefficient matrix that contains all combinations of x^k and y^l. Using this insight, we write the defining equation for the subvector $\mathbf{M_{X^k}}$ in matrix form, similar to (4.1) in the univariate case.

$$\mathbf{M}_{\mathbf{X}^{\mathbf{k}}} = \mathbf{P} \cdot \begin{bmatrix} \boxed{\begin{matrix} x_1^k y_1^0 & x_1^k y_1^1 & x_1^k y_1^2 & \dots & x_1^k y_1^{n-1} \\ x_1^k y_2^0 & x_1^k y_2^1 & x_1^k y_2^2 & \dots & x_1^k y_2^{n-1} \\ \vdots & & & \ddots & \vdots \\ x_1^k y_n^0 & x_1^k y_n^1 & x_1^k y_n^2 & \dots & x_1^k y_n^{n-1} \end{matrix}} \\ \boxed{\begin{matrix} x_2^k y_1^0 & x_2^k y_1^1 & x_2^k y_1^2 & \dots & x_2^k y_1^{n-1} \\ x_2^k y_2^0 & x_2^k y_2^1 & x_2^k y_2^2 & \dots & x_2^k y_2^{n-1} \\ \vdots & & & \ddots & \vdots \\ x_2^k y_n^0 & x_2^k y_n^1 & x_2^k y_n^2 & \dots & x_2^k y_n^{n-1} \end{matrix}} \\ \begin{matrix} \vdots & \qquad & \qquad & \ddots & \vdots \end{matrix} \\ \boxed{\begin{matrix} x_m^k y_1^0 & x_m^k y_1^1 & x_m^k y_1^2 & \dots & x_m^k y_1^{n-1} \\ x_m^k y_2^0 & x_m^k y_2^1 & x_m^k y_2^2 & \dots & x_m^k y_2^{n-1} \\ \vdots & & & \ddots & \vdots \\ x_m^k y_n^0 & x_m^k y_n^1 & x_m^k y_n^2 & \dots & x_m^k y_n^{n-1} \end{matrix}} \end{bmatrix} \tag{4.2}$$

Now note, that each block of the support matrix in (4.2) is of the form $x_i^k \cdot \mathbf{A_Y}$, where $\mathbf{A_Y}$ is the univariate support matrix of Y. Therefore, we obtain

$$\mathbf{M}_{\mathbf{X}^{\mathbf{k}}} = \mathbf{P} \cdot \begin{bmatrix} x_1^k \mathbf{A_Y} \\ x_2^k \mathbf{A_Y} \\ \vdots \\ x_m^k \mathbf{A_Y} \end{bmatrix} \tag{4.3}$$

By replacing the subvectors in the moment vector $\mathbf{M}$ using (4.3), we write $\mathbf{M}$ as

$$\mathbf{M} = \mathbf{P} \cdot \begin{bmatrix} x_1^0 \mathbf{A_Y} & x_1^1 \mathbf{A_Y} & \dots & x_1^{m-1} \mathbf{A_Y} \\ x_2^0 \mathbf{A_Y} & x_2^1 \mathbf{A_Y} & \dots & x_2^{m-1} \mathbf{A_Y} \\ \vdots & & & \\ x_m^0 \mathbf{A_Y} & x_m^1 \mathbf{A_Y} & \dots & x_m^{m-1} \mathbf{A_Y} \end{bmatrix} \tag{4.4}$$

The support matrix of (4.4) looks surprisingly similar to the Vandermonde matrix of the univariate case (4.1), only that each entry of the matrix is matrix-valued and multiplied by $\mathbf{A_Y}$. This special combination of two matrices is known in literature as the *Kronecker product* and is usually denoted as $\otimes$. Using this notation, we can concisely express the generalization to two variables as

$$\mathbf{M} = \mathbf{P} \cdot \underbrace{(\mathbf{A_X} \otimes \mathbf{A_Y})}_{\mathbf{A}} \tag{4.5}$$

It is left to show that equation (4.5) has a unique solution. To prove this, we note that the inverse $(\mathbf{A} \otimes \mathbf{B})^{-1}$ exists if, and only if, both $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ exist. Then the inverse satisfies the relation

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1}.$$

As already observed in the previous section, the matrices $\mathbf{A_X}$ and $\mathbf{A_Y}$ are Vandermonde matrices and therefore invertible. From this we conclude that the inverse $\mathbf{A}^{-1}$ exists and uniquely specifies the joint distribution of X and Y, which can be computed as

$$\mathbf{P} = \mathbf{M} \cdot \mathbf{A}^{-1} = \mathbf{M} \cdot \left(\mathbf{A_X}^{-1} \otimes \mathbf{A_Y}^{-1}\right).$$

Multivariate Distributions. Generalizing these observations to the multivariate case can be done by induction, where the base case corresponds to the univariate case in Lemma 4.1 and the induction step is analogous to the bivariate case. By those means, we arrive at the concluding theorem of this section.

Theorem 4.2. *Let $X_1, X_2, \dots, X_n$ be a collection of random variables, where the support of each X_i is given by some finite set. Then the joint distribution of those random variables is* uniquely *defined by their mixed moments and can be computed as*

$$\mathbf{P} = \mathbf{M} \cdot \mathbf{A}^{-1} = \mathbf{M} \cdot \left(\mathbf{A_{X_1}}^{-1} \otimes \mathbf{A_{X_2}}^{-1} \otimes \dots \otimes \mathbf{A_{X_n}}^{-1}\right)$$

with $\mathbf{A_{X_i}}$ as in (4.1) *and the vectors* $\mathbf{P}$ *and* $\mathbf{M}$ *lexicographically ordered with $X_1 < X_2 < \dots < X_n$.*

By Theorem 4.2, we conclude that we can *always* and *fully automatically* extract the joint probability distribution of multiple random variables with finite support, when given the mixed moments. As stated before, one possible approach to obtain these moments is presented in [BKS19, MSBK22]. By the method presented in this chapter, we hence enable the extraction of the joint probability distribution for *all programs in the programming model* of [MSBK22], as long as the variables of interest are over finite support. We implemented the methodology presented here in the tool Polar and the so obtained distribution can then be used to perform inference on this loop class, which complements the class of loops presented in Chapter 3.

CHAPTER 5

Analysis of Infinite-State Loops

After analyzing the behavior of programs with finite state-spaces in Chapter 3, we turn our attention to infinite-state programs, i.e., programs that cannot be represented by a finite-state Markov chain as in Definition 8. The possibly simplest infinite-state probabilistic program is an unbounded, symmetric, one-dimensional random walk, similar to Example 2 and as shown in the following program:

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

Despite its seeming simplicity, even this program cannot be represented using finitely many states, which renders the task of program analysis much more challenging, as one need to analyze potentially (un)countable Markov chains as in Definitions 6-7. The vast majority of existing approaches is restricted to statically bounded loops, that is, loops where the number of loop iterations can be bounded a priori, see e.g., [GMV16, HVM20, SCG13, CDM14, SRM21]. However, in general there is no upper bound on the number of loop iterations, such as in the preceding illustrating example, where the number of loop iterations is an unbounded random variable. We will investigate the properties of such potentially unbounded loops in the following sections.

Here we will write $f_X(t)$ to denote both the probability mass and density function of some *random variable* X and, to avoid distinguishing the discrete and continuous case, refer to both as *(probability) density function*. The probability distribution of the random variable will be denoted as $F_X(t)$. In Chapter 2 we noted that the execution of a probabilistic program can be represented by a stochastic process $\langle \Phi_n \rangle$ in the run space. Therefore, the value of some *program variable* x after a specific number of n steps is a random variable and will be denoted as x_n.

It is well known, that for two *independent* random variables X and Y, the density of their sum can be characterized by the convolution of their densities, in symbols

$$f_{X+Y}(t) = (f_X * f_Y)(t)$$

In case of two discrete random variables X and Y, convolution is defined as

$$(f_X * f_Y)(t) := \sum_{m=-\infty}^{\infty} f_X(m) f_Y(t-m)$$

Similarly, for continuous random variables, the operator is defined as

$$(f_X * f_Y)(t) := \int_{-\infty}^{\infty} f_X(\tau) f_Y(t-\tau) d\tau$$

In case of repeated convolution, we introduce the following shorthand notation.

$$\left(\mathop{\Huge *}_{i=0}^{k} f_i \right)(t) := (f_0 * \dots * f_k)(t)$$

When we refer to a random variable with $\textsc{Categorical}(p_1, \dots, p_m)$ distribution, where $\sum p_i = 1$, then we mean a distribution that is characterized by $\mathbb{P}(X = i) = p_i$ for all $1 \leq i \leq m$.

The outline of this section is briefly summarized as follows. We first restrict ourselves to potentially unbounded for-loops (Section 5.1) and show under which conditions the probability density functions of such loops can be characterized (Lemma 5.1). Based on this result, we translate for-loops into equivalent, loop-free programs. We study special classes of such loops, improving the state-of-the-art in what is computable for unbounded loops (Sections 5.1.1-5.1.2). In particular, we show that solvable loops [Kov08] and constant update loops [GGH19] can be translated into equivalent loop-free fragments for the purpose of distribution recovery.

We next relax the setting of Section 5.1 and in Section 5.2, instead of fully recovering distributions, we provide a method to compute *moments* of program variables (Lemma 5.2). Finally, in Section 5.3, we introduce a type system (Figure 5.1) for probabilistic programs which is based on closure properties of probability distributions and is able to identify the distribution of random variables in unbounded while-loops.

5.1 Analysis of Potentially Unbounded For-Loops

In this section, we focus on potentially unbounded for-loops. These are loops that increment a loop counter by one, starting from some lower bound, most commonly zero or one, until it reaches the value held by some random variable d.

We recall that our ultimate goal is to enable inference for probabilistic programs, which requires evaluating the likelihood of the observed evidence and hence the probability density of the outcome. However, as probabilistic programs are strictly more general than deterministic programs, any hope of a full characterization is futile, due to the undecidability of the Halting problem. Therefore, it will be necessary to restrict our initially presented programming model from Figure 2.1 to a tractable fragment, which must be powerful enough to model useful programs and remain manageable at the same time.

In the programming model for this section, the loop body may contain probabilistic and deterministic variable updates and a single variable that accumulates values over all iterations. The increments in each iteration may follow an arbitrary distribution and are left unrestricted for now. A generic template for such a program is shown in Figure 5.1.

```
for $i$ in $1 \dots d$ do
    $u_1, u_2, \dots, u_m \leftarrow \ell_1, \ell_2, \dots, \ell_m$        ▷ Here, all $\ell_i$ are random values
    $inc \leftarrow u_1\ [p_1]\ u_2\ [p_2] \dots [p_{m-1}]\ u_m\ [p_m]$
    $s \leftarrow s + inc$
end for
```

Figure 5.1: A for-loop with probabilistic updates.

In this general form, each iteration of the loop first generates values for the variables $u_1, \dots, u_m$, then probabilistically picks one of them and assigns it to the random variable *inc*. Finally, the accumulator variable s is incremented by the value in *inc*. Clearly, the result of the computation s after termination is solely dependent on d and the increments u_i. Hence, knowledge of the probability densities of u_i completely characterizes the distribution of s. In the remainder of this section, we will formalize this idea and examine the conditions under which we can efficiently compute the resulting density of s.

In general, the distribution of the variables need not be the same across all iterations and to denote this, we index the density of $u_i, 1 \leq i \leq m$ in iteration n as $f_{i,n}$. The increment of variable s in iteration n is specified by the random variable inc_n. This variable is constructed as a *mixture* of the previously generated random variables, and its distribution is given by a convex combination of the mixture components u_i, in symbols,

$$f_{inc_n}(x) = \sum_{i=0}^{m} p_i \cdot f_{u_{i,n}}(x).$$

Most generally, the increment inc_n in some iteration n may be stochastically dependent on the value inc_{n-1} in the previous iteration. This renders an exact analysis of the sum s prohibitively difficult. Therefore, we focus on programs where the *increment in each iteration is independent of the increment in the previous iteration.*

Under this assumption, we can utilize the fact that for two independent random variables, the density of their sum is given by the convolution of their individual densities. This allows us to write down an explicit expression for the density of variable s after n loop iterations:

$$f_{s_n}(t) = (f_{inc_1} * f_{inc_2} * \ldots * f_{inc_n})(t) = \mathop{\Huge *}_{i=1}^{n} f_{inc_i}(t)$$

As the density inc_n is a mixture of multiple distributions, we can also write:

$$f_{s_n}(t) = \mathop{\Huge *}_{i=1}^{n} \sum_{j=1}^{m} p_j \cdot f_{u_{j,i}}(t)$$

Under the mild preceding assumption, we arrived at this first, general result characterizing the probability density of the result. The complexity of the resulting computation will be highly dependent on the number of mixture components and the specific probability distributions.

To provide more specific and detailed results, we will need to strengthen our assumptions. One reasonable assumption is that the mixture components u_i are not only independent, but also identically distributed in each loop iteration. Surprisingly, this assumption is sufficient to replace the loop by an *equivalent, loop-free* program, which may be analyzed more efficiently. The following lemma now asserts the previous claim, by showing that we can "exchange" the summation and convolution of the individual mixture components, as the order of the summations does not matter. The resulting distribution is then a (dependent) mixture of convolutions.

Lemma 5.1. *If the mixture components are independent and identically distributed, i.e., for all $1 \leq i \leq m$ and all iterations n, it holds that $f_{u_{i,n}} = f_{u_i}$, then it holds that*

$$f_{s_n}(t) = \mathop{\Huge *}_{j=1}^{m} \mathop{\Huge *}_{i=1}^{h_j} f_{u_j}(t), \ \ \textit{where } (h_1, \ldots h_m) \sim \textsc{Multinomial}(n; p_1, \ldots, p_n).$$

Equivalently, the multinomial distribution can be made implicit as

$$f_{s_n}(t) = \sum_{h_1+\cdots+h_m=n} \binom{n}{h_1, \ldots, h_m} p_1^{h_1} \ldots p_m^{h_m} \cdot \mathop{\Huge *}_{j=1}^{m} \mathop{\Huge *}_{i=1}^{h_j} f_{u_j}(t).$$

Proof. For the proof, we replace the probabilistic choice with a new random variable J, that is assigned an index from 1 to m in each iteration. Any potential increment $u_{j,i}$ is only added to the accumulator variable in case $J_i = j$ and masked out otherwise. This allows us to exchange the sums and replace the indicator variables, by using the fact that the sum over a categorical distribution has a multinomial distribution.

$$\begin{aligned}
s_n &= \sum_{i=1}^{n} inc_i \\
&= \sum_{i=1}^{n} \sum_{j=1}^{m} u_{j,i} \, \mathbb{I}(J_i = j) \qquad J_i \sim \textsc{Categorical}(p_1, \ldots, p_n) \\
&= \sum_{j=1}^{m} \left(\sum_{i=1}^{n} u_{j,i} \, \mathbb{I}(J_i = j) \right) \\
&= \sum_{j=1}^{m} \left(\sum_{i=1}^{h_s} u_j \right) \qquad (h_1, \ldots h_m) \sim \textsc{Multinomial}(n; p_1, \ldots, p_n)
\end{aligned}$$

□

Lemma 5.1 is quite powerful, as there exist a plethora of relationships among probability distributions, especially regarding convolution. For example, we have the following closure properties of probability distributions.

$$\begin{aligned}
&\sum_{i=1}^{n} \textsc{Bernoulli}(p) = \textsc{Binomial}(n; p) \\
&\sum_{i=1}^{n} \textsc{Geometric}(p) = \textsc{NegativeBinomial}(n; p) \\
&\sum_{i=1}^{n} \textsc{Poisson}(\lambda_i) = \textsc{Poisson}\left(\sum_{i=1}^{n} \lambda_i \right)
\end{aligned}$$

Equipped with Lemma 5.1 and the preceding relations, we are ready to eliminate certain loops from programs and replace them with loop-free program fragments, as illustrated in the following example.

Example 9. Consider the concrete program in Figure 5.2a that follows the scheme presented in Figure 5.1. By Lemma 5.1 and the listed closure properties, the program in Figure 5.2a is equivalent to the loop-free program in Figure 5.2b.

To conclude, this section presented an approach that is capable of replacing a class of loops by an *equivalent, loop-free* program, even though the original loop is potentially unbounded and hence cannot be analyzing by unrolling. In the following two sections, we will instantiate and further specialize the previous result for more specific classes of programs known in literature.

```
for $i$ in $1 \dots d$ do
    $u_1, u_2, u_3 \leftarrow$ Bernoulli$(q_1)$, Geometric$(q_2)$, Poisson$(\lambda_1)$
    $inc \leftarrow u_1\ [p_1]\ u_2\ [p_2]\ u_3[p_3]$
    $s \leftarrow s + inc$
end for
```

(a) A concrete for-loop with probabilistic updates.

```
$(h_1, h_2, h_3) \leftarrow$ Multinomial$(d; p_1; p_2; p_3)$
$x \leftarrow x_0 +$ Binomial$(h_1; q_1)$ + NegativeBinomial$(h_2; q_2)$ + Poisson$(h_3\lambda_1)$
```

(b) An equivalent loop-free program.

Figure 5.2: Two equivalent probabilistic programs.

5.1.1 Special Case: Solvable Loops

If the loop body does not utilize probabilistic choice and hence is purely deterministic, then for some loop classes, such as single-path programs with affine[1] updates, it is possible to compute a closed-form solution of the values of program variables in terms of the loop counter [Kov08]. This simplifies the task of program analysis even further, since we can replace the loop with an assignment of the closed-form solution evaluated at iteration d.

5.1.2 Special Case: Constant Updates

Another notable subclass of the generic loop template in Figure 5.1 is the class of constant-update loops, where the problem of termination has been studied in existing work [GGH19]. Here, all mixture components u_i are assigned constants c_i and hence distributed according to a Dirac distribution, in symbols $f_{u_i}(t) = \delta(t - c_i)$. It can be shown, that the Dirac distribution is closed under convolution and the resulting distribution accumulates the displacements, in symbols

$$\mathop{\Huge\ast}_{i=1}^{k} \delta(t - c_i) = \delta\left(t - \sum_{i=1}^{k} c_i\right). \tag{5.1}$$

Now, from Lemma 5.1, we conclude that in the constant-update case, the program in Figure 5.1 can be written as the following, loop-free program.

```
$(h_1, \dots, h_m) \leftarrow$ Multinomial$(d; p_1; \dots; p_m)$
$s \leftarrow s_0 + h_1 c_1 + \dots + h_m c_m$
```

To write down the closed-form probability density f_{s_n}, we start from Lemma 5.1 and

[1] An affine expression is a polynomial of degree one.

repeatedly apply (5.1) as follows:

$$
\begin{aligned}
f_{s_n}(t) &= \sum_{h_1+\dots+h_m=n} \binom{n}{h_1,\dots,h_m} p_1^{h_1}\dots p_m^{h_m} \cdot \mathop{\Large\ast}_{j=1}^{m} \mathop{\Large\ast}_{i=1}^{h_j} \delta(t-c_j) \\
&= \sum_{h_1+\dots+h_m=n} \binom{n}{h_1,\dots,h_m} p_1^{h_1}\dots p_m^{h_m} \cdot \mathop{\Large\ast}_{j=1}^{m} \delta(t-h_j c_j) \\
&= \sum_{h_1+\dots+h_m=n} \binom{n}{h_1,\dots,h_m} p_1^{h_1}\dots p_m^{h_m} \cdot \delta\left(t-\sum_{j=1}^{m} h_j c_j\right) \\
&= \sum_{\substack{h_1+\ldots+h_m=n\\ c_1h_1+\ldots+c_mh_m=t}} \binom{n}{h_1,\dots,h_m} p_1^{h_1}\dots p_m^{h_m}
\end{aligned}
$$

For general constants $c_1,\dots,c_m$, one needs to consider all possible vectors $(h_1,\dots,h_m) \in \mathbb{N}_0^m$ such that $\sum_{i=0}^{m} h_i c_i = t$. This is computationally hard, as it is equivalent to partitioning t into m fixed integers.

However, the problem is feasible if the increments are all pairwise relative prime, i.e., $\forall i,j : i \neq j \Rightarrow gcd(c_i,c_j) = 1$. Then, the prime decomposition of t yields a unique way of writing t as a linear combination of the increments c_i.

In the case that $m = 2$, the multinomial distribution collapses into a binomial distribution where $h_1 \sim \textsc{Binomial}(d;p_1)$, and $h_2 = d - h_1$. Therefore, in this case we get the even simpler loop-free program

```
$h_1 \leftarrow \textsc{Binomial}(d;p_1)$
$s \leftarrow s_0 + h_1c_1 + (n-h_1)c_2 \equiv s_0 + h_1(c_1-c_2) + c_2d$
```

5.2 Moment-Generating Function Based Loop Analysis

Even though the investigations presented in Section 5.1 may considerably reduce the effort and computing power necessary to analyze a program, the level of precision provided may not be needed in all cases. Instead of analyzing the full distribution, it is often sufficient to compute a *finite set of moments* to characterize the main properties of the distribution.

This section provides an elegant method based on *generating functions* to infer the moments of potentially unbounded for-loops *if the distribution of the loop bound is known.*

Let X be a discrete random variable, then the *probability-generating function (PGF)* $G_X(z)$ is defined as z-transform of the density function:

$$G_X(z) := \mathbb{E}\left[z^X\right] = \sum_x f_X(x) \cdot z^x$$

Similarly, for a real-valued random variable, the *moment-generating function (MGF)* $M_X(t)$ corresponds to the Laplace transform of the density function:

$$M_X(t) := \mathbb{E}\left[e^{tX}\right] = \int_{-\infty}^{\infty} f_X(x) \cdot e^{tx} dx$$

As witnessed by the following lemma, the summation of a random number of independent and identically distributed random variables is closely related to their generating functions.

Lemma 5.2. *Let $\langle X_i \rangle_{i \in \mathbb{N}}$ be a sequence of independent and identically distributed random variables. Further, let N be another random variables over $\mathbb{N}$ and let Z be the sum of the first N terms, in symbols $Z = X_1 + X_2 + \ldots + X_N$. Then the MGF of Z is given as*

$$M_Z(t) = G_N(M_X(t))$$

Proof. This can be shown by first observing that in case the upper bound N is known, then $\mathbb{E}[e^{tZ} \mid N] = M_X(t)^N$, since summation of independent random variables is equivalent to the multiplication of their moment-generating functions. Then, by the law of total expectation, we arrive at the relation

$$M_Z(t) = \mathbb{E}\left[e^{tZ}\right] = \mathbb{E}\left[\mathbb{E}\left[e^{tZ} \mid N\right]\right] = \mathbb{E}\left[M_X(t)^N\right] = G_N(M_X(t)).$$

□

Now if we assume again that the increment in the loop body of Figure 5.1 is distributed independently and identically in each iteration, then the following relation holds by Lemma 5.2:

$$M_s(t) = G_d(M_{inc}(t))$$

Now if both, the PGF of the upper-bound d and the MGFs of the increments u_i exists, then we can use the fact, that for a mixture distribution, the MGF is the convex combination of the component MGFs to obtain:

$$M_s(t) = G_d\left(\sum_{i=1}^{m} p_i M_{u_i}(t)\right)$$

Now, by the properties of the moment generating function, we can extract arbitrary moments of s by the formula

$$\mathbb{E}\left[s^k\right] = \left.\frac{d^k M_s(t)}{dt^k}\right|_{t=0}$$

We conclude, that if the density of the upper bound d is explicitly known and the increments are independently distributed, then this method provides a powerful way of analyzing the moments of a potentially unbounded loop, summarized by the following lemma

Lemma 5.3. *Let $\mathcal{P}$ be a concrete program of the type given in Figure 5.1. Then if both the PGF of the random upper bound d and the MGFs of the independent increments u_i exist, then it is possible to compute arbitrary moments $\mathbb{E}\left[s^k\right]$ of the random variable s.*

In the following example, we illustrate the power of the approach, by showing that this methodology may even be used to analyze unbounded while-loops.

Example 10. In Figure 5.3, we have a loop that closely follows the scheme presented in Figure 5.1, with the notable exception that the loop terminates only when some parameterized coin-flip shows heads.

```
$s \leftarrow 0$
while $\textsc{Bernoulli}(p) = 0$ do
    $u_1, u_2, u_3 \leftarrow \textsc{Normal}(5, 2), \textsc{Exponential}(1/3), 3$
    $inc \leftarrow u_1[1/4]u_2[1/3]u_3[5/12]$
    $s \leftarrow s + inc$
end while
```

Figure 5.3: A while-loop with probabilistic updates.

It is clear that the increments are identically distributed and additionally, the number of loop iterations has a Geometric distribution, since the number of summations is equivalent to the number of times the coin lands tails.

Therefore, the moment-generating function of s is obtained by chaining the probability-generating function of a geometric distribution with parameter p and the convex combination of the moment-generating functions of the increments. When looking up the individual transformations in a table and after carrying out the necessary computations, one arrives at the moment-generating function, symbolic in p:

$$M_s(t) = \frac{(36t-12)p}{36t - 4p - 8 + 15e^{3t}(p-1)(t-1/3) + e^{2t^2+5t}(3 - 9t + 9pt - 3p)}$$

We stress that the previous equation *uniquely* identifies the distribution of s. This can be used to *automatically* compute arbitrary moments by simple differentiation and evaluation, e.g., for the first two moments:

$$\mathbb{E}[s] = \left.\frac{dM_s(t)}{dt}\right|_{t=0} = \frac{7-7p}{2p}$$

$$\mathbb{E}[s^2] = \left.\frac{d^2M_s(t)}{dt^2}\right|_{t=0} = \frac{15p^2-64p+49}{2p^2}$$

5.3 A Type System for Distribution Identification

In Section 5.1, we investigated potentially unbounded for-loops, where we utilized closure properties of distributions to provide a corresponding, loop-free program. This section will focus on identification of distributions, with the purpose of finding the distribution of the mixture components of Section 5.1, if not explicitly given. In addition, the presented approach is also able to characterize the distribution of some *unbounded while-loops*, representing forever lasting stochastic processes. Existing work investigated the extraction of moments for restricted kinds of such loops [BKS19, MSBK22], but lacks the power to provide a full characterization of the resulting distribution. One key observation is, that if the type or shape of the distribution can be inferred by other means, then a finite set of moments is sufficient to fully characterize the distribution. This is what we aim to do in this section.

As an illustrating example, consider the following program.

```
$x \leftarrow 1$
$y \leftarrow x + \textsc{Normal}(1, 4)$
while $\star$ do
    $y \leftarrow y + \textsc{Normal}(5, 2)$
end while
```

By the closure properties of the normal distribution, we can infer that y is normally distributed during each loop iteration. Given that there are plenty of closure properties among probability distributions, our aim in this section is to construct a type system that can infer the distribution of variables in such programs *in a purely syntactic way*, paving the way to fully characterize the distribution of variables.

It is important to note, that it is crucial to ensure stochastic independence of added variables, since otherwise closure properties do not hold. Another noteworthy limitation is that we only consider single-path loops, i.e., loops without branching statements and without probabilistic choice. This stems from the restriction that both constructs would allow the construction of mixture distributions, which result in increased complexity.

To make statement boundaries more visible, we will wrap statements into parenthesis and separate statements with a simple dot, e.g., $(x \leftarrow 1).(y \leftarrow 5)$. Further, all programs are either terminated by an empty statement of the form $\mathbf{0}$ or an unbounded loop. In our language, we only have assignment statements and unbounded while-loops, hence we are ready to introduce the first typing rules for statements:

$$\textsc{Empty}\frac{}{\Gamma \vdash \mathbf{0}} \qquad \textsc{Assign}\frac{\Gamma \vdash e : \mathsf{T} \qquad x : \mathsf{T}' \text{ in } \Gamma \Rightarrow \mathsf{T} = \mathsf{T}' \qquad \Gamma, x : \mathsf{T} \vdash P}{\Gamma \vdash (x \leftarrow e).P}$$

$$\textsc{Loop}\frac{\Gamma \vdash P.\mathbf{0}}{\Gamma \vdash (\textbf{while} \star \textbf{do } P \textbf{ end while})}$$

Table 5.1: Typing rules for program statements of Figure 5.1.

Next, we introduce type checking for deterministic expressions.

Some distributions, such as Normal, admit closure properties among all parameterizations, hence we can immediately give typing rules that ignore the concrete instantiation

$$\textsc{Num}\ \frac{e \in \mathbb{R}}{\Gamma \vdash e : \mathsf{Dirac}} \qquad \textsc{Arith}\ \frac{\Gamma \vdash e_1 : \mathsf{Dirac} \qquad \Gamma \vdash e_1 : \mathsf{Dirac} \qquad \bowtie \in \{+, -, \times, /\}}{\Gamma \vdash e_1 \bowtie e_2 : \mathsf{Dirac}}$$

Table 5.2: Typing rules for deterministic expressions.

of the distribution. Note that in the rules that perform addition, it is crucial that the random variable on the right-hand side is *fresh*, to ensure stochastic independence.

$$\textsc{Norm}\ \frac{e = \textsc{Normal}(\mu, \sigma^2) \quad \mu, \sigma^2 \in \mathbb{R}}{\Gamma \vdash e : \mathsf{Normal}} \qquad \textsc{Norm-Add}\ \frac{\Gamma \vdash e_1 : \mathsf{Normal} \quad \mu, \sigma^2 \in \mathbb{R}, \mathbb{R}^+}{\Gamma \vdash e_1 + \textsc{Normal}(\mu, \sigma^2) : \mathsf{Normal}}$$

$$\textsc{Poi}\ \frac{e = \textsc{Poisson}(\lambda) \qquad \lambda \in \mathbb{R}^+}{\Gamma \vdash e : \mathsf{Poisson}} \qquad \textsc{Poi-Add}\ \frac{\Gamma \vdash e_1 : \textsc{Poisson} \qquad \lambda \in \mathbb{R}^+}{\Gamma \vdash e_1 + \textsc{Poisson}(\lambda) : \mathsf{Poisson}}$$

$$\textsc{Cau}\ \frac{e = \textsc{Cauchy}(a, \gamma) \quad a, \gamma \in \mathbb{R}}{\Gamma \vdash e : \mathsf{Cauchy}} \qquad \textsc{Cau-Add}\ \frac{\Gamma \vdash e_1 : \mathsf{Cauchy} \quad a, \gamma \in \mathbb{R}, \mathbb{R}^+}{\Gamma \vdash e_1 + \textsc{Cauchy}(\mu, \sigma^2) : \mathsf{Cauchy}}$$

Table 5.3: Typing rules for parameter-agnostic distribution expressions of Figure 5.1.

Other distributions, such as Binomial or NegativeBinomial are only closed under addition if certain parameters are identical. Therefore, we parameterize the type of the distribution with the respective parameter as done in the following rules:

$$\textsc{Bin}\ \frac{\begin{array}{c} e = \textsc{Binomial}(n, p) \\ n \in \mathbb{N}^+ \quad 0 < p < 1 \end{array}}{\Gamma \vdash e : \mathsf{Binom}_p} \qquad \textsc{Bin-Add}\ \frac{\begin{array}{c} \Gamma \vdash e_1 : \mathsf{Binom}_p \\ n \in \mathbb{N}^+ \quad p = p' \end{array}}{\Gamma \vdash e_1 + \textsc{Binomial}(n, p') : \mathsf{Binom}_p}$$

$$\textsc{NegB}\ \frac{\begin{array}{c} e = \textsc{NegativeBinomial}(n, p) \\ n \in \mathbb{N}^+ \quad 0 < p < 1 \end{array}}{\Gamma \vdash e : \mathsf{NBinom}_p} \qquad \textsc{NegB-Add}\ \frac{\begin{array}{c} \Gamma \vdash e_1 : \mathsf{NBinom}_p \\ n \in \mathbb{N}^+ \quad p = p' \end{array}}{\Gamma \vdash e_1 + \textsc{NegativeBinomial}(n, p') : \mathsf{NBinom}_p}$$

Table 5.4: Typing rules for parameter-sensitive distribution expressions of Figure 5.1.

Finally, some distributions allow closed forms for their sums, but the sums are distributed according to other distributions, e.g., $\sum_{i=1}^{n} \textsc{Bernoulli}(p) \sim \textsc{Binomial}(n, p)$. To handle this kind of "casting", we introduce the concept of sub-typing. In the context of the previous closure property, we would allow the promotion of $\mathsf{Bernoulli}_p$ to $\mathsf{Binomial}_p$.

Apart from standard type theoretic constructs, the safety and correctness of the presented type system follows directly from the closure properties among the probability distributions, as for example found in [FT13].

Lemma 5.4. *The type system presented in Table 5.1 and the subsequent typing rules of Tables 5.2-5.5 are safe and allow us to reason about probabilistic programs of the form shown in Figure 5.1.*

The type system presented above is a proof-of-concept, but nevertheless presents important concepts in how to type check probabilistic programs and their closure properties. We

$$\textsc{Refl}\,\frac{}{\mathsf{T} \leq \mathsf{T}} \qquad \textsc{Trans}\,\frac{\mathsf{T} \leq \mathsf{T}' \quad \mathsf{T}' \leq \mathsf{T}''}{\mathsf{T} \leq \mathsf{T}''}$$

$$\textsc{Subsum}\,\frac{\Gamma \vdash e : \mathsf{T}' \quad \mathsf{T}' \leq \mathsf{T}}{\Gamma \vdash e : \mathsf{T}}$$

$$\textsc{Bern}\,\frac{e = \textsc{Bernoulli}(p) \quad 0 < p < 1}{\Gamma \vdash e : \mathsf{Bernoulli}_p} \qquad \textsc{Bern-Bin}\,\frac{}{\mathsf{Bernoulli}_p \leq \mathsf{Binom}_p}$$

$$\textsc{Geo}\,\frac{e = \textsc{Geometric}(p) \quad 0 < p < 1}{\Gamma \vdash e : \mathsf{Geometric}_p} \qquad \textsc{Geo-NegB}\,\frac{}{\mathsf{Geometric}_p \leq \mathsf{NBinom}_p}$$

$$\textsc{Zero-Variance}\,\frac{}{\mathsf{Dirac} \leq \mathsf{Normal}}$$

Table 5.5: Typing rules for sub-typing.

anticipate that it is easily extended for other closure properties, and close the section by an example that illustrates the use of the type system:

Example 11. Consider the following probabilistic program.

```
$g \leftarrow \textsc{Geometric}(0.5)$
while $\star$ do
    $g \leftarrow g + \textsc{Geometric}(0.5)$
end while
```

To check if the distributions follow some 'nice' distribution in each iteration, we try to type check the program. For brevity, we denote Geometric as Geo.

$$\textsc{Assign}\,\frac{\dfrac{(5.3)}{\emptyset \vdash \textsc{Geo}(0.5) : \mathsf{NBinom}_{0.5}} \qquad \dfrac{g : \mathsf{T}' \text{ not in } \emptyset}{g : \mathsf{T}' \text{ in } \emptyset \Rightarrow T = T'} \qquad \textsc{Loop}\,\dfrac{\dfrac{(5.4)}{g : \mathsf{NBinom}_{0.5} \vdash (g \leftarrow \textsc{Geo}(0.5)).\mathbf{0}}}{g : \mathsf{NBinom}_{0.5} \vdash (\mathbf{while} \star \mathbf{do} \ldots)}}{\emptyset \vdash (g \leftarrow \textsc{Geo}(0.5)).(\mathbf{while} \star \mathbf{do} \ldots)} \tag{5.2}$$

$$\textsc{Subsum}\,\frac{\textsc{Geo}\,\dfrac{\textsc{Geo}(0.5) = \textsc{Geo}(0.5) \quad 0 < 0.5 < 1}{\emptyset \vdash \textsc{Geo}(0.5) : \mathsf{Geometric}_{0.5}} \qquad \textsc{Geo-NegB}\,\dfrac{}{\mathsf{Geometric}_{0.5} \leq \mathsf{NBinom}_{0.5}}}{\emptyset \vdash \textsc{Geo}(0.5) : \mathsf{NBinom}_{0.5}} \tag{5.3}$$

$$\textsc{Assign}\,\frac{\dfrac{(5.3)}{\emptyset \vdash \textsc{Geo}(0.5) : \mathsf{NBinom}_{0.5}} \qquad \dfrac{}{g : \mathsf{T}' \text{ in } g : \mathsf{NBinom}_{0.5} \Rightarrow \mathsf{NBinom}_{0.5} = T'} \qquad \textsc{Empty}\,\dfrac{}{g : \mathsf{NBinom}_{0.5} \vdash \mathbf{0}}}{g : \mathsf{NBinom}_{0.5} \vdash (g \leftarrow \textsc{Geo}(0.5)).\mathbf{0}} \tag{5.4}$$

From the derivation in (5.2), we conclude that g is distributed according to a negative binomial distribution where $p = 0.5$. The derivation illustrates the necessity of subtyping to promote the geometric distribution to a more general negative binomial distribution that can be used to apply further typing rules.

CHAPTER 6

Strongest Polynomial Invariant for Probabilistic Loops

The overall goal of this thesis is to analyze probabilistic loops in order to enable or simplify inference. Ultimately, loop analysis is intimately tied to the problem of abstracting the “effect” of a loop, e.g., in the form of *loop invariants*. Such invariants assert that certain relationships among program variables will hold during program execution, displaying the nature of the loop. Unfortunately, finding the right abstraction is an intrinsically hard and often undecidable problem [MS04b, HOPW19]. In addition, the notion of an invariant for probabilistic programs is not straightforward to define, due to the underlying randomness.

In this chapter, we review existing concepts for non-probabilistic programs and present a definition for probabilistic invariants, where we focus on *polynomial* invariants for programs that feature *polynomial* assignments. We will explore the existing literature to introduce the *concept of (strongest) polynomial invariants for deterministic and nondeterministic programs* (Sections 6.1-6.3), define a probabilistic equivalent and discuss similarities as well as differences between the (non-)deterministic and the probabilistic case (Section 6.4). Finally, in Section 6.5, we will show the hardness of an open problem in deterministic invariant synthesis, by relating it to a longstanding problem in mathematics.

6.1 Preliminaries

The general concept of invariants, as presented in this section, dates back to the work of Karr [Kar76] and an extension by Müller-Olm and Seidl [MS04b] who started to investigate polynomial invariants for programs with polynomial assignments and restricted subclasses. Given a program with a finite set of locations Q and variables $x_1, \ldots x_k$ over the rationals $\mathbb{Q}$, some polynomial equation $p(x_1, \ldots, x_k) = 0$ is said to be a polynomial invariant with respect to some location $q \in Q$, if all reachable configurations in location q satisfy the given equation.

In accordance, the *strongest polynomial invariant* is defined to be the (infinite) set of all polynomial invariants. Interestingly, this typically infinite set does always admit a finite representation. To see this, note that the set of invariants form a polynomial ideal I, i.e., the set I is a subset of the polynomial ring $\overline{\mathbb{Q}}[x_1, \dots, x_k]$ that satisfies the ideal condition [RK04]. This means that I is (1) closed under addition, and (2) multiplying an arbitrary ring element with an ideal element yields an ideal element, in symbols $\forall p \in \mathbb{Q}[x_1, \dots, x_k], \forall i \in I : pi \in I$.

It turns out, that all polynomial ideals can be represented by a finite number of polynomials and even admit a unique finite representation [Buc06]. Let $\mathbb{K}$ be a field, e.g., $\mathbb{Q}$, and let $f_1, \dots, f_s$ be polynomials in $\mathbb{K}[x_1, \dots, x_k]$, then we define the set $\langle f_1, \dots, f_s \rangle$ generated by the polynomials as

$$\langle f_1, \dots, f_s \rangle := \left\{ \sum_{i=1}^{s} h_i f_i : h_1, \dots, h_s \in \mathbb{K}[x_1, \dots, x_k] \right\}.$$

It is easy to show that $\langle f_1, \dots f_s \rangle$ is always an ideal of $\mathbb{K}[x_1, \dots, x_k]$ [CLO97, Chapter 1, §4, Lemma 3]. Conversely, Hilbert's basis theorem, as stated below, guarantees the existence of a finite representation, a *basis* for every polynomial ideal, and especially for the set of polynomial invariants.

Theorem 6.1 (Hilbert's Basis Theorem)**.** *Every ideal $I \subset \mathbb{K}[x_1, \dots, x_k]$ over some field $\mathbb{K}$ has a finite generating set. That is, $I = \langle f_1, \dots, f_s \rangle$ for some $f_1, \dots, f_s \in I$.*

Therefore, we directly conclude that we can represent the strongest polynomial invariant by a *finite basis*. Polynomial ideals exhibit useful properties, such as the decidability of membership testing, that is, it is decidable whether some polynomial relationship is part of the ideal, as stated by the following lemma.

Lemma 6.2 (Ideal Membership [Buc06, CLO97, Chapter 8])**.** *Given a polynomial ideal I, it is decidable whether a given polynomial f lies in I.*

There is another, equivalent formulation for the strongest polynomial invariant, preferred by some authors [HOPW19, VK23]. To present it, let us introduce the concept of an affine variety, the fundamental objects of study in algorithmic geometry.

Definition 14 (Affine Variety [CLO97, Chapter 1])**.** Let $\mathbb{K}$ be a field and let $f_1, \dots, f_s$ be polynomials in $\mathbb{K}[x_1, \dots, x_k]$. Then the *affine variety* $\mathbf{V}(f_1, \dots, f_s)$ is defined as the set of all points in $\mathbb{K}^k$, where all polynomials vanish, in symbols

$$\mathbf{V}(f_1, \dots, f_s) := \left\{ (a_1, \dots, a_k) \in \mathbb{K}^k : f_i(a_1, \dots, a_k) = 0 \text{ for all } 1 \leq i \leq s \right\}$$

Now the *affine variety* $\mathbf{V}(I)$ *of a polynomial ideal* I is defined as the set of common roots of all polynomials in the ideal and equivalently, if $I = \langle f_1, \dots, f_s \rangle$, then $\mathbf{V}(I) = \mathbf{V}(f_1, \dots, f_s)$. At the same time, $\mathbf{V}(I(S))$ is the *Zariski closure* of the set of reachable configurations S, that is, the variety of the strongest polynomial ideal as defined above. Some authors, e.g., [HOPW19, VK23], also refer to the strongest polynomial invariant as the Zariski closure $\mathbf{V}(I(S))$ of the polynomial invariant ideal.

In Section 6.5, we will show that finding the strongest polynomial invariant for a specific class of programs, namely single-path polynomial loops, is at least as hard as the Skolem problem, a long-standing, open problem in mathematics. Let us briefly introduce it.

Let $u(n), n \in \mathbb{N}_0$ be a sequence over the integers $\mathbb{Z}$. We say that the sequence is *C-finite* (of order k), if there are numbers $a_0, \dots a_{k-1} \in \mathbb{Z}$ with $a_0 \neq 0$ such that

$$u(n+k) = a_{k-1}u(n+k-1) + \ldots + a_1u(n+1) + a_0u(n) \qquad (n \geq 0)$$

The sequence $u(n)$ is uniquely defined by the coefficients $a_0, \dots, a_{k-1}$ and the initial values $u(0), \dots, u(k-1)$ [KP11]. One long-standing, open problem of surprising simplicity is related to the zeros of such a sequence:

The Skolem problem [EvdPSW03, Tao08]: Given a C-finite sequence $u(n), n \in \mathbb{N}_0$, does there exist some $m \in \mathbb{N}_0$ such that $u(m) = 0$?

After this brief introduction to the object of interest, we are ready to give an overview of the established results and extend them.

6.2 Deterministic Programs

In his foundational paper [Kar76], Karr showed that for affine[1] single-path programs, i.e., without conditional branching, the problem of finding the strongest *affine* invariant is decidable. Later, Müller-Olm and Seidl provided a simpler version of Karr's algorithm [MS04b] and Kovács [Kov08] showed that for affine single-path programs, it is possible to compute the strongest *polynomial* invariant.

After considering the case for affine single-path programs, we turn to the problem of computing invariants for polynomial single-path programs. As shown by Müller-Olm and Seidl [MS04a], it is possible to compute the set of polynomial invariants of *bounded degree* in the case of polynomial single-path programs. However, the more general task of *finding the strongest polynomial invariant for single-path programs remains unsolved up to date.*

Nevertheless, in Section 6.5, we provide a novel proof, showing that the problem of *finding the strongest polynomial invariant for single-path polynomial loops is at least as hard as the Skolem problem.* This implies, that decidability of the invariant synthesis problem

[1] An affine expression is a polynomial of degree one.

for single-path polynomial loops would solve the Skolem problem, a major unsolved problem in mathematics.

Note that until now, we solely considered single-path programs. It is easy to show, that for deterministic multi-path programs, the problem is undecidable, even finding affine invariants for affine programs, since they can simulate 2-counter-machines. In contrast to a standard Turing machine, a 2-counter-machine has access to two *non-negative, unbounded* registers which it can increment, decrement and compare to zero. Despite the restrictions, 2-counter-machines and Turing machines are equally expressive [HU69], which implies that the Halting problem for 2-counter-machines is undecidable.

The transition relation of the 2-counter-machine can be simulated by affine expressions, since a register may only be incremented or decremented. The zero-testing of the registers can be simulated either by equality or inequality conditions, since the registers are assumed to be non-negative. To see why the problem of finding the strongest affine invariant is undecidable, simulate an arbitrary 2-counter-machine using a deterministic multi-path program and introduce a new variable t, which is initially zero and set to one once the machine halts.

Note that the affine relation $t = 0$ holds if and only if the machine does not halt. Now, if we could compute the strongest affine invariant (or even test candidates of degree one), we could test whether $t = 0$ is part of the strongest affine invariant, since ideal membership is decidable by Lemma 6.2. By this reasoning, we conclude that the problem of finding the *strongest affine invariant for programs with affine assignments and affine equality/inequality conditions is undecidable.* It is immediately evident that this result also holds for polynomial invariants and programs.

However, it is possible to over-approximate the set of reachable configurations by treating conditionals as non-deterministic choice, i.e., by ignoring the condition. This approach has proven itself to be quite powerful, as shown in the following section.

6.3 Non-Deterministic Programs

When ignoring branching conditions, the resulting set of states is an over-approximation of the actually reachable states, hence the resulting set of invariants is a subset of the strongest polynomial invariant. Note that any nondeterministic program is naturally a multi-path program, hence we will refer to a nondeterministic multi-path program simply as a nondeterministic program.

It has already been noticed by Karr in his foundational paper [Kar76], that replacing conditional branching by nondeterminism considerably simplifies the problem of finding the strongest polynomial invariant. More specifically, the method presented in his work allows computing the strongest affine invariant for nondeterministic affine programs.

Later, Müller-Olm and Seidl showed that for nondeterministic polynomial programs, the problem of computing the strongest polynomial invariant of *bounded degree* is decidable, even when admitting disequality guards [MS04a]. Note, that in their publication, the authors do not allow classical if-then-else statements, but rather guarded transitions, i.e., some transitions can be blocked, but it is not possible to encode a deterministic choice. Shortly after, the same authors proved that introducing equality guards (again, for transitions), renders both the problem of invariant-testing a given polynomial and finding the strongest polynomial invariant of bounded degree undecidable [MS04b], by a reduction from Post's Correspondence Problem, which is undecidable [Pos46]. The reduction is easily adapted to inequality conditions, by noticing that for integer variables x, y the equality $x = y$ can be encoded as two inequalities $1 > x - y > -1$. These insights show an interesting discrepancy between (in-)equality and disequality.

The problem of finding unbounded polynomial invariants for affine programs has been the subject of intense research afterward. Works by Rodríguez-Carbonell & Kapur [RK07] and Kovács [Kov08] provided solutions for restricted classes of nondeterministic programs. Only recently, it has been shown that the problem is decidable for all affine programs, but undecidable for nondeterministic polynomial programs [HOPW19]. The latter proof does not rely on any guarded transitions like the undecidability proof in [MS04b]. As it turns out, the proof in [HOPW19] can be adapted to show that the problem is already undecidable when admitting only quadratic updates [VK23], therefore the realm of decidability ends in between linear and quadratic updates.

We briefly summarize decidability results for strongest invariants in Table 6.1, both for the deterministic and the nondeterministic case, including our own results from Section 6.5.

Program Model	Str. Affine Inv.		Str. Poly. Inv.	
Single-Path Deterministic, Affine	✓	[Kar76]	✓	[Kov08]
Single-Path Deterministic, Poly.	✓	[MS04a]	SKOLEM-hard	Theorem 6.9
Multi-Path Deterministic, Affine	✗ (Halting Problem)			
Non-Deterministic, Affine	✓	[Kar76]	✓	[HOPW19]
Non-Deterministic, Polynomial	✓	[MS04a]	✗	[HOPW19]

Table 6.1: Decidability results for strongest invariants. The symbol '✓' denotes decidable problems, while '✗' denotes undecidable problems.

6.4 Probabilistic Programs

After introducing related concepts for non-probabilistic programming models, we will now give a brief overview of the state-of-the-art in the probabilistic world. The term "invariant" is vastly overloaded with different meanings in the probabilistic context, and we start by giving an overview of concepts that can be found in the literature.

Existing Concepts. Since probabilistic reasoning is intrinsically uncertain, it has been noted very early that for efficient computations it is necessary to reason about expected values, rather than distributions [Koz83, MM05]. These approaches reason about the expected value of *fixed expressions*, with respect to a given probabilistic program, in a weakest-precondition style of reasoning. In this context, an invariant may be best described by an expression for which the expected value does not change during a loop iteration, i.e., a fixpoint of the loop-characteristic function [GKM13]. However, finding these fixed points is a hard problem and for verification purposes, it may be sufficient to bound the result. Such upper/lower bounds are in this context called super/sub-invariants [KKM19, GKM13].

Another style of reasoning, as in [CNZ17], considers an invariant as an expression that is violated with a bounded probability, i.e., the invariant may not hold in some traces, but the probability that a violation occurs does not exceed some bound. Yet other authors consider invariants as expressions whose expected value *remains positive* throughout the execution of the program [CS14] or utilize martingale expressions [BEFH16].

Note that the different meanings presented so far are not suited to define the notion of the strongest invariant for probabilistic programs, as these invariants do not form ideals or are relative to some expression, while invariants as presented in the previous sections form ideals and relate *all* variables.

Another approach that exists in the literature is to reason about moments of program variables and compute closed form solutions for the moments of program variables, of bounded degree [BKS19]. This approach can be combined with the ideas presented in [Kov08] to compute all polynomial relationships between the moments of program variables. This line of thought is the one most similar to the one presented in the previous sections, and will be used to define a probabilistic equivalent of (strongest) invariants in the following sections.

(Strongest) Probabilistic Polynomial Invariants. As in the preceding sections, all variables $x_1, \dots x_k$ are assumed to be over the rationals $\mathbb{Q}$, our base field $\mathbb{K}$. To precisely define the semantics of the expected value operator as used here, we denote variables with an additional subscript when necessary, i.e., $\mathbb{E}[x_{1,n}]$ denotes the expected value of x_1 at the nth loop iteration.

To add randomness to our programming model while still keeping the language simple, we additionally allow finite, discrete probabilistic choice when assigning a variable. In symbols, we allow the (tuple) assignment

$$(x, y, z, \dots) \leftarrow (x_1, y_1, z_1, \dots)\ [p_1]\ (x_2, y_2, z_2, \dots)\ [p_2] \dots (x_m, y_m, z_m, \dots)\ [p_m],$$

where $\sum_i p_i = 1$ and $x_i, y_i, z_i, \dots \in \mathbb{Q}$. The semantic of the previous assignment is that after the instruction, it holds that $\mathbb{P}(x = x_i \wedge y = y_i \wedge z = z_i \wedge \dots) = p_i$.

When reasoning about moments of program variables, typically neither $\mathbb{E}[x^\ell] = \mathbb{E}[x]^\ell$, nor $\mathbb{E}[xy] = \mathbb{E}[x]\mathbb{E}[y]$ is true. Therefore, any hope to describe all polynomial relationships among all moments by polynomials over finitely many variables is futile. A natural restriction is to consider polynomials over the expected value of *finitely many monomials*, i.e., expressions of the form $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_k^{\alpha_k}$.

Definition 15 (Monomials of Bounded Degree)**.** Let ℓ be a positive integer, then the *set of monomials of order less than or equal* ℓ is defined as

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

Remark. The introduction of a single bound ℓ for all monomial powers is for the sake of simplicity and the framework would allow for more flexibility here.

This definition allows us to define a probabilistic polynomial invariant for probabilistic loops. In this section, we will restrict ourselves to unbounded loops that allow sampling from finite, discrete probability distributions, as well as branching. We explicitly exclude nondeterminism and hence enforce branching conditions. In this sense, an invariant of order ℓ is a polynomial $p \in \mathbb{Q}\left[\mathbb{E}^{\leq \ell}\right]$ that vanishes over all possible configurations at the loop head, i.e., $\forall n : p\left(\mathbb{E}[x_{1,n}], \ldots, \mathbb{E}[x_{1,n}^{\ell} \cdots x_{k,n}^{\ell}]\right) = 0$. Note that for simplicity of presentation, we restrict the location where the invariant must hold to the loop head, to enforce that it actually holds at every iteration.

The restriction to unbounded loops is crucial here for the following reason. In the (non-)deterministic setting, a loop condition may terminate a run through the program, but the termination of a run does not influence the correctness of an invariant. For illustration, if at the loop head, $x - y = 0$ always holds, then the relation will hold both after the loop and at the beginning of the next iteration. However, in the probabilistic setting this is not necessarily the case. Consider the following example:

```
$x \leftarrow 0$
while $x < 1$ do
    $x \leftarrow -1$ [1/2] 1
end while
```

In this example, whenever the loop head is reached, it holds that $\mathbb{E}[x] = 0$. However, the loop condition terminates some runs through the program and removes some probability mass, such that the invariant does not hold afterward. Even worse, the relation $\mathbb{E}[x] = 0$ may hold every time the loop head is reached, but it does not hold after termination, since then certainly $x = 1$. This illustrates that in the probabilistic case, handling a loop condition is non-trivial and can be partially solved by implementing "stuttering" semantics as in [CS14, MSBK22]. Finding invariants that hold at termination is an interesting topic for future work and could be based on the foundations laid here.

Decidability. It is an interesting question to ask how results from the (non-)deterministic case transfer to the probabilistic case. We can interpret deterministic programs as a subclass of probabilistic programs, where for all program variables x and all iterations n, it holds that $\mathbb{E}[x_n] = x_n$. *Therefore, the strongest probabilistic invariant of degree one coincides with the strongest deterministic polynomial invariant*, since both consist of all the polynomials over x (resp. $\mathbb{E}[x]$) that vanish over the reachable configurations. This directly implies, that solving the problem in the probabilistic setting cannot possibly be easier than in the deterministic setting. Therefore, the problem of finding the strongest probabilistic invariant for single-path polynomial loops is Skolem-hard as well, following from the observation above and Theorem 6.9.

It has been shown, that for single-path, affine programs, the problem of computing expected values of monomials in closed form is decidable [BKS19]. Once these are computed, the approach presented in [Kov08] allows us to compute the strongest polynomial invariant over these monomials, leading us to the conclusion that the problem can be solved for single-path affine loops. Later, it has been shown that as long as all branching statements are affine and *over variables with finite domain*, the respective branching statements can be eliminated and hence the problem is also decidable for such multi-path affine programs [MSBK22]. This also implies, that the problem for polynomial programs with branching over finite variables is equally hard as the problem for polynomial programs without branching, but including probabilistic choice.

This naturally leads to the question of *what can be said about multi-path affine programs with unbounded branching.* In the following, we show that [MS04b] can be transformed from the nondeterministic to the probabilistic case to show that probabilistic branching in combination with a single guarded assignment leads to undecidability. Interestingly, the only modification required is to select a successor configuration probabilistically instead of nondeterminstically.

Lemma 6.3 (Undecidability of Strongest Probabilistic Polynomial Invariant)**.** *The problem of finding the strongest polynomial invariant is undecidable for the class of probabilistic multi-path affine programs, if branching over* (in)equality *conditions involving a variable with infinite domain is allowed.*

Proof. The proof is by reduction from Post's correspondence problem (PCP) and closely follows the nondeterministic proof from [MS04b]. An PCP instance then 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 to the instance is a sequence of indices (i_k), $1 \leq k \leq K$ where each $i_k \in \{1, \ldots, N\}$ and it holds that the concatenation of the substrings indexed by the sequence are identical, in symbols:

$$x_{i_1} \cdot x_{i_2} \cdot \ldots \cdot x_{i_K} = y_{i_1} \cdot y_{i_2} \cdot \ldots \cdot y_{i_K}$$

Note that the tuple elements may be of different size, and that any instance of the PCP over a finite alphabet Σ can be equivalently represented over the alphabet $\{0, 1\}$ by a binary encoding. It is well known that the PCP is undecidable [Pos46].

Now, given an instance of the (binary) PCP, construct the following affine, probabilistic program that encodes the binary strings as integers:

```
$x, y, z, t \leftarrow 0, 0, 0, 0$
while $\star$ do
    $(k_x, k_y, \Delta x, \Delta y) \leftarrow \left(2^{|x_1|}, 2^{|y_1|}, x_1, y_1\right)[1/N] \ldots \left(2^{|x_N|}, 2^{|y_N|}, x_N, y_N\right)[1/N]$
    $x \leftarrow k_x x + \Delta x$
    $y \leftarrow k_y y + \Delta y$
    $z \leftarrow x - y$
    if $z = 0$ then                                        ▷ $1 > z > -1$
        $t \leftarrow 1$
    end if
end while
```

The idea is to randomly pick a pair of integer-encoded strings and append them to the string built so far, by left-shifting the existing bits of the string and adding the randomly selected string. This program only performs a single unbounded branch.

Now, if the PCP instance does not have a solution, it will hold that $\forall n : t_n = 0$. Hence, $\mathbb{E}[t] = 0$ is part of the strongest polynomial ideal. In case the PCP instance does have a solution $(i_k), 1 \leq k \leq K$, then after exactly K iterations it holds that $\mathbb{P}(x_K = y_K) \geq \left(\frac{1}{N}\right)^K$, as this is the probability of always "guessing" the right index. As t is an indicator variable, i.e., $t = [x = y]$, it holds that $\mathbb{E}[t_n] = P(t_n = 1) = P(x_n = y_n) \geq \left(\frac{1}{N}\right)^K \geq 0$. Hence, $\mathbb{E}[t_n] \neq 0$ in iteration n and, we conclude that it is not part of the strongest polynomial invariant.

From this reduction, we conclude that the PCP instance has a solution, if and only if the polynomial relation $\mathbb{E}[t] = 0$ is part of the strongest polynomial ideal. As checking the ideal membership of a polynomial is a decidable problem, see Lemma 6.2, we conclude that computing the strongest polynomial invariant for the given class of programs is undecidable. □

Note that the proof requires only affine assignments and affine invariants, implying the undecidability also for polynomial programs and invariants.

The question remains **what can be stated about the polynomial case with restricted branching**, and whether the proof for nondeterministic programs from [HOPW19] can be adapted.

The proof in [HOPW19] reduces the Boundedness problem for Reset Vector Addition System with States (VASS) to the problem of finding the strongest (deterministic) polynomial invariant. In a Reset VASS, the system may change between a finite set of locations and any transition may increment, decrement or reset a vector of unbounded, non-negative variables. Most importantly, a transition can *only be executed if no zero-valued variable is decremented*, which implies that all variables must be non-negative at all times. The *Boundedness Problem for Reset VASS* asks whether, given a reset VASS and a specific location, does it hold that the set of reachable vector valuations is finite. This problem in known to be undecidable [DFS98].

In the reduction, an arbitrary Reset VASS $\mathcal{V}$ over n variables $a_1, \ldots, a_n$ is simulated by a nondeterministic polynomial program $\mathcal{P}$ over $n + 1$ variables $b_0, \ldots b_n$. Note that the programming model is purely nondeterministic, i.e., without equality guards, since introducing them would render the problem immediately undecidable [MS04b]. To avoid zero-testing the variables before executing a transition, the crucial point in the reduction is to map all invalid traces to the all-zero vector and faithfully simulate all valid executions. By properties of the reduction, it holds that the configuration $(b_0, \dots, b_n)$ is reachable in $\mathcal{P}$, if and only if there exists a corresponding configuration $1/b_0 \cdot (b_1, \dots, b_n)$ in $\mathcal{V}$. Essential to the reduction 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.

Motivated by the successful transfer of the proof for unbounded branching from the nondeterministic to the probabilistic setting in Lemma 6.3, a self-suggesting idea would be to try to do the same for the proof in [HOPW19] and replace nondeterministic choice by probabilistic choice. However, transferring this proof to the probabilistic world poses considerable problems. The main issue is, that in the nondeterministic setting any trace is independent of all other traces, while this does not hold in the probabilistic setting. 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 will be 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 point where a reduction similar in spirit would fail.

At this point, it has to be noted how well-suited the Boundedness Problem for Reset VASS is for this reduction, since the model is not powerful enough to determine if a variable is zero, yet the 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, barring a straight-forward simulation Therefore, it seems that proving decidability or undecidability of the probabilistic variant requires a new methodology that is outside the scope of this thesis.

To conclude this section on strongest invariants, we report the current state-of-the-art for probabilistic invariants in Table 6.2, including our own results (Theorem 6.9 and Lemma 6.3).

Program Model	Str. Invariant	
Single-Path Prob., Affine	✓	[BKS19]
Single-Path Prob., Polynomial	Skolem-hard	Theorem 6.9
Multi-Path Prob., Unbounded	✗	Lemma 6.3
Multi-Path Prob., Bounded, Affine	✓	[MSBK22]

Table 6.2: Decidability results for strongest *probabilistic* invariants. The symbol '✓' denotes decidable problems, while '✗' denotes undecidable problems and unsolved problems are marked with '?'.

6.5 Skolem-Hardness of Strongest Invariant for Single-Path Polynomial Loops

As mentioned in Section 6.2, there are no existing results on the decidability of finding the strongest invariant for single-path polynomial loops. We will refer to this problem as SPInv:

The SPInv Problem: Given a single-path polynomial loop, compute a basis of its strongest polynomial ideal.

The main result of this section, presented in Theorem 6.9, is that SPInv it at least as hard as Skolem, in symbols Skolem $\leq$ SPInv. *This implies, that proving the computability of SPInv would be equivalent to proving decidability of Skolem, a major unsolved problem of number theory* [EvdPSW03, Tao08].

This result has immediate impact on probabilistic loop analysis, as it formally proves that characterizing the relationship of variables in probabilistic programs is a hard problem. Moreover, it shows that inclusion of non-affine arithmetic in the programming model makes the task of analysis and inference much more difficult.

Given an instance of Skolem, we will say that the instance is *positive*, if there exists some index such that the sequence described by the linear recurrence is zero at this index and negative otherwise.

Outline of the proof. Given an instance of Skolem, we construct an instance of SPInv that has finitely many configurations if and only if the instance of Skolem is positive (Corollary 6.7). It turns out, that if a program has only finitely many configurations, then the variety of the invariant ideal is finite as well (Lemma 6.8). As the problem of determining whether an affine variety is finite is decidable, using an oracle for SPInv, we could decide whether an arbitrary instance of Skolem is positive or negative.

Consider an instance of Skolem of order k, i.e., a sequence $u(n), n \in \mathbb{N}_0$ and coefficients $a_0, \dots a_{k-1} \in \mathbb{Z}$ with $a_0 \neq 0$ such that

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

Together with the initial values $u(0), \dots, u(k-1)$, the coefficients uniquely specify the sequence $u(n)$.

For the reduction, we construct a loop over $2k$ variables, namely $x_0, \dots, x_{k-1}$ and $s_0, \dots, s_{k-1}$, such that the ratio $\frac{x_0(n)}{s_0(n)}$ equals the sequence values $u(n)$. To initialize these variables, let us inductively define the constant values $\hat{x}_i$ and $\hat{s}_i$, where $0 \leq i < k$ and $\hat{s}_0 := 1$ and $\hat{x}_0 := u(0)$. Further, to define $\hat{s}_i$ and $\hat{x}_i$, assume that for all $0 \leq \ell < i$, $\hat{s}_\ell$ and $\hat{x}_\ell$ have been initialized and set

$$\hat{s}_i := \prod_{\ell=0}^{i-1} (2\hat{x}_\ell) \qquad \hat{x}_i := u(i) \cdot \hat{s}_i.$$

From the Skolem instance, we construct the single-path polynomial loop shown in Figure 6.1. Intuitively, the x_i variables are a non-linear variant of the sequence $u(n)$ that, once they reach 0, remain 0 forever. To make sure that the program has finitely many configurations if and only if the sequence $u(n)$ has a zero, we use the other set of variables s_i. As x_i, once some variable s_i reaches zero, it remains zero forever. However, in case $u(n)$ does not have a zero, then the values of s_i at least double in each iteration, ensuring infinitely many configurations.

```
$\begin{bmatrix} x_0 & s_0 & \dots & x_{k-1} & s_{k-1} \end{bmatrix} \leftarrow \begin{bmatrix} \hat{x}_0 & \hat{s}_0 & \dots & \hat{x}_{k-1} & \hat{s}_{k-1} \end{bmatrix}$
while $\star$ do
    $\begin{bmatrix} x_0 \\ s_0 \\ \vdots \\ x_{k-2} \\ s_{k-2} \\ x_{k-1} \\ s_{k-1} \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ s_1 \\ \vdots \\ x_{k-1} \\ s_{k-1} \\ \sum_{i=0}^{k-1} a_i \cdot x_i \cdot \prod_{\ell=i}^{k-1}(2x_\ell) \\ 2 \cdot x_{k-1} \cdot s_{k-1} \end{bmatrix}$
end while
```

Figure 6.1: The program constructed in the reduction from Skolem to SPInv.

We now prove the main property of the reduction.

Lemma 6.4. *For the SPInv instance in Figure 6.1, it holds that $\forall n \geq 0$ and $0 \leq i < k$:*

$$x_i(n) = s_i(n) \cdot u(n+i) \tag{6.1}$$

$$s_i(n) = \prod_{\ell=0}^{n-1}(2x_0(\ell)) \cdot \prod_{\ell=0}^{i-1}(2x_\ell(n)) \tag{6.2}$$

Proof. We will prove these 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: $n = 0$. Assume $i = 0$, then it is easy to see that (6.1) and (6.2) hold, by definition of $\hat{s}_0 := 1 = \prod_{\ell=0}^{-1}(2x_0(\ell)) \cdot \prod_{\ell=0}^{-1}(2x_\ell(0))$ and $\hat{x}_0 := u(0) = s_0(0) \cdot u(0)$.

To see that (6.1) holds for $0 < i < k$, note that the initial values satisfy the equation by definition, as $x_i(0) = \hat{x}_i := u(i) \cdot \hat{s}_i = u(i) \cdot s_i(0)$.

For (6.2), this follows from the definition as well, since

$$s_i(0) = \hat{s}_i := \prod_{\ell=0}^{i-1}(2\hat{x}_\ell) = \prod_{\ell=0}^{i-1}(2x_\ell(0)).$$

Induction Step - Case 1: $n > 0 \ \wedge \ 0 \leq i < k-1$. By the lexicographical ordering, $(n, i+1) < (n+1, i)$, hence we may assume that (6.1) and (6.2) hold for $(n, i+1)$, i.e.,

$$x_{i+1}(n) = s_{i+1}(n) \cdot u(n+i+1) \tag{6.3}$$

$$s_{i+1}(n) = \prod_{\ell=0}^{n-1}(2x_0(\ell)) \cdot \prod_{\ell=0}^{i}(2x_\ell(n)) \tag{6.4}$$

To prove (6.1) for $n+1$, we are required to show that

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

From the assignments in the program, we conclude that $x_i(n{+}1) = x_{i+1}(n)$ and $s_i(n{+}1) = s_{i+1}(n)$ and hence the desired relation follows directly from (6.3).

To show that (6.2) holds, we need to prove that

$$s_i(n+1) = \prod_{\ell=0}^{n}(2x_0(\ell)) \cdot \prod_{\ell=0}^{i-1}(2x_\ell(n+1))$$

This is done by instantiating (6.4), then performing index manipulation and using the relations $x_i(n{+}1) = x_{i+1}(n)$ and $s_i(n{+}1) = s_{i+1}(n)$:

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

Induction Step - Case 2: $(n > 0$ and $i = k-1)$. We will show that (6.1) holds by directly proving equivalence to the assignment in the program. To do so, we first write down the desired relation and replace $s_{k-1}(n{+}1)$ by the assigned value and replace $u(n{+}k)$ by the recurrence relation.

$$\begin{aligned} x_{k-1}(n{+}1) &= s_{k-1}(n{+}1) \cdot u(n{+}k) \\ &= 2 \cdot s_{k-1}(n) \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n{+}i)\right) \end{aligned}$$

Next, we rearrange and apply the induction hypothesis (6.2) for $(n, k-1)$ and (n, i), to obtain:

$$
\begin{aligned}
x_{k-1}(n{+}1) &= 2 \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n{+}i) \cdot s_{k-1}(n) \right) \\
&= 2 \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n{+}i) \cdot \underbrace{\prod_{\ell=0}^{n-1} (2x_0(\ell)) \cdot \prod_{\ell=0}^{k-2} (2x_\ell(n))}_{s_{k-1}(n) \text{ by } (6.2)} \right) \\
&= 2 \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n{+}i) \cdot \underbrace{\prod_{\ell=0}^{n-1} (2x_0(\ell)) \cdot \prod_{\ell=0}^{i-1} (2x_\ell(n))}_{=s_i(n) \text{ by } (6.2)} \cdot \prod_{\ell=i}^{k-2} (2x_\ell(n)) \right) \\
&= 2 \cdot x_{k-1}(n) \cdot \left(\sum_{i=0}^{k-1} a_i \cdot u(n{+}i) \cdot s_i(n) \cdot \prod_{\ell=i}^{k-2} (2x_\ell(n)) \right) \\
&= \sum_{i=0}^{k-1} a_i \cdot u(n{+}i) \cdot s_i(n) \cdot \prod_{\ell=i}^{k-1} (2x_\ell(n))
\end{aligned}
$$

It is now possible to apply the induction hypothesis (6.1) to replace $u(n{+}i) \cdot s_i(n)$ with $x_i(n)$ and arrive at the relation:

$$
x_{k-1}(n{+}1) = \sum_{i=0}^{k-1} a_i \cdot x_i(n) \cdot \prod_{\ell=i}^{k-1} (2x_\ell(n))
$$

This is exactly the assignment in the constructed program, hence we conclude that the claim holds.

To prove (6.2) is done by using the assignment and the induction hypothesis for $(n, k-1)$.

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

As we have closed all the cases, this concludes the proof of the lemma. □

From Lemma 6.4, we are now able to derive two crucial properties of the reduction, that relate the number of program configurations to the Skolem problem.

Lemma 6.5. *If the Skolem instance is positive, then the SPInv instance in Figure 6.1 has only finitely many configurations.*

Proof. As the instance of Skolem is positive by assumption, the respective sequence has at least one zero, hence there is some smallest $N \in \mathbb{N}_0$ such that $u(N) = 0$. From (6.1) of Lemma 6.4, we infer

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

By using this equation and (6.2), we see that for $n > N$, each s_i contains $x_0(N)$ as a factor and is hence zero.

Therefore, all values of $s_i(n), n > N$ are zero. Additionally, as $x_i(n) = s_i(n) \cdot u(n+i)$ by (6.1), we conclude that for all $n > N$, it holds that $x_i(n) = s_i(n) = 0$. As N is finite, there are only finitely many configurations before all program variables remain zero forever. □

Lemma 6.6. *If the Skolem instance is negative, then the SPInv instance in Figure 6.1 has infinitely many configurations.*

Proof. By assumption, the sequence specified by the Skolem problem does not have a zero, i.e., for all $n \geq 0$, $u(n) \neq 0$. As we consider integer sequences $u(n)$ and only perform integer operations, we have that $x_i(n) \in \mathbb{Z}$ and hence $|2x_i(n)| > 1$. For now assume that $x_i(n)$ does not have a zero, then by the assignment $s_{k-1} \leftarrow 2 \cdot s_{k-1} \cdot x_{k-1}$, the absolute value of s_{k-1} will strictly increase in each iteration, hence there must be infinitely many configurations.

It is left to show that $x_i(n) \neq 0$ and $s_i(n) \neq 0$ for all n. We prove this by induction and by using Lemma 6.4.

Base Case: $n = 0$**.** Then $s_0(0) = \hat{s}_0 := 1$ and $x_0(0) = \hat{x}_0 := u(0)$, which is non-zero by assumption. Now assume $x_\ell(0) > 0$ and $s_\ell(0) > 0$ for $\ell < i$, then $x_i(0) > 0$ and $s_i(0) > 0$ by the reason following. We have $s_i(0) = \hat{s}_i := \prod_{\ell=0}^{i-1}(2\hat{x}_\ell)$ and this is non-zero, as $\hat{x}_\ell = x_\ell(0) \neq 0$. Additionally, $x_i(0) = \hat{x}_i := \hat{s}_i \cdot u(i)$, which are both non-zero. Hence, we conclude that for all $0 \leq i < k$, $s_i(0) \neq 0$ and $x_i(0) \neq 0$.

Induction Step: $n > 0$**.** By the induction hypothesis, $x_i(n) \neq 0$ and $s_i(n) \neq 0$ for all $0 \leq i < k$. Now if $0 \leq i < k-1$, then $s_i(n+1) = s_{i+1}(n) \neq 0$ and $x_i(n+1) = x_{i+1}(n) \neq 0$, hence only the case $i = k-1$ is left.

By the assignment in the program, $s_{k-1}(n+1) = 2 \cdot x_{k-1}(n) \cdot s_{k-1}(n)$, and by the induction hypothesis, both factors are non-zero, hence $s_{k-1}(n) \neq 0$. By (6.1), $x_{k-1}(n) = s_{k-1}(n) \cdot u(n+i)$, and by assumption $u(n+i) \neq 0$, hence $x_{k-1}(n) \neq 0$. □

We combine the previous two lemmas into a corollary:

Corollary 6.7. *The SPInv instance from Figure 6.1 has finitely many configurations if and only if the Skolem instance is positive.*

We now proceed to the final intermediate lemma of the proof, which asserts that the Zariski closure of a finite set is still finite.

Lemma 6.8. *Let $\mathbb{K}$ be a field and let $S \subset \mathbb{K}^k$ be a set of points, then the Zariski closure $\mathbf{V}(I(S))$ of S is finite if and only if S is finite.*

Proof. The Zariski closure $\mathbf{V}(I(S))$ is the smallest set that contains S, and is affine, that is, it can be written as the set of common roots of a polynomial ideal I. Now if the set S is infinite, then any set containing it is trivially infinite. To show that the closure is finite in case the set is finite, note that for any point $s = (s_1, \dots, s_k) \in S$, we can construct the polynomial ideal $I_s = \langle x_1 - s_1, \dots, x_k - s_k \rangle$ over $\mathbb{K}[x_1, \dots, x_k]$ that has $\mathbf{V}(I_s) = \{s\}$. As the finite union of affine varieties is an affine variety, we conclude that $\mathbf{V}(I(S)) = S$, which is finite by assumption. □

We now see, that the dimension of the Zariski closure of the program configurations relates to the solution of the Skolem problem. The following theorem puts together the previous insights:

Theorem 6.9. *SPInv is Skolem-hard, that is, under the assumption that SPInv is computable, the Skolem problem is decidable (Skolem $\leq$ SPInv).*

Proof. Given an instance of the Skolem problem, by Corollary 6.7, we can construct an instance of SPInv, i.e., a single-path polynomial loop, that has finitely many configurations if and only if the Skolem instance is positive. By Lemma 6.8, we conclude that this relation does not only hold for the set of configurations, but also its Zariski closure.

Assuming we could solve SPInv, giving us a basis of the strongest polynomial ideal I, then the variety $\mathbf{V}(I)$ is finite if and only if the original instance of Skolem is positive. As the problem of deciding the finiteness of an affine variety is decidable [CLO97, Chapter 5, §3, Theorem 6], we close the proof. □

CHAPTER 7

Related Work

At the very heart of probabilistic programming lies the tasks of characterizing the underlying distribution and performing inference to account for observations. As these problems are intrinsically hard, there have been various approaches that try to solve these problems, offering different tradeoffs between performance and approximate results. The different existing approaches can be roughly partitioned into two sets, one being exact *program analysis-based methods*, while the other one consists of approximate, *sampling-based methods*. In this chapter, we first provide an overview of existing program analysis-based methods (Section 7.1), followed by a review of sampling-based methods (Section 7.2). Finally, we summarize the state-of-the-art in probabilistic program analysis, with special emphasis on loops (Section 7.3).

7.1 Program Analysis-Based Methods

7.1.1 Foundations and Calculi-Based Methods

The very foundation of probabilistic programming has been laid in the seminal work of Kozen [Koz79], where he introduced probabilistic programming and presented semantics. Shortly after, Kozen also presented means to verify probabilistic programs in the style of Dijkstra's weakest precondition calculus [Koz83]. Crucially, the framework chosen by Kozen does not reason about distributions, but over the expected value of measurable functions. This line of research has been further advanced by McIver and Morgan, who deepened the relation to Dijkstra's calculus and introduced nondeterminism in the probabilistic programming landscape [MM05]. Moreover, the work of [OGJ$^+$18] showed how to incorporate conditioning into such calculi.

In principle, this framework is able to verify all types of probabilistic programs. However, the main limitation lies in the verification of loops, as it requires a fixpoint computation [Mor96, OGJ$^+$18]. Computing this fixpoint is undecidable in general, and even worse, already finding sufficiently strong bounds for verification purposes is undecidable [KKM19]. Even though there has been some work regarding automated analysis, existing methods either rely on program templates [GKM13, FZJ+17], or are restricted to specially-shaped loops [BKS19].

Another recent approach uses generating functions to represent probability distributions and reason about them using program transformers [KBK+20]. Nevertheless, for loop analysis their approach also requires manual guidance, akin to loop invariants.

7.1.2 Path-Exploration-Based Methods

In case a probabilistic program only has finitely many possible executions, it is apparent that path enumeration is a feasible strategy to extract the modelled probability distribution. In [GDV12], the authors consider loop-free programs with linear integer arithmetic in combination with discrete, uniform distributions. By restricting itself to linear programs and conditionals, the program actually describes a collection of convex polyhedra in a higher dimensional space. As only discrete, uniform distributions are contained in their model, the probability mass is uniformly distributed, hence computing the probability of an event can be reduced to a simpler model counting problem.

However, in probabilistic program verification, it may be sufficient to assert that a given propositional formula holds with a certain probability, which can be often be achieved by considering finitely many paths in which enough probability mass is concentrated. In [FPV13], the authors thus extend the previous approach by allowing loops, where they avoid enumerating all paths by considering a finite subset of control paths. Consequently, they can bound the probability that a certain assertion will hold at termination. The restriction to discrete, uniform distributions is lifted in [SCG13], at the price of increased complexity when computing the probability mass inside the polyhedra, which may require approximate techniques.

Another proposed approach is to directly track the (joint) probability distribution as it is modified step-by-step by the program. When allowing arbitrary distributions, the disadvantage here is that the resulting computations may become complex and no complete treatment can be guaranteed. The techniques employing this technique therefore may return unevaluated integrals [GMV16] or resort to approximate Markov chain Monte Carlo (MCMC) techniques [NCR+16], where both tools are restricted to finitely many paths through the program.

In traditional program verification, symbolic techniques had a major impact, by utilizing symmetries within the model. Similarly, for discrete programs, it is advantageous to represent the distribution symbolically, by using a data structure similar to a BDD [CRN+13]. This work provides complete treatment for loop-free programs, while for loop analysis, a fixpoint computation may be necessary. As no termination guarantee can be given, in practice the fixpoint computation is truncated after the distribution does not significantly change anymore.

7.1.3 Knowledge Compilation-Based Methods

A commonly used approach to computationally hard problems is given by knowledge compilation [DM02]. By compiling a model into an intermediate representation, subsequent computations "queries" can be performed more efficiently. Typically, the computational overhead of intermediate compilation is especially beneficial in case the same model is queried multiple times. Depending on the nature of possible queries, different representations have different advantages, as shown in [DM02, KdBR12]. However, note that different restrictions must be placed on the original program such that it can be compiled into a fixed representation.

Weighted Model Counting. Already for Bayesian networks, *weighted model counting* has been successfully utilized [CD08]. This approach has also proved fruitful for probabilistic programs [FdBT$^+$12]. In particular, a discrete probabilistic model with finite domain can be compiled into a conjunctive normal form (CNF) formula and a weight function. Then each model of the formula is assigned a weight that provides information about the probability of some query. The burden of probabilistic analysis is hence reduced to the problem of enumerating models of a CNF formula and evaluating the weight function, which can be done by utilizing today's powerful SAT solvers or compiling into more powerful representations such as BDDs. However, this technique is inherently exclusively applicable to *discrete* domains.

Recently, it has been proposed to compile a probabilistic program into two BDDs, one modelling the program, one modelling observations [HVM20]. This allows very efficient inference, at the price of restricting itself to finite state spaces and the limitation to statically bounded loops.

Weighted Model Integration. A natural generalization to continuous domains is not to count discrete models, but to integrate over continuous regions of the probability space, often called *weighted model integration* or *weighted volume computation*. The idea is to assign each variable a weight function and a set of constraints, which define a region and a density in the probability space [CDM14, BPdB15, ZdB19, ADDN17, MDR19, KMS$^+$18]. Commonly, constraints are formulated in linear real arithmetic and such that the resulting regions in the probability space are convex polyhedra. Explicitly finding (or approximating) the polyhedra is a model counting problem, and typically left to an satisfiability modulo theory (SMT) solver. However, integrating the volume of arbitrary polyhedra is #P-hard and probabilistic inference over the reals is #P-complete *already in loop-free programs* [CDM14].

It has also been noted, that *sum-product expressions* are well suited for efficient inference [SG12, SRM21]. However, to guarantee that such a representation is always possible, the programming model includes a limitation to statically bounded loops and requires branches of a conditional statement to have the same structure, i.e., branches may only deviate in expressions assigned to variables, but not in control flow.

7.1.4 Approximate Symbolic Methods

To offer more flexibility, some proposed techniques perform analysis by partitioning the state-space into finitely many intervals, which then are assigned a discrete probability [HDM21, BOZ22]. These methods are not exact, but provide convergence guarantees in the sense that for infinitely many intervals the distribution on intervals converges towards the exact distribution. Such techniques may provide a further compromise between sampling-based methods, presented below, and exact methods from program analysis.

7.2 Operational Analysis

Loosely similar to the wave-particle duality in physics, a probabilistic program offers two perspectives. Probabilistic programs are used to create a *model* and can be viewed as such, but at the same time, they are *programs* in the traditional sense. Therefore, a probabilistic program can be executed just like a traditional program, where random number generators are used for draws from probability distributions. Therefore, some concrete trace is *sampled* from the full space of possible executions. It has to be stressed, that in any concrete trace, a variable does here hold some *value*, while from the model perspective, a variable always holds a distribution.

Nevertheless, the collection of all concrete traces through the program completely specifies the distribution of the model, as each possible trace has a certain result and some probability. By exploring all possible paths through the program, it is thus possible to compute the *exact* distribution of the model. However, as the number of paths through a program is often immense (or even infinite), this becomes quickly impractical [GS14]. Therefore, all practical sampling-based approaches resort to approximations and extrapolate from a finite number of samples.

7.2.1 Importance Sampling

A simple and conceptually appealing idea is to run the probabilistic program a fixed number of times, and then use the resulting samples to characterize the expected value of some function under the *posterior distribution*. This can be done by assigning a weight to each execution, intuitively corresponding to its probability, and then approximating the expected value as a weighted average [CL08, vdMPYW18]. A natural and commonly used choice for the weight function is the likelihood function, in which case importance sampling is also known as *likelihood weighting*.

However, one crucial issue remains; the problem of taking enough samples to achieve satisfactory coverage. To provide a useful approximation of the posterior, it may be that the number of necessary samples is prohibitively high, and that a large fraction of samples is taken in uninteresting regions of the probability space, as it is hard to tell where the function of interest has high probability density. To avoid this odyssey in the probability space, several techniques have been proposed, where the most well-known one is presented in the following section.

7.2.2 Markov Chain Monte Carlo

To avoid the unguided wandering of the sampling process in the probability space and hence reduce the number of necessary samples, it is desirable to specifically sample from regions with high probability density *under the posterior*. However, how to take samples from high probability regions in an unknown distribution is not obvious, to say the least. Astonishingly, this is nevertheless possible by a powerful approach known as *Markov Chain Monte Carlo* [vdMPYW18, CL08, Bet17].

The key idea is to construct a Markov chain that has the posterior distribution as its stationary distribution, and therefore, by traversing the Markov chain, high probability regions in the probability space are sampled. The main task left is to construct a Markov kernel, such that the resulting Markov chain actually has the posterior distribution as its stationary distribution.

Various approaches have been suggested, such as *Gibbs sampling* or its generalization *Metropolis-Hastings sampling* [vdMPYW18, CL08, Bet17]. These approaches have been successfully deployed in an impressive amount of real-world software, such as the influential Bayesian inference using Gibbs sampling (BUGS) family [LSTB09].

Some practical issue are, that it is hard to tell if the Markov chain has traversed sufficient portions of the state-space and that the Markov chain requires some time to converge towards the stationary distribution [Bet17]. As probabilistic models have become more complex, convergence speed has become an issue, especially as it is very hard to tell how long the Markov chain takes until converging.

One recent improvement is *Hamiltonian Monte Carlo (HMC)* and especially the No U-Turn Sampler (NUTS), which are MCMC instantiations that provide improved convergence properties and are at the heart of today's MCMCs inference systems, such as Stan [CGH^{+}17] and PyMC [SWF15]. However, these require differentiable models, as they depend also on the gradient of the posterior, which is especially problematic for discrete models [Bet17, CL08].

7.2.3 Variational Inference

In contrast to MCMC, which tries to synthesize a distribution solely out of the model, methods based on variational inference (VI) additionally assume some template, more specifically a parameterized distribution, and tune the parameters of the distribution to fit the posterior as well as possible.

In *expectation propagation* as used by e.g., Infer.NET [MWG^{+}18], the tuning step is done by iteratively minimizing the Kullback–Leibler (KL)-divergence between the posterior and the proposal distribution, in the hope that this process converges [Min01]. Similar VI methods generalize this and allow the proposal distribution to be a product of multiple distributions, which allows more flexibility at the price of increased complexity [vdMPYW18, CL08]. One popular approach is to fit the distribution by minimizing the gradient of the KL-divergence. To minimize this gradient, automatic differentiation is a powerful tool and automatic differentiation variational inference (ADVI) is a powerful and recent approach implemented in state-of-the-art tools as an alternative to MCMC [KTR$^+$16, CGH$^+$17, SWF15]. However, as HMC this requires the model to be differentiable, which is especially problematic for discrete models.

7.3 The State of Probabilistic Loop Analysis

In general, there is no approach that can handle arbitrary probabilistic loops, since there is not even an approach that can handle arbitrary deterministic loops.

The work of [SCG13] and [FPV13] in some sense allow approximate inference on loops, by increasing the number of iterations to analyze, but cannot provide a complete characterization in general. In the work of [CRN$^+$13], a single iteration of the loop is peeled-off until a fixpoint is (approximately) reached, however no convergence guarantees exist. Finally, the works of [SG12] and [CMS22] consider recursive programs over finite domains, which allows them to build finite equation systems over method calls. However, the resulting equation systems may be very large and are not guaranteed to be efficiently, let alone exactly, solvable.

One approach that focuses solely on potentially unbounded probabilistic loops is presented in [MSBK22], where a program is considered as describing a recurrence relation over the moments of program variables. If these recurrence relations can be solved, i.e., if the recurrences are linear, then arbitrary moments can be computed. However, severe restrictions are based on variables influencing the control flow and the programming model lacks observe-statements.

Calculus-based approaches such as [Koz83, Mor96, MM05, OGJ$^+$18] can theoretically perform inference on arbitrary probabilistic programs, however they are severely limited by the necessity to compute fixed points or come up with appropriate loop invariants, which is unfortunately undecidable [KKM19].

Sampling based approaches can handle loops in principle, but the probability of long-running executions may be close to zero, hence MCMC or VI techniques may require very long time to explore these possibilities. Additionally, for MCMC techniques convergence remains an issue, while for HMC and ADVI the parameter is assumed to be differentiable, which poses an issue for discrete parameters. We note that these techniques have their strengths and shortcomings, but are tools that complement each other when estimating the posterior distribution [BKM17] based on operational inference. However, all of them are approximate rather than exact.

CHAPTER 8

Conclusion

The analysis of probabilistic programs is a challenging and complex undertaking, however, equally rewarding at the same time, as it is a topic of extraordinary elegance and importance. For, it gives a natural way to *model and analyze probabilistic processes* and, at the same time, is able to bring *domain knowledge to machine learning*.

This thesis investigates the extraction of distributions modelled by probabilistic programs, with the ultimate goal of enabling (Bayesian) inference. As program verification is a notoriously hard undertaking, with constantly present undecidability issues, the problem requires creative approaches. We investigated a variety of techniques suited to the analysis of probabilistic programs, including a direct analysis based on the Markov chain semantics, distribution recovery based on moments over program variables, analytic methods utilizing properties of distributions, a type system and invariant techniques.

More specifically, in Chapter 3, we provided a comprehensive method for finite-state programs and implemented it in the new tool Blizzard. As the benchmarks show, the approach is competitive with existing works, e.g., in [GMV16, HVM20].

To allow analysis of another program class, we investigated the recovery of discrete distributions from finitely many moments in Chapter 4. This allows recovery of distributions for infinite-state programs where the variables of interest have finite support.

In order to analyze specifically infinite-state programs, in Sections 5.1-5.2 we present a class of programs, where probabilistic loops can be replaced by an equivalent, loop-free program. This considerably simplifies both analysis and inference, for exact as well as approximate techniques. Additionally, in Section 5.3, we developed a type system, utilizing relations among probability distributions, to identify the distribution of variables by a type-checking approach.

Finally, in Chapter 6, we investigate the hardness of probabilistic program analysis and what makes it difficult, by studying the problem of invariant synthesis. We present non-probabilistic invariant techniques and research the applicability of those techniques in the probabilistic setting. At the same time, we extended the state-of-the-art for deterministic programs by proving that the open problem of finding the strongest polynomial invariant for a single-path polynomial loop is harder than the Skolem problem, a major unsolved problem in number theory.

The solutions presented in this thesis advance the current state-of-the-art by enabling exact program analysis for specific classes of probabilistic programs. These insights can be used to efficiently perform inference for probabilistic programs. We illustrate inference for probabilistic programs in the following section.

8.1 Performing Inference for Probabilistic Programs

Consider the bounded random walk from Example 3, where the values of x after termination specify the random variable X. This program can be analyzed fully automatically using Blizzard, as shown in Chapter 3, where we obtain the likelihood function $\mathbb{P}(X = x \mid p)$ in dependence of the symbolic parameter p. Now assume, that we observed the following concrete outcomes of the modelled process:

Trial i	1	2	3	4	5	6	7	8	9	10
Outcome $\hat{x}_i$	-1	10	0	10	10	0	12	11	11	0

Table 8.1: Observed outcomes of the process modelled in Example 3.

The task of inference is to now use the data in Table 8.1 and conclude probable values for the parameter p. There are two major paradigms to perform inference, *frequentist* and *Bayesian*, and we will illustrate one method of each paradigm.

Frequentist inference - Maximum likelihood estimator. The *maximum likelihood estimator* computes the value of p that maximizes the probability that the observed values in Table 8.1 are sampled from the stochastic process modelled in Example 3. The likelihood function $\mathcal{L}(p)$ is defined as the probability that the observed values are produced, and for independent observations, can be computed as the product over the likelihoods of the individual observations. For the observations in Table 8.1, the likelihood function is shown in Figure 8.1.

$$\mathcal{L}(p) \coloneqq \prod_{i=1}^{10} \mathbb{P}(X = \hat{x}_i \mid p)$$

Then the maximum likelihood estimator is defined as the value $\hat{p}$, that maximizes the likelihood function, in symbols

$$\hat{p} = \arg\max_{p} \mathcal{L}(p)$$

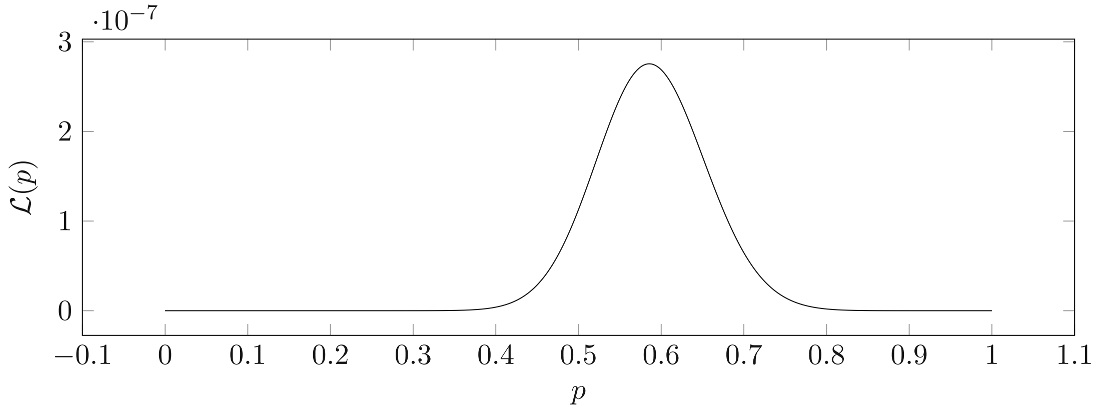


Figure 8.1: The likelihood function $\mathcal{L}(p)$ for the observations in Table 8.1.

As we obtained closed-form solutions for the likelihood functions $\mathbb{P}(X = \hat{x}_i)$ in Chapter 3, we can now compute $\hat{p}$ by using a CAS. For our observations, we compute $\hat{p} = 0.5856$ as the most likely value of p.

Bayesian inference. In contrast to frequentist statistics, Bayesian statistics model a *belief* about the parameter and assign each possible parameter value some probability. In the presence of observations, the belief is updated using Bayes' theorem (Theorem 2.1).

For the bounded random walk of Example 3, we assume that all values of p are equally probable and use a uniform distribution as prior, i.e., $f(p) = [0 \leq p \leq 1]$, where $[\cdot]$ is the Iverson bracket, which is 1 in case $\cdot$ is true, and 0 otherwise.

In the presence of observations $\hat{x}$, we then update the prior $f(p)$ and obtain the posterior $f(p \mid \hat{x})$ according to Bayes' theorem:

$$f(p \mid \hat{x}) = \frac{f(\hat{x} \mid p) f(p)}{f(\hat{x})}$$

Here, $f(\hat{x})$ is the marginal likelihood, where p is marginalized out, in symbols

$$f(\hat{x}) = \int_p f(\hat{x} \mid p) f(p)\, dp.$$

After observing $\hat{x}_1 = -1$, we therefore update the prior as follows and obtain the posterior plotted in Figure 8.2. It can be noted, that the belief about p shifts towards the lower end of the scale, as we observed that the random walk drifted towards the origin, indicating a small value of p.

$$f(p \mid \hat{x}_1) = \frac{\mathbb{P}(X = -1 \mid p)\ [0 \leq p \leq 1]}{\int_p \mathbb{P}(X = -1 \mid p)\ [0 \leq p \leq 1]\, dp} = 9.54\ \mathbb{P}(X = -1 \mid p)\ [0 \leq p \leq 1]$$

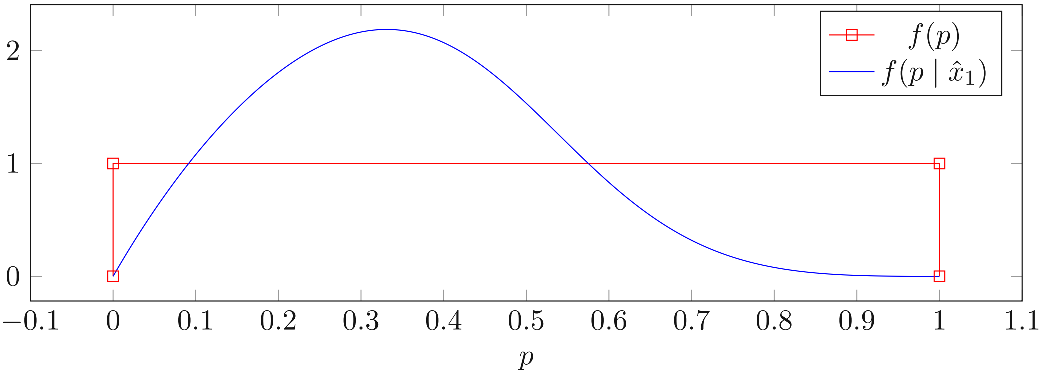


Figure 8.2: The prior $f(p)$ and the posterior $f(p \mid \hat{x}_1)$ after application of Bayes' theorem.

Due to the choice of the uniform prior, the final posterior after repeated application of Bayes' theorem is directly proportional to the likelihood function $\mathcal{L}(p)$ in Figure 8.1.

Note that both inference paradigms can be performed immediately and without further obstacles, once the distribution of X is computed, the problem addressed by this thesis.

8.2 Future Work

There are several interesting avenues to extend this thesis. It would be very interesting to investigate approximate techniques, both in the context of distribution recovery and in the context of infinite state loops.

More specifically, as mentioned in Chapter 4, there is a plethora of existing work on the moment problem, e.g., see [Sch17]. It would be interesting to (approximately) recover distributions that are not necessarily over a finite support given their moments. Moreover, it would be interesting to investigate which restrictions can be placed on probabilistic programs such that distribution recovery from moments is feasible.

Also, for infinite state loops, it may be possible to approximate the distribution of summation variables by the normal distribution, under the assumption that the loop body is executed sufficiently often. The well known central-limit theorem of probability theory then guarantees, that under rather mild conditions, the sum of random variables converges towards a normal distribution. However, the central-limit theorem only refers to the cumulative density, not the probability density, therefore additional conditions are necessary to guarantee the convergence [Bry96].

Another promising research area is the relation of martingales and affine invariants of probabilistic loops. Especially interesting is whether invariant techniques can be used to synthesize martingales for probabilistic loops. This is particularly interesting, as Doob's optional stopping time theorem would then provide some leverage over probabilistic while-loops with loop guards, i.e., it would allow reasoning about the distribution of programs that have a complex stopping criterion.

Acronyms

ADVI automatic differentiation variational inference. 72

BDD binary decision diagram. 20, 24, 68, 69

BUGS Bayesian inference using Gibbs sampling. 71

CAS computer algebra system. 19, 20, 24, 75

CNF conjunctive normal form. 69

HMC Hamiltonian Monte Carlo. 71, 72

KL Kullback–Leibler. 71, 72

MCMC Markov chain Monte Carlo. 68, 71, 72

MGF moment-generating function. 46, 47

NUTS No U-Turn Sampler. 71

PCP Post's correspondence problem. 58, 59

PGF probability-generating function. 45–47

SMT satisfiability modulo theory. 69

VASS Vector Addition System with States. 59, 60

VI variational inference. 71, 72

Bibliography

[ADDN17] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: Probabilistic verification of program fairness. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[BEFH16] Gilles Barthe, Thomas Espitau, Luis María Ferrer Fioriti, and Justin Hsu. Synthesizing probabilistic invariants via doob's decomposition. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 43–61. Springer, 2016.

[Bet17] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo, 2017.

[BGHS17] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. In *POPL*, pages 161–174. ACM, 2017.

[BKM17] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[BKS19] Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. Automatic generation of moment-based invariants for prob-solvable loops. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 255–276. Springer, 2019.

[BOZ22] Raven Beutner, C.-H. Luke Ong, and Fabian Zaiser. Guaranteed bounds for posterior inference in universal probabilistic programming. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 536–551. ACM, 2022.

[BPdB15] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2770–2776. AAAI Press, 2015.

[Bry96] Wlodzimierz Bryc. Limit theorems of probability theory: Sequences of independent random variables (valentin v. petrov). *SIAM Rev.*, 38(3):527, 1996.

[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.*, 41(3-4):475–511, 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CD08] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6–7):772–799, apr 2008.

[CDM14] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in smt and value estimation for probabilistic programs, 2014.

[CGH$^+$17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–32, 2017.

[CL08] Bradley P. Carlin and Thomas A. Louis. *Bayesian Methods for Data Analysis Third Edition*, pages 1–538. Taylor and Francis, June 2008.

[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.)*. Undergraduate texts in mathematics. Springer, 1997.

[CMS22] David Chiang, Colin McDonald, and Chung-chieh Shan. Exact recursive probabilistic programming. *CoRR*, abs/2210.01206, 2022.

[CNZ17] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. Stochastic invariants for probabilistic termination. In *POPL*, pages 145–160. ACM, 2017.

[CRN$^+$13] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM*

SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pages 92–102. ACM, 2013.

[CS13] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013.

[CS14] Aleksandar Chakarov and Sriram Sankaranarayanan. Expectation invariants for probabilistic program loops as fixed points. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2014.

[DDM16] David Deininger, Rayna Dimitrova, and Rupak Majumdar. Symbolic model checking for factored probabilistic models. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 444–460, 2016.

[DFS98] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.

[DL93] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artif. Intell.*, 60(1):141–153, 1993.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Int. Res.*, 17(1):229–264, sep 2002.

[Dur10] Rick Durrett. *Probability: Theory and Examples, 4th Edition.* Cambridge University Press, 2010.

[EvdPSW03] Graham Everest, Alfred J. van der Poorten, Igor E. Shparlinski, and Thomas Ward. *Recurrence Sequences*, volume 104 of *Mathematical surveys and monographs.* American Mathematical Society, 2003.

[FdBT+12] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted cnf's. *CoRR*, abs/1202.3719, 2012.

[FPV13] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 622–631, 2013.

[FT13] Ionuţ Florescu and Ciprian Tudor. *Handbook of Probability.* John Wiley & Sons, Ltd, 2013.

[FZJ+17] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 400–416. Springer, 2017.

[GDV12] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176. ACM, 2012.

[GGH19] Jürgen Giesl, Peter Giesl, and Marcel Hark. Computing expected runtimes for constant probability programs. In *CADE*, volume 11716 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2019.

[GHNR14] Andy Gordon, Thomas A. Henzinger, Aditya Nori, and Sriram Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE Future of Software Engineering)*. IEEE, May 2014.

[GKM13] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Prinsys - on a quest for probabilistic loop invariants. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2013.

[GKM14] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Evaluation*, 73:110–132, 2014.

[GMV16] Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

[GS06] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Society, 2006.

[GS14] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages, 2014. Accessed: 2022-10-18.

[HDM21] Zixin Huang, Saikat Dutta, and Sasa Misailovic. AQUA: automated quantized inference for probabilistic programs. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 229–246. Springer, 2021.

[HJK+20] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm, 2020.

[HJV+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 Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 577–601. Springer, 2021.

[HOPW19] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. On Strongest Algebraic Program Invariants. Submitted, 2019.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley series in computer science and information processing. Addison-Wesley, 1969.

[HVM20] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang. (OOPSLA)*, 2020.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[KBK+20] Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler. Generating functions for probabilistic programs. In Maribel Fernández, editor, *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*, volume 12561 of *Lecture Notes in Computer Science*, pages 231–248. Springer, 2020.

[KdBR12] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *CoRR*, abs/1211.4475, 2012.

[KKM19] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. On the hardness of analyzing probabilistic programs. *Acta Informatica*, 56(3):255–285, 2019.

[KMS+18] Samuel Kolb, Martin Mladenov, Scott Sanner, Vaishak Belle, and Kristian Kersting. Efficient symbolic integration for probabilistic inference. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5031–5037. ijcai.org, 2018.

[KMS+22] Ahmad Karimi, Marcel Moosbrugger, Miroslav Stankovic, Laura Kovács, Ezio Bartocci, and Efstathia Bura. Distribution estimation for probabilistic loops. In *QEST*, volume 13479 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2022.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[Kov08] Laura Kovács. Reasoning algebraically about p-solvable loops. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2008.

[Koz79] Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979*, pages 101–114. IEEE Computer Society, 1979.

[Koz83] Dexter Kozen. A probabilistic PDL. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 291–297. ACM, 1983.

[KP11] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates.* Texts & Monographs in Symbolic Computation. Springer, 2011.

[KTR+16] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference, 2016.

[LSTB09] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The bugs project: Evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049–3067, 2009.

[MDR19] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7825–7833. AAAI Press, 2019.

[Min01] Thomas P. Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[Mor96] Carroll Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Conference on Refinement*, FAC-RW'96, page 10, Swindon, GBR, 1996. BCS Learning & Development Ltd.

[MS04a] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91(5):233–244, 2004.

[MS04b] Markus Müller-Olm and Helmut Seidl. A note on karr's algorithm. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer, 2004.

[MSBK22] Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács. This is the moment for probabilistic loops. *CoRR*, abs/2204.07185, 2022.

[MWG$^+$18] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[NCR$^+$16] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 2016.

[OGJ$^+$18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. Conditioning in probabilistic programming. *ACM Trans. Program. Lang. Syst.*, 40(1):4:1–4:50, 2018.

[Par03] David Anthony Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, University of Birmingham, UK, 2003.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264 – 268, 1946.

[RK04] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*, pages 266–273. ACM, 2004.

[RK07] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, 2007.

[SCG13] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458. ACM, 2013.

[Sch17] Konrad Schmüdgen. *The Moment Problem*. Springer, 2017.

[SG12] Andreas Stuhlmüller and Noah D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *CoRR*, abs/1206.3555, 2012.

[SK20] Bahare Salmani and Joost-Pieter Katoen. Bayesian inference by symbolic model checking. In Marco Gribaudo, David N. Jansen, and Anne Remke, editors, *Quantitative Evaluation of Systems - 17th International Conference, QEST 2020, Vienna, Austria, August 31 - September 3, 2020, Proceedings*, volume 12289 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2020.

[SRM21] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, jun 2021.

[SWF15] John Salvatier, Thomas Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc, 2015.

[Tao08] Terrence Tao. *Structure and Randomness*. American Mathematical Society, 2008.

[vdMPYW18] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *CoRR*, abs/1809.10756, 2018.

[VK23] Anton Varonka and Laura Kovács. On the Undecidability of Loop Analysis. Submitted,, 2023.

[ZdB19] Zhe Zeng and Guy Van den Broeck. Efficient search-based weighted model integration. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 175–185. AUAI Press, 2019.