Towards Formal Verification of Bayesian Inference in Probabilistic Programming via Guaranteed Bounds

Fabian Zaiser

St Catherine's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2024

Abstract

Probabilistic models are an indispensable tool in many scientific fields, from the social and medical sciences to physics and machine learning. In probabilistic programming, such models are specified as computer programs: a flexible yet precise representation that allows for automated analysis. Such probabilistic programs can express both randomized algorithms and statistical or machine learning models. This thesis focuses on the Bayesian framework for reasoning and learning under uncertainty, where prior beliefs about the world (represented as a probability distribution on the model parameters) are updated based on observed data using Bayes' rule. The computation of this posterior distribution, known as Bayesian inference, presents a major challenge.

Probabilistic programming systems allow practitioners to reuse general-purpose inference algorithms, rather than manually deriving new ones for each model. These automated inference methods fall into two categories: exact and approximate. Exact inference is often intractable or imposes restrictions on supported models, whereas approximate methods are more scalable and widely applicable, but the quality of the approximation can be difficult to assess.

This thesis introduces the concept of *guaranteed bounds* on the posterior distribution as a middle ground between exact and approximate inference. These bounds can be computed for a much larger class of probabilistic programs than exact methods and, unlike approximate methods, provide non-stochastic guarantees about the posterior distribution, such as: "the posterior probability of a given event is at least p and at most q". Crucially, these bounds are fully automated and require no user-provided auxiliary information like loop invariants. Thus they can serve as a ground truth to verify and debug approximate inference. As a special case, they can be applied to the automated analysis of randomized algorithms, which is itself a research area of considerable interest.

The main contribution of this thesis is a set of methods for computing guaranteed bounds for probabilistic programs. The first approach uses abstract interpretation with interval arithmetic and is applicable to a universal probabilistic programming language with higher-order functions, continuous distributions, and recursion. The remainder of the thesis deals with imperative probabilistic programs limited to linear operations and discrete observations. The second approach can transform any exact inference algorithm for loop-free programs into a method that computes guaranteed bounds for programs with loops. This is applied to a novel exact inference method for loop-free programs, which leverages probability generating functions and is of independent interest. The third approach provides more informative bounds for probabilistic programs with only discrete distributions, including bounds on moments and tail asymptotics.

All these methods enjoy desirable theoretical properties, such as soundness and, in many cases, convergence to the exact result as the computational budget increases. Experimental evaluations demonstrate that they are applicable to more probabilistic programs than exact inference and can detect problems with approximate inference. Overall, these contributions mark a significant step towards formal verification of Bayesian inference results in probabilistic programming.

Zusammenfassung (German Abstract)

Wahrscheinlichkeitsbasierte Modelle sind ein unverzichtbares Werkzeug in vielen wissenschaftlichen Bereichen, von den Sozial- und Medizinwissenschaften bis hin zur Physik und zum maschinellen Lernen. In der probabilistischen Programmierung werden solche Modelle als Computerprogramme spezifiziert: eine flexible und doch präzise Darstellung, die eine automatisierte Analyse ermöglicht. Solche probabilistischen Programme können sowohl randomisierte Algorithmen als auch Modelle aus der Statistik oder dem maschinellen Lernen ausdrücken. Diese Arbeit konzentriert sich auf die Bayessche Herangehensweise an das Schlussfolgern und Lernen unter Ungewissheit, wobei A-priori-Annahmen über die Welt (in Form einer Wahrscheinlichkeitsverteilung auf den Modellparametern) auf der Grundlage beobachteter Daten mit Hilfe der Bayesschen Regel aktualisiert werden. Die Berechnung dieser A-posteriori-Verteilung, genannt Bayessche Inferenz, stellt eine erhebliche Herausforderung dar.

Probabilistische Programmiersysteme ermöglichen es Praktikern, allgemeine Inferenzalgorithmen wiederzuverwenden, anstatt manuell für jedes Modell neue Algorithmen abzuleiten. Diese automatisierten Inferenzmethoden fallen in zwei Kategorien: exakt und approximativ. Exakte Inferenz ist häufig undurchführbar oder schränkt die unterstützten Modelle ein, wohingegen Näherungsmethoden besser skalierbar und breit anwendbar sind, aber die Qualität der Näherung schwer zu beurteilen sein kann.

Diese Dissertation führt das Konzept von *garantierten Schranken* für die A-posteriori-Verteilung als Mittelweg zwischen exakter und approximativer Inferenz ein. Diese Schranken können für eine viel größere Klasse probabilistischer Programme als exakte Methoden berechnet werden und liefern im Gegensatz zu approximativen Methoden nicht-stochastische Garantien für die A-posteriori-Verteilung, wie etwa: "Die A-posteriori-Wahrscheinlichkeit eines bestimmten Ereignisses ist mindestens p und höchstens q." Entscheidend ist, dass diese Schranken vollständig automatisiert sind und keine vom Benutzer bereitgestellten Zusatzinformationen wie Schleifeninvarianten erfordern. Somit können sie als Grundwahrheit zur Prüfung und Fehlersuche bei approximativer Inferenz dienen. Als Spezialfall können sie auf die automatisierte Analyse von randomisierten Algorithmen angewendet werden, was an sich schon ein Forschungsgebiet von beträchtlichem Interesse ist.

Der Hauptbeitrag dieser Arbeit ist eine Reihe von Methoden zur Berechnung garantierter Schranken für probabilistische Programme. Der erste Ansatz verwendet abstrakte Interpretation mit Intervallarithmetik und ist anwendbar auf eine universelle probabilistische Programmiersprache mit Funktionen höherer Ordnung, stetigen Verteilungen und unbeschränkter Rekursion. Der Rest der Dissertation befasst sich mit imperativen probabilistischen Programmen, die auf lineare Operationen und diskrete Beobachtungen beschränkt sind. Der zweite Ansatz kann jeden exakten Inferenzalgorithmus für schleifenfreie Programme in eine Methode umwandeln, die garantierte Schranken für Programme mit Schleifen berechnet. Dies wird auf eine neue exakte Inferenzmethode für schleifenfreie Programme angewandt, die sich wahrscheinlichkeitserzeugende Funktionen zunutze macht und von eigenständigem Interesse ist. Der dritte Ansatz liefert

informativere Schranken für probabilistische Programme mit nur diskreten Verteilungen, einschließlich Schranken für die Momente und die Asymptotik des Verteilungsrandes.

All diese Verfahren weisen wünschenswerte theoretische Eigenschaften auf, wie Korrektheit und in vielen Fällen die Konvergenz zum exakten Ergebnis bei steigender Rechenzeit. Experimentelle Untersuchungen belegen, dass sie auf mehr probabilistische Programme anwendbar sind als exakte Inferenz und Probleme bei approximativer Inferenz erkennen können. Insgesamt stellen diese Beiträge einen bedeutenden Schritt auf dem Weg zur formalen Verifikation von Bayes'schen Inferenzergebnissen in der probabilistischen Programmierung dar.

Acknowledgements

First, I would like to thank my supervisors, Luke Ong and Andrzej Murawski. I am grateful to Luke for accepting me as a PhD student and introducing me to probabilistic programming, as well as his encouragement and the freedom to explore my own research interests and to complete two industry internships during my PhD. Additionally, I sincerely appreciate Luke inviting me twice to visit Nanyang Technological University, his new university in Singapore, for several months. Lastly, I am grateful to Luke for entrusting me to cover his tutorial teaching at Merton College during his sabbatical, which was a great opportunity for me to gain teaching and mentoring experience.

I am thankful to Andrzej for stepping in to co-supervise after Luke's departure from Oxford, despite probabilistic programming not being his primary area of research. His fresh perspective and questions about my research were very valuable and helped clarify my research ideas. I greatly appreciated his attention to detail when reading my drafts and providing feedback.

I would like to thank my examiners, Joost-Pieter Katoen and Sam Staton, for their time and effort in examining my thesis and for their insightful comments, which sparked an interesting discussion and helped me improve my thesis.

I am also grateful to my departmental advisor Jeremy Gibbons for inviting me to the "Algebra of Programming" meetings, which broadened my horizon at the start of my PhD. I would like to express my gratitude to Janis Voigtländer, who supervised my bachelor's thesis in computer science and who first introduced me to functional programming and programming language theory. Without him, I might not have ended up in programming languages research. I am also immensely grateful to my mentor Eva Darulova as part of the SIGPLAN Long-Term Mentoring (SIGPLAN-M), who gave me an independent perspective and a lot of invaluable advice on how to navigate the academic world.

I would also like to thank my co-authors Raven Beutner and Carol Mak for the enjoyable and fruitful collaborations during my PhD, in spite of the fact that most of our interaction took place online due to COVID. In fact, Raven and I did not meet in person before the conference where I presented our work, yet I valued our collaboration greatly.

I would like to thank my office mates both in Oxford and Singapore: Toby Cathcart Burn, Nathan Corbyn, Maria Craciun, Jerome Jochens, Xuan-Bach Le, Guanyan Li, Carol Mak, Rolf Morel, Dominik Wagner, Peixin Wang, and Leon Witzmann. I am also grateful to the many other PhD students and postdocs in Oxford with whom I had inspiring conversations: Swaraj Dash, Maximilian Doré, Johannes Hartmann, Mathieu Huot, Younesse Kaddar, Jack Liell-Cock, Christina Matache, Sean Moss, Hugo Paquet, Paolo Perrone, Maximilian Pflüger, Tim Reichelt, and Dario Stein. It was a privilege to be surrounded by so many brilliant minds and supportive individuals.

Beyond Oxford, I benefited enormously from interactions with researchers worldwide: Nathanael Arkor, Sacha-Élie Ayoun, Kevin Batz, Christopher Brix, Benjamin Kaminski, Yong Kiam, Lutz Klinkenberg, Alex Lew, Tom Rainforth, Philip Schröer, Matthijs Vákár, Conrad Watt, Stefan Zetzsche, and Đorđe Žikelić. I am also grateful to my internship mentors Daniel Schwartz-Narbonne and Jean-Baptiste Tristan in the

Automated Reasoning Group at Amazon. The internships were an invaluable opportunity to gain experience with formal verification in industry, and I learned a lot from them.

Of course, there is more to life than work, and my PhD would not have been as rewarding without my amazing friends in and outside of Oxford. In particular, thank you, Max Doré, Kalina Naidoo, Max Pflüger, Marius Weber, and Sara Zemljič. Special thanks also go to Maria Craciun for our delightful adventures in Singapore, to Patrick Kaschel for many conversations about Bayesian statistics and life in general, and to my wonderful housemates and friends Michael van de Noort and Laura Wheatley for cheering me up after a tough day of work (sorry I was away from home so much traveling!).

I want to express my deep gratitude to my family, especially my sister Rebecca and my parents, Ingeborg and Frank, for always being there for me. My parents have nurtured my curiosity since I was a child and their support of my interests has steadily continued into my academic endeavors. *Danke!*

Finally, I am profoundly grateful to Linda for accompanying me through the highs and lows of my PhD journey and moving literally across the globe to stay with me. The past four years would not have been the same without you!

Preface

At the start of my PhD, when I first learned about probabilistic programming, my knowledge of statistics and machine learning was quite basic. My main interests within computer science were programming language theory and formal verification. From my anecdotal experience, people in formal methods tend to be skeptical of probability and statistics due to their "fuzziness"; instead they prefer the precision of logical and algebraic methods. Initially, I shared this perspective. However, I have since come to appreciate that probability and statistics are extraordinarily useful in many areas of computer science and science more generally. In fact, I believe that formal methods can contribute significantly to these fields by providing rigorous foundations and techniques for reasoning about probabilistic systems. This thesis represents my efforts to advance this goal by developing methods to compute guaranteed bounds on the posterior distribution of probabilistic programs.

Guide for the reader Given my initially limited background in probability and statistics, the PhD was a big learning experience for me. This thesis documents this journey: I have tried to provide motivation and context that I wish I had had when starting out.

In terms of prerequisites, I assume basic knowledge of programming language theory (e.g. semantics, lambda calculus, type theory) and measure-theoretic probability, which I very briefly review in Section 2.1.

I introduce Bayesian statistics and probabilistic programming in greater detail than strictly necessary (Sections 1.1 to 1.4), with the aim of offering you additional context for the problem this thesis addresses. Moreover, I extensively discuss the literature on probabilistic programming, its history, automated inference, and formal analysis to give you a sense of the state of the art in the field (Sections 2.4 to 2.7). If you are already familiar with these topics, you may want to skim or skip the relevant sections.

The thesis is intended to be read chronologically, but many of the chapters are fairly self-contained and make few references to previous chapters. If you already know probabilistic programming, I recommend that you read at least Sections 1.5 to 1.9, 2.2 and 2.3 before jumping into the chapters with novel contributions (Chapters 3 to 6).

Contents

Abstract ii

Zusammenfassung (German Abstract) iii

Acknowledgements v

Preface vii

Contents viii

List of Abbreviations xii

List of Notations xiv

1 Introduction 1
- 1.1 Probabilistic Modeling . . . 1
- 1.2 Statistical Inference: Frequentism & Bayesianism . . . 3
- 1.3 Probabilistic Programming . . . 6
  - 1.3.1 Graphical Models . . . 6
  - 1.3.2 Probabilistic Programming Languages . . . 7
  - 1.3.3 Advantages of Probabilistic Programming . . . 9
  - 1.3.4 Uses Outside Statistics . . . 10
- 1.4 Automating Inference . . . 11
- 1.5 Problem Statement . . . 13
- 1.6 Guaranteed Bounds . . . 13
- 1.7 Contributions . . . 14
- 1.8 Limitations . . . 16
- 1.9 Notation and Conventions . . . 16

2 Background and Related Work 18
- 2.1 Measure Theory and Probability Theory . . . 19
- 2.2 Probabilistic Programming Languages . . . 22
  - 2.2.1 SGCL: an Imperative PPL . . . 23
  - 2.2.2 Discrete SGCL: a Restricted Fragment of SGCL . . . 26
  - 2.2.3 SPCF: a Functional PPL . . . 26
- 2.3 Semantics of PPLs . . . 29
  - 2.3.1 Transformer Semantics for SGCL . . . 29
  - 2.3.2 Trace Semantics for SPCF . . . 33
- 2.4 Exact Inference . . . 35
- 2.5 Approximate Inference . . . 36
  - 2.5.1 Rejection Sampling . . . 37
  - 2.5.2 Importance Sampling . . . 38

2.5.3 Markov Chain Monte Carlo . . . 39
2.5.4 Particle-Based Inference . . . 40
2.5.5 Other Methods . . . 41
2.5.6 Statistical Checks for Inference . . . 42
2.6 Formal Methods for Probabilistic Programs . . . 44
2.6.1 Applications to Bayesian Inference . . . 47
2.7 A Brief History of Probabilistic Programming . . . 48

3 Guaranteed Bounds via Trace Abstraction 51
3.1 Introduction . . . 52
3.1.1 Contributions . . . 54
3.1.2 Limitations . . . 54
3.2 Interval Trace Semantics . . . 55
3.2.1 Interval Arithmetic . . . 55
3.2.2 Interval Traces and Interval SPCF . . . 55
3.2.3 Bounds from Interval Traces . . . 57
3.2.4 Alternative Reduction Rules for Conditionals . . . 59
3.3 Soundness and Completeness . . . 60
3.3.1 Infinite Trace Semantics . . . 60
3.3.2 Soundness . . . 61
3.3.3 Symbolic Execution . . . 62
3.3.4 Completeness . . . 65
3.3.5 Completeness Proof . . . 69
3.4 Weight-aware Interval Type System . . . 74
3.4.1 Interval Types . . . 75
3.4.2 Type System . . . 76
3.4.3 Constraint-based Type Inference . . . 77
3.5 GuBPI and Optimizations for Linear Expressions . . . 78
3.5.1 GuBPI . . . 78
3.5.2 Standard Interval Trace Semantics . . . 80
3.5.3 Linear Interval Trace Semantics . . . 80
3.6 Practical Evaluation . . . 82
3.6.1 Probability Estimation . . . 83
3.6.2 Comparison with Exact Inference . . . 83
3.6.3 Recursive Models . . . 86
3.6.4 Comparison with Statistical Validation . . . 87
3.7 Related Work . . . 88
3.8 Conclusion . . . 90

4 Exact Inference via Generating Functions 92
4.1 Introduction . . . 93
4.1.1 Contributions . . . 93
4.1.2 Limitations . . . 94
4.1.3 Introductory example . . . 94
4.2 Generating Functions . . . 96
4.2.1 Generating Function Semantics . . . 97

4.2.2 Correctness . . . 101
4.2.3 Possible Language Extensions . . . 108
4.3 Implementation and Optimizations . . . 109
4.3.1 Automatic Differentiation and Taylor polynomials . . . 109
4.3.2 Analysis of the Running Time . . . 112
4.3.3 Reducing the Number of Variables . . . 114
4.3.4 Implementation . . . 118
4.4 Empirical Evaluation . . . 121
4.4.1 Comparison with Exact Inference Methods . . . 121
4.4.2 Comparison with Approximate Inference Methods . . . 122
4.4.3 Benchmarks with Infinite-Support Distributions . . . 124
4.5 Related Work . . . 126
4.6 Conclusion . . . 127

5 Guaranteed bounds via Unrolling and Residual Mass 128
5.1 Introduction . . . 129
5.1.1 Contributions . . . 130
5.1.2 Limitations . . . 130
5.2 Lower Bounds via Unrolling . . . 130
5.3 Upper Bounds via Residual Mass . . . 133
5.4 Implementation . . . 135
5.4.1 Discrete SGCL . . . 136
5.4.2 Full SGCL with Generating Functions . . . 136
5.4.3 Overapproximating the Measure Support . . . 137
5.5 Comparison with Previous Work . . . 138
5.6 Conclusion . . . 139

6 Guaranteed Bounds via Eventually Geometric Distributions 140
6.1 Introduction . . . 141
6.1.1 Contributions . . . 142
6.1.2 Limitations . . . 143
6.2 Contraction Invariants . . . 143
6.3 Eventually Geometric Distributions (EGDs) . . . 144
6.4 Geometric Bound Semantics . . . 149
6.4.1 Examples . . . 152
6.5 Theoretical Analysis . . . 157
6.5.1 Decidability . . . 157
6.5.2 Properties of the Solution Set . . . 158
6.5.3 Soundness . . . 159
6.5.4 Loop-free Fragment . . . 162
6.5.5 Sufficient Conditions for the Existence of Bounds . . . 163
6.5.6 Necessary Conditions for the Existence of Bounds . . . 166
6.5.7 Convergence . . . 168
6.6 Implementation . . . 176
6.6.1 Constraint Solving . . . 177
6.6.2 Optimization . . . 178

6.6.3 Our ADAM-based Solver and Optimizer . . . 179
6.7 Empirical Evaluation . . . 179
6.7.1 Applicability of the Geometric Bound Semantics . . . 179
6.7.2 Quality of the Geometric Bounds . . . 182
6.7.3 Comparison with Previous Work . . . 182
6.7.4 Comparison with Residual Mass Bounds . . . 185
6.8 Related Work . . . 187
6.9 Conclusion . . . 188

7 Conclusion **190**
7.1 Future Directions . . . 191

Appendices

A Supplementary Material for Chapter 3 **194**
A.1 Supplementary Material for Section 3.3 . . . 194
A.1.1 Primitive Functions . . . 194
A.2 Supplementary Material for Section 3.4 . . . 196
A.2.1 Soundness . . . 196
A.2.2 Weak Completeness . . . 200
A.2.3 Constraint-based Type Inference . . . 201

B Supplementary Material for Chapter 4 **205**
B.1 Details on the Empirical Evaluation . . . 205
B.1.1 Details on the Benchmarks . . . 205
B.1.2 Additional Results . . . 207

C Supplementary Material for Chapter 6 **211**
C.1 Relation of EGDs and Generating Functions . . . 211

References **214**

Index **231**

List of Abbreviations

AD . . . . . Automatic differentiation
a.e. . . . . . Almost everywhere (i.e. except for a set of measure zero)
a.s. . . . . . Almost surely (i.e. with probability one)
AST . . . . . Almost surely terminating (terminating with probability one)
CDF . . . . . Cumulative distribution function
cpo . . . . . Complete partial order
EGD . . . . . Eventually geometric distribution
ELBO . . . . . Evidence lower bound
ESS . . . . . Effective sample size
GF . . . . . Generating function
GMM . . . . . Gaussian mixture model
HMC . . . . . Hamiltonian Monte Carlo
HMM . . . . . Hidden Markov model
ICDF . . . . . Inverse cumulative distribution function (quantile function)
iff . . . . . If and only if
i.i.d. . . . . . Independent and identically distributed
IPMCMC . . . . Interacting particle Markov chain Monte Carlo
IS . . . . . Importance sampling
KL . . . . . Kullback-Leibler divergence
LHS . . . . . Left-hand side
LMH . . . . . Lightweight Metropolis-Hastings
MCMC . . . . . Markov chain Monte Carlo
MH . . . . . Metropolis-Hastings
NUTS . . . . . No-U-Turn Sampler
PAST . . . . . positive almost surely terminating (i.e. finite expected running time)
PDF . . . . . Probability density function
pGCL . . . . . Probabilistic guarded command language
PGibbs . . . . . Particle Gibbs
PGF . . . . . Probability generating function
PMCMC . . . . Particle Markov chain Monte Carlo
PMF . . . . . Probability mass function

PP . . . . . Probabilistic programming
PPL . . . . . Probabilistic programming language
PPS . . . . . Probabilistic programming system
RHS . . . . . Right-hand side
RMH . . . . . Random-walk Metropolis-Hastings
SBC . . . . . Simulation-based calibration
SGCL . . . . . Statistical guarded command language
SGD . . . . . Stochastic gradient descent
SMC . . . . . Sequential Monte Carlo
SPCF . . . . . Statistical Programming Computable Functions
SVI . . . . . Stochastic variational inference
TVD . . . . . Total variation distance
VI . . . . . Variational inference

List of Notations

General mathematical notation

$[\varphi]$ . . . . . Iverson brackets: $[\varphi] = 1$ if φ is true, 0 otherwise.

$\mathbf{0}$ . . . . . Zero tensor, function, or measure.

$\mathbf{1}$ . . . . . Constant one function.

$[n]$ . . . . . Set $\{0, 1, \ldots, n-1\}$.

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

$\mathbb{R}_{\geq 0}$ . . . . . Non-negative real numbers.

$\mathcal{P}(S)$ . . . . . Power set of set S.

$\dot{-}$ . . . . . Monus operator: $a \dot{-} b := \max\{0, a - b\}$.

$\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\delta}$ . . Vectors in $\mathbb{R}^n$.

$\mathbf{0}_n$ . . . . . Zero vector $(0, \ldots, 0) \in \mathbb{R}^n$.

$\mathbf{1}_n$ . . . . . Ones vector $(1, \ldots, 1) \in \mathbb{R}^n$.

$\boldsymbol{x}[k \mapsto x]$ . . . . Vector $\boldsymbol{x}$ with k-th element replaced by x.

$\boldsymbol{x}^{\boldsymbol{\alpha}}$ . . . . . Monomial $\prod_{i=1}^{n} x_i^{\alpha_i}$.

$\mathbb{R}^{[d_1]\times\cdots\times[d_n]}$ . . . Space of tensors (multidimensional arrays) with n axes, of dimensions $d_1, \ldots, d_n$; each axis k is indexed from 0 to $d_k - 1$.

$\mathbf{P}, \mathbf{Q}, \mathbf{R}, \mathbf{S}, \mathbf{T}$ . . . Tensors (multi-dimensional arrays) in $\mathbb{R}_{\geq 0}^{[d_1]\times\cdots\times[d_n]}$.

$|\mathbf{T}|$ . . . . . Dimensions $(d_1, \ldots, d_n)$ of a tensor $\mathbf{T} \in \mathbb{R}^{[d_1]\times\cdots\times[d_n]}$.

$\mathbf{0}_{[d_1]\times\cdots\times[d_n]}$ . . . Zero tensor in $\mathbb{R}^{[d_1]\times\cdots\times[d_n]}$.

$\boldsymbol{i}$ . . . . . Multi-index $\boldsymbol{i} = (i_1, \ldots, i_n) \in \mathbb{N}^n$.

$|\boldsymbol{i}|$ . . . . . Multi-index sum: $|\boldsymbol{i}| = i_1 + \cdots + i_n$.

$\boldsymbol{i}!$ . . . . . Multi-index factorial: $\boldsymbol{i}! = i_1! \cdots i_n!$.

$\partial^{\boldsymbol{i}} f(\boldsymbol{x})$ . . . . . Partial derivative $\frac{\partial^{i_1}}{\partial x_1^{i_1}} \cdots \frac{\partial^{i_n}}{\partial x_n^{i_n}} f(x_1, \ldots, x_n)$.

$\mathbf{T}_{\boldsymbol{i}}$ . . . . . Element of a tensor $\mathbf{T}$ at multi-index $\boldsymbol{i} = (i_1, \ldots, i_n)$.

$\mathbf{T}_{k:j}$ . . . . . Subtensor of $\mathbf{T}$ at index j along axis k.

Probability theory and statistics

Σ_Ω . . . . . A σ-algebra on the set Ω.

μ, ν . . . . . Measures.

λ_n . . . . . Lebesgue measure on $\mathbb{R}^n$.

$\mathrm{Meas}(\Omega)$ . . . . Space of measures on a set Ω.

$\mu|_A$ . . . . . Restriction of measure μ to a set A.

$f_*(\mu)$ . . . . . Pushforward measure of μ along f.

pdf_μ . . . . . Probability density function of a probability measure μ.

cdf_μ . . . . . Cumulative distribution function of a probability measure μ on $\mathbb{R}$.

icdf_μ . . . . . Quantile function of a probability measure μ on $\mathbb{R}$.

$\mathrm{supp}(\mu)$ . . . . . Support of measure μ on Ω: $\mathrm{supp}(\mu) = \{x \in \Omega \mid \mu(N) > 0 \text{ for all neighborhoods } N \text{ of } x\}$.

$\int f(x)\,\mu(\mathrm{d}x)$ . . Lebesgue integral of f with respect to measure μ.

$\frac{\mathrm{d}\nu}{\mathrm{d}\mu}$ . . . . . Radon-Nikodym derivative of ν with respect to μ.

$\mathbb{P}[E]$ . . . . . Probability of event E.

X, Y, Z . . . . . Random variables.

$X \sim D$ . . . . . Random variable X has distribution D.

$\mathbb{E}[X]$ . . . . . Expected value of random variable X.

$\mathbb{V}[X]$ . . . . . Variance of random variable X.

$\mathrm{Skew}[X]$ . . . . Skewness of random variable X.

$\mathrm{Kurt}[X]$ . . . . . Kurtosis of random variable X.

Syntax & semantics of the probabilistic programming languages SGCL & SPCF

$\boldsymbol{X}$ . . . . . Set of variables $(X_1, \ldots, X_n)$ of an SGCL program.

$\lightning$ . . . . . Failure state.

$\mathbb{S}$ . . . . . State space $\mathbb{R}^n_{\geq 0}$.

$\mathbb{S}_\lightning$ . . . . . Extended state space $\mathbb{S} \cup \{\lightning\}$.

$\mathbb{N}^n$ . . . . . Discrete state space.

$\mathbb{N}^n_\lightning$ . . . . . Extended discrete state space $\mathbb{N}^n \cup \{\lightning\}$.

$\preceq$ . . . . . Partial order on measures.

$\sqsubseteq$ . . . . . Partial order on measure transformers.

$\mathrm{lfp}(f)$ . . . . . Least fixed point of f.

$[\![P]\!]$ . . . . . Simplified standard semantics of an SGCL program P.

$[\![P]\!]$ . . . . . Unnormalized distribution of an SPCF program P.

$[\![P]\!]_\lightning$ . . . . . Standard semantics of an SGCL program P.

normalize . . . . Normalization function for SGCL or SPCF.

$\mathbf{R}$ . . . . . Type of real numbers in SPCF.

$\alpha \to \beta$ . . . . . Function type from α to β.

$\Gamma \vdash t : \tau$ . . . . . Typing judgment ("t has type τ in context Γ").

$M[N/x]$ . . . . . Term resulting from substituting the term N for the variable x in the term M.

$\underline{r}, \underline{f}$ . . . . . Literals in SPCF.

$\to$ . . . . . Reduction relation in SPCF.

$\mathbb{T}$ . . . . . Trace space $\bigcup_{n\in\mathbb{N}}[0,1]^n$.

$\mu_{\mathbb{T}}$ . . . . . Measure on $\mathbb{T}$ given by $\mu_{\mathbb{T}}(U) := \sum_{n\in\mathbb{N}} \lambda_n(U \cap [0,1]^n)$.

$\boldsymbol{s}$ . . . . . Trace in $\mathbb{T}$.

$\langle r_1, \ldots, r_n \rangle$ . . . Notation for a trace containing the values $r_1, \ldots, r_n$.

$\boldsymbol{s}\boldsymbol{s}'$ . . . . . Concatenation of traces $\boldsymbol{s}$ and $\boldsymbol{s}'$.

$r\,\boldsymbol{s}$ . . . . . Prepending a value r to a trace $\boldsymbol{s}$.

$E[\cdot]$ . . . . . Evaluation context in SPCF.

val_P . . . . . Value function of an SPCF program P.

wt_P . . . . . Weight function of an SPCF program P.

Notation in Chapter 3

$\mathbb{I}_X$ . . . . . Set of intervals with endpoints in X.

$\mathbb{I}$ . . . . . Set of all intervals on $\mathbb{R}$.

$\sqsubseteq$ . . . . . Interval inclusion.

$\sqcap$ . . . . . Meet (greatest lower bound) of two intervals with respect to $\sqsubseteq$.

$\sqcup$ . . . . . Join (least upper bound) of two intervals with respect to $\sqsubseteq$.

$\bot$ . . . . . Bottom element of the interval lattice, representing $\varnothing$.

$f^{\mathbb{I}}$ . . . . . Lifting $f^{\mathbb{I}} : \mathbb{I}^n \to \mathbb{I}$ of a function $f : \mathbb{R}^n \to \mathbb{R}$ to operate on intervals.

$\mathbb{T}_{\mathbb{I}}$ . . . . . Interval trace space $\bigcup_{n\in\mathbb{N}}(\mathbb{I}_{[0,1]})^n$.

$\boldsymbol{t}$ . . . . . Interval trace in $\mathbb{T}_{\mathbb{I}}$.

$I\,\boldsymbol{t}$ . . . . . Prepending an interval I to an interval trace $\boldsymbol{t}$.

$\triangleleft$ . . . . . Refinement relation between concrete and interval traces or SPCF and Interval SPCF terms.

$(\!|\boldsymbol{t}|\!)$ . . . . . Set of traces refining an interval trace $\boldsymbol{t}$: $\{\boldsymbol{s} \in \mathbb{T} \mid \boldsymbol{s} \triangleleft \boldsymbol{t}\}$.

$\underline{[a,b]}$ . . . . . Interval literal in Interval SPCF.

$\to_{\mathbb{I}}$ . . . . . Interval-based reduction relation.

$\mathsf{val}_P^{\mathbb{I}}$ . . . . . Interval value function of an (interval) SPCF program P.

$\mathsf{wt}_P^{\mathbb{I}}$ . . . . . Interval weight function of an (interval) SPCF program P.

$\mathsf{vol}(\boldsymbol{t})$ . . . . . Volume of an interval trace $\boldsymbol{t}$.

$\mathsf{lowerBd}_P^{\mathcal{T}}$ . . . . Lower bound on $[\![P]\!]$ for a set $\mathcal{T}$ of interval traces.

$\mathsf{upperBd}_P^{\mathcal{T}}$ . . . . Upper bound on $[\![P]\!]$ for a set $\mathcal{T}$ of interval traces.

$\mathbb{T}_\infty$ . . . . . Infinite trace space $[0,1]^\omega$.

$\mu_{\mathbb{T}_\infty}$ . . . . . Measure on $\mathbb{T}_\infty$ given by $\mu_{\mathbb{T}_\infty}(U \times \mathbb{T}_\infty) = \lambda_n(U)$ when $U \in \Sigma_{[0,1]^n}$.

$cover(\boldsymbol{t})$ . . . . . Set of infinite traces covered by $\boldsymbol{t}$: $(\!|\boldsymbol{t}|\!) \times \mathbb{T}_\infty$.

$\mathcal{V}, \mathcal{W}$ . . . . . Symbolic values.

$\mathcal{M}, \mathcal{N}, \mathcal{P}$ . . . . Symbolic terms.

Δ . . . . . Set of symbolic constraints.

Ξ . . . . . Set of symbolic values.

Ψ . . . . . Symbolic path.

$\rightsquigarrow$ . . . . . Symbolic reduction.

$\mathcal{V}[\boldsymbol{s}/\overline{\alpha}]$ . . . . . Result of replacing the sample variables $\overline{\alpha}$ with the values $\boldsymbol{s}$ in $\mathcal{V}$.

$\mathrm{Sat}_n(\Delta)$ . . . . . Set of values for n sample variables satisfying all constraints in Δ.

$[\![\Psi]\!]$ . . . . . Denotation of a symbolic path Ψ.

$\Subset$ . . . . . Tight subset inclusion, i.e. the difference is a null set.

σ . . . . . Weightless interval type.

$\mathcal{A}$ . . . . . Weighted interval type.

$\left\{ \begin{matrix} \sigma \\ I \end{matrix} \right\}$ . . . . . Weighted interval type with weight I.

$\sqsubseteq_\sigma$ . . . . . Subtype relation for weightless interval types.

$\sqsubseteq_{\mathcal{A}}$ . . . . . Subtype relation for weighted interval types.

Notation in Chapter 4

$\mathrm{Taylor}^d_{\boldsymbol{w}}(f)$ . . . Taylor polynomial of $f : \mathbb{R}^n \to \mathbb{R}$ at $\boldsymbol{w} \in \mathbb{R}^n$ of order d.

$\mathrm{gf}(\mu)$ . . . . . Generating function of a distribution μ.

$G|_E$ . . . . . Restriction of a generating function G to an event E.

$[\![P]\!]_{\mathrm{gf}}$ . . . . . Generating function semantics of an SGCL program P.

$\mathrm{normalize}_{\mathrm{gf}}$ . . . Normalization function for generating functions.

Notation in Chapters 5 and 6

$P^{(u)}$ . . . . . u-fold unrolling of an SGCL program P.

$[\![P]\!]_{\mathrm{lo}}$ . . . . . Lower bound semantics of an SGCL program P.

$[\![P]\!]_{\mathrm{res}}$ . . . . . Residual mass bound of an SGCL program P.

$[\![P]\!]^{\lightning}_{\mathrm{gf}}$ . . . . . Extended generating function semantics of an SGCL program P.

$\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ . . . Eventually geometric distribution with initial block $\mathbf{P}$ and decay rates $\boldsymbol{\alpha}$.

$\preceq_{\mathrm{EGD}}$ . . . . . Order on EGDs

$\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_E^{\text{geo}}$.	Geometric bound semantics of an SGCL event E applied to an EGD $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$.
$[\![P]\!]^{\text{geo}}$	Geometric bound semantics of an SGCL program P.
Join	Join relation for the geometric bound semantics.
Join*	Strict join relation for the geometric bound semantics.

All models are wrong but some are useful.

— George E. P. Box

1 Introduction

Contents

1.1 Probabilistic Modeling 1
1.2 Statistical Inference: Frequentism & Bayesianism 3
1.3 Probabilistic Programming 6
1.3.1 Graphical Models 6
1.3.2 Probabilistic Programming Languages 7
1.3.3 Advantages of Probabilistic Programming 9
1.3.4 Uses Outside Statistics 10
1.4 Automating Inference 11
1.5 Problem Statement 13
1.6 Guaranteed Bounds 13
1.7 Contributions 14
1.8 Limitations 16
1.9 Notation and Conventions 16

Summary The purpose of this introductory chapter is to motivate and provide context for basic ideas in probabilistic modeling, statistics, and probabilistic programming, to present the problem statement, and to outline the contributions of this thesis.

1.1 Probabilistic Modeling

Our world is full of uncertainty: in most situations, we have to learn from and make decisions based on incomplete or imperfect information. This may be a fundamental aspect of nature: according to our current understanding of physics, some quantum processes like radioactive decay are truly random. In many other cases, handling uncertainty is a practical necessity due to our ignorance of underlying processes, imprecise measurements, and limited computational power, as exemplified by simple coin flips and dice rolls, or complex behavior like that of humans. As such, probabilistic models have been applied in many fields, such as social, political, and medical sciences, psychology, physics, biology, economics, and computer science.

Aside from its necessity, randomness can also be an asset: for many games, e.g. Rock-paper-scissors, the optimal strategies are stochastic, and some distributed consensus problems cannot be solved without randomization [Fischer et al. 1983; Ben-Or 1983]. Randomness also frequently helps solve computational problems faster and avoid worst-case scenarios: a classic example is randomized quicksort. Moreover, approximate randomized algorithms are essential for data analysis in large distributed systems [Heule et al. 2013]. Another application of randomization is to protect the privacy of individuals: such *differential privacy* techniques were used in the US census of 2020 [Garfinkel and Leclerc 2020].

In all these cases, the only viable option is to turn to ***probability theory*** to describe the uncertainty/randomness in a mathematically rigorous way. The *interpretation* of probability itself has been subject to significant philosophical debate [DeGroot and Schervish 2018, Section 1.2]. We briefly discuss the two most common interpretations, which influence the main paradigms of statistics.

The *frequency interpretation* of the probability of an outcome of a process is the relative frequency of that outcome if the process were repeated a "large number" of times "under similar conditions". For example, the probability of a fair coin flip coming up heads is $\frac{1}{2}$ because if we were to toss the coin repeatedly, we'd expect heads about half the time. This definition has problems in terms of vagueness and it only applies to processes that can be repeated many times, which is not always possible even in principle.

The *subjective interpretation* of the probability of an outcome represents a person's own judgment of the likelihood of the outcome. In other words, it quantifies the person's degree of belief in the outcome. There are also problems with this definition of course: it is very difficult, if even possible, for a human to have consistent beliefs about a large number of possible scenarios; and there is no "objective" basis for two people to agree on a common evaluation of the state of knowledge about (an aspect of) the world. On the other hand, one can argue that there may not always be an "objective" answer – even scientists sometimes disagree about the likelihood of uncertain outcomes.

Fortunately, these philosophical debates are irrelevant to the formal mathematical study of uncertainty: probability theory is indifferent to the interpretation of probabilities. So we will not discuss this matter further. The two interpretations have led to the two main paradigms in statistics however: *frequentism* and *Bayesianism*.

While probability theory is concerned with the description of random processes, the aim of (inferential) ***statistics*** is to *learn* about the underlying process from observations and collected data. Statistics is a broad field with many practical considerations such as experimental design, survey planning, data collection, data presentation, and data summarization (descriptive statistics). We focus on ***statistical inference***, which draws conclusions about the statistical model of the process that produces the data. Such conclusions can take many forms: parameter estimation, hypothesis testing, and prediction, among others.

Machine learning, like statistics, is about data analysis, but differs in its focus and applications. Whereas the purpose of classical statistics is to understand the data generating process and relationships between its variables, the primary goal of machine learning is accurate prediction based on much larger datasets. As a trade-off, machine

learning models such as neural networks are typically less interpretable than classical statistical results.

Statistical and machine learning models use the following concepts and notation: θ often denotes the *parameters*, i.e. the unobservable quantities about a system or experiment, and y denotes the *observed data*, which is usually a vector of several data points $y = (y_1, \ldots, y_n)$. In supervised learning or prediction problems, the data is split into two parts (x, y): the *inputs* x (or "explanatory variables") explain or predict the *outputs* y (or "response/outcome variables"). For example, in a clinical trial, θ could be the probabilities of survival under each treatment for randomly chosen members of the population, x could be the age and previous health of each patient, and y could be the number of survivors in each treatment group [Gelman et al. 2013, Section 1.2]. In a machine learning context, x could be images and y could be the labels of the depicted objects.

There are broadly two approaches to modeling in machine learning, in particular for supervised learning: discriminative and generative [Bishop 2006, Section 1.5.4]. ***Discriminative approaches*** aim to learn a predictive model directly, by optimizing its parameters for minimal prediction error. Such a predictive model is usually a parameterized mapping f_θ from inputs x to outputs y, often by maximizing the predicted probability $p_\theta(y \mid x)$ over outputs for a given input x. A classic example of this is logistic regression. To predict the output $\tilde{y}$ at a new input $\tilde{x}$, we just apply the learned function: $\tilde{y} = f_\theta(\tilde{x})$. The discriminative concept has been extraordinarily successful, as exemplified by the recent breakthroughs in image classification with convolutional neural networks [Krizhevsky et al. 2012] and machine translation with transformers [Vaswani et al. 2017]. On the other hand, they typically require enormous amounts of data and struggle to quantify the confidence/certainty in their predictions.

By contrast, a ***generative model*** describes the data generating process as a probabilistic model, which can then be used to explain the observations. More specifically, it models the full distribution $p_\theta(y)$ of the data. In the case of supervised learning, this means the *joint* distribution $p_\theta(x, y)$ of inputs x and outputs y, rather than merely a conditional distribution $p_\theta(y \mid x)$, as in discriminative methods. A classic example of this is the naive Bayes classifier [Ng and Jordan 2001]. As a consequence, generative approaches make stronger assumptions about the problem. This can be a blessing and a curse: such model assumptions can be wrong and become less important as the amount of data increases; but they also allow for the incorporation of domain-specific expertise, which is often a requirement for effective performance in applications with limited or restricted data. This thesis is only concerned with generative modeling.

1.2 Statistical Inference: Frequentism & Bayesianism

There are two main paradigms in statistical inference: frequentism and Bayesianism, which are inspired by the two interpretations of probability mentioned above. In a nutshell, frequentists assume the model parameters are fixed while Bayesians assume the observations are fixed. This is explained in more detail below.

The classical or ***frequentist*** approach to statistics is based on the idea that the parameters of the model are fixed but unknown, and the data is a random sample. A

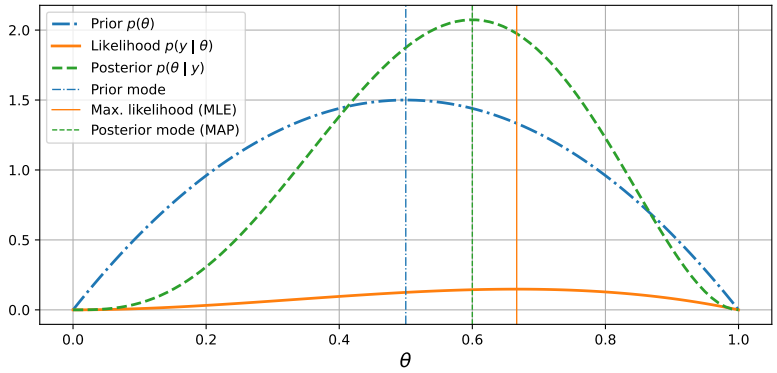


Figure 1.1: Prior density, likelihood function, and posterior density for the Bayesian coin model (Example 1.2.1).

key focus is on repeatability and generalization to other observations that could have been made if the experiment were repeated. It relies only on the ***likelihood*** function $p(y \mid \theta)$, which characterizes how likely the data is for given parameter values. For instance, the very common maximum likelihood estimator (MLE) picks the parameter value that maximizes the likelihood function. Besides such point estimates, standard tools in frequentist statistics include confidence intervals and hypothesis tests.

In the ***Bayesian paradigm***, the model specifies the joint distribution[1] $p(\theta, y)$ of the parameters θ and the data y. (In particular, the Bayesian approach requires a generative model.) This distribution is decomposed into a product of a ***prior*** distribution $p(\theta)$, i.e. our initial belief about the parameters before seeing any data, and the likelihood function $p(y \mid \theta)$. The prior and likelihood can be combined into a ***posterior*** distribution $p(\theta \mid y)$, which represents our updated belief about the parameters after seeing the data. The posterior is computed using ***Bayes' rule***:

$$p(\theta \mid y) = \frac{p(\theta) \cdot p(y \mid \theta)}{p(y)} \tag{1.1}$$

The denominator $p(y)$ is a normalizing constant known as the ***marginal likelihood*** or ***evidence***, which is the probability of the data averaged over all possible parameter values. An important result about Bayesian inference concerns its behavior in the limit of infinite data: the Bernstein-von-Mises theorem states, roughly speaking, that the posterior distribution converges to a Gaussian distribution centered around the true parameter value with asymptotically vanishing variance, provided the model is correctly specified and the prior is not too restrictive [van der Vaart 1998, Section 10.2]. While Bayesian inference emphasizes the posterior distribution as a measure of uncertainty about the true parameters θ, there are Bayesian equivalents of frequentist concepts like point estimates, confidence intervals, and hypothesis tests.

Example 1.2.1. Suppose we have a coin with unknown bias θ (probability for heads) and we flip it 3 times, resulting in the observations $y_1 = 1$, $y_2 = 0$, and $y_3 = 1$ where 1 means

[1] Technically, $p(\theta, y)$ is a probability density, but this distinction is often ignored for convenience in statistics and machine learning. We will make this more precise in Chapter 2.

heads and 0 means tails. The likelihood function is $p(y \mid \theta) = \theta^{y_1+y_2+y_3}(1-\theta)^{3-y_1-y_2-y_3} = \theta^2(1-\theta)$. The maximum likelihood estimator is, unsurprisingly, $\hat{\theta} = \frac{2}{3}$.
Suppose we have a weak prior belief that the coin is fair. We could express this with a Beta$(2,2)$ prior on θ, i.e. $p(\theta) = 6 \cdot \theta(1-\theta)$, whose mode and mean are $\frac{1}{2}$. Then the joint density is $p(\theta, y) = 6 \cdot \theta^{y_1+y_2+y_3+1}(1-\theta)^{4-y_1-y_2-y_3} = 6 \cdot \theta^3(1-\theta)^2$ and we obtain the posterior distribution $p(\theta \mid y) = 60 \cdot \theta^3(1-\theta)^2$ with the normalizing constant $p(y) = \frac{1}{10}$. Its posterior mean is $\frac{4}{7}$, which was updated upward from the prior mean of $\frac{1}{2}$ due to the larger number of heads in the data. At the same time, the posterior variance decreased to $\frac{3}{98}$ compared to the prior variance of $\frac{1}{20}$, reflecting the increased certainty about the bias. The mode of the posterior distribution, i.e. the maximum a posteriori estimator (MAP), is $\hat{\theta} = \frac{3}{5}$, which is less extreme than the MLE of $\frac{2}{3}$ due to the (weak) prior belief in fairness. A visualization of the prior, likelihood, and posterior is shown in Fig. 1.1.

It is instructive to view the two paradigms in the light of decision theory [Robert 2007, Section 2.3]. Suppose we have observations y and there is a true parameter θ that generated the data but that we do not know. We want to make decisions based on the data, i.e. there is a decision rule $d(y)$ that maps the data to an action. Since we are interested in statistical inference, the decision rule will be an estimator and the action will be an estimate $\hat{\theta}$ of θ. An action/estimate $\hat{\theta}$ has a loss (i.e. negative utility) $L(\hat{\theta}, \theta)$ associated with it, which depends on the true parameter θ. Both frequentist and Bayesian decision theory aim to minimize the loss $L(d(y), \theta)$ over all decision rules d. Unfortunately, this term is not just a function of d, but also of y and θ, so we have to reduce this to a single number. Frequentists average over y, i.e. they try to minimize the *frequentist risk* $R(d, \theta) := \mathbb{E}_{y \sim p(y|\theta)}[L(d(y), \theta)]$ over all possible data distributions, while Bayesians average over θ, i.e. choose the estimate $d(y)$ that minimizes the *posterior expected loss* $r(d, y) := \mathbb{E}_{\theta \sim p(\theta|y)}[L(d(y), \theta)]$ over the posterior distribution of the parameter given the data. The frequentist risk $R(d, \theta)$ is still a function of θ, so there is usually no uniformly optimal decision rule unless the space of possible decision rules is restricted (as in the case of uniformly minimum-variance unbiased estimators). Another option is the minimax criterion, which aims to find the decision rule that minimizes $\max_\theta R(d, \theta)$. In contrast, $r(d, y)$ is a function of y, which is known! So in the Bayesian case, there is an optimal decision rule, which is called the *Bayes estimator*. Morally speaking, the Bayesian approach is "optimistic" in that it optimizes the *expected* loss rather than the *worst-case* loss over all possible parameter values as in the frequentist minimax criterion.

Both paradigms have their strengths and weaknesses. Frequentist methods are more objective, have worst-case guarantees, ensure repeatability, and are often computationally simpler. On the other hand, they depend on the experimental procedure for repeatability, which leads to widespread misunderstandings of its concepts like confidence intervals and p-values. Another criticism of frequentism is its reliance on point estimates and lack of a coherent framework for uncertainty quantification. The Bayesian approach is more flexible, as it allows for incorporating prior knowledge and domain expertise in a principled way. This is often necessary for effective modeling and makes it applicable to situations with less data. On the other hand, the choice of prior is subjective and can be difficult in practice, although this problem is somewhat mitigated by *non-informative priors*. It should be noted that no statistical method is free from subjectivity: experimental design, model selection, and the choice of which data

to include are all subjective decisions shared by both paradigms [Gelman et al. 2013, Section 1.5]. While the Bayesian approach offers principled uncertainty quantification, it obviously does not account for model misspecifications or *unknown unknowns* and may thus underestimate the uncertainty [Rainforth 2017, Section 3.4.2]. Another concern is about generalization and repeatability: pure Bayesianism rejects empirical checks like cross validation. Finally, Bayesian inference is both conceptually harder and more computationally expensive than frequentist inference.

In practice, which paradigm to use depends on the application and the problem characteristics. For example, Bayesian modelling is preferable for small datasets with strong prior knowledge. The two methods are also not mutually exclusive: Gelman et al. [2013] advocate for a combination of Bayesian modeling and inference with frequentist techniques for checking model fit, such as posterior predictive checks.

In this thesis, we are almost exclusively concerned with Bayesian inference for generative probabilistic models: the challenges of Bayesian model specification and inference were the main motivation for the development of probabilistic programming systems, and this is where we turn next.

1.3 Probabilistic Programming

How should the probabilistic models from Section 1.1 be expressed? In this thesis, we work with *probabilistic programs* as unambiguous and flexible specifications of probabilistic models, which allow for automated analysis and inference. The benefits of probabilistic programming in machine learning are also highlighted by Ghahramani [2015]. An early overview of the subject can be found in Gordon et al. [2014], and a more thorough/textbook-style introduction in Rainforth [2017] and van de Meent et al. [2018]. This section introduces probabilistic programming informally. A more formal treatment will be given in Chapter 2, which also reviews existing probabilistic programming systems. Before diving into the details, we discuss alternative specifications of probabilistic models.

The most obvious and traditional way to describe a probabilistic model is to write down a mathematical expression for its probability distribution, typically in terms of its density function. Though very general and flexible, this approach is often cumbersome and error-prone: the density function of complex models can be difficult to write down and is typically long. Moreover, the density function may be hard to understand and interpret, and one cannot easily compute samples from it, which goes counter to the spirit of generative modeling.

1.3.1 Graphical Models

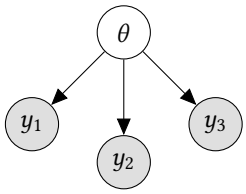
A popular alternative are ***probabilistic graphical models (PGMs)*** [Bishop 2006, Chapter 8]. They exploit the fact that many variables in a probabilistic model are (conditionally) independent of one another. The density function thus decomposes into factors, whose dependency structure is represented by a graph where the nodes are the variables and the edges represent *direct* dependencies between them. Note that the absence of an edge does not imply independence. There are two main types of

PGMs: Markov random fields (undirected graphs) and Bayesian networks (directed acyclic graphs).

Bayesian networks are also known as *Bayes nets*, *belief networks*, *decision networks*, or *directed acyclic graphs (DAGs)*[2]. We focus on this class of graphical models because it is closer to the generative idea and is a precursor to probabilistic programming. Let $x = (x_1, \ldots, x_n)$ be the random variables of the model, i.e. the nodes of the graph, and $\text{parents}(x_i)$ be the set of nodes with an edge into x_i. Then the graphical model corresponds to the factorization of the joint distribution as $p(x) = \prod_{i=1}^{n} p(x_i \mid \text{parents}(x_i))$. So the description of the complex joint distribution is reduced to the conditional distributions of each variable given its parents. These conditional distributions are usually well-known probability distributions whose parameters are determined by the parent variables. For example, it could be a normal distribution with the mean and variance given by the parents. In particular, it is easy to sample from the joint distribution: we just sample from the conditional distributions in topological order (sometimes called *ancestral sampling*). Such a Bayesian network is often written using the following notation:

$$\begin{aligned} x_1 \mid \text{parents}(x_1) &\sim D_1(\text{parents}(x_1)) \\ &\vdots \\ x_n \mid \text{parents}(x_n) &\sim D_n(\text{parents}(x_n)) \end{aligned}$$

where each D_i is a distribution with parameters $\text{parents}(x_i)$. For instance, the Bayesian coin model (Example 1.2.1) can be expressed as a Bayesian network with three Bernoulli-distributed variables y_1, y_2, y_3 and a Beta-distributed variable θ as their parent.



$$\begin{aligned} \theta &\sim \text{Beta}(2, 2) \\ y_1 \mid \theta &\sim \text{Bernoulli}(\theta) \\ y_2 \mid \theta &\sim \text{Bernoulli}(\theta) \\ y_3 \mid \theta &\sim \text{Bernoulli}(\theta) \end{aligned}$$

Similar notation is used throughout the statistical literature, in particular for hierarchical models [Gelman et al. 2013]. While immensely useful, Bayesian networks have expressivity limitations: the structure of the model is fixed (it cannot depend on the variables), which makes it impossible to express nonparametric models with a varying or unbounded number of variables.

1.3.2 Probabilistic Programming Languages

Probabilistic programming (PP) is a more general and flexible approach to generative probabilistic modeling and Bayesian inference.[3] It is sometimes called ***Bayesian***

[2] The term "DAG" is popular in statistics, which is unfortunate because it has a much broader meaning in computer science.

[3] Technically, there is no reason for probabilistic programming to be tied to the Bayesian paradigm. In fact, the Stan language has support for maximum likelihood estimation [Carpenter et al. 2017]. But the overwhelming majority of probabilistic programming languages are designed for Bayesian inference, so we will focus on this aspect.

probabilistic programming [P. Wang et al. 2024], ***statistical probabilistic programming*** [Vákár et al. 2019], or even ***Bayesian statistical probabilistic programming***. The basic idea is to express probabilistic models as computer programs written in a ***probabilistic programming language (PPL)***. Such programs can make use of general-purpose programming constructs like branching, loops, and recursion, increasing their expressiveness compared to graphical models. Compared to ordinary programming languages, PPLs have three additional concepts:

1. **Random sampling:** a sample construct to draw a sample from a distribution. This corresponds to the prior $p(\theta)$ in a Bayesian model.
2. **Observations:** an observe construct to record the likelihood of an observed data point. This corresponds to the likelihood $p(y \mid \theta)$ in a Bayesian model.
3. **Inference:** a normalize construct to infer the posterior distribution. This corresponds to the division by the normalizing constant $p(y)$ in Bayes' rule. We will usually leave this normalization *implicit*, see the discussion below.

Random sampling Obviously, a probabilistic program needs to be able to draw random samples. Most general-purpose programming languages provide access to a (pseudo-)random number generator, with varying support for different distributions. This is essential for many applications where randomness is desired (see Section 1.1), such as simulations and randomized algorithms. In the fields of randomized algorithms and program analysis, randomization is enough to qualify as a probabilistic programming language. To distinguish it from our definition, we will sometimes call PPLs with sample but without observe and normalize ***randomized programming*** or ***non-statistical probabilistic programming*** languages, although these terms are not standard.

Observations In the fields of Bayesian statistics and machine learning, randomization alone is not enough: while it allows for the specification of a generative model, observations are needed to condition on data and to perform Bayesian inference. Whereas sample is used to construct the prior distribution, observe is used to specify the data points and their likelihood.

Inference With sample for the prior and observe for the likelihood, probabilistic programs can describe the joint distribution of a model. The final step is to compute the posterior distribution, i.e. to perform Bayesian inference. A ***probabilistic programming system (PPS)*** comes with an inference engine that supports various general-purpose inference algorithms for this task (see Section 1.4). The language construct responsible for inference has various names across PPLs: besides normalize, it is also called infer, query, or conditional. In fact, most languages leave it implicit: inference is performed on the joint distribution described by the whole program. Explicitly normalizing the distribution of only a part of a program is called ***nested inference***, which has applications in Bayesian experimental design and the simulation of agents reasoning about other agents [Rainforth 2017, Chapter 10]. However, nested inference is a fairly niche topic and we will ignore it in this thesis. Instead, we will assume that the normalize construct is implicitly invoked on the whole program, unless otherwise stated.

```
count := 0;
while (sample Bernoulli(p) = 0) {
  count += 1;
}
```

(a) Sampler for Geometric(p) (not statistical/ Bayesian)

```
normalize {
  θ := sample Beta(2, 2);
  observe y1 ~ Bernoulli(θ);
  observe y2 ~ Bernoulli(θ);
  observe y3 ~ Bernoulli(θ);
}
```

(b) Example 1.2.1 (Bayesian coin model)

Figure 1.2: Examples of probabilistic programs

Examples A precise description of the PPLs studied in this thesis will be given in Chapter 2. To give an intuitive idea of what probabilistic programs may look like, we provide two simple examples (Fig. 1.2). The first is a classic non-statistical example: sampling from a geometric distribution with success probability p given only access to a coin with probability p of coming up heads. This can be achieved by flipping the coin repeatedly until it comes up heads, which is expressed with a loop in the probabilistic program (Fig. 1.2a). The second example demonstrates how to express the Bayesian model from Example 1.2.1 as a probabilistic program (Fig. 1.2b). It first samples θ from the prior Beta$(2, 2)$ and then records the three observed coin flips y_1, y_2, y_3. Just this once, for clarity, we included the top-level normalize construct, which normalizes the joint distribution to the posterior distribution, but we will leave it implicit from now on.

Probabilistic programming offers several advantages over traditional methods of probabilistic modeling, which we will discuss now.

1.3.3 Advantages of Probabilistic Programming

Unambiguous accessible specification Probabilistic programs can be given a mathematically precise semantics (see Chapter 2), so there can never be ambiguity about their meaning. This is a clear advantage over the mix of prose and equations that is traditionally used in the literature. Even compared to graphical models, this offers advantages: all the information about the model is contained in the program, whereas the specification of the conditional distributions of a Bayesian network suffers from the same problems around ambiguity. Another advantage of programs as specifications is that they are arguably more accessible than statistical notation to practitioners with a programming background, which is increasingly common in many fields, most certainly in computer science.

Expressive power Some of the most influential probabilistic programming languages (PPLs) used in practice are *universal*, i.e. the underlying deterministic language is Turing-complete. Using stochastic branching and loops or recursion, universal PPLs can express arbitrarily complex models, such as probabilistic context free grammars [Manning

and Schütze 1999], statistical phylogenetics [Ronquist et al. 2021], and even physics simulations [Baydin et al. 2019].

Automating inference A major goal of probabilistic programming is to decouple specification and inference. In the past, practitioners were limited to a small set of Bayesian models for which the posterior distribution had already been worked out, or they had to derive it by hand in tedious and error-prone calculations. Probabilistic programming systems automate the inference part, which reduces the burden on the modeler and facilitates experimentation with novel or more complex models that are tailored to their applications. It also makes Bayesian modeling more accessible to non-experts, as evidenced by the success of the probabilistic programming system *Stan* [Carpenter et al. 2017] in applied disciplines.

Program analysis and formal verification Besides inference, the precise specification of a model as a program enables other forms of automated analysis. In particular, formal methods can be used to verify properties of probabilistic programs, such as (conditional) independence of variables and necessary conditions for certain inference algorithms to be applicable. Beyond verification, automated program analyses and program transformations can improve inference quality and efficiency. We discuss this in more detail in Section 2.6.

Advancing probability theory On a more abstract level, research into the semantics of probabilistic programs with higher-order functions and recursion has contributed to new insights into the foundations of probability theory and connections with category theory. For example, it has led to the notion of *quasi-Borel spaces* [Heunen et al. 2017], which are an alternative to the standard notion of measure spaces and avoid measure-theoretic problems around function spaces.

1.3.4 Uses Outside Statistics

Apart from being a specification language for statistical models, probabilistic programming can be used for other purposes. We touched on this in Section 1.3.2 and provide more details below.

Non-uniform random number generation Typically, random number generators produce independent and uniformly distributed bits. Many applications require random numbers from other distributions, such as a Bernoulli or a Poisson distribution. A sampler for such a distribution needs to transform the input randomness into the desired shape. It is natural to implement such samplers in a probabilistic programming language, which makes them amenable to formal analysis and mechanized correctness proofs as in Hurd [2003]. An example of such a sampler for the geometric distribution is given in Fig. 1.2a, assuming access to Bernoulli-distributed random numbers.

Bayesian inference algorithms Not just samplers for standard distributions, but also sampling algorithms for more complex distributions can be implemented in a probabilistic programming language. The Gen language [Cusumano-Towner et al. 2019] advocates for *programmable inference*, where the Bayesian inference algorithm for a statistical model is expressed in the same language and thus can be customized by the user.

Randomized algorithms The non-statistical version of probabilistic programming can be used to implement randomized algorithms. The promise of this idea was already recognized by Kozen [1981]. Such randomized algorithms fall into two categories: Las Vegas and Monte Carlo algorithms. ***Las Vegas algorithms*** always produce the correct result, but their running time is random. A classic example is randomized quicksort, which has an expected running time of $O(n \log n)$ instead of $O(n^2)$ for the deterministic version. On the other hand, ***Monte Carlo algorithms*** only produce the correct result with positive (usually high) probability. Classic examples include the *Miller-Rabin primality test* [Rabin 1980], whose output is always correct for prime numbers but may be incorrect for composite numbers with low probability, and *Freivalds' algorithm* to verify matrix multiplication with high probability [Freivalds 1979], which has quadratic time complexity and is strictly faster than the fastest known deterministic algorithm with complexity $O(n^{2.373})$. A more recent example is the *HyperLogLog algorithm* for approximating the number of distinct elements in a multiset with space complexity $O(\log \log n)$ [Flajolet et al. 2007], which is widely used in large distributed systems [Heule et al. 2013]. There has been a lot of work on program analysis and verification of randomized algorithms expressed as probabilistic programs, which we discuss further in Chapter 2.

The main focus of this thesis is on the statistical version of probabilistic programming for Bayesian modeling. Note that all results and techniques developed in this thesis are also applicable to non-statistical probabilistic programs.

1.4 Automating Inference

How can we calculate the posterior distribution of a statistical model? At first glance, this question seems trivial because Bayes' rule gives us a simple formula for the posterior distribution. However, ***exact inference***, i.e. computing the exact posterior distribution, faces several major challenges.

Computing the joint distribution While the joint distribution $p(\theta, y)$ takes a simple form in graphical models, its computation is more complex in probabilistic programs. Conditional branches can quickly increase the size of a closed-form expression for the joint distribution. Unbounded loops and recursion pose additional challenges because their analysis is even harder than in ordinary programming languages [Kaminski et al. 2019].

Normalization constant To compute the denominator $p(y)$ in Bayes' rule from the joint distribution $p(\theta, y)$, we have to marginalize out the parameters θ. Calculating this normalization constant amounts to evaluating the sum $\sum_\theta p(\theta, y)$ in the discrete

case and the integral $\int p(\theta, y)\,\mathrm{d}\theta$ in the continuous case. The use of distributions with infinite support has to be restricted to keep this computation feasible. Even then, the computational cost can be prohibitive for large and high-dimensional parameter spaces. This issue is known as the ***curse of dimensionality*** in numerical analysis and machine learning.

Representation of the posterior In cases where we can compute the normalization constant exactly, Bayes' rule gives us an exact closed-form expression for the posterior. But even then, there are several problems: this closed form is typically huge and unwieldy. Moreover, in most applications, it is not what practitioners are ultimately interested in. Instead, they might care about:

- posterior probabilities of certain events,
- the marginal posterior distribution of some variables,
- summary statistics (e.g. posterior mean, variance, or higher moments),
- more generally, expected values of other functions over the posterior distribution,
- making predictions about new data points from the posterior distribution,
- or obtaining samples from the posterior distribution for visual inspection.

All this information is almost always difficult to extract from a closed-form expression of the posterior distribution. (However, some of this becomes feasible in a restricted setting, see Chapter 4.)

Approximate inference For these reasons, exact inference is only possible for small and restricted models. Instead, practical Bayesian inference usually relies on sampling-based *approximations* following the Monte Carlo approach.[4] ***Approximate inference*** merely aims to approximate the posterior distribution by drawing samples from it. More concretely, Monte Carlo inference algorithms produce samples $\hat{\theta}_1, \ldots, \hat{\theta}_N$ from the posterior distribution $p(\theta \mid y)$ or an approximation thereof that converges to the true posterior as $N \to \infty$. Such samples are easier to work with than a closed form for the posterior distribution: marginalizing out a variable is as simple as ignoring its realization in each sample, and expected values can be *estimated* by averaging:

$$\mathbb{E}_{\theta \sim p(\theta|y)}[f(\theta)] \approx \frac{1}{N} \sum_{i=1}^{N} f(\hat{\theta}_i)$$

For such a sampler to be useful, it is usually required for this estimator to be ***consistent***, i.e. that it converges to the true expected value as $N \to \infty$.

The Monte Carlo approach to approximate inference has been very successful and is the basis for almost all Bayesian inference algorithms. We will discuss specific algorithms in Chapter 2. It is also highly composable: outputs of one Monte Carlo method can be used as input for another. As an example, transforming a distribution by a function can be achieved by applying the function to each sample from the original distribution. Nevertheless, approximate inference has its problems, which we will discuss in the next section.

[4] There are other approximations such as *variational inference*, which we will discuss in Chapter 2.

1.5 Problem Statement

Despite the success of approximate inference, its inexactness causes problems. While most Monte Carlo methods converge to the true posterior distribution as the number of samples increases, the rate of convergence can be very slow, such as for multimodal distributions, i.e. when the probability mass is concentrated in multiple disjoint regions. Even worse, it can be hard to tell when the approximation is good enough. There are convergence diagnostics to detect convergence issues [Roy 2020], but they are not foolproof: if they detect a problem, the approximation is certainly bad, but a lack of a detection does not guarantee that the approximation is good [Gelman et al. 2013, Section 11.5].

This problem is exacerbated for probabilistic programs, which allow for the specification of more complex models. For instance, in Chapter 3, we discuss the "pedestrian example" (Example 2.2.3): a probabilistic program involving loops where a popular inference algorithm in the PPS *Pyro* [Bingham et al. 2019] fails to converge, due to an undocumented assumption.

Exact inference avoids approximation errors, but imposes severe restrictions on the probabilistic programs that can be analyzed, such as disallowing loops/recursion. Even if applicable, exact inference often suffers from scalability issues and is computationally infeasible for large models. So the problem that this thesis aims to address is the following:

> *Can we obtain* hard guarantees *on the posterior distribution of a probabilistic program, without the restrictions of exact inference?*

1.6 Guaranteed Bounds

As a solution to this problem, we propose the concept of ***guaranteed bounds*** on the posterior distribution of a probabilistic program. "Guaranteed" here refers to formal methods for program analysis that compute sound and deterministic (non-stochastic) results about the mathematical denotation of a program. Such bounds are applicable more often than exact inference, e.g. in the presence of loops/recursion, and provide more assurance than approximate methods, which have at best stochastic guarantees. Concretely, we aim to bound the posterior probability of events, e.g. a variable taking a specific value or a value in a certain interval[5] (Chapter 3), and in some cases even the posterior moments and tail asymptotics (Chapter 6). Why are such bounds useful?

Checking approximate inference As motivated in the previous section, the primary purpose of guaranteed bounds in the context of Bayesian inference is to check and debug approximate inference algorithms and their implementations. If the approximate results clearly contradict the bounds, then either it has not converged yet, the inference algorithm is incorrect, or some of its assumptions are violated. Figure 1.3 provides an example of this: our guaranteed bounds can detect the issue with the inference results for Example 2.2.3 (details in Section 3.6.3). Another problem with approximate inference

[5] By repeated application of our method on a discretization of the domain we can compute histogram-like bounds.

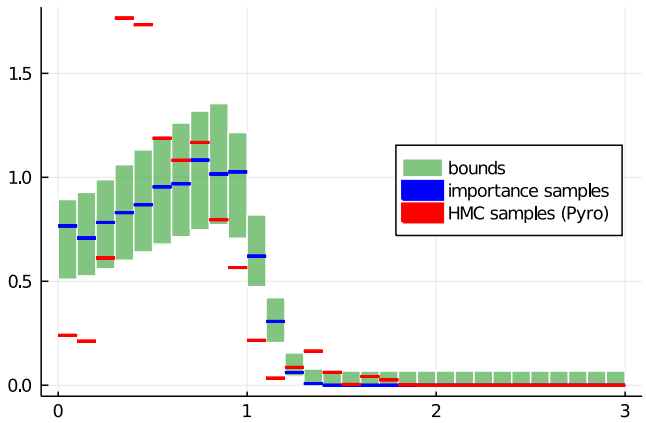


Figure 1.3: Histogram of samples from two inference algorithms (importance sampling and Pyro's HMC), and our guaranteed bounds. The bounds show that Pyro's HMC produces wrong results. (Details in Chapter 3.)

is the tail behavior of the posterior distribution, which is often crucial for the quality of the approximation [Liang et al. 2023]. Using techniques from Chapter 6, we can also find guaranteed bounds on the tail asymptotics of the posterior distribution in many cases.

Partial correctness properties Guaranteed bounds can also be viewed as partial correctness properties of probabilistic programs in the tradition of quantitative program analysis. For instance, probabilistic safety properties can be verified by bounding the probability of reaching an unsafe state. Bounding reachability probabilities is also a common problem in probabilistic model checking and quantitative program verification, yet it has not seen much attention in the context of probabilistic programming with conditioning, aside from our work described in Chapter 3 and P. Wang et al. [2024].

1.7 Contributions

This thesis introduces the concept of *guaranteed bounds* on the posterior distribution of probabilistic programs and presents several techniques for their computation. Contrary to exact inference methods, these techniques are sufficiently general to handle expressive PPLs with *infinite-support* distributions and *loops/recursion*. Along the way, we also discovered a new exact inference algorithm for (loop-free) discrete models with infinite support, which can be viewed as a special case of *tight* guaranteed bounds but is also an important ingredient for one of our bounding techniques. In all cases, our methods are *fully automatic* and require no additional user input like loop invariants.

Overview This thesis is structured as follows. Chapter 2 provides more detailed background on probabilistic programming. It precisely defines the PPLs considered in this thesis and their semantics. It also reviews the existing literature on probabilistic programming, quantitative program analysis, and automated inference for probabilistic

Table 1.1: Overview of the contributions of each chapter of this thesis. (Type: type of results (exact or guaranteed bounds); PPL: functional or imperative; C.s.: support for sampling continuous distributions; C.o.: support for observations from continuous distributions; Loops: support for loops/recursion; Prob.: can compute/bound probabilities; Mom.: can compute/bound moments; Tail: can bound the tail asymptotics.)

Contribution	Type	PPL	C.s.	C.o.	Loops	Prob.	Mom.	Tail
GuBPI (Ch. 3)	bound	functional	✓	✓	✓	✓	✗	✗
Genfer (Ch. 4)	exact	imperative	✓	✗	✗	✓	✓	✗
Residual mass (Ch. 5)	bound	imperative	✓	✗	✓	✓	✗	✗
Geometric bounds (Ch. 6)	bound	imperative	✗	✗	✓	✓	✓	✓

programs. The main (and novel) **contributions** of this thesis are presented in the subsequent chapters:

- Chapter 3 presents *GuBPI*, the first method for guaranteed bounds on posterior probabilities. It works on the most general PPL of this thesis: a higher-order functional language with recursion and both discrete and continuous distributions.
- Chapter 4 presents *Genfer*, a new exact inference algorithm for discrete models with infinite support. It works on an imperative loop-free PPL with support for infinite-support distributions, discrete observations, and discrete/continuous sampling. This exact inference method for loop-free programs serves as a basis for guaranteed bounds for programs with loops in the next chapter, but is also of independent interest.
- Chapter 5 presents *residual mass bounds*, a simple yet effective technique to turn exact inference for loop-free programs into guaranteed bounds on posterior probabilities for programs with loops. It works on the same PPL as Chapter 4, but adds support for unbounded loops. Though less expressive than GuBPI (Chapter 3), it is simpler and more efficient when applicable.
- Chapter 6 presents *geometric bounds*, a powerful technique to bound the *whole* posterior distribution. Not only does it yield bounds on posterior probabilities, but also on the posterior moments and tail asymptotics. It works on a restriction of the imperative PPL from Chapter 5 to discrete distributions.

The comparison of each chapter's contributions is summarized in Table 1.1. Finally, Chapter 7 concludes the thesis and discusses future work. Supplementary material for each chapter, including tedious or routine proofs, additional examples, plots, and tangential discussions, is provided in the *appendices*, so as not to clutter the presentation.

Publications The content of Chapter 3 was published in Beutner et al. [2022a]; Chapter 4 in Zaiser et al. [2023]; and a paper on the contents of Chapter 5 and Chapter 6 will be published in Zaiser et al. [2025].

I was also involved in other research during my PhD, which is *not covered* in this thesis, but resulted in the following publications:

- Zaiser and Ong [2020]. "The Extended Theory of Trees and Algebraic (Co)data-types", *VPT/HCVS@ETAPS 2020*.

- Hougardy, Zaiser, and Zhong [2020]. "The approximation ratio of the 2-Opt Heuristic for the metric Traveling Salesman Problem", *Operations Research Letters 48.4*.
- Mak, Zaiser, and Ong [2021b]. "Nonparametric Hamiltonian Monte Carlo", *ICML 2021*.
- Mak, Zaiser, and Ong [2022]. "Nonparametric Involutive Markov Chain Monte Carlo", *ICML 2022*.

1.8 Limitations

Formal methods for program analysis are inherently limited by undecidability results like the halting problem. Even in decidable cases like finite state spaces, the time complexity of such decision procedures can be prohibitively high. In probabilistic programming, not only do we have to reason about large state spaces, but even about probability distributions on them, which makes the formal analysis of probabilistic programs much harder than their deterministic counterparts. This claim is supported by theoretical results on computational complexity [Kaminski et al. 2019].

While our techniques for guaranteed bounds can lift the expressiveness restrictions that exact inference methods impose, they still suffer from the scalability issues. The curse of dimensionality strikes again: the running time of our methods usually scales exponentially in the dimension of the program distribution. This means that our methods are only feasible for small and medium-sized models, and most importantly, with a small number of variables/samples, to keep the dimension low. A more detailed discussion of the limitations of each approach can be found in the respective chapters. We emphasize that scalability is a general issue in probabilistic program analysis owing to the hardness of the problem [Dagum and Luby 1993]: formal methods for probabilistic program analysis in the literature are usually only evaluated on small programs of at most a few dozen lines.

On the bright side, probabilistic programs are typically smaller and simpler than their deterministic counterparts, which renders the above concerns less severe. Furthermore, pathological behavior of approximate inference algorithms often manifests already for small models (e.g. Fig. 3.6d and Fig. 3.8). This suggests that our guaranteed bounds can be used as a ground truth to unit test approximate inference implementations and to debug small versions of real-world models before scaling them up.

1.9 Notation and Conventions

A summary of the mathematical notation used in this thesis is given on page xiv. Notation around measure theory, probability theory and program semantics is introduced in Chapter 2. We discuss general mathematical notation and conventions here.

The natural numbers $\mathbb{N}$ start at 0, i.e. $\mathbb{N} = \{0, 1, 2, \dots\}$. We use the Iverson brackets $[\varphi]$ to mean 1 if φ is satisfied and 0 otherwise. For example, the indicator function $\mathbf{1}_S$ of a set S can be defined using Iverson brackets: $\mathbf{1}_S(x) = [x \in S]$. We usually write variables representing vectors in bold ($\boldsymbol{\alpha}$), tensors (multidimensional arrays) in uppercase and

bold ($\mathbf{T}$), and random variables in uppercase (X). We write $\mathbf{0}$ for the constant zero function or the zero measure, and $\mathbf{1}$ for the constant one function.

We write $\mathbf{0}_n := (0, \dots, 0)$ and $\mathbf{1}_n = (1, \dots, 1)$ for the zero and ones vector in $\mathbb{R}^n$. Vectors $\boldsymbol{\alpha} \in \mathbb{R}^n$ are indexed as $\alpha_1, \dots, \alpha_n$. To set the k-th component of a vector $\boldsymbol{\alpha}$ to v, we write $\boldsymbol{\alpha}[k \mapsto v]$.

We abbreviate $[d] := \{0, \dots, d-1\}$. A ***tensor*** $\mathbf{T} \in \mathbb{R}^{[d_1] \times \cdots \times [d_n]}$ is a multidimensional array and indexed as $\mathbf{T}_{i_1,\dots,i_n}$ where i_k ranges from 0 to $d_k - 1$. We write $\mathbf{0}_{[d_1] \times \cdots \times [d_n]}$ or simply $\mathbf{0}$ for the zero tensor in $\mathbb{R}^{[d_1] \times \cdots \times [d_n]}$. We write $|\mathbf{T}| = (d_1 \dots, d_n)$ for the dimensions of $\mathbf{T} \in \mathbb{R}^{[d_1] \times \cdots \times [d_n]}$, and in particular $|\mathbf{T}|_i = d_i$. To index $\mathbf{T}$ along the k-th dimension, we write $\mathbf{T}_{k:j} \in \mathbb{R}^{[d_1] \times \cdots \times [d_{k-1}] \times [d_{k+1}] \times \cdots \times [d_n]}$, which is defined by $(\mathbf{T}_{k:j})_{i_1,\dots,i_{k-1},i_{k+1},\dots,i_n} = \mathbf{T}_{i_1,\dots,i_{k-1},j,i_{k+1},\dots,i_n}$.

We often write multi-indices as $\boldsymbol{i} := (i_1, \dots, i_n)$ for brevity. We use multi-index notation for sums $|\boldsymbol{i}| := i_1 + \cdots + i_n$, factorials $\boldsymbol{i}! := i_1! \cdots i_n!$, partial derivatives $\partial^{\boldsymbol{i}} f(\boldsymbol{x}) := \frac{\partial^{i_1}}{\partial x_1^{i_1}} \cdots \frac{\partial^{i_n}}{\partial x_n^{i_n}} f(x_1, \dots, x_n)$, and powers $\boldsymbol{\alpha}^{\boldsymbol{i}} := \prod_{k=1}^{n} \alpha_i^{i_k}$. Other binary operations ($+$, $-$, min, max, etc.) work elementwise on vectors and tensors, e.g. $(\boldsymbol{\alpha} + \boldsymbol{\beta})_j := \alpha_j + \beta_j$ and $\boldsymbol{\alpha} \leq \boldsymbol{\beta}$ if and only if $\alpha_j \leq \beta_j$ for all j.

At a purely formal level, one could call probability theory the study of measure spaces with total measure one, but that would be like calling number theory the study of strings of digits which terminate.

— Terence Tao

2

Background and Related Work

Contents

2.1 Measure Theory and Probability Theory . . . 19
2.2 Probabilistic Programming Languages . . . 22
2.2.1 SGCL: an Imperative PPL . . . 23
2.2.2 Discrete SGCL: a Restricted Fragment of SGCL . . . 26
2.2.3 SPCF: a Functional PPL . . . 26
2.3 Semantics of PPLs . . . 29
2.3.1 Transformer Semantics for SGCL . . . 29
2.3.2 Trace Semantics for SPCF . . . 33
2.4 Exact Inference . . . 35
2.5 Approximate Inference . . . 36
2.5.1 Rejection Sampling . . . 37
2.5.2 Importance Sampling . . . 38
2.5.3 Markov Chain Monte Carlo . . . 39
2.5.4 Particle-Based Inference . . . 40
2.5.5 Other Methods . . . 41
2.5.6 Statistical Checks for Inference . . . 42
2.6 Formal Methods for Probabilistic Programs . . . 44
2.6.1 Applications to Bayesian Inference . . . 47
2.7 A Brief History of Probabilistic Programming . . . 48

Summary This chapter explains fundamental concepts and technical preliminaries, on which the rest of this thesis builds. We start by recalling basic measure and probability theory. Next, we introduce the two probabilistic programming languages (SGCL and SPCF) used in this thesis and their formal semantics, i.e. how programs are interpreted as (posterior) probability distributions. Finally, we review the literature on probabilistic programming, inference algorithms, and other analyses of probabilistic programs, explaining all these concepts in more detail.

2.1 Measure Theory and Probability Theory

We assume familiarity with basic measure theory and probability theory, and refer to the textbook by Pollard [2002] for details. The purpose of this section is to recall important definitions, fix notation, and establish some conventions.

Measure theory A ***measurable space*** is a pair (Ω, Σ_Ω) where Ω is a set (of outcomes) and $\Sigma_\Omega \subseteq \mathcal{P}(\Omega)$ is a σ-algebra defining the measurable subsets of Ω. A ***measurable function*** between two measurable spaces (Ω, Σ_Ω) and $(\Omega', \Sigma_{\Omega'})$ is a function $f : \Omega \to \Omega'$ such that $f^{-1}(A) \in \Sigma_\Omega$ for all $A \in \Sigma_{\Omega'}$. A ***measure*** on (Ω, Σ_Ω) is a function $\mu : \Sigma_\Omega \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that satisfies $\mu(\varnothing) = 0$ and is σ-additive, i.e. $\mu(\bigcup_{i=1}^\infty A_i) = \sum_{i=1}^\infty \mu(A_i)$ for all countable collections $A_1, A_2, \ldots$ of pairwise disjoint sets in Σ_Ω. Its ***total measure*** or ***total mass*** is $\mu(\Omega)$. A ***null set*** is a set of measure zero. The set of measures on a measurable space Ω is denoted by $\mathrm{Meas}(\Omega)$. A ***measure space*** $(\Omega, \Sigma_\Omega, \mu)$ is a measurable space (Ω, Σ_Ω) with a measure μ. We say that a property holds ***almost everywhere (a.e.)*** on a measure space Ω if it holds on $\Omega \setminus N$ for some null set $N \in \Sigma_\Omega$. The ***restriction of a measure*** $\mu \in \mathrm{Meas}(\Omega)$ to a measurable subset $\Omega' \subseteq \Omega$ is defined by $\mu|_{\Omega'}(A) := \mu(A \cap \Omega')$. The ***pushforward measure*** of a measurable function $f : \Omega \to \Omega'$ is the measure $f_*(\mu)$ on Ω' defined by $f_*(\mu)(A) := \mu(f^{-1}(A))$ for all $A \in \Sigma_{\Omega'}$. The ***Lebesgue integral*** of a measurable function f with respect to a measure μ is written $\int f \, \mathrm{d}\mu$ or $\int f(x) \, \mu(\mathrm{d}x)$. For $A \in \Sigma_\Omega$ we define the bounded integral $\int_A f \, \mathrm{d}\mu := \int [x \in A] \cdot f(x) \, \mu(\mathrm{d}x)$.

Standard measures The ***Borel σ-algebra*** on a topological space Ω is the smallest σ-algebra containing all open sets. Unless otherwise specified, the Euclidean space $\mathbb{R}^n$ carries the Euclidean topology, the ***Borel σ-algebra*** $\Sigma_{\mathbb{R}^n}$, and the ***Lebesgue measure*** λ_n. Countable sets Ω carry the discrete σ-algebra $\mathcal{P}(\Omega)$, which is also its Borel σ-algebra, and the ***counting measure*** $\mu(A) := |A|$. Similarly, products of measurable spaces carry the ***product σ-algebra*** and products of measure spaces carry the ***product measure***. We call these measures the ***standard measures*** on those spaces.

Probability theory We will use the words ***distribution*** and measure interchangeably. A ***probability distribution*** or a ***probability measure*** is a measure μ with total mass $\mu(\Omega) = 1$. A ***subprobability distribution*** is a measure with total mass $\mu(\Omega) \leq 1$. The measure space Ω is called the ***sample space***. For (sub-)probability spaces, the measurable sets are also called ***events***, and the measure of an event E is called the ***probability*** of E and written $\mathbb{P}[E]$. An event E is said to happen ***almost never*** if $\mathbb{P}[E] = 0$ and ***almost surely (a.s.)*** if its complement $\Omega \setminus E$ happens almost never. A ***random variable (RV)*** X taking values in a measurable space Ω', typically $\mathbb{R}$ or $\mathbb{R}^n$, is a measurable function $X : \Omega \to \Omega'$. Its ***distribution*** D is the pushforward measure $D := X_*(\mu)$, which is often written as $X \sim D$. Events involving a random variable X are often written as logical predicates identifying subsets of Ω, e.g. $X \in S$ for a measurable subset $S \subseteq \Omega'$ stands for the event $\{\omega \in \Omega \mid X(\omega) \in S\}$. The ***joint distribution*** of RVs $X_1, \ldots, X_n$ is the pushforward measure $D_{X_1,\ldots,X_n} := f_*(\mu)$ along $f(\omega) := (X_1(\omega), \ldots, X_n(\omega))$. To ***marginalize out*** a variable X_i, we take the pushforward measure $\pi_*(D_{X_1,\ldots,X_n})$ along the projection on the other components $\pi(x_1, \ldots, x_n) := (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. The

marginal distribution of X_i is the result of marginalizing out all other variables, or equivalently, the pushforward measure $(\pi_i)_*(D_{X_1,\dots,X_n})$ along the projection $\pi_i(x_1, \dots, x_n) := x_i$. RVs are called ***independent*** if their joint distribution is the product measure of their marginals. The ***expected value*** or ***mean*** of a RV on $\mathbb{R}^n$ is defined as $\mathbb{E}[X] := \int X \, \mathrm{d}\mu$ and often also denoted by μ. More generally, its k-th ***moment*** or ***raw moment*** of a RV on $\mathbb{R}$ is $\mu'_k := \mathbb{E}[X^k] := \int X^k \, \mathrm{d}\mu$. Its ***variance*** is $\mathbb{V}[X] := \mathbb{E}[(X - \mathbb{E}[X])^2]$ and more generally, its k-th ***central moment*** is $\mu_k := \mathbb{E}[(X - \mathbb{E}[X])^k]$. Its ***standard deviation*** is defined as $\sigma := \sqrt{\mathbb{V}[X]}$; therefore the variance is often denoted by σ^2. The k-th ***standardized moment*** of X is μ_k/σ^k where μ_k is the k-th central moment and σ is the standard deviation. The third standardized moment is called the ***skewness*** $\mathrm{Skew}[X]$ and the fourth standardized moment is called the ***kurtosis*** $\mathrm{Kurt}[X]$.

Densities Let Ω be a measurable space with a measure μ. Another measure ν on Ω is ***absolutely continuous*** with respect to μ if $\mu(A) = 0$ implies $\nu(A) = 0$ for all measurable subsets A. A measure on a subset of $\mathbb{R}^n$ is called ***continuous*** if it is absolutely continuous with respect to the Lebesgue measure, and ***discrete*** if it is absolutely continuous with respect to the counting measure on a subset of $\mathbb{R}^n$, typically $\mathbb{Z}^n$ or $\mathbb{N}^n$. If ν is absolutely continuous with respect to μ, there exists a measurable function $f : \Omega \to \mathbb{R}$ such that $\nu(A) = \int_A f \, \mathrm{d}\mu$ for all measurable A. Such a function is called a ***Radon-Nikodym derivative*** or ***density*** and written $\frac{\mathrm{d}\nu}{\mathrm{d}\mu}$. If ν is a probability measure on a space with standard measure μ, the Radon-Nikodym derivative is called the ***probability density function (PDF)*** $\mathrm{pdf}_\nu : \Omega \to \mathbb{R}_{\geq 0}$ of ν. Note that the Radon-Nikodym derivative is only unique up to a null set. However, in practice, we can often choose the PDF to be a continuous function and this requirement usually ensures uniqueness. Another way to define the density at a point ω uniquely is via a limit of the ratio of the (non-zero) measure of sets converging in some sense to ω [Piccioni 1983; Fraser et al. 1995]. If μ is the counting measure, the Radon-Nikodym derivative is called ***mass function*** $\mathrm{pmf}_\nu : \Omega \to [0, 1]$, or ***probability mass function (PMF)*** if ν is a probability measure, and defined by $\mathrm{pmf}_\nu(x) := \nu(\{x\})$. By abuse of notation, we will usually write ν for pmf_ν as well.

Further properties of distributions The ***cumulative distribution function (CDF)*** of a probability measure ν on $\mathbb{R}$ is the function $\mathrm{cdf}_\nu : \mathbb{R} \to [0, 1]$ given by $\mathrm{cdf}_\nu(x) := \nu((-\infty, x])$. The ***quantile function*** or ***inverse cumulative distribution function (ICDF)*** of a probability measure ν is the function $\mathrm{icdf}_\nu : [0, 1] \to \mathbb{R}$ given by $\mathrm{icdf}_\nu(p) := \inf\{x \in \mathbb{R} \mid \mathrm{cdf}_\nu(x) \geq p\}$. If $X \sim \mathrm{Uniform}(0, 1)$ then $\mathrm{icdf}_\nu(X)$ has distribution ν. A ***mode*** of a distribution ν is a value x where the PDF $\mathrm{pdf}_\nu(x)$ attains a local maximum. The ***support*** of a measure on a topological space Ω with the Borel σ-algebra is defined as $\mathrm{supp}(\mu) := \{x \in \Omega \mid \mu(N) > 0 \text{ for all open sets } N \text{ containing } x\}$. For probability distributions ν on Ω with a density function with respect to the standard measure on Ω, the support is simply $\mathrm{supp}(\nu) = \overline{\{x \in \Omega \mid \mathrm{pdf}_\nu(x) > 0\}}$ where $\overline{U}$ denotes the topological closure of the set U. The ***tail*** of a distribution refers to the region far away from the mean, i.e. $n \gg 0$ for a distribution μ on $\mathbb{N}$. Such a distribution has ***tail asymptotics*** $f(n)$ if $\mathrm{pmf}_\mu(n) = \Theta(f(n))$. The ***total variation distance (TVD)*** between two probability measures ν and ν' on a measurable space Ω is defined as

Table 2.1: Common probability distributions.

(a) Discrete distributions on $\mathbb{N}$.

Distribution D	$\mathrm{pmf}_D(n)$
Dirac(x)	$[n = x]$
Bernoulli(ρ)	$\begin{cases} 1-\rho & \text{if } n = 0 \\ \rho & \text{if } n = 1 \end{cases}$
Binomial(m, ρ)	$\binom{m}{n} \cdot \rho^n \cdot (1-\rho)^{m-n}$
Categorical$(p_0, \ldots, p_m)$	$\sum_{i=0}^{m} p_i \cdot [n = i]$
Geometric(ρ)	$(1-\rho)^n \cdot \rho$
NegBinomial(r, ρ)	$\binom{n+r-1}{r-1} \cdot \rho^r \cdot (1-\rho)^n$
Poisson(λ)	$\frac{\lambda^n}{n!} \cdot e^{-\lambda}$
Uniform$\{a, \ldots, b\}$	$\frac{[a \le n \le b]}{b-a+1}$

(b) Continuous distributions on $\mathbb{R}$.

Distribution D	$\mathrm{pdf}_D(x)$
Dirac(x)	(see text)
Beta(α, β)	$\frac{x^{\alpha-1} \cdot (1-x)^{\beta-1}}{B(\alpha,\beta)}$
Exponential(λ)	$\lambda \cdot e^{-\lambda x}$
Gamma(α, β)	$\frac{\beta^\alpha}{\Gamma(\alpha)} \cdot x^{\alpha-1} \cdot e^{-\beta x}$
Normal(μ, σ^2)	$\frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
Uniform$[a, b]$	$\frac{[a \le x \le b]}{b-a}$

$\sup_{A \in \Sigma_\Omega} |\nu(A) - \nu'(A)|$. If ν and ν' have densities p and q with respect to μ, then the total variation distance is $\frac{1}{2} \int |p(x) - q(x)| \mu(\mathrm{d}x)$.

We list common probability distributions on $\mathbb{N}$ with their mass function and on $\mathbb{R}$ with their density function in Table 2.1. The Dirac(x) distribution does not have a density function with respect to the Lebesgue measure. Instead, it is given by Dirac$(x)(U) = [x \in U]$ for all measurable sets U.

Conditioning The ***conditional probability*** of an event E given another event F is defined as $\mathbb{P}[E \mid F] := \frac{\mathbb{P}[E \cap F]}{\mathbb{P}[F]}$ if $\mathbb{P}[F] > 0$. From this definition, it is easy to derive ***Bayes' rule***: $\mathbb{P}[E \mid F] = \frac{\mathbb{P}[F|E] \cdot \mathbb{P}[E]}{\mathbb{P}[F]}$. If F has probability 0, the definition is more involved and requires the following concepts.

A ***measure kernel*** from a measurable space Ω to another measurable space Ω' is a function $K : \Omega \times \Sigma_{\Omega'} \to \mathbb{R}_{\ge 0}$ such that $K(\omega, \cdot)$ is a measure on Ω' for all $\omega \in \Omega$ and $K(\cdot, A)$ is measurable for all $A \in \Sigma_{\Omega'}$. It is called a ***probability kernel*** or ***Markov kernel*** if $K(\omega, \Omega') = 1$ for all $\omega \in \Omega$. Let μ be a measure on Ω and $f : \Omega \to \Omega'$ measurable. An f-***disintegration*** of μ is a measure kernel $K : \Omega' \times \Sigma_\Omega \to \mathbb{R}_{\ge 0}$ such that $K(x, \{y \in \Omega' \mid f(y) \neq x\}) = 0$ for $f_*(\mu)$-almost all $x \in \Omega'$ and $\mu(A) = \int_{\Omega'} K(x, A)\, f_*(\mu)(\mathrm{d}x)$ for all $A \in \Sigma_\Omega$. If μ is a probability measure and X a random variable, then an X-disintegration K of μ is called a ***regular conditional probability distribution*** of μ given X if K is a probability kernel. This distribution is only unique up to a null set, similarly to densities. If it exists, this allows us to define a conditional distribution given $X = x$ as $K(x, \cdot)$, even if this event has probability 0.

Typically, the conditional distribution given $X = x$ can be computed in terms of the density functions of the joint distribution and the marginal distribution. Let p be the density of a probability measure μ with respect to ν; K be a regular conditional distribution of μ given X; and K' be an X-disintegration of ν. Then under mild conditions, $X_*(\mu)$ has the ***marginal density*** $q(x) := \int p(y)\, K'(x, \mathrm{d}y)$ and $K(x, \cdot)$ has the ***conditional density*** $p(y \mid X = x) := \frac{p(y)}{q(x)}$ where the denominator is nonzero and finite almost everywhere (see Pollard [2002, Section 5.4] for details). In statistics, Ω is usually a product $\Theta \times D$ of a parameter space Θ and an observation space D. The

distribution μ has joint density $p(\theta, y)$ with respect to the product measure ν of ν_Ω and ν_D. Let $Y : \Omega \to D$ be the random variable corresponding to the observations, given by the projection onto D. Then $K'(y, \cdot) := \nu_\Theta$ for all $y \in Y$ is a Y-disintegration of ν; the marginal density is given by $p_Y(y) := \int p(\theta, y)\, \nu_\Theta(\mathrm{d}\theta)$; and the conditional distribution $K(\theta, \cdot)$ of μ given Y has the density $p(\theta \mid Y = y) = \frac{p(\theta,y)}{p_Y(y)}$. By applying the same reasoning with respect to the projection $\Omega \to \Theta$, it is easy to derive the density version of Bayes' rule: $p(\theta \mid Y = y) = \frac{p(y|\Theta=\theta)\cdot p_\Theta(\theta)}{p_Y(y)}$ where $p_\Theta(\theta) := \int p(\theta, y)\, \nu_D(\mathrm{d}y)$ is the other marginal density and $p(y \mid \Theta = \theta) := \frac{p(\theta,y)}{p(\theta)}$ is the conditional density of Y given Θ. Like the PDF, the conditional density is only unique up to a null set. Again, there are other ways to define it in terms of a limit that ensures uniqueness in many cases [Piccioni 1983; Fraser et al. 1995]. In particular, we refer to Borgström et al. [2013, Section 3.2] for a unique definition in the context of probabilistic programming.

2.2 Probabilistic Programming Languages

Probabilistic programming (PP) does not have a universally agreed upon definition, but it is generally understood as the practice of using programming languages to describe probabilistic models and perform inference on them. As mentioned in Section 1.3, ***probabilistic programming languages (PPLs)*** extend ordinary programming languages with two additional constructs: one for *sampling* from probability distributions and one for *conditioning* on observed values. The PPLs of interest in this thesis are built on top of Turing-complete languages with branching and loops/recursion that allow for an unbounded number of samples in a program. To emphasize this aspect, such PPLs are called ***universal probabilistic programming languages***.

In this thesis, we consider two PPLs. The first is an imperative PPL called *Statistical GCL (SGCL)* on a fixed number of variables with discrete or continuous sampling and discrete observations [Zaiser et al. 2023]. For some applications, such as Chapter 6, it is restricted further to a fragment called *discrete SGCL* [Zaiser et al. 2025]. The second is a functional PPL called *Statistical PCF (SPCF)* with sampling and observations from discrete and continuous distributions [Mak et al. 2021a]. SPCF is used in Chapter 3 and (versions of) SGCL in Chapters 4 to 6. In this section, we describe both languages in detail and explain their semantics. But first, let's build some intuition with the following example.

Example 2.2.1. For illustration, consider the following puzzle due to Elchanan Mossel. (It seems to have spread by word of mouth and there is no written record to cite.)

> You throw a fair six-sided die repeatedly until you get a 6. You observe only even numbers during the throws. What is the expected number of throws (including the 6) conditioned on this event?

The puzzle can be expressed as the following SGCL program.

$$
\begin{aligned}
&\mathit{Throws} := 0; \mathit{Die} := 0;\\
&\textsf{while}\ \mathit{Die} \neq 6\ \{\\
&\quad \mathit{Die} \sim \textsf{Uniform}\{1, \ldots, 6\};\\
&\quad \textsf{observe}\ \mathit{Die} \in \{2, 4, 6\};
\end{aligned}
$$

Throws += 1
}

This is a surprisingly tricky problem and most people get it wrong on the first try[1], based on the incorrect assumption that it is equivalent to throwing a die with only the three faces 2, 4, and 6. Probability theory and statistics abound with such counterintuitive results (e.g. the Monty-Hall problem), and probabilistic programming offers a precise way to disambiguate their description and make them amenable to automatic analysis and inference tools.

2.2.1 SGCL: an Imperative PPL

SGCL is based on the ***probabilistic guarded command language (pGCL)*** from Klinkenberg et al. [2020] but augments it with statistical features like conditioning on events, which is why we call it ***Statistical GCL (SGCL)***. Its loop-free fragment is described in Zaiser et al. [2023].

Syntax An SGCL program operates on a fixed set of variables $\boldsymbol{X} = (X_1, \ldots, X_n)$, each taking values in the nonnegative real numbers $\mathbb{R}_{\geq 0}$. We usually assume that all variables are set to zero at the start of a program. Some of the variables may be discrete, i.e. they take values only in $\mathbb{N}$. If a variable is required to be discrete in the syntax, we mark it as $X_k^{(\mathbb{N})}$. A grammar of SGCL ***programs*** P and ***events*** E looks like this:

$$
\begin{aligned}
P &::= \mathsf{skip} \mid P_1; P_2 \mid X_k := a_1 X_1 + \cdots + a_n X_n + c \mid X_k^{(\mathbb{N})} \dot{-}= 1 \mid \mathsf{if}\ E\ \{P_1\}\ \mathsf{else}\ \{P_2\} \\
&\quad \mid \mathsf{while}\ E\ \{P_1\} \mid \mathsf{fail} \mid X_k \sim D \mid X_k \sim D(X_j) \\
E &::= X_k^{(\mathbb{N})} = m \mid \mathsf{flip}(\rho) \mid \neg E_1 \mid E_1 \wedge E_2
\end{aligned}
$$

where $a_1, \ldots, a_n, c \in \mathbb{R}_{\geq 0}, m \in \mathbb{N}, \rho \in [0, 1]$ are literals. To explain the program syntax: skip leaves everything unchanged; $P_1; P_2$ runs P_1 and then P_2; and $X_k := \mathbf{a}^\top \boldsymbol{X} + c$ performs an affine transformation where $\mathbf{a} \in \mathbb{R}^n_{\geq 0}$ and $c \in \mathbb{R}_{\geq 0}$ (to ensure that the support of X_k remains a subset of $\mathbb{R}_{\geq 0}$). The decrement operation $X_k \dot{-}= 1$ assigns $X_k \dot{-} 1$ to the discrete variable X_k, where $\dot{-}$ denotes the ***monus*** operator, i.e. subtraction clamped at 0: $a \dot{-} b = \max\{0, a - b\}$. The if construct allows branching on an event and a while loop repeats a subprogram as long as an event occurs. The fail construct is responsible for conditioning: it states a contradictory observation, i.e. observe false. Finally, $X_k \sim D$ samples X_k from a primitive distribution (with constant parameters)

$$
\begin{aligned}
\mathcal{D} = \{&\mathsf{Bernoulli}(p), \mathsf{Categorical}(p_0, \ldots, p_m), \mathsf{Binomial}(m, p), \mathsf{Uniform}\{l..m\}, \\
&\mathsf{NegBinomial}(m, p), \mathsf{Geometric}(p), \mathsf{Poisson}(\lambda), \\
&\mathsf{Exponential}(\alpha), \mathsf{Gamma}(\alpha, \beta), \mathsf{Uniform}[a, b] \\
&\mid p \in [0, 1], \alpha, \beta \in \mathbb{R}_{\geq 0}, l, m \in \mathbb{N}, a, b \in \mathbb{R}_{\geq 0}, a \leq b\}
\end{aligned}
$$

[1] In a survey on Gil Kalai's blog, only 27% of participants chose the correct answer (https://gilkalai.wordpress.com/2017/09/07/tyi-30-expected-number-of-dice-throws/).

whereas $X_k \sim D(X_j)$ samples X_k from a ***compound distribution***, i.e. a distribution with parameters that depend on another variable X_j:

$$\mathcal{D}(X_j) = \{\text{Binomial}(X_j^{(\mathbb{N})}, p), \text{NegBinomial}(X_j^{(\mathbb{N})}, p), \text{Poisson}(\lambda \cdot X_j), \text{Bernoulli}(X_j) \mid p \in [0,1], j \in \{1,\dots,n\}, \lambda \in \mathbb{R}_{\geq 0}\}$$

For $\text{Bernoulli}(X_j)$, the parameter X_j is obviously restricted to values in $[0,1]$. By $D^{(\mathbb{N})}$ and $D^{(\mathbb{N})}(X_j)$ we denote discrete distributions, i.e. all except Exponential, Gamma, and $\text{Uniform}[a,b]$.

Events E are the expressions that can be used in if statements and observations. The event $X_k = m$ occurs if the value of the discrete variable X_k equals m. The $\text{flip}(\rho)$ event occurs with probability ρ, like a coin flip with bias ρ coming up heads, or more formally, sampling 1 from an independent $\text{Bernoulli}(\rho)$ distribution.[2] The logical operators $\neg$, $\wedge$ denote the complement and intersection of events. Note that the non-probabilistic fragment of the language is already Turing-complete because it can simulate a three-counter machine.

Discreteness of variables Since discreteness of variables is important, we briefly discuss which operations preserve discreteness as required. Affine assignments leave X_k discrete if all coefficients $a_1, \dots, a_n, c$ are integers and X_i is discrete for all i with $a_i \neq 0$. Sampling preserves the discreteness of X_k if the sampled distribution is discrete. Note that for a compound distribution $D(X_j)$, discreteness of X_j is not always required: $\text{Bernoulli}(X_j)$ and $\text{Poisson}(\lambda \cdot X_j)$ yield discrete values even for continuous parameters. For instance, the model from Fig. 1.2b can be expressed in SGCL. These requirements could easily be expressed in a type system, but we omit this for simplicity.

Syntactic sugar For convenience, we add some syntactic sugar. First, for events, we define the usual comparison operators, logical connectives, and drawing a value from a discrete distribution:

$$\begin{aligned}
\text{false} &\rightsquigarrow \text{flip}(0) \\
\text{true} &\rightsquigarrow \neg\text{false} \\
E_1 \vee E_2 &\rightsquigarrow \neg(\neg E_1 \wedge \neg E_2) \\
X_k^{(\mathbb{N})} \in \{a_1,\dots,a_m\} &\rightsquigarrow \text{false} \vee X_k^{(\mathbb{N})} = a_1 \vee \dots \vee X_k^{(\mathbb{N})} = a_m \\
X_k^{(\mathbb{N})} \notin \{a_1,\dots,a_m\} &\rightsquigarrow \neg(X_k^{(\mathbb{N})} \in \{a_1,\dots,a_m\}) \\
X_k^{(\mathbb{N})} \neq m &\rightsquigarrow \neg(X_k^{(\mathbb{N})} = m) \\
X_k^{(\mathbb{N})} < a &\rightsquigarrow X_k^{(\mathbb{N})} \in \{0,\dots,a-1\} \\
X_k^{(\mathbb{N})} \leq a &\rightsquigarrow X_k^{(\mathbb{N})} < a+1
\end{aligned}$$

[2] This construct is not usually considered an event because it does not correspond to a subset of the state space. We included it for convenience: it allows us to write probabilistic choice (if $\text{flip}(\rho)$ {…} else {…}) and loops with fixed branch probabilities (while $\text{flip}(\rho)$ {…}) without introducing auxiliary variables. It also facilitates a limited form of probabilistic sampling in the more restricted language *discrete SGCL* (see Section 2.2.2 below).

$$
\begin{aligned}
X_k^{(\mathbb{N})} \geq a &\rightsquigarrow \neg(X_k^{(\mathbb{N})} < a) \\
X_k^{(\mathbb{N})} > a &\rightsquigarrow \neg(X_k^{(\mathbb{N})} \leq a) \\
m \sim D^{(\mathbb{N})} &\rightsquigarrow \mathsf{flip}(\mathrm{pmf}_{D^{(\mathbb{N})}}(m))
\end{aligned}
$$

Second, we introduce the following syntactic sugar for statements. Some of them require an extra variable X_{n+1}, which can always be added to $\boldsymbol{X}$ as needed.

$$
\begin{aligned}
X_k := c &\rightsquigarrow X_k := \mathbf{0}_n^\top \boldsymbol{X} + c \\
X_k \mathrel{+}= c &\rightsquigarrow X_k := (\mathbf{0}_n[k \mapsto 1])^\top \boldsymbol{X} + c \\
X_k \mathrel{+}= X_l &\rightsquigarrow X_k := (\mathbf{0}_n[k \mapsto 1, l \mapsto 1])^\top \boldsymbol{X} \\
X_k \mathrel{+}\sim D &\rightsquigarrow X_{n+1} \sim D; X_k \mathrel{+}= X_{n+1}; X_{n+1} := 0 \\
X_k \mathrel{+}\sim D(X_j) &\rightsquigarrow X_{n+1} \sim D(X_j); X_k \mathrel{+}= X_{n+1}; X_{n+1} := 0 \\
\{P_1\}\ [\rho]\ \{P_2\} &\rightsquigarrow \mathsf{if}\ \mathsf{flip}(\rho)\ \{P_1\}\ \mathsf{else}\ \{P_2\} \\
\mathsf{observe}\ E &\rightsquigarrow \mathsf{if}\ E\ \{\mathsf{skip}\}\ \mathsf{else}\ \{\mathsf{fail}\} \\
\mathsf{observe}\ m \sim D^{(\mathbb{N})}(X_j) &\rightsquigarrow X_{n+1} \sim D^{(\mathbb{N})}(X_j); \mathsf{observe}\ X_{n+1} = m; X_{n+1} := 0
\end{aligned}
$$

This explains how observations can be expressed in terms of fail: if the observed event does not happen, the program fails. The meaning of fail is explained in more detail in Section 2.3.1. Note that in observations, only discrete distributions and comparisons of discrete variables are allowed.

Hard and soft conditioning Discrete PPLs usually only support ***hard conditioning***, which means that the likelihood function can only take the values 1 (if all observations hold) and 0 (if at least one observation fails). SGCL only supports hard conditioning as a primitive: the fail statement sets the likelihood to 0. However, ***soft conditioning*** restricted to likelihood weights in $[0, 1]$ can be expressed using the syntactic sugar observe $\mathsf{flip}(\rho)$, which effectively multiplies the likelihood by $\rho \in [0, 1]$. Languages with continuous observations (such as SPCF) also allow for the likelihood to assume values greater than 1 (see Section 2.2.3).

Restrictions Our language imposes several syntactic restrictions in order to guarantee that the program distribution can be exactly represented in the loop-free case (needed in Chapter 4), or effectively bounded (needed in Chapters 5 and 6):

- only affine functions are supported (e.g. no X^2 or $\exp(X)$),
- only comparisons between discrete variables and constants are supported (e.g. no test for equality $X = Y$),
- only observations from discrete distributions on $\mathbb{N}$ and only comparisons of such random variables are supported (e.g. no observe $1.5 \sim \mathsf{Normal}(0, 1)$),
- a particular choice of distributions and their composites is supported.

Note that for discrete variables, most of these restrictions do not fundamentally limit the expressive power: the deterministic fragment of the language is already Turing-complete, so arithmetic operations like integer multiplication and comparisons of two variables are not hard to encode, albeit at the cost of increasing the program's

complexity. The restrictions on continuous variables and distributions cannot easily be circumvented, however. Our second PPL, SPCF, is not subject to these limitations on continuous variables.

2.2.2 Discrete SGCL: a Restricted Fragment of SGCL

Some techniques, notably those in Chapter 6, require that program variables be restricted to values in $\mathbb{N}$. For this purpose, ***discrete SGCL*** is defined as the subset of SGCL with the following grammar:

$$P ::= \mathsf{skip} \mid P_1; P_2 \mid X_k := 0 \mid X_k \mathrel{+}= c \mid X_k \mathrel{\dot{-}}= 1 \mid \mathsf{if}\, E\, \{P_1\}\, \mathsf{else}\, \{P_2\} \mid \mathsf{while}\, E\, \{P_1\} \mid \mathsf{fail}$$

where $c \in \mathbb{N}$ and events E are defined in full SGCL. Note that the non-probabilistic fragment of the language is still Turing-complete because it can simulate a three-counter machine. As for full SGCL, the discrete variant is kept minimal to simplify the presentation, but we introduce the following syntactic sugar:

$$\begin{aligned} X_k := c &\rightsquigarrow X_k := 0; X_k \mathrel{+}= c \\ X_k \sim \mathsf{Bernoulli}(\rho) &\rightsquigarrow \{X_k := 1\}\ [\rho]\ \{X_k := 0\} \\ X_k \sim \mathsf{Geometric}(\rho) &\rightsquigarrow X_k := 0; \mathsf{while}\, \neg\mathsf{flip}(\rho)\, \{X_k \mathrel{+}= 1\} \end{aligned}$$

Sampling The above language does not include a sampling construct, but sampling from the following distributions can be encoded easily: all finite discrete distributions (e.g. Bernoulli, Binomial, Uniform, Categorical), the Geometric and Negative Binomial distributions, as well as shifted versions thereof. Finite discrete distributions can be expressed with branching and the $\mathsf{flip}(\rho)$ event, similarly to $\mathsf{Bernoulli}(\rho)$ shown above. Sampling from a geometric distribution can be expressed as shown above. A negative binomial distribution can be expressed similarly, as a sum of i.i.d. geometrically distributed variables. Sampling from other distributions, such as the Poisson distribution, can also be expressed in principle since our language is Turing-complete, but we are not aware of a simple encoding.

Expressivity Discrete SGCL is similar to the `cReDiP` language in Klinkenberg et al. [2024], but with more restricted sampling. Like `cReDiP`, our language does not support negative integers, continuous distributions/variables, more complex arithmetic like multiplication, or comparisons of two variables. Note that the non-probabilistic fragment of the language is already Turing-complete because it can simulate a three-counter machine. Thus these missing constructs could be encoded in principle, at the expense of program complexity.

2.2.3 SPCF: a Functional PPL

As our second PPL of study, we use ***Statistical PCF (SPCF)*** [Mak et al. 2021a], a typed variant of the probabilistic lambda calculus by Borgström et al. [2016]. SPCF includes

primitive operations which are measurable functions $f : \mathbb{R}^{|f|} \to \mathbb{R}$, where $|f| \geq 0$ denotes the ***arity*** of the function. ***Values*** and ***terms*** of SPCF are defined as follows:

$$\begin{aligned} V &::= x \mid \underline{r} \mid \lambda x.M \mid \mathrm{fix}_x^{\varphi}.\, M \\ M, N, P &::= V \mid M\, N \mid \mathrm{if}\, M \leq 0\, \{N\}\, \mathrm{else}\, \{P\} \mid \underline{f}(M_1, \ldots, M_{|f|}) \\ &\quad \mid \mathrm{sample} \mid \mathrm{score}(M) \end{aligned}$$

where x and φ are variables, f is a (measurable) primitive operation, and $r \in \mathbb{R}$. We use the underline ($\underline{r}$, $\underline{f}$) to distinguish the literals from the mathematical objects. However, we may omit it when the meaning is clear from context. Note that we write $\mathrm{fix}_x^{\varphi}.\, M$ instead of the Y combinator $\mathrm{Y}(\lambda\varphi x.M)$ for the fixpoint construct, which is used to define recursive functions. All values are terms, and application of terms is written by juxtaposition. The branching construct is $\mathrm{if}\, M \leq 0\, \{N\}\, \mathrm{else}\, \{P\}$, which evaluates to N if $M \leq 0$ and P otherwise. In SPCF, sample draws a random value from the uniform distribution on $[0, 1]$, and $\mathrm{score}(M)$ weights the current execution with the value of M (see next paragraph). Samples from a different real-valued distribution D can be obtained by applying the quantile function for D to a uniform sample, if this function is available as a primitive. The set of primitive operations depends on the application; whenever this is relevant in Chapter 3, we will specify the set of primitives.

Continuous observations The score construct is useful to express observations from continuous distributions and deserves more explanation. Observations from discrete distributions are easy to understand: $\mathrm{observe}\, v \sim D$ for a discrete distribution D is equivalent to $X \sim D; \mathrm{observe}\, X = v$. But for continuous distributions, a naive interpretation runs into trouble because then the event $X = v$ has probability 0. This becomes even more complicated for observations that involve several continuous variables, e.g. $X \sim \mathrm{Normal}(0, 1); Y \sim \mathrm{Normal}(0, 1); \mathrm{observe}\, X = Y$. In measure theory, conditioning on such measure-zero events requires the concept of ***disintegration*** [J. T. Chang and Pollard 1997; Pollard 2002], or a limiting process [Piccioni 1983; Fraser et al. 1995]. In the context of probabilistic programming, one has to be careful with measure-zero events to avoid paradoxes [Shan and Ramsey 2017; Jacobs 2021; Stein and Staton 2021].

Most probabilistic programming languages avoid these complications by only supporting continuous observations in the form $\mathrm{observe}\, y \sim D$ where y is a constant value observed from the distribution D. This case can simply be handled by likelihood weighting: multiplying the likelihood by the probability density $\mathrm{pdf}_D(v)$, see the discussion of conditional densities in Section 2.1.[3] The $\mathrm{score}(s)$ construct takes care of this: it multiplies the likelihood weight by the "score" s [Staton 2017]. So the continuous observation $\mathrm{observe}\, v \sim D$ can be expressed as $\mathrm{score}(\underline{\mathrm{pdf}}_D(v))$. In particular, score enables ***soft conditioning*** with a likelihood greater than 1. Of course, hard conditioning is also possible: fail can be expressed as $\mathrm{score}(0)$. However, note the important difference that contrary to fail in SGCL, $\mathrm{score}(0)$ does not abort the program execution, but just sets the likelihood to 0.

[3] Despite its popularity, this approach still has issues in certain situations [Wu et al. 2018; Radul and Alexeev 2021].

$$\frac{}{\Gamma \vdash \underline{r} : \mathbf{R}} \qquad \frac{}{\Gamma, x : \alpha \vdash x : \alpha} \qquad \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta} \qquad \frac{\Gamma, \varphi : \alpha \to \beta, x : \alpha \vdash M : \beta}{\Gamma \vdash \mathrm{fix}^{\varphi}_{x}.\, M : \alpha \to \beta}$$

$$\frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\ N : \beta} \qquad \frac{\{\Gamma \vdash M_i : \mathbf{R}\}_{i=1}^{|f|}}{\Gamma \vdash \underline{f}(M_1, \ldots, M_{|f|}) : \mathbf{R}}$$

$$\frac{\Gamma \vdash M : \mathbf{R} \qquad \Gamma \vdash N : \alpha \qquad \Gamma \vdash P : \alpha}{\Gamma \vdash \mathrm{if}\ M \leq 0\ \{N\}\ \mathrm{else}\ \{P\} : \alpha} \qquad \frac{}{\Gamma \vdash \mathrm{sample} : \mathbf{R}} \qquad \frac{\Gamma \vdash M : \mathbf{R}}{\Gamma \vdash \mathrm{score}(M) : \mathbf{R}}$$

Figure 2.1: Typing rules for SPCF.

Syntactic sugar As for SGCL, we have kept the syntax of SPCF minimal for conceptual simplicity. For practical convenience, we introduce some syntactic sugar for sampling and continuous observations (as discussed above), and for variable binding:

$$\begin{aligned}
\mathrm{let}\ x = M;\ N &\rightsquigarrow (\lambda x.N)M \\
M; N &\rightsquigarrow \mathrm{let}\ _ = M;\ N \\
\mathrm{letrec}\ x\ y = M;\ N &\rightsquigarrow \mathrm{let}\ x = \mathrm{fix}^{x}_{y}.\, M;\ N \\
\mathrm{if}\ M \leq M'\ \{N\}\ \mathrm{else}\ \{P\} &\rightsquigarrow \mathrm{if}\ M - M' \leq 0\ \{N\}\ \mathrm{else}\ \{P\} \\
\{M\}\ [p]\ \{N\} &\rightsquigarrow \mathrm{if}\ \mathrm{sample} \leq \underline{p}\ \{M\}\ \mathrm{else}\ \{N\} \\
\mathrm{sample}(D) &\rightsquigarrow \underline{\mathrm{icdf}_D}(\mathrm{sample}) \\
\mathrm{observe}\ M \sim D &\rightsquigarrow \mathrm{score}(\underline{\mathrm{pdf}_D}(M)) \\
\mathrm{fail} &\rightsquigarrow \mathrm{score}(\underline{0})
\end{aligned}$$

Typing The type system of our language is as expected, with simple types being generated by $\alpha, \beta ::= \mathbf{R} \mid \alpha \to \beta$, where $\mathbf{R}$ denotes the type of real numbers $\mathbb{R}$ and $\alpha \to \beta$ denotes function types. Typing contexts Γ are sets of variable-type pairs. The typing rules are given in Fig. 2.1.

Example 2.2.2 (Die paradox in SPCF)**.** Example 2.2.1 can be written in SPCF as follows, where $\mathbf{1}_A$ is the indicator function of a set A.

$$\begin{aligned}
&\mathrm{letrec}\ \mathit{roll}\ \mathit{throws} = \\
&\quad \mathrm{let}\ \mathit{die} = \mathrm{sample}(\mathrm{Uniform}\{1, \ldots, 6\}); \\
&\quad \mathrm{score}(\mathbf{1}_{\{2,4,6\}}(\mathit{die})); \\
&\quad \mathrm{if}\ \mathit{die} \leq 5\ \{\mathit{roll}\ (\mathit{throws} + 1)\}\ \mathrm{else}\ \{\mathit{throws} + 1\}; \\
&\mathit{roll}\ 0
\end{aligned}$$

For an example with continuous distributions, consider the following problem from Mak et al. [2021a] and Mak et al. [2021b].

Example 2.2.3 (Pedestrian)**.** A pedestrian has gotten lost on a long road and only knows that they are a random distance between 0 and 3 km from their home. They repeatedly walk a uniform random distance of at most 1 km in either direction, until they find their home. When they arrive, a step counter tells them that they have traveled a distance of 1.1 km in total. Assuming that the measured distance is normally distributed around the true distance with standard deviation 0.1 km, what is the posterior distribution of the starting point? We can model this with a probabilistic program:

$$
\begin{aligned}
&\text{let } \mathit{start} = \text{sample } \mathsf{Uniform}(0, 3);\\
&\text{letrec } \mathit{walk}\ x = \text{if } x \leq 0\ \{0\}\ \text{else } \{\\
&\qquad \text{let } \mathit{step} = \text{sample } \mathsf{Uniform}(0, 1);\\
&\qquad \mathit{step} + \mathit{walk}(\{x + \mathit{step}\}\ [0.5]\ \{x - \mathit{step}\})\\
&\quad \};\\
&\text{let } \mathit{distance} = \mathit{walk}\ \mathit{start};\\
&\text{observe } \mathit{distance} \sim \mathsf{Normal}(1.1, 0.1);\\
&\mathit{start}
\end{aligned}
$$

Note that this program does not match the story exactly: if x is negative, *walk* should return x instead of 0. However, we follow the original version by Mak et al. [2021a] for consistency.

2.3 Semantics of PPLs

As for programming languages in general, there are two kinds of semantics for PPLs: ***operational semantics*** and ***denotational semantics***. Operational semantics describe how programs are executed, while denotational semantics describe the meaning of programs, usually as a distribution for PPLs. For the purposes of this thesis, it will be convenient to use a denotational semantics for SGCL and an operational semantics for SPCF.

2.3.1 Transformer Semantics for SGCL

For deterministic imperative programs, a ***transformer semantics*** denotes a program as a function transforming the program state before running the program into the state afterward. For an imperative PPL like SGCL, the semantics of a program is a ***distribution transformer*** [Kozen 1981], i.e. a function transforming a distribution on the state space before running the program into a distribution of the state space afterward.

State space For probabilistic programs without conditioning, the ***state space*** of n variables would be $\mathbb{S} := \mathbb{R}^n_{\geq 0}$ [Kozen 1981]. Unless otherwise stated, we assume that all variables are set to 0 at the start of the program: the initial distribution is $\mu := \mathsf{Dirac}(\mathbf{0}_n)$. Nontermination is expressed by the zero measure, so a program that terminates with probability $t < 1$ denotes a *subprobability* distribution with total mass t. Oftentimes, conditioning is also expressed via subprobability distributions, where a failed observation denotes the zero measure [Hur et al. 2015; Zaiser et al. 2023]. However, the distinction

between nontermination and observation failure can be relevant [Bichsel et al. 2018; Klinkenberg et al. 2024], for instance, to define the termination probability of a program in the presence of conditioning. To distinguish the two kinds of failures, we track observations failures explicitly because this is easier than tracking nontermination. Concretely, we extend the state space of the program with a ***failure state*** $\lightning$, which signifies a failed observation. So the semantics of programs transforms measures on the ***extended state space*** $\mathbb{S}_{\lightning} := \mathbb{R}^n_{\geq 0} \cup \{\lightning\}$. The intuitive idea is that when actually running a probabilistic program, fail moves to the failure state $\lightning$ and aborts the program. For *discrete SGCL* programs, the state space can be restricted to $\mathbb{N}^n$ and the extended state space to $\mathbb{N}^n_{\lightning} := \mathbb{N}^n \cup \{\lightning\}$.

Orders on measures and transformers To define the semantics, we first need to define partial orders on measures and measure transformers, both of which are standard [Kozen 1981]. Given two measures μ, ν on Ω, we say that $\mu \preceq \nu$ if and only if $\mu(S) \leq \nu(S)$ for all sets $S \subseteq \Omega$. On measure transformers $\varphi, \psi : \text{Meas}(\Omega) \to \text{Meas}(\Omega)$, we define the pointwise lifting of this order: $\varphi \sqsubseteq \psi$ if and only if $\varphi(\mu) \preceq \psi(\mu)$ for all $\mu \in \text{Meas}(\Omega)$. Either order is a ***$\omega$-complete partial order (ω-cpo)***, i.e. a partially ordered set where every ω-chain $x_1 \leq x_2 \leq \ldots$ has a least upper bound $\sup_{k \in \mathbb{N}} x_k$.

Semantics The ***standard semantics*** is a straightforward adaptation of Kozen [1981] and Hur et al. [2015] to the extended state space with a failure state. It consists of the semantics of events (Table 2.2a), i.e. how a measure on program states is "restricted" to an event, and the semantics of statements (Table 2.2b), i.e. how each statement affects the measure on program states. Regarding notation, we write $\mu|_{\lightning}$ for $\mu|_{\{\lightning\}}$, i.e. the restriction of μ to the failure state $\lightning$, and conversely $\mu|_{\mathbb{S}}$ for the restriction of μ to $\mathbb{S}$, excluding the failure state $\lightning$.

Event semantics The event semantics describes how a measure μ on program states is "restricted" to an event E (Table 2.2a), suggestively written $\mu|_E$ like the restriction of μ to a subset of $\mathbb{S}_{\lightning}$, even though some events (e.g. flip(ρ)) do not correspond to such a subset. The event $X_k = m$ restricts μ to the set of states where $X_k = m$;. The flip(ρ) event scales μ by the probability ρ of the coin flip coming up heads. The semantics of the complement $\neg E_1$ of an event E_1 is given by subtracting the semantics of E_1 from the original measure (excluding the failure state). Finally, the semantics of an intersection of events $E_1 \wedge E_2$ is the composition of the semantics of E_1 and E_2.

Statement semantics The statement semantics $[\![P]\!]_{\lightning} : \text{Meas}(\mathbb{S}_{\lightning}) \to \text{Meas}(\mathbb{S}_{\lightning})$ describes how a program P transforms the measure on program states (Table 2.2b). The skip command does nothing, so the distribution μ of the program is unchanged. Chaining commands $P_1; P_2$ has the effect of composing the corresponding transformations of μ. Affine transformations of program variables yield a measure of S that is given by the original measure of the preimage of the event S under the affine map. For conditionals if $X_k \in A$ $\{P_1\}$ else $\{P_2\}$, we consider only outcomes satisfying $X_k \in A$ in the first branch and the complement in the second branch and then sum both possibilities. Sampling from distributions $X_k \sim D$ is a bit more complicated. Note that $\mu(S) = \int [\boldsymbol{X} \in S] \, \mathrm{d}\mu(\boldsymbol{X})$.

Event E	Restriction $\mu\vert_E$
$X_k = m$	$S \mapsto \mu(\{\boldsymbol{X} \in S \mid X_k = m\})$
flip(ρ)	$\rho \cdot \mu\vert_{\mathbb{S}}$
$\neg E_1$	$\mu\vert_{\mathbb{S}} - \mu\vert_{E_1}$
$E_1 \wedge E_2$	$(\mu\vert_{E_1})\vert_{E_2}$

(a) Standard semantics of events.

Statement P	Semantics $[\![P]\!]_{\lightning}(\mu)$
skip	μ
$P_1; P_2$	$[\![P_2]\!]_{\lightning}([\![P_1]\!]_{\lightning})(\mu)$
$X_k := \mathbf{a}^\top \boldsymbol{X} + c$	$S \mapsto \mu\vert_{\lightning}(S) + \mu(\{\boldsymbol{X} \in \mathbb{S} \mid \boldsymbol{X}[k \mapsto \mathbf{a}^\top \boldsymbol{X} + c] \in S\})$
$X_k \doteq 1$	$S \mapsto \mu\vert_{\lightning}(S) + \mu(\{\boldsymbol{X} \in \mathbb{S} \mid \boldsymbol{X}[k \mapsto X_k \doteq 1] \in S\})$
$X_k \sim D$	$S \mapsto \mu\vert_{\lightning}(S) + \int\int [\boldsymbol{X}[k \mapsto Y] \in S]\, D(\mathrm{d}Y)\, \mu(\mathrm{d}\boldsymbol{X})$
$X_k \sim D(X_j)$	$S \mapsto \mu\vert_{\lightning}(S) + \int\int [\boldsymbol{X}[k \mapsto Y] \in S]\, (D(X_j))(\mathrm{d}Y)\, \mu(\mathrm{d}\boldsymbol{X})$
if E $\{P_1\}$ else $\{P_2\}$	$\mu\vert_{\lightning} + [\![P_1]\!]_{\lightning}(\mu\vert_E) + [\![P_2]\!]_{\lightning}(\mu\vert_{\neg E})$
while E $\{P_1\}$	$\mathrm{lfp}(\Phi_{E,P_1})(\mu)$
fail	$\mu(\mathbb{S}_{\lightning}) \cdot \mathrm{Dirac}(\lightning)$

(b) Standard semantics of statements.

Table 2.2: Standard semantics for probabilistic programs.

To obtain the measure after sampling $X_k \sim D$, we care about the new state $\boldsymbol{X}[k \mapsto Y]$ being in S, where Y has distribution D. This is exactly expressed by the double integral. The rule for compound distributions $X_k \sim D(X_j)$ is essentially the same as the one for distributions with constant parameters $X_k \sim D$, except the dependency on X_j needs to be respected.

Loops An interesting case is while loops, whose semantics is $[\![\text{while } E\ \{P_1\}]\!]_{\lightning} := \mathrm{lfp}(\Phi_{E,P_1})$ where

$$\Phi_{E,P_1} : (\mathrm{Meas}(\mathbb{S}_{\lightning}) \to \mathrm{Meas}(\mathbb{S}_{\lightning})) \to (\mathrm{Meas}(\mathbb{S}_{\lightning}) \to \mathrm{Meas}(\mathbb{S}_{\lightning}))$$
$$\Phi_{E,P_1}(\psi)(\mu) := \mu\vert_{\lightning} + \mu\vert_{\neg E} + \psi([\![P_1]\!]_{\lightning}(\mu\vert_E))$$

is the unrolling operator and $\mathrm{lfp}(\Phi_{E,P_1})$ denotes its least fixed point with respect to $\sqsubseteq$. Another way to look at it is that $W := [\![\text{while } E\ \{P_1\}]\!]_{\lightning}$ is the least solution to the equation:

$$W(\mu) = \mu\vert_{\lightning} + \mu\vert_{\neg E} + W([\![P_1]\!]_{\lightning}(\mu\vert_E)) \quad \forall \mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$$

By the Kleene fixed-point theorem, the least fixed point is given by

$$\mathrm{lfp}(\Phi_{E,P_1}) = \sup_{n \in \mathbb{N}} \Phi^n_{E,P_1}(\mathbf{0})$$

Conditioning Even if we start with a probability distribution on $\mathbb{S}$, some of the mass may get lost over the course of the program due to nontermination and some

of it may move to the failure state $\lightning$ due to failed observations (fail). Ultimately, we are interested in the distribution *conditioned* on the observations. This conditional probability of the event $\boldsymbol{X} \in U \subseteq \mathbb{S}$ is defined as:

$$\mathbb{P}[\boldsymbol{X} \in U \mid \boldsymbol{X} \neq \lightning] = \frac{\mathbb{P}[\boldsymbol{X} \in U \wedge \boldsymbol{X} \neq \lightning]}{\mathbb{P}[\boldsymbol{X} \neq \lightning]} = \frac{\mathbb{P}[\boldsymbol{X} \in U]}{1 - \mathbb{P}[X = \lightning]}$$

Thus we have to remove the mass on $\lightning$ from the subprobability measure μ of the program and ***normalize*** it, which is achieved by the function normalize:

$$\begin{aligned} \text{normalize} &: \text{Meas}(\mathbb{S}_\lightning) \to \text{Meas}(\mathbb{S}) \\ \text{normalize}(\mu) &:= \frac{\mu - \mu|_\lightning}{1 - \mu(\lightning)} \end{aligned}$$

Thus the posterior probability distribution of a program P with initial distribution μ (usually $\text{Dirac}(\mathbf{0}_n)$) is given by $\text{normalize}(\llbracket P \rrbracket_\lightning(\mu))$.

Operationally, we can think of normalization as a *rejection sampler* of the posterior distribution: it runs the program repeatedly, rejects all runs that end in $\lightning$, and only yields the results of the remaining runs as samples.

Nontermination Due to the possibility of nontermination, even the normalized measure $\text{normalize}(\mu)$ may not be a probability distribution, only a subprobability distribution. For example, the loop $\texttt{while true \{skip\}}$ will never terminate, so both its unnormalized and normalized semantics are always the zero measure. This is not a problem in practice because nontermination is usually considered a bug for statistical models. Tracking observation failures is useful for defining termination, however.

Definition 2.3.1. A program P is ***almost surely terminating (AST)*** on a *finite* measure $\mu \in \text{Meas}(\mathbb{S}_\lightning)$ if $\llbracket P \rrbracket_\lightning(\mu)(\mathbb{S}_\lightning) = \mu(\mathbb{S}_\lightning)$.

Simplified semantics If a program terminates almost surely, the distinction between nontermination and observation failure is unnecessary. To simplify the semantics for this common case, which holds in Chapters 4 and 6, we introduce a variant of the semantics that does not require tracking the failure state $\lightning$. The ***simplified standard semantics*** $\llbracket P \rrbracket : \text{Meas}(\mathbb{S}) \to \text{Meas}(\mathbb{S})$ of a program P is defined as $\llbracket P \rrbracket(\mu) := (\llbracket P \rrbracket_\lightning(\mu))|_\mathbb{S}$ where μ is implicitly extended to $\mathbb{S}_\lightning$, as necessary. It is not difficult to check that the simplified semantics can be defined compositionally like the standard semantics (Table 2.2), by removing any mention of the failure state $\lightning$. The simplified semantics suffices to compute the normalizing constant and the normalized distribution of an AST program. If P terminates almost surely on $\mu \in \text{Meas}(\mathbb{S})$ then

$$\llbracket P \rrbracket_\lightning(\mu)(\lightning) = \llbracket P \rrbracket_\lightning(\mu)(\mathbb{S}_\lightning) - \llbracket P \rrbracket_\lightning(\mu)(\mathbb{S}) = \mu(\mathbb{S}_\lightning) - \llbracket P \rrbracket(\mu)(\mathbb{S})$$

As a consequence, the normalized distribution of a program P terminating almost surely on an initial *probability* distribution μ on $\mathbb{S}$ is given by

$$\text{normalize}(\llbracket P \rrbracket_\lightning(\mu)) = \frac{\llbracket P \rrbracket_\lightning(\mu) - \llbracket P \rrbracket_\lightning(\mu)|_\lightning}{1 - \llbracket P \rrbracket_\lightning(\mu)(\lightning)} = \frac{\llbracket P \rrbracket(\mu)}{\llbracket P \rrbracket(\mu)(\mathbb{S})}$$

Both semantics $[\![P]\!]_{\lightning}$ and $[\![P]\!]$ satisfy the usual properties listed below, analogously to Kozen [1981] and Klinkenberg et al. [2024]. Recall that a map f between two ω-cpos is ω-***continuous*** if it preserves the least upper bound of ω-chains, i.e. if $f(\sup_{k\in\mathbb{N}} x_k) = \sup_{k\in\mathbb{N}} f(x_k)$ for all ω-chains $x_1 \leq x_2 \leq \ldots$.

Lemma 2.3.2. *For any program P, the transformation $[\![P]\!]_{\lightning} : \mathrm{Meas}(\mathbb{S}_{\lightning}) \to \mathrm{Meas}(\mathbb{S}_{\lightning})$ is linear and ω-continuous, so in particular monotonic. It also has the property that the total measure does not increase: $[\![P]\!]_{\lightning}(\mu)(\mathbb{S}_{\lightning}) \leq \mu(\mathbb{S}_{\lightning})$ for all $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$. The transformation $[\![P]\!] : \mathrm{Meas}(\mathbb{S}) \to \mathrm{Meas}(\mathbb{S})$ has the same properties, restricted to measures $\mu \in \mathrm{Meas}(\mathbb{S})$.*

2.3.2 Trace Semantics for SPCF

For a higher-order functional language, denotational semantics are much more complicated due to issues of measure theory around function spaces. There are ways around this, such as quasi-Borel spaces [Heunen et al. 2017] in combination with domain theory [Vákár et al. 2019]. However, for the purposes of this thesis, we take the simpler route of an operational semantics.

Following Borgström et al. [2016], we endow SPCF with a trace-based operational semantics. We evaluate a probabilistic program P on a fixed ***trace*** $\boldsymbol{s} = \langle r_1, \ldots, r_n \rangle$ in the ***trace space*** $\mathbb{T} := \bigcup_{n\in\mathbb{N}} [0,1]^n$, which *predetermines* the probabilistic choices made during the evaluation. Our semantics therefore operates on ***configurations*** of the form $(M, \boldsymbol{s}, w)$ where M is an SPCF term, $\boldsymbol{s}$ is a trace and $w \in \mathbb{R}_{\geq 0}$ a ***weight***. The ***call-by-value reduction*** is given by the rules in Fig. 2.2, where $r\,\boldsymbol{s}$ prepends r to $\boldsymbol{s}$ and $E[\cdot]$ denotes a call-by-value ***evaluation context***. The definition is standard [Borgström et al. 2016; Mak et al. 2021a; Beutner and Ong 2021]. Given a program $\vdash P : \mathbf{R}$, we call a trace $\boldsymbol{s}$ ***terminating*** if $(P, \boldsymbol{s}, 1) \to^* (V, \langle\rangle, w)$ for some value V and weight w, i.e. if the samples drawn are as specified by $\boldsymbol{s}$, the program P terminates. Note that we require the trace $\boldsymbol{s}$ to be completely used up. As P is of type $\mathbf{R}$ we can assume that $V = \underline{r}$ for some $r \in \mathbb{R}$. Each terminating trace $\boldsymbol{s}$ therefore uniquely determines the returned ***value*** $r =: \mathrm{val}_P(\boldsymbol{s}) \in \mathbb{R}$ and the ***weight*** $w =: \mathrm{wt}_P(\boldsymbol{s}) \in \mathbb{R}_{\geq 0}$ of the execution. For a nonterminating trace $\boldsymbol{s}$, $\mathrm{val}_P(\boldsymbol{s})$ is undefined and $\mathrm{wt}_P(\boldsymbol{s}) := 0$.

Example 2.3.3. Consider Example 2.2.3. In desugared form, the program looks like this:

$$
\begin{aligned}
&\mathsf{let}\ \mathit{start} = 3 \times \mathsf{sample};\\
&\mathsf{let}\ \mathit{walk} = \mathsf{fix}^{\varphi}_{x}.\\
&\quad \mathsf{if}\ x \leq 0\ \{0\}\ \mathsf{else}\ \{\\
&\qquad \mathsf{let}\ \mathit{step} = \mathsf{sample};\\
&\qquad \mathit{step} + \varphi(\mathsf{if}\ \mathsf{sample} \leq 0.5\ \{x + \mathit{step}\}\ \mathsf{else}\ \{x - \mathit{step}\})\\
&\quad \};\\
&\mathsf{let}\ \mathit{distance} = \mathit{walk}\ \mathit{start};\\
&\mathsf{score}(\mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}(\mathit{distance}));\\
&\mathit{start}
\end{aligned}
$$

On the trace $\boldsymbol{s} = \langle 0.1, 0.2, 0.4, 0.7, 0.8 \rangle \in [0,1]^5 \subseteq \mathbb{T}$, the pedestrian walks 0.2 away from their home (taking the left branch as $0.4 \leq 0.5$) and 0.7 towards their home (as $0.8 > 0.5$),

$$\frac{}{((\lambda x.M)V, \boldsymbol{s}, w) \to (M[V/x], \boldsymbol{s}, w)} \qquad \frac{}{((\mathsf{fix}_x^{\varphi}.M)V, \boldsymbol{s}, w) \to (M[V/x, (\mathsf{fix}_x^{\varphi}.M)/\varphi], \boldsymbol{s}, w)}$$

$$\frac{r \le 0}{(\mathsf{if}\ \underline{r} \le 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{s}, w) \to (N, \boldsymbol{s}, w)} \qquad \frac{r > 0}{(\mathsf{if}\ \underline{r} \le 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{s}, w) \to (P, \boldsymbol{s}, w)}$$

$$\frac{}{(\mathsf{sample}, r\,\boldsymbol{s}, w) \to (\underline{r}, \boldsymbol{s}, w)} \qquad \frac{r \ge 0}{(\mathsf{score}(\underline{r}), \boldsymbol{s}, w) \to (\underline{r}, \boldsymbol{s}, w \cdot r)}$$

$$\frac{}{(f(\underline{r_1}, \dots, \underline{r_{|f|}}), \boldsymbol{s}, w) \to (\underline{f(r_1, \dots, r_{|f|})}, \boldsymbol{s}, w)} \qquad \frac{(R, \boldsymbol{s}, w) \to (M, \boldsymbol{s}', w')}{(E[R], \boldsymbol{s}, w) \to (E[M], \boldsymbol{s}', w')}$$

Figure 2.2: Standard call-by-value reduction rules for SPCF ($\to$).

hence:

$$\mathrm{val}_P(\boldsymbol{s}) = 3 \times 0.1 = 0.3, \quad \mathrm{wt}_P(\boldsymbol{s}) = \mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}(0.2 + 0.7)$$

In order to do measure theory, we need to turn our set of traces into a measurable space. The trace space $\mathbb{T}$ is equipped with the σ-algebra $\Sigma_{\mathbb{T}} := \{\bigcup_{n\in\mathbb{N}} U_n \mid U_n \in \Sigma_{[0,1]^n}\}$ where $\Sigma_{[0,1]^n}$ is the Borel σ-algebra on $[0,1]^n$. We define a measure $\mu_{\mathbb{T}}$ by $\mu_{\mathbb{T}}(U) := \sum_{n\in\mathbb{N}} \lambda_n(U \cap [0,1]^n)$, as in Borgström et al. [2016].

We can now define the semantics of an SPCF program $\vdash P : \mathbf{R}$ by using the weight and returned value of (executions of P determined by) individual traces. Given $U \in \Sigma_{\mathbb{R}}$, we need to define the likelihood of P evaluating to a value in U. To this end, we set

$$\mathrm{val}_P^{-1}(U) := \{\boldsymbol{s} \in \mathbb{T} \mid (P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w), r \in U\}$$

in other words, the set of traces on which the program P reduces to a value in U. As shown in Borgström et al. [2016, Lem. 9], $\mathrm{val}_P^{-1}(U)$ is measurable. Thus, we can define (cf. Borgström et al. [2016] and Mak et al. [2021a]) the ***unnormalized distribution*** of P as:

$$[\![P]\!](U) := \int_{\mathrm{val}_P^{-1}(U)} \mathrm{wt}_P(\boldsymbol{s})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s})$$

That is, the integral takes all traces $\boldsymbol{s}$ on which P evaluates to a value in U, weighting each $\boldsymbol{s}$ with the weight $\mathrm{wt}_P(\boldsymbol{s})$ of the corresponding execution.

Termination A program P is called ***almost surely terminating (AST)*** if it terminates with probability 1, i.e. $\mu_{\mathbb{T}}(\mathrm{val}_P^{-1}(\mathbb{R})) = 1$. This is a necessary assumption for Monte Carlo inference algorithms because they execute the program. See Borgström et al. [2016] for a more in-depth discussion of this (standard) sampling-style semantics.

Normalization Of course, as for SGCL, we are usually interested in the *normalized posterior*. As before, we define the ***normalizing constant*** as the total measure of $[\![P]\!]$, i.e. $Z_P := [\![P]\!](\mathbb{R})$. Contrary to SGCL, the normalizing constant may be greater than 1,

even infinite! Hence we call P ***integrable*** if $0 < Z_P < \infty$. If P is integrable, we obtain the ***normalized distribution*** or ***posterior distribution*** of P as

$$\text{normalize}(\llbracket P \rrbracket) := \frac{\llbracket P \rrbracket}{\llbracket P \rrbracket(\mathbb{R})}$$

Otherwise the posterior distribution does not exist.

2.4 Exact Inference

As discussed in Section 1.4, computing the exact posterior distribution of a probabilistic program is intractable in general because it requires analytical solutions to infinite sums or integrals [Gehr et al. 2016]. For this reason, existing probabilistic programming systems either restrict the programming language to tractable constructs or cannot guarantee successful inference. The key is to find an exact representation of the posterior distribution that can easily be manipulated symbolically. For example, finite discrete distributions can be represented by their (finite) probability mass function, which is also easy to manipulate. However, infinite discrete distributions and continuous distributions are much harder to handle, which is why most exact inference systems restrict their use. Another problematic construct is unbounded loops, which are not fully supported by any of the existing systems.

Restricted expressivity One way to ensure the feasibility of exact inference is to restrict the class of supported models to a tractable subset where the resulting distributions have a compact representation that is easy to manipulate. This was first studied for graphical models: Shachter et al. [1990] focus on finite discrete distributions, whereas K. Chang and Fung [1995] also allow for Gaussian distributions.

In probabilistic programming, Claret et al. [2013] present an inference algorithm for a PPL with only Boolean variables. It is exact for loop-free programs but can merely approximate the distributions of loops. In their approach, probability distributions are represented as *algebraic decision diagrams (ADDs)*, which support efficient inference.

Dice [Holtzen et al. 2020] is a scalable inference system for finite discrete distributions. It reduces the inference problem to weighted model counting on weighted Boolean formulae. For scalability, these formulae are represented as *binary decision diagrams (BDDs)*, which exploit the structure of the program to factorize the inference problem. Contrary to ADDs used by Claret et al. [2013], BDDs can better capture conditional independence, which makes them typically much smaller and thus more efficient [Holtzen et al. 2020, Section 6.3].

SPPL [Saad et al. 2021] also leverages factorization but uses a different representation, based on sum-product networks. Therefore it can support continuous distributions, but is limited to finite discrete priors and univariate transformations. Both Dice's and SPPL's representations are instances of probabilistic circuits [Choi et al. 2020], which allow for efficient and exact inference.

Genfer (Chapter 4) represents distributions by their generating functions. It supports many prior distributions with infinite support, but lacks nonlinear transformations and observations involving continuous distributions.

Computer algebra Exact inference systems for (almost) unrestricted PPLs typically rely on computer algebra techniques to find a closed-form solution for the posterior distribution. Their drawback is that successful inference is not guaranteed because such a form need not exist in general and, even if it does, the system may fail to find it. Moreover, the running time is unpredictable and unscalable [Saad et al. 2021]. In this category are the systems PSI [Gehr et al. 2016] and Hakaru [Narayanan et al. 2016], which impose almost no syntactic restrictions and support both discrete and continuous distributions.

PSI is a symbolic analysis system to answer queries about the posterior distribution and expected values, using algebraic simplification and symbolic integration. A later extension, λPSI [Gehr et al. 2020], adds support for higher-order functions and nested inference.

Hakaru is a probabilistic programming system allowing for a variety of inference algorithms, but unlike other such systems has support for program transformations and simplifications using the computer algebra system Maple, and for symbolic disintegration [Shan and Ramsey 2017], facilitating exact inference.

A related approach is the compiler by Bhat et al. [2013], which extracts the probability density function of the posterior distribution from a probabilistic program (if it exists), based on the type system by Bhat et al. [2012]. However, it does not simplify integrals symbolically; instead it uses numerical methods.

Loops To our knowledge, ***Prodigy*** [Klinkenberg et al. 2024] is the only system that tackles exact inference for loops. Like Genfer (Chapter 4), Prodigy builds on the idea of generating functions as an exact representations of discrete distributions with infinite support. It can perform exact Bayesian inference on a *single probabilistic loop without nesting*, but only if the user provides a loop invariant template. This template specifies the shape of the loop invariant, but may contain holes that have to be filled with real numbers. Prodigy can synthesize these numbers and verify the resulting invariant, from which it is easy to obtain the exact posterior distribution, parameterized in the program inputs. However, the hard problem of finding the right shape of the loop invariant falls on the user. In fact, even if a loop invariant exists, it cannot always be specified in their language [Klinkenberg et al. 2024, Example 25]. These severe limitations highlight the difficulty of exact inference for loops.

2.5 Approximate Inference

Due to the downsides of exact inference mentioned above, practitioners often use ***approximate inference*** methods. These methods can be applied to a wider range of models and are often more scalable. Most of them follow the Monte Carlo approach: they produce samples from the posterior distribution or samples whose distribution converges to the posterior as the number of samples grows. We discuss such methods in the following. This section is based in part on the excellent introduction by Rainforth [2017].

For the most part, we will discuss these approximate methods in the general setting of Bayesian inference: we assume that we can sample from the prior density $p(\theta)$ and evaluate the likelihood function $p(y \mid \theta)$ for any θ and y. As the observations y are fixed, we also write $\gamma(\theta)$ for the unnormalized posterior density $p(\theta, y)$, and

$\pi(\theta)$ for the normalized posterior density $p(\theta \mid y)$. The normalization constant is $Z := p(y) = \int \gamma(\theta)\,\mathrm{d}\theta$.

In terms of the SPCF semantics, sampling from the prior can be implemented as follows: we run the program P on an initially empty trace $\boldsymbol{s} = \langle\rangle$ and extend it with random values sampled from Uniform$(0,1)$ until P terminates on $\boldsymbol{s}$, i.e. $(P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w)$. Under the assumption that P is AST, the trace extension process ends almost surely. Then $\boldsymbol{s}$ can be viewed as a parameter[4] value θ sampled from the prior distribution $\mathrm{p}(\theta)$ and $w = \mathrm{wt}_P(\boldsymbol{s})$ is its likelihood $p(y \mid \theta)$.

For SGCL, we did not provide an operational semantics, but intuitively the parameters θ correspond to the variable assignment $\boldsymbol{X}$ and we can sample from the prior $p(\theta)$ by running the program like an imperative program.[5] The only differences are that for probabilistic events flip(ρ), we pick the then-path with probability ρ; for sample statements $X_k \sim D$, we assign to X_k a random value drawn from D; and for observation failures fail, we abort the program in the failure state $\lightning$. Then the final assignment to $\boldsymbol{X}$ is the sample from the prior and the likelihood is $p(y \mid \theta) = 0$ if the program ends in the failure state and $p(y \mid \theta) = 1$ otherwise.

2.5.1 Rejection Sampling

Rejection sampling is probably the simplest Monte Carlo inference method and produces exact samples from the posterior distribution. It requires a proposal distribution $q(\theta)$ that is easy to sample from and such that there exists a constant $C > 0$ with $C \cdot \gamma(\theta) \le q(\theta)$ for almost all θ. In probabilistic programming, the proposal $q(\theta)$ is often chosen to be the prior distribution $p(\theta)$.

A rejection sampler now works as follows. It first samples $\hat{\theta} \sim q$ and $\hat{u} \sim$ Uniform$(0, q(\hat{\theta}))$. One can think of $(\hat{\theta}, \hat{u})$ as a uniform sample from the "area" under the graph of q because it has density $q(\hat{\theta}) \cdot \frac{1}{q(\hat{\theta})} = 1$. Next, it *accepts* the sample $\hat{\theta}$ if $\hat{u} \le C \cdot \gamma(\hat{\theta})$, else it *rejects* and discards it. The idea is that the accepted samples $(\hat{\theta}, \hat{u})$ are uniformly distributed in the "area" under the graph of $C \cdot \gamma$. The acceptance probability is

$$\int \int [\hat{u} \le C \cdot \gamma(\hat{\theta})]\,\mathrm{d}\hat{u}\,\mathrm{d}\hat{\theta} = \int C \cdot \gamma(\hat{\theta})\,\mathrm{d}\hat{\theta} = C \cdot Z$$

The density function of accepted samples is thus $\frac{1}{C \cdot Z} \cdot [\hat{u} \le C \cdot \gamma(\hat{\theta})]$. Marginalizing out $\hat{u}$ then yields

$$\int \frac{1}{C \cdot Z} \cdot [\hat{u} \le C \cdot \gamma(\hat{\theta})]\,\mathrm{d}\hat{u} = \frac{1}{C \cdot Z} \cdot C \cdot \gamma(\hat{\theta}) = \frac{1}{Z} \cdot \gamma(\hat{\theta}) = \pi(\hat{\theta})$$

which is the normalized posterior density, as desired.

[4] One could argue that the parameter of interest is actually $r = \mathrm{val}_P(\boldsymbol{s})$ and $\boldsymbol{s}$ is merely a collection of ***latent variables*** that is marginalized out at the end. The terminology does not seem to be consistent in the statistics literature, so we opted for the simpler option here. In any case, $\boldsymbol{s}$ contains enough information to reconstruct r.

[5] As for SPCF, one could argue that only a subset of the variables are the parameters of interest and the rest are latent variables.

The efficiency of rejection sampling critically depends on the value of C because this determines the acceptance rate. In high-dimensional spaces, it is often difficult to find a good proposal distribution that allows for C to be large. However, in low-dimensional models, rejection sampling can be very effective and the fact that it produces exact samples is a big advantage. If the likelihood is bounded (as in SGCL), the prior can be chosen as the proposal distribution and the bound on $p(y \mid x)$ for C. This is the approach taken by most probabilistic programming systems and enables full automation of rejection sampling.

2.5.2 Importance Sampling

Another foundational inference algorithm is (self-normalized) ***importance sampling (IS)***. Unlike rejection sampling, it produces ***weighted samples***, i.e. pairs $(\hat{\theta}_n, w_n)$ where $n \in \{1, \dots, N\}$. One can think of these as approximating the posterior distribution with a weighted sum of Dirac distributions:

$$\pi \approx \sum_{n=1}^{N} w_n \cdot \text{Dirac}(\hat{\theta}_n)$$

For example, to estimate the expected value of a function f under the posterior distribution, one can use the weighted samples:

$$\mathbb{E}_{\theta \sim \pi(\theta)}[f(\theta)] \approx \sum_{n=1}^{N} w_n \cdot f(\hat{\theta}_n)$$

As in rejection sampling, we sample $\hat{\theta}_n \sim q$ from a proposal distribution, but we relax the conditions on q: we only require absolute continuity, i.e. $q(\theta) > 0$ wherever $\gamma(\theta) > 0$. Instead of an accept/reject step, we assign an ***importance weight*** w_n to each sample $\hat{\theta}_n$ to correct for the fact that we are sampling from q instead of π. To figure out the importance weight, we rewrite the expected value $\mathbb{E}[f(\theta)]$:

$$\mathbb{E}_{\theta \sim \pi(\theta)}[f(\theta)] = \int f(\theta)\pi(\theta)\,\mathrm{d}\theta = \int f(\theta)\frac{\gamma(\theta)}{Z \cdot q(\theta)} q(\theta)\,\mathrm{d}\theta \approx \frac{1}{N}\sum_{n=1}^{N} f(\hat{\theta}_n) \cdot \frac{\gamma(\hat{\theta}_n)}{Z \cdot q(\hat{\theta}_n)}$$

Hence we would like to pick the weights $w_n := \frac{\gamma(\hat{\theta}_n)}{N \cdot Z \cdot q(\hat{\theta}_n)}$. While we can easily compute the *unnormalized weights* $w'_n := \frac{\gamma(\hat{\theta}_n)}{q(\hat{\theta}_n)}$, the normalizing constant Z is unknown. However, we can estimate it from the unnormalized weights: $\hat{Z}_N := \frac{1}{N}\sum_{n=1}^{N} w'_n$. This estimator is unbiased:

$$\mathbb{E}[\hat{Z}_N] = \frac{1}{N}\sum_{n=1}^{N} \mathbb{E}[w'_n] = \frac{1}{N}\sum_{n=1}^{N} \int \frac{\gamma(\theta)}{q(\theta)} q(\theta)\,\mathrm{d}\theta = \frac{1}{N}\sum_{n=1}^{N} \gamma(\hat{\theta}_n) = Z$$

Using it, we define the *normalized weights* as $w_n := \frac{w'_n}{N \cdot \hat{Z}_N} = \frac{w'_n}{\sum_{n=1}^{N} w'_n}$. If unweighted samples are required, those can be obtained by *resampling*, i.e. sampling from the distribution $\sum_{n=1}^{N} w_n \cdot \text{Dirac}(\hat{\theta}_n)$, at the expense of higher variance in the estimator.

Importance sampling is more generally applicable than rejection sampling: the prior is always a valid proposal distribution, in which case the unnormalized weights are just the likelihoods. A downside of importance sampling is that its samples are no longer exact, they only converge to the true posterior as $N \to \infty$. In particular, the estimator for $\mathbb{E}_{\theta \sim \pi}[f(\theta)]$ is consistent, but (usually) biased.

2.5.3 Markov Chain Monte Carlo

In high-dimensional spaces, it is often difficult to find good proposal distributions for rejection sampling or importance sampling. It is another instance of the curse of dimensionality that the quality of the samples of these methods (as measured by the effective sample size, see below) decreases exponentially with the dimension [Rainforth 2017]. Fortunately, in many applications, most of the mass of a posterior distribution is concentrated in a small part of the total space, often in the neighborhood of a lower-dimensional manifold around the mode [Betancourt 2018].

Markov chain Monte Carlo (MCMC) algorithms exploit this *unimodality* ("single-peakedness") by making local moves in the parameter space that first "climb" towards the mode and then "explore" the neighborhood around it. The main idea is to construct a Markov chain that converges to the posterior. A ***Markov chain*** is a sequence of random variables $\Theta_0, \Theta_1, \ldots$ such that the distribution of the next variable depends only on the current one: $p(\Theta_{n+1} \mid \Theta_1, \ldots, \Theta_n) = p(\Theta_{n+1} \mid \Theta_n)$. Our Markov chains will be time-homogenous, which means that the transition probability is independent of n, i.e. there is a fixed *transition kernel* $T(\theta_{n+1}, \theta_n)$, also written $T(\theta_{n+1} \leftarrow \theta_n)$ with $p(\Theta_{n+1} = \theta' \mid \Theta_n = \theta) = T(\theta' \leftarrow \theta)$. We want to construct a transition kernel T such that the Markov chain for any starting point θ_0 converges to the target distribution π, i.e. $\lim_{n\to\infty} p(\Theta_n = \theta) = \pi(\theta)$.

Metropolis Hastings (MH) is a method to construct a Markov chain with the desired target distribution $\pi(\theta)$ from a (reasonable) proposal distribution $q(\theta' \leftarrow \theta)$, which depends on the current state θ [Metropolis et al. 1953; Hastings 1970]. MH samples a proposal $\theta' \sim q(\theta' \leftarrow \theta_n)$ based on the current point θ_n, and sets $\theta_{n+1} := \theta'$ (i.e. accepts) with probability $\min(1, \frac{\gamma(\theta')q(\theta_n \leftarrow \theta')}{\gamma(\theta_n)q(\theta' \leftarrow \theta_n)})$ or $\theta_{n+1} = \theta_n$ otherwise (i.e. rejects). Crucially, it only requires access to the unnormalized posterior $\gamma(\theta)$, not the unknown $\pi(\theta)$. Typically, q will propose small random changes to θ_n. But finding a good proposal is nontrivial: if the proposal makes too large steps, the acceptance rate will be too low, but if it makes too small steps, the *correlation* between samples θ_n increases and the accuracy of estimates decreases.

A proposal is called *symmetric* if $q(\theta' \leftarrow \theta) = q(\theta \leftarrow \theta')$. MH with such a symmetric proposal is called ***random-walk Metropolis Hastings (RMH)***. For instance, $q(\theta' \leftarrow \theta) = \text{Normal}(\theta, \sigma^2)$ for some σ is a common choice. ***Hamiltonian Monte Carlo (HMC)*** leverages a sophisticated proposal that exploits the geometry of the posterior distribution via the gradient of its log-density, which enables it to explore the *typical set* of the distribution efficiently [Duane et al. 1987; Betancourt 2018]. The ***No-U-Turn Sampler (NUTS)*** is a widely used extension of HMC that automatically tunes the settings of HMC [Hoffman and Gelman 2014].

A problem with RMH and HMC in the context of probabilistic programming is that the dimension of the proposal θ' can be different from the current point θ because the length of the trace $\boldsymbol{s}$ can vary between executions. One way to address this is to update a single component of the trace θ_m and to resample the rest of the trace after θ_m by rerunning the program. This is known as ***single-site Metropolis Hastings*** [Wingate et al. 2011][6]. The index m of the trace to be updated is usually chosen uniformly at random. If the proposal distribution for θ_m is the prior, this is called ***lightweight MH (LMH)***, which is a common choice for probabilistic programs of varying dimensionality. There is also some work on extending HMC to the probabilistic programming setting with varying dimensionality [Mak et al. 2021a].

All these samplers are called *reversible* because they satisfy the *detailed balance* condition: $\pi(\theta)T(\theta' \leftarrow \theta) = \pi(\theta')T(\theta \leftarrow \theta')$. This criterion ensures that π is a stationary distribution of the Markov chain. Recently, there has been interest in *non-reversible* MCMC methods, which violate detailed balance. They are typically harder to analyze but often converge much faster than reversible chains. ***Involutive MCMC (iMCMC)*** by Neklyudov et al. [2020] is a unifying framework for MCMC algorithms, which facilitates the design of non-reversible samplers from reversible ones and has been automated using probabilistic programming [Cusumano-Towner et al. 2020].

Multimodal distributions A distribution is called ***multimodal*** if it has more than one peak, i.e. high-probability regions separated by low-probability regions. It is well-known that MCMC algorithms struggle with multimodal distributions, such as mixture models [Celeux et al. 2000]. Local proposals make it extremely difficult to move to a different mode of the distribution, leading to very slow convergence. Mangoubi et al. [2018] even found that the popular Hamiltonian Monte Carlo method mixes as slowly as random-walk Metropolis for multimodal targets. Yao et al. [2022, Section 4] give a good account of approaches to deal with multimodality: reparameterization to make the posterior unimodal (e.g. Gorinova et al. [2020]); adding a temperature parameter λ and sampling from the "tempered" distribution $p(\theta \mid y)^\lambda$ to lower the barrier between modes; or averaging multiple chains with weights optimized for prediction error, called "Bayesian stacking". However, none of these approaches are fully satisfactory, and multimodality remains a challenge for MCMC.

2.5.4 Particle-Based Inference

Sequential Monte Carlo (SMC) or ***particle filtering*** is the simplest form of particle-based inference [Doucet et al. 2001]. The key idea is to break the inference problem down into a sequence of *intermediate* distributions $\gamma_1, \ldots, \gamma_T$ that can be viewed as "stepping stones" to the full posterior [Rainforth 2017]. Such a sequential structure is often present in probabilistic models, e.g. the time steps in time series modeling. In the context of probabilistic programming, γ_t typically corresponds to the unnormalized posterior after the t-th observation. At each intermediate step t, SMC samples from a proposal distribution, which may depend on the parameters sampled in previous steps. These samples, also called ***particles***, are propagated to the next step and assigned likelihood

[6] The acceptance probability in Wingate et al. [2011] is wrong, see Rainforth [2017].

weights, as in importance sampling. However, SMC adds a crucial step: it *resamples* unweighted particles from the weighted ones. In this way, it reallocates resources to areas that are expected to have higher posterior density. As a consequence, SMC is much more effective than IS for problems with a rich structure that can be exploited, as is often the case for probabilistic programs.

SMC has many variants and extensions. One important choice is the proposal distribution. The simplest one is the *bootstrap proposal*, which samples from the "prior" transition distribution at each step. Another option is to add MCMC moves to the particles between the steps, which are called *rejuvenation moves*. ***Particle MCMC (PMCMC)***, such as ***particle Gibbs (PGibbs)***, uses SMC to construct a proposal distribution for an MCMC sampler [Andrieu et al. 2009]. ***Interacting Particle MCMC (IPMCMC)*** is based on an interacting pool of SMC samplers [Rainforth et al. 2016].

2.5.5 Other Methods

Variatonal inference The goal of ***variational inference (VI)*** is to approximate the posterior distribution $\pi(\theta)$ with a simpler distribution from a *variational family* $q_\lambda(\theta)$ parameterized by λ. VI frames the inference problem as an optimization problem: minimizing the distance between the true posterior π and the *variational distribution* q_λ (also called *guide*). The most common distance measure is the ***Kullback-Leibler divergence*** $\mathrm{KL}(q_\lambda \parallel \pi) := \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{\pi(\theta)} \, \mathrm{d}\theta$. Minimizing the KL divergence is equivalent to maximizing the ***evidence lower bound (ELBO)*** $\mathbb{E}_{\theta \sim q_\lambda}[\log(\gamma(\theta)) - \log(q_\lambda(\theta))]$, which does not require the normalized distribution. Thus the best approximation q_{λ^*} to the posterior π is a solution to the optimization problem

$$\lambda^* := \arg\max_\lambda \mathbb{E}_{\theta \sim q_\lambda}[\log(\gamma(\theta)) - \log(q_\lambda(\theta))]$$

For the optimization, one could try to use gradient ascent, but computing the gradient of this expectation exactly is usually infeasible. Instead, ***stochastic variational inference (SVI)*** runs ***stochastic gradient descent (SGD)***, which only requires Monte Carlo estimates of the gradient [Hoffman et al. 2013; Ranganath et al. 2014]. To estimate the gradient, one only needs to be able to sample from q_λ and evaluate the densities of $\gamma(\theta)$ and $q_\lambda(\theta)$, as well as their gradients. This makes SVI a popular choice for probabilistic programming in combination with ***automatic differentiation (AD)*** [Kucukelbir et al. 2017]. SVI tends to be faster and scale better than MCMC, but the approximation can be poor if the variational family is not expressive enough. An overview of VI techniques is provided by C. Zhang et al. [2019].

Graphical Models There are many exact and approximate inference algorithms for graphical models that leverage their dependency structure for efficient inference [Bishop 2006, Section 8.4, 10.4.1, 10.7].

Exact inference is feasible when the marginals are tractable, such as with variables that have finite discrete support. ***Variable elimination*** [N. L. Zhang and Poole 1994] can compute the marginal of a single variable conditioned on observations by successively marginalizing out other variables, though this approach is efficient only in sparse models. ***Message passing*** algorithms, on the other hand, compute all marginals simultaneously

by exchanging information, or "messages", between adjacent nodes. *Belief propagation* [Pearl 1988] was the first message-passing algorithm and is applicable to tree-structured (acyclic) models. For general graphs with cycles, *junction tree algorithms* perform belief propagation on a junction tree, a modified graph that eliminates cycles by clustering them into single nodes.

While these methods are exact, there are often more efficient approximate algorithms exploiting the graph structure. Unlike Monte Carlo methods, these approximations are deterministic and do not rely on sampling. For instance, running belief propagation on a graph with cycles, known as *loopy belief propagation* [Frey and MacKay 1997], is no longer exact but can be effective in practice. Other deterministic approximate methods include *expectation propagation* [T. P. Minka 2001] and *variational message passing* [Winn and Bishop 2005], which approximate the posterior distribution through moment matching (adjusting the parameters of approximation to have the correct moments) or variational inference.

Exploiting program structure As probabilistic programs can contain branches, the density function of a program is usually not smooth in all its parameters, but discontinuous in some of them. The language LF-PPL [Zhou et al. 2019] can automatically distinguish continuous and discontinuous parameters, improving the efficiency of inference algorithms such as *discontinuous Hamiltonian Monte Carlo* [Nishimura et al. 2020]. The inference system Qi [Chaganty et al. 2013] exploits the structure of a given probabilistic program for efficient sampling: it uses symbolic execution to determine feasible paths of the program, and in a second step, propagates observations backwards through the program. The Divide, Conquer, and Combine method by Zhou et al. [2020] also decomposes an input program into its paths (what they call straight-line programs), but does so using stochastic sampling instead of program analysis. Through this decomposition, it can achieve much higher accuracy for the posterior of models with varying dimension.

2.5.6 Statistical Checks for Inference

There are a number of statistical checks to validate the results of approximate inference algorithms and to assess the quality of the approximation.

Convergence diagnostics Mature probabilistic programming systems, such as Stan [Carpenter et al. 2017] and Pyro [Bingham et al. 2019], include heuristics to let the user know how well the inference has converged. The most basic one is the ***effective sample size (ESS)***, which measures the information contained in a (weighted) sample set. Initially introduced for importance sampling, it can also be computed for MCMC methods using autocorrelation [Gelman et al. 2013, Section 11.5]. Roughly speaking, balanced weights (in the case of importance sampling) and low correlation of samples (in the case of MCMC) lead to a high effective sample size. Another diagnostic is the ***Gelman-Rubin statistic*** $\hat{R}$, also called ***potential scale reduction factor***. It evaluates the convergence of Markov chains by comparing the variance within a chain with that between the chains [Gelman and Rubin 1992; Brooks and Gelman 1998]. It is a unit-less

ratio that indicates convergence if it is close to 1. However, all these diagnostics are only heuristics: even if posterior samples pass these tests, they may still be far from convergence [Gelman et al. 2013, Section 11.5].

Simulation-based calibration A general approach to validating inference algorithms for a generative Bayesian model is ***simulation-based calibration (SBC)*** [Talts et al. 2018]. SBC draws a sample θ from the prior distribution of the parameters, generates data y for these parameters, and runs the inference algorithm to produce posterior samples $\theta_1, \ldots, \theta_L$ given y. If the posterior samples follow the true posterior distribution, the rank statistic (i.e. position in a sorted list) of the prior sample θ relative to the posterior samples will be uniformly distributed. If the empirical distribution of the rank statistic after many such simulations is non-uniform, this indicates a problem with the inference. While SBC is very general, it is computationally expensive because it performs inference in every simulation. Moreover, as SBC is a stochastic validation approach, any fixed number of samples may fail to diagnose inference errors that only occur in a low-probability region.

Stochastic bounds Grosse et al. [2015] introduce *bidirectional Monte Carlo*, a method to estimate the log marginal likelihood of a model on data *simulated* from the same model, by constructing stochastic lower/upper bounds. They show that the true value can be sandwiched between these two stochastic bounds with high probability.

Grosse et al. [2016] extend this idea to measure the accuracy of the inference algorithms AIS (annealed importance sampling) and SMC by estimating upper bounds (in expectation) on the Jeffreys divergence between the distribution of the approximate samples and the true posterior distribution. To apply this method to real-world data instead of simulated data, they infer hyperparameters on real data, measure the inference quality on simulated data with these hyperparameters, and compare the inference behavior between real and simulated data.

In closely related work, Cusumano-Towner and Mansinghka [2017] present a more general approach, estimating upper bounds (in expectation) on the Jeffreys divergence between the approximating distributions of *two* inference algorithms. In this way, their method can compare the accuracy of certain "target" inference algorithms (AIS, SMC, or VI) with a "gold-standard" inference algorithm (e.g. rejection sampling) on real-world data.

In contrast to these stochastic bounds, the guaranteed bounds in this thesis are deterministic and apply to the posterior distribution directly.

Model checking In the Bayesian workflow, it is essential to evaluate the model fit, i.e. how well the model, with the posterior parameter distribution, describes the observed data [Gelman et al. 2020]. This step goes beyond assessing the quality of the inference in that it also checks the modeling assumptions and can detect model misspecifications. Commonly used techniques include posterior predictive checks, sensitivity analysis of the chosen prior, cross-validation, and others. As the focus of this thesis is Bayesian inference, we will not go into detail here, but refer to Gelman et al. [2013] for a comprehensive overview.

2.6 Formal Methods for Probabilistic Programs

This section reviews various formal verification techniques that have been applied to probabilistic programs. Most of them target randomized programming, but some are relevant to Bayesian inference as well.

The formal analysis of probabilistic programs is significantly more challenging than that of deterministic programs because it requires reasoning about probability distributions instead of individual program states. This exacerbates classical difficulties in program analysis, such as poor scalability to large state spaces or even undecidability. Unbounded loops and recursion are essential for the expressiveness of PPLs, but analyzing their properties like termination is also complexity-theoretically harder than in the deterministic setting [Kaminski et al. 2019].[7] Despite these challenges, a variety of formal methods have been developed to reason about probabilistic programs, which we will explore in the following.

Program logics The first logic to reason about probabilistic programs was *probabilistic propositional dynamic logic* introduced by Kozen [1985]. It can express expected values of program variables and termination probabilities, but no nondeterminism or observations. Since then, a multitude of extensions have been proposed, so we can only scratch the surface in this summary.

Tetsuya Sato et al. [2019] present a framework of three program logics, called PPV, to prove unary and relational properties of (higher-order) probabilistic programs with discrete and continuous variables. This includes reasoning about Monte Carlo approximation, convergence, and probabilistic inference. Among the shortcomings of PPV is the fact that it only allows restricted forms of recursion in the language (no general while-loops).

Similar logics are presented in Barthe et al. [2018] and Aguirre et al. [2018], but these are restricted to discrete distributions. The former, called Ellora [Barthe et al. 2018], is an assertion-based logic for imperative probabilistic programs with support for while loops (contrary to Tetsuya Sato et al. [2019]). The latter [Aguirre et al. 2018] is suitable for reasoning about Markov chains using couplings (a proof technique to compare random variables).

Weakest preexpectation Among program logics, the weakest pre-expectation calculus deserves special mention. The weakest preexpectation calculus for probabilistic programs with nondeterministic choice was introduced by Morgan et al. [1996]. It is based on the idea of expectation transformers, which goes back to Kozen [1985].

The *weakest preexpectation* (wp) of a program P with respect to a function $f : \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ on program states Σ is a function $\mathrm{wp}[\![P]\!](f) : \Sigma \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ mapping each initial program state σ to the expected value of f with respect to the distribution of program states after running P on the initial state σ. The program P is typically written in the ***probabilistic guarded command language (pGCL)***

[7] Even Monte Carlo algorithms are not immune to these difficulties: for example, an almost surely terminating loop may have infinite expected running time, rendering sampling-based approaches impractical since they require executing the program.

[McIver and Morgan 2005], an imperative language with only discrete distributions and without conditioning. However, there are extensions of the weakest preexpectation semantics to Bayesian probabilistic programming with continuous distributions and soft conditioning [Szymczak and Katoen 2019]. Weakest preexpectation calculi have also been used to verify almost sure termination and expected running times (see Kaminski [2019] for an overview). A downside of the wp approach and program logics in general is that they require a lot of manual calculation, although there are early efforts towards automation [Schröer et al. 2023].

Abstract interpretation P. Cousot and R. Cousot [1977] introduced ***abstract interpretation*** as a theoretical framework to soundly approximate a program's behavior. The key idea is to use an ***abstract domain*** as an overapproximation of concrete program states. For instance, sets of concrete real values can be approximated by intervals as abstract states (see Chapter 3).

Abstract interpretation has been extended to the setting of randomized programs. Monniaux [2000] develops an abstract domain which abstracts a distribution as a finite sum of measures, each represented by the region on which it is concentrated and an upper bound on its total mass. Unfortunately, this abstraction sacrifices a lot of precision and is not very useful in practice. The techniques developed in this thesis yield much more precise approximations. Pierro et al. [2005] present a different version of probabilistic abstract interpretation for the probabilistic λ-calculus, which is based on Hilbert spaces and linear operators instead of partially ordered sets as in Cousot's framework. Contrary to the standard approach to abstract interpretation, it is unclear how to extract a probability bound from their approximations [Sankaranarayanan et al. 2013, Section 6]. P. Cousot and Monerau [2012] introduce a unifying and generalizing approach to probabilistic abstract interpretation, which allows lifting any classical analysis method to the probabilistic setting. They also demonstrate that several existing analysis techniques, such as weakest preexpectations, can be viewed as instances of their general framework. All these theoretical developments seem to have had little impact on the practical verification of probabilistic programs.

Smith [2008] presents a more practically useful probabilistic abstract domain: truncated normal distributions. They consider a simple imperative programming language with while loops where all expressions are linear in the input variables and there is only a fixed number of memory locations. The language does not contain any probabilistic primitives itself but simply transforms the input distribution. Since normal distributions are closed under linear transformations, truncated normal distributions are a convenient abstract domain and can be used to obtain bounds on the output distribution of the program.

Symbolic execution Symbolic execution [King 1976] is a program analysis technique that treats inputs as symbolic variables and works with symbolic expressions instead of concrete values during the execution of the program. To handle conditionals, symbolic execution explores all paths through a program and the symbolic states includes symbolic constraints (*path conditions*) that need to hold along that path.

Geldenhuys et al. [2012] extend symbolic execution with path probabilities, which they compute via model counting. Their approach is restricted to finite discrete probabilistic programs with only uniform random variables. The inference system Qi by Chaganty et al. [2013], discussed in Section 2.5, also uses symbolic execution. Sankaranarayanan et al. [2013] use symbolic execution together with approximate volume computations to estimate the probability of assertions (Boolean expressions) holding true during the execution of a probabilistic program with discrete and continuous distributions but without conditioning. Rather than dealing with all (possibly infinitely many) paths, they describe a heuristic to find a finite set of paths that suffices for good estimates of these probabilities. Albarghouthi et al. [2017] extend their approach to compute the probability of events defined by arbitrary SMT constraints but require non-recursive programs without conditioning. In Chapter 3, we apply symbolic execution in a more general setting: probabilistic programs with recursion and conditioning.

Moment-based loop analysis There has been some work on studying probabilistic loop behavior by analyzing the moments of the random variables, in particular, invariant synthesis. Bartocci et al. [2019] consider Prob-solvable loops, which do not have a stopping condition and whose body consists of polynomial assignments with random coefficients. They obtain invariants for the moments of the program variables by solving a system of recurrence equations arising from the fixed structure of the loop body. Amrollahi et al. [2022] extend this method to a larger class of loops, with fewer restrictions on variable dependencies. Moosbrugger et al. [2022] support (restricted) if-statements and state-dependent distribution parameters. They use the moments to bound tail probabilities and can reconstruct the probability masses of the distribution if it has finite support. Kofnov et al. [2024] allow Prob-solvable loops with exponential and trigonometric updates in the loop body, by leveraging the characteristic function.

Cost Analysis & Martingales Martingales are a common technique for cost analysis of probabilistic programs, i.e. computing bounds on the total cost of a program. The cost can be running time (each execution step has unit cost), resource usage, or a utility/reward. A martingale is a sequence of random variables $X_0, X_1, \ldots$ such that $\mathbb{E}[X_i] < \infty$ and $\mathbb{E}[X_{i+1} \mid X_1, \ldots, X_i] = X_i$ for all $i \in \mathbb{N}$. A supermartingale only requires $\mathbb{E}[X_{i+1} \mid X_1, \ldots, X_i] \leq X_i$. A ranking supermartingale (RSM) requires $\mathbb{E}[X_{i+1} \mid X_1, \ldots, X_i] \leq X_i - \varepsilon \cdot [X_n > 0]$ for some $\varepsilon > 0$. (There are small variations in the definitions.)

Chakarov and Sankaranarayanan [2013] introduce the concept of a RSM to prove almost sure termination of probabilistic programs. They aim to find a function $\varphi : \Sigma \to \mathbb{R}$ where Σ denotes the program state space (including the location in the program), such that $X_i := \varphi(S_i)$ is a RSM, where S_i is the state after the i-th execution step. They also require that $X_i > 0$ as long as the program has not terminated at step i. If such a ranking supermartingale exists, the program terminates almost surely. Chatterjee et al. [2016] extend RSMs to probabilistic programs with nondeterminism and use them to derive expected and tail bounds on running times. They define the stopping time $T := \min\{n \mid X_n = 0\}$, which is an upper bound on the number of execution steps until termination. If a RSM exists, the program's expected termination time is $\mathbb{E}[T] \leq \frac{\mathbb{E}[X_0]}{\varepsilon}$. Assuming the martingale is difference-bounded (i.e. $X_{i+1} - X_i \in [a, b]$ for all i), they

derive the tail bound $\mathbb{P}[T > n] \leq \exp\left(-\frac{2(\varepsilon((n-1)-X_1))^2}{(n-1)(b-a)^2}\right)$ from Azuma's inequality. (This is a slight simplification; full details in Chatterjee et al. [2016].)

Kura et al. [2019] also deal with termination probabilities, but extend the method to higher moments of the running time. From bounds on higher moments $\mathbb{E}[T^k]$, they use a variant of Markov's inequality ($\mathbb{P}[T > n] \leq \frac{\mathbb{E}[T^k]}{n^k}$) to obtain polynomial tail bounds on the termination time. They are asymptotically weaker than Chatterjee et al. [2016]'s exponential bounds, but more generally applicable (no need for bounded differences). D. Wang et al. [2021] extend this to bounding variance and higher *central* moments, from which they obtain better tail bounds than Kura et al. [2019] via Chebyshev's inequality and higher-order analogues.

Ngo et al. [2018] extend this to more general (but still nonnegative) costs than just running time, using symbolic methods. P. Wang et al. [2019] even support negative costs and rewards, using martingales again. Both papers yield polynomial bounds on expected costs. Recent work by [Chatterjee et al. 2024] further relaxes the requirements around bounded variable updates and nonnegative costs, and also derives tail cost bounds for probabilistic programs that need not terminate almost-surely.

Mechanizations in proof assistants The semantics of several PPLs have been formalized in interactive theorem provers and used to mechanize proofs of their properties. For instance, the semantics of pGCL has been formalized in HOL by Hurd et al. [2005] and in Isabelle by Cock [2012] and Hölzl [2016], who also developed mechanisms for program verification. Eberl et al. [2015] implement and prove soundness of the density compiler by Bhat et al. [2012] in Isabelle. The program logic Ellora (Barthe et al. [2018], mentioned above) is implemented in the EasyCrypt theorem prover [Barthe et al. 2013], which was originally designed to verify cryptographic protocols. Hirata et al. [2022] formalize a core part of the PPV framework (Tetsuya Sato et al. [2019], see above), again in Isabelle. Tassarotti and Tristan [2023] present a verified compiler for the PPS Stan (see Section 2.7) in the Coq proof assistant. In particular, they prove the correctness of several optimization passes. However, they did not verify the NUTS sampler used by Stan. The Zar system by Bagnall et al. [2023] goes a step further for *discrete* probabilistic programs with loops: it compiles such programs to simple rejection samplers, and is fully verified in the Coq proof assistant.

2.6.1 Applications to Bayesian Inference

Formal methods have also seen applications in Bayesian probabilistic programming, often with the goal to verify or improve Bayesian inference algorithms. The following pieces of work are steps in this direction, but fully verifying inference for probabilistic programs remains an open challenge.

Program transformations Automated analysis and semantics-preserving program transformations have the potential to improve Bayesian inference. R2 [Nori et al. 2014] propagates observations backwards through the program so that they directly follow sample statements. With a custom MH algorithm, they can exploit this structure to avoid rejections, which leads to more efficient inference. Hur et al. [2015] further

improve on this with a technique called *slicing*, which removes parts of the program that are irrelevant for the target distribution. Gorinova et al. [2020] present two dynamic analysis techniques, using effect handlers, to automatically reparameterize probabilistic programs, which improves inference robustness and efficiency. Gorinova et al. [2022] introduce a type system to infer conditional independence properties among parameters in the program distribution. They apply it to inference for mixed discrete-continuous models: using this type system, they design a program transformation to marginalize out discrete parameters efficiently, which facilitates gradient-based inference methods for the remaining continuous parameters.

Verifying inference Lew et al. [2020] target PPSs with programmable inference, such as Venture, Pyro, and Gen (cf. Section 2.7). They introduce a type system ensuring that well-typed inference programs enjoy certain necessary soundness properties by construction. To this end, they introduce *trace types* that track the space of possible execution traces of the program.

Lee et al. [2020] investigate the correctness of variational inference for probabilistic programs. They expose implicit assumptions made by the inference algorithm about the model and describe automatable rules for verifying these assumptions. An implementation of their approach as a static program analysis for the PPS Pyro was able to find a bug in one of Pyro's examples.

There are several program analysis techniques for verifying gradient estimators in the literature, often with applications to variational inference. Khajwal et al. [2023] develop a trace type system to enforce preconditions for the unbiasedness of gradient estimators and correctness of SVI. Lee et al. [2023] describe a static analysis for discovering differentiable subprograms to improve gradient estimates for SVI. Lew et al. [2023] study automatic differentiation of expected values in probabilistic programs more generally. Their type system tracks the differentiability of terms in order to safely expose non-differentiable primitives in their language.

On the applied side, Dutta et al. [2018] extend the concept of compiler fuzzing to probabilistic programming systems. Their fuzzer ProbFuzz was able to find dozens of unknown bugs in the popular PPSs Edward, Pyro, and Stan (see Section 2.7). While fuzzing can only demonstrate the presence of bugs, not their absence, it is a useful tool for improving the robustness of PPSs. Furthermore, it can find numerical issues with floating point arithmetic that is abstracted as exact real arithmetic in many formal verification approaches.

2.7 A Brief History of Probabilistic Programming

We conclude this chapter with a brief overview of existing probabilistic programming languages. Due to the large number of such systems, we focus on the most influential or popular ones, presented roughly in chronological order and grouped by related approaches.

Probabilistic Logic Programming Logic programming languages like Prolog have been extended with probabilistic constructs (see Raedt and Kimmig [2015] for an

overview). ***PRISM*** [Taisuke Sato and Kameya 1997] is one of the earliest probabilistic logic languages. It is an extension of Prolog where facts can be annotated with probabilities, thereby defining a probability distribution over logic programs. Probabilistic inference in this context involves computing the success probability of a query over this distribution, possibly conditioned on certain facts. ***ProbLog*** [Fierens et al. 2015] is a similar extension of Prolog, but relaxes some of PRISM's requirements, such as acyclic rules.

Languages for Graphical Models ***BUGS*** [Gilks et al. 1994] is a statistical software for Bayesian inference on Bayesian networks using MCMC (originally only Gibbs sampling). Its model specification language is one of the earliest PPLs and has been very influential. It is a declarative language designed for Bayesian networks, so it does not support unbounded loops or branching. There have been many implementations of the BUGS language [Lunn et al. 2009], such as WinBUGS, OpenBUGS, and JAGS.

PyMC [Pla et al. 2023], created in the early 2000s, is a PPL embedded in Python for Bayesian networks. It has since gained support for many inference algorithms: SMC, MCMC algorithms including the gradient-based HMC and NUTS, and variational inference.

There are also systems supporting factor graphs, a class of graphical models encompassing Bayesian networks (directed) and Markov random fields (undirected). ***Infer.NET*** [T. Minka et al. 2018], created in 2008, is a library on the .NET platform for such factor graphs and ***Factorie*** [McCallum et al. 2009] is a toolkit for probabilistic modeling with factor graphs in Scala. Both focus on message-passing algorithms for inference, which are specific to graphical models, but also support Gibbs sampling.

More Expressive PPLs ***IBAL*** [Pfeffer 2001] is an early functional PPL with algebraic data types, higher-order functions, and recursion. It supports sampling from finite discrete distributions and conditioning on events. While IBAL is conceptually innovative, its inference method compiles the program to a factor graph and performs variable elimination, which is too slow for practical use.

BLOG [Milch et al. 2005] is a declarative PPL and among the first to move away from the fixed structure of graphical models. It supports *open-universe* probabilistic models where the number of objects is unknown and their identity is uncertain.

Stan [Carpenter et al. 2017], created in 2012, is one of the most mature and popular probabilistic programming systems, widely used in the social, biological, and physical sciences. Its modeling language is an imperative probabilistic programming language for specifying a joint density. The language is inspired by BUGS, but focuses on continuous distributions, so discrete parameters have to be marginalized out manually. Stan supports maximum-likelihood estimation and variational inference, but is best known for popularizing the innovative NUTS sampler. It also provides users with extensive diagnostics for checking the quality of the inference (cf. Section 2.5.6).

Universal Probabilistic Programming Languages ***Church*** [Goodman et al. 2008] has been very influential as the first *universal* PPL, i.e. a Turing-complete programming language extended with probabilistic constructs to express statistical models with an

unbounded number of random variables. Through stochastic memoization, it can express nonparametric models like a Dirichlet process. Church is based on the untyped λ-calculus and implemented in Scheme. For inference, it supports rejection sampling and MH with a simple proposal.

Anglican [Wood et al. 2014] is a universal PPL embedded in Clojure that improves on Church with more efficient inference algorithms, particularly particle-based methods.

WebPPL [Goodman and Stuhlmüller 2014] is a universal PPL for the web, embedded in JavaScript.

Customizable Inference The previous systems come with a fixed selection of built-in inference algorithms. While these may work for a restricted class of problems, more challenging models require customized inference algorithms, typically composed of basic inference algorithms.

Turing [Ge et al. 2018] is a universal PPL embedded in Julia, focusing on efficient inference with composable MCMC kernels based on particle-based methods, HMC, and NUTS.

Edward [Tran et al. 2017] and ***Pyro*** [Bingham et al. 2019] are known as *deep* PPLs: they are designed for complex probabilistic models on large data sets incorporating techniques from deep learning such as neural networks and GPU acceleration. To achieve scalability, their inference algorithms are gradient-based (primarily SVI, but also HMC and NUTS) and composable. The variational family for SVI is also customizable and can be specified in the same PPL. Edward is built on static computation graphs, whereas Pyro allows more dynamic control flow and recursion.

Programmable Inference ***Venture*** [Mansinghka et al. 2014] goes further in terms of inference customization and advocates for *programmable inference* [Mansinghka et al. 2018]. It enables users, among other things, to dynamically decompose inference problems into subproblems and apply composable inference *tactics* (e.g. IS, SMC, MCMC, VI) to these subproblems.

Gen [Cusumano-Towner et al. 2019], a universal PPL embedded in Julia, makes inference programming even more efficient and flexible: for instance, users can write custom proposal distributions for IS, SMC, and MCMC in the same PPL.

3

Guaranteed Bounds via Trace Abstraction

Contents

3.1 Introduction . . . 52
3.1.1 Contributions . . . 54
3.1.2 Limitations . . . 54
3.2 Interval Trace Semantics . . . 55
3.2.1 Interval Arithmetic . . . 55
3.2.2 Interval Traces and Interval SPCF . . . 55
3.2.3 Bounds from Interval Traces . . . 57
3.2.4 Alternative Reduction Rules for Conditionals . . . 59
3.3 Soundness and Completeness . . . 60
3.3.1 Infinite Trace Semantics . . . 60
3.3.2 Soundness . . . 61
3.3.3 Symbolic Execution . . . 62
3.3.4 Completeness . . . 65
3.3.5 Completeness Proof . . . 69
3.4 Weight-aware Interval Type System . . . 74
3.4.1 Interval Types . . . 75
3.4.2 Type System . . . 76
3.4.3 Constraint-based Type Inference . . . 77
3.5 GuBPI and Optimizations for Linear Expressions . . . 78
3.5.1 GuBPI . . . 78
3.5.2 Standard Interval Trace Semantics . . . 80
3.5.3 Linear Interval Trace Semantics . . . 80
3.6 Practical Evaluation . . . 82
3.6.1 Probability Estimation . . . 83
3.6.2 Comparison with Exact Inference . . . 83
3.6.3 Recursive Models . . . 86
3.6.4 Comparison with Statistical Validation . . . 87
3.7 Related Work . . . 88
3.8 Conclusion . . . 90

Summary This chapter proposes a new method to compute *guaranteed* bounds on the posterior distributions of probabilistic programs written in SPCF (Section 2.2.3), a recursive, higher-order probabilistic programming language with continuous distributions. The key idea is to apply abstract interpretation on the trace space of the program, in particular with interval arithmetic. Via an interval-based trace semantics for SPCF, we can compute lower and upper bounds on the posterior distribution, taking the form of super-/subadditive measures (Section 3.2). These lower/upper bounds are non-stochastic and provably correct: using the semantics, we prove that they are *sound*, i.e. the actual posterior of a given program is sandwiched between the lower and upper bounds, and *complete*, i.e. the bounds converge to the posterior (Section 3.3). As a practical and sound approximation, we introduce a weight-aware interval type system, which automatically infers interval bounds on not just the return value but also the weight of program executions, simultaneously (Section 3.4). We have built a tool implementation, called GuBPI, which automatically computes these posterior lower/upper bounds and includes optimizations for linear expressions (Section 3.5). Our evaluation on examples from the literature shows that the bounds are useful, and can even be used to recognize wrong outputs from stochastic posterior inference procedures (Section 3.6).

Publication This chapter is based on a paper published at the *43rd International Conference on Programming Language Design and Implementation (PLDI 2022)*. The full reference is:

> Raven Beutner, Luke Ong, and Fabian Zaiser [2022a]. "Guaranteed bounds for posterior inference in universal probabilistic programming". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pp. 536–551. DOI: 10.1145/3519939.3523721

Supplementary material for the paper is available in Beutner et al. [2022b].

Author contributions The work presented in this chapter was completed in collaboration with Raven Beutner under the supervision of Luke Ong. Roughly speaking, I primarily contributed to the first half of the chapter and Raven Beutner to the second half.

More specifically, I came up with the idea of guaranteed bounds and their motivation, designed the interval semantics (Section 3.2), and proved the soundness and completeness results (Section 3.3). Raven Beutner designed the interval type system (Section 3.4) and the optimizations for linear expressions (Section 3.5). Both of us worked together on the implementation and the experimental evaluation (Section 3.6). Luke Ong supervised the project, provided feedback, and helped write and revise the paper.

3.1 Introduction

The expressivity of universal PPLs comes at the cost of complicating the posterior inference. For instance, Example 2.2.3 (page 29) is a challenging model for inference algorithms in several regards: not only does the program use stochastic branching and recursion, but the number of random variables generated is unbounded. To approximate

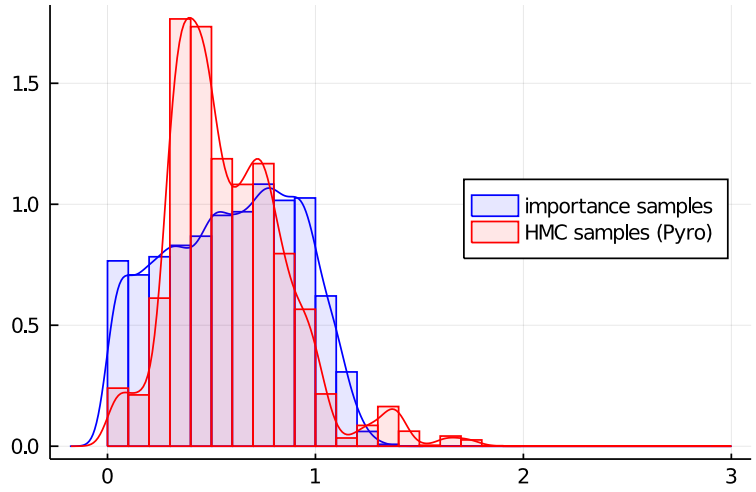


Figure 3.1: Histogram of samples from the posterior distribution of Example 2.2.3 and wrong samples produced by the probabilistic programming system Pyro. The x-axis is the result value of the program (*start*) and the y-axis the estimated posterior probability density.

the posterior distribution of the program, we apply (likelihood-weighted) importance sampling (Section 2.5.2) and Hamiltonian Monte Carlo (HMC) (Section 2.5.3). Figure 3.1 shows the results of the two inference methods as implemented in Anglican [Tolpin et al. 2015] for IS and Pyro [Bingham et al. 2019] for HMC, with the same number of samples: they clearly disagree! But how is the user supposed to know which (if any) of the two results is correct? Note that *exact* inference methods (see Section 2.4) are only applicable to non-recursive models, and so they do not work for Example 2.2.3.

Instead of computing approximate or exact results, this work is concerned with computing *guaranteed* bounds on the posterior distribution of a probabilistic program. Concretely, given a probabilistic program P and a measurable set $U \subseteq \mathbb{R}$ (given as an interval), we infer upper and lower bounds on $[\![P]\!](U)$ (see Section 2.3.2), i.e. the posterior probability of P on U.[1] Such bounds provide a ground truth to compare approximate inference results with: if the approximate results violate the bounds, the inference algorithm has not converged yet or is even ill-suited to the program in question. Crucially, our method is applicable to arbitrary (and in particular recursive) programs of a universal PPL. For Example 2.2.3, the bounds computed by our method are tight enough to separate the IS and HMC output (details in Section 3.6.3). In this case, our method infers that the results given by HMC are wrong (i.e. violate the guaranteed bounds) whereas the IS results are plausible (i.e. lie within the guaranteed bounds). To the best of our knowledge, no previous methods can provide such definite answers for programs of a universal PPL.

[1] We can compute histogram-like bounds by partitioning the domain into intervals and applying our method to each interval to obtain bounds on the height of each bar.

3.1.1 Contributions

The starting point of our work is an interval-based operational semantics [Beutner and Ong 2021]. In our semantics, we evaluate a program on *interval traces* (i.e. sequences of intervals of reals with endpoints between 0 and 1) to approximate the outcomes of sampling, and use interval arithmetic [Moore 1966] to approximate numerical operations (Section 3.2). The semantics is sound in the sense that any (compatible and exhaustive) set of interval traces yields lower and upper bounds on the posterior distribution of a program. These lower/upper bounds are themselves super-/subadditive measures. Moreover, under mild conditions (mostly restrictions on primitive operations), the semantics is also complete, i.e. for any $\varepsilon > 0$ there exists a countable set of interval traces that provides ε-tight bounds on the posterior. Our proofs hinge on a combination of stochastic symbolic execution and the convergence of Riemann sums, providing a natural correspondence between the interval trace semantics and the theory of Riemann integration (Section 3.3).

Based on our interval trace semantics, we present a practical algorithm to automate the computation of guaranteed bounds. It employs an interval type system (together with constraint-based type inference) that bounds both the value of an expression in a refinement-type fashion *and* the score weight of any evaluation thereof. The (interval) bounds inferred by our type system fit naturally in the domain of our semantics. This enables a sound approximation of the behavior of a program with finitely many interval traces (Section 3.4).

We implemented our approach in a tool called ***GuBPI***[2] (**Gu**aranteed **B**ounds for **P**osterior **I**nference), described in Section 3.5, and evaluate it on a suite of benchmark programs from the literature. We find that the bounds computed by GuBPI are competitive in many cases where the posterior could already be inferred exactly. Moreover, GuBPI's bounds are useful (in the sense that they are precise enough to rule out erroneous approximate results as in Fig. 3.1, for instance) for recursive models that could not be handled rigorously by any method before (Section 3.6).

3.1.2 Limitations

While our approach supports very general probabilistic programs, this broad applicability comes at the cost of computational complexity, due to the path explosion problem and the curse of dimensionality. Like other methods that analyze individual program paths, such as symbolic execution, GuBPI struggles with programs containing many branches because the number of paths grows exponentially with the number of branches. Moreover, the curse of dimensionality means that the number of interval traces needed to obtain tight bounds grows quickly with the dimension of the sample space and so does the running time. We believe GuBPI to be most useful for unit-testing of implementations of Bayesian inference algorithms such as Example 2.2.3, or to compute results on probabilistic programs, in particular recursive ones, when non-stochastic, guaranteed bounds are needed (see Section 3.6).

[2]GuBPI (pronounced "guppy") is available at gubpi-tool.github.io.

3.2 Interval Trace Semantics

In order to obtain guaranteed bounds on the distribution denotation $[\![P]\!]$ of a program P, we present an interval-based semantics. This allows us to compute bounds on the normalizing constant Z_P, and thereby also on the normalized denotation. All bounds reported in this chapter (in particular in Section 3.6) refer to the *normalized* denotation.

In our semantics, we approximate the outcomes of sample with intervals and handle arithmetic operations by means of interval arithmetic (which is similar to the approach by Beutner and Ong [2021] in the context of termination analysis). Our semantics enables us to reason about the denotation of a program *without* considering the uncountable space of traces explicitly.

3.2.1 Interval Arithmetic

For our purposes, an ***interval*** has the form $[a, b]$ which denotes the set $\{x \in \mathbb{R} \mid a \leq x \leq b\}$, where $a \in \mathbb{R} \cup \{-\infty\}$, $b \in \mathbb{R} \cup \{\infty\}$, and $a \leq b$. For consistency, we write $[0, \infty]$ instead of the more typical $[0, \infty)$. For $X \subseteq \mathbb{R} \cup \{-\infty, \infty\}$, we denote by $\mathbb{I}_X$ the set of intervals with endpoints in X, and simply write $\mathbb{I}$ for $\mathbb{I}_{\mathbb{R}\cup\{-\infty,\infty\}}$. An n-dimensional ***box*** is the Cartesian product of n intervals.

Intervals $\mathbb{I}$ form a partially ordered set under interval inclusion ($\sqsubseteq$). We will sometimes need the ***meet*** $\sqcap$ and ***join*** $\sqcup$ operations, corresponding to the greatest lower bound and the least upper bound of two intervals. Note that the meet of two intervals does not exist if the two intervals are disjoint. Concretely, these two operations are given by $[a, b] \sqcap [c, d] := [\max(a, c), \min(b, d)]$ (if the two intervals overlap) and $[a, b] \sqcup [c, d] := [\min(a, c), \max(b, d)]$.

For some applications (e.g. the interval type system), we need the interval domain to be a true ***lattice***. To turn $\mathbb{I}$ into a lattice, we add a ***bottom element*** $\bot$ (signifying an empty interval). The definition of the meet $\sqcap$ and join $\sqcup$ is extended in the natural way. The meet $\sqcap$ is extended by defining $I_1 \sqcap \bot = \bot \sqcap I_2 = \bot$ and $I_1 \sqcap I_2 = \bot$ if the two intervals $I_1, I_2 \in \mathbb{I}$ are disjoint. The join $\sqcup$ satisfies $I \sqcup \bot = \bot \sqcup I = I$.

We can lift functions on real numbers to intervals as follows: for each $f : \mathbb{R}^n \to \mathbb{R}$ we define $f^{\mathbb{I}} : \mathbb{I}^n \to \mathbb{I}$ by

$$f^{\mathbb{I}}([a_1, b_1], \ldots, [a_n, b_n]) := [\inf F, \sup F]$$

where $F := f([a_1, b_1], \ldots, [a_n, b_n])$. For common functions like $+$, $-$, $\times$, $|\cdot|$, min, max, and monotonically increasing or decreasing functions $f : \mathbb{R} \to \mathbb{R}$, their interval-lifted counterparts can easily be computed, from the values of the original function on just the endpoints of the input interval (Fig. 3.2).

3.2.2 Interval Traces and Interval SPCF

In our interval interpretation, probabilistic programs are run on *interval traces*. An ***interval trace***, $\langle I_1, \ldots, I_n \rangle \in \mathbb{T}_{\mathbb{I}} := \bigcup_{n\in\mathbb{N}} (\mathbb{I}_{[0,1]})^n$, is a finite sequence of intervals $I_1, \ldots, I_n$, each with endpoints between 0 and 1. To distinguish ordinary traces $\boldsymbol{s} \in \mathbb{T}$ from interval traces $\boldsymbol{t} \in \mathbb{T}_{\mathbb{I}}$, we call the former *concrete* traces.

$$c^{\mathbb{I}} = [c, c] \qquad -^{\mathbb{I}}[x_1, y_1] = [-y_1, -x_1]$$

$$|[x_1, y_1]|^{\mathbb{I}} = \begin{cases} [0, \max(|x_1|, |y_1|)] & \text{if } x_1 \le 0 \le y_1 \\ [\min(|x_1|, |y_1|), \max(|x_1|, |y_1|)] & \text{else} \end{cases}$$

$$[x_1, y_1] +^{\mathbb{I}} [x_2, y_2] = [x_1 + x_2, y_1 + y_2] \qquad [x_1, y_1] -^{\mathbb{I}} [x_2, y_2] = [x_1 - y_2, y_1 - x_2]$$

$$[x_1, y_1] \times^{\mathbb{I}} [x_2, y_2] = [\min(x_1x_2, x_1y_2, y_1x_2, y_1y_2), \max(x_1x_2, x_1y_2, y_1x_2, y_1y_2)]$$

$$\min^{\mathbb{I}}([x_1, y_1], [x_2, y_2]) = [\min(x_1, x_2), \min(y_1, y_2)]$$

$$\max^{\mathbb{I}}([x_1, y_1], [x_2, y_2]) = [\max(x_1, x_2), \max(y_1, y_2)]$$

$$f_{\nearrow}^{\mathbb{I}}([x_1, y_1]) = [f_{\nearrow}(x_1), f_{\nearrow}(y_1)] \qquad f_{\searrow}^{\mathbb{I}}([x_1, y_1]) = [f_{\searrow}(y_1), f_{\searrow}(x_1)]$$

Figure 3.2: Interval-lifted versions of constants $c \in \mathbb{R}$ (i.e. nullary functions), common functions like $+, -, \times, |\cdot|$, min, max, monotonically increasing functions $f_{\nearrow} : \mathbb{R} \to \mathbb{R}$, and monotonically decreasing functions $f_{\searrow} : \mathbb{R} \to \mathbb{R}$.

We define the ***refinement*** relation $\triangleleft$ between concrete and interval traces as follows: for $\boldsymbol{s} = \langle r_1, \dots, r_n \rangle \in \mathbb{T}$ and $\boldsymbol{t} = \langle I_1, \dots, I_m \rangle \in \mathbb{T}_{\mathbb{I}}$, we define $\boldsymbol{s} \triangleleft \boldsymbol{t}$ just if $n = m$ and for all i, $r_i \in I_i$. For each interval trace $\boldsymbol{t}$, we denote by $(\!|\boldsymbol{t}|\!) := \{\boldsymbol{s} \in \mathbb{T} \mid \boldsymbol{s} \triangleleft \boldsymbol{t}\}$ the set of all refinements of $\boldsymbol{t}$.

To define a reduction of a term on an interval trace, we extend SPCF with *interval literals* $\underline{[a, b]}$, which replace the literals $\underline{r}$ but are still considered values of type $\mathbf{R}$. In fact, $\underline{r}$ can be read as an abbreviation for $\underline{[r, r]}$. We call such terms ***interval terms***, and the resulting language ***Interval SPCF***. We can also define a refinement relation $M \triangleleft M'$ ("M refines M'") between a standard term M and an interval term M', if M is obtained from M' by replacing every occurrence of $\underline{[a, b]}$ with some $r \in [a, b]$.

Reduction The interval-based reduction $\to_{\mathbb{I}}$ now operates on configurations $(M, \boldsymbol{t}, w)$ of interval terms M, interval traces $\boldsymbol{t} \in \mathbb{T}_{\mathbb{I}}$, and interval weights $w \in \mathbb{I}_{\mathbb{R}_{\ge 0} \cup \{\infty\}}$. The redexes and evaluation contexts of SPCF extend naturally to interval terms. The reduction rules are given in Fig. 3.3.

For conditionals, the interval bound is not always precise enough to decide which branch to take, so the reduction can get stuck if $a \le 0 < b$. We could include additional rules to overapproximate the branching behavior (see Section 3.2.4). But the rules given here simplify the presentation and are sufficient to prove soundness and completeness.

Given a program $\vdash P : \mathbf{R}$, the reduction relation $\to_{\mathbb{I}}$ allows us to define the *interval weight* function ($\mathrm{wt}_P^{\mathbb{I}} : \mathbb{T}_{\mathbb{I}} \to \mathbb{I}_{\mathbb{R}_{\ge 0} \cup \{\infty\}}$) and *interval value* function ($\mathrm{val}_P^{\mathbb{I}} : \mathbb{T}_{\mathbb{I}} \to \mathbb{I}$) by:

$$\mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}) := \begin{cases} w & \text{if } (P, \boldsymbol{t}, 1) \to_{\mathbb{I}}^* (\underline{I}, \langle\rangle, w) \\ [0, \infty] & \text{otherwise} \end{cases}$$

$$\frac{}{((\lambda x.M)V, \boldsymbol{t}, w) \to_{\mathbb{I}} (M[V/x], \boldsymbol{t}, w)} \qquad \frac{}{((\mathsf{fix}_x^{\varphi}.\, M)V, \boldsymbol{t}, w) \to_{\mathbb{I}} (M[V/x, (\mathsf{fix}_x^{\varphi}.\, M)/\varphi], \boldsymbol{t}, w)}$$

$$\frac{b \le 0}{(\mathsf{if}\ \underline{[a,b]} \le 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{t}, w) \to_{\mathbb{I}} (N, \boldsymbol{t}, w)} \qquad \frac{a > 0}{(\mathsf{if}\ \underline{[a,b]} \le 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{t}, w) \to_{\mathbb{I}} (P, \boldsymbol{t}, w)}$$

$$\frac{}{(\mathsf{sample}, I\,\boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{I}, \boldsymbol{t}, w)} \qquad \frac{a \ge 0}{(\mathsf{score}(\underline{[a,b]}), \boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{[a,b]}, \boldsymbol{t}, w \times^{\mathbb{I}} [a,b])}$$

$$\frac{}{(\underline{f}(\underline{I_1}, \ldots, \underline{I_{|f|}}), \boldsymbol{t}, w) \to_{\mathbb{I}} (\underline{f^{\mathbb{I}}(I_1, \ldots, I_{|f|})}, \boldsymbol{t}, w)} \qquad \frac{(R, \boldsymbol{t}, w) \to_{\mathbb{I}} (M, \boldsymbol{t}', w')}{(E[R], \boldsymbol{t}, w) \to_{\mathbb{I}} (E[M], \boldsymbol{t}', w')}$$

Figure 3.3: Interval reduction rules for (interval) SPCF ($\to_{\mathbb{I}}$).

$$\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) := \begin{cases} I & \text{if } (P, \boldsymbol{t}, 1) \to_{\mathbb{I}}^{*} (\underline{I}, \langle\rangle, w) \\ [-\infty, \infty] & \text{otherwise} \end{cases}$$

Lemma 3.2.1. *Let $\vdash P : \mathbf{R}$ be a program. For any interval trace $\boldsymbol{t}$ and concrete trace $\boldsymbol{s} \triangleleft \boldsymbol{t}$, we have $\mathrm{wt}_P(\boldsymbol{s}) \in \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t})$ and $\mathrm{val}_P(\boldsymbol{s}) \in \mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t})$ (provided $\mathrm{val}_P(\boldsymbol{s})$ is defined).*

Proof. If the interval reduction $\to_{\mathbb{I}}$ gets stuck, $\mathrm{wt}_P^{\mathbb{I}}$ is $[0, \infty]$ and $\mathrm{val}_P^{\mathbb{I}}$ is $[-\infty, \infty]$, so the claim is certainly true. Otherwise, for each $(M_{\mathbb{I}}, \boldsymbol{t}, w_{\mathbb{I}}) \to_{\mathbb{I}} (M'_{\mathbb{I}}, \boldsymbol{t}', w'_{\mathbb{I}})$ reduction step, we can do a reduction step $(M, \boldsymbol{s}, w) \to (M', \boldsymbol{s}', w')$ where $M' \triangleleft M'_{\mathbb{I}}$ and $w' \in w'_{\mathbb{I}}$, and $\boldsymbol{s}' \triangleleft \boldsymbol{t}'$ if $M \triangleleft M_{\mathbb{I}}$, $w \in w_{\mathbb{I}}$, and $\boldsymbol{s} \triangleleft \boldsymbol{t}$. Since the reduction does not get stuck, we end up with a value $\underline{r} \triangleleft \underline{[a,b]}$, so $\mathrm{val}_P(\boldsymbol{s}) = r \in [a,b] = \mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t})$. □

3.2.3 Bounds from Interval Traces

Lower bounds How can we use this interval trace semantics to obtain lower bounds on $[\![P]\!]$? We need a few definitions. Two intervals $[a_1, b_1], [a_2, b_2] \in \mathbb{I}$ are called ***almost disjoint*** if $b_1 \le a_2$ or $b_2 \le a_1$. Interval traces $\langle I_1, \ldots, I_m\rangle$ and $\langle J_1, \ldots, J_n\rangle \in \mathbb{T}_{\mathbb{I}}$ are called ***compatible*** if there is an index $i \in \{1, \ldots, \min(m,n)\}$ such that I_i and J_i are almost disjoint. A set of interval traces is called *compatible* if its elements are pairwise compatible. We define the ***volume*** of an interval trace $\boldsymbol{t} = \langle [a_1, b_1], \ldots, [a_n, b_n]\rangle$ as $\mathrm{vol}(\boldsymbol{t}) := \prod_{i=1}^{n} (b_i - a_i)$.

Let $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ be a countable and compatible set of interval traces. Define the *lower bound* on $[\![P]\!]$ by

$$\mathrm{lowerBd}_P^{\mathcal{T}}(U) := \sum_{\boldsymbol{t} \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t}) \cdot (\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t})) \cdot \left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \subseteq U\right]$$

for $U \in \Sigma_{\mathbb{R}}$. That is, we sum over each interval trace in $\mathcal{T}$ whose value is *guaranteed* to be in U, weighted by its volume and the lower bound of its weight interval.

Note that, in general, $\mathrm{lowerBd}_P^{\mathcal{T}}$ is not a measure, but merely a *superadditive* measure. A ***superadditive measure*** μ on (Ω, Σ_Ω) is a measure, except that σ-additivity is

replaced by σ-superadditivity: $\mu(\bigcup_{i\in\mathbb{N}} U_i) \geq \sum_{i\in\mathbb{N}} \mu(U_i)$ for a countable, pairwise disjoint family $(U_i)_{i\in\mathbb{N}} \in \Sigma_\Omega$.

Exhaustivity For upper bounds, we require the notion of a set of interval traces being *exhaustive*, which is easiest to express in terms of infinite traces. Let $\mathbb{T}_\infty := [0,1]^\omega$ be the set of ***infinite traces***. The σ-algebra on $\mathbb{T}_\infty$ is defined as the smallest σ-algebra that contains all sets $U \times \mathbb{T}_\infty$ where $U \in \Sigma_{[0,1]^n}$ for some $n \in \mathbb{N}$. The measure $\mu_{\mathbb{T}_\infty}$ is the unique measure with $\mu_{\mathbb{T}_\infty}(U \times \mathbb{T}_\infty) = \lambda_n(U)$ when $U \in \Sigma_{[0,1]^n}$. Every interval trace $\boldsymbol{t}$ *covers* the set of infinite traces with a prefix contained in $\boldsymbol{t}$, i.e. $\mathit{cover}(\boldsymbol{t}) := (\!|\boldsymbol{t}|\!) \times \mathbb{T}_\infty$ where the Cartesian product $\times$ can be viewed as trace concatenation.

A countable set of (finite) interval traces $\mathcal{T} \subseteq \mathbb{T}_\mathbb{I}$ is called ***exhaustive*** if $\bigcup_{\boldsymbol{t}\in\mathcal{T}} \mathit{cover}(\boldsymbol{t})$ covers almost all of $\mathbb{T}_\infty$, i.e. $\mu_{\mathbb{T}_\infty}(\mathbb{T}_\infty \setminus \bigcup_{\boldsymbol{t}\in\mathcal{T}} \mathit{cover}(\boldsymbol{t})) = 0$. Phrased differently, almost all concrete traces must have a finite prefix that is contained in some interval trace in $\mathcal{T}$. Therefore, the analysis in the interval semantics on $\mathcal{T}$ covers the behavior on almost all concrete traces (in the original semantics).

Example 3.2.2. Here are several examples and counterexamples for exhaustivity.

(i) $\{\langle\rangle\}$ is an (uninteresting) exhaustive set and only useful for deterministic programs.

(ii) The singleton set $\{\langle[0,1],[0,0.6]\rangle\}$ is *not* exhaustive as, e.g. all infinite traces $\langle r_1, r_2, \dots\rangle$ with $r_2 > 0.6$ are not covered.

(iii) The set $\{\langle[0,0.6]\rangle, \langle[0.3,1]\rangle\}$ is exhaustive, but not compatible.

(iv) $\{\langle[2^{-n-1}, 2^{-n}]\rangle \mid n \in \mathbb{N}\}$ is exhaustive because only the trace $\langle 0\rangle$ (with measure 0) is not covered.

(v) Define $\mathcal{T}_1 := \{\langle[\frac{1}{2},1]^{\cdots n}, [0,\frac{1}{3}]\rangle \mid n \in \mathbb{N}\}$ and $\mathcal{T}_2 := \{\langle[\frac{1}{2},1]^{\cdots n}, [0,\frac{1}{2}]\rangle \mid n \in \mathbb{N}\}$ where $x^{\cdots n}$ denotes n-fold repetition of x. $\mathcal{T}_1$ is compatible but not exhaustive. For example, it does not cover the set $[\frac{1}{2},1] \times (\frac{1}{3},\frac{1}{2}) \times \mathbb{T}_\infty$, i.e. all traces $\langle r_1, r_2, \dots\rangle$ where $r_1 \in [\frac{1}{2},1]$ and $r_2 \in (\frac{1}{3},\frac{1}{2})$. $\mathcal{T}_2$ is compatible and exhaustive because the set of non-covered traces $(\frac{1}{2},1]^\omega$ has measure 0.

(vi) Let $a_n \geq 0$ be a converging series, i.e. $\sum_{i=1}^n a_n < \infty$, for example $a_n = n^{-2}$. Define

$$\mathcal{T} := \{\langle[0,e^{-a_1}], \dots, [0,e^{-a_n}], [e^{-a_{n+1}},1]\rangle \mid n \in \mathbb{N}\}$$

This is not an exhaustive set since it does not cover any of the traces in $[0,e^{-a_1}) \times [0, e^{-a_2}) \times \cdots$, which has measure $\prod_{i=1}^\infty e^{-a_i} = \exp\left(-\sum_{i=1}^\infty a_i\right) > 0$.

We also note that exhaustivity can be expressed just in terms of finite traces as well, at the cost of a more complicated definition.

Lemma 3.2.3. *A set of interval traces $\mathcal{T}$ is exhaustive if and only if*

$$\mu_\mathbb{T}\left([0,1]^n \setminus \left(\bigcup_{\langle I_1,\dots,I_m\rangle\in\mathcal{T}, m\leq n} I_1 \times \cdots \times I_m \times [0,1]^{n-m}\right)\right) \to 0$$

as $n \to \infty$.

Proof. Let $S := \mathbb{T}_\infty \setminus \bigcup_{t \in \mathcal{T}} \mathit{cover}(\boldsymbol{t})$. By the definition of exhaustivity, $\mu_{\mathbb{T}_\infty}(S) = 0$. Let

$$S_n = [0,1]^n \setminus \left(\bigcup_{\langle I_1, \ldots, I_m \rangle \in \mathcal{T}, m \leq n} I_1 \times \cdots \times I_m \times [0,1]^{n-m} \right)$$

It is easy to see that $S = \bigcap_{n=0}^{\infty} S_n \times \mathbb{T}_\infty$ where $S_n \times \mathbb{T}_\infty$ is a decreasing sequence of sets: $S_1 \times \mathbb{T}_\infty \supseteq S_2 \times \mathbb{T}_\infty \supseteq \cdots$. Since measures are continuous from above, we have $\lim_{n\to\infty} \mu_{\mathbb{T}}(S_n) = \lim_{n\to\infty} \mu_{\mathbb{T}_\infty}(S_n \times \mathbb{T}_\infty) = \mu_{\mathbb{T}_\infty}(S) = 0$, as desired. □

Upper bounds Let $\mathcal{T} \subseteq \mathbb{T}_\mathbb{I}$ be a countable and exhaustive set of interval traces. Define the *upper bound* on $[\![P]\!]$ by

$$\mathrm{upperBd}_P^{\mathcal{T}}(U) := \sum_{\boldsymbol{t} \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t}) \cdot (\sup \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t})) \cdot \left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \cap U \neq \varnothing \right]$$

for $U \in \Sigma_\mathbb{R}$. That is, we sum over each interval trace in $\mathcal{T}$ whose value *may* be in U, weighted by its volume and the upper bound of its weight interval.

Note that $\mathrm{upperBd}_P^{\mathcal{T}}$ is not a measure but only a *subadditive* measure. A ***subadditive measure*** μ on (Ω, Σ_Ω) is a measure, except that σ-additivity is replaced by σ-subadditivity: $\mu(\bigcup_{i\in\mathbb{N}} U_i) \leq \sum_{i\in\mathbb{N}} \mu(U_i)$ for a countable, pairwise disjoint family $(U_i)_{i\in\mathbb{N}} \in \Sigma_\Omega$.

3.2.4 Alternative Reduction Rules for Conditionals

The interval semantics as presented has the unfortunate property that even a simple program like

$$\mathsf{if}\ \mathsf{sample} \leq 0\ \{\mathsf{score}(0)\}\ \mathsf{else}\ \{\mathsf{score}(1)\}$$

requires infinitely many interval traces to achieve a finite upper bound. The reason is that the right branch $\mathsf{score}(1)$ is taken if the sampled value is in the open interval $(0,1]$. To approximate this using closed intervals $[a,b]$ that our analysis supports, we need infinitely many intervals, e.g. $\{[2^{-n-1}, 2^{-n}] \mid n \in \mathbb{N}\}$. Adding (half-)open intervals to the semantics would solve this specific problem, but not more general ones, where the guard condition is for example $\mathsf{sample} - \mathsf{sample} \leq 0$. In that case, we have to approximate the set $\{(x,y) \in [0,1]^2 \mid x \leq y\}$. For the lower bounds, that is not an issue, but for the upper bounds, we need an infinite number of interval traces again. We would like to use the interval traces $\langle [0, \frac{1}{2}], [0, \frac{1}{2}] \rangle$ and $\langle [\frac{1}{2}, 1], [0,1] \rangle$ to cover this set, but the reduction gets stuck on them because it is not clear which branch should be taken.

To remedy this, we could add the following two rules.

$$\frac{a \leq 0 < b}{(\mathsf{if}\ \underline{[a,b]} \leq 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{t}, w) \to_\mathbb{I} (N, \boldsymbol{t}, w \times^{\mathbb{I}} [0,1])}$$

$$\frac{a \leq 0 < b}{(\mathsf{if}\ \underline{[a,b]} \leq 0\ \{N\}\ \mathsf{else}\ \{P\}, \boldsymbol{t}, w) \to_\mathbb{I} (P, \boldsymbol{t}, w \times^{\mathbb{I}} [0,1])}$$

They basically express that if the interval bounds are not precise enough to decide what branch to take, we can take both, but have to allow the weight to be zero because it is not guaranteed that the taken branch can actually happen. This change can only improve the upper bounds, not the lower bounds because the lower bound on each weight is zero if the additional rules are used. Then the definition of upper bound can be modified in the following way:

$$\mathrm{upperBd}_P^{\mathcal{T}}(U) := \sum_{\boldsymbol{t}\in\mathcal{T}} \sum_{\substack{(P,\boldsymbol{t},[1,1])\to_{\mathbb{I}}^* \\ (\underline{[a,b]},\langle\rangle,[w_1,w_2])}} \mathrm{vol}(\boldsymbol{t}) \cdot w_2 \cdot \left[[a,b] \cap U \neq \varnothing\right]$$

This is the strategy we use for our implementation and is a natural extension of the existing semantics: it requires very few changes to the soundness and completeness proofs. In the presence of recursion, there may be infinitely many reduction sequences, so the sum has to be (soundly) approximated, like in Section 3.4.

A downside of the previous approach is that the bounds are not always very tight: for the term if ... {score(50)} else {score(100)}, it returns bounds $[0, 150]$ instead of $[50, 100]$. To improve this, we could omit the multiplication with $[0, 1]$.

$$\frac{a \le 0 < b}{(\text{if } \underline{[a,b]} \le 0 \{N\} \text{ else } \{P\}, \boldsymbol{t}, w) \to_{\mathbb{I}} (N, \boldsymbol{t}, w)}$$

$$\frac{a \le 0 < b}{(\text{if } \underline{[a,b]} \le 0 \{N\} \text{ else } \{P\}, \boldsymbol{t}, w) \to_{\mathbb{I}} (P, \boldsymbol{t}, w)}$$

However, this complicates the equations of our bounds. With this semantics, we have to compute infima and suprema instead of a simple sum:

$$\mathrm{lowerBd}_P^{\mathcal{T}}(U) := \sum_{\boldsymbol{t}\in\mathcal{T}} \inf_{\substack{(P,\boldsymbol{t},[1,1])\to_{\mathbb{I}}^* \\ (\underline{[a,b]},\langle\rangle,[w_1,w_2])}} \mathrm{vol}(\boldsymbol{t}) \cdot w_1 \cdot \left[[a,b] \subseteq U\right]$$

$$\mathrm{upperBd}_P^{\mathcal{T}}(U) := \sum_{\boldsymbol{t}\in\mathcal{T}} \sup_{\substack{(P,\boldsymbol{t},[1,1])\to_{\mathbb{I}}^* \\ (\underline{[a,b]},\langle\rangle,[w_1,w_2])}} \mathrm{vol}(\boldsymbol{t}) \cdot w_2 \cdot \left[[a,b] \cap U \neq \varnothing\right]$$

This would be harder to implement due to the computation of infima and suprema instead of sums. In the presence of recursion, there may even be infinitely many reduction sequences, so the infimum and supremum would have to be (soundly) approximated, similarly to Section 3.4. Proving soundness and completeness for these bounds would require more substantial changes to the proofs.

3.3 Soundness and Completeness

3.3.1 Infinite Trace Semantics

A convenient alternative to the (finite) trace semantics is using infinite traces [Culpepper and Cobb 2017; Kenyon-Roberts and Ong 2021]. We use the symbol $\boldsymbol{u}$ for an infinite trace

in $\mathbb{T}_\infty$. For a finite trace $\boldsymbol{s}$ and infinite trace $\boldsymbol{u}$ we write $\boldsymbol{su} \in \mathbb{T}_\infty$ for their concatenation. For any infinite trace $\boldsymbol{u} \in \mathbb{T}_\infty$, there is at most one prefix $\boldsymbol{s} \in \mathbb{T}$ with $\mathrm{wt}_P(\boldsymbol{s}) > 0$ since the reduction is deterministic. We can therefore define $\mathrm{wt}_P^\infty(\boldsymbol{u}) := \mathrm{wt}_P(\boldsymbol{s})$ and $\mathrm{val}_P^\infty(\boldsymbol{u}) := \mathrm{val}_P(\boldsymbol{s})$ if such a prefix $\boldsymbol{s}$ exists, and $\mathrm{wt}_P^\infty(\boldsymbol{u}) := 0$ and $\mathrm{val}_P^\infty(\boldsymbol{u})$ is undefined otherwise. The infinite trace semantics of a term is then defined as

$$[\![P]\!](U) := \int_{(\mathrm{val}_P^\infty)^{-1}(U)} \mathrm{wt}_P^\infty(\boldsymbol{u})\, \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u})$$

Lemma 3.3.1. *The finite and infinite trace semantics agree, that is:*

$$\int_{(\mathrm{val}_P)^{-1}(U)} \mathrm{wt}_P(\boldsymbol{s})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s}) = \int_{(\mathrm{val}_P^\infty)^{-1}(U)} \mathrm{wt}_P^\infty(\boldsymbol{u})\, \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u})$$

Proof. Observe that $\mathrm{val}_P^\infty(\boldsymbol{su}) = \mathrm{val}_P(\boldsymbol{s})$ and $\mathrm{wt}_P^\infty(\boldsymbol{su}) = \mathrm{wt}_P(\boldsymbol{s})$ for all $\boldsymbol{u} \in \mathbb{T}_\infty$ if $\mathrm{wt}_P(\boldsymbol{s}) > 0$. Then we get:

$$\begin{aligned}
\int_{(\mathrm{val}_P)^{-1}(U)} \mathrm{wt}_P(\boldsymbol{s})\, \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s}) &= \int_{(\mathrm{wt}_P)^{-1}(\mathbb{R}_{>0})} [\mathrm{val}_P(\boldsymbol{s}) \in U] \mathrm{wt}_P(\boldsymbol{s}) \int_{\mathbb{T}_\infty} \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u}) \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s}) \\
&= \int_{(\mathrm{wt}_P)^{-1}(\mathbb{R}_{>0})} \int_{\mathbb{T}_\infty} [\mathrm{val}_P^\infty(\boldsymbol{su}) \in U] \mathrm{wt}_P^\infty(\boldsymbol{su}) \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u}) \mu_{\mathbb{T}}(\mathrm{d}\boldsymbol{s}) \\
&= \int_{(\mathrm{wt}_P)^{-1}(\mathbb{R}_{>0}) \times \mathbb{T}_\infty} [\mathrm{val}_P^\infty(\boldsymbol{u}) \in U] \mathrm{wt}_P^\infty(\boldsymbol{u}) \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u}) \\
&= \int_{(\mathrm{val}_P^\infty)^{-1}(U)} \mathrm{wt}_P^\infty(\boldsymbol{u}) \mu_{\mathbb{T}_\infty}(\mathrm{d}\boldsymbol{u})
\end{aligned}$$

where we used the fact that the sets $\{\boldsymbol{s}\} \times \mathbb{T}_\infty$ are disjoint for different $\boldsymbol{s} \in \mathrm{wt}_P^{-1}(\mathbb{R}_{>0})$ because otherwise we would be able to find a trace $\boldsymbol{s}$ as a prefix of $\boldsymbol{s}'$ and both having positive weight, which is impossible due to the deterministic reduction. Therefore $(\mathrm{wt}_P)^{-1}(\mathbb{R}_{>0}) \times \mathbb{T}_\infty = (\mathrm{wt}_P^\infty)^{-1}(\mathbb{R}_{>0})$ and everything works as desired. Note that the second to last equality follows from Fubini's theorem and the fact that the product measure of $\mu_{\mathbb{T}}$ and $\mu_{\mathbb{T}_\infty}$ is $\mu_{\mathbb{T}_\infty}$ again. □

The following lemma establishes a correspondence between infinite trace semantics and interval trace semantics.

Lemma 3.3.2. *For any interval trace $\boldsymbol{t}$ and infinite trace $\boldsymbol{s}_\infty$ with a prefix $\boldsymbol{s}$ such that $\boldsymbol{s} \triangleleft \boldsymbol{t}$, we have $\mathrm{wt}_P^\infty(\boldsymbol{s}_\infty) \in \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t})$ and $\mathrm{val}_P^\infty(\boldsymbol{s}_\infty) \in \mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t})$.*

Proof. Follows directly from the definition of infinite trace semantics and Lemma 3.2.1. □

3.3.2 Soundness

We show that the two bounds described above are *sound*, in the following sense.

Theorem 3.3.3 (Sound lower bounds)**.** *Let $\mathcal{T}$ be a countable and compatible set of interval traces and $\vdash P : \mathbf{R}$ a program. Then* $\mathsf{lowerBd}_P^{\mathcal{T}} \leq [\![P]\!]$.

Proof. For any $U \in \Sigma_{\mathbb{R}}$, we have:

$$\begin{aligned}
\mathsf{lowerBd}_P^{\mathcal{T}}(U) &= \sum_{t \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t})(\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))\left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \subseteq U\right] \\
&= \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} (\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))\left[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \subseteq U\right] \mathrm{d}\boldsymbol{s} \\
&\leq \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} \qquad (3.1) \\
&= \int_{\bigcup_{t \in \mathcal{T}} (\!|\boldsymbol{t}|\!)} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} \qquad (3.2) \\
&\leq \int_{\mathbb{T}} \mathrm{wt}_P(\boldsymbol{s})\left[\mathrm{val}_P(\boldsymbol{s}) \in U\right] \mathrm{d}\boldsymbol{s} = [\![P]\!](U) \qquad (3.3)
\end{aligned}$$

where Eq. (3.1) follows from Lemma 3.2.1, Eq. (3.2) from compatibility, and Eq. (3.3) from $\bigcup_{t \in \mathcal{T}} (\!|\boldsymbol{t}|\!) \subseteq \mathbb{T}$. □

Theorem 3.3.4 (Sound upper bounds). *theorem Let $\mathcal{T}$ be a countable and exhaustive set of interval traces and $\vdash P : \mathbf{R}$ a program. Then $[\![P]\!] \leq \mathsf{upperBd}_P^{\mathcal{T}}$.*

Proof. The idea is that each interval trace $\boldsymbol{t}$ summarizes all infinite traces starting with $(\!|\boldsymbol{t}|\!)$, i.e. all traces in $\mathit{cover}(\boldsymbol{t})$. Exhaustivity ensures that almost all infinite traces are "covered". Concretely, for any $U \in \Sigma_{\mathbb{R}}$, we have

$$\begin{aligned}
\mathsf{upperBd}_P^{\mathcal{T}}(U) &= \sum_{t \in \mathcal{T}} \mathrm{vol}(\boldsymbol{t})(\sup \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \cap U \neq \varnothing] \\
&= \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} (\sup \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}))[\mathrm{val}_P^{\mathbb{I}}(\boldsymbol{t}) \cap U \neq \varnothing] \mathrm{d}\boldsymbol{s} \\
&\geq \sum_{t \in \mathcal{T}} \int_{(\!|\boldsymbol{t}|\!)} \int_{\mathbb{T}_\infty} \mathrm{wt}_P^{\infty}(\boldsymbol{su})[\mathrm{val}_P^{\infty}(\boldsymbol{su}) \in U] \mathrm{d}\boldsymbol{u}\, \mathrm{d}\boldsymbol{s} \qquad (3.4) \\
&\geq \int_{\bigcup_{t \in \mathcal{T}} (\!|\boldsymbol{t}|\!) \times \mathbb{T}_\infty} \mathrm{wt}_P^{\infty}(\boldsymbol{u})[\mathrm{val}_P^{\infty}(\boldsymbol{u}) \in U] \mathrm{d}\boldsymbol{s} \\
&\geq \int_{\mathbb{T}_\infty} \mathrm{wt}_P^{\infty}(\boldsymbol{u})[\mathrm{val}_P^{\infty}(\boldsymbol{u}) \in U] \mathrm{d}\boldsymbol{u} \qquad (3.5) \\
&= [\![P]\!](U) \qquad (3.6)
\end{aligned}$$

where Eq. (3.4) follows from Lemma 3.3.2, Eq. (3.5) from exhaustivity and Eq. (3.6) from Lemma 3.3.1. □

3.3.3 Symbolic Execution

In this section we formally introduce stochastic symbolic execution [Mak et al. 2021a; Geldenhuys et al. 2012; Chaganty et al. 2013]. We make use of this form of symbolic execution in two separate ways. First, our completeness proof hinges on guarantees provided by the symbolic execution in order to identify a suitable set of interval traces. Second, our tool GuBPI relies on the symbolic execution as a first step in the program analysis, in order to identify relevant paths and independent subexpressions, and to avoid repeated evaluation in a small-step semantics.

$$\frac{}{((\lambda x.\mathcal{M})\mathcal{V}, n, \Delta, \Xi) \rightsquigarrow (\mathcal{M}[\mathcal{V}/x], n, \Delta, \Xi)}$$

$$\frac{}{((\mathsf{fix}^{\varphi}_{x}.\,\mathcal{M})\mathcal{V}, n, \Delta, \Xi) \rightsquigarrow (\mathcal{M}[\mathcal{V}/x, (\mathsf{fix}^{\varphi}_{x}.\,\mathcal{M})/\varphi], n, \Delta, \Xi)}$$

$$\frac{}{(\mathsf{if}\,\mathcal{V} \le 0\,\{\mathcal{N}\}\,\mathsf{else}\,\{\mathcal{P}\}, n, \Delta, \Xi) \rightsquigarrow (\mathcal{N}, n, \Delta \cup \{\mathcal{V} \le 0\}, \Xi)}$$

$$\frac{}{(\mathsf{if}\,\mathcal{V} \le 0\,\{\mathcal{N}\}\,\mathsf{else}\,\{\mathcal{P}\}, n, \Delta, \Xi) \rightsquigarrow (\mathcal{P}, n, \Delta \cup \{\mathcal{V} > 0\}, \Xi)}$$

$$\frac{}{(\mathsf{sample}, n, \Delta, \Xi) \rightsquigarrow (\alpha_{n+1}, n+1, \Delta, \Xi)}$$

$$\frac{}{(\mathsf{score}(\mathcal{V}), n, \Delta, \Xi) \rightsquigarrow (\mathcal{V}, n, \Delta \cup \{\mathcal{V} \ge 0\}, \Xi \cup \{\mathcal{V}\})}$$

$$\frac{(\mathcal{R}, n, \Delta, \Xi) \rightsquigarrow (\mathcal{M}, n', \Delta', \Xi')}{(\mathcal{E}[\mathcal{R}], n, \Delta, \Xi) \rightsquigarrow (\mathcal{E}[\mathcal{M}], n', \Delta', \Xi')}$$

Figure 3.4: Reduction rules for symbolic execution

High-level idea The overarching idea of symbolic execution is to postpone the evaluation of sample expressions and instead use a *sample variable* to symbolically represent its outcome. The result of symbolic execution is thus a symbolic value: a term consisting of sample variables and delayed primitive function applications. As a consequence, branching and scoring steps cannot be executed concretely, so we record them symbolically instead.

Symbolic terms To postpone concrete sample decisions we introduce ***sample variables*** $\alpha_1, \alpha_2, \ldots$ into our language. We then define *symbolic terms* and *symbolic values* by extending interval terms and values by adding two new constructs: every sample variable α_j is a symbolic value and for every primitive function f and symbolic values $\mathcal{V}_1, \ldots, \mathcal{V}_{|f|}$, the symbolic term $\underline{f}(\mathcal{V}_1, \ldots, \mathcal{V}_{|f|})$ is a symbolic value, denoting a function application that is postponed until all sample variables are instantiated. We denote ***symbolic terms*** by $\mathcal{M}, \mathcal{N}, \mathcal{P}$ and ***symbolic values*** by $\mathcal{V}, \mathcal{W}$. Formally we define

$$\begin{aligned}\mathcal{V} &:= x \mid \underline{r} \mid \lambda x.\mathcal{M} \mid \mathsf{fix}^{\varphi}_{x}.\,\mathcal{M} \mid \alpha_i \mid \underline{f}(\mathcal{V}_1, \ldots, \mathcal{V}_{|f|})\\ \mathcal{M}, \mathcal{N}, \mathcal{P} &:= \mathcal{V} \mid \mathcal{M}\mathcal{N} \mid \mathsf{if}\,\mathcal{M} \le 0\,\{\mathcal{N}\}\,\mathsf{else}\,\{\mathcal{P}\} \mid \underline{f}(\mathcal{M}_1, \ldots, \mathcal{M}_{|f|})\\ &\quad \mid \mathsf{sample} \mid \mathsf{score}(\mathcal{M})\end{aligned}$$

The definition of redex and evaluation context extends naturally (recall that we regard α_j as a value).

Symbolic execution A ***symbolic constraint*** is a pair $(\mathcal{V} \bowtie r)$ where $\mathcal{V}$ is a symbolic value, $\bowtie \in \{\le, <, \ge, >\}$ and $r \in \mathbb{R}$. A ***symbolic configuration*** has the form $\psi = (\mathcal{M}, n,$

Δ, Ξ) where $\mathcal{M}$ is a symbolic term, $n \in \mathbb{N}$ a natural number (used to obtain fresh sample variables), Δ a set of symbolic constraints (which track the symbolic conditions on the current execution path), and Ξ is a set of symbolic values (which records all symbolic values scored on the current path). When executing symbolically: (1) we evaluate each sample to a fresh sample variable, (2) we postpone function application, (3) for each conditional, we explore both branches (our reduction is nondeterministic) and record the symbolic inequalities that must hold along the current path, and (4) we record the symbolic values that we scored with. We give the reduction rules in Fig. 3.4.

We call a tuple $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ (a symbolic configuration where the symbolic term is a value) a ***symbolic path***. For a symbolic configuration ψ, we write $symPaths(\psi)$ for the set of symbolic paths reached when evaluating from ψ. Note that $symPaths(\psi)$ is countable.

Example 3.3.5. Consider the symbolic execution of Example 2.3.3 where the first step moves the pedestrian towards their home (taking the right branch of $\{\ldots\}\ [0.5]\ \{\ldots\}$) and the second step moves away from their home (the left branch). We reach a configuration $(\mathcal{M}, 5, \Delta, \Xi)$ where

$$\begin{aligned}
\mathcal{M} &:= \mathsf{score}\big(\mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}\ (\alpha_2 + \alpha_4 + \mathcal{N}(3\alpha_1 - \alpha_2 + \alpha_4))\big); 3\alpha_1 \\
\mathcal{N} &:= \mathsf{fix}^{\varphi}_{x}.\ \mathsf{if}\ x \le 0\ \{0\}\ \mathsf{else}\ \{\big(\lambda \mathsf{step.step} + \varphi(\{x + \mathsf{step}\}\ [0.5]\ \{x - \mathsf{step}\})\big)\mathsf{sample}\} \\
\Delta &:= \{3\alpha_1 > 0, \alpha_3 > \tfrac{1}{2}, 3\alpha_1 - \alpha_2 > 0, \alpha_5 \le \tfrac{1}{2}\} \\
\Xi &:= \varnothing
\end{aligned}$$

Here α_1 is the initial sample for *start*; α_2, α_4 the two samples of *step*; and α_3, α_5 the samples to decide which probabilistic branch to take in $\{\ldots\}\ [0.5]\ \{\ldots\}$.

Let $\mathcal{V}$ be a symbolic value of type $\mathbf{R}$ (no λ-abstraction or fixed point) with sample variables within $\{\alpha_1, \ldots, \alpha_n\}$. For a trace $\boldsymbol{s} = \langle r_1, \ldots, r_n \rangle \in [0,1]^n$, we define $\mathcal{V}[\boldsymbol{s}/\overline{\alpha}]$ as the value (in $\mathbb{R}$) obtained by replacing the sample variables in $\overline{\alpha}$ with $\boldsymbol{s}$ and evaluate the postponed primitive function applications. Given a set of symbolic constraints Δ with sample variables contained in $\{\alpha_1, \ldots, \alpha_n\}$ we define

$$\mathrm{Sat}_n(\Delta) := \bigcap_{(\mathcal{V} \bowtie r) \in \Delta} \{\boldsymbol{s} \in [0,1]^n \mid \mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \bowtie r\}$$

as the set of actual values for the sample variables that satisfy all constraints. For a symbolic path $\Psi = (\mathcal{V}, n, \Delta, \Xi)$, we define its denotation along Ψ as

$$[\![\Psi]\!](U) = \int_{\mathrm{Sat}_n(\Delta)} [\mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \in U] \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]\, \mathrm{d}\boldsymbol{s}$$

i.e. the integral of the product of the score weights Ξ over the traces of length n where the result value is in U and all the constraints Δ are satisfied.

Correctness of symbolic execution We can now establish a correspondence between symbolic execution and the ordinary reduction. If we wish to symbolically analyze a term P, we consider the (symbolic) reductions starting from $(P, 0, \varnothing, \varnothing)$, resulting in the symbolic paths $symPaths(P, 0, \varnothing, \varnothing)$.

Lemma 3.3.6. *Let* $\vdash P : \mathbf{R}$ *and suppose we have* $(\mathcal{V}, n, \Delta, \Xi) \in \mathit{symPaths}(P, 0, \varnothing, \varnothing)$*, where* P *is interpreted as a symbolic term. Then for any* $\boldsymbol{s} \in \mathit{Sat}_n(\Delta)$*, we have*

$$(P, \boldsymbol{s}, 1) \to^* (\mathcal{V}[\boldsymbol{s}/\overline{\alpha}], \langle\rangle, \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}])$$

Proof. A similar proof can be found in [Mak et al. 2021a, Theorem 1]. □

Lemma 3.3.7. *Let* $\vdash P : \mathbf{R}$ *and suppose* $(P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w)$ *for some* $r \in \mathbb{R}$*. Then there exists a* unique $(\mathcal{V}, n, \Delta, \Xi) \in \mathit{symPaths}(P, 0, \varnothing, \varnothing)$ *such that* $\boldsymbol{s} \in \mathit{Sat}_n(\Delta)$*. For this unique symbolic path we have* $w = \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$ *and* $r = \mathcal{V}[\boldsymbol{s}/\overline{\alpha}]$*.*

Proof sketch. Choose the same branches in the $\rightsquigarrow$-reduction of P as in its $\to$-reduction. Then it is straightforward to see that this correspondence holds at every symbolic reduction step: if $(P, 0, \varnothing, \varnothing) \rightsquigarrow^* (\mathcal{P}', n, \Delta, \Xi)$ then the corresponding $\to$-reduction steps yield $(P, \boldsymbol{s}\boldsymbol{s}', 1) \to^* (P', \boldsymbol{s}', w)$ where $\boldsymbol{s}$ has length n, P' is $\mathcal{P}'[\boldsymbol{s}/\overline{\alpha}]$ (after evaluating delayed primitive function applications), Δ records the guards $C[\boldsymbol{s}/\overline{\alpha}] \le 0$ or $C[\boldsymbol{s}/\overline{\alpha}] > 0$ that need to hold for the trace $\boldsymbol{s}$, and finally, the weight w is given by $\prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$ at any point. □

We can recover the denotation of a program P (as defined in Section 2.3.2) from all its symbolic paths starting from the configuration $(P, 0, \varnothing, \varnothing)$.

Theorem 3.3.8. *Let* $\vdash P : \mathbf{R}$ *be a program and* $U \in \Sigma_{\mathbb{R}}$*. Then*

$$[\![P]\!](U) = \textstyle\sum_{\Psi \in \mathit{symPaths}(P,0,\varnothing,\varnothing)} [\![\Psi]\!](U)$$

Proof.

$$\begin{aligned}
[\![P]\!](U) &= \sum_{n \in \mathbb{N}} \int_{[0,1]^n} [\mathrm{val}_P(\boldsymbol{s}) \in U] \mathrm{wt}_P(\boldsymbol{s})\, \mathrm{d}\boldsymbol{s} \\
&= \sum_{n \in \mathbb{N}} \int_{[0,1]^n} \sum_{(\mathcal{V},n,\Delta,\Xi)} [\boldsymbol{s} \in \mathrm{Sat}_n(\Delta)][\mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \in U] \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]\, \mathrm{d}\boldsymbol{s} \\
&= \sum_{(\mathcal{V},n,\Delta,\Xi)} \int_{\mathrm{Sat}_n(\Delta)} [\mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \in U] \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]\, \mathrm{d}\boldsymbol{s} \\
&= \sum_{(\mathcal{V},n,\Delta,\Xi)} [\![(\mathcal{V}, n, \Delta, \Xi)]\!](U)
\end{aligned}$$

where the sum ranges over symbolic paths $(\mathcal{V}, n, \Delta, \Xi) \in \mathit{symPaths}(P, 0, \varnothing, \varnothing)$. The first equality is by definition, the second one by Lemmas 3.3.6 and 3.3.7, the third by noting that $\mathrm{Sat}_n(\Delta) \subseteq [0,1]^n$ and exchanging the infinite sum and integral (which is allowed because everything is nonnegative) and the fourth by definition. □

3.3.4 Completeness

The soundness results for upper and lower bounds allow us to derive bounds on the denotation of a program. One would expect that a finer partition of interval traces will yield more precise bounds. In this section, we show that for a program P and an interval $I \in \mathbb{I}$, the approximations $\mathrm{lowerBd}_P^{\mathcal{T}}(I)$ and $\mathrm{upperBd}_P^{\mathcal{T}}(I)$ can in fact come arbitrarily close to $[\![P]\!](I)$ for suitable $\mathcal{T}$. However, this is only possible under certain assumptions.

Use of sampled values Interval arithmetic is imprecise if the same value is used more than once: consider, for instance, let s = sample; if $s - s \leq 0\,\{0\}$ else $\{1\}$ which deterministically evaluates to 0. However, in interval arithmetic, if x is approximated by an interval $[a, b]$ with $a < b$, the difference $x - x$ is approximated as $[a - b, b - a]$, which always contains both positive and negative values. So no non-trivial interval trace can separate the two branches.

To avoid this, we could consider a call-by-name semantics (as done in Beutner and Ong [2021]) where sample values can only be used once by definition. However, many of our examples cannot be expressed in the call-by-name setting, so we instead propose a less restrictive criterion to guarantee completeness for call-by-value: we allow sample values to be used more than once, but at most once in the guard of each conditional, at most once in each score expression, and at most once in the return value. While this prohibits terms like the one above, it allows, e.g. let s = sample; if $s \leq 0\,\{\underline{f}(s)\}$ else $\{\underline{g}(s)\}$. We can formally express this assumption about a given program $\vdash P : \mathbf{R}$ as follows.

Assumption 3.3.9. Let $\vdash P : \mathbf{R}$ be a program. We require for each symbolic path $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ that $\mathcal{V}$, each C with $C \bowtie 0 \in \Delta$, and each $\mathcal{W} \in \Xi$ contain each sample variable α_i at most once.

Example 3.3.10. The pedestrian example (Example 2.3.3) satisfies Assumption 3.3.9 because the symbolic paths have the form $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ with:

$$
\begin{aligned}
\mathcal{V} &= 3\alpha_1 \\
n &= 2k + 1 \\
\Delta &= \{\alpha_3 - \tfrac{1}{2} \bowtie 0, \alpha_5 - \tfrac{1}{2} \bowtie 0, \ldots, \alpha_{2k+1} - \tfrac{1}{2} \bowtie 0\} \\
&\quad \cup \{3\alpha_1 > 0, \\
&\qquad 3\alpha_1 \pm \alpha_2 > 0, \\
&\qquad \ldots, \\
&\qquad 3\alpha_1 \pm \alpha_2 \pm \alpha_4 \cdots \pm \alpha_{2k-2} > 0, \\
&\qquad 3\alpha_1 \pm \alpha_2 \pm \alpha_4 \cdots \pm \alpha_{2k} \leq 0\} \\
\Xi &= \{\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}(\alpha_2 + \alpha_4 + \cdots + \alpha_{2k})\}
\end{aligned}
$$

As we can see, none of the symbolic values contains a sample variable twice, so the assumption is satisfied.

Many examples we encountered in the literature satisfy this assumption. A typical class of counterexamples are Bayesian models where several parameters $a_1, \ldots, a_n$ have prior distributions $D(b)$ with the same parameter b sampled from some prior distribution and where observations involve a combination of these parameters, e.g. $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$ in a linear regression model. The assumption is violated for such models because the score expressions implicitly contain the sample b multiple times.

Primitive functions In addition, we require mild assumptions on the primitive functions, called *boxwise continuity* and *interval separability*.

We need to be able to approximate a program's weight function by step functions in order to obtain tight bounds on its integral. A function $f : \mathbb{R}^n \to \mathbb{R}$ is ***boxwise***

continuous if it can be written as the countable union of continuous functions on boxes, i.e. if there is a countable union of pairwise almost disjoint boxes B_i such that $\bigcup B_i = \mathbb{R}^n$ and the restriction $f|_{B_i}$ is continuous for each B_i.

Furthermore, we need to approximate preimages. Formally, we say that A is a ***tight subset*** of B (written $A \Subset B$) if $A \subseteq B$ and $B \setminus A$ is a null set. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called ***interval separable*** if for every interval $[a, b] \in \mathbb{I}$, there is a countable set $\mathcal{B}$ of boxes in $\mathbb{R}^n$ that tightly approximates the preimage, i.e. $\bigcup \mathcal{B} \Subset f^{-1}([a, b])$. A sufficient condition for checking this is the following.

Lemma 3.3.11. *If a function $f : \mathbb{R}^n \to \mathbb{R}$ is boxwise continuous and preimages of points are null sets then f is interval separable.*

Proof. We decompose $f^{-1}([a, b]) = f^{-1}((a, b)) \cup f^{-1}(\{a, b\})$ and deal with the former set first. By boxwise continuity, $f = \bigcup_i f|_{B_i}$ where $\bigcup_i B_i = \mathbb{R}^n$ and each $f|_{B_i}$ is continuous on B_i. To show that the preimage $f^{-1}((a, b))$ can be tightly approximated by a countable set of boxes, it suffices to show this for each $(f|_{B_i})^{-1}((a, b))$. This set is open in B_i by continuity of $f|_{B_i}$, so it can be written as a countable union of boxes (e.g. by taking a box within B_i around each rational point, which exists because it is an open set). By the assumption, the preimage $f^{-1}(\{a, b\})$ is a null set. Hence $f^{-1}([a, b])$ can be approximated by a null set. □

Note that a composition of interval separable functions need *not* be interval separable. Beutner and Ong [2021], whose (simpler) semantics also relies on interval traces, overlook this important point, rendering their completeness proof incorrect. (To fix their Theorem 3.8, one needs to make the additional assumption that the set of primitive functions be closed under composition.) To see this, let $f, g : \mathbb{R} \to \mathbb{R}$ be interval separable functions and I an interval. By definition, there are intervals B_i such that $\bigcup_i B_i \cup N = f^{-1}(I)$ where N is a null set. Then by interval separability, the preimage $g^{-1}(\bigcup_i B_i)$ can be tightly approximated by intervals B'_j, but the preimage $g^{-1}(N)$ need not be a null set. It is also not clear at all whether one can approximate the preimage $(f \circ g)^{-1}(I)$ tightly using intervals without further restrictions on f and g. For this reason, we require the assumption that the set of primitive functions be closed under composition. We also require constant functions to be available.[3]

Assumption 3.3.12. We assume the set $\mathcal{F} \subset \bigcup_{n=0}^{\infty}(\mathbb{R}^n \to \mathbb{R})$ of primitive functions contains all constant functions $\mathbb{R} \to \mathbb{R}$, is closed under composition, and each $f \in \mathcal{F}$ is boxwise continuous and interval separable.

It is not obvious that interesting sets of primitive functions satisfying this assumption exist. However, the set of *submersions*, with a slight modification to allow constant functions, does meet the criteria. To keep the focus on the completeness property, we relegate this discussion to Appendix A.1.1.

The completeness theorem Using these two assumptions, we can state completeness of our interval semantics.

[3] This last point was not made very clearly in Beutner et al. [2022b]. Together with closure under composition, it ensures that partial applications of primitive functions to constants are still boxwise continuous and interval separable.

Theorem 3.3.13 (Completeness of interval approximations). *Let $I \in \mathbb{I}$ and $\vdash P : \mathbf{R}$ be an almost surely terminating program satisfying Assumptions 3.3.9 and 3.3.12. Then, for all $\varepsilon > 0$, there is a countable set of interval traces $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ that is compatible and exhaustive such that*

$$\mathsf{upperBd}_P^{\mathcal{T}}(I) - \varepsilon \leq [\![P]\!](I) \leq \mathsf{lowerBd}_P^{\mathcal{T}}(I) + \varepsilon$$

Proof sketch. We consider each branching path through the program separately. The set of relevant traces for a given path is a preimage of intervals under compositions of interval separable functions, hence can essentially be partitioned into boxes. By boxwise continuity, we can refine this partition such that the weight function is continuous on each box. To approximate the integral, we pass to a refined partition again, essentially computing Riemann sums. The latter converge to the Riemann integral, which agrees with the Lebesgue integral under our conditions, as desired. The full proof will be given in Section 3.3.5. □

For the lower bound, we can actually derive ε-close bounds using only finitely many interval traces:

Corollary 3.3.14. *Let $I \in \mathbb{I}$ and $\vdash P : \mathbf{R}$ be as in Theorem 3.3.13. There is a sequence of finite, compatible sets of interval traces $\mathcal{T}_1, \mathcal{T}_2, \ldots \subseteq \mathbb{T}_{\mathbb{I}}$ such that $\lim_{n\to\infty} \mathsf{lowerBd}_P^{\mathcal{T}_n}(I) = [\![P]\!](I)$.*

Proof. By Theorem 3.3.13, we can find for each $n \in \mathbb{N}$ a set of interval traces $\mathcal{T}_n'$ such that $\mathsf{lowerBd}_P^{\mathcal{T}_n'}(I) > [\![P]\!](I) - 1/n$. Since the lower bound is defined as a sum over $\mathcal{T}_n'$, there is a finite subset $\mathcal{T}_n$ such that $\mathsf{lowerBd}_P^{\mathcal{T}_n}(I) > [\![P]\!](I) - 2/n$. Since $\mathcal{T}_n'$ is still compatible, the soundness result yields $\mathsf{lowerBd}_P^{\mathcal{T}_n}(I) \leq [\![P]\!](I)$, implying the claim. □

For the upper bound, a restriction to finite sets $\mathcal{T}$ of interval traces is, in general, not possible: if the weight function for a program is unbounded, it is also unbounded on some $\boldsymbol{t} \in \mathcal{T}$. Then $\mathsf{wt}_P^{\mathbb{I}}(\boldsymbol{t})$ is an infinite interval, implying $\mathsf{upperBd}_P^{\mathcal{T}}(I) = \infty$.

Example 3.3.15. Consider the following probabilistic loop.

$$P \equiv \mathsf{letrec}\ \mathit{loop}\ \mathit{thresh} = \mathsf{if}\ \mathsf{sample} \leq \mathit{thresh}\ \{\mathsf{score}(2); \varphi(\mathit{thresh}/2)\}\ \mathsf{else}\ \{1\};\ \mathit{loop}\ 1$$

This program P only requires addition and scalar multiplication, so clearly satisfies Assumptions 3.3.9 and 3.3.12. It has the weight function

$$\mathsf{wt}_P(\langle t_0, \ldots, t_n \rangle) = \begin{cases} 2^n & \text{if } t_n > 2^{-n} \wedge \forall i \in \{0, \ldots, n-1\} : t_i \leq 2^{-i} \\ 0 & \text{otherwise} \end{cases}$$

P is integrable because the normalizing constant is

$$Z = \int_{\mathbb{T}} \mathsf{wt}_P(\boldsymbol{s})\, \mathrm{d}\boldsymbol{s} = \sum_{n=1}^{\infty} 2^n \times (1 - 2^{-n}) \prod_{i=0}^{n-1} 2^{-i} = \sum_{n=1}^{\infty} 2^n (1 - 2^{-n}) 2^{-n(n-1)/2} < \infty$$

We claim that P requires infinitely many interval traces for the upper bound to converge to the true denotation. Define the sets of traces T_n for $n \geq 1$ by

$$T_n = [0, 2^0] \times [0, 2^{-1}] \times \cdots \times [0, 2^{-n+1}] \times (2^{-n}, 1]$$

Suppose we are given an arbitrary finite exhaustive set of interval traces. This set needs to cover all of the T_n's, so one interval trace, say $\boldsymbol{t}$, must cover infinitely many T_n's. Since $\mathrm{wt}_P(\boldsymbol{s}) = 2^n$ for $\boldsymbol{s} \in T_n$, the weight function on $\boldsymbol{t}$ is unbounded. Therefore, the only possible upper bound for $\boldsymbol{t}$ is ∞, *even if our semantics could compute the set* $\{\mathrm{wt}_P(\boldsymbol{s}) \mid \boldsymbol{s} \in (\!|\boldsymbol{t}|\!)\}$ *exactly*. Hence any finite exhaustive interval trace has upper bound ∞, while the true denotation is finite. As we have seen, this is not because of imprecision of interval analysis, but an inherent problem if the weight function is unbounded. So we cannot hope for complete upper bounds with finitely many interval traces.

Despite the (theoretical) need for countably infinite many interval traces, we can, in many cases, compute finite upper bounds by making use of an interval-based static approximation, formalized as a type system in Section 3.4.

3.3.5 Completeness Proof

For the completeness proof, we will need a few auxiliary lemmas. The first one states that a countable set of interval traces can always be turned into an equivalent compatible one.

Lemma 3.3.16 (Making interval traces compatible)**.** *Given a countable set of interval traces $\mathcal{T} \subseteq \mathbb{I}^n$, there is a countable set of interval traces $\mathcal{T}' \subseteq \mathbb{I}^n$ that is compatible and satisfies $\bigcup_{\boldsymbol{t}\in\mathcal{T}} (\!|\boldsymbol{t}|\!) = \bigcup_{\boldsymbol{t}\in\mathcal{T}'} (\!|\boldsymbol{t}|\!)$.*

Proof. Let $A : \mathbb{N} \to \mathcal{T}$ be an enumeration of $\mathcal{T}$. Define the modified enumeration $A' : \mathbb{N} \to \Sigma_{\mathbb{R}^n}$ by $m \mapsto \overline{A(m) \setminus \bigcup_{i=0}^{m-1} A(i)}$ where $\overline{S}$ denotes the closure of S. Then the collection $\{A'(m) \mid m \in \mathbb{N}\}$ is pairwise almost disjoint, and each $A'(m)$ can be written as a finite union of boxes, proving the claim. □

The following lemma states that for a fixed symbolic path, the set of traces resulting in a certain value in a given interval can be tightly approximated by a countable set of interval traces.

Lemma 3.3.17 (Approximating $\mathrm{val}_P^{-1}([x,y])$ for a fixed symbolic path)**.** *Let $\mathcal{V}$ a symbolic value of ground type containing each sample variable $\alpha_1, \ldots, \alpha_n$ at most once and $[x,y]$ an interval. Then there is a countable set of pairwise disjoint interval traces $\mathcal{T} \subset \mathbb{I}^n_{[0,1]}$ such that*

$$\bigcup_{\boldsymbol{t}\in\mathcal{T}} (\!|\boldsymbol{t}|\!) \Subset \{\boldsymbol{s} \in [0,1]^n \mid \mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \in [x,y]\}$$

Proof. If $\mathcal{V}$ is of ground type, then it is simply a composition of primitive functions applied to sample variables and literals. Since the set of primitive functions is closed under composition and since no sample variable occurs twice, this composition is still an interval separable function f of the sample variables. By definition of interval separability, there is a countable set of interval traces $\mathcal{J}$ such that $\bigcup_{\boldsymbol{t}\in\mathcal{J}} (\!|\boldsymbol{t}|\!) \Subset f^{-1}([x,y])$, as desired. □

The next lemma states that for a fixed symbolic path, the integral of the weight over an interval trace can be approximated arbitrarily closely by a sum over a countable subdivision. These sums are essentially Riemann sums used to define the Riemann integral, which coincides with the Lebesgue integral in this setting.

Lemma 3.3.18 (Approximating the weight integral by Riemann sums)**.** *Let $\boldsymbol{t} \in \mathbb{I}^n$ be an interval trace and Ξ a set of symbolic values with sample variables from $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$. Then for any $\varepsilon > 0$, there is a countable subdivision $\mathcal{T}$ of $\boldsymbol{t}$ such that*

$$\sum_{\boldsymbol{t}' \in \mathcal{T}} \operatorname{vol}(\boldsymbol{t}') \sup_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}] - \varepsilon \leq \int_{(\!|\boldsymbol{t}|\!)} \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}] \, \mathrm{d}\boldsymbol{s} \leq \sum_{\boldsymbol{t}' \in \mathcal{T}} \operatorname{vol}(\boldsymbol{t}') \min_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}] + \varepsilon$$

Proof. Values are simply boxwise continuous functions applied to the sample variables. Intersecting the boxes for each $\mathcal{W} \in \Xi$, we see that the function

$$f : (\!|\boldsymbol{t}|\!) \to \mathbb{R}, \quad \boldsymbol{s} \mapsto \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$$

is boxwise continuous. We can thus find a countable subdivision $\mathcal{T}_{\mathsf{cont}}$ of $\boldsymbol{t}$ such that f is continuous on each $\boldsymbol{t}' \in \mathcal{T}_{\mathsf{cont}}$.

Since we can sum over the $\boldsymbol{t}' \in \mathcal{T}_{\mathsf{cont}}$, it suffices to prove that each integral $\int_{(\!|\boldsymbol{t}'|\!)} f(\boldsymbol{s}) \, \mathrm{d}\boldsymbol{s}$ can be approximated arbitrarily closely. Note that each such integral is finite because a continuous function is bounded on a compact set and the measure of $(\!|\boldsymbol{t}'|\!)$ is finite. But then such approximations are given by Riemann sums, i.e. the sums that are used to define the Riemann integral. As a concrete example, one can consider the subdivision $\mathcal{T}_m$ of $\boldsymbol{t}'$ in m equidistant sections in each dimension (consisting of m^n parts overall). Then $\sum_{\boldsymbol{t}'' \in \mathcal{T}_m} \operatorname{vol}(\boldsymbol{t}'') \min_{\boldsymbol{s} \in (\!|\boldsymbol{t}''|\!)} f(\boldsymbol{s})$ converges to the Riemann integral $\int_{(\!|\boldsymbol{t}'|\!)} f(\boldsymbol{s}) \, \mathrm{d}\boldsymbol{s}$ as $m \to \infty$ (and similarly for the supremum). Since it is known that the Riemann integral and the Lebesgue integral have the same value for continuous functions on a Cartesian product of compact intervals, the claim follows immediately. □

The following lemma describes the relationship between symbolic execution and interval semantics.

Lemma 3.3.19 (Relating symbolic and interval semantics)**.** *Let $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ be a symbolic path of P and $\boldsymbol{t}$ an interval trace with $(\!|\boldsymbol{t}|\!) \subseteq \operatorname{Sat}_n(\Delta)$. Suppose furthermore that all the symbolic values contain each of the sample variables $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ at most once. Then there is a subdivision $\mathcal{T}$ of $\boldsymbol{t}$ such that for all $\boldsymbol{t}' \in \mathcal{T}$, the interval semantics for the value is precise:*

$$\operatorname{val}_P^{\mathbb{I}}(\boldsymbol{t}') = \{\operatorname{val}_P(\boldsymbol{s}) \mid \boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)\}$$

For each symbolic score value $\mathcal{W} \in \Xi$, let $[\mathcal{W}_{\boldsymbol{t}'}^-, \mathcal{W}_{\boldsymbol{t}'}^+]$ be its interval approximation, i.e. $\mathcal{W}_{\boldsymbol{t}'}^- = \min_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$ and $\mathcal{W}_{\boldsymbol{t}'}^+ = \sup_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$. Then

$$\operatorname{wt}_P^{\mathbb{I}}(\boldsymbol{t}') = \left[\prod_{\mathcal{W} \in \Xi} \mathcal{W}_{\boldsymbol{t}'}^-, \prod_{\mathcal{W} \in \Xi} \mathcal{W}_{\boldsymbol{t}'}^+ \right]$$

Proof. The symbolic value $\mathcal{V}$ is a composition of primitive functions applied to α's. Hence the are boxwise continuous functions of the α's. We pick a suitable subdivision

$\mathcal{T}$ such that all these functions are continuous when restricted to any $\boldsymbol{t}' \in \mathcal{T}$. For any such function f, we have

$$f_{\mathbb{I}}([x_1, y_1], \ldots, [x_m, y_m]) = [\inf F, \sup F]$$

where $F := f([x_1, y_1] \times \cdots \times [x_m, y_m]) \subseteq \mathbb{R}$, by definition. Then continuity implies that the image of any box is a compact and path-connected subset of $\mathbb{R}$, i.e. an interval. Hence we even have $f_{\mathbb{I}}([x_1, y_1], \ldots, [x_m, y_m]) = F$, i.e. the image of any box equals its interval approximation, proving the claim about the value semantics.

For the interval semantics of the weight, note that the previous argument applies to every symbolic score value $\mathcal{W} \in \Xi$, proving that

$$\{\mathcal{W}[\boldsymbol{s}/\overline{\alpha}] \mid \boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)\} = [\mathcal{W}^-_{\boldsymbol{t}'}, \mathcal{W}^+_{\boldsymbol{t}'}]$$

Since the interval semantics multiplies the interval approximation of each score value in interval arithmetic, this implies the claim. □

Note that a seemingly natural strengthening of the previous lemma where we replace the product of the extrema of the weights with the extrema of the product of the weights does not hold. In fact, the interval approximation of the weight is imprecise in the following sense:

$$\mathrm{wt}^{\mathbb{I}}_P(\boldsymbol{t}') \neq \left\{ \prod_{\mathcal{W} \in \Xi} \mathcal{W}[\boldsymbol{s}/\overline{\alpha}] \;\middle|\; \boldsymbol{s} \in (\!|\boldsymbol{t}'|\!) \right\}$$

As an example, if $\Xi = \{\alpha_1, 1 - \alpha_1\}$ and $\boldsymbol{t}'' = \langle [0,1] \rangle$ then the left-hand side is $[0,1]$ because each of the weights is approximated by $[0,1]$, but the right-hand side is $[0, \frac{1}{4}]$ because the function $\alpha_1(1 - \alpha_1)$ attains its maximum at $\frac{1}{4}$, not 1.

The final lemma addresses this discrepancy. It shows that the extrema of the weight function can be approximated arbitrarily closely by the interval weight function after passing to yet another subdivision.

Lemma 3.3.20 (Approximating extrema of wt_P by $\mathrm{wt}^{\mathbb{I}}_P$)**.** *Let $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ be a symbolic path of P and $\boldsymbol{t}$ an interval trace with $(\!|\boldsymbol{t}|\!) \subseteq \mathrm{Sat}_n(\Delta)$. Suppose furthermore that all the symbolic values contain each of the sample variables $\overline{\alpha} = \alpha_1, \ldots, \alpha_n$ at most once. Then for all $\varepsilon > 0$, there is a subdivision $\mathcal{T}$ of $\boldsymbol{t}$ such that for all $\boldsymbol{t}' \in \mathcal{T}$, we have $\min_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \mathrm{wt}_P(\boldsymbol{s}) \leq \min \mathrm{wt}^{\mathbb{I}}_P(\boldsymbol{t}') + \varepsilon$ and $\sup_{\boldsymbol{s} \in (\!|\boldsymbol{t}'|\!)} \mathrm{wt}_P(\boldsymbol{s}) \geq \sup \mathrm{wt}^{\mathbb{I}}_P(\boldsymbol{t}') - \varepsilon$.*

Proof. Since for each $\mathcal{W} \in \Xi$, the function $f : (\!|\boldsymbol{t}|\!) \to \mathbb{R}, \boldsymbol{s} \mapsto \mathcal{W}[\boldsymbol{s}/\overline{\alpha}]$ is boxwise continuous (a property of primitive functions), we can find a countable subdivision $\mathcal{T}'$ of $\boldsymbol{t}$, such that for all $\boldsymbol{t}' \in \mathcal{T}'$, f is continuous on $(\!|\boldsymbol{t}'|\!)$. Hence it suffices to prove the statement for each $\boldsymbol{t}'$.

Since $(\!|\boldsymbol{t}'|\!)$ is compact (because it is closed and bounded), f attains a maximum $W < \infty$ on $(\!|\boldsymbol{t}'|\!)$ and is even uniformly continuous on $(\!|\boldsymbol{t}'|\!)$. Hence there is a $\delta > 0$ such that whenever $||\boldsymbol{s} - \boldsymbol{s}'|| < \delta$ then $|f(\boldsymbol{s}) - f(\boldsymbol{s}')| < \varepsilon' := \frac{\varepsilon}{W^{|\Xi|-1}}$.

Let $\mathcal{T}$ be a subdivision where each interval trace $\boldsymbol{t} \in \mathcal{T}$ has diameter less than δ. For $\boldsymbol{t} \in \mathcal{T}$ and $\mathcal{W} \in \Xi$, let $\mathcal{W}^-_{\boldsymbol{t}} := \min_{\boldsymbol{s} \in (\!|\boldsymbol{t}|\!)} \mathcal{W}[\boldsymbol{s}/\alpha]$ and $\mathcal{W}^+_{\boldsymbol{t}} := \sup_{\boldsymbol{s} \in (\!|\boldsymbol{t}|\!)} \mathcal{W}[\boldsymbol{s}/\alpha]$. By the choice of $\mathcal{T}$, we have $\mathcal{W}^+_{\boldsymbol{t}} \leq \mathcal{W}^-_{\boldsymbol{t}} + \varepsilon'$ for $\boldsymbol{t} \in \mathcal{T}$ where ε' is chosen small enough, such

that $(W+\varepsilon')^{|\Xi|} - W^{|\Xi|} \leq \varepsilon$. By Lemma 3.3.19, we find that $\sup \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}) = \prod_{\mathcal{W}\in\Xi} \mathcal{W}_{\boldsymbol{t}}^+$ and $\min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}) = \prod_{\mathcal{W}\in\Xi} \mathcal{W}_{\boldsymbol{t}}^-$. As a consequence, we have

$$\begin{aligned}\sup \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}) - \min \mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t}) &= \prod_{\mathcal{W}\in\Xi} \mathcal{W}_{\boldsymbol{t}}^+ - \prod_{\mathcal{W}\in\Xi} \mathcal{W}_{\boldsymbol{t}}^- \\ &< \prod_{\mathcal{W}\in\Xi} (\mathcal{W}_{\boldsymbol{t}}^- + \varepsilon') - \prod_{\mathcal{W}\in\Xi} \mathcal{W}_{\boldsymbol{t}}^- \\ &\overset{(*)}{\leq} (W+\varepsilon')^{|\Xi|} - W^{|\Xi|} \leq \varepsilon\end{aligned}$$

where $(*)$ follows from $\mathcal{W}_{\boldsymbol{t}}^- \leq W$ and by multiplying out the product and canceling the respective subtracted term. So the interval $\mathrm{wt}_P^{\mathbb{I}}(\boldsymbol{t})$ has diameter less than ε. Since the interval $\{\mathrm{wt}_P(\boldsymbol{s}) \mid \boldsymbol{s} \in (\!|\boldsymbol{t}|\!)\}$ is contained in it (by soundness), the claim follows. □

With all these lemmas in place, we can finally prove the completeness theorem, which is stated again below for convenience.

Theorem 3.3.13 (Completeness of interval approximations). *Let $I \in \mathbb{I}$ and $\vdash P : \mathbf{R}$ be an almost surely terminating program satisfying Assumptions 3.3.9 and 3.3.12. Then, for all $\varepsilon > 0$, there is a countable set of interval traces $\mathcal{T} \subseteq \mathbb{T}_{\mathbb{I}}$ that is compatible and exhaustive such that*

$$\mathsf{upperBd}_P^{\mathcal{T}}(I) - \varepsilon \leq [\![P]\!](I) \leq \mathsf{lowerBd}_P^{\mathcal{T}}(I) + \varepsilon$$

Proof. First, we give a brief outline of how the proof works. The idea is to cover $\{\boldsymbol{s} \in \mathbb{T} \mid \mathrm{val}_P(\boldsymbol{s}) \in I\}$ using boxes (interval traces). We can achieve this using symbolic execution: for a fixed path through the program, the result value is just a composition of primitive functions applied to the samples. Similarly, the weight function is a product of such functions, hence boxwise continuous. By passing to smaller boxes, we can assume that it is continuous on each box. In order to approximate the integral of the weight function, we use Riemann sums (as used in the definition of the Riemann integral). We partition the domain into smaller and smaller boxes such that the lower bound and the upper bound of the weight function come arbitrarily close (by continuity). Then by properties of the Riemann integral, the bounds arising from the interval traces representing the boxes in this partition converge to the desired integral of the weight function. The details of the proof are as follows.

Step 1: approximating the branching inequalities. Let $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ a symbolic path of P. To find a countable set $\mathcal{T}_\Psi \subseteq \mathbb{T}_{\mathbb{I}}$ such that $\bigcup_{\boldsymbol{t}\in\mathcal{T}_\Psi} (\!|\boldsymbol{t}|\!) \Subset \mathrm{Sat}_n(\Delta)$. Note that $\mathrm{Sat}_n(\Delta)$ is a finite intersection of sets of the form $\{\boldsymbol{s} \in [0,1]^n \mid C[\boldsymbol{s}/\overline{\alpha}] \bowtie 0\}$ where $\bowtie \in \{\leq, >\}$. In the $\leq$ case, we can write this constraint as $C[\boldsymbol{s}/\overline{\alpha}] \in \bigcup_{n\in\mathbb{N}} [-n, 0]$ and in the $>$ case as $C[\boldsymbol{s}/\overline{\alpha}] \in \bigcup_{n\in\mathbb{N}} [1/n, n]$. By applying Lemma 3.3.17 to each of the compact intervals in these unions, we obtain a countable union of boxes that is a tight subset of $\{\boldsymbol{s} \in [0,1]^n \mid C[\boldsymbol{s}/\overline{\alpha}] \bowtie 0\}$. Since the intersection of two boxes is a box and since $\mathrm{Sat}_n(\Delta)$ is a finite intersection of such countable unions of boxes, it can be rewritten as a countable union of boxes. This yields $\mathcal{T}_\Psi$, such that $\bigcup_{\boldsymbol{t}\in\mathcal{T}_\Psi} (\!|\boldsymbol{t}|\!) \Subset \mathrm{Sat}_n(\Delta)$.

Step 2: handling the result value. By applying Lemma 3.3.17 and intersecting the obtained interval traces with $\mathcal{T}_\Psi$, we obtain a countable set $\mathcal{T}'_{\Psi,I} \subseteq \mathbb{T}_{\mathbb{I}}$ such that

$\bigcup_{t\in\mathcal{T}'_{\Psi,I}}(\!|t|\!) \Subset \{s \in \mathrm{Sat}(\Delta) \mid \mathcal{V}[s/\overline{\alpha}] \in I\}$. By the same lemma, we find a countable set $\mathcal{T}'_{\Psi,I^c} \subseteq \mathbb{T}_{\mathbb{I}}$ such that $\bigcup_{t\in\mathcal{T}'_{\Psi,I^c}}(\!|t|\!) \Subset \{s \in \mathrm{Sat}(\Delta) \mid \mathcal{V}[s/\overline{\alpha}] \notin I\}$ because the complement of I can be written as a countable union of intervals. By Lemma 3.3.19, we find subdivisions $\mathcal{T}_{\Psi,I}$ and $\mathcal{T}_{\Psi,I^c}$ that even satisfy $\bigcup_{t\in\mathcal{T}_{\Psi,I}}(\!|t|\!) \Subset \mathrm{val}_P^{-1}(I) \cap \mathrm{Sat}(\Delta)$ and $\bigcup_{t\in\mathcal{T}_{\Psi,I^c}}(\!|t|\!) \Subset \mathrm{val}_P^{-1}(\mathbb{R}\setminus I) \cap \mathrm{Sat}(\Delta)$. By Lemma 3.3.16, we can assume that the interval traces $\mathcal{T}_{\Psi,I}$ are compatible. Because of the almost sure termination assumption, the set of traces $\bigcup_{(\mathcal{V},n,\Delta,\Xi)} \mathrm{Sat}(\Delta) = \mathrm{val}_P^{-1}(\mathbb{R})$ where the union ranges over all symbolic paths of P has measure 1. As a consequence, $\bigcup_{\Psi\in symPaths(P,0,\varnothing,\varnothing)}(\mathcal{T}_{\Psi,I} \cup \mathcal{T}_{\Psi,I^c})$ is a compatible and exhaustive set of interval traces. Now

$$\begin{aligned}
[\![P]\!](I) &= \sum_{(\mathcal{V},n,\Delta,\Xi)} [\![(\mathcal{V},n,\Delta,\Xi)]\!](I) \\
&= \sum_{(\mathcal{V},n,\Delta,\Xi)} \int_{\mathrm{Sat}_n(\Delta)} [\mathcal{V}[s/\overline{\alpha}] \in I] \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}]\,\mathrm{d}s \\
&= \sum_{(\mathcal{V},n,\Delta,\Xi)} \sum_{t\in\mathcal{T}_{(\mathcal{V},n,\Delta,\Xi),I}} \int_{(\!|t|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}]\,\mathrm{d}s
\end{aligned}$$

where the outer sum ranges over the symbolic paths $(\mathcal{V},n,\Delta,\Xi) \in symPaths(P,0,\varnothing,\varnothing)$. The first equality holds by Theorem 3.3.8, the second one by definition and the last one by the construction of $\mathcal{T}_{(\mathcal{V},n,\Delta,\Xi),I}$.

Step 3: approximating the weight function. Let

$$\mathcal{T}' := \bigcup_{\Psi\in symPaths(P,0,\varnothing,\varnothing)} \mathcal{T}_{\Psi,I}$$

For each $t \in \mathcal{T}'$, fix some $\varepsilon_t > 0$, such that $\sum_{t\in\mathcal{T}'} \varepsilon_t = \varepsilon$. This can be achieved, for example, by enumerating $\mathcal{T}'$ as $t^{(1)}, t^{(2)}, \ldots$ and choosing $\varepsilon_{t^{(i)}} = 2^{-i}\varepsilon$. By Lemma 3.3.18, we can find for each $t \in \mathcal{T}'$ a countable set $\mathcal{S}_t$ of interval traces such that:

$$\begin{aligned}
\sum_{t'\in\mathcal{S}_t} \mathrm{vol}(t') \sup_{s\in(\!|t'|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}] - \varepsilon_t/2 &\le \int_{(\!|t|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}]\,\mathrm{d}s \\
&\le \sum_{t'\in\mathcal{S}_t} \mathrm{vol}(t') \min_{s\in(\!|t'|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}] + \varepsilon_t/2
\end{aligned}$$

Next, choose $\varepsilon_{t'} > 0$ for each t' in such a way that $\sum_{t'\in\mathcal{S}_t} \varepsilon_{t'} < \varepsilon_t/2$. By Lemma 3.3.20, we can find for each t' a finite subdivision $\mathcal{S}'_{t'}$ such that for all $t'' \in \mathcal{S}'_{t'}$, we have

$$\begin{aligned}
\sup_{s\in(\!|t''|\!)} \mathrm{wt}_P(s) = \sup_{s\in(\!|t''|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}] &\ge \sup \mathrm{wt}_P^{\mathbb{I}}(t'') - \varepsilon_{t'} \\
\min_{s\in(\!|t''|\!)} \mathrm{wt}_P(s) = \min_{s\in(\!|t''|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}] &\le \min \mathrm{wt}_P^{\mathbb{I}}(t'') + \varepsilon_{t'}
\end{aligned}$$

Multiplying by $\mathrm{vol}(t'')$ and summing over all t'', we find together with the previous inequality

$$\sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'') \sup \mathrm{wt}_P^{\mathbb{I}}(t'') - \varepsilon_t \le \sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'')(\sup \mathrm{wt}_P^{\mathbb{I}}(t'') - \varepsilon_{t'}) - \varepsilon_t/2$$

$$\begin{aligned}
&\leq \int_{(\!|t|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}]\,\mathrm{d}s \\
&\leq \sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'')(\min \mathrm{wt}_P^{\mathbb{I}}(t'') + \varepsilon_{t'}) + \varepsilon_t/2 \\
&\leq \sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'') \min \mathrm{wt}_P^{\mathbb{I}}(t'') + \varepsilon_t
\end{aligned}$$

because $\sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'') \leq 1$ and thus the contribution of all the $\varepsilon_{t'}$ is at most $\varepsilon_t/2$.

Overall, the desired trace set is given by

$$\mathcal{T} := \bigcup_{\Psi=(\mathcal{V},n,\Delta,\Xi)} \left(\mathcal{T}_{\Psi,I^c} \cup \bigcup_{t\in\mathcal{T}_{\Psi,I}} \bigcup_{t'\in\mathcal{S}_t} \mathcal{S}'_{t'} \right)$$

is compatible and exhaustive because it is a subdivision of $\mathcal{T}_{\Psi,I^c}$ and $\mathcal{T}_{\Psi,I}$. By construction, we have $\mathrm{val}_P^{\mathbb{I}}(t) \subseteq I$ for $t \in \mathcal{T}_{\Psi,I}$ and $\mathrm{val}_P^{\mathbb{I}}(t) \cap I = \varnothing$ for $t \in \mathcal{T}_{\Psi,I^c}$. Hence the $\mathcal{T}_{\Psi,I^c}$-summands vanish in the sum for the bounds and we obtain

$$\begin{aligned}
\mathrm{upperBd}_P^{\mathcal{T}}(I) - \varepsilon &= \sum_{t\in\mathcal{T}} \mathrm{vol}(t)(\sup \mathrm{wt}_P^{\mathbb{I}}(t))[\mathrm{val}_P^{\mathbb{I}}(t) \cap I \neq \varnothing] - \varepsilon \\
&= \sum_{\Psi=(\mathcal{V},n,\Delta,\Xi)} \sum_{t\in\mathcal{T}_{\Psi,I}} \left(\sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'') \sup \mathrm{wt}_P^{\mathbb{I}}(t'') - \varepsilon_t \right) \\
&\leq \underbrace{\sum_{\Psi=(\mathcal{V},n,\Delta,\Xi)} \sum_{t\in\mathcal{T}_{\Psi,I}} \int_{(\!|t|\!)} \prod_{\mathcal{W}\in\Xi} \mathcal{W}[s/\overline{\alpha}]\,\mathrm{d}s}_{=[\![P]\!](I)} \\
&\leq \sum_{\Psi=(\mathcal{V},n,\Delta,\Xi)} \sum_{t\in\mathcal{T}_{\Psi,I}} \left(\sum_{t'\in\mathcal{S}_t} \sum_{t''\in\mathcal{S}'_{t'}} \mathrm{vol}(t'') \min \mathrm{wt}_P^{\mathbb{I}}(t'') + \varepsilon_t \right) \\
&= \sum_{t\in\mathcal{T}} \mathrm{vol}(t)(\min \mathrm{wt}_P^{\mathbb{I}}(t))[\mathrm{val}_P^{\mathbb{I}}(t) \subseteq I] + \varepsilon \\
&= \mathrm{lowerBd}_P^{\mathcal{T}}(I) + \varepsilon
\end{aligned}$$

□

3.4 Weight-aware Interval Type System

To obtain sound bounds on the denotation with only finitely many interval traces, we present an interval-based type system that can derive static bounds on a program. Crucially, our type-system is *weight-aware*: we bound not only the return value of a program but also the weight of an execution. Our analyzer GuBPI uses it for two purposes. First, it allows us to derive upper bounds even for areas of the sample space not covered with interval traces. Second, we can use our analysis to derive a *finite* (and sound) approximation of the infinite number of symbolic execution paths of a program (more details are given in Section 3.5). Note that the bounds inferred by our

system are *interval bounds*, which allow for seamless integration with our interval trace semantics. In this section, we present the interval type system and sketch a constraint-based type inference method.

3.4.1 Interval Types

We define interval types by the following grammar:

$$\sigma := I \mid \sigma \to \mathcal{A} \qquad \mathcal{A} := \begin{Bmatrix} \sigma \\ I \end{Bmatrix}$$

where $I \in \mathbb{I}$ is an interval. For readers familiar with refinement types, it is easiest to view the type $\sigma = I$ as the refinement type $\{x : \mathbb{R} \mid x \in I\}$. The definition of the syntactic category $\mathcal{A}$ by mutual recursion with σ gives a bound on the weight of the execution. We call a type σ *weightless* and a type $\mathcal{A}$ *weighted*. The following examples should give some intuition about the types.

Example 3.4.1. Consider the example term

$$\left(\mathsf{fix}^{\varphi}_{x}.\, \{5 \times x\}\ [0.5]\ \{sigm(\varphi\, x + \mathsf{score}(\mathsf{sample}))\}\right)(4 \times \mathsf{sample})$$

where $sigm : \mathbb{R} \to [0,1]$ is the sigmoid function. In our type system, this term can be typed with the weighted type $\begin{Bmatrix} [0,20] \\ [0,1] \end{Bmatrix}$, which indicates that any terminating execution of the term reduces to a value (a number) within $[0,20]$ and the weight of any such execution lies within $[0,1]$.

Example 3.4.2. We consider the fixpoint subexpression of the pedestrian example in Example 2.3.3 which is

$$\mathcal{N} = \mathsf{fix}^{\varphi}_{x}.\, \mathsf{if}\ x \le 0\ \{0\}\ \mathsf{else}\ \{\left(\lambda \mathsf{step}.\mathsf{step} + \varphi(\{x + \mathsf{step}\}\ [0.5]\ \{x - \mathsf{step}\})\right)\mathsf{sample}\}$$

Using the typing rules (defined below), we can infer the type

$$\left\{ [a,b] \to \begin{Bmatrix} [0,\infty] \\ [1,1] \end{Bmatrix} \right\}_{[1,1]}$$

for any a, b. This type indicates that any terminating execution reduces to a function value (of simple type $\mathbf{R} \to \mathbf{R}$) with weight within $[1,1]$. If this function value is then called on a value within $[a,b]$, any terminating execution reduces to a value within $[0,\infty]$ with a weight within $[1,1]$.

Subtyping The partial order on intervals naturally extends to our type system. For base types I_1 and I_2, we define $I_1 \sqsubseteq_\sigma I_2$ just if $I_1 \sqsubseteq I_2$, where $\sqsubseteq$ is interval inclusion. We then extend this via:

$$\frac{\sigma_2 \sqsubseteq_\sigma \sigma_1 \qquad \mathcal{A}_1 \sqsubseteq_{\mathcal{A}} \mathcal{A}_2}{\sigma_1 \to \mathcal{A}_1 \sqsubseteq_\sigma \sigma_2 \to \mathcal{A}_2} \qquad \frac{\sigma_1 \sqsubseteq_\sigma \sigma_2 \qquad I_1 \sqsubseteq I_2}{\begin{Bmatrix} \sigma_1 \\ I_1 \end{Bmatrix} \sqsubseteq_{\mathcal{A}} \begin{Bmatrix} \sigma_2 \\ I_2 \end{Bmatrix}}$$

$$\frac{}{\Gamma \vdash \underline{r} : \left\{ \begin{matrix} [r,r] \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Lit} \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \left\{ \begin{matrix} \sigma \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Var} \qquad \frac{\Gamma \vdash M : \mathcal{A} \qquad \mathcal{A} \sqsubseteq_{\mathcal{A}} \mathcal{B}}{\Gamma \vdash M : \mathcal{B}}\ \textsc{Sub}$$

$$\frac{\Gamma; x : \sigma \vdash M : \mathcal{A}}{\Gamma \vdash \lambda x.M : \left\{ \begin{matrix} \sigma \to \mathcal{A} \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Abs} \qquad \frac{\Gamma; \varphi : \sigma \to \mathcal{A}; x : \sigma \vdash M : \mathcal{A}}{\Gamma \vdash \mathrm{fix}^{\varphi}_{x}.\, M : \left\{ \begin{matrix} \sigma \to \mathcal{A} \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Fix}$$

$$\frac{\Gamma \vdash M : \left\{ \begin{matrix} \sigma_1 \to \left\{ \begin{matrix} \sigma_2 \\ [e,f] \end{matrix} \right\} \\ [a,b] \end{matrix} \right\} \qquad \Gamma \vdash N : \left\{ \begin{matrix} \sigma_1 \\ [c,d] \end{matrix} \right\}}{\Gamma \vdash MN : \left\{ \begin{matrix} \sigma_2 \\ [a,b] \times^{\mathbb{I}} [c,d] \times^{\mathbb{I}} [e,f] \end{matrix} \right\}}\ \textsc{App}$$

$$\frac{\Gamma \vdash M_1 : \left\{ \begin{matrix} [a_1,b_1] \\ [c_1,d_1] \end{matrix} \right\} \qquad \cdots \qquad \Gamma \vdash M_{|f|} : \left\{ \begin{matrix} [a_{|f|},b_{|f|}] \\ [c_{|f|},d_{|f|}] \end{matrix} \right\}}{\Gamma \vdash f(M_1,\ldots,M_{|f|}) : \left\{ \begin{matrix} f^{\mathbb{I}}([a_1,b_1],\ldots,[a_{|f|},b_{|f|}]) \\ (\times^{\mathbb{I}})_{i=1}^{|f|} [c_i,d_i] \end{matrix} \right\}}\ \textsc{Prim}$$

$$\frac{\Gamma \vdash M : \left\{ \begin{matrix} [_,_] \\ [a,b] \end{matrix} \right\} \qquad \Gamma \vdash N : \left\{ \begin{matrix} \sigma \\ [c,d] \end{matrix} \right\} \qquad \Gamma \vdash P : \left\{ \begin{matrix} \sigma \\ [c,d] \end{matrix} \right\}}{\Gamma \vdash \mathrm{if}\ M \le 0\ \{N\}\ \mathrm{else}\ \{P\} : \left\{ \begin{matrix} \sigma \\ [a,b] \times^{\mathbb{I}} [c,d] \end{matrix} \right\}}\ \textsc{If}$$

$$\frac{}{\Gamma \vdash \mathrm{sample} : \left\{ \begin{matrix} [0,1] \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Sample} \qquad \frac{\Gamma \vdash M : \left\{ \begin{matrix} [a,b] \\ [c,d] \end{matrix} \right\}}{\Gamma \vdash \mathrm{score}(M) : \left\{ \begin{matrix} [a,b] \sqcap [0,\infty] \\ [c,d] \times^{\mathbb{I}} ([a,b] \sqcap [0,\infty]) \end{matrix} \right\}}\ \textsc{Score}$$

Figure 3.5: Weight-aware interval type system for SPCF. We abbreviate $\mathbf{1} := [1,1]$.

Note that in the case of weighted types, the subtyping requires not only that the weightless types be subtype-related ($\sigma_1 \sqsubseteq_\sigma \sigma_2$) but also that the weight bound be refined $I_1 \sqsubseteq I_2$. It is easy to see that both $\sqsubseteq_{\mathcal{A}}$ and $\sqsubseteq_\sigma$ are partial orders on types with the same underlying base type.

3.4.2 Type System

As for the interval-based semantics, we assume that every primitive operation $f : \mathbb{R}^n \to \mathbb{R}$ has an overapproximating interval abstraction $f^{\mathbb{I}} : \mathbb{I}^n \to \mathbb{I}$ (cf. Section 3.2.1). Interval

typing judgments have the form $\Gamma \vdash M : \mathcal{A}$ where Γ is a typing context mapping variables to types σ. They are given via the rules in Fig. 3.5. Our system is sound in the following sense, which we here only state for programs of ground type.

Theorem 3.4.3. *Let* $\vdash P : \mathbf{R}$ *be a simply-typed program. If* $\vdash P : \left\{\begin{matrix}[a,b]\\[c,d]\end{matrix}\right\}$ *and* $(P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w)$ *for some* $\boldsymbol{s} \in \mathbb{T}$ *and* $r, w \in \mathbb{R}$*, then* $r \in [a,b]$ *and* $w \in [c,d]$.

Proof sketch. The proof relies on standard lemmas: a substitution lemma and a (weighted) subject reduction lemma. Both are shown by a fairly straightforward structural induction. This theorem then follows from a repeated application of the subject reduction lemma. More details can be found in Appendix A.2.1. □

Note that the bounds derived by our type system only refer to terminating executions, i.e. they are partial correctness statements. Theorem 3.4.3 formalizes the intuition of an interval type, i.e. every type derivation in our system bounds *both* the returned value (in typical refinement-type fashion [Freeman and Pfenning 1991]) and the weight of this derivation. Our type system also comes with a weak completeness statement: for each term, we can derive (possibly trivial) bounds in our system.

Theorem 3.4.4. *Let* $\vdash P : \alpha$ *be a simply-typed program. There exists a weighted interval type* $\mathcal{A}$ *such that* $\vdash P : \mathcal{A}$.

Proof sketch. The idea is to replace $\mathbf{R}$ in α by $[-\infty, \infty]$ and add the weight $[0, \infty]$ to obtain $\mathcal{A}$. It is easy to see that these trivial bounds admit a type derivation. More details can be found in Appendix A.2.2. □

3.4.3 Constraint-based Type Inference

We briefly discuss the automated type *inference* in our system, as needed in our tool GuBPI. For space reasons, we restrict ourselves to an informal overview (see Appendix A.2.3 for a full account).

Given a program P, we can derive the symbolic skeleton of a type derivation (the structure of which is determined by P), where each concrete interval is replaced by a placeholder variable. The validity of a typing judgment within this skeleton can then be encoded as constraints. Crucially, as we work in the fixed interval domain and the subtyping structure $\sqsubseteq_{\mathcal{A}}$ is compositional, they are simple constraints over the placeholder variables in the abstract interval domain. Solving the resulting constraints via a naive fixpoint iteration might not terminate since the interval abstract domain has infinite ascending chains that never stabilize (such as $[0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq \ldots$). Instead, we approximate the least fixpoint (where the fixpoint denotes a solution to the constraints) using ***widening***, a standard approach to ensure termination of static analysis on domains with infinite strictly ascending chains [P. Cousot and R. Cousot 1977]. This is computationally much cheaper compared to, say, types with general first-order refinements where constraints are typically phrased as constrained Horn clauses (see e.g. Champion et al. [2020]). This gain in efficiency is crucial to making our GuBPI tool practical.

3.5 GuBPI and Optimizations for Linear Expressions

In this section, we describe the overall structure of our tool GuBPI (gubpi-tool.github.io), which builds upon symbolic execution. We also outline how the interval-based semantics can be accelerated for programs containing linear subexpressions. This section is less rigorous than the previous ones, as it serves as a high-level overview of our tool and its optimizations.

Algorithm 1 Symbolic Analysis in GuBPI.

1: **Input:** Program $\vdash P : \mathbf{R}$, depth limit $D \in \mathbb{N}$, and $I \in \mathbb{I}$
2: $\psi_{\mathsf{init}} := (P, 0, \varnothing, \varnothing); S := \{(\psi_{\mathsf{init}}, 0)\}; T := \varnothing$
3: **while** $\exists(\psi, \mathit{depth}) \in S$ **do**
4: **if** ψ has terminated **then**
5: $T := T \cup \{\psi\}; S := S \setminus \{(\psi, \mathit{depth})\}$
6: **else if** ψ contains no fixpoints or $\mathit{depth} \leq D$ **then**
7: $S := S \setminus \{(\psi, \mathit{depth})\}$
8: **for** ψ' with $\psi \rightsquigarrow \psi'$ **do**
9: $S := S \cup \{(\psi', \mathit{depth} + 1)\}$
10: **else**
11: $S := (S \setminus \{(\psi, \mathit{depth})\}) \cup \{(\mathit{approxFix}(\psi), \mathit{depth})\}$
12: **return** $\left[\sum_{\Psi \in T} [\![\Psi]\!]_{lb}(I), \sum_{\Psi \in T} [\![\Psi]\!]_{ub}(I)\right]$

3.5.1 GuBPI

In this section, we outline the structure of our analysis tool GuBPI (sketched in Algorithm 1). GuBPI's analysis begins with symbolic execution of the input term to accumulate a set of symbolic *interval* paths T. Analogously to Interval SPCF (Section 3.2), we define ***symbolic interval terms*** as symbolic terms that may contain intervals (and similarly for ***symbolic interval values***, ***symbolic interval configurations***, and ***symbolic interval paths***). If a symbolic configuration ψ has exceeded the user-defined depth limit D and still contains a fixpoint, we overapproximate all paths that extend ψ to ensure a finite set T. We accomplish this by using the interval type system (Section 3.4) to overapproximate all fixpoint subexpressions, thereby obtaining strongly normalizing terms (in line 11). Formally, given a symbolic configuration $\psi = (\mathcal{M}, n, \Delta, \Xi)$ we derive a typing judgment for the term $\mathcal{M}$ in the system from Section 3.4. Each first-order fixpoint subterm is thus given a (weightless) type of the form $[a, b] \to \left\{\begin{matrix}[c, d] \\ [e, f]\end{matrix}\right\}$. We replace this fixpoint with $\lambda_.\big(\mathsf{score}([e, f])\,;\,[c, d]\big)$. We denote this operation on configurations by $\mathit{approxFix}(\psi)$ (it extends to higher-order fixpoints as expected). Note that $\mathit{approxFix}(\psi)$ is a symbolic *interval* configuration.

Example 3.5.1. Consider the symbolic configuration given in Example 3.3.5. In the term $\mathcal{N}(3\alpha_1 - \alpha_2 + \alpha_4)$, we first infer the type of the argument as $\left\{\begin{matrix}[-1, 4] \\ [1, 1]\end{matrix}\right\}$. As in Example 3.4.2

we then infer the type of $\mathcal{N}$:

$$\mathcal{N} = \mathrm{fix}^{\varphi}_{x}.\ \mathrm{if}\ x \leq 0\ \{0\}\ \mathrm{else}\ \{(\lambda \mathrm{step}.\mathrm{step} + \varphi(\{x + \mathrm{step}\}\ [0.5]\ \{x - \mathrm{step}\}))\mathrm{sample}\}$$
$$: [-1, 4] \to \begin{Bmatrix} [0, \infty] \\ [1, 1] \end{Bmatrix}$$

The function *approxFix* replaces $\mathcal{N}$ with $\lambda_.\mathrm{score}([1,1]); [0,\infty]$. By evaluating the resulting symbolic interval configuration further, we obtain the symbolic interval path $(3\alpha_1, 5, \Delta, \Xi)$ where Δ is as in Example 3.3.5 and $\Xi = \{\mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}(\alpha_2 + \alpha_4 + [0,\infty])\}$. Note that, in general, the further evaluation of $\mathit{approxFix}(\psi)$ can result in multiple symbolic interval paths.

Afterwards, we are left with a finite set T of symbolic interval paths. Due to the presence of intervals, we cannot define a denotation of such paths directly and instead define lower and upper bounds. For a symbolic interval value $\mathcal{V}$ that contains *no* sample variables, we define $\ulcorner \mathcal{V} \urcorner \subseteq \mathbb{R}$ as the set of all values that the term can evaluate to by replacing every interval $[a, b]$ with some value $r \in [a, b]$. Given a symbolic interval path $\Psi = (\mathcal{V}, n, \Delta, \Xi)$ and $U \in \Sigma_{\mathbb{R}}$ we define its lower bound by considering only those concrete traces that satisfy the constraints in Ψ for *all* concrete values in the intervals and take the infimum over all scoring expressions:

$$[\![\Psi]\!]_{lb}(U) := \int \left[\ulcorner \mathcal{V}[s/\overline{\alpha}] \urcorner \subseteq U\right] \prod_{C \bowtie r \in \Delta} \left[\forall t \in \ulcorner C[s/\overline{\alpha}] \urcorner.\ t \bowtie r\right] \prod_{\mathcal{W} \in \Xi} \inf(\ulcorner \mathcal{W}[s/\overline{\alpha}] \urcorner)\, \mathrm{d}s$$

Similarly, we define the upper bound as

$$[\![\Psi]\!]_{ub}(U) := \int \left[\ulcorner \mathcal{V}[s/\overline{\alpha}] \urcorner \cap U \neq \varnothing\right] \prod_{C \bowtie r \in \Delta} \left[\exists t \in \ulcorner C[s/\overline{\alpha}] \urcorner.\ t \bowtie r\right] \prod_{\mathcal{W} \in \Xi} \sup(\ulcorner \mathcal{W}[s/\overline{\alpha}] \urcorner)\, \mathrm{d}s$$

We note that, if Ψ contains no intervals, $[\![\Psi]\!]$ is defined and we have $[\![\Psi]\!]_{lb} = [\![\Psi]\!]_{ub} = [\![\Psi]\!]$. We can now state the following theorem that formalizes the observation that $\mathit{approxFix}(\psi)$ soundly approximates all symbolic paths that result from ψ.

Theorem 3.5.2. *Let ψ be a symbolic (interval-free) configuration and $U \in \Sigma_{\mathbb{R}}$. Define $A = \mathit{symPaths}(\psi)$ as the (possibly infinite) set of all symbolic paths reached when evaluating ψ and $B = \mathit{symPaths}(\mathit{approxFix}(\psi))$ as the (finite) set of symbolic* interval *paths reached when evaluating $\mathit{approxFix}(\psi)$. Then*

$$\sum_{\Psi \in B} [\![\Psi]\!]_{lb}(U) \leq \sum_{\Psi \in A} [\![\Psi]\!](U) \leq \sum_{\Psi \in B} [\![\Psi]\!]_{ub}(U)$$

The correctness of Algorithm 1 is then a direct consequence of Theorems 3.3.8 and 3.5.2:

Corollary 3.5.3. *Let T be the set of symbolic interval paths computed when at line 12 of Algorithm 1 and $U \in \Sigma_{\mathbb{R}}$. Then*

$$\sum_{\Psi \in T} [\![\Psi]\!]_{lb}(U) \leq [\![P]\!](U) \leq \sum_{\Psi \in T} [\![\Psi]\!]_{ub}(U)$$

What remains is to compute (lower bounds on) $[\![\Psi]\!]_{lb}(I)$ and (upper bounds on) $[\![\Psi]\!]_{ub}(I)$ for a symbolic interval path $\Psi \in T$ and $I \in \mathbb{I}$. We first present the standard interval trace semantics (Section 3.5.2) and then a more efficient analysis for the case that Ψ contains only linear functions (Section 3.5.3).

3.5.2 Standard Interval Trace Semantics

For any symbolic interval path Ψ, we can employ the semantics as introduced in Section 3.2. Instead of applying the analysis to the entire program, we can restrict to the current path Ψ (intuitively, by adding a score(0) to all other program paths). The interval traces split the domain of each sample variable in Ψ into intervals. It is easy to see that for any compatible and exhaustive set of interval traces $\mathcal{T}$, we have $\mathrm{lowerBd}_\Psi^{\mathcal{T}}(U) \leq [\![\Psi]\!]_{lb}(U)$ and $[\![\Psi]\!]_{ub}(U) \leq \mathrm{upperBd}_\Psi^{\mathcal{T}}(U)$ for any $U \in \Sigma_{\mathbb{R}}$ (see Theorem 3.3.3 and 3.3.4). Applying the interval-based semantics on the level of symbolic interval paths maintains the attractive features, namely soundness and completeness (relative to the current path) of the semantics. Note that the intervals occurring in Ψ seamlessly integrate with our interval-based semantics.

3.5.3 Linear Interval Trace Semantics

In case the score values and the guards of all conditionals are linear, we can improve and speed up the interval-based semantics.

Assume all symbolic interval values appearing in Ψ are interval linear functions of $\overline{\alpha}$ (i.e. functions $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} +^{\mathbb{I}} [a,b]$ for some $\mathbf{w} \in \mathbb{R}^n$ and $[a,b] \in \mathbb{I}$). We assume, for now, that each symbolic value $\mathcal{W} \in \Xi$ denotes an interval-free linear function (i.e. a function $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} + r$). Fix some interval $I \in \mathbb{I}$. We first note that both $[\![\Psi]\!]_{lb}(I)$ and $[\![\Psi]\!]_{ub}(I)$ are the integral of a polynomial over a convex polytope: define

$$\mathfrak{P}_{lb} := \left\{ \boldsymbol{s} \in \mathbb{R}^n \mid \ulcorner \mathcal{V}[\boldsymbol{s}/\overline{\alpha}] \urcorner \subseteq I \wedge \bigwedge_{C \bowtie r \in \Delta} \forall t \in \ulcorner C[\boldsymbol{s}/\overline{\alpha}] \urcorner . t \bowtie r \right\}$$

which is a polytope.[4] Then $[\![\Psi]\!]_{lb}(I)$ is the integral of the polynomial $\overline{\alpha} \mapsto \prod_{\mathcal{W} \in \Xi} \mathcal{W}$ over $\mathfrak{P}_{lb}$. The definition of $\mathfrak{P}_{ub}$ (as the region of integration for $[\![\Psi]\!]_{ub}(I)$) is similar. Such integrals can be computed exactly [Baldoni et al. 2011], e.g. with the LattE tool [Loera et al. 2013]. Unfortunately, our experiments showed that this does not scale to interesting probabilistic programs.

Instead, we derive guaranteed bounds on the denotation by means of iterated volume computations. This has the additional benefit that we can handle non-uniform samples and non-linear expressions in Ξ. We follow an approach similar to that of the interval-based semantics in Section 3.3 but do not split/bound *individual sample variables* and instead directly bound *linear functions* over the sample variables. Let $\Xi = \{\mathcal{W}_1, \ldots, \mathcal{W}_k\}$. We define a ***box*** (by abuse of language) as an element $\boldsymbol{t} = \langle [a_1,b_1], \ldots, [a_k,b_k] \rangle$, where $[a_i,b_i]$ gives a bound on $\mathcal{W}_i$.[5] We define $lb(\boldsymbol{t}) := \prod_{i=1}^k a_i$ and $ub(\boldsymbol{t}) := \prod_{i=1}^k b_i$. The box $\boldsymbol{t}$ naturally defines a subset of $\mathfrak{P}_{lb}$ given by $\mathfrak{P}_{lb}^{\boldsymbol{t}} = \left\{ \boldsymbol{s} \in \mathfrak{P}_{lb} \mid \bigwedge_{i=1}^k \mathcal{W}_i[\boldsymbol{s}/\overline{\alpha}] \in [a_i,b_i] \right\}$. Then $\mathfrak{P}_{lb}^{\boldsymbol{t}}$ is again a polytope and we write $\mathrm{vol}(\mathfrak{P}_{lb}^{\boldsymbol{t}})$ for its volume. The definition of $\mathfrak{P}_{ub}^{\boldsymbol{t}}$ and $\mathrm{vol}(\mathfrak{P}_{ub}^{\boldsymbol{t}})$ is analogous. As for interval traces, we call two boxes $\boldsymbol{t}_1, \boldsymbol{t}_2$ *compatible* if the intervals are almost disjoint in at least one position. A set of boxes B is *compatible* if its

[4] For example, if C denotes the function $\overline{\alpha} \mapsto \mathbf{w}^\intercal \overline{\alpha} + [a,b]$ we can transform a constraint $\forall t \in \ulcorner C[\boldsymbol{s}/\overline{\alpha}] \urcorner . t \leq r$ into the linear constraint $\mathbf{w}^\intercal \overline{\alpha} + b \leq r$.

[5] Note the similarity to the interval trace semantics. While the ith position in an interval trace bounds the value of the ith sample variable, the ith entry of a box bounds the ith score value.

elements are pairwise compatible and *exhaustive* if $\bigcup_{t\in B} \mathfrak{P}^t_{lb} = \mathfrak{P}_{lb}$ and $\bigcup_{t\in B} \mathfrak{P}^t_{ub} = \mathfrak{P}_{ub}$ (cf. Section 3.2.3).

Lemma 3.5.4. *Let B be a compatible and exhaustive set of boxes. Then $\sum_{t\in B} lb(t) \cdot \mathrm{vol}\left(\mathfrak{P}^t_{lb}\right) \leq [\![\Psi]\!]_{lb}(I)$ and $[\![\Psi]\!]_{ub}(I) \leq \sum_{t\in B} ub(t) \cdot \mathrm{vol}\left(\mathfrak{P}^t_{ub}\right)$.*

As in the standard interval semantics, a finer partition into boxes yields more precise bounds. While the volume computation involved in Lemma 3.5.4 is expensive [Dyer and Frieze 1988], the number of splits on the linear functions is much smaller than that needed in the standard interval-based semantics. Our experiments empirically demonstrate that the direct splitting of linear functions (if applicable) is usually superior to the standard splitting. In GuBPI we compute a set of exhaustive boxes by first computing a lower and upper bound on each $\mathcal{W}_i \in \Xi$ over $\mathfrak{P}_{lb}$ (or $\mathfrak{P}_{ub}$) by solving a linear program (LP) and splitting the resulting range in evenly sized chunks.

Beyond uniform samples To allow for non-uniform samples, we can combine the standard interval trace semantics with the linear optimization. That is, in addition to bounding linear score functions, we also split and bound each non-uniform sample (as in the standard interval trace semantics). Suppose that α_i is sampled from some continuous distribution D. We then split the real line into chunks (the size and number of which is selected by a heuristic depending on D). For each such chunk $[a, b]$, we compute the volume and multiply by the lower and upper bounds of pdf_D on $[a, b]$. In this way, we can even approximate integrals where not all variables are sampled from a uniform distribution (without needing to resort to quantile functions). Our experiments show, that as long as the guards are still linear, this approach is advantageous compared to the naive interval-based semantics.

Beyond linear score functions While guards on conditionals are often linear, this is rarely the case for score expressions because one usually observes values from some non-uniform distribution with a non-linear PDF. Consider the pedestrian example again. While all guards are linear, the score expression has the form $\mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}(\mathcal{V})$. We can handle such *non-linear* score values by applying linear optimization to the linear subexpressions and interval arithmetic to the nonlinear parts. Formally, we assume that each symbolic value $\mathcal{W}_i \in \Xi$ we score with has the form $\mathcal{W}_i = f_i(\mathcal{Z}^1_i, \ldots, \mathcal{Z}^{m_i}_i)$ where $\mathcal{Z}^j_i$ for $1 \leq j \leq m_i$ denote *linear* functions of the sample variables and $f_i : \mathbb{R}^{m_i} \to \mathbb{R}$ is a possibly non-linear function. Every score expression can be written in this way. For instance, in the pedestrian example, we have $f_1 = \mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}$.

Let $\Xi' = \{\mathcal{Z}^j_i \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}\}$ be the set of all such linear functions. We bound each linear function in Ξ' using linear optimization as before. We obtain a box b (which now has dimension $|\Xi'|$ instead of $|\Xi|$) and define the weight $weight(b)$ by applying the interval liftings $f^{\mathbb{I}}_i$ of the non-linear functions f_i to the bounds for each argument. Formally,

$$weight(b) = f^{\mathbb{I}}_1(b^1_1, \ldots, b^{m_1}_1) \times^{\mathbb{I}} \cdots \times^{\mathbb{I}} f^{\mathbb{I}}_n(b^1_n, \ldots, b^{m_n}_n)$$

where b^j_i is the interval bound on $\mathcal{Z}^j_i$. Note that this strictly generalizes the approach outlined before since we can choose f_i as the identity if $\mathcal{W}_i$ is already linear. The

Table 3.1: Evaluation on selected benchmarks from Sankaranarayanan et al. [2013]. We give the times (in seconds) and bounds computed by Sankaranarayanan et al. [2013] and GuBPI.

			Sankaran. et al.		GuBPI	
Program	**Q**	**Query**	***t***	**Result**	***t***	**Result**
tug-of-war	Q1	$total_a_b < total_t_s$	1.29	$[0.6126, 0.6227]$	0.72	$[0.6134, 0.6135]$
tug-of-war	Q2	$total_a_s < total_b_t$	1.09	$[0.5973, 0.6266]$	0.79	$[0.6134, 0.6135]$
beauquier-etal-3	Q1	$count < 1$	1.15	$[0.5000, 0.5261]$	22.5	$[0.4999, 0.5001]$
example-book-simple	Q1*	$count \geq 2$	8.48	$[0.6633, 0.7234]$	6.52	$[0.7417, 0.7418]$
example-book-simple	Q2*	$count \geq 4$	10.3	$[0.3365, 0.3848]$	8.01	$[0.4137, 0.4138]$
example-cart	Q1	$count \geq 1$	2.41	$[0.8980, 1.1573]$	67.3	$[0.9999, 1.0001]$
example-cart	Q2	$count \geq 2$	2.40	$[0.8897, 1.1573]$	68.5	$[0.9999, 1.0001]$
example-cart	Q3	$count \geq 4$	0.15	$[0.0000, 0.1150]$	67.4	$[0.0000, 0.0001]$
example-ckd-epi-simple	Q1*	$f_1 \leq 4.4 \wedge f \geq 4.6$	0.15	$[0.5515, 0.5632]$	0.86	$[0.0003, 0.0004]$
example-ckd-epi-simple	Q2*	$f_1 \geq 4.6 \wedge f \leq 4.4$	0.08	$[0.3019, 0.3149]$	0.84	$[0.0003, 0.0004]$
example-fig6	Q1	$c \leq 1$	1.31	$[0.1619, 0.7956]$	21.2	$[0.1899, 0.1903]$
example-fig6	Q2	$c \leq 2$	1.80	$[0.2916, 1.0571]$	21.4	$[0.3705, 0.3720]$
example-fig6	Q3	$c \leq 5$	1.51	$[0.4314, 2.0155]$	24.7	$[0.7438, 0.7668]$
example-fig6	Q4	$c \leq 8$	3.96	$[0.4400, 3.0956]$	27.4	$[0.8682, 0.9666]$
example-fig7	Q1	$x \leq 1000$	0.04	$[0.9921, 1.0000]$	0.18	$[0.9980, 0.9981]$
example4	Q1	$x + y > 10$	0.02	$[0.1910, 0.1966]$	0.31	$[0.1918, 0.1919]$
example5	Q1	$x + y > z + 10$	0.06	$[0.4478, 0.4708]$	0.27	$[0.4540, 0.4541]$
herman-3	Q1	$count < 1$	0.47	$[0.3750, 0.4091]$	124	$[0.3749, 0.3751]$

definition of $approx(b)$ with the new weight definition still satisfies Lemma 3.5.4. This way, we can even approximate integrals over non-linear functions by means of simple volume computations. As our experiments (e.g. on the pedestrian example) show, this approximation is precise enough to obtain useful bounds on the posterior. It is important to note that while we can deal with non-linear score values in this way, we have to fall back to the standard semantics for non-linear guards.

Example 3.5.5. Consider the path $\Psi = (3\alpha_1, 5, \Delta, \Xi)$ derived in Example 3.5.1. We use 1-dimensional boxes to bound $\alpha_2 + \alpha_4$ (the single linear sub-expression of the symbolic values in Ξ). To obtain a lower bound on $[\![\Psi]\!]_{lb}(I)$, we sum over all boxes $\boldsymbol{t} = \langle [a_1, b_1] \rangle$ and take the product of vol $\left(\mathfrak{P}^{\boldsymbol{t}}_{lb}\right)$ with the lower interval bound of $\mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}([a_1, b_1] + [0, \infty])$ (evaluated in interval arithmetic). Analogously, for the upper bound we take the product of vol $\left(\mathfrak{P}^{\boldsymbol{t}}_{ub}\right)$ with the upper interval bound of $\mathrm{pdf}_{\mathsf{Normal}(1.1,0.1)}([a_1, b_1] + [0, \infty])$.

3.6 Practical Evaluation

We have implemented our semantics in the prototype GuBPI (gubpi-tool.github.io), written in F#. In cases where we apply the linear optimization of our semantics, we use Vinci [Büeler et al. 2000] to discharge volume computations of convex polytopes. We set out to answer the following questions:

1. How does GuBPI perform on instances that could already be solved (e.g. by PSI [Gehr et al. 2016])?

2. Is GuBPI able to infer useful bounds on recursive programs that could not be handled rigorously before?
3. How does GuBPI compare to statistical validation methods?

Experimental setup Our experiments were performed on a server running Ubuntu 18.04 with a 4-core/8-thread Intel(R) Xeon(R) E3-1271 v3 @ 3.60 GHz CPU and 32 GB of RAM. The current version of GuBPI is not parallelized and makes no use of the additional cores. The running times on a Macbook Pro with Apple M1 were comparable, and sometimes even faster.

3.6.1 Probability Estimation

We collected a suite of 18 benchmarks from Sankaranarayanan et al. [2013]. Each benchmark consists of a program P and a query φ over the variables of P. We bound the probability of the event described by φ using the tool from Sankaranarayanan et al. [2013] and GuBPI (Table 3.1). While our tool is generally slower than the one in Sankaranarayanan et al. [2013], the completion times are still reasonable. Moreover, in many cases, the bounds returned by GuBPI are tighter than those of Sankaranarayanan et al. [2013]. In addition, for benchmarks marked with a $\star$, the two pairs of bounds contradict each other.[6] We should also remark that GuBPI cannot handle all benchmarks proposed in Sankaranarayanan et al. [2013] because the heavy use of conditionals causes our precise symbolic analysis to suffer from the well-documented path explosion problem [Boonstoppel et al. 2008]. Perhaps unsurprisingly, Sankaranarayanan et al. [2013] can handle such examples much better, as one of their core contributions is a stochastic method to reduce the number of paths considered (see Section 3.7). Also note that Sankaranarayanan et al. [2013] is restricted to uniform samples, linear guards and score-free programs, whereas we tackle a much more general problem.

3.6.2 Comparison with Exact Inference

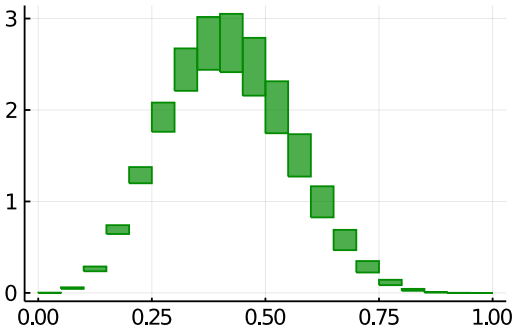
To evaluate our tool on instances that can be solved exactly, we compared it with PSI [Gehr et al. 2016; Gehr et al. 2020], a symbolic solver which can, in certain cases, compute a closed-form solution of the posterior. We note that whenever exact inference is possible, exact solutions will always be superior to mere bounds and, due to the overhead of our semantics, will often be found faster. Because of the different output formats (i.e. exact results vs. bounds), a direct comparison between exact methods and GuBPI is challenging. As a consistency check, we collected benchmarks from the PSI repository where the output domain is finite and GuBPI can therefore compute *exact* results (tight bounds). They agree with PSI in all cases, which includes 8 of the 21 benchmarks from Gehr et al. [2016]. We report the computation times in Table 3.2.

We then considered examples where GuBPI computes non-tight bounds. The bounds computed by GuBPI and a short description of each example are shown in Fig. 3.6. We

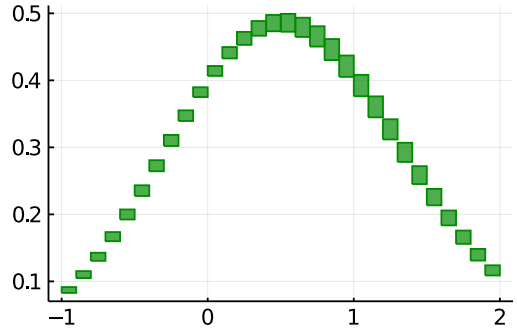
[6] A stochastic simulation using 10^6 samples in Anglican [Tolpin et al. 2015] yielded results that fall within GuBPI's bounds but violate those computed by Sankaranarayanan et al. [2013].

Table 3.2: Probabilistic programs with discrete domains from PSI [Gehr et al. 2016]. The times for PSI and GuBPI are given in seconds.

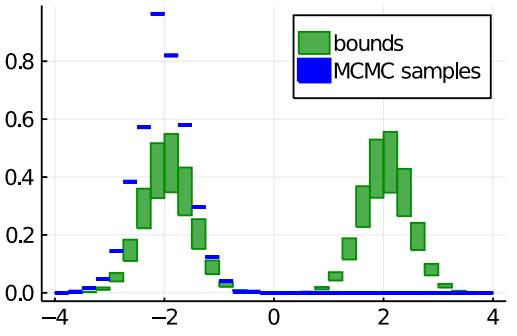
Instance	t_{PSI}	t_{GuBPI}
burglarAlarm	0.06	0.21
coins	0.04	0.18
twoCoins	0.04	0.21
ev-model1	0.04	0.21
grass	0.06	0.37
ev-model2	0.04	0.20
noisyOr	0.14	0.72
murderMystery	0.04	0.19
bertrand	0.04	0.22
coinBiasSmall	0.13	1.92
coinPattern	0.04	0.19
gossip	0.08	0.24



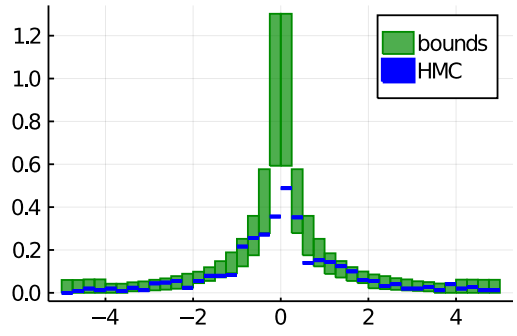
(a) `coinBias` example from Gehr et al. [2016]. The program samples a beta prior on the bias of a coin and observes repeated coin flips (26 seconds).



(b) `max` example from Gehr et al. [2016]. The program computes the maximum of two i.i.d. normal samples (31 seconds).

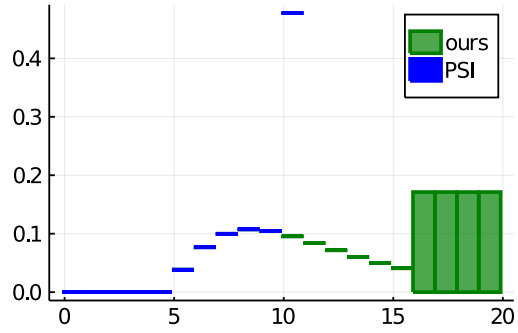


(c) Binary Gaussian Mixture Model from Zhou et al. [2019] (39 seconds). MCMC methods usually find only one mode.

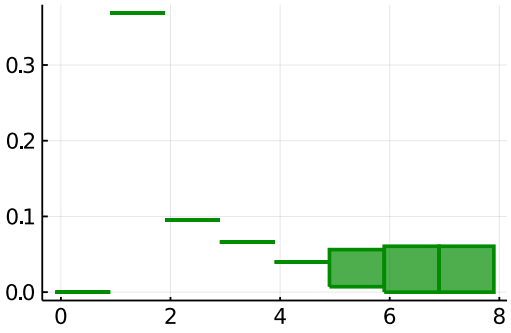


(d) Neal's funnel from Neal [2003] and Gorinova et al. [2020] (2.8 seconds). HMC misses some probability mass around 0.

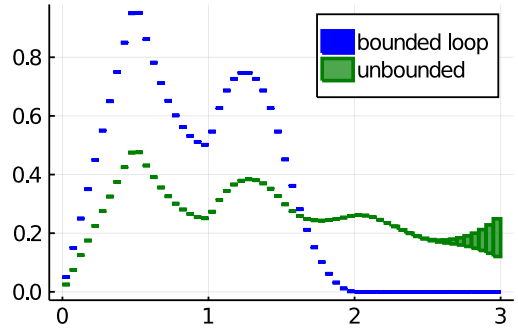
Figure 3.6: Guaranteed Bounds computed by GuBPI for a selection of non-recursive models from Gehr et al. [2016], Gehr et al. [2020], Zhou et al. [2019], and Neal [2003].



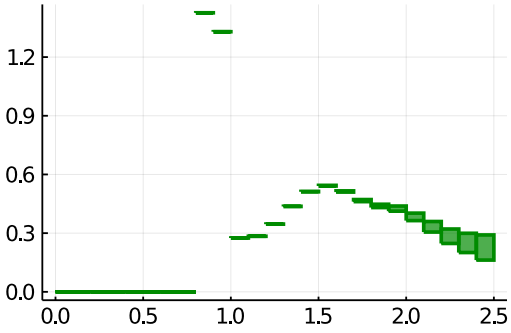
(a) `cav-example-7`. Program taken from the PSI repository. PSI bounds the depth resulting in a spike at 10, whereas GuBPI can compute bounds on the denotation of the unbounded program (112 seconds).



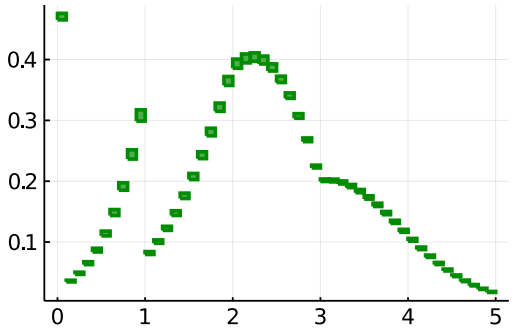
(b) `cav-example-5`. Program taken from the PSI repository. PSI cannot handle this program due to the unbounded loops (192 seconds).



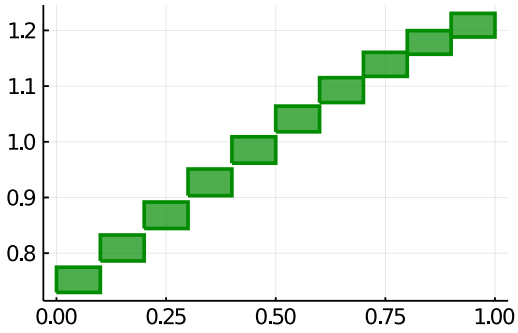
(c) `add_uniform_with_counter_large`. Program taken from the PSI repository. GuBPI can handle the unbounded loop, whereas PSI unrolls to a fixed depth (21 seconds).



(d) `random-box-walk`. The program models the cumulative distance traveled by a biased random walk. If a uniformly sampled step s has length less than $\frac{1}{2}$, we move s to the left, otherwise s to the right. The walk stops when it crosses a threshold (167 seconds).



(e) `growing-walk`. The program models a geometric random walk where (with increasing distance) the step size of the walk is increased. The cumulative distance is observed from a normal distribution centered at 3 (67 seconds).



(f) `param-estimation-recursive`. We sample a uniform prior p and (in each step) travel to the left with probability p and to the right with probability $(1-p)$. We observe the walk to come to a halt at location 1 (observed from a normal) and wish to find the posterior on p (162 seconds).

Figure 3.7: Guaranteed bounds computed by GuBPI for a selection of recursive models.

can see that, despite the relatively loose bounds, they are still useful and provide the user with a rough – and most importantly, *guaranteed to be correct* – idea of the denotation.

The success of exact solvers such as PSI depends on the underlying symbolic solver (and the optimizations implemented). Consequently, there are instances where the symbolic solver cannot compute a closed-form (integral-free) solution. Conversely, while our method is (theoretically) applicable to a very broad class of programs, there exist programs where the symbolic solver finds solutions but the analysis in GuBPI becomes infeasible due to the large number of interval traces required.

3.6.3 Recursive Models

We also evaluated our tool on complex models that *cannot* be handled by any of the existing methods. Unexpectedly, we found recursive models in the PSI repository: there are examples that are created by unrolling loops to a fixed depth. This fixed unrolling changes the posterior of the model. Using our method we can handle those examples *without* bounding the loop. Three such examples are shown in Figs. 3.7a to 3.7c. In Fig. 3.7a, PSI bounds the iterations resulting in a spike at 10 (the unrolling bound). For Fig. 3.7b, PSI does not provide any solution whereas GuBPI provides useful bounds. For Fig. 3.7c, PSI bounds the loop to compute results (displayed in blue) whereas GuBPI computes the green bounds on the unbounded program. It is obvious that the bounds differ significantly, highlighting the impact that unrolling to a fixed depth can have on the denotation. This again strengthens the claim that rigorous methods that can handle unbounded loops/recursion are needed. There also exist unbounded discrete examples where PSI computes results for the bounded version that differ from the denotation of the unbounded program. Figs. 3.7d to 3.7f depict further recursive examples (alongside a small description).

Lastly, as a *very* challenging example, we consider the pedestrian example (Example 2.2.3) again. The bounds computed by GuBPI are given in Fig. 3.8 together with the two stochastic results from Fig. 3.1. The bounds are clearly precise enough to rule out the HMC samples generated by Pyro [Bingham et al. 2019].[7] Since this example has infinite expected running time, it is very challenging and GuBPI takes about 1.5h (84 min). While the running time seems high, we note that Pyro's HMC took about an hour to generate 10^4 samples and produce the (wrong) histogram. Diagnostic methods like simulation-based calibration took even longer (>30h) and delivered inconclusive results (see Section 3.6.4 for details). In fact, guaranteed bounds are the only method to recognize the wrong samples with certainty.

[7]The samples were obtained as follows. To address the infinite expected running time of the original pedestrian program, we introduced a stopping condition in the random walk, exiting the loop if the distance exceeded 10. This has a negligible impact on the posterior distribution because the weight of such a trace is $\leq \mathrm{pdf}_{\mathrm{Normal}(1.1,0.1)}(10) < 10^{-1700}$. We ran 10 Markov chains with 1000 samples each using the HMC sampler (step size: 0.1, step count: 50). Although we also tried the NUTS sampler, it performed worse. The chains' running time varied from one minute to nearly an hour, depending on trace length (ranging from 2 to about 200). Chains with acceptance rates under 1% were discarded (as this indicates bad mixing), and the remaining chains (over 50% acceptance) were aggregated for the histogram plots.

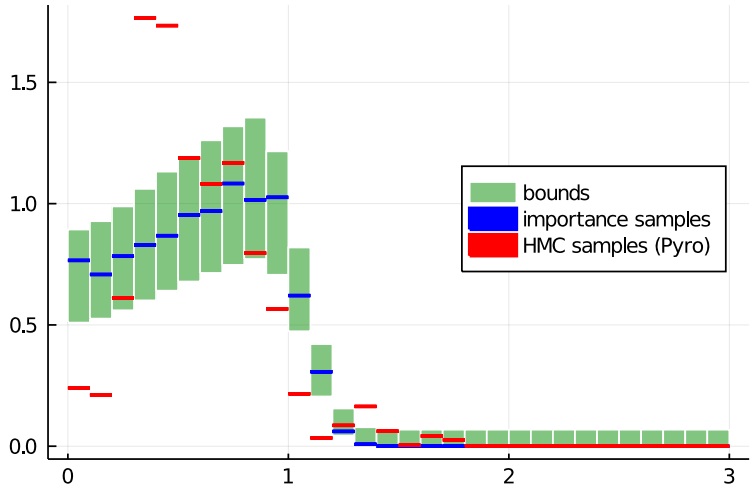


Figure 3.8: Bounds for the pedestrian example (Example 2.2.3).

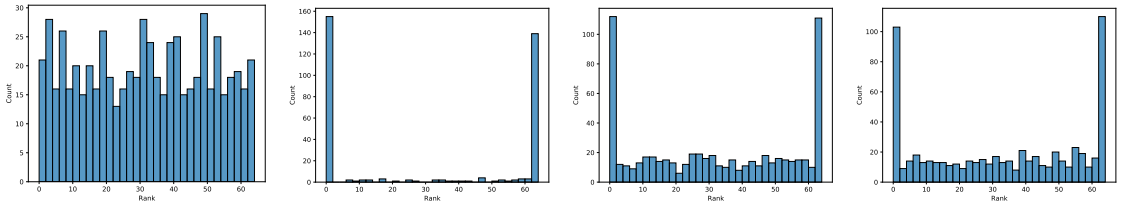
Table 3.3: Running times of GuBPI and SBC for (Pyro's) HMC. Times are given in seconds (s) and hours (h).

Instance	t_{GuBPI}	t_{SBC}
Binary GMM (1-dimensional) (Fig. 3.6c)	39s	1h
Binary GMM (2-dimensional)	4h	1.5h
Pedestrian Example (Fig. 3.8)	1.5h	>300h

3.6.4 Comparison with Statistical Validation

Methodology We compare the running times of GuBPI and a statistical validation method for three examples where Pyro's HMC yields wrong results (Table 3.3). Specifically, we implemented *simulation-based calibration* (SBC, see Section 2.5.6) for both likelihood-weighted importance sampling and Pyro's HMC. As hyperparameters for SBC, we picked $L = 63$ samples per simulation (following the suggestion in Talts et al. [2018] to take one less than a power of two) and N simulations with $N = 10L$ (also following the paper's suggestion). Note that the number of samples is much less than the 10000 samples used for Figs. 3.1 and 3.8 (10 chains with 1000 samples each). But setting $L = 1000$ would be at least 100 times slower because N has to increase proportionally to L. Also note that for Pyro, we ran HMC with L warmup steps before generating L samples. Both importance samples and HMC samples exhibited significant autocorrelation. As suggested in Talts et al. [2018], we applied thinning to reduce its effect, choosing a thinning factor of around $\frac{L}{L_{eff}}$ where L_{eff} is the effective sample size.

Results We first discuss the pedestrian example. For importance sampling, the rank histogram looks fairly uniform (Fig. 3.9a), which means that SBC does not detect an issue with the inference and thus increases the confidence in the validity of the importance samples. For Pyro's HMC, simulation-based calibration is very slow: the



(a) Importance sampling on the pedestrian example, thinning factor 100.

(b) HMC on the pedestrian example, no thinning. (Only 332 simulations were used because the rest had too low acceptance rates.)

(c) NUTS on the binary Gaussian mixture model, thinning factor 10.

(d) NUTS run on the *two-dimensional* binary Gaussian mixture model, thinning factor 10.

Figure 3.9: Simulation-based calibration: rank histogram plots (630 simulations with 63 samples each).

rank histogram (Fig. 3.9b) took 32 hours (!) to produce. (Note that only 332 simulations could be used, the rest were discarded because the acceptance rate was too low.) The results were also *inconclusive*. The spikes at the boundary indicate that the samples have high autocorrelation and in fact, the effective sample size L_{eff} was at most $\frac{L}{10}$, often much lower (depending on the chain). The suggestion in Talts et al. [2018] is thus to apply thinning, with a factor of $\frac{L}{L_{eff}}$, which is at least 10 in our case. This would increase the running time of SBC by the same factor, to at least 300 hours, but probably 600 or more. We did not consider it a good use of resources to carry out this experiment.

We also considered the binary GMM (Fig. 3.6c) and a two-dimensional version of the same model. The spikes at the boundary could again be a sign of high autocorrelation, but in this case, we already applied thinning with a factor of 10 (again based on the effective sample size). Instead, as discussed in Talts et al. [2018], this symmetric U-shape indicates that the computed data-averaged posterior is underdispersed relative to the prior distribution. This interpretation is consistent with our knowledge about the model: HMC only finds one mode in the posterior distribution and misses the others. Hence SBC successfully detects the issue, and in the case of the higher-dimensional model, it does so in less time than GuBPI (cf. Table 3.3). For the other models, including the pedestrian example, GuBPI is faster.

3.7 Related Work

Several related approaches were already discussed in Chapter 2, such as exact inference (Section 2.4), statistical checks for inference (Section 2.5.6), and quantitative verification techniques (Section 2.6). Here we focus on approaches that are most relevant to our work.

Polynomial solving The most closely related approach is by P. Wang et al. [2024]. They aim to solve the same problem as GuBPI but achieve better efficiency and precision of the bounds via two new techniques. Their setting is more restricted than ours: their imperative PPL is more similar to SGCL than SPCF because it operates on a fixed number of variables and supports while loops but no general recursion. Furthermore,

its primitive functions are restricted to addition and multiplication. However, like SPCF and unlike SGCL, their PPL supports soft conditioning with a score statement, which can express observations from continuous distributions.

Their first approach improves the efficiency of GuBPI via a different fixpoint approximation technique. They use an expected-weight transformer semantics, which is closer to the measure transformer semantics (Section 2.3.1) than our interval trace semantics. To bound the distribution of loops, their main idea is to approximate fixpoints of this transformer with polynomials, which are more precise than our piecewise constant bounds on the weight function. They synthesize such polynomial inductive invariants by solving the linear or semidefinite constraints arising from the inductive property. Similarly to our interval traces, they also split the support of program variables into intervals and analyze each partition separately, but their method is less reliant on a fine splitting due to the higher precision of the polynomial approximation. Furthermore, since they do not rely on unrolling or symbolic execution, their approach does not suffer from the path explosion problem. As a consequence, their method is typically faster (by a factor of 1.5 to 15) and usually yields better bounds than GuBPI on the benchmarks from Section 3.6.3.

Their second approach targets loops containing score(w) statements with a weight $w > 1$. For such *score-recursive* loops, GuBPI yields trivial upper bounds because the interval type system can only bound the total weight of the loop by ∞. They can find nontrivial bounds on the posterior distribution of score-recursive loops with bounded variable updates by means of a new multiplicative variant of the classical Optional Stopping Theorem from martingale theory.

In terms of theoretical guarantees for both approaches, they prove soundness but lack a completeness result like our Theorem 3.3.13.

Interval trace semantics and Interval SPCF Our interval trace semantics to compute bounds on the denotation is inspired by the semantics introduced by Beutner and Ong [2021], who study an interval approximation to obtain *lower* bounds on the termination probability. By contrast, we study the more challenging problem of bounding the program denotation. This requires us to track the weight of an execution, and to prove that the denotation approximates a Lebesgue integral, which depends on novel proof ideas. Moreover, whereas the termination probability of a program is always upper bounded by 1, here we derive both lower and *upper* bounds.

Probability estimation Sankaranarayanan et al. [2013] introduced a static analysis framework to infer bounds on a class of definable events in (*score-free*) probabilistic programs. The idea of their approach is that if we find a finite set $\mathcal{T}$ of symbolic traces with cumulative probability at least $1 - c$, and a given event φ occurs with probability at most b on the traces in $\mathcal{T}$, then φ occurs with probability at most $b + c$ on the entire program. In the presence of conditioning, the problem becomes vastly more difficult, as the aggregate weight on the unexplored paths can be unbounded, giving ∞ as the only derivable upper bound (and therefore also ∞ as the best upper bound on the normalizing constant). In order to infer guaranteed bounds, it is necessary to analyze *all* paths in the program, which we accomplish via static analysis and in

particular our interval type system. The approach from Sankaranarayanan et al. [2013] was extended by Albarghouthi et al. [2017] to compute the probability of events defined by arbitrary SMT constraints but is restricted to score-free and non-recursive programs. Our interval-based approach, which may be seen as a variant of theirs, is founded on a complete semantics (Theorem 3.3.13), can handle recursive program with scoring, and is applicable to a broad class of primitive functions.

Note that we consider programs with *soft* conditioning in which scoring cannot be reduced to volume computation directly.[8] Intuitively, soft conditioning performs a global re-weighting of the set of traces, which cannot be captured by local volume computations. In our interval trace semantics, we instead track an approximation of the weight along each interval trace.

Huang et al. [2021] discretize the domain of continuous samples into interval cubes and derive posterior approximations on each cube. The resulting approximation converges to the true posterior (similarly to approximate/stochastic methods) but does not provide exact/guaranteed bounds and is not applicable to recursive programs.

Probabilistic abstract interpretation Monniaux [2000] develops an abstract domain for score-free probabilistic programs given by a weighted sum of abstract regions. Smith [2008] considers truncated normal distributions as an abstract domain and developed analyses restricted to score-free programs with only linear expressions. Extending both approaches to support soft conditioning is non-trivial as it requires the computation of integrals on the abstract regions. In our interval-based semantics, we abstract the concrete traces (by means of interval traces) and not the denotation. This allows us to derive bounds on the weight along the abstracted paths.

Refinement types Our interval type system (Section 3.4) may be viewed as a *refinement type system* [Freeman and Pfenning 1991] that refines not just the value of an expression but also its weight. To our knowledge, no existing refinement type system can bound the weight of program executions. Moreover, the seamless integration with our interval trace semantics by design allows for much cheaper type inference, without resorting to an SMT or Horn constraint solver. This is a crucial advantage since a typical GuBPI execution queries the analysis numerous times.

3.8 Conclusion

We have studied the problem of inferring guaranteed bounds on the posterior of programs written in a universal PPL. Our work is based on the interval trace semantics, and our weight-aware interval type system gives rise to a tool that can infer useful bounds on the posterior of interesting recursive programs. This is a capability beyond the reach of existing methods, such as exact inference. As a method of Bayesian inference for statistical probabilistic programs, we can view our framework as occupying a useful middle ground between approximate stochastic inference and exact inference.

[8] For programs including only hard-conditioning (i.e. scoring is only possible with 0 or 1), the posterior probability of an event φ can be computed by dividing the probability of all traces with weight 1 on which φ holds by the probability of all traces with weight 1.

Future Improvements The theoretical foundations of our interval-based semantics ensure that GuBPI is applicable to a very broad class of programs (cf. Section 3.3). In practice, as usual for exact methods, GuBPI does not handle all examples equally well.

Firstly, as we already saw in Section 3.6.1, the symbolic execution – which forms the entry point of the analysis – suffers from path explosion. On some extreme recursion-free programs (such as example-ckd-epi from Sankaranarayanan et al. [2013]), our tool cannot compute all (finitely many) symbolic paths in reasonable time, let alone analyze them in our semantics. Extending the approach from Sankaranarayanan et al. [2013], to sample representative program paths (in the presence of conditioning), is an interesting future direction that we can combine with the rigorous analysis provided by our interval type system.

Secondly, our interval-based semantics imposes bounds on each sampled variable and thus scales exponentially with the dimension of the model; this is amplified in the case where the optimized semantics (Section 3.5.3) is not applicable. It would be interesting to explore whether this can be alleviated using different trace splitting techniques.

Lastly, the bounds inferred by our interval type system take the form of a single interval with no information about the exact distribution on that interval. For example, the most precise bound derivable for the term $\mathsf{fix}^{\varphi}_{x}.\,\{x\}\ [0.5]\ \{\{\varphi(x+\mathsf{sample})\}\ [0.5]\ \{\varphi(x-\mathsf{sample})\}\}$ is $[a,b] \to \left\{ \begin{matrix} [-\infty,\infty] \\ [1,1] \end{matrix} \right\}$ for any a, b. After unrolling to a fixed depth, the approximation of the paths not terminating within the fixed depth is therefore imprecise. For future work, it would be interesting to improve the bounds in our type system to provide more information about the distribution by means of rigorous approximations of the denotation of the fixpoint in question, i.e. a probabilistic summary of the fixpoint [D. Wang et al. 2018].

The full beauty of the subject of generating functions emerges only from tuning in on both channels: the discrete and the continuous.

— Herbert Wilf

4

Exact Inference via Generating Functions

Contents

4.1 Introduction . . . 93
4.1.1 Contributions . . . 93
4.1.2 Limitations . . . 94
4.1.3 Introductory example . . . 94
4.2 Generating Functions . . . 96
4.2.1 Generating Function Semantics . . . 97
4.2.2 Correctness . . . 101
4.2.3 Possible Language Extensions . . . 108
4.3 Implementation and Optimizations . . . 109
4.3.1 Automatic Differentiation and Taylor polynomials . . . 109
4.3.2 Analysis of the Running Time . . . 112
4.3.3 Reducing the Number of Variables . . . 114
4.3.4 Implementation . . . 118
4.4 Empirical Evaluation . . . 121
4.4.1 Comparison with Exact Inference Methods . . . 121
4.4.2 Comparison with Approximate Inference Methods . . . 122
4.4.3 Benchmarks with Infinite-Support Distributions . . . 124
4.5 Related Work . . . 126
4.6 Conclusion . . . 127

Summary This chapter introduces an exact Bayesian inference method for *loop-free* probabilistic programs written in SGCL (Section 2.2.1). In the context of this thesis, it lays important groundwork for establishing guaranteed bounds for SGCL programs with loops in Chapter 5. Furthermore, it represents a significant contribution in its own right, as a novel exact inference method for a large class of discrete inference problems, even with infinite support and continuous priors.

Our key tool is *probability generating functions*: they provide a compact closed-form representation of distributions that are definable by programs, thus enabling the exact computation of posterior probabilities, expectation, variance, and higher moments. Our

inference method is provably correct and fully automated in a tool called *Genfer*, which uses automatic differentiation (specifically, Taylor polynomials), but does not require computer algebra. Our experiments show that Genfer is often faster than the existing exact inference tools PSI [Gehr et al. 2016], Dice [Holtzen et al. 2020], and Prodigy [Klinkenberg et al. 2024]. On a range of real-world inference problems that none of these exact tools can solve, Genfer's performance is competitive with approximate Monte Carlo methods, while avoiding approximation errors.

Publication This chapter is based on a paper accepted for oral presentation at *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. The full reference is:

> Fabian Zaiser, Andrzej S. Murawski, and Luke Ong [2023]. "Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*

This chapter is a slightly extended version: it includes a generating function semantics for more general events and for the $X \doteq 1$ statement.

Author contributions Andrzej Murawski and Luke Ong supervised the project, provided feedback, and helped revise the paper. I conceived the project idea, developed the theory, implemented the approach, conducted the experiments, and wrote the paper.

4.1 Introduction

A key challenge in Bayesian statistics is computing the posterior distribution: analytical solutions are usually impossible or intractable, which necessitates the use of approximate methods, such as Markov-chain Monte Carlo (MCMC) or variational inference. In this work, we identify a large class of discrete models for which exact inference is in fact possible, in particular time series models of count data, such as autoregressive models, hidden Markov models, and change point models. We achieve this by leveraging ***probability generating functions (GFs)*** as a representation of distributions. The GF of a random variable X is defined to be the function $G(x) := \mathbb{E}[x^X]$. In probability theory, GFs are a well-known tool related to the moment generating and the characteristic functions [Gut 2013, Chapter 4], and they are particularly useful for studying random variables with infinite support, such as in the context of branching processes [Feller 1968, Chapter 12]. In computer science, GFs have previously been used for the analysis of probabilistic programs [Klinkenberg et al. 2020; Chen et al. 2022], which provided the original inspiration for our work, and for exact inference on certain classes of graphical models [Winner and Sheldon 2016; Winner et al. 2017]. Here we apply them systematically in a much more general context via probabilistic programming, enabling exact inference on more expressive Bayesian models.

4.1.1 Contributions

We provide a new framework for exact inference on discrete Bayesian models.

- Our method is applicable to a large class of *discrete models with infinite support*, in particular time series models of count data, such as autoregressive models for population dynamics, Bayesian change point models, mixture models, and hidden Markov models. To our knowledge, no exact inference method for any of these was known before, except for the simplest population models.
- The models are specified as loop-free probabilistic programs in SGCL, which supports stochastic branching, continuous and discrete priors, and conditioning on events involving discrete variables.
- Every loop-free SGCL program can be *translated to a generating function* that represents the posterior distribution in an automatic and provably correct way. From this generating function, one can extract posterior *mean, variance, and higher moments*, as well as the posterior *probability masses* (for a discrete distribution).
- We have built an *optimized tool*, called ***Genfer*** ("**Gen**erating **f**unctions for in**fer**ence"), that takes a probabilistic program as input and *automatically computes* the aforementioned set of descriptive statistics about the posterior distribution.
- We demonstrate that (1) on benchmarks with finite support, Genfer's performance is often better than existing *exact inference tools*, and (2) on a range of real-world examples that no existing exact tool supports, Genfer is competitive with *approximate Monte Carlo methods*, while achieving zero approximation error.

4.1.2 Limitations

Exact posterior inference is already PSPACE-hard for loop-free probabilistic programs involving only finite discrete distributions [Holtzen et al. 2020, Section 6]. It follows that our method cannot always be performant and has to restrict the supported class of probabilistic programs. Indeed, our programming language SGCL forbids certain constructs, such as nonlinear transformations and observations of continuous variables, in order to preserve a closed-form representation of the generating functions (Section 2.2.1). For the same reason, our method cannot compute probability density functions for continuous parameters (but probability masses for discrete parameters and exact moments for all parameters can be computed).

Regarding performance, the running time of inference is polynomial in the numbers observed in the program and exponential in the number of program variables (Section 4.3). Despite these limitations, our approach shows that exact inference is possible in the first place, and our evaluation (Section 4.4) demonstrates support for a range of real-world models with efficient exact inference in practice.

4.1.3 Introductory example

We first discuss a simple program using a simplified example based on population ecology (cf. Winner and Sheldon [2016]), whose posterior distribution is challenging to compute, however. Suppose you are a biologist trying to estimate the size of an animal population migrating into a new habitat. The immigration of animals is often modeled using a Poisson distribution. You cannot count the animals exhaustively (otherwise we would not need estimation techniques), so we assume that each individual is observed

independently with a certain probability; in other words, the count is binomially distributed. For simplicity, we assume that the rate of the Poisson and the probability of the binomial distribution are known, say 20 and 0.1. (For more realistic population models, see Section 4.4.) As a generative model, this would traditionally be written as $X \sim \text{Poisson}(20); Y \sim \text{Binomial}(X, 0.1)$.

Suppose you observe $Y = 2$ animals. The Bayesian inference problem is to compute the posterior distribution $\mathbb{P}(X = x \mid Y = 2)$ given by Bayes' rule as

$$\mathbb{P}(X = x \mid Y = 2) = \frac{\mathbb{P}(X = x)\ \mathbb{P}(Y = 2 \mid X = x)}{\mathbb{P}(Y = 2)}$$

where $\mathbb{P}(X = x)$ is called the prior probability, $\mathbb{P}(Y = 2 \mid X = x)$ the likelihood and $\mathbb{P}(Y = 2)$ the evidence or normalization constant.

Example 4.1.1. In SGCL, this simplified population model would be expressed as:

$$\begin{aligned} &X \sim \text{Poisson}(20); \\ &Y \sim \text{Binomial}(X, 0.1); \\ &\text{observe } Y = 2 \end{aligned}$$

We briefly and informally review the simplified semantics of SGCL programs (see Section 2.3.1). Such a program denotes a joint probability distribution of its program variables, which are viewed as random variables. The program statements manipulate this joint distribution. After the first two sampling statements, the distribution has the probability mass function (PMF)

$$p_1(x, y) = \mathbb{P}[X = x, Y = y] = \text{Poisson}(x; 20) \cdot \text{Binomial}(y; x, 0.1)$$

Observing $Y = 2$ restricts this to the PMF

$$p_2(x, y) = \mathbb{P}[X = x, Y = y = 2] = \begin{cases} p_1(x, 2) & \text{if } y = 2 \\ 0 & \text{otherwise} \end{cases}$$

Note that this is not a probability, but a *subprobability*, distribution because the total mass is less than 1. So, as a final step, we need to *normalize*, i.e. rescale the subprobability distribution back to a probability distribution. This corresponds to the division by the evidence in Bayes' rule, yielding the PMF

$$p_3(x, y) = \frac{p_2(x, y)}{\sum_{x,y} p_2(x, y)}$$

To obtain the posterior $\mathbb{P}[X = x \mid Y = 2]$, which is a distribution of the single variable X, not the joint of X and Y, we need to marginalize p_3 and find, as desired:

$$\sum_y p_3(x, y) = \frac{\sum_y p_2(x, y)}{\sum_{x,y} p_2(x, y)} = \frac{p_1(x, 2)}{\sum_x p_1(x, 2)} = \frac{\mathbb{P}[X = x, Y = 2]}{\mathbb{P}[Y = 2]} = \mathbb{P}[X = x \mid Y = 2]$$

Table 4.1: GFs for common distributions

(a) GFs for common distributions with constant parameters

Distribution D	$\mathrm{gf}(D)(x)$
Dirac(a)	x^a
Bernoulli(p)	$px + 1 - p$
Categorical$(p_0, \ldots, p_n)$	$\sum_{i=0}^{n} p_i x^i$
Binomial(n, p)	$(px + 1 - p)^n$
Uniform$\{a..b\}$	$\frac{1}{b-a+1} \sum_{i=a}^{b} x^i$
NegBinomial(r, p)	$\left(\frac{p}{1-(1-p)x}\right)^r$
Geometric(p)	$\frac{p}{1-(1-p)x}$
Poisson(λ)	$e^{\lambda(x-1)}$
Exponential(λ)	$\frac{\lambda}{\lambda - \log x}$
Gamma(α, β)	$\left(\frac{\beta}{\beta - \log x}\right)^\alpha$
Uniform$[a, b]$	$\begin{cases} \frac{x^b - x^a}{(b-a)\log x} & x \neq 1 \\ 1 & x = 1 \end{cases}$

(b) GFs for common compound distributions

Distribution $D(Y)$	$\mathrm{gf}(D(Y))(x)$
Binomial(Y, p)	$\mathrm{gf}(Y)(1 - p + px)$
NegBinomial(Y, p)	$\mathrm{gf}(Y)\left(\frac{p}{1-(1-p)x}\right)$
Poisson$(\lambda \cdot Y)$	$\mathrm{gf}(Y)(e^{\lambda(x-1)})$
Bernoulli(Y)	$1 + (x - 1) \cdot (\mathrm{gf}(Y))'(1)$

Consider Example 4.1.1. Even though the example is an elementary exercise in probability, it is challenging to compute the normalizing constant, because one needs to evaluate an infinite sum:

$$\mathbb{P}[Y = 2] = \sum_{x \in \mathbb{N}} \mathbb{P}[Y = 2 \mid X = x]\ \mathbb{P}[X = x] = \sum_{x \in \mathbb{N}} \binom{x}{2} 0.1^2\, 0.9^{x-2} \cdot \frac{20^x e^{-20}}{x!}$$

It turns out to be $2e^{-2}$ (see Example 4.2.1), but it is unclear how to arrive at this result in an automated way. If X had been a continuous variable, we would even have to evaluate an integral. We will present a technique to compute such posteriors mechanically. It relies on probability generating functions, which often admit a closed form despite their definition including an infinite sum or integral, thus enabling the exact computation of posteriors.

4.2 Generating Functions

Definition Probability generating functions are a well-known tool in probability theory to study random variables and their distributions, especially discrete ones. The probability generating function of a random variable X is defined to be the function $G(x) := \mathbb{E}[x^X]$. For a discrete random variable supported on $\mathbb{N}$, this can also be written as a ***power series*** $G(x) = \sum_{n \in \mathbb{N}} \mathbb{P}[X = n] \cdot x^n$. Since we often deal with subprobability distributions, we omit "probability" from the name and refer to $G(x) := \mathbb{E}_{X \sim \mu}[x^X]$, where μ is a subprobability measure, simply as a ***generating function (GF)***. For continuous variables, it is often called ***factorial moment generating function***, but we stick with the former name in all contexts. We will use the notation $\mathrm{gf}(\mu)$ or $\mathrm{gf}(X)$ for the GF of μ or X. Note that for discrete random variables supported on $\mathbb{N}$, the GF is always defined on $[-1, 1]$, and for continuous ones at $x = 1$, but it need not be defined at other points x.

Probability masses and moments Many common distributions admit a ***closed form*** for their GF (see Table 4.1). In such cases, GFs are a compact representation of a distribution, even if it has infinite or continuous support like the Poisson or Exponential distributions, respectively. Crucially, one can extract *probability masses* and *moments* of a distribution from its generating function. For discrete random variables X, $P[X = n]$ is the n-th coefficient in the power series representation of G, so can be computed as the ***Taylor coefficient*** $\frac{1}{n!}G^{(n)}(0)$ at 0, hence the name "*probability* generating function". For a discrete or continuous random variable X, its expected value is $\mathbb{E}[X] = G'(1)$, and more generally, its n-th ***factorial moment*** is

$$\mathbb{E}[X(X-1)\cdots(X-n+1)] = \mathbb{E}\left[\frac{\mathrm{d}^n}{\mathrm{d}x^n}x^X\right]\Bigg|_{x=1} = G^{(n)}(1)$$

hence the name "*factorial moment* generating function". The raw and central moments can easily be computed from the factorial moments. For instance, the variance is $\mathbb{V}[X] = G''(1) + G'(1) - G'(1)^2$. We will exploit these properties of generating functions through automatic differentiation.

Multivariate case The definition of GFs is extended to *multidimensional distributions* in a straightforward way: for random variables $\boldsymbol{X} = (X_1, \ldots, X_n)$, their GF is the function $G(\boldsymbol{x}) := \mathbb{E}[\boldsymbol{x}^{\boldsymbol{X}}]$ where we write $\boldsymbol{x} := (x_1, \ldots, x_n)$ and $\boldsymbol{x}^{\boldsymbol{X}} := x_1^{X_1} \cdots x_n^{X_n}$. We generally follow the convention of using uppercase letters for random and program variables, and lowercase letters for the corresponding parameters of the generating function.

Marginalization can also be expressed in terms of generating functions: to obtain the GF $\tilde{G}$ of the joint distribution of $(X_1, \ldots, X_{n-1})$, i.e. to marginalize out X_n, one simply substitutes 1 for x_n in G: $\tilde{G}(x_1, \ldots, x_{n-1}) = G(x_1, \ldots, x_{n-1}, 1)$. This allows us to compute probability masses and moments of a random variable in a joint distribution: we marginalize out all the other variables and then use the previous properties of the derivatives.

4.2.1 Generating Function Semantics

Standard semantics In Section 2.3.1, we have defined the distribution transformer semantics of SGCL, where each program statement transforms the *joint distribution* of all the variables $\boldsymbol{X} = (X_1, \ldots, X_n)$. We introduced the *standard semantics* of a program P as $[\![P]\!]_{\lightning} : \mathrm{Meas}(\mathbb{S}_{\lightning}) \to \mathrm{Meas}(\mathbb{S}_{\lightning})$ on the extended state space $\mathbb{S}_{\lightning}$. However, since we only consider *loop-free* programs in this section, there is no need to distinguish between observation failures and nontermination. In this chapter, we will therefore work with the *simplified standard semantics* $[\![P]\!] : \mathrm{Meas}(\mathbb{S}) \to \mathrm{Meas}(\mathbb{S})$ on $\mathbb{S} := \mathbb{R}^n_{\geq 0}$ and refer to it as the *standard semantics* for brevity.

As discussed in Section 2.3.1, if the initial distribution μ is a probability distribution then the *normalized* posterior distribution is $\mathrm{normalize}([\![P]\!](\mu))$ where $\mathrm{normalize}(\mu) := \frac{\mu}{\mu(\mathbb{S})}$. This reduces the Bayesian inference problem for a given program to computing its semantics, starting from the joint distribution $\mathrm{Dirac}(\mathbf{0}_n)$ in which all n variables are initialized to 0 with probability 1. While mathematically useful, the distribution transformer semantics is hardly amenable to computation as it involves integrals and infinite sums.

Table 4.2: Generating function semantics for events

Event E	$G\|_E(\boldsymbol{x})$
$X_k = m$	$\frac{\partial_k^m G(\mathbf{x}[k \mapsto 0])}{m!} x_k^m$
flip(ρ)	$\rho \cdot G(\boldsymbol{x})$
$\neg E_1$	$G(\boldsymbol{x}) - G\|_{E_1}(\boldsymbol{x})$
$E_1 \wedge E_2$	$(G\|_{E_1})\|_{E_2}(\boldsymbol{x})$

Generating function semantics Instead, we compute the *generating function* of the posterior distribution represented by the probabilistic program. Then we can extract posterior probability masses and moments using automatic differentiation. Each statement in the programming language transforms the generating function of the distribution of program states, i.e. the joint distribution of the values of the variables $\boldsymbol{X} = (X_1, \ldots, X_n)$. Initially, we start with the constant function $G = \mathbf{1}$, which corresponds to all variables being initialized with 0 since $\mathbb{E}[x^0] = 1$.

The ***generating function semantics*** of a program $[\![P]\!]_{\text{gf}}$ describes how to transform G to the generating function $[\![P]\!]_{\text{gf}}(G)$ for the distribution at the end. It is defined in Table 4.2 for events and in Table 4.3 for statements. The update notation $\boldsymbol{x}[i \mapsto a]$ denotes $(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n)$ and $\mathbf{1}_n$ means the n-tuple $(1, \ldots, 1)$. The correctness of the semantics is stated and proven in Section 4.2.2.

Event semantics The restriction $G|_E$ of a GF G to an event E can be represented exactly as in the standard semantics (Table 4.2). The restrictions to the events flip(ρ), $\neg E_1$, and $E_1 \wedge E_2$ are defined analogously to the standard semantics. The interesting case is the $X_k = m$ event, whose GF semantics is already given in Chen et al. [2022]. To get intuition for it, remember that the GF G of discrete random variables X_1, X_2 with distribution μ on $\mathbb{N}^2$ is $G(x_1, x_2) = \sum_{X_1, X_2 \in \mathbb{N}} \mu(X_1, X_2) \cdot x_1^{X_1} x_2^{X_2}$. If we want to obtain the generating function restricted to $X_1 = m$, we want to keep only terms where x_1 has exponent m. In other words, we need the Taylor coefficient $\frac{\partial_1^m G(0, x_2)}{m!}$.

Statement semantics We explain the semantics of each statement (Table 4.3) in detail. The skip command leaves the distribution and hence the generating function unchanged. Chaining commands $P_1; P_2$ is again achieved by composing the individual transformers for each command. To obtain some intuition for the affine transformation case $X_k := \mathbf{a}^\top \boldsymbol{X} + c$, suppose we only have 2 variables: $X_1 := 2X_1 + 3X_2 + 5$. The idea is that the transformed generating function is given by

$$\begin{aligned}
[\![X_1 := 2X_1 + 3X_2 + 5]\!]_{\text{gf}}(G)(\boldsymbol{x}) &= \mathbb{E}[x_1^{2X_1 + 3X_2 + 5} x_2^{X_2}] \\
&= x_1^5 \cdot \mathbb{E}[(x_1^2)^{X_1} (x_1^3 x_2)^{X_2}] \\
&= x_1^5 \cdot G(x_1^2, x_1^3 x_2)
\end{aligned}$$

For the decrement statement $X_k \dot{-}= 1$, we again consider the case of two variables.

$$[\![X_1 \dot{-}= 1]\!]_{\text{gf}}(G)(x_1, x_2) = \mathbb{E}[[X_1 \geq 1] \cdot x_1^{X_1 - 1} x_2^{X_2} + [X_1 = 0] \cdot x_1^{X_1} x_2^{X_2}]$$

Table 4.3: Generating function semantics for statements

Statement P	$[\![P]\!]_{\mathrm{gf}}(G)(\boldsymbol{x})$
skip	$G(\boldsymbol{x})$
$P_1; P_2$	$[\![P_2]\!]_{\mathrm{gf}}([\![P_1]\!]_{\mathrm{gf}}(G))(\boldsymbol{x})$
$X_k := \mathbf{a}^\top \boldsymbol{X} + c$	$x_k^c \cdot G(\boldsymbol{x}')$ where $x_k' := x_k^{a_k}$ and $x_i' := x_i x_k^{a_i}$ for $i \neq k$
$X_k \doteq 1$	$\begin{cases} \dfrac{G(\boldsymbol{x}) - G(\boldsymbol{x}[k \mapsto 0])}{x_k} + G(\boldsymbol{x}[k \mapsto 0]) & \text{if } x_k \neq 0 \\ \partial_k G(\boldsymbol{x}[k \mapsto 0]) + G(\boldsymbol{x}[k \mapsto 0]) & \text{if } x_k = 0 \end{cases}$
if E $\{P_1\}$ else $\{P_2\}$	$[\![P_1]\!]_{\mathrm{gf}}(G\vert_E) + [\![P_2]\!]_{\mathrm{gf}}(G\vert_{\neg E})$
$X_k \sim D$	$G(\mathbf{x}[k \mapsto 1]) \cdot \mathrm{gf}(D)(x_k)$
$X_k \sim D(X_j)$	$G(\mathbf{x}[k \mapsto 1, j \mapsto x_j \cdot \mathrm{gf}(D(1))(x_k)])$ for $D \in \{$ Binomial$(-, p)$, NegBinomial$(-, p)$, Poisson$(\lambda \cdot -)$ $\}$ and $G(\boldsymbol{x}[k \mapsto 1]) + x_j(x_k - 1) \cdot \partial_j G(\boldsymbol{x}[k \mapsto 1])$ for $D =$ Bernoulli$(-)$, assuming X_j has support in $[0, 1]$
fail	0
Normalization	$\mathrm{normalize}(G) := \frac{G}{G(1_n)}$

$$
\begin{aligned}
&= \frac{\mathbb{E}[x_1^{X_1} x_2^{X_2} - [X_1 = 0] \cdot x_1^{X_1 - 1} x_2^{X_2}]}{x_1} + G|_{X_1 = 0}(x_1, x_2) \\
&= \frac{G(x_1, x_2) - G|_{X_1 = 0}(x_1, x_2)}{x_1} + G|_{X_1 = 0}(x_1, x_2) \\
&= \frac{G(x_1, x_2) - G(0, x_2)}{x_1} + G(0, x_2)
\end{aligned}
$$

For $x_1 = 0$, the fraction is not defined, but taking the limit as $x_1 \to 0$ gives the desired result via L'Hôpital's rule.

For a conditional if E $\{P_1\}$ else $\{P_2\}$, we first restrict the GF G to E and apply $[\![P_1]\!]_{\mathrm{gf}}$ for the then-branch, then restrict to $\neg E$ and apply $[\![P_2]\!]_{\mathrm{gf}}$ for the else-branch, and finally sum the two results.

For sampling, say $X_1 \sim D$, we first marginalize out the old X_1 by substituting 1 for x_1 in G: $\mathbb{E}[x_2^{X_2}] = \mathbb{E}[1^{X_1} x_2^{X_2}] = G(1, x_2)$. Then we multiply with the GF of the new distribution D.

The rules for compound distributions $D(X_j)$ are more involved, so we offer the following intuition. For the distributions $D(n) =$ Binomial(n, p), NegBinomial(n, p), Poisson$(\lambda \cdot n)$, we make essential use of the property $\mathrm{gf}(D(n))(x) = (\mathrm{gf}(D(1))(x))^n$. So if $G(x) := \mathbb{E}[x^X]$ is the GF of X and $Y \sim D(X)$, then

$$
\begin{aligned}
\mathbb{E}_{X,Y}[x^X y^Y] &= \mathbb{E}_X[x^X \mathbb{E}_{Y \sim D(X)}[y^Y]] \\
&= \mathbb{E}_X[x^X \, \mathrm{gf}(D(X))(y)] \\
&= \mathbb{E}_X[x^X (\mathrm{gf}(D(1))(y))^X] \\
&= \mathbb{E}[(x \cdot \mathrm{gf}(D(1))(y))^X] \\
&= G(x \cdot \mathrm{gf}(D(1))(y))
\end{aligned}
$$

This is not a fully formal argument, but it should give some intuition for the rule. For $D(p) =$ Bernoulli(p), the intuition is that if $G(x) := \mathbb{E}[x^X]$ is the GF of X and

if $Y \sim \text{Bernoulli}(X)$, then

$$\begin{aligned} H(x,y) &= \mathbb{E}[x^X y^Y] \\ &= \mathbb{E}[x^X((1-X)y^0 + Xy^1)] \\ &= \mathbb{E}[x^X - (y-1)Xx^X] \\ &= G(x) - (y-1)xG'(x) \end{aligned}$$

The GF semantics of fail is the zero function because its distribution is zero everywhere. Just like the subprobability distribution defined by a program has to be normalized as a last step, we have to normalize the generating function. The normalizing constant is calculated by marginalizing out all variables: $G(1,\ldots,1)$. So we obtain the generating function representing the normalized posterior distribution by rescaling with the inverse:

$$\text{normalize}_{\text{gf}}(G) := \frac{G}{G(\mathbf{1}_n)}$$

Novelty The first six rules in Table 4.3 were essentially presented in Klinkenberg et al. [2020] and Chen et al. [2022], though only for discrete distributions. The GF for sampling from $\text{Binomial}(X_j, p)$ was first given in Winner and Sheldon [2016]. Chen et al. [2022] also implicitly describe the GF semantics of sampling from $\text{Binomial}(X_j, p)$ and $\text{NegBinomial}(X_j, p)$ by expressing it as summing X_j i.i.d. samples from $\text{Bernoulli}(p)$ and $\text{Geometric}(p)$, respectively.

To our knowledge, the GF semantics of the compound distributions $\text{Poisson}(\lambda \cdot X_j)$ and $\text{Bernoulli}(X_j)$ is novel, and the former is required to support most models in Section 4.4. While the GF of observations has been considered in the context of a specific model [Winner and Sheldon 2016], this has not been done in the general context of a probabilistic programming language before. More generally, previous works involving GFs only considered discrete distributions, whereas we also allow sampling from continuous distributions. This is a major generalization and requires different proof techniques because the power series representation $\sum_{i\in\mathbb{N}} \mathbb{P}[X=i]x^i$, on which the proofs in Winner and Sheldon [2016], Klinkenberg et al. [2020], and Chen et al. [2022] rely, is not valid for continuous distributions.

Example 4.2.1 (GF translation)**.** Consider Example 4.1.1. We can find the posterior distribution mechanically by applying the rules from the GF semantics. We start with the GF $A(x,y) = \mathbb{E}[x^0y^0] = 1$ corresponding to X and Y being initialized to 0. Sampling X changes this to GF $B(x,y) = A(1,y)e^{20(x-1)} = e^{20(x-1)}$. Sampling Y yields $C(x,y) = B(x(0.1y+0.9),1) = e^{2x(y+9)-20}$. Observing $Y=2$ yields

$$D(x,y) = \frac{1}{2!}y^2\frac{\partial^2}{\partial y^2}C(x,0) = 2x^2y^2e^{18x-20}$$

To normalize, we divide by $D(1,1) = 2e^{-2}$, obtaining:

$$E(x,y) = \frac{D(x,y)}{D(1,1)} = x^2y^2e^{18(x-1)}$$

As described above, we can extract from this GF the posterior probability of, for example, exactly 10 individuals $\mathbb{P}[X = 10] = \frac{1}{10!}\frac{\partial^{10}}{\partial x^{10}}E(0,1) = 991796451840e^{-18}$ and the expected value of the posterior $\mathbb{E}[X] = \frac{\partial}{\partial x}E(1,1) = 20$.

4.2.2 Correctness

For the correctness theorem, we need to describe the set of $\boldsymbol{x} \in \mathbb{R}^n$ where the generating function is defined. For $R > 0$ and a measure μ on $\mathbb{R}^n_{\geq 0}$, define it to be the product of open intervals

$$T(R,\mu) := (Q_1, R) \times \cdots \times (Q_n, R)$$

where $Q_i = -R$ if the i-th component of μ is supported on $\mathbb{N}$ (i.e. $\mu(S) = \mu(\{\boldsymbol{X} \in S \mid X_i \in \mathbb{N}\})$ for any measurable S) and $Q_i = 0$ otherwise. The case split is necessary because the generating functions are better-behaved, and thus have a larger domain, if the measure is supported on $\mathbb{N}$. In fact, it is important that the GF be defined at $x_i = 0$ in the discrete case because otherwise we would not be able to extract probability masses from the GF.

Theorem 4.2.2 (Correctness of the GF event semantics)**.** *The GF event semantics (Table 4.2) is correct w.r.t. the standard event semantics (Table 2.2a): for any SGCL event E and subprobability distribution μ on $\mathbb{R}^n_{\geq 0}$, we have*

$$\mathrm{gf}(\mu|_E) = \mathrm{gf}(\mu)|_E$$

If $\mathrm{gf}(\mu)$ is defined on $T(R,\mu)$ for some $R > 1$ then $\mathrm{gf}(\mu|_E)$ is defined on $T(R,\mu)$ as well.

Proof. The proof is by induction on the structure of the event E. The cases $\mathsf{flip}(\rho)$, $\neg E_1$, and $E_1 \wedge E_2$ are defined analogously to the standard semantics, so the claim is straightforward using the induction hypothesis and linearity of $\mathrm{gf}(-)$:

$$\begin{aligned}
\mathrm{gf}(\mu|_{\mathsf{flip}(\rho)}) &= \mathrm{gf}(\rho \cdot \mu) = \rho \cdot \mathrm{gf}(\mu) = \mathrm{gf}(\mu)|_{\mathsf{flip}(\rho)} \\
\mathrm{gf}(\mu|_{\neg E_1}) &= \mathrm{gf}(\mu - \mu_{E_1}) = \mathrm{gf}(\mu) - \mathrm{gf}(\mu|_{E_1}) = \mathrm{gf}(\mu) - \mathrm{gf}(\mu)|_{E_1} = \mathrm{gf}(\mu)|_{\neg E_1} \\
\mathrm{gf}(\mu|_{E_1 \wedge E_2}) &= \mathrm{gf}((\mu|_{E_1})|_{E_2}) = \mathrm{gf}(\mu|_{E_1})|_{E_2} = (\mathrm{gf}(\mu)|_{E_1})|_{E_2} = \mathrm{gf}(\mu)|_{E_1 \wedge E_2}
\end{aligned}$$

For all these cases, it is clear that $\mathrm{gf}(\mu|_E)$ is defined on $T(R,\mu)$, using the induction hypothesis.

The remaining case $X_k = i$ is the most interesting:

$$\begin{aligned}
\mathrm{gf}(\mu|_{X_k=i})(\boldsymbol{x}) &= \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu|_{X_k=i}(\boldsymbol{X}) \\
&= \int x_1^{X_1} \cdots x_{k-1}^{X_{k-1}} \cdot x_k^{X_k} \cdot x_{k+1}^{X_{k+1}} \cdots x_n^{X_n} \cdot [X_k = i] \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= x_k^i \int x_1^{X_1} \cdots x_{k-1}^{X_{k-1}} \cdot x_{k+1}^{X_{k+1}} \cdots x_n^{X_n} \cdot \left(\frac{1}{i!} \left. \frac{\partial^i x_k^{X_k}}{\partial x_k^i} \right|_{x_k=0} \right) \mathrm{d}\mu(\boldsymbol{X})
\end{aligned}$$

The reason for the last step is that for the function $f(x) = x^l$, we have

$$f^{(i)}(x) = \begin{cases} 0 & \text{if } l < i \\ x^{l-i} \cdot \prod_{j=l-i+1}^{l} j & \text{if } l \geq i \end{cases}$$

Since 0^{l-i} is 0 for $l > i$ and 1 for $l = i$, we find that $f^{(i)}(0) = [i = l] \cdot i!$. Above, we used this fact with $l = X_k$ and $x = x_k$. We continue:

$$\begin{aligned}
\cdots &= x_k^i \int \frac{1}{i!} \left(x_1^{X_1} \cdots x_{k-1}^{X_{k-1}} \cdot \frac{\partial^i x_k^{X_k}}{\partial x_k^i} \cdot x_{k+1}^{X_{k+1}} \cdots x_n^{X_n} \right) \Bigg|_{x_k=0} \mathrm{d}\mu(\boldsymbol{X}) \\
&= \int \frac{\partial^i}{\partial x_k^i} \boldsymbol{x}^{\boldsymbol{X}} \, \mathrm{d}\mu(\boldsymbol{X}) \Bigg|_{x_k=0} \cdot \frac{x_k^i}{i!} \\
&= \frac{\partial^i}{\partial x_k^i} \int \boldsymbol{x}^{\boldsymbol{X}} \, \mathrm{d}\mu(\boldsymbol{X}) \Bigg|_{x_k=0} \cdot \frac{x_k^i}{i!} \\
&= \partial_k^i G(\boldsymbol{x}[k \mapsto 0]) \cdot \frac{x_k^i}{i!}
\end{aligned}$$

where Lemma 4.2.5 ensures that the integral and differentiation operators can be interchanged and that the GF $\mathrm{gf}(\mu|_{X_k=i})$ is defined on $T(R, \mu)$. □

Theorem 4.2.3 (Correctness of the GF statement semantics)**.** *The GF statement semantics (Table 4.3) is correct w.r.t. the standard semantics $[\![-]\!]$ (Table 2.2b): for any SGCL program P and subprobability distribution μ on $\mathbb{R}^n_{\geq 0}$, we have*

$$[\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)) = \mathrm{gf}([\![P]\!](\mu)) \quad \textit{and} \quad \mathrm{normalize}_{\mathrm{gf}}(\mathrm{gf}(\mu)) = \mathrm{gf}(\mathrm{normalize}(\mu))$$

In particular, it correctly computes the GF of the posterior distribution of P where all variables are initialized with 0:

$$\mathrm{normalize}_{\mathrm{gf}}([\![P]\!]_{\mathrm{gf}}(\mathbf{1})) = \mathrm{gf}(\mathrm{normalize}([\![P]\!](\mathrm{Dirac}(\mathbf{0}_n))))$$

If $\mathrm{gf}(\mu)$ is defined on $T(R, \mu)$ for some $R > 1$ then $[\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))$ is defined on $T(R', [\![P]\!](\mu))$ for some $R' > 1$. In particular, the GFs $[\![P]\!]_{\mathrm{gf}}(\mathbf{1})$ and thus $\mathrm{normalize}([\![P]\!]_{\mathrm{gf}}(\mathbf{1}))$ are defined on $T(R, [\![P]\!](\mu))$ for some $R > 1$.

Proof. As usual for such statements, we prove this by induction on the structure of the program. The program P can take one of the following forms:

Skip. For the trivial program, we have $[\![P]\!]_{\mathrm{gf}}(G) = G$ and $[\![P]\!](\mu) = \mu$, so the claim is trivial.

Chaining. If P is $P_1; P_2$, we know by the inductive hypothesis that $[\![P_1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)) = \mathrm{gf}([\![P_1]\!](\mu))$ and similarly for P_2. Taking this together, we find by the definitions of the semantics:

$$\begin{aligned}
[\![P_1; P_2]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)) &= [\![P_2]\!]_{\mathrm{gf}}([\![P_1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))) = [\![P_2]\!]_{\mathrm{gf}}(\mathrm{gf}([\![P_1]\!](\mu))) \\
&= \mathrm{gf}([\![P_2]\!]([\![P_1]\!](\mu))) = \mathrm{gf}([\![P_1; P_2]\!](\mu))
\end{aligned}$$

The claim about the domain of $[\![P]\!]_{\mathrm{gf}}(G)$ follows directly from the inductive hypotheses.

Affine assignments. If P is $X_k := \mathbf{a}^\top \boldsymbol{X} + c$, we have $[\![P]\!](\mu)(S) = \mu(\{\boldsymbol{X} \in \mathbb{R}^n \mid \boldsymbol{X}[k \mapsto \mathbf{a}^\top \boldsymbol{X} + c] \in S\})$ and thus

$$\mathrm{gf}([\![P]\!](\mu))(\boldsymbol{x}) = \int \boldsymbol{x}^{\boldsymbol{X}} \, \mathrm{d}([\![P]\!](\mu))(\boldsymbol{X})$$

$$
\begin{aligned}
&= \int x_1^{X_1} \cdots x_{k-1}^{X_{k-1}} x_k^{a_1 X_1 + \cdots + a_n X_n + c} x_{k+1}^{X_{k+1}} \cdots x_n^{X_n} \,\mathrm{d}\mu(X_1, \ldots, X_n) \\
&= x_k^c \cdot \int (x_1 x_k^{a_1})^{X_1} \cdots (x_{k-1} x_k^{a_{k-1}})^{X_{k-1}} (x_k^{a_k})^{X_k} \cdot \\
&\qquad\qquad (x_{k+1} x_k^{a_{k+1}})^{X_{k+1}} \cdots (x_n x_k^{a_n})^{X_n} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= x_k^c \cdot \mathrm{gf}(\mu)(x_1 x_k^{a_1}, \ldots, x_{k-1} x_k^{a_{k-1}}, x_k^{a_k}, x_{k+1} x_k^{a_{k+1}}, \ldots, x_n x_k^{a_n}) \\
&= [\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))(\boldsymbol{x})
\end{aligned}
$$

The claim that it is defined on some $T(R', [\![P]\!](\mu))$ holds if we choose $R' = \min(\sqrt{R}, \min_{i=1}^n R^{1/(2a_i)}) > 1$ because then $|x_i x_k^{a_i}| < \sqrt{R}(R^{1/(2a_1)})^{a_i} \leq R$ for $i \neq k$ and $|x_k^{a_k}| < (R^{1/(2a_1)})^{a_k} \leq \sqrt{R} \leq R$.

Decrement. If P is $X_k \mathrel{\dot{-}}= 1$, then X_k must be supported on $\mathbb{N}$ and we find:

$$
\begin{aligned}
\mathrm{gf}([\![P]\!](\mu))(\boldsymbol{x}) &= \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}([\![P]\!](\mu))(\boldsymbol{X}) \\
&= \int \boldsymbol{x}^{\boldsymbol{X}[k \mapsto X_k \dot{-} 1]} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int [X_k \geq 1] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto X_k \dot{-} 1]} \,\mathrm{d}\mu(\boldsymbol{X}) + \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto X_k \dot{-} 1]} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int [X_k \geq 1] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto X_k - 1]} \,\mathrm{d}\mu(\boldsymbol{X}) + \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto 0]} \,\mathrm{d}\mu(\boldsymbol{X})
\end{aligned}
$$

We have to distinguish two cases: $x_k = 0$ and $x_k \neq 0$. If $x_k = 0$, the first integrand vanishes unless $X_k = 1$, so we get:

$$
\begin{aligned}
\cdots &= \int [X_k = 1] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto 0]} \,\mathrm{d}\mu(\boldsymbol{X}) + \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}[k \mapsto 0]} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int \boldsymbol{x}^{\boldsymbol{X}[k \mapsto 0]} \,\mathrm{d}\mu|_{X_k = 1}(\boldsymbol{X}) + \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \mathrm{gf}([\![X_k := 0]\!](\mu|_{X_k=1}))(\boldsymbol{x}) + \mathrm{gf}(\mu|_{X_k=0})(\boldsymbol{x}) \\
&= [\![X_k := 0]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)|_{X_k=1})(\boldsymbol{x}) + \mathrm{gf}(\mu)|_{X_k=0}(\boldsymbol{x}) \\
&= \mathrm{gf}(\mu)|_{X_k=1}(\boldsymbol{x}[k \mapsto 1]) + \mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 0]) \\
&= \partial_k \,\mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 0]) + \mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 0]) \\
&= [\![X_k \mathrel{\dot{-}}= 1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))(\boldsymbol{x})
\end{aligned}
$$

by the semantics of $X_k := 0$ and of the event $X_k = 1$. If $x_k \neq 0$, we use the fact that $[X_k \geq 1] = 1 - [X_k = 0]$ since X_k is supported on $\mathbb{N}$. We reason as follows:

$$
\begin{aligned}
\cdots &= \frac{\int (1 - [X_k = 0]) \cdot \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X})}{x_k} + \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \frac{\int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) - \int [X_k = 0] \cdot \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X})}{x_k} + \mathrm{gf}(\mu|_{X_k=0})(\boldsymbol{x}) \\
&= \frac{\mathrm{gf}(\mu)(\boldsymbol{x}) - \mathrm{gf}(\mu|_{X_k=0})(\boldsymbol{x})}{x_k} + \mathrm{gf}(\mu)|_{X_k=0}(\boldsymbol{x})
\end{aligned}
$$

$$= \frac{\mathrm{gf}(\mu)(\boldsymbol{x}) - \mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 0])}{x_k} + \mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 0])$$
$$= [\![X_k \doteq 1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))(\boldsymbol{x})$$

Conditionals. If P is if E $\{P_1\}$ else $\{P_2\}$, we use the linearity of $\mathrm{gf}(-)$, the induction hypothesis, and Theorem 4.2.2 to find:

$$\begin{aligned}\mathrm{gf}([\![P]\!](\mu)) &= \mathrm{gf}([\![P_1]\!](\mu|_E)) + \mathrm{gf}([\![P_2]\!](\mu|_{\neg E}))\\ &= [\![P_1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu|_E)) + [\![P_2]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu|_{\neg E}))\\ &= [\![P_1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)|_E) + [\![P_2]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)|_{\neg E})\\ &= [\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))\end{aligned}$$

Again by Theorem 4.2.2, we know that $\mathrm{gf}(\mu)|_E$ and $\mathrm{gf}(\mu)|_{\neg E}$ are defined on $T(R, \mu)$, so by the induction hypothesis, both $[\![P_1]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)|_E)$ and $[\![P_2]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu)|_{\neg E})$ are defined on some $T(R'', [\![P_1]\!](\mu|_E))$ and $T(R''', [\![P_2]\!](\mu|_{\neg E}))$, respectively. Hence $[\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))$ is defined on $T(R', [\![P]\!](\mu))$ for $R' := \min(R'', R''') > 1$.

Sampling. If P is $X_k \sim D$, we find:

$$\begin{aligned}\mathrm{gf}([\![P]\!](\mu))(\boldsymbol{x}) &= \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}([\![P]\!](\mu))(\boldsymbol{X})\\ &= \int\int \boldsymbol{x}^{\boldsymbol{X}[k\mapsto Y]} \,\mathrm{d}D(Y)\,\mathrm{d}\mu(\boldsymbol{X})\\ &= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \int x_k^Y \,\mathrm{d}D(Y)\,\mathrm{d}\mu(\boldsymbol{X})\\ &= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \cdot \int x_k^Y \,\mathrm{d}D(Y)\\ &= \mathrm{gf}(\mu)(\boldsymbol{x}[k \mapsto 1])\,\mathrm{gf}(D)(x_k)\\ &= [\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))(\boldsymbol{x})\end{aligned}$$

Regarding the domain of $[\![P]\!]_{\mathrm{gf}}(\mathrm{gf}(\mu))$, note that $\mathrm{gf}(D)$ is defined on all of $\mathbb{R}$ for any finite distribution and for the Poisson distribution (cf. Table 4.1a). For $D \in \{\mathsf{Geometric}(p), \mathsf{NegBinomial}(n, p)\}$, the GF $\mathrm{gf}(D)$ is defined on $(-\frac{1}{1-p}, \frac{1}{1-p})$, so we can pick $R' := \min(R, \frac{1}{1-p}) > 1$. The GF of $\mathsf{Exponential}(\lambda)$ is defined for $\log x < \lambda$, i.e. $x < \exp(\lambda)$, so we can pick $R' := \min(R, e^\lambda) > 1$. The GF of $\mathsf{Gamma}(\alpha, \beta)$ is defined for $\log x < \beta$, i.e. $x < \exp(\beta)$, so we can pick $R' := \min(R, e^\beta) > 1$. The GF of $\mathsf{Uniform}[a, b]$ is defined on $\mathbb{R}$, so $R' := R$ works.

Compound distributions. If P is $X_k \sim D(X_j)$, we use the fact that $\mathrm{gf}(D(n))(\boldsymbol{x}) = (\mathrm{gf}(D(1))(\boldsymbol{x}))^n$ for $D(n) \in \{\mathsf{Binomial}(n, p), \mathsf{NegBinomial}(n, p), \mathsf{Poisson}(\lambda \cdot n)\}$, which can easily be checked by looking at their generating functions (cf. Table 4.1b).

$$\begin{aligned}\mathrm{gf}([\![P]\!](\mu))(\boldsymbol{x}) &= \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}([\![P]\!](\mu))(\boldsymbol{X})\\ &= \int\int \boldsymbol{x}^{\boldsymbol{X}[k\mapsto Y]} \,\mathrm{d}D(X_j)(Y)\,\mathrm{d}\mu(\boldsymbol{X})\\ &= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \int x_k^Y \,\mathrm{d}D(X_j)(Y)\,\mathrm{d}\mu(\boldsymbol{X})\end{aligned}$$

$$
\begin{aligned}
&= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \operatorname{gf}(D(X_j))(x_k) \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} (\operatorname{gf}(D(1))(x_k))^{X_j} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int (\boldsymbol{x}[k \mapsto 1, j \mapsto x_j \cdot \operatorname{gf}(D(1))(x_k)])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \operatorname{gf}(\mu)(\boldsymbol{x}[k \mapsto 1, j \mapsto x_j \cdot \operatorname{gf}(D(1))(x_k)]) \\
&= [\![P]\!]_{\mathrm{gf}}(\operatorname{gf}(\mu))(\boldsymbol{x})
\end{aligned}
$$

Regarding the domain of $[\![P]\!]_{\mathrm{gf}}(\operatorname{gf}(\mu))$, we have to ensure that $|x_j \cdot \operatorname{gf}(D(1))(x_k)| < R$. For the Binomial$(1, p)$ distribution, we choose $R' = \sqrt{R} > 1$ such that $|x_j||1 - p + px_k| < \sqrt{R}(1 - p + p\sqrt{R}) \leq \sqrt{R}((1-p)\sqrt{R} + p\sqrt{R}) < R$. For the NegBinomial$(1)$ distribution, we choose $R' = \min(\sqrt{R}, \frac{\sqrt{R}-p}{\sqrt{R}(1-p)}) = \min(\sqrt{R}, 1 + \frac{p(\sqrt{R}-1)}{\sqrt{R}-p\sqrt{R}}) > 1$ because for any $x \in (-R', R)$, we have

$$
\left|\frac{p}{1-(1-p)x_j}\right| < \frac{p}{1-(1-p)R'} \leq \frac{p}{1-\frac{\sqrt{R}-p}{\sqrt{R}}} = \frac{p\sqrt{R}}{p} = \sqrt{R}
$$

and thus $\left|x_j \frac{p}{1-(1-p)x_k}\right| < R$. For the Poisson$(\lambda)$ distribution, we choose $R' = \min(\sqrt{R}, 1 + \frac{\log(R)}{2\lambda}) > 1$ because then $|x_j \cdot \exp(\lambda(x_k - 1))| < \sqrt{R} \exp(\lambda \frac{\log(R)}{2\lambda}) = R$.

For the Bernoulli distribution $D(X_j) = \text{Bernoulli}(X_j)$, we reason as follows:

$$
\begin{aligned}
\operatorname{gf}([\![P]\!](\mu))(\boldsymbol{x}) = \cdots &= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \operatorname{gf}(D(X_j))(x_k) \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} (1 + X_j(x_k - 1)) \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) + (x_k - 1) \cdot \int X_j (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \operatorname{gf}(\mu)(\boldsymbol{x}[k \mapsto 1]) + (x_k - 1) \cdot \int x_j \cdot \partial_j (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \operatorname{gf}(\mu)(\boldsymbol{x}[k \mapsto 1]) + x_j(x_k - 1) \cdot \partial_j \int (\boldsymbol{x}[k \mapsto 1])^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \operatorname{gf}(\mu)(\boldsymbol{x}[k \mapsto 1]) + x_j(x_k - 1) \cdot \partial_j \operatorname{gf}(\mu)(\boldsymbol{x}[k \mapsto 1]) \\
&= [\![P]\!]_{\mathrm{gf}}(\operatorname{gf}(\mu))(\boldsymbol{x})
\end{aligned}
$$

Note that the interchange of integral and differentiation is allowed by Lemma 4.2.5. It also implies that $[\![P]\!]_{\mathrm{gf}}(\operatorname{gf}(\mu))(\boldsymbol{x})$ is defined for $x_j \in (0, R)$ by Lemma 4.2.5.

Fail. If P is fail, then $[\![P]\!](\mu)$ is the zero measure and $[\![P]\!]_{\mathrm{gf}}(\operatorname{gf}(\mu))$ is the zero function, so the claim holds trivially. This GF is clearly defined everywhere.

Normalization. For normalization, we make use of the linearity of $\operatorname{gf}(-)$.

$$
\begin{aligned}
\operatorname{gf}(\text{normalize}(\mu)) &= \operatorname{gf}\left(\frac{\mu}{\int \mathrm{d}\mu(\boldsymbol{X})}\right) \\
&= \frac{\operatorname{gf}(\mu)}{\int \mathrm{d}\mu(\boldsymbol{X})}
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{\mathrm{gf}(\mu)}{\int 1^{X_1} \cdots 1^{X_n} \,\mathrm{d}\mu(\boldsymbol{X})} \\
&= \frac{\mathrm{gf}(\mu)}{\mathrm{gf}(\mu)(\mathbf{1}_n)} \\
&= \mathrm{normalize}_{\mathrm{gf}}(\mathrm{gf}(\mu))
\end{aligned}
$$

Furthermore, if $\mathrm{gf}(\mu)$ is defined on $T(R,\mu)$, then so is $\mathrm{normalize}(\mathrm{gf}(\mu))$ on $T(R,\mu) = T(R, \mathrm{normalize}(\mu))$. This finishes the induction and proves the claim. □

Remark 4.2.4. As an example of what can go wrong with derivatives at zero if the measure is not supported on $\mathbb{N}$, consider the Dirac measure $\mu = \mathrm{Dirac}(\frac{1}{2})$. It has the generating function $G(x) := \mathrm{gf}(\mu)(x) = \sqrt{x}$ defined for $x \in \mathbb{R}_{\geq 0}$. Its derivative is $G'(x) := \frac{1}{2\sqrt{x}}$, which is not defined for $x = 0$. This is why $T(R,\mu) = (0,R)$ does not include the point 0 for this measure.

The proofs rely on the following two technical lemmas that justify interchanging derivatives and integrals in the above situations.

Lemma 4.2.5. *If the integral $\int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X})$ for a measure μ on $\mathbb{R}^n_{\geq 0}$ is defined for all $\boldsymbol{x} \in T(R,\mu)$, we have*

$$\partial_{i_1} \cdots \partial_{i_m} \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) = \int \partial_{i_1} \cdots \partial_{i_m} \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X})$$

and both sides are defined for all $\boldsymbol{x} \in T(R,\mu)$.
Furthermore, the right-hand side has the form $\int p(\boldsymbol{X}) \boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} \,\mathrm{d}\mu(\boldsymbol{X})$ for a polynomial p in n variables and $\boldsymbol{w} \in \mathbb{N}^n$, with the property that $p(\boldsymbol{X}) = 0$ whenever $X_j < w_j$ for some j, in which case we define $p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} := 0$, even if $\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}$ is undefined.

Proof. The proof is by induction on m. If $m = 0$ then the statement is trivial. For the induction step $m > 0$, we may assume that

$$\partial_{i_2} \cdots \partial_{i_m} \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) = \int \partial_{i_2} \cdots \partial_{i_m} \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) = \int p(\boldsymbol{X}) \boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} \,\mathrm{d}\mu(\boldsymbol{X})$$

By Lemma 4.2.6, we reason

$$
\begin{aligned}
\partial_{i_1} \cdots \partial_{i_m} \int \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X}) &= \partial_{i_1} \int p(\boldsymbol{X}) \boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int (X_{i_1} - w_{i_1}) p(\boldsymbol{X}) \boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}[i_1 \mapsto w_{i_1}+1]} \,\mathrm{d}\mu(\boldsymbol{X}) \\
&= \int \partial_{i_1} \cdots \partial_{i_m} \boldsymbol{x}^{\boldsymbol{X}} \,\mathrm{d}\mu(\boldsymbol{X})
\end{aligned}
$$

which establishes the induction goal with the polynomial $\tilde{p}(\boldsymbol{X}) := (X_{i_1} - w_{i_1}) p(\boldsymbol{X})$ and $\tilde{\boldsymbol{w}} := w[i_1 \mapsto w_{i_1} + 1]$. The property $\tilde{p}(\boldsymbol{X}) = 0$ whenever $X_j < \tilde{w}_j$ for some j still holds due to the added factor $(X_{i_1} - w_{i_1})$ taking care of the case $X_{i_1} = w_{i_1} = \tilde{w}_{i_1} - 1$. □

Lemma 4.2.6. *Let $\boldsymbol{w} \in \mathbb{N}^n$, p be a polynomial in n variables, and μ a measure on $\mathbb{R}^n_{\geq 0}$. We define $p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} := 0$ whenever $p(\boldsymbol{X}) = 0$, even if $x_j \leq 0$ and $X_j < w_j$ for some j. Suppose*

$p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}$ is defined μ-almost everywhere and μ-integrable for all $\boldsymbol{x} \in T(R, \mu)$. Then

$$\begin{aligned}\frac{\partial}{\partial x_i}\int p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}\,\mathrm{d}\mu(\boldsymbol{X}) &= \int \frac{\partial}{\partial x_i} p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}\,\mathrm{d}\mu(\boldsymbol{X}) \\ &= \int (X_i - w_i)p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]}\,\mathrm{d}\mu(\boldsymbol{X})\end{aligned}$$

holds and is defined for all $\boldsymbol{x} \in T(R, \mu)$.

Proof. The proof is about verifying the conditions of the Leibniz integral rule. The nontrivial condition is the boundedness of the derivative by an integrable function. The proof splits into two parts, depending on whether the i-th component of μ is supported on $\mathbb{N}$. If it is, then $x_i \in (-R, R)$ by the definition of $T(R, \mu)$ and can be nonpositive, but $X_i - w_i < 0$ and $p(\boldsymbol{X}) \neq 0$ happens μ-almost never. Otherwise, we have $x \in (0, R)$ and it is thus guaranteed to be positive, but $X_i - w_i$ may be negative. Hence the two cases need slightly different treatment.

We first deal with the case that the i-th component is not supported on $\mathbb{N}$. Let $0 < \varepsilon < R' < R'' < R$. We first prove $\frac{\partial}{\partial x_i}\int p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}}\,\mathrm{d}\mu(\boldsymbol{X}) = \int \frac{\partial}{\partial x_i} p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}}\,\mathrm{d}\mu(\boldsymbol{X})$ for any $x_i \in (\varepsilon, R')$, using the Leibniz integral rule. For this purpose, we need to bound the derivative:

$$\begin{aligned}\left|\frac{\partial}{\partial x_i} p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}\right| &\leq |(X_i - w_i)p(\boldsymbol{X})|\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]} \leq \frac{1}{x_i}|X_i - w_i||p(\boldsymbol{X})|\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}} \\ &\leq \frac{1}{\varepsilon}|X_i - w_i||p(\boldsymbol{X})|\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}\end{aligned}$$

If $X_i > w_i$, we can choose $M > 1$ sufficiently large such that $|X_i - w_i|x_i^{X_i-w_i} \leq (X_i - w_i)R'^{X_i-w_i} \leq R''^{X_i-w_i}$ whenever $|X_i - w_i| > M$. (Such an M can be found because the exponential function $y \mapsto \left(\frac{R''}{R'}\right)^y$ grows more quickly than the linear function $y \mapsto y$.) If $X_i - w_i \leq M$ then we also have $|X_i - w_i|x_i^{X_i-w_i} \leq M \cdot R''^{X_i-w_i}$. So the derivative is thus bounded on the set $\{\boldsymbol{X} \mid X_i > w_i\}$ by $\frac{M}{\varepsilon} \cdot |p(\boldsymbol{X})|(\boldsymbol{x}[i \mapsto R''])^{\boldsymbol{X}-\boldsymbol{w}}$, which is integrable by assumption.

On the complement set $\{\boldsymbol{X} \mid 0 \leq X_i \leq w_i\}$, we find $|X_i - w_i|x_i^{X_i-w_i} \leq |X_i - w_i|\varepsilon^{X_i-w_i} \leq w_i\varepsilon^{X_i-w_i}$. Hence the derivative is bounded by $\frac{w_i}{\varepsilon} \cdot |p(\boldsymbol{X})|(\boldsymbol{x}[i \mapsto \varepsilon])^{\boldsymbol{X}-\boldsymbol{w}}$, which is again integrable by assumption. So the derivative is bounded by an integrable function and thus, by the Leibniz integral rule, interchanging differentiation and the integral is valid for all $x_i \in (\varepsilon, R')$ and thus for all $x_i \in (0, R)$.

Next, consider the case that the i-th component of μ is supported on $\mathbb{N}$. Let $0 < R' < R'' < R$. Note that the set $\{\boldsymbol{X} \mid X_i < w_i \wedge p(\boldsymbol{X}) \neq 0\}$ has measure zero because otherwise the integrand $p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}$ would not be defined for $x_i \leq 0$ due to the negative exponent $X_i - w_i$. Hence, for μ-almost all $\boldsymbol{X}$, we have $p(\boldsymbol{X}) = [\boldsymbol{X} \geq \boldsymbol{w}]p(\boldsymbol{X})$ and can bound the partial derivative for $|x_i| < R'$:

$$\begin{aligned}\left|\frac{\partial}{\partial x_i} p(\boldsymbol{X})\boldsymbol{x}^{\boldsymbol{X}-\boldsymbol{w}}\right| &\leq |(X_i - w_i)p(\boldsymbol{X})||\boldsymbol{x}|^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]} \\ &\leq [\boldsymbol{X} \geq \boldsymbol{w}]|X_i - w_i||p(\boldsymbol{X})||\boldsymbol{x}|^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]} \\ &\leq [\boldsymbol{X} \geq \boldsymbol{w}[i \mapsto w_i + 1]]|X_i - w_i||p(\boldsymbol{X})||\boldsymbol{x}|^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]}\end{aligned}$$

$$\leq [\boldsymbol{X} \geq \boldsymbol{w}[i \mapsto w_i + 1]]|X_i - w_i||p(\boldsymbol{X})||\boldsymbol{x}[i \mapsto R']|^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]}$$
$$\leq [\boldsymbol{X} \geq \boldsymbol{w}]|X_i - w_i||p(\boldsymbol{X})||\boldsymbol{x}[i \mapsto R']|^{\boldsymbol{X}-\boldsymbol{w}}$$

because the factor $|X_i - w_i|$ vanishes for $X_i = w_i$ and thus ensures that negative values for the exponent $\boldsymbol{X} - \boldsymbol{w}[i \mapsto w_i + 1]$ do not matter, so that we can use the monotonicity of $|\boldsymbol{x}|^{\boldsymbol{X}-\boldsymbol{w}[i\mapsto w_i+1]}$ in $|\boldsymbol{x}|$.

Similarly to the first case, for a sufficiently large $M > \max(1, w_i)$, we have $|X_i - w_i|R'^{X_i-w_i} \leq R''^{X_i-w_i}$ whenever $X_i - w_i > M$. If $X_i - w_i \leq M$ then $[\boldsymbol{X} \geq \boldsymbol{w}]|X_i - w_i| \leq M$ and we also have $[\boldsymbol{X} \geq \boldsymbol{w}]|X_i - w_i|R'^{X_i-w_i} \leq M \cdot R''^{X_i-w_i}$. The derivative is thus bounded by $M \cdot |p(\boldsymbol{X})||\boldsymbol{x}[i \mapsto R'']|^{\boldsymbol{X}-\boldsymbol{w}}$, which is integrable by assumption. By the Leibniz integral rule, interchanging differentiation and the integral is valid for all $x_i \in (-R', R')$ and thus for all $x_i \in (-R, R)$. □

4.2.3 Possible Language Extensions

The syntax of our language guarantees that the generating function of the distribution of any probabilistic program admits a closed form. But are the syntactic restrictions necessary or are there other useful programming constructs that can be supported? We believe the following constructs preserve closed forms of the generating function and could thus be supported:

- *Additional distributions with constant parameters:* any such distribution could be supported as long as its GF is defined on $[0, 1+\varepsilon)$ for some $\varepsilon > 0$ and its derivatives can be evaluated.
- *Additional compound distributions:* we spent a lot of time trying to find additional compound distributions with a closed-form GF since this would be the most useful for probabilistic models, but we were largely unsuccessful. The only such distributions we found are Binomial(m, X_k), which could just be written as a sum of m i.i.d. Bernoulli(X_k)-variables, and Gamma(X_k, β) with shape parameter X_k, which could be translated to a GF with the same idea as for Binomial(X_k, p), NegBinomial(X_k, p), and Poisson$(\lambda \cdot X_k)$.
- *Modulo, i.i.d. sums:* The event $X_k \mod 2 = 0$ and $X_k :=$ sum_iid(D, X_l) (i.e. X_k is assigned the sum of X_l independent random variables with distribution D) can also be supported, as shown in Chen et al. [2022] and Klinkenberg et al. [2024].
- *Nonlinear functions on variables with finite support*: if a variable has finite support, arbitrary functions on it could be performed by exhaustive case analysis of those finitely many values.
- *Soft conditioning:* another supportable construct could be score(a^{X_k}) for $a \in \mathbb{R}_{\geq 0}$, which multiplies the likelihood of the current path by a^{X_k} like scoring in SPCF (Section 2.2.3). For $a > 1$, this could lead to integrability issues, however, so a restriction to $a \leq 1$ may be necessary. We think score$(q(X_k))$ where $q(t) := \sum_{i=0}^{m} q_i t^i$ is a polynomial with $q \geq 0$ on the support of X_k could also be supportable using similar techniques as for observations from Bernoulli(X_k).

None of these extensions seemed very useful for real-world models, in particular, they are not needed for our benchmarks. For this reason we did not formalize, implement, or prove them.

Loops Support for loops is another big question. Bounded loops are easy enough to deal with by fully unrolling them. However exact inference for programs with unbounded loops seems extremely difficult, unless the inference algorithm is given additional information, e.g. a probabilistic loop invariant as in Chen et al. [2022] or at least a loop invariant template with only a few holes to be filled with real constants [Klinkenberg et al. 2024]. However, finding such a loop invariant for nontrivial programs is exceedingly difficult (see the discussion in Section 2.4). Furthermore, the variables of looping programs may have infinite expected values, so the generating function may not be definable at the point 1. But in our semantics, evaluating at 1 is very important for marginalization. For these reasons, we consider exact inference in the presence of loops an interesting but very hard research problem. In Chapter 5, we instead discuss an approach to derive guaranteed bounds on SGCL programs with loops, building on the exact inference results from this chapter.

4.3 Implementation and Optimizations

4.3.1 Automatic Differentiation and Taylor polynomials

The main difficulty in implementing the GF semantics is the computation of the partial derivatives. A natural approach (as followed by Chen et al. [2022] and Klinkenberg et al. [2024]) is to manipulate symbolic representations of the generating functions and to use computer algebra for the derivatives. However this usually scales badly, as demonstrated by Winner et al. [2017], because the size of the generating functions typically grows quickly with the data conditioned on. To see why, note that every observe $X_k = d$ statement in the program is translated to a d-th partial derivative. Since probabilistic programs tend to contain many data points, it is common for the total order of derivatives to be in the hundreds. The size of the symbolic representation of a function can (and typically does) grow exponentially in the order of the derivative: the derivative of the product of two functions $f \cdot g$ is the sum of two products $f' \cdot g + f \cdot g'$, so the representation doubles in size. Hence the running time would be $\Omega(2^d)$ where d is the sum of all observed values, which is clearly unacceptable.

Automatic differentiation Instead, we exploit the fact that we do not need to generate the full representation of a GF, but merely to *evaluate it and its derivatives*. ***Automatic differentation (AD)*** is a technique to evaluate derivatives of functions specified by a computation graph or computer program automatically (see Baydin et al. [2017] for an introduction and survey). It is very popular in machine learning, but has also seen applications in other fields such as optimization and physics. Our semantics seems to be a unique situation where we want to compute derivatives of very high order. We have not come across another application of AD requiring derivatives of order greater than, say, 4 or 5. As a consequence, existing AD frameworks are not designed for our use case of derivatives of order in the hundreds. In fact, we tried the popular AD framework PyTorch, but it performed poorly in our tests.

For this reason, we decided to implement our own AD framework. Winner et al. [2017] also use AD for the population model (with only one variable), and we extend

this to our more general setting with multiple variables. However, we work with ***Taylor polynomials*** instead of higher derivatives directly because the computation of the latter includes additional factorial factors, which can easily lead to overflows. Our strategy is to compute the Taylor expansion in all variables up to some order d of the generating function instead of a symbolic expression of the d-th derivative. This has the advantage of growing polynomially in d, not exponentially like symbolic derivatives. In this approach, derivatives are the easy part as they can be read off the Taylor coefficients, but the composition of Taylor polynomials is the bottleneck.

Taylor polynomials More formally, we define the ***multivariate Taylor polynomial*** $\text{Taylor}_{\boldsymbol{w}}^d(G)$ of a function $G : \mathbb{R}^n \to \mathbb{R}$ at $\boldsymbol{w} \in \mathbb{R}^n$ of order d as the polynomial

$$\text{Taylor}_{\boldsymbol{w}}^d(G) := \sum_{\boldsymbol{\alpha} \in \mathbb{N}^n : |\boldsymbol{\alpha}| \leq d} \frac{1}{\boldsymbol{\alpha}!} \partial^{\boldsymbol{\alpha}} G(\boldsymbol{w}) \cdot (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$$

where we used multi-index notation: $|\boldsymbol{\alpha}|$ stands for $\alpha_1 + \cdots + \alpha_n$; $\boldsymbol{\alpha}!$ for $\alpha_1! \cdots \alpha_n!$; and $\partial^{\boldsymbol{\alpha}}$ for $\partial_1^{\alpha_1} \ldots \partial_n^{\alpha_n}$. The coefficients $c_{\boldsymbol{\alpha}} := \frac{1}{\boldsymbol{\alpha}!} \partial^{\boldsymbol{\alpha}} G(\boldsymbol{w})$ are the *Taylor coefficients* of G at $\boldsymbol{w}$ and are stored in a multidimensional array in our implementation.

Operations The operations on generating functions (Section 4.2) are implemented in terms of their effect on the Taylor expansions. Let $G, H : \mathbb{R}^n \to \mathbb{R}$ be two functions with Taylor expansions $\text{Taylor}_{\boldsymbol{w}}^d(G) = \sum_{|\boldsymbol{\alpha}| \leq d} g_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$ and $\text{Taylor}_{\boldsymbol{w}}^d(H) = \sum_{|\boldsymbol{\alpha}| \leq d} h_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$. Addition $F = G + H$ of two generating functions is implemented by adding the coefficients:

$$\text{Taylor}_{\boldsymbol{w}}^d(G + H) = \sum_{|\boldsymbol{\alpha}| \leq d} (g_{\boldsymbol{\alpha}} + h_{\boldsymbol{\alpha}})(\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$$

Scalar multiplication $F = c \cdot G$ is implemented by multiplying the coefficients:

$$\text{Taylor}_{\boldsymbol{w}}^d(c \cdot G) = \sum_{|\boldsymbol{\alpha}| \leq d} (c \cdot g_{\boldsymbol{\alpha}})(\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$$

Multiplication $F = G \cdot H$ of two generating functions is implemented by the Cauchy product:

$$\text{Taylor}_{\boldsymbol{w}}^d(G \cdot H) = \sum_{|\boldsymbol{\alpha}| \leq d} \left(\sum_{\boldsymbol{\alpha}_1 + \boldsymbol{\alpha}_2 = \boldsymbol{\alpha}} g_{\boldsymbol{\alpha}_1} h_{\boldsymbol{\alpha}_2} \right) (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\alpha}}$$

Division, exponentiation, logarithms, and powers can be implemented as well (see Griewank and Walther [2008, Chapter 13] for details). Partial derivatives essentially correspond to shifting the index and a multiplication:

$$\text{Taylor}_{\boldsymbol{w}}^{d-1}(\partial_k G) = \sum_{|\boldsymbol{\alpha}| \leq d-1} (\alpha_k + 1) g_{\boldsymbol{\alpha}[k \mapsto \alpha_k + 1]} \boldsymbol{x}^{\boldsymbol{\alpha}}$$

The most complicated case is composition/substitution, i.e. $F(\boldsymbol{x}) = G(\boldsymbol{x}[k \mapsto H(\boldsymbol{x})])$. For this, we let $\boldsymbol{w}' := w[k \mapsto h_0]$ where $\text{Taylor}_{\boldsymbol{w}}^d(H) = \sum_{|\boldsymbol{\beta}| \leq d} h_{\boldsymbol{\beta}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\beta}}$ and we let $\text{Taylor}_{\boldsymbol{w}'}^d(G) = \sum_{|\boldsymbol{\alpha}| \leq d} g_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w}')^{\boldsymbol{\alpha}}$. Then we find, writing "h.o.t." for "higher order terms":

$$G(\boldsymbol{x}[k \mapsto H(\boldsymbol{x})]) = G\left(\boldsymbol{x}\left[k \mapsto \textstyle\sum_{|\boldsymbol{\beta}| \leq d} h_{\boldsymbol{\beta}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\beta}} + \text{h.o.t.}\right]\right)$$

$$= \sum_{|\boldsymbol{\alpha}| \leq d} g_{\boldsymbol{\alpha}} \left(\boldsymbol{x} \left[k \mapsto h_0 + \textstyle\sum_{1 \leq |\boldsymbol{\beta}| \leq d} h_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\beta}} + \text{h.o.t.} \right] - \boldsymbol{w}' \right)^{\boldsymbol{\alpha}} + \text{h.o.t.}$$
$$= \sum_{|\boldsymbol{\alpha}| \leq d} g_{\boldsymbol{\alpha}} \left((\boldsymbol{x} - \boldsymbol{w}) \left[k \mapsto \textstyle\sum_{1 \leq |\boldsymbol{\beta}| \leq d} h_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\beta}} \right] \right)^{\boldsymbol{\alpha}} + \text{h.o.t.}$$

because by definition $\boldsymbol{x}[k \mapsto h_0] - w' = (\boldsymbol{x} - \boldsymbol{w})[k \mapsto 0]$. This means that we can obtain $\text{Taylor}^d_{\boldsymbol{w}}(F)$ by substituting $\sum_{1 \leq |\boldsymbol{\beta}| \leq d} h_{\boldsymbol{\alpha}} (\boldsymbol{x} - \boldsymbol{w})^{\boldsymbol{\beta}}$ for $x_k - w'_k$ in $\text{Taylor}^d_{\boldsymbol{w}'}(G)$.

Example 4.3.1 (Calculations with Taylor polynomials)**.** Here we show how the expectation for Example 4.1.1 can be computed using Taylor polynomials. We use the same naming as in Example 4.2.1. Recall that to compute $\mathbb{E}[X]$, we need to evaluate $\partial_1 E(1,1)$, so we compute the Taylor expansion of E at $(1,1)$ of order 1, i.e. $\text{Taylor}^1_{(1,1)}(E)$. To compute $E(x,y) = \frac{D(x,y)}{D(1,1)}$, we need $D(x,y)$, so we need $\text{Taylor}^1_{(1,1)}(D)$. For $D(x,y) = \frac{1}{2!} y^2 \partial_y^2 C(x,0)$, we need the Taylor expansion of C at $(1,0)$ of order $1+2=3$. For $C(x,y) = B(x(0.1y+0.9),1)$, we need the Taylor expansion of B at $(1(0.1 \cdot 0 + 0.9), 1) = (0.9, 1)$ of order 3. Since

$$B(x,y) = \exp(20(x-1)) = \exp(20((x-0.9)-0.1)) = \exp(20(x-0.9)-2)$$

this Taylor expansion is

$$\text{Taylor}^3_{(0.9,1)}(B) = e^{-2} + 20e^{-2} \cdot (x-0.9) + 200e^{-2} \cdot (x-0.9)^2 + \frac{4000}{3} e^{-2} (x-0.9)^3$$

Furthermore, since

$$C(1+(x-1), 0+y) = B((1+(x-1))(0.1y+0.9), 1) = B(0.9+0.9(x-1)+0.1y+0.1(x-1)y, 1)$$

the Taylor expansion of C at $(1,0)$ is given by replacing $(x-0.9)$ with $0.9(x-1)+0.1y+0.1(x-1)y$ and we get

$$\begin{aligned}\text{Taylor}^3_{(1,0)}(C) = e^{-2}\big(1 + 2y + 2y^2 + \frac{4}{3}y^3 + 18(x-1) + 38(x-1)y + 40(x-1)y^2 \\ + 162(x-1)^2 + 360(x-1)^2 y + 972(x-1)^3\big)\end{aligned}$$

For $D(x,y) = \frac{1}{2!} y^2 \partial_y^2 C(x,0)$, we compute the second partial derivative w.r.t. y, substitute $y = 0$ and then multiply by $y^2 = (1 + (y-1))^2$, yielding

$$\begin{aligned}D(x,y) &= \frac{1}{2!} y^2 e^{-2} \left(4 + 8y + 80(x-1) + \text{higher order terms}\right)\big|_{y=0} \\ &= e^{-2}(4 + 80(x-1) + \text{higher order terms})(1 + (y-1))^2 \\ &= e^{-2}(4 + 80(x-1) + 8(y-1) + \text{higher order terms})\end{aligned}$$

As a consequence, we find that $D(1,1) = 4e^{-2}$, so $\text{Taylor}^1_{(1,1)}(E) = 1 + 20(x-1) + 2(y-1)$. Thus $\mathbb{E}[X] = \partial_x E(1,1) = 20$, as we've seen before.

Memoizing intermediate results When computing the Taylor expansion of a function G, it is important to memoize the intermediate Taylor expansions of subexpressions

of G if they occur more than once. For example, conditionals are translated to a generating function that uses the previous generating function twice. Evaluating it repeatedly would double the running time for each conditional. With memoization, it is possible to handle programs with lots of branching (e.g. $> 2^{100}$ paths in the mixture model, cf. Table 4.5) in a reasonable amount of time.

4.3.2 Analysis of the Running Time

Size of Taylor polynomials If the program makes observations (or compares variables with constants) $d_1, \ldots, d_m$, the running time of our exact inference method will depend on these constants. Indeed, each observation of (or comparison of a variable with) d_i requires the computation of the d_i-th partial derivative of the generating function. In our Taylor polynomial approach, this means that the highest order of Taylor polynomials computed will be $d := \sum_{i=1}^{m} d_i$. In fact, the decrement statement $X_k \mathrel{\dot{-}}= 1$ and sampling from compound Bernoulli distributions $X_k \sim \text{Bernoulli}(X_j)$ may also require computing derivatives, so their count has to be included in the sum. If the program has n variables, storing the coefficients of this polynomial alone requires $O(d^n)$ space. This can be reduced if the generating function is a low-degree polynomial, in which case we do not store the zero coefficients of the terms of higher order.

Complexity of operations The bottleneck of the Taylor polynomial operations is composition. For n variables, naive composition of Taylor polynomials takes $O(d^{3n})$ time, where d is the order of the Taylor expansion. Note that this is polynomial in d, contrary to the symbolic approach, but exponential in the number of variables n. This is not as bad as it seems because in many cases, the number of variables can be kept to one or two, as opposed to the values of data points (such as the models from Section 4.4). In fact, we exploit the specific composition structure of the generating functions arising from our semantics to achieve $O(d^3)$ for $n = 1$ and $O(d^{n+3})$ for $n \geq 2$ in the worst case, while often being faster in practice. Overall, our implementation takes $O(sd^{n+3})$ time in the worst case where s is the number of statements.

Theorem 4.3.2. *Our exact inference method can evaluate the k-th derivative of a generating function in $O(s \cdot d^{n+3})$, and $O(sd^3)$ for $n = 1$, where s is the number of statements, n is the number of variables of the program, and $d := k + m + b + o$ is the degree of the Taylor polynomial to be computed, given by the sum of k, the number m of decrement statements $X_k \mathrel{\dot{-}}= 1$, the number b of compound Bernoulli samples $X_k \sim \text{Bernoulli}(X_j)$, and the sum o of all values that are observed or compared against in the program. (Note that the summands m and b were erroneously missing in the original statement of the theorem by Zaiser et al. [2023].)*

Proof. The above arguments already analyze the bottlenecks of the algorithm. But for completeness' sake, we consider all operations being performed on generating functions and their Taylor expansions (cf. Section 4.3.1).

The maximum order of differentiation needed is $d = k + m + b + o$ because we want to evaluate the k-th derivative of the generating function of the program, which itself requires up to one partial derivative for each decrement statement $X_k \mathrel{\dot{-}}= 1$, one for each compound Bernoulli sample $X_k \sim \text{Bernoulli}(X_j)$, and v partial derivatives for each

observation of or comparison with a value v. Since there are n variables, the maximum number of Taylor coefficients that need to be stored is $O(d^n)$.

Addition of two such Taylor polynomials and scalar multiplications trivially take $O(d^n)$ time. Since in the GF semantics, a GF is only ever multiplied by one with at most two variables, the multiplications take $O(d^{n+2})$ time. Division only happens with a divisor of degree at most 1, which can also be implemented in $O(d^{n+1})$ time [Griewank and Walther 2008, Chapter 13]. Exponentiation exp, logarithms log, and powers $(-)^m$ are only performed on polynomials with one variable, where they can be implemented in $O(d^2)$ time [Griewank and Walther 2008, Table 13.2]. Finally, the only substitutions that are required are ones where a term with at most two variables is substituted for a variable, i.e. of the form $p(\boldsymbol{x}[k \mapsto q(x_1, x_2)])$ where $p(\boldsymbol{x}) = \sum_{|\boldsymbol{\alpha}| \leq d} p_{\boldsymbol{\alpha}} \boldsymbol{x}^{\boldsymbol{\alpha}}$. Then

$$p(\boldsymbol{x}[k \mapsto q(x_1, x_2)]) = p_0(\boldsymbol{x}) + q(x_1, x_2) \cdot (p_1(\boldsymbol{x}) + q(x_1, x_2) \cdot (p_2(\boldsymbol{x}) + \cdots))$$

where $p_i(\boldsymbol{x}) = \sum_{\boldsymbol{\alpha}:\alpha_k = i} p_{\boldsymbol{\alpha}} \boldsymbol{x}^{\boldsymbol{\alpha}[k \mapsto 0]}$. This needs d multiplications of $q(x_1, x_2)$ in 2 variables and a polynomial in n variables, each of which takes $O(d^{n+2})$ time. In total, the substitution can be performed in $O(d^{n+3})$ time. Finally, if there is only one variable, the substitution can be performed in $O(d^3)$ time because q can only have one variable.

In summary, the input SGCL program has s statements, and each of them is translated to a bounded number of operations on generating functions. These operations are performed using Taylor expansions of degree at most d, each of which takes at most $O(d^{n+3})$ time (and at most $O(d^3)$ time for $n = 1$). So overall, the running time is $O(s \cdot d^{n+3})$ and $O(sd^3)$ for $n = 1$. □

Potential improvements of the running time Using various tricks, such as the *fast Fourier transform (FFT)*, the running time of the composition of Taylor polynomials, i.e. substitution of polynomials, can be improved to $O((d \log d)^{3n/2})$ [Brent and Kung 1978]. This is still exponential in the number of variables n, but would improve our asymptotic running time for $n < 6$ program variables. In fact, we experimented with an implementation of FFT-based multiplication and composition, but discarded this option because it led to much greater numerical instabilities. It is possible that the running time could instead be improved with the first algorithm in Brent and Kung [1978], which relies on fast matrix multiplication instead of FFT. In addition, it should often be possible to exploit independence structure in programs, so the GF can be factored into independent GFs for subsets of the variables. We did not implement this because it was not relevant for our benchmarks.

Performance in practice Note that, for many programs, the running time will be better than the worst case. Indeed, substitutions are the most expensive operation, if the substitute expression is of high degree. For some models, like the population model, the substitutions take the form of Theorem 4.3.4, i.e. $G(\boldsymbol{x}[k \mapsto (1-p)x_k])$, which is of degree 1. Such a substitution can be performed in $O(d^n)$ time, because one simply needs to multiply each coefficient by a power of $(1-p)$. This explains why the population model is so fast in the experimental section (Section 4.4). In other cases as well, we can often make use of the fact that the polynomials have low degree to reduce the space needed to store the coefficients and to speed up the operations.

4.3.3 Reducing the Number of Variables

Given the exponential running time in n, it is crucial to reduce the number of variables when writing probabilistic programs. For one thing, program variables that are no longer needed can often be reused for a different purpose later. Furthermore, sampling can be combined with addition:

$$X_k \mathrel{+}\!\sim D \quad \text{stands for} \quad X_{n+1} \sim D; X_k := X_k + X_{n+1}$$

The GFs for these statements can easily be computed without introducing the temporary variable X_{n+1}. This is useful for the population models in Section 4.4.

Theorem 4.3.3. *The GF semantics of $X_k \mathrel{+}\!\sim D$ can be computed without introducing a new variable:*

$$[\![X_k \mathrel{+}\!\sim D]\!]_{\mathrm{gf}}(G)(\boldsymbol{x}) = G(\boldsymbol{x}) \cdot \mathrm{gf}(D)(x_k)$$

Proof. Let $G'(\boldsymbol{x}')$ with $\boldsymbol{x}' := (x_1, \ldots, x_{n+1})$ be a generating function of $\boldsymbol{X}' := (X_1, \ldots, X_{n+1})$ with the GF of the marginal distribution of $(X_1, \ldots, X_n)$ being $G'(\boldsymbol{x}, 1) = G(\boldsymbol{x})$. We use the desugaring on this GF G' and marginalize out X_{n+1} at the end by substituting 1 for x_{n+1}:

$$\begin{aligned}
[\![X_k \mathrel{+}\!\sim D]\!]_{\mathrm{gf}}(G)(\boldsymbol{x}) &= [\![X_k \mathrel{+}\!\sim D]\!]_{\mathrm{gf}}(G')(x_1,\ dots, x_n, 1) \\
&= [\![X_k := X_k + X_{n+1}]\!]_{\mathrm{gf}}([\![X_{n+1} \sim D]\!]_{\mathrm{gf}}(G'))(\boldsymbol{x}, 1) \\
&= [\![X_{n+1} \sim D]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}, x_k) \\
&= G'(\boldsymbol{x}, 1) \cdot \mathrm{gf}(D)(x_k) \\
&= G(\boldsymbol{x}) \cdot \mathrm{gf}(D)(x_k)
\end{aligned}$$

□

Observations from compound distributions A similar optimization can be applied to observations from compound distributions. They are generally the bottleneck for the running time because they are very common in Bayesian models and their desugaring also introduce a new temporary variable:

$$\text{observe}\, d \sim D(X_k) \quad \text{stands for} \quad X_{n+1} \sim D(X_k); \text{observe}\, X_{n+1} = d$$

We can find more efficient ways of computing their semantics without the extra variable X_{n+1}. In fact, this optimization is essential to achieving good performance for many of the examples in Section 4.4.

To this end, we simplify $[\![\text{observe}\, d \sim D(X_k)]\!]_{\mathrm{gf}}(G)$ for an initial generating function $G(\boldsymbol{x})$. As the syntactic sugar introduces a new variable, we temporarily have to work with a GF $G'(\boldsymbol{x}')$ of $\boldsymbol{X}' := (X_1, \ldots, X_{n+1})$ with parameters $\boldsymbol{x}' := (x_1, \ldots, x_{n+1})$. Of course, if we marginalize out the extra variable X_{n+1}, we must obtain the original GF $G(\boldsymbol{x}) = G'(\boldsymbol{x}, 1)$. Thus $\text{observe}\, d \sim D(X_k)$ transforms G by switching from G to G', transforming G', and then marginalizing out X_{n+1} at the end, when it is no longer needed, by substituting 1 for x_{n+1}:

$$\begin{aligned}
&[\![\text{observe}\, d \sim D(X_k)]\!]_{\mathrm{gf}}(G)(\boldsymbol{x}) \\
&= [\![\text{observe}\, X_{n+1} = d]\!]_{\mathrm{gf}}([\![X_{n+1} \sim D(X_k)]\!]_{\mathrm{gf}}(G'))(x_1, \ldots, x_n, 1)
\end{aligned}$$

$$= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} [\![X_{n+1} \sim D(X_k)]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}') \Big|_{x_{n+1}=0} \tag{4.1}$$

The subsequent computations depend on the distribution D and are given in the following theorems.

Theorem 4.3.4. *Observing from compound binomial distributions can be implemented without introducing a new variable:*

$$[\![\mathsf{observe}\, d \sim \mathsf{Binomial}(X_k, p)]\!]_{\mathrm{gf}}(G) = \frac{1}{d!}(p x_k)^d \cdot \partial_k^d G(\boldsymbol{x}[k \mapsto (1-p)x_k])$$

Proof. A proof for the discrete setting was given in Winner et al. [2017]. In our setting, which allows continuous distributions, we cannot use the usual argument about power series anymore. Instead, we continue Eq. (4.1) as follows:

$$\begin{aligned}
&[\![\mathsf{observe}\, d \sim \mathsf{Binomial}(X_k, p)]\!]_{\mathrm{gf}}(G) \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} [\![X_{n+1} \sim \mathsf{Binomial}(X_k)]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}') \Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} G'(\boldsymbol{x}'[k \mapsto x_k \cdot (p x_{n+1} + 1 - p), n+1 \mapsto 1]) \Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} G(\boldsymbol{x}[k \mapsto x_k \cdot (p x_{n+1} + 1 - p)]) \Big|_{x_{n+1}=0} \\
&\overset{(*)}{=} \frac{1}{d!} \partial_k^d G'(\boldsymbol{x}[k \mapsto x_k \cdot (p x_{n+1} + 1 - p)]) \cdot (p x_k)^d \Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} (p x_k)^d \partial_k^d G(\boldsymbol{x}[k \mapsto (1-p)x_k])
\end{aligned}$$

where $(*)$ follows by iterating the following argument:

$$\begin{aligned}
&\frac{\partial}{\partial x_{n+1}} G(\boldsymbol{x}[k \mapsto x_k \cdot (p x_{n+1} + 1 - p)]) \\
&= \partial_k G(\boldsymbol{x}[k \mapsto x_k \cdot (p x_{n+1} + 1 - p)]) \cdot \frac{\partial (x_k \cdot (p x_{n+1} + 1 - p))}{\partial x_{n+1}} \\
&= \partial_k G(\boldsymbol{x}[k \mapsto x_k \cdot (p x_{n+1} + 1 - p)]) \cdot p x_k
\end{aligned}$$

which holds by the chain rule. □

Theorem 4.3.5. *Observing from compound Poisson distributions can be implemented without introducing a new variable:*

$$[\![\mathsf{observe}\, d \sim \mathsf{Poisson}(\lambda X_k)]\!]_{\mathrm{gf}}(G) = \frac{1}{d!} D_{k,\lambda}^d(G)(\boldsymbol{x}[k \mapsto e^{-\lambda} x_k])$$

where

$$D_{k,\lambda}(G)(\boldsymbol{x}) := \lambda x_k \cdot \partial_k G(\boldsymbol{x})$$

Proof. Let $F : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then

$$\begin{aligned}
&\frac{\partial}{\partial x_{n+1}} F(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}]) \\
&= \partial_k F(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}]) \cdot x_k e^{\lambda(x_{n+1}-1)} \cdot \lambda \\
&= D_{k,\lambda}(F)(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}])
\end{aligned}$$

Inductively, we get

$$\frac{\partial^d}{\partial x_{n+1}^d} G(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}]) = D_{k,\lambda}^d(G)(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}])$$

Substituting $x_{n+1} \mapsto 0$ in the final expression and dividing by $d!$ yields the following continuation of Eq. (4.1):

$$\begin{aligned}
&[\![\mathsf{observe}\, d \sim \mathsf{Poisson}(\lambda X_k)]\!]_{\mathrm{gf}}(G) \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} [\![X_{n+1} \sim \mathsf{Poisson}(\lambda X_k)]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}')\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} G'(\boldsymbol{x}'[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}, n+1 \mapsto 1])\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} \frac{\partial^d}{\partial x_{n+1}^d} G(\boldsymbol{x}[k \mapsto x_k \cdot e^{\lambda(x_{n+1}-1)}])\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!} D_{k,\lambda}^d(G)(\boldsymbol{x}[k \mapsto e^{-\lambda} x_k])
\end{aligned}$$

□

Theorem 4.3.6. *Observing from compound negative binomial distributions can be implemented without introducing a new variable:*

$$[\![\mathsf{observe}\, d \sim \mathsf{NegBinomial}(X_k, p)]\!]_{\mathrm{gf}}(G) = \frac{1}{d!} \sum_{i=0}^{d} \partial_k^i G(\boldsymbol{x}[k \mapsto p \cdot x_k]) \cdot p^i x_k^i \cdot (1-p)^d L_{d,i}$$

where the numbers $L_{d,i}$ are known as Lah numbers and defined recursively as follows:

$$\begin{aligned}
L_{0,0} &:= 1 && \\
L_{d,i} &:= 0 && \text{for } i < 0 \text{ or } d < i \\
L_{d+1,i} &:= (d+i) L_{d,i} + L_{d,i-1} && \text{for } 0 \le i \le d+1
\end{aligned}$$

Proof. Let $F : \mathbb{R}^n \to \mathbb{R}$ be a smooth function and $y : \mathbb{R} \to \mathbb{R}$ given by $y(z) := \frac{1}{1-(1-p)z}$. Then $y'(x) = -\frac{1}{(1-(1-p)x)^2} \cdot (-(1-p)) = (1-p)y(x)^2$ and

$$\begin{aligned}
&\frac{\partial}{\partial x_{n+1}} F(\boldsymbol{x}[k \mapsto \frac{x_k \cdot p}{1-(1-p)x_{n+1}}]) \\
&= \partial_k F(\boldsymbol{x}[k \mapsto x_k p \cdot y(x_{n+1})]) \cdot p x_k \cdot (1-p) y(x_{n+1})^2
\end{aligned}$$

We claim that we can write

$$\frac{\partial^d}{\partial x_{n+1}^d} F(\boldsymbol{x}[k \mapsto \frac{p x_k}{1-(1-p)x_{n+1}}])$$
$$= \sum_{i=0}^{d} \partial_k^i F(\boldsymbol{x}[k \mapsto x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^d L_{d,i} \cdot y(x_{n+1})^{d+i}$$

The claim is proved by induction. First the case $d = 0$:

$$F(\boldsymbol{x}[k \mapsto \frac{p x_k}{1-(1-p)x_{n+1}}]) = F(\boldsymbol{x}[k \mapsto \frac{p x_k}{1-(1-p)x_{n+1}}]) \cdot (1-p)^0 x_k^0 L_{0,0} \cdot y(x_{n+1})^0$$

Next, the induction step:

$$\frac{\partial^{d+1}}{\partial x_{n+1}^{d+1}} F(\boldsymbol{x}[k \mapsto \frac{p x_k}{1-(1-p)x_{n+1}}])$$
$$= \frac{\partial}{\partial x_{n+1}} \left(\sum_{i=0}^{d} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^d L_{d,i} \cdot y(x_{n+1})^{d+i} \right)$$
$$= \sum_{i=0}^{d} \partial_k^{i+1} F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p x_k y'(x_{n+1}) \cdot p^i x_k^i (1-p)^d L_{d,i} y(x_{n+1})^{d+i}$$
$$+ \sum_{i=0}^{d} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i (1-p)^d \cdot L_{d,i}(d+i) y(x_{n+1})^{d+i-1} \cdot y'(x_{n+1})$$
$$= \sum_{i=0}^{d} \partial_k^{i+1} F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^{i+1} x_k^{i+1} \cdot (1-p)^{d+1} y(x_{n+1})^2 \cdot L_{d,i} y(x_{n+1})^{d+i}$$
$$+ \sum_{i=0}^{d} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^{d+1} y(x_{n+1})^2 \cdot (d+i) L_{d,i} y(x_{n+1})^{d+i-1}$$
$$= \sum_{i=0}^{d+1} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^{d+1} L_{d,i-1} y(x_{n+1})^{d+i+1}$$
$$+ \sum_{i=0}^{d+1} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^{d+1} (d+i) L_{d,i} y(x_{n+1})^{d+i+1}$$
$$= \sum_{i=0}^{d+1} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^{d+1} ((d+i) L_{d,i} + L_{d,i-1}) y(x_{n+1})^{d+i+1}$$
$$= \sum_{i=0}^{d+1} \partial_k^i F(\boldsymbol{x}[k \mapsto p x_k \cdot y(x_{n+1})]) \cdot p^i x_k^i \cdot (1-p)^{d+1} L_{d+1,i} y(x_{n+1})^{d+i+1}$$

Substituting $x_{n+1} \mapsto 0$ in the final expression and dividing by $d!$ yields the following continuation of Eq. (4.1):

$$[\![\text{observe } d \sim \text{NegBinomial}(X_k, p)]\!]_{\text{gf}}(G)$$

$$
\begin{aligned}
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d}[\![X_{n+1} \sim \mathsf{NegBinomial}(X_k, p)]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}')\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d} G'(\boldsymbol{x}'[k \mapsto \frac{x_k \cdot p}{1-(1-p)x_{n+1}}, n+1 \mapsto 1])\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d} G(\boldsymbol{x}[k \mapsto \frac{x_k \cdot p}{1-(1-p)x_{n+1}}])\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!}\sum_{i=0}^{d} \partial_k^i G(\boldsymbol{x}[k \mapsto p \cdot x_k]) \cdot p^i x_k^i \cdot (1-p)^d L_{d,i} \qquad \square
\end{aligned}
$$

Theorem 4.3.7. *Observing from compound Bernoulli distributions can be implemented without introducing a new variable:*

$$
[\![\mathsf{observe}\, d \sim \mathsf{Bernoulli}(X_k)]\!]_{\mathrm{gf}}(G) = \begin{cases} G(\boldsymbol{x}) - x_k \partial_j G(\boldsymbol{x}) & d = 0 \\ x_k \cdot \partial_j G(\boldsymbol{x}) & d = 1 \\ 0 & \textit{otherwise} \end{cases}
$$

Proof. We continue Eq. (4.1) as follows:

$$
\begin{aligned}
&[\![\mathsf{observe}\, d \sim \mathsf{Bernoulli}(X_k)]\!]_{\mathrm{gf}}(G)(\boldsymbol{x}) \\
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d}[\![X_{n+1} \sim \mathsf{Bernoulli}(X_k)]\!]_{\mathrm{gf}}(G')(\boldsymbol{x}')\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d}\left(G'(\boldsymbol{x}'[n+1 \mapsto 1]) + x_k(x_{n+1}-1)\cdot \partial_k G'(\boldsymbol{x}'[n+1 \mapsto 1])\right)\Big|_{x_{n+1}=0} \\
&= \frac{1}{d!}\frac{\partial^d}{\partial x_{n+1}^d}\left(G(\boldsymbol{x}) + x_k(x_{n+1}-1)\cdot \partial_k G(\boldsymbol{x})\right)\Big|_{x_{n+1}=0}
\end{aligned}
$$

If $d = 0$, the right-hand side is $G(\boldsymbol{x}) - x_k \partial_k G(\boldsymbol{x})$. If $d = 1$, the right-hand side is $x_k \partial_k G(\boldsymbol{x})$. For larger $d \geq 2$, the d-th derivative vanishes, which completes the proof. $\square$

4.3.4 Implementation

Overview Our tool ***Genfer*** ("**Gen**erating **f**unctions for in**fer**ence") takes an SGCL program P with n variables as input, and a variable X_i whose posterior distribution is to be computed. It translates it to the GF $[\![P]\!]_{\mathrm{gf}}(\mathbf{1})$ according to the GF semantics and normalizes it: $G := \mathrm{normalize}([\![P]\!]_{\mathrm{gf}}(\mathbf{1}))$. This G is the GF of the posterior distribution of the program, from which it extracts the posterior moments of X_i and its probability masses (if X_i is discrete).

Computation of moments First, it computes the first four (raw) posterior moments from the factorial moments, which are obtained by differentiation of G:

$$
\begin{aligned}
M_1 &:= \mathbb{E}[X_i] = \partial_i G(\mathbf{1}_n) \\
M_2 &:= \mathbb{E}[X_i^2] = \mathbb{E}[(X_i)(X_i - 1)] + \mathbb{E}[X_i] = \partial_i^2 G(\mathbf{1}_n) + \partial_i G(\mathbf{1}_n)
\end{aligned}
$$

$$\begin{aligned}
M_3 := \mathbb{E}[X_i^3] &= \mathbb{E}[X_i(X_i-1)(X_i-2)] + 3\mathbb{E}[X_i(X_i-1)] + \mathbb{E}[X_i] \\
&= \partial_i^3 G(\mathbf{1}_n) + 3\partial_i^2 G(\mathbf{1}_n) + \partial_i G(\mathbf{1}_n) \\
M_4 := \mathbb{E}[X_i^4] &= \mathbb{E}[X_i(X_i-1)(X_i-2)(X_i-3)] + 6\mathbb{E}[X_i(X_i-1)(X_i-2)] \\
&\quad + 7\mathbb{E}[X_i(X_i-1)] + \mathbb{E}[X_i] \\
&= \partial_i^4 G(\mathbf{1}_n) + 6\partial_i^3 G(\mathbf{1}_n) + 7\partial_i^2 G(\mathbf{1}_n) + \partial_i G(\mathbf{1}_n)
\end{aligned}$$

From the raw moments, it computes the first four (centered/standardized) moments, i.e. the mean, variance, skewness, and kurtosis:

$$\begin{aligned}
\mu := \mathbb{E}[X_i] &= M_1 \\
\sigma^2 := \mathbb{V}[X_i] &= \mathbb{E}[(X_i-\mu)^2] = \mathbb{E}[X_i^2] - 2\mu\mathbb{E}[X_i] + \mu^2 = M_2 - \mu^2 \\
\operatorname{Skew}[X_i] &= \frac{\mathbb{E}[(X_i-\mu)^3]}{\sigma^3} = \frac{\mathbb{E}[X_i^3] - 3\mu\mathbb{E}[X_i^2] + 3\mu^2\mathbb{E}[X_i] - \mu^3}{\sigma^3} = \frac{M_3 - 3\mu M_2 + 2\mu^3}{\sigma^3} \\
\operatorname{Kurt}[X_i] &= \frac{\mathbb{E}[(X_i-\mu)^4]}{\sigma^4} = \frac{\mathbb{E}[X_i^4] - 4\mu\mathbb{E}[X_i^3] + 6\mu^2\mathbb{E}[X_i^2] - 4\mu^3\mathbb{E}[X_i] + \mu^4}{\sigma^4} \\
&= \frac{M_4 - 4\mu M_3 + 6\mu^2 M_2 - 3\mu^4}{\sigma^4}
\end{aligned}$$

Computation of probability masses If X_i is a discrete variable, the tool computes the posterior probabilities $\mathbb{P}[X_i = k]$ for all $k \in \{0, \ldots, m\}$ up to a threshold m by marginalizing out all the other variables (substituting 1 for them) and calculating the Taylor coefficient at $x_i = 0$:

$$\mathbb{P}[X_i = k] = \frac{1}{k!}\partial_i^k G(\mathbf{1}_n[i \mapsto 0])$$

The threshold m can be set by the user, but by default, it is chosen such that the tail probability $\mathbb{P}[X_i \geq m]$ is guaranteed to be below $\frac{1}{256}$. This is achieved by setting $m := \mu + 4\sqrt[4]{\mathbb{E}[(X_i-\mu)^4]}$ where $\mu := \mathbb{E}[X_i]$ because

$$\mathbb{P}[X_i \geq m] \leq \mathbb{P}\left[|X_i - \mu| \geq 4\sqrt[4]{\mathbb{E}[(X_i-\mu)^4]}\right] = \mathbb{P}\left[(X-\mu)^4 \geq 256\mathbb{E}[(X_i-\mu)^4]\right] \leq \frac{1}{256}$$

by Markov's inequality. Note that in practice, the tail probabilities are typically much smaller than $\frac{1}{256}$, usually in the order of 10^{-5}. For all $k \in \{0, \ldots, m\}$, It would be desirable to compute posterior *densities* for continuous distributions as well. In fact, there are mathematical ways of recovering the probability density function from a generating function via an inverse Laplace transform. However, this cannot be automated in practice because it requires solving integrals, which is intractable.

Implementation details Genfer is implemented in Rust [Matsakis and Klock 2014], a safe systems programming language. The main reasons were low-level control and performance: the operations on the Taylor polynomials to evaluate derivatives of the generating function need to be fast and are optimized to exploit the structure of GFs arising from probabilistic programs (see Section 4.3.2). C or C++ would have satisfied the performance criterion as well, but Rust's language features like memory safety,

enums (tagged unions), and pattern matching made the implementation a lot more pleasant, robust, and easier to maintain.

The coefficients of the Taylor polynomials are stored in a multidimensional array provided by the `ndarray` library[1]. Arbitrary-precision floating-point numbers and unbounded rational numbers are provided by the `rug` library[2], which is an interface to the GNU libraries GMP (for rationals) and MPFR (for floats). Genfer is available on GitHub: github.com/fzaiser/genfer

Number representation Genfer can use several number formats for its computations: 64-bit floating point (the default and fastest), floating point with a user-specified precision, and rational numbers (if no irrational numbers occur). If our implementation is used in floating-point mode, there is the possibility of rounding errors and numerical instabilities, which could invalidate the results. To enable users to detect this, we implemented an option in our tool to use interval arithmetic in the computation: instead of a single (rounded) number, it keeps track of a lower and upper bound on the correctly rounded result. If this interval is very wide, it indicates numerical instabilities.

Numerical instability Initially, we observed large numerical errors (caused by catastrophic cancellation) for probabilistic programs involving continuous distributions. The reason for this turned out to be the term $\log(x)$ that occurs in the GF of the continuous distributions. The Taylor coefficients of $\log(x)$ at points $0 < z < 1$ are not well-behaved:

$$\log(x) = \log(z) + \sum_{n=1}^{\infty} \frac{(-1)^n z^{-n}}{n} (x - z)^n$$

because the coefficients grow exponentially for $z < 1$ (due to the term z^{-n}) and the alternating sign exacerbates this problem by increasing the likelihood of catastrophic cancellations.

To fix this problem, we adopted a slightly modified representation. Instead of computing the Taylor coefficients of the GF G directly, we compute those of the function $H(\boldsymbol{x}) := G(\boldsymbol{x}')$ where $x_i' := x_i$ if X_i is discrete and $x_i' := \exp(x_i)$ if X_i is continuous, thus canceling the logarithm.[3] (The reason we do not use $\boldsymbol{x}' := \exp(\boldsymbol{x})$ for all variables is that for discrete variables, we may need to evaluate at $x_i' = 0$, corresponding to $x_i = -\infty$, which is impossible.) It is straightforward to adapt the GF semantics to this modified representation. Due to the technical nature of the adjustment (case analysis on whether an index corresponds to a discrete variable or not), we do not describe it in detail. With this modified representation, we avoid the catastrophic cancellations and obtain numerically stable results for programs using continuous distributions as well. Using Genfer's interval arithmetic option, we verified that our results are accurate up to at least 5 significant digits for all the examples in Section 4.4.2.

[1] `https://crates.io/crates/ndarray`

[2] `https://crates.io/crates/rug`

[3] Note that this boils down to using the moment generating function (MGF) $\mathbb{E}[\exp(x_i X_i)]$ for continuous variables X_i and the probability generating function (PGF) $\mathbb{E}[x_i^{X_i}]$ for discrete variables X_i. In mathematics, the MGF is more often used for continuous distributions and the PGF more commonly for discrete ones. It is interesting to see that this distinction is also preferred from a numerical perspective.

Table 4.4: Comparison of inference times of tools for exact inference on PSI's benchmarks [Gehr et al. 2016]. (F: only finite discrete distributions, C: continuous priors, FP: floating point mode, $\mathbb{Q}$: rational number mode, unsupp.: unsupported, timeout: 1 hour)

Benchmark	Genfer (FP)	Dice (FP)	Genfer ($\mathbb{Q}$)	Dice ($\mathbb{Q}$)	Prodigy	PSI
alarm (F)	**0.0005s**	0.0067s	**0.0012s**	0.0066s	0.011s	0.0053s
clickGraph (C)	**0.11s**	unsupp.	**3.4s**	unsupp.	unsupp.	46s
clinicalTrial (C)	**150s**	unsupp.	**1117s**	unsupp.	unsupp.	timeout
clinicalTrial2 (C)	**0.0024s**	unsupp.	**0.031s**	unsupp.	unsupp.	0.46s
digitRecognition (F)	**0.021s**	0.83s	**0.11s**	2.7s	31s	146s
evidence1 (F)	**0.0002s**	0.0057s	**0.0003s**	0.0056s	0.0030s	0.0016s
evidence2 (F)	**0.0002s**	0.0056s	**0.0004s**	0.0057s	0.0032s	0.0018s
grass (F)	**0.0008s**	0.0067s	**0.0044s**	0.0067s	0.019s	0.014s
murderMystery (F)	**0.0002s**	0.0055s	**0.0003s**	0.0057s	0.0028s	0.0021s
noisyOr (F)	**0.0016s**	0.0085s	0.019s	**0.0088s**	0.21s	0.055s
twoCoins (F)	**0.0002s**	0.0054s	**0.0003s**	0.0057s	0.0032s	0.0017s

4.4 Empirical Evaluation

In the empirical evaluation, we aim to answer two questions:

1. How does Genfer compare with other exact inference systems on benchmarks that could already be solved by other tools?
2. How does Genfer compare with approximate inference methods on benchmarks that could not be solved exactly before?

Experimental setup All experiments were run on a laptop computer with a Intel® Core™ i5-8250U CPU @ 1.60GHz × 8 processor and 16 GiB of RAM, running Ubuntu 22.04.2 and took a few hours to complete overall. The code to run the benchmarks and instructions on how to reproduce our experiments are available in the supplementary material of Zaiser et al. [2023].

4.4.1 Comparison with Exact Inference Methods

Baselines We compare our tool Genfer with the following tools for exact Bayesian inference: Dice [Holtzen et al. 2020], which uses weighted model counting, PSI [Gehr et al. 2016], which manipulates density functions using computer algebra, and Prodigy [Klinkenberg et al. 2024], which is based on generating functions like Genfer, but uses computer algebra instead of automatic differentiation.[4]

Benchmarks We evaluate them on the PSI benchmarks [Gehr et al. 2016], excluding those that only PSI supports, e.g. due to observations from continuous distributions.[5]

[4]We were not able to run Hakaru [Narayanan et al. 2016], which uses the computer algebra system Maple as a backend, but we do not expect it to produce better results than the other systems, given that PSI is generally more performant [Gehr et al. 2016].

[5]Specifically, the benchmarks were taken from the PSI paper [Gehr et al. 2016, Table 1], excluding "HIV", "LinearRegression1", "TrueSkill", "AddFun/max", "AddFun/sum", and "LearningGaussian". While

These benchmarks were translated mostly manually to the tools' respective formats, and we checked that the outputs of the tools are consistent with each other. Most benchmarks only use finite discrete distributions (labeled "F"), but three feature continuous priors (labeled "C").

Methodology We measured the wall-clock inference time for each tool, excluding startup time and input file parsing, and recorded the minimum from 5 consecutive runs with a one-hour timeout.[6] Dice and Genfer default to floating-point (FP) numbers, whereas Prodigy and PSI use rational numbers, which is slower but prevents rounding errors. For a fair comparison, we evaluated all tools in rational mode and separately compared Dice with Genfer in FP mode. For the "clinicalTrial" benchmark, Genfer required 400-bit floating-point numbers because 64-bit numbers did not provide enough accuracy. PSI was run with the flag `--dp` on finite discrete benchmarks for better performance.

Results The results (Table 4.4) demonstrate Genfer's speed even on finite discrete models, despite our primary focus on models with infinite support. It should be noted that the number of variables in these benchmarks is quite small. On models with many variables, Genfer would not be competitive with Dice, which is designed for scalable inference. However, Genfer can handle models with infinite support and continuous distributions, which Dice cannot.

4.4.2 Comparison with Approximate Inference Methods

It is impossible to compare our approach with other exact inference methods on realistic models with infinite support (Section 4.4.3): the scalable systems Dice [Holtzen et al. 2020] and SPPL [Saad et al. 2021] do not support such priors and the symbolic solvers PSI [Gehr et al. 2016] and Prodigy [Klinkenberg et al. 2024] run out of memory or time out after an hour.

Truncation As an alternative, we considered approximating the posterior by truncating discrete distributions with infinite support. This reduces the problem to finite discrete inference, which is more amenable to exact techniques. We decided against this approach because Winner and Sheldon [2016] already demonstrated its inferiority to GF-based exact inference on their graphical model. Moreover, it is harder to truncate general probabilistic programs, and truncation does not apply to continuous priors.

Monte Carlo inference Hence, we compare our approach with Monte Carlo inference methods. Specifically, we choose the Anglican [Tolpin et al. 2015] probabilistic

Genfer supports continuous priors, the excluded benchmarks use continuous distributions in other ways that are not supported (e.g. in observations). Note that Dice and Prodigy have no support for continuous distributions at all.

[6] We manually patched the tools Dice, Prodigy, and PSI to measure and output the time taken exclusively for inference to ensure a fairer comparison and because the time a tool takes to startup and parsing times are not very interesting from a research perspective. We ran each tool 5 times in a row and took the minimum running time to reduce the effect of noise (garbage collection, background activity on the computer).

programming system because it offers the best built-in support for discrete models with many state-of-the-art inference algorithms. Other popular systems are less suitable: Gen [Cusumano-Towner et al. 2019] specializes in programmable inference; Stan [Carpenter et al. 2017], Turing [Ge et al. 2018], and Pyro [Bingham et al. 2019] mainly target continuous models; and WebPPL's [Goodman and Stuhlmüller 2014] discrete inference algorithms are less extensive than Anglican's (e.g. no support for interacting particle MCMC).

Methodology The approximation error of Anglican's inference algorithms depends on their *settings*[7] (e.g. the number of particles for SMC) and decreases with the number of samples the longer it is run. To ensure a fair comparison, we use the following setup: for each inference problem, we run several inference algorithms with various *configurations* (settings and sampling budgets) and report the approximation error and the elapsed time. The *sampling budgets* were 1000 or 10000 samples, because significantly lower sample sizes gave unusable results and significantly higher sample sizes took much more time than our exact method. We discard the first 20% of the samples, a standard procedure called "burn-in". We list the inference algorithms and their settings below.

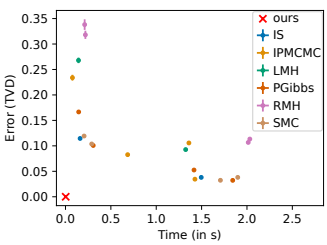
- Importance Sampling (IS): no settings
- Lightweight Metropolis Hastings (LMH): no settings
- Random-Walk Metropolis Hastings (RMH): α (the probability of using a local MCMC move) and σ (the spread of the local move). We used the default $\alpha = 0.5$, $\sigma = 1$ and another setting facilitating local moves with $\alpha = 0.8, \sigma = 2$.
- Sequential Monte Carlo (SMC): number of particles $\in \{1, 1000\}$. The default is 1 and we also picked 1000 to see the effect of a larger number of particles while not increasing the running time too much.
- Particle Gibbs (PGibbs): number of particles $\in \{2, 1000\}$. The default is 2 and we also picked 1000 as the other setting for the same reason as for SMC.
- Interacting Particle MCMC (IPMCMC): "number of particles per sweep", "number of nodes running SMC and CSMC", "number of nodes running CSMC", and "whether to return all particles (or just one)". We left the last three settings at their defaults (32 nodes, 16 nodes running CSMC, return all particles). For the number of nodes and the number of nodes we chose the default setting (2) and a higher one (1000) for the same reason as SMC.

To measure the quality of the approximation, we report the *total variation distance (TVD)* between the exact solution and the approximate posterior distribution, as well as the *approximation errors* of the posterior mean μ, standard deviation σ, skewness, and kurtosis. To ensure invariance of our error metrics under translation and scaling of the distribution, we compute the error of the mean as $\frac{|\hat{\mu}-\mu|}{\sigma}$, where μ is the true posterior mean, $\hat{\mu}$ its approximation, and σ the true posterior standard deviation; the relative error of the standard deviation σ (because σ is invariant under translation but not scaling), and the absolute error of skewness and kurtosis (because they are invariant under both

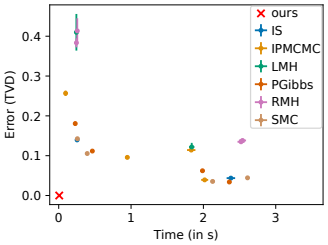
[7] See https://probprog.github.io/anglican/inference/ for a full list of Anglican's inference algorithms and their settings.

Table 4.5: Summary of the benchmarks: n is the number of variables, o is the number of observations, d is the sum of all the observed values, i.e. the total order of derivatives that needs to be computed, s is the number of statements, p the number of program paths, and "cont.?" indicates whether continuous priors are used.

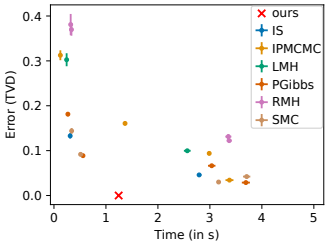
Benchmark	n	o	d	s	p	cont.?
population model (Fig. 4.1a)	1	4	254	13	1	✗
population (modified, Fig. 4.1b)	1	4	254	21	16	✗
population (two types, Fig. 4.1c)	2	8	277	30	1	✗
change point (Fig. 4.2a)	2	109	188	12433	111	✓
mixture model (Figs. 4.2b and 4.2c)	2	218	188	329	$2^{109} \approx 6 \cdot 10^{32}$	✗
HMM (Fig. 4.2d)	3	60	51	152	$2^{30} \approx 10^9$	✗



(a) Original model



(b) Modified arrival rate



(c) Two-type branching process

Figure 4.1: Comparison of the population model and its modifications with approximate inference: mean and standard error of the computation time and TVD over 20 repeated runs.

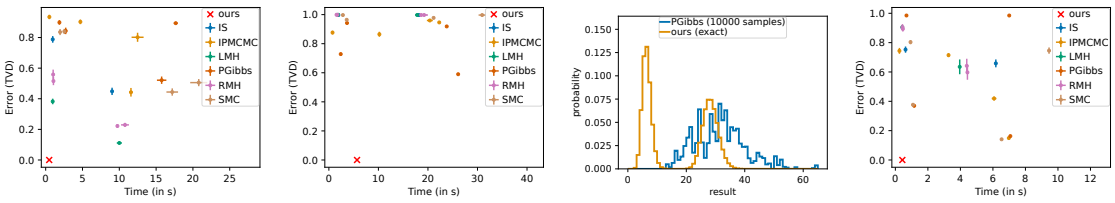
translation and scaling). To reduce noise, we average all these error measures and the computation times over 20 runs and report the standard error as error bars.

Note that this setup is generous to the approximate methods because we only report the average time for one run of each configuration. However, in practice, one does not know the best configuration, so the algorithms need to be run several times with different settings. By contrast, our method requires only one run because the result is exact.

4.4.3 Benchmarks with Infinite-Support Distributions

We discuss the benchmarks used in our evaluation below. A summary with important information on each benchmark is given in Table 4.5, including some statistics about the probabilistic programs. More details including the SGCL code for each benchmark can be found in Appendix B.1.1.

Population ecology Our first benchmark comes from Winner and Sheldon [2016] and Winner et al. [2017] and models animal populations. We have seen a simplified version in Example 4.1.1. Here we model a population N_k at time steps $k = 0, \ldots, m$. At each time step, there is a Poisson-distributed number of new arrivals, which are added to the binomially distributed number of survivors from the previous time step. Each individual is observed with a fixed probability δ, so the number of observed



(a) Change point: TVD error **(b)** Mixture: TVD error **(c)** Mixture: histogram **(d)** HMM: TVD error

Figure 4.2: Plots for the change point, mixture, and hidden Markov model.

individuals is binomially distributed:

$$
\begin{aligned}
&New_k \sim \text{Poisson}(\lambda_k);\\
&Survivors_k \sim \text{Binomial}(N_{k-1}, \delta);\\
&N_k := New_k + Survivors_k;\\
&\text{observe } y_k \sim \text{Binomial}(N_k, \rho);
\end{aligned}
$$

where the model parameters $\lambda_k \in \mathbb{R}, \delta \in [0, 1]$ are taken from Winner and Sheldon [2016]; the detection probability ρ is set to 0.2 (Winner and Sheldon [2016] consider a range of values, but we pick one for space reasons); and the observed number $y_k \in \mathbb{N}$ of individuals in the population at time step k is simulated from the same ground truth as Winner and Sheldon [2016]. The goal is to infer the final number N_m of individuals in the population. We set the population size model parameter and hence the observed values which influence the running time to be 4 times larger than the largest in Winner and Sheldon [2016] (see details in Appendix B.1.1). The results (Fig. 4.1a) show that our method is superior to MCMC methods in both computation time and accuracy since it is exact.

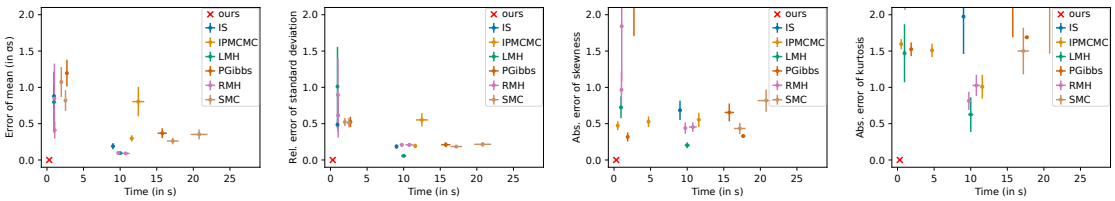
Modifications While this model was already solved exactly in Winner and Sheldon [2016], our probabilistic programming approach makes it trivial to modify the model, since the whole inference is automated and one only needs to change a few lines of the program:

(a) we can model the possibility of natural disasters affecting the offspring rate with a conditional: $Disaster \sim \text{Bernoulli}(0.1)$; if $Disaster = 1$ $\{New_k \sim \text{Poisson}(\lambda')\}$ else $\{New_k \sim \text{Poisson}(\lambda)\}$, or

(b) instead of a single population, we can model populations of two kinds of individuals that interact, i.e. a *multitype* branching process (see Fig. 4.1c).

None of these modifications can be handled by Winner and Sheldon [2016] or Winner et al. [2017]. The results of the first modification (Fig. 4.1b) are very similar. The more complex second modification takes longer to solve exactly, but less time than approximate inference with 10000 samples, and achieves zero error.

Change point model Our second benchmark is Bayesian change point analysis, which is about detecting a change in the frequency of certain events over time. We use



(a) Error of the mean **(b)** Error of the std. dev. **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure 4.3: Comparison of moments for the change point model: approximate inference vs our method.

the model from Salvatier et al. [2016] with continuous priors and its 111 real-world data points about the frequency of coal-mining accidents. We compare both the moment errors (Fig. 4.3) and the TVD errors (Fig. 4.2a). In both cases, the approximations are less accurate and take longer than our exact method.

Mixture model We consider a binary mixture model on the same data set, with equal mixture weights and a geometric prior for the rates: each data point is observed from a mixture of two Poisson distributions with different rates and the task is to infer these rates. Due to their multimodality, mixture models are notoriously hard for approximate inference methods, which is confirmed in Fig. 4.2b. Even the runs with the lowest error cover only one of the two modes (cf. the sample histogram in Fig. 4.2c).

Hidden Markov model We use a hidden Markov model based on Saad et al. [2021, Section 2.2], but involving infinite (geometric) priors. It is a two-state system with known transition probabilities and the rate for the observed data depends on the hidden state. We run this model on 30 simulated data points. For this model as well, our method clearly outperforms approximate methods (Fig. 4.2d).

Discussion To our knowledge, our approach is the first to find an exact solution to these problems, except the very first problem without the modifications, which appeared in Winner and Sheldon [2016]. The results show that our method is competitive with approximate inference methods on these benchmarks in terms of computation time and accuracy. For brevity, we only presented the most important aspects of these benchmarks, relegating their encoding as probabilistic programs to Appendix B.1.1. Additional data and plots are available in Appendix B.1.2 and confirm our conclusions.

4.5 Related Work

Exact inference is discussed at length in Section 2.4. In this section we discuss work more closely related to our generating function approach.

Probabilistic circuits In the context of Bayesian inference, probabilistic generating circuits (PGCs) have recently been proposed to boost the expressiveness of probabilistic circuits by leveraging generating functions [H. Zhang et al. 2021] and to enable efficient

inference [Harviainen et al. 2023]. However, PGCs only handle binary variables and thus cannot be applied to distributions with infinite support, which is the focus of our approach. Moreover, Agarwal and Bläser [2024] call into question the importance of generating functions for probabilistic circuits with a key insight: the power of PGCs is not actually due to generating functions but a consequence of their ability to represent negative weights. In fact, they show that probabilistic circuits with negative weights *subsume* PGCs in a certain sense.

Graphical models Generating functions have also been applied to probabilistic graphical models: Winner and Sheldon [2016] find a symbolic representation of the generating function for a Poisson autoregressive model and extract posterior probabilities from it. Subsequently, Winner et al. [2017] extend this model to latent variable distributions other than the Poisson distribution, where symbolic manipulation is no longer tractable. Instead they evaluate generating functions and their derivatives using automatic differentiation, which enables exact inference for this class of graphical models.

Probabilistic programming In the context of discrete probabilistic programs without conditioning, Klinkenberg et al. [2020] use generating functions to (manually) analyze loop invariants and determine termination probabilities. In follow-up work, Chen et al. [2022] extend these techniques to automatically check (under certain restrictions) whether a looping program generates a specified distribution. Their analysis tool Prodigy recently gained the ability to perform Bayesian inference as well Klinkenberg et al. [2024]. It supports discrete distributions with infinite support but no continuous priors and is less scalable than our automatic-differentiation approach (see Section 4.4) since it relies on computer algebra.

4.6 Conclusion

By leveraging generating functions, we have developed and proven correct a framework for exact Bayesian inference on discrete models, even with infinite support and continuous priors. We have demonstrated competitive performance on a range of models specified in an expressive probabilistic programming language, which our tool Genfer processes automatically.

Future work It is a natural question how our method could be integrated with a more general probabilistic programming system. For example, in a sampling-based inference algorithm, one could imagine using generating functions to solve subprograms exactly whenever this is possible. More generally, it would be desirable to explore how the compositionality of the GF translation can be improved. As it stands, our GF translation describes the joint distribution of all program variables – we never reason "locally" about a subset of the variables. A compositional approach would likely facilitate the application of the GF method to functional probabilistic languages like Anglican.

5

Guaranteed bounds via Unrolling and Residual Mass

Contents

5.1 Introduction . . . 129
5.1.1 Contributions . . . 130
5.1.2 Limitations . . . 130
5.2 Lower Bounds via Unrolling . . . 130
5.3 Upper Bounds via Residual Mass . . . 133
5.4 Implementation . . . 135
5.4.1 Discrete SGCL . . . 136
5.4.2 Full SGCL with Generating Functions . . . 136
5.4.3 Overapproximating the Measure Support . . . 137
5.5 Comparison with Previous Work . . . 138
5.6 Conclusion . . . 139

Summary This short chapter presents a simple yet effective technique that transforms an exact inference method for the loop-free fragment of a PPL into a method to compute guaranteed bounds for the full language. In particular, it can extend the generating function semantics for loop-free SGCL programs (Chapter 4) to guaranteed bounds for SGCL programs with loops.

Lower bounds are easy to obtain by *unrolling loops* a finite number of times (Kleene iteration). For upper bounds, we introduce the *residual mass semantics*, which is a flat bound based on the residual probability mass of a loop. The approach is simple, efficient, and enjoys desirable theoretical properties. In particular, we prove soundness and convergence, i.e. the bounds converge to the exact posterior as loops are unrolled further. On the practical side, we describe a fully-automated implementation of the semantics in a tool called *Diabolo* and demonstrate its benefits in a comparison with GuBPI (Chapter 3).

Publication This chapter is based in part on a paper accepted for publication at the *52nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2025)*. The full reference is:

> Fabian Zaiser, Andrzej S. Murawski, and Luke Ong [2025]. "Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops". In: *Proc. ACM Program. Lang.* 9.POPL. DOI: 10.1145/3704874

Supplementary material for the paper is available in Zaiser et al. [2024]. While the paper exclusively deals with discrete SGCL, this chapter generalizes the results to the full SGCL language.

Author contributions I contributed the idea, the theoretical development, the implementation, the empirical evaluation, and the writing. Andrzej Murawski and Luke Ong supervised the project, provided feedback, and helped revise the draft.

5.1 Introduction

Can we extend the exact inference method from Chapter 4 to programs with loops? For instance, consider the following simple probabilistic loop.

Example 5.1.1 (asymmetric random walk)**.** Consider the program P representing a biased random walk on $\mathbb{N}$, starting at 1, and stopping when reaching 0:

$$
\begin{aligned}
&X_1 := 1;\\
&X_2 := 0;\\
&\textsf{while}\, X_1 > 0\,\{\\
&\quad X_2 \mathrel{+}= 1;\\
&\quad \{X_1 \mathrel{+}= 1\}\ [r]\ \{X_1 \mathrel{\dot{-}}= 1\}\\
&\}
\end{aligned}
$$

where X_1 is the current position, X_2 is the number of steps taken, and the bias $r < \frac{1}{2}$ is the probability of going right.
From the theory of discrete random walks and hitting times, it is known that the distribution of this program is given by

$$
[\![P]\!]_{\lightning}(\mathrm{Dirac}(0,0))(\boldsymbol{x}) = \begin{cases} \frac{1}{x_2}\binom{x_2}{(x_2-1)/2} \cdot r^{(x_2-1)/2}(1-r)^{(x_2+1)/2} & \text{if } x_1 = 0 \text{ and } x_2 \text{ is odd} \\ 0 & \text{else (including } \boldsymbol{x} = \lightning) \end{cases}
$$

This distribution is nontrivial to compute by hand, and out of reach for current automated methods, even though it does not use conditioning. But can we at least compute bounds on the distribution? More generally, can we compute guaranteed bounds on the distribution of a loop without the need to find the least fixpoint of the loop semantics exactly? We have already answered this question in the affirmative in Chapter 3. It turns out that for SGCL programs, there is a simpler and more efficient technique, which we discuss in this chapter.

5.1.1 Contributions

We develop a new method to compute guaranteed bounds on the distribution of discrete probabilistic programs with loops and conditioning. Upper bounds are the main challenge because lower bounds can be found by unrolling each loop a finite number of times. To attack this problem, we introduce the ***residual mass semantics***, which is based on the simple idea of bounding the remaining probability mass after the loop unrollings. Despite its simplicity, it has not previously been described, to our knowledge. We make the following contributions:

- We introduce the residual mass semantics as a simple but effective idea for lower bounds (Section 5.2) and upper bounds (Section 5.3).
- We prove soundness and convergence of the bounds to the true distribution (as loops are unrolled further and further). The convergence result is stronger than the corresponding completeness result for GuBPI (Theorem 3.3.13).
- We implement the semantics in a tool called *Diabolo* (Section 5.4) and demonstrate empirically that the implementation is more efficient than previous systems (Section 5.5).

5.1.2 Limitations

Our semantics applies to SGCL programs, which are subject to the expressiveness restrictions discussed in Section 2.2.1. In particular, contrary to GuBPI (Chapter 3), programs cannot apply nonlinear functions or observe values from continuous distributions. Moreover, the initial values of the program variables are fixed: our methods cannot reason parametrically about these inputs.

Our semantics can yield bounds on the distribution of any SGCL program, but convergence with increasing unrolling is only guaranteed if the program terminates almost surely. Moreover, if the program distribution has unbounded support, we cannot bound the moments or tails: the residual mass bound does not contain enough information for this. Finally, since our approach leverages exact inference for the loop-free fragment, it also inherits the associated scalability limitations.

5.2 Lower Bounds via Unrolling

Lower bounds can be computed by unrolling the loop a finite number of times and discarding the part of the distribution that has not exited the loop after the unrollings. Formally, define the ***u-fold unrolling*** $P^{(u)}$ of a program P by unrolling each loop u times:

$$
\begin{aligned}
(P_1; P_2)^{(u)} &:= P_1^{(u)}; P_2^{(u)} \\
(\text{if } E \{P_1\} \text{ else } \{P_2\})^{(u)} &:= \text{if } E \{P_1^{(u)}\} \text{ else } \{P_2^{(u)}\} \\
(\text{while } E \{P\})^{(u)} &:= (\text{while } E \{P\})_0^{(u)} \\
(\text{while } E \{P\})_v^{(u)} &:= \text{if } E \{P^{(u)}; (\text{while } E \{P\})_{v+1}^{(u)}\} \text{ else } \{\text{skip}\}; \quad \text{for } v < u \\
(\text{while } E \{P\})_u^{(u)} &:= \text{while } E \{P^{(u)}\}
\end{aligned}
$$

$$P^{(u)} := P \quad \text{otherwise}$$

Lemma 5.2.1. *Unrolling does not change the semantics:* $[\![P]\!]_{\lightning} = [\![P^{(u)}]\!]_{\lightning}$ *for all programs* P *and* $u \in \mathbb{N}$.

Proof. The proof is a straightforward induction on the structure of the program P. The only nontrivial case is $P = \mathsf{while}\, E\, \{P_1\}$, where we use the fact that the unrolling operator Φ_{E,P_1} corresponds to unrolling the loop syntactically. More specifically, we claim that $[\![(\mathsf{while}\, E\, \{P_1\})_v^{(u)}]\!]_{\lightning} = [\![\mathsf{while}\, E\, \{P_1\}]\!]_{\lightning}$ for all $v \le u$. To see this, note that the unrolling operators agree: $\Phi_{E,P_1^{(u)}} = \Phi_{E,P_1}$ because by the induction hypothesis, we have

$$\Phi_{E,P_1^{(u)}}(\psi)(\mu) = \mu_{\lightning} + \mu|_{\neg E} + \psi([\![P_1^{(u)}]\!]_{\lightning}(\mu|_E)) = \mu_{\lightning} + \mu|_{\neg E} + \psi([\![P_1]\!]_{\lightning}(\mu|_E)) = \Phi_{E,P_1}(\psi)(\mu)$$

Thus the case $v = u$ follows from

$$[\![(\mathsf{while}\, E\, \{P_1\})_u^{(u)}]\!]_{\lightning} = [\![\mathsf{while}\, E\, \{P_1^{(u)}\}]\!]_{\lightning} = \mathrm{lfp}(\Phi_{E,P_1^{(u)}}) = \mathrm{lfp}(\Phi_{E,P_1}) = [\![\mathsf{while}\, E\, \{P_1\}]\!]_{\lightning}$$

For $v < u$, we find by induction (decreasing v):

$$\begin{aligned}
[\![(\mathsf{while}\, E\, \{P_1\})_v^{(u)}]\!]_{\lightning}(\mu) &= [\![\mathsf{if}\, E\, \{P_1^{(u)}; (\mathsf{while}\, E\, \{P_1\})_{v+1}^{(u)}\}\, \mathsf{else}\, \{\mathsf{skip}\}]\!]_{\lightning}(\mu) \\
&= \mu_{\lightning} + [\![(\mathsf{while}\, E\, \{P_1\})_{v+1}^{(u)}]\!]_{\lightning}([\![P_1^{(u)}]\!]_{\lightning}(\mu|_E)) + \mu|_{\neg E} \\
&= \Phi_{E,P_1^{(u)}}([\![(\mathsf{while}\, E\, \{P_1\})_{v+1}^{(u)}]\!]_{\lightning})(\mu) \\
&= \Phi_{E,P_1^{(u)}}([\![\mathsf{while}\, E\, \{P_1\}]\!]_{\lightning})(\mu) \qquad \text{by ind. hyp.} \\
&= \Phi_{E,P_1}([\![\mathsf{while}\, E\, \{P_1\}]\!]_{\lightning})(\mu) \\
&= \Phi_{E,P_1}(\mathrm{lfp}(\Phi_{E,P_1}))(\mu) \\
&= \mathrm{lfp}(\Phi_{E,P_1})(\mu) \\
&= [\![\mathsf{while}\, E\, \{P_1\}]\!]_{\lightning}(\mu)
\end{aligned}$$

□

The ***lower bound semantics*** $[\![P]\!]_{\mathsf{lo}} : \mathrm{Meas}(\mathbb{S}_{\lightning}) \to \mathrm{Meas}(\mathbb{S}_{\lightning})$ is then defined just like $[\![P]\!]_{\lightning}$ except that $[\![\mathsf{while}\, E\, \{P_1\}]\!]_{\mathsf{lo}}(\mu) := \mathbf{0}$. In other words, $[\![P]\!]_{\mathsf{lo}} = [\![P']\!]_{\lightning}$ where P' is the program P where all loops are replaced by infinite loops, effectively setting the measure to zero. Its correctness follows from the monotonicity of the standard semantics.

Theorem 5.2.2 (Soundness of lower bounds)**.** *For all measures* $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$ *and programs* P*, we have* $[\![P]\!]_{\mathsf{lo}}(\mu) \preceq [\![P]\!]_{\lightning}(\mu)$.

Proof. The proof is by induction on the structure of the program P. For $P = P_1; P_2$, we have

$$[\![P]\!]_{\mathsf{lo}}(\mu) = [\![P_2]\!]_{\mathsf{lo}}([\![P_1]\!]_{\mathsf{lo}}(\mu)) \preceq [\![P_2]\!]_{\lightning}([\![P_1]\!]_{\mathsf{lo}}(\mu)) \preceq [\![P_2]\!]_{\lightning}([\![P_1]\!]_{\lightning}(\mu)) = [\![P]\!]_{\lightning}(\mu)$$

by the induction hypothesis and the monotonicity of the standard semantics. For conditionals $P = \mathsf{if}\, E\, \{P_1\}\, \mathsf{else}\, \{P_2\}$, we find

$$[\![P]\!]_{\mathsf{lo}}(\mu) = \mu_{\lightning} + [\![P_1]\!]_{\mathsf{lo}}(\mu|_E) + [\![P_2]\!]_{\mathsf{lo}}(\mu|_{\neg E}) \preceq \mu_{\lightning} + [\![P_1]\!]_{\lightning}(\mu|_E) + [\![P_2]\!]_{\lightning}(\mu|_{\neg E}) = [\![P]\!]_{\lightning}(\mu)$$

by the induction hypothesis. For loops $P = \mathsf{while}\, E\, \{P_1\}$, we have $[\![P]\!]_{\mathsf{lo}}(\mu) = \mathbf{0} \preceq [\![P]\!]_{\lightning}(\mu)$ by definition. In all other cases, $[\![P]\!]_{\mathsf{lo}}(\mu)$ and $[\![P]\!]_{\lightning}(\mu)$ are equal by definition. □

Convergence The lower bounds will get better as loops are unrolled further and further. In fact, they enjoy a stronger convergence property than the interval trace semantics from Corollary 3.3.14. Whereas the latter requires a "good" choice of interval traces, the lower bounds in this section simply converge as the unrolling limit is increased.

Theorem 5.2.3 (Convergence of lower bounds)**.** *For all* finite *measures* $\mu \in \mathrm{Meas}(\mathbb{S}_\lightning)$ *and programs* P*, the lower bound* $[\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)$ *converges (in total variation distance) monotonically to the true distribution* $[\![P]\!]_\lightning(\mu)$ *as* $u \to \infty$*.*

Proof. The proof is by induction on the structure of the program P. Let $P_*^{(u)}$ denote the program $P^{(u)}$ with all loops replaced by diverge := while true {skip}, such that $[\![P^{(u)}]\!]_{\mathrm{lo}} = [\![P_*^{(u)}]\!]_\lightning$. Note that $P_*^{(u)}$ can be defined inductively just like $P^{(u)}$. It is straightforward to show $[\![P^{(u)}]\!]_{\mathrm{lo}} = [\![P_*^{(u)}]\!]_\lightning$ because $[\![\mathrm{diverge}]\!]_\lightning = \mathbf{0}$. We claim that $([\![P_*^{(u)}]\!]_\lightning(\mu))_{u\in\mathbb{N}}$ is an ω-chain (monotone sequence of measures) with supremum $[\![P]\!]_\lightning(\mu)$. This implies

$$[\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(\mathbb{S}_\lightning) \to [\![P]\!]_\lightning(\mu)(\mathbb{S}_\lightning) \le \mu(\mathbb{S}_\lightning) < \infty$$

and the convergence is monotone. Therefore the total variation distance between the two measures converges to 0 as well.

Simple statements. For statements with $P = P_*^{(u)}$, the claim is clear. The remaining cases are the following.

Sequential composition. For the statement $P = P_1; P_2$, we define $\nu_u := [\![P_*^{(u)}]\!]_\lightning(\mu) = [\![{P_1}_*^{(u)}]\!]_\lightning([\![{P_2}_*^{(u)}]\!]_\lightning(\mu))$. This is clearly monotone in u and $[\![P]\!]_\lightning(\mu)$ is an upper bound because

$$[\![P_*^{(u)}]\!]_\lightning(\mu) = [\![{P_1}_*^{(u)}]\!]_\lightning([\![{P_2}_*^{(u)}]\!]_\lightning(\mu)) \le [\![{P_1}_*^{(u)}]\!]_\lightning([\![P_2]\!]_\lightning(\mu)) \le [\![P_1]\!]_\lightning([\![P_2]\!]_\lightning(\mu)) = [\![P]\!]_\lightning(\mu)$$

To show that $[\![P]\!]_\lightning(\mu)$ is the least upper bound, let $\nu'_{u,v} := [\![{P_1}_*^{(u)}]\!]_\lightning([\![{P_2}_*^{(v)}]\!]_\lightning(\mu))$. We claim that any ν is an upper bound on $\{\nu'_{u,v} \mid u, v \in \mathbb{N}\}$ if and only if it is an upper bound on ν_u. The forward direction is immediate. For the reverse direction, let ν be an upper bound on ν_u for all $u \in \mathbb{N}$. Then ν is an upper bound on $\nu'_{u,v}$ for all $u, v \in \mathbb{N}$ because $\nu'_{u,v} \le \nu_{\max(u,v)} \le \nu$. By monotonicity, we have

$$\begin{aligned}
\sup_{u\in\mathbb{N}} [\![P_*^{(u)}]\!]_\lightning(\mu) &= \sup_{u\in\mathbb{N}} \nu_u \\
&= \sup_{u\in\mathbb{N}, v\in\mathbb{N}} \nu'_{u,v} \\
&= \sup_{u\in\mathbb{N}} \sup_{v\in\mathbb{N}} [\![{P_1}_*^{(u)}]\!]_\lightning([\![{P_2}_*^{(v)}]\!]_\lightning(\mu)) \\
&= \sup_{u\in\mathbb{N}} [\![{P_1}_*^{(u)}]\!]_\lightning([\![P_2]\!]_\lightning(\mu)) \\
&= [\![P_1]\!]_\lightning([\![P_2]\!]_\lightning(\mu)) \\
&= [\![P]\!]_\lightning(\mu)
\end{aligned}$$

Branching. For $P =$ if E {P_1} else {P_2}, we have $[\![P_*^{(u)}]\!]_\lightning(\mu) = \mu_\lightning + [\![{P_1}_*^{(u)}]\!]_\lightning(\mu|_E) + [\![{P_2}_*^{(u)}]\!]_\lightning(\mu|_{\neg E})$. This is clearly monotone in u by the induction hypothesis and by the

linearity of suprema, we find that the supremum is in fact $\mu_{\lightning} + [\![P_1]\!]_{\lightning}(\mu|_E) + [\![P_2]\!]_{\lightning}(\mu|_{\neg E}) = [\![P]\!]_{\lightning}(\mu)$.

Loops. Let $P = \text{while } E\,\{P_1\}$. By induction hypothesis $[\![P_{1*}^{(u)}]\!]_{\lightning}(\mu)$ is an ω-chain with supremum $[\![P_1]\!]_{\lightning}(\mu)$. We use the standard result that the least fixpoint $\mathrm{lfp}(\Phi_{E,P_1})$ is the supremum of the ω-chain $\Phi^n_{E,P_1}(\mathbf{0})$ in n. We first claim that $\Phi_{E,P_{1*}^{(u)}}(\psi)$ is an ω-chain in u for any ω-continuous ψ, with supremum $\Phi_{E,P_1}(\psi)$. This follows inductively from

$$\begin{aligned}
\sup_{u\in\mathbb{N}} \Phi_{E,P_{1*}^{(u)}}(\psi)(\mu) &= \mu_{\lightning} + \mu|_{\neg E} + \sup_{u\in\mathbb{N}} \psi([\![P_{1*}^{(u)}]\!]_{\lightning}(\mu|_E)) \\
&\preceq \mu_{\lightning} + \mu|_{\neg E} + \psi(\sup_{u\in\mathbb{N}} [\![P_{1*}^{(u)}]\!]_{\lightning}(\mu|_E)) && \text{by } \omega\text{-continuity} \\
&\preceq \mu_{\lightning} + \mu|_{\neg E} + \psi([\![P_1]\!]_{\lightning}(\mu|_E)) && \text{by induction hypothesis} \\
&= \Phi_{E,P_1}(\psi)(\mu)
\end{aligned}$$

We thus find:

$$\begin{aligned}
\sup_{u\in\mathbb{N}} [\![P_*^{(u)}]\!]_{\lightning} &= \sup_{u\in\mathbb{N}} \sup_{n\in\mathbb{N}} \Phi^n_{E,P_{1*}^{(u)}}(\mathbf{0}) && \text{characterization of least fixpoint} \\
&= \sup_{n\in\mathbb{N}} \sup_{u\in\mathbb{N}} \Phi^n_{E,P_{1*}^{(u)}}(\mathbf{0}) && \text{swap suprema} \\
&= \sup_{n\in\mathbb{N}} \Phi^n_{E,P_1}(\mathbf{0}) && \text{above claim} \\
&= \mathrm{lfp}(\Phi_{E,P_1})(\mathbf{0}) && \text{characterization of least fixpoint} \\
&= [\![P]\!]_{\lightning}(\mu) && \square
\end{aligned}$$

Note that the lower bound semantics can employ any exact method for the loop-free fragment (e.g. Genfer from Chapter 4) to find lower bounds for programs with loops. Jansen et al. [2016] follow a related approach: they combine unrolling with probabilistic model checking to obtain lower bounds on reachability probabilities and expectations.

5.3 Upper Bounds via Residual Mass

Upper bounds are a more challenging problem than lower bounds. The latter are easy because $\mathbf{0}$ is clearly a lower bound that can be used as a starting point for the fixed point iteration. This strategy does not work for upper bounds since there is no such obvious starting point.

Instead, a simple idea is to use the fact that the total mass of the state distribution can only decrease after program execution (Lemma 2.3.2). In other words, for a program P with initial distribution $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$, the total mass of the distribution after running P is $[\![P]\!]_{\lightning}(\mu)(\mathbb{S}_{\lightning}) \leq \mu(\mathbb{S}_{\lightning})$. So the part of the distribution that we miss in the lower bounds by cutting loops short has a probability mass bounded by the distance to the total mass $\mu(\mathbb{S}_{\lightning})$ at the start. We call this gap the ***residual mass*** of the program P with initial measure μ and define it as

$$[\![P]\!]_{\mathrm{res}}(\mu) := \mu(\mathbb{S}_{\lightning}) - [\![P]\!]_{\mathrm{lo}}(\mu)(\mathbb{S}_{\lightning}) \in \mathbb{R}_{\geq 0}$$

The ***residual mass semantics*** builds on the lower bound semantics and the concept of residual mass to bound probabilities $[\![P]\!]_{\lightning}(\mu)(S)$ of the program distribution for a measurable set $S \subseteq \mathbb{S}_{\lightning}$ by analyzing the gap $[\![P]\!]_{\lightning}(\mu)(S) - [\![P]\!]_{\mathsf{lo}}(\mu)(S)$. The gap is maximized if $S = \mathbb{S}_{\lightning}$ and thus bounded by $[\![P]\!]_{\lightning}(\mu)(\mathbb{S}_{\lightning}) - [\![P]\!]_{\mathsf{lo}}(\mu)(\mathbb{S}_{\lightning})$. Since $[\![P]\!]_{\lightning}(\mu)(\mathbb{S}_{\lightning}) \leq \mu(\mathbb{S}_{\lightning})$ by Lemma 2.3.2, this gap is always bounded by the residual mass.

The residual mass semantics can also bound the normalizing constant $1 - [\![P]\!]_{\lightning}(\mu)(\lightning)$ by setting $S := \{\lightning\}$ in the above argument. Thus it also yields bounds on the (normalized) posterior probabilities. The above soundness argument is made fully rigorous in the following theorem.

Theorem 5.3.1 (Soundness of the residual mass semantics)**.** *Let P be a program and $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$. Then for all measurable $S \subseteq \mathbb{S}_{\lightning}$, we can bound the unnormalized probabilities:*

$$[\![P]\!]_{\mathsf{lo}}(\mu)(S) \leq [\![P]\!]_{\lightning}(\mu)(S) \leq [\![P]\!]_{\mathsf{lo}}(\mu)(S) + [\![P]\!]_{\mathsf{res}}(\mu)$$

If μ is a probability measure and $S \subseteq \mathbb{S}$ measurable, we can bound the posterior probabilities:

$$\frac{[\![P]\!]_{\mathsf{lo}}(\mu)(S)}{1 - [\![P]\!]_{\mathsf{lo}}(\mu)(\lightning)} \leq \mathrm{normalize}([\![P]\!]_{\lightning}(\mu))(S) \leq \frac{[\![P]\!]_{\mathsf{lo}}(\mu)(S) + [\![P]\!]_{\mathsf{res}}(\mu)}{1 - [\![P]\!]_{\mathsf{lo}}(\mu)(\lightning) - [\![P]\!]_{\mathsf{res}}(\mu)} \tag{5.1}$$

Proof. For the first part, we note that

$$\begin{aligned}
[\![P]\!]_{\lightning}(\mu)(S) &= [\![P]\!]_{\mathsf{lo}}(\mu)(S) + ([\![P]\!]_{\lightning}(\mu) - [\![P]\!]_{\mathsf{lo}}(\mu))(S) \\
&\leq [\![P]\!]_{\mathsf{lo}}(\mu)(S) + ([\![P]\!]_{\lightning}(\mu) - [\![P]\!]_{\mathsf{lo}}(\mu))(\mathbb{S}_{\lightning}) \\
&\leq [\![P]\!]_{\mathsf{lo}}(\mu)(S) + (\mu(\mathbb{S}_{\lightning}) - [\![P]\!]_{\mathsf{lo}}(\mu)(\mathbb{S}_{\lightning})) \\
&= [\![P]\!]_{\mathsf{lo}}(\mu)(S) + [\![P]\!]_{\mathsf{res}}(\mu)
\end{aligned}$$

by monotonicity of measures and Lemma 2.3.2.

For the second part we use soundness of lower bounds and the residual mass bound and the inequality $[\![P]\!]_{\mathsf{lo}}(\mu)(\lightning) \leq [\![P]\!]_{\lightning}(\mu)(\lightning) \leq [\![P]\!]_{\mathsf{lo}}(\mu)(\lightning) + [\![P]\!]_{\mathsf{res}}(\mu)$:

$$\frac{[\![P]\!]_{\mathsf{lo}}(\mu)(S)}{1 - [\![P]\!]_{\mathsf{lo}}(\mu)(\lightning)} \leq \underbrace{\frac{[\![P]\!]_{\lightning}(\mu)(S)}{1 - [\![P]\!]_{\lightning}(\lightning)}}_{\mathrm{normalize}([\![P]\!]_{\lightning}(\mu))(S)} \leq \frac{[\![P]\!]_{\mathsf{lo}}(\mu)(S) + [\![P]\!]_{\mathsf{res}}(\mu)}{1 - [\![P]\!]_{\mathsf{lo}}(\mu)(\lightning) - [\![P]\!]_{\mathsf{res}}(\mu)}$$

□

Like the lower bound semantics, the residual mass semantics can employ any exact inference method for the loop-free fragment, such as Genfer from Chapter 4. We illustrate the semantics with an example.

Example 5.3.2 (Residual mass semantics)**.** Consider again the "die paradox" program (Example 2.2.1) where the variables are $\boldsymbol{X} := (\mathit{Throws}, \mathit{Die})$. After three unrollings, the lower bound distribution $\mu := [\![P^{(3)}]\!]_{\mathsf{lo}}(\mu_0)$ for the initial distribution $\mu_0 = \mathrm{Dirac}(\mathbf{0})$ is given by

$$\mu(\boldsymbol{x}) = \begin{cases} \frac{1}{2} + \frac{1}{3} \cdot \frac{1}{2} = \frac{2}{3} & \text{if } \boldsymbol{x} = \lightning \\ \frac{1}{6} & \text{if } \boldsymbol{x} = (1, 6) \\ \frac{1}{3} \cdot \frac{1}{6} = \frac{1}{18} & \text{if } \boldsymbol{x} = (2, 6) \\ 0 & \text{otherwise} \end{cases}$$

because the failure probability is $\frac{1}{2}$ (for $Die \in \{1,3,5\}$) and the second iteration is entered in a state other than $\lightning$ with probability $\frac{1}{3}$ (for $Die \in \{2,4\}$). The residual mass is given by $[\![P^{(3)}]\!]_{\mathrm{res}}(\mu_0) = \mu_0(\mathbb{S}_{\lightning}) - \mu(\mathbb{S}_{\lightning}) = 1 - (\frac{2}{3} + \frac{1}{6} + \frac{1}{18}) = \frac{1}{9}$. This yields the following bounds on $[\![P]\!]_{\lightning}(\mu_0)(s)$ for $s \in \mathbb{S}_{\lightning}$:

$$[\![P]\!]_{\lightning}(\mu_0)(\boldsymbol{x}) \in \begin{cases} [\frac{2}{3}, \frac{7}{9}] & \text{if } \boldsymbol{x} = \lightning \\ [\frac{1}{6}, \frac{5}{18}] & \text{if } \boldsymbol{x} = (1,6) \\ [\frac{1}{18}, \frac{1}{6}] & \text{if } \boldsymbol{x} = (2,6) \\ [0, \frac{1}{9}] & \text{otherwise} \end{cases}$$

Hence the normalizing constant $1 - [\![P]\!]_{\lightning}(\mu_0)$ is in $[\frac{2}{9}, \frac{1}{3}]$, which yields the following normalized bounds. They are not very tight but can be improved by increasing the unrolling depth.

$$\mathrm{normalize}([\![P]\!]_{\lightning}(\mu_0))(\boldsymbol{x}) \in \begin{cases} [\frac{1}{6} \cdot 3, \frac{5}{18} \cdot \frac{9}{2}] = [\frac{1}{2}, \frac{5}{4}] & \text{if } \boldsymbol{x} = (1,6) \\ [\frac{1}{18} \cdot 3, \frac{1}{6} \cdot \frac{9}{2}] = [\frac{1}{6}, \frac{3}{4}] & \text{if } \boldsymbol{x} = (2,6) \\ [0, \frac{1}{9} \cdot \frac{9}{2}] = [0, \frac{1}{2}] & \text{otherwise} \end{cases}$$

The residual mass bounds enjoy a stronger convergence property than the corresponding completeness theorem of GuBPI (Theorem 3.3.13). Whereas the latter requires a suitable choice of a possibly infinite number of interval traces, the residual mass bounds can be made arbitrarily tight just by unrolling the loops further:

Theorem 5.3.3 (Convergence of the residual mass semantics)**.** *Let P be a program terminating almost surely on a* finite *measure $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$. Then the residual mass $[\![P^{(u)}]\!]_{\mathrm{res}}(\mu)$ converges monotonically to 0 as $u \to \infty$. In particular, the bounds on $\mathrm{normalize}([\![P^{(u)}]\!]_{\lightning}(\mu))(S)$ from Eq. (5.1) converge to the true posterior probability as $u \to \infty$.*

Proof. The first part follows from:

$$\begin{aligned} [\![P^{(u)}]\!]_{\mathrm{res}}(\mu) &= \mu(\mathbb{S}_{\lightning}) - [\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(\mathbb{S}_{\lightning}) \\ &= [\![P]\!]_{\lightning}(\mu)(\mathbb{S}_{\lightning}) - [\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(\mathbb{S}_{\lightning}) && \text{by almost sure termination} \\ &\to 0 && \text{by Theorem 5.2.3} \end{aligned}$$

Together with $[\![P^{(u)}]\!]_{\mathrm{lo}}(\mu) \to [\![P]\!]_{\lightning}(\mu)$ (by Theorem 5.2.3), this immediately implies the second part:

$$\frac{[\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(S)}{1 - [\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(\lightning)} \to \frac{[\![P]\!]_{\lightning}(\mu)(S)}{1 - [\![P]\!]_{\lightning}(\mu)(\lightning)} = \mathrm{normalize}([\![P]\!]_{\lightning}(\mu))(S)$$

$$\frac{[\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(S) + [\![P^{(u)}]\!]_{\mathrm{res}}(\mu)}{1 - [\![P^{(u)}]\!]_{\mathrm{lo}}(\mu)(\lightning) - [\![P^{(u)}]\!]_{\mathrm{res}}(\mu)} \to \frac{[\![P]\!]_{\lightning}(\mu)(S)}{1 - [\![P]\!]_{\lightning}(\mu)(\lightning)} = \mathrm{normalize}([\![P]\!]_{\lightning}(\mu))(S) \qquad \square$$

5.4 Implementation

For a practical implementation, there is the question of what exact inference method for the loop-free fragment to use, and in particular, how to represent the distributions.

Table 5.1: Extended generating function semantics for statements

Statement P	Extended GF semantics $[\![P]\!]^{\lightning}_{\text{gf}}(G, g)$
skip	(G, g)
$P_1; P_2$	$[\![P_2]\!]^{\lightning}_{\text{gf}}([\![P_1]\!]^{\lightning}_{\text{gf}}(G, g))$
$X_k := \mathbf{a}^\top \boldsymbol{X} + c$	$([\![X_k := \mathbf{a}^\top \boldsymbol{X} + c]\!]_{\text{gf}}(G), g)$
$X_k \dot{-}= 1$	$([\![X_k \dot{-}= 1]\!]_{\text{gf}}(G), g)$
if E $\{P_1\}$ else $\{P_2\}$	$(G_1 + G_2, g + g_1 + g_2)$ where $[\![P_1]\!]^{\lightning}_{\text{gf}}(G\|_E, 0) = (G_1, g_1)$ and $[\![P_2]\!]^{\lightning}_{\text{gf}}(G\|_{\neg E}, 0) = (G_2, g_2)$
$X_k \sim D$	$([\![X_k \sim D]\!]_{\text{gf}}(G), g)$
$X_k \sim D(X_j)$	$([\![X_k \sim D(X_j)]\!]_{\text{gf}}, g)$
fail	$(\mathbf{0}, G(\mathbf{1}_n) + g)$
Normalization	$\text{normalize}(G, g) := \frac{G}{1-g}$

For discrete SGCL, all distributions are finite and discrete (assuming all variables are initially zero), so can be represented and manipulated using arrays of probability masses. For full SGCL, we use the generating function representation and inference method from Chapter 4.

5.4.1 Discrete SGCL

The state space of *discrete SGCL* programs is $\mathbb{N}^n_{\lightning} = \mathbb{N}^n \cup \{\lightning\}$ (see Section 2.2.1). Furthermore, loop-free programs can only reach a finite number of states if the initial distribution is Dirac$(\mathbf{0}_n)$. Therefore, the distribution of loop-free discrete SGCL programs has finite support $\subseteq [d_1] \times \cdots \times [d_n]$. (Recall that $[d]$ stands for $\{0, \ldots, d-1\}$.) Its probability mass function $\mathbb{N}^n_{\lightning} \to [0, 1]$ thus takes only finitely many nonzero values and can be represented as a tensor in $\mathbb{R}^{[d_1] \times \cdots \times [d_n]}$.

The residual mass semantics with this representation of distributions is implemented in a tool called ***Diabolo*** ("**Dia**crete **Dis**tribution **A**nalysis via **Bo**unds, supporting **L**oops and **O**bservations"). Diabolo takes a discrete SGCL program as input and outputs bounds on probability masses, moments, and tail asymptotics of the posterior distribution of a specified program variable. The quality of the bounds and the computational effort can be controlled with an option to set the loop unrolling limit u. Diabolo is implemented in Rust [Matsakis and Klock 2014] for performance and uses exact rational number computations instead of floating point to ensure the correctness of the results. The code is archived on Zenodo [Zaiser 2024] and the latest version is available on Github: `https://github.com/fzaiser/diabolo`. Diabolo also implements the geometric bound semantics from Chapter 6, see Section 6.6.

5.4.2 Full SGCL with Generating Functions

For full SGCL programs, we leverage generating functions to represent the distributions as in Chapter 4. In the context of exact inference, we built the generating function semantics on the simplified standard semantics on the state space $\mathbb{S} = \mathbb{R}^n_{\geq 0}$. However,

for the residual mass semantics, we need to track the probability of failure, so we have to work on the extended state space $\mathbb{S}_{\lightning}$.

Fortunately, it is easy to extend the GF semantics $[\![-]\!]_{\mathrm{gf}}$ to an ***extended GF semantics*** $[\![-]\!]^{\lightning}_{\mathrm{gf}}$ operating on pairs (G, g) of a generating function G and the probability g of the failure state $\lightning$. The definition of $[\![-]\!]^{\lightning}_{\mathrm{gf}}$ (Table 5.1) follows the same structure regarding the failure probabilities as the standard semantics (Table 2.2b).

Theorem 5.4.1. *Let P be a loop-free SGCL program and $\mu \in \mathrm{Meas}(\mathbb{S}_{\lightning})$ a finite measure. Let $\mu' := [\![P]\!]_{\lightning}(\mu)$ be the transformed measure. Then*

$$[\![P]\!]^{\lightning}_{\mathrm{gf}}(\mathrm{gf}(\mu|_{\mathbb{S}}), \mu(\lightning)) = (\mathrm{gf}(\mu'|_{\mathbb{S}}), \mu'(\lightning))$$

Furthermore, if $\mathrm{gf}(\mu|_{\mathbb{S}})$ is defined on $T(R, \mu)$ (as defined in Section 4.2.2) for some $R > 1$ then $\mathrm{gf}(\mu'|_{\mathbb{S}})$ is defined on $T(R', \mu')$ for some $R' > 0$.

Proof. The proof is a straightforward induction on the structure of the program P, based on the correctness of $[\![-]\!]_{\mathrm{gf}}$ (Theorem 4.2.3). □

We extended the *Genfer* tool (Section 4.3.4) with experimental support for the residual mass semantics. The code is available at github.com/fzaiser/genfer. The tool takes a probabilistic program as input and outputs bounds on probability masses and moments of the posterior distribution of a specified program variable. The quality of the bounds can be controlled with an option for the loop unrolling limit u. The remaining options are inherited from Genfer (Section 4.3.4).

5.4.3 Overapproximating the Measure Support

The residual mass bound is a bound on the total mass of the distribution $\nu := [\![P]\!](\mu) - [\![P]\!]_{\mathrm{lo}}(\mu)$. In practice, ν often has bounded support. This is useful for our bounds because we know that for measurable sets $S \subseteq \mathbb{S}_{\lightning}$ outside the support of ν, the lower bound is tight: $[\![P]\!]_{\mathrm{lo}}(\mu)(S) = [\![P]\!]_{\lightning}(\mu)(S)$. Suppose the support of ν is contained in a Cartesian product of intervals $J := J_1 \times \cdots \times J_n$. Then for discrete variables X_k, the lower bound on the probability mass $\mathbb{P}[X_k = i]$ is exact for $i \notin J_k$. Even more, if J_k is bounded, then we can even compute upper bounds on moments:

$$\mathbb{E}_{X \sim [\![P]\!](\mu)}[X_k] - \mathbb{E}_{X \sim [\![P]\!]_{\mathrm{lo}}(\mu)}[X_k] = \mathbb{E}_{X \sim \nu}[X_k] \leq \nu(\mathbb{S}_{\lightning}) \cdot \max J_k \leq [\![P]\!]_{\mathrm{res}}(\mu) \cdot \max J_k$$

To compute such an overapproximation $J := J_1 \times \cdots \times J_n$ of the support of ν, we use the standard technique of abstract interpretation with interval arithmetic. Since the lower bound ignores loops, computing J amounts to bounding the range of values of variables after loops. Such a program analysis is standard, so we describe it only briefly. In fact, it can be viewed as a much simpler, imperative version of the interval type system in Section 3.4 without the need to handle higher-order functions or weighting.

As the abstract domain, we use products of intervals $I_1 \times \cdots \times I_n$ where each I_j is an interval $[a_j, b_j]$ with $a_j \in \mathbb{N}$ and $b_j \in \mathbb{N} \cup \{\infty\}$. This domain is extended with a special element $\bot = \varnothing \times \cdots \times \varnothing$ representing an empty support, i.e. the zero measure. The join operation is defined as the pointwise join of the intervals. The abstract versions

Table 5.2: Comparison of the running time of GuBPI [Beutner et al. 2022a] and our residual mass semantics (as implemented in Diabolo) to produce the same bounds

Benchmark	GuBPI	Residual mass semantics	Speedup factor
Simple counter (Example 6.4.2)	0.7 s	0.006 s	$1.2 \cdot 10^2$
Asymmetric random walk (Example 5.1.1)	90 s	0.002 s	$4.5 \cdot 10^4$
Mossel's die paradox (Example 2.2.1)	156 s	0.0008 s	$2 \cdot 10^5$

of addition and subtraction are standard. As the widening operator $\triangledown$ on intervals, we use $[a,b] \triangledown [c,d] = [e,f]$ with $e = a$ if $a \leq c$ else $e = 0$, and $f = b$ if $b \geq d$ else $f = \infty$. This is extended pointwise to products of intervals.

Example 5.4.2. For the asymmetric random walk example (Example 5.1.1), the interval analysis finds that the support of $\nu := [\![P^{(u)}]\!](\mathrm{Dirac}(0,0)) - [\![P^{(u)}]\!]_{\mathrm{lo}}(\mathrm{Dirac}(0,0))$ is contained in $[0,0] \times [u,\infty]$. This tells us that the lower bounds on the probability masses $\mathbb{P}[X_2 = i]$ are exact for $i < u$.

5.5 Comparison with Previous Work

Our residual mass semantics shares many characteristics with GuBPI from Chapter 3: both support infinite-support distributions, conditioning, and can bound posterior probabilities, but not moments or tails (see also Table 6.1). However, GuBPI is designed for a more general setting of SPCF programs with nonlinear functions and soft conditioning. As a consequence, when applied to much more restricted SGCL programs, there is a lot of overhead compared to the residual mass semantics designed specifically for SGCL. Another advantage of the method presented in this chapter is that it does not suffer from the path explosion problem. The reason is that it does not split the trace space or use symbolic execution, but directly works with bounds on the program distribution. Finally, its convergence theorem (Theorem 5.3.3) is stronger than the completeness result for GuBPI (Theorem 3.3.13).

Experimental comparison To demonstrate the performance advantage in practice, we ran both GuBPI and the residual mass semantics as implemented in Diabolo on Examples 6.4.2, 5.1.1 and 2.2.1, configured to produce the same bounds. All benchmarks and code to reproduce the results are available [Zaiser 2024]. The results (Table 5.2) show that Diabolo is several orders of magnitude faster than GuBPI.

P. Wang et al. [2024] subsequently improved on GuBPI's results (see Section 3.7), but we were unable to compare our work with their system because they use proprietary software. However, they report performance improvements of at most a factor of 15 over GuBPI, which is insufficient to bridge the factor of over 100 for Example 6.4.2 between GuBPI and our residual mass semantics, let alone the factor of over 10^5 for Example 2.2.1. This demonstrates that the residual mass semantics is more effective for SGCL programs than existing tools.

5.6 Conclusion

Our residual mass semantics makes it easy to extend an exact inference system for loop-free SGCL programs, such as Genfer (Chapter 4), to compute guaranteed bounds for programs with loops. Compared to GuBPI (Chapter 3), this approach is less broadly applicable, but is simpler, faster (see Section 5.5), and has stronger convergence guarantees (Theorem 5.3.3).

Future work Since the residual mass is just a thin layer on top of an exact inference method for loop-free programs, it benefits immensely from improvements to the latter. For discrete SGCL (Section 5.4.1), it would be interesting to build upon a highly optimized exact inference system for finite discrete distributions like Dice [Holtzen et al. 2020]. For full SGCL (Section 5.4.2), the potential improvements to Genfer outlined in Chapter 4 could similarly improve the efficiency of the residual mass semantics.

The residual mass semantics for discrete SGCL can easily be extended to handle variables in $\mathbb{Z}$ instead of $\mathbb{N}$ since it only requires an exact semantics for the loop-free fragment, which is straightforward for distributions with finite support. Lifting restrictions on continuous variables in full SGCL would be desirable but very difficult because continuous distributions are already hard to represent exactly in the loop free fragment.

6

Guaranteed Bounds via Eventually Geometric Distributions

Contents

6.1 Introduction . . . 141
6.1.1 Contributions . . . 142
6.1.2 Limitations . . . 143
6.2 Contraction Invariants . . . 143
6.3 Eventually Geometric Distributions (EGDs) . . . 144
6.4 Geometric Bound Semantics . . . 149
6.4.1 Examples . . . 152
6.5 Theoretical Analysis . . . 157
6.5.1 Decidability . . . 157
6.5.2 Properties of the Solution Set . . . 158
6.5.3 Soundness . . . 159
6.5.4 Loop-free Fragment . . . 162
6.5.5 Sufficient Conditions for the Existence of Bounds . . . 163
6.5.6 Necessary Conditions for the Existence of Bounds . . . 166
6.5.7 Convergence . . . 168
6.6 Implementation . . . 176
6.6.1 Constraint Solving . . . 177
6.6.2 Optimization . . . 178
6.6.3 Our ADAM-based Solver and Optimizer . . . 179
6.7 Empirical Evaluation . . . 179
6.7.1 Applicability of the Geometric Bound Semantics . . . 179
6.7.2 Quality of the Geometric Bounds . . . 182
6.7.3 Comparison with Previous Work . . . 182
6.7.4 Comparison with Residual Mass Bounds . . . 185
6.8 Related Work . . . 187
6.9 Conclusion . . . 188

Summary This chapter proposes the *geometric bound semantics* as a novel approach to compute guaranteed bounds on the posterior distribution of discrete SGCL programs. Compared to Chapter 5, this method is more sophisticated and expensive, but yields much more informative bounds if applicable. It operates on a novel family of distributions, called *eventually geometric distributions* (EGDs), and can bound the distribution of loops with a new form of loop invariants called *contraction invariants*. The invariant synthesis problem reduces to a system of polynomial inequality constraints, which is a decidable problem with automated solvers. If a solution exists, it yields an exponentially decreasing bound on the *whole* distribution, and can therefore find upper bounds on moments and tail asymptotics as well, not just on probabilities as in previous chapters.

The semantics enjoys desirable theoretical properties, such as soundness and convergence to the exact posterior as loops are unrolled further. We also investigate sufficient and necessary conditions for the existence of geometric bounds. On the practical side, we describe a fully-automated implementation of the semantics and evaluate it on a variety of benchmarks from the literature, demonstrating its general applicability and the utility of the resulting bounds.

Publication Like Chapter 5, this chapter is based on a paper accepted for publication at the *52nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2025)*. The full reference is:

> Fabian Zaiser, Andrzej S. Murawski, and Luke Ong [2025]. "Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops". In: *Proc. ACM Program. Lang.* 9.POPL. DOI: 10.1145/3704874

Supplementary material for the paper is available in Zaiser et al. [2024].

Author contributions I contributed the idea, the theoretical development, the implementation, the empirical evaluation, and the writing. Andrzej Murawski and Luke Ong supervised the project, provided feedback, and helped revise the draft.

6.1 Introduction

None of the methods from previous chapters can find nontrivial upper bounds on the moments of variables with unbounded support. Consider again Mossel's die paradox (Example 2.2.1), which asks for the expected number of the variable *Throws* in the following SGCL program.

$$
\begin{aligned}
&\mathit{Throws} := 0;\\
&\mathit{Die} := 0;\\
&\textsf{while}\ \mathit{Die} \neq 6\ \{\\
&\quad \mathit{Die} \sim \mathsf{Uniform}\{1, \ldots, 6\};\\
&\quad \textsf{observe}\ \mathit{Die} \in \{2, 4, 6\};\\
&\quad \mathit{Throws} \mathrel{+}= 1\\
&\}
\end{aligned}
$$

Table 6.1: Comparison of our approaches with the most relevant related work on probabilistic programs with loops. (Cond.: supports (Bayesian) conditioning; Inf.: allows branching on variables with infinite support; Cont.: allows continuous distributions; Auto.: fully automated; Prob.: computes/bounds probabilities; Mom.: computes/bounds moments; Tails: computes/bounds tail asymptotics of this shape; A.a. (always applicable): yields a result for any program expressible in the respective language. Partial support is denoted by "~".)

	Type	Cond.?	Inf.?	Cont.?	Auto.?	Prob.?	Mom.?	Tails?	A.a.?
Moosbrugger et al. [2022]	exact	✗	✗	✓	✓	~	✓	$O(n^{-k})$	✗
GuBPI (Chapter 3)	bounds	✓	✓	✓	✓	✓	✗	✗	✓
P. Wang et al. [2024]	bounds	✓	✓	✓	✓	✓	✗	✗	✗
Klinkenberg et al. [2024]	exact	✓	✓	✗	✗	✓	✓	✗	✗
Resid. mass sem. (Chapter 5)	bounds	✓	✓	✗	✓	✓	✗	✗	✓
Geom. bounds (Chapter 6)	bounds	✓	✓	✗	✓	✓	✓	$O(c^n)$	✗

Both GuBPI (Chapter 3) and the residual mass semantics (Chapter 5) can be used to bound the posterior probabilities $\mathbb{P}[\mathit{Throws} = n]$, but since *Throws* can take any positive integer value, this only yields the trivial upper bound of ∞ on the expected value of *Throws*. None of the formal methods from Section 2.6 can be applied either as they do not support conditioning. The approach by Klinkenberg et al. [2024] via generating functions (see Section 2.4) is theoretically applicable, but requires the user to provide a loop invariant template. This presupposes that the user already deeply understands the program distribution, so it is not a satisfactory solution.

Problem statement Given a probabilistic program with posterior distribution μ on $\mathbb{N}$, our goal is to bound:

1. *probability masses:* given $n \in \mathbb{N}$, find $l, u \in [0, 1]$ such that $l \leq \mathbb{P}_{X\sim\mu}[X = n] \leq u$;
2. *moments:* given $k \in \mathbb{N}$, find $l, u \in \mathbb{R}_{\geq 0}$ such that $l \leq \mathbb{E}_{X\sim\mu}[X^k] \leq u$;
3. *tail asymptotics:* find $c \in [0, 1)$ such that $\mathbb{P}_{X\sim\mu}[X = n] = O(c^n)$.

6.1.1 Contributions

In this chapter, we develop a new method to compute guaranteed bounds on the distribution of discrete SGCL programs (see Section 2.2.1). As before, upper bounds are the main challenge because lower bounds can be found by unrolling each loop a finite number of times. To address this, we introduce the ***geometric bound semantics*** (Section 6.4). The idea is to bound the distribution of loops in a more fine-grained manner with geometric tails, rather than a flat bound as in Chapter 5. A comparison of this approach with the most relevant related work is presented in Table 6.1 and a detailed account can be found in Section 6.8. Specifically, our contributions are as follows:

- We present the concept of a *contraction invariant* for a loop, which yields upper bounds on the distribution (Section 6.2).
- We introduce a family of distributions called *eventually geometric distributions (EGDs)* that are used as an abstract domain to overapproximate the distribution of a loop (Section 6.3).

- We present the *geometric bound semantics* (Section 6.4) which reduces the synthesis problem of such a contraction invariant to solving a system of polynomial inequalities. If successful, it immediately yields bounds on probability masses, moments, and tail probabilities of the program distribution.
- We prove soundness of the semantics and convergence of the bounds, as loops are unrolled further and further (Section 6.5).
- We identify necessary conditions and sufficient conditions for the existence of geometric bounds (Section 6.5).
- We fully automate it in our tool *Diabolo* (Section 6.6): it does not rely on the user to provide a loop invariant (template).
- We demonstrate its applicability on a large proportion of benchmarks from the literature and compare it with previous approaches (Section 6.7).

6.1.2 Limitations

Our work deals with *discrete* SGCL programs, i.e. programs cannot sample or observe from continuous distributions. The initial values of the program variables are fixed: our methods cannot reason parametrically about these inputs.

The geometric bound semantics aims to find EGD bounds, which allow bounding moments and tails. However, such EGD bounds do not exist for all programs because the moments of the program distribution may be infinite. Due to the Turing completeness of discrete SGCL, it is also possible that our method cannot find an EGD bound even though one exists. Our experiments show that this is not a big concern for many probabilistic programs with loops in practice: EGD bounds can be derived for a large majority of examples we found in the literature.

Another limitation of EGD bounds is that they cannot represent correlation of the tails of two variables, which may lead to imprecise tail bounds or failing to find bounds at all.

Finally, solving the system of polynomial inequalities arising from the semantics, while decidable, can be hard in practice and does not scale to very large programs. It should be noted that scalability is a general issue in probabilistic program analysis owing to the hardness of the problem [Dagum and Luby 1993] and not specific to our work.

However, in many cases, EGD bounds can be found that also yield bounds on moments and tails for distributions with infinite support. In the following sections, we describe the key ideas that make this possible.

6.2 Contraction Invariants

If a loop $L = \textsf{while}\, E\, \{B\}$ terminates almost surely on an initial distribution μ, all the probability mass that entered the loop must eventually exit the loop. Inspired by this observation, we consider the following assumption:

$$[\![B]\!]_{\lightning}(\mu|_E) \preceq c \cdot \mu \text{ with } 0 \leq c < 1 \tag{6.1}$$

In words: the probability distribution at the start of the next loop iteration decreases uniformly by a factor of c. This assumption looks very strong and is often violated

in practice, but if we assume for a minute that it holds, what can we derive? Recall the fixpoint equation of loops:

$$\begin{aligned}[\![\text{while } E\,\{B\}]\!]_{\lightning}(\mu) &= \mu|_{\lightning} + \mu|_{\neg E} + [\![\text{while } E\,\{B\}]\!]_{\lightning}([\![B]\!]_{\lightning}(\mu|_E)) \\ &\leq \mu|_{\lightning} + \mu|_{\neg E} + c \cdot [\![\text{while } E\,\{B\}]\!]_{\lightning}(\mu) \qquad \text{by Eq. (6.1) and linearity}\end{aligned}$$

By rearranging, we find the upper bound $[\![\text{while } E\,\{B\}]\!]_{\lightning}(\mu) \leq \frac{\mu|_{\lightning}+\mu|_{\neg E}}{1-c}$.

Unfortunately, the initial distribution μ will usually not satisfy Eq. (6.1) directly. But we may be able to increase μ to $\nu \geq \mu$ such that ν satisfies $[\![B]\!]_{\lightning}(\nu|_E) \leq c \cdot \nu$ with $0 \leq c < 1$. We then call ν a ***contraction invariant*** or ***c-contraction invariant*** of μ.[1] Surprisingly, such a contraction invariant ν can often be found in practice (see Section 6.7.1), and we can then derive the following upper bound on the distribution after the loop:

$$[\![\text{while } E\,\{B\}]\!]_{\lightning}(\mu) \leq [\![\text{while } E\,\{B\}]\!]_{\lightning}(\nu) \leq \frac{\nu|_{\lightning} + \nu|_{\neg E}}{1-c}$$

6.3 Eventually Geometric Distributions (EGDs)

How can we find such a contraction invariant ν? We clearly have to restrict the candidate set for ν in some way. Specifically, we consider a class of distributions that generalize geometric distributions. Recall that a Geometric(β) distribution has probability masses $p(i) = \beta(1-\beta)^i \propto (1-\beta)^i$ with a decay rate of $1-\beta$. For the multivariate setting, we consider products of independent geometric distributions, but with a twist: probabilities of small values are allowed to differ for additional flexibility. This section makes extensive use of vector and tensor notations as described in Section 1.9.

Definition 6.3.1. Let $\mathbf{P} \in \mathbb{R}_{\geq 0}^{[d_1+1]\times\cdots\times[d_n+1]}$ be an n-dimensional tensor and $\boldsymbol{\alpha} \in [0,1)^n$. The n-dimensional ***eventually geometric distribution (EGD)*** with ***initial block*** $\mathbf{P}$ and ***decay rates*** $\boldsymbol{\alpha}$, written $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \in \mathrm{Meas}(\mathbb{N}^n)$ is defined by the following mass function (for $\boldsymbol{i} = (i_1, \ldots, i_n) \in \mathbb{N}^n$):

$$\begin{aligned}\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})(\boldsymbol{i}) &= \mathbf{P}_{\min(\boldsymbol{i}, |\mathbf{P}|-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}^{\max(\boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n, \mathbf{0}_n)} \\ &= \mathbf{P}_{\min(i_1,d_1),\ldots,\min(i_n,d_n)} \cdot \alpha_1^{\max(i_1-d_1,0)} \cdots \alpha_n^{\max(i_n-d_n,0)}\end{aligned}$$

Note that an EGD is not necessarily a probability measure. One can think of the distribution $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ as a scaled product of independent Geometric$(1-\alpha_i)$ distributions, except it may differ in a finite "prefix" of size d_j in each dimension j. Outside this initial block, the probability masses are extended like in a geometric distribution with decay rate α_j in each dimension j. In other words: *eventually*, for values $i_j > d_j$, the j-th component of $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ behaves like a Geometric$(1-\alpha_j)$ distribution; hence the name. They can be seen as a subclass of multivariate discrete phase-type distributions [Campillo Navarro 2018]. (Discrete phase-type distributions [Neuts 1975] describe the absorption time in a Markov chain with one absorbing state [Bladt and Nielsen 2017, Section 1.2.6].)

[1] The name is inspired by contraction mappings in mathematics where they describe functions $f : X \to Y$ of metric spaces with $d(f(x), f(y)) \leq c \cdot d(x, y)$ with $c < 1$.

During the early development of our work, EGDs were motivated by the special shape of their generating functions, which take the form:

$$\frac{p(\boldsymbol{x})}{(1-\alpha_1 x_1)\cdots(1-\alpha_n x_n)}$$

where p is a polynomial in n variables and $\boldsymbol{\alpha} \in [0,1)^n$. The relationship between EGDs and generating functions is further explored in Appendix C.1.

Example 6.3.2. Consider the two-dimensional EGD given by $\mathrm{EGD}(\mathbf{P}, (\alpha_1, \alpha_2))$ with $\mathbf{P} \in \mathbb{R}_{\geq 0}^{[2]\times[2]}$, i.e. $\mathbf{P}$ is a matrix $\mathbf{P} = \begin{pmatrix} \mathbf{P}_{0,0} & \mathbf{P}_{0,1} \\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} \end{pmatrix}$. Its probability masses are given in the following table:

$\mathrm{EGD}(\mathbf{P}, (\alpha_1, \alpha_2))(x_1, x_2)$	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$	…	$x_2 = j$
$x_1 = 0$	$\mathbf{P}_{0,0}$	$\mathbf{P}_{0,1}$	$\mathbf{P}_{0,1} \cdot \alpha_2$	…	$\mathbf{P}_{0,1} \cdot \alpha_2^{j-1}$
$x_1 = 1$	$\mathbf{P}_{1,0}$	$\mathbf{P}_{1,1}$	$\mathbf{P}_{1,1} \cdot \alpha_2$	…	$\mathbf{P}_{1,1} \cdot \alpha_2^{j-1}$
$x_1 = 2$	$\mathbf{P}_{1,0} \cdot \alpha_1$	$\mathbf{P}_{1,1} \cdot \alpha_1$	$\mathbf{P}_{1,1} \cdot \alpha_1\alpha_2$	…	$\mathbf{P}_{1,1} \cdot \alpha_1 \cdot \alpha_2^{j-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$x_1 = i$	$\mathbf{P}_{1,0} \cdot \alpha_1^{i-1}$	$\mathbf{P}_{1,1} \cdot \alpha_1^{i-1}$	$\mathbf{P}_{1,1} \cdot \alpha_1^{i-1} \cdot \alpha_2$	…	$\mathbf{P}_{1,1} \cdot \alpha_1^{i-1} \cdot \alpha_2^{j-1}$

Here we can see that after the initial block of size 2×2, the probability masses are extended with decay rate α_1 in the first dimension and α_2 in the second dimension.

Why did we pick EGDs as the shape of our bounds? There are several reasons.

1. *Interpretability:* EGDs can easily be understood as geometric distributions, where the probability masses for small values (up to some threshold) have been modified. The relationship with the geometric distribution also makes it easier to compute its moments and the tail asymptotics can be read directly off the parameter $\boldsymbol{\alpha}$ (see Lemma 6.3.4 below).
2. *Expressiveness:* the reason we allow the "start" of the distribution to deviate from geometric distributions is necessary for flexibility. If only exact geometric distributions were allowed, typical program operations like increasing or decreasing a variable (i.e. shifting the geometric distribution) could not be represented precisely enough and the whole approach would fail.
3. *Tractability:* a sufficient condition for the order $\preceq$ is easy to check for EGDs, as we will see, because the probability masses follow a simple pattern. In fact, this condition is equivalent to the satisfiability of a system of polynomial inequalities. If we had based our semantics on an extension of, say, negative binomial distributions (of which geometric distributions are a special case), deciding the order would be harder because binomial coefficients start appearing in the probability masses.

As evidence for interpretability and tractability, we show how to marginalize EGDs and how to compute their moments. This is also needed to extract bounds on the moments from EGD bounds.

Lemma 6.3.3 (Marginalizing EGDs). *Marginalizing out the k-th dimension from* $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ *yields* $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ *where*

$$\mathbf{Q} = \sum_{j=0}^{|\mathbf{P}|_k - 2} \mathbf{P}_{k:j} + \frac{\mathbf{P}_{k:|\mathbf{P}|_k - 1}}{1 - \alpha_k} \quad \text{and} \quad \boldsymbol{\beta} = (\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots \alpha_n)$$

Proof. For a measure μ on $\mathbb{N}^n$, marginalizing out the k-th dimension yields the measure μ' with

$$\mu'(\boldsymbol{x}[\hat{k}]) := \sum_{x_k \in \mathbb{N}} \mu(x_1, \ldots, x_n)$$

where $\boldsymbol{x}[\hat{k}]$ denotes the omission of x_k in $\boldsymbol{x}$. For EGDs, we find, as desired (using $|\mathbf{Q}| = |\mathbf{P}|[\hat{k}]$):

$$\begin{aligned}
&\sum_{j \in \mathbb{N}} \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})(\boldsymbol{x}[k \mapsto j]) \\
&= \sum_{j \in \mathbb{N}} \mathbf{P}_{\min(\boldsymbol{x}[k \mapsto j], |\mathbf{P}| - \mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}[k \mapsto j] - |\mathbf{P}| + \mathbf{1}_n)} \\
&= \sum_{j=0}^{\infty} \mathbf{P}_{\min(\boldsymbol{x}[k \mapsto j], |\mathbf{P}| - \mathbf{1}_n)} \alpha_k^{\max(0, j - |\mathbf{P}|_k + 1)} \boldsymbol{\alpha}[\hat{k}]^{\max(\mathbf{0}_{n-1}, \boldsymbol{x}[\hat{k}] - |\mathbf{Q}| + \mathbf{1}_{n-1})} \\
&= \left(\sum_{j=0}^{|\mathbf{P}|_k - 2} \mathbf{P}_{\min(\boldsymbol{x}[k \mapsto j], |\mathbf{P}| - \mathbf{1}_n)} + \sum_{j'=0}^{\infty} \mathbf{P}_{\min(\boldsymbol{x}[k \mapsto |\mathbf{P}|_k - 1], |\mathbf{P}| - \mathbf{1}_n)} \alpha_k^{j'} \right) \boldsymbol{\alpha}[\hat{k}]^{\max(\mathbf{0}_{n-1}, \boldsymbol{x}[\hat{k}] - |\mathbf{Q}| + \mathbf{1}_{n-1})} \\
&= \left(\sum_{j=0}^{|\mathbf{P}|_k - 2} \mathbf{P}_{\min(\boldsymbol{x}[k \mapsto j], |\mathbf{P}| - \mathbf{1}_n)} + \frac{\mathbf{P}_{\min(\boldsymbol{x}[k \mapsto |\mathbf{P}|_k - 1], |\mathbf{P}| - \mathbf{1}_n)}}{1 - \alpha_k} \right) \boldsymbol{\alpha}[\hat{k}]^{\max(\mathbf{0}_{n-1}, \boldsymbol{x}[\hat{k}] - |\mathbf{Q}| + \mathbf{1}_{n-1})} \\
&= \left(\sum_{j=0}^{|\mathbf{P}|_k - 2} (\mathbf{P}_{k:j})_{\min(\boldsymbol{x}[\hat{k}], |\mathbf{Q}| - \mathbf{1}_n)} + \frac{(\mathbf{P}_{k:|\mathbf{P}|_k - 1})_{\min(\boldsymbol{x}[\hat{k}], |\mathbf{Q}| - \mathbf{1}_{n-1})}}{1 - \alpha_k} \right) \boldsymbol{\alpha}[\hat{k}]^{\max(\mathbf{0}_{n-1}, \boldsymbol{x}[\hat{k}] - |\mathbf{Q}| + \mathbf{1}_{n-1})} \\
&= \mathbf{Q}_{\min(\boldsymbol{x}[\hat{k}], |\mathbf{Q}| - \mathbf{1}_n)} \boldsymbol{\alpha}[\hat{k}]^{\max(\mathbf{0}_{n-1}, \boldsymbol{x}[\hat{k}] - |\mathbf{Q}| + \mathbf{1}_{n-1})} \\
&= \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})(\boldsymbol{x}[\hat{k}])
\end{aligned}$$

□

Lemma 6.3.4 (Moments and tails of EGDs). *Let* $\mathrm{EGD}(\mathbf{P}, \alpha)$ *be a one-dimensional EGD with* $\mathbf{P} \in \mathbb{R}_{\geq 0}^{[d+1]}$. *Then its tail asymptotics is* $\mathrm{EGD}(\mathbf{P}, \alpha)(n) = O(\alpha^n)$ *and its k-th moment is*

$$\mathbb{E}_{X \sim \mathrm{EGD}(\mathbf{P}, \alpha)}[X^k] = \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \frac{\mathbf{P}_d}{1 - \alpha} \sum_{i=0}^{k} \binom{k}{i} d^{k-i} \mathbb{E}_{Y \sim \mathrm{Geometric}(1-\alpha)}[Y^i]$$

and can thus be computed from the k-th moment of a geometric distribution. In particular, the expected value is

$$\mathbb{E}_{X \sim \mathrm{EGD}(\mathbf{P}, \alpha)}[X] = \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j + \frac{\mathbf{P}_d}{1 - \alpha} \left(d + \frac{\alpha}{1 - \alpha} \right)$$

Proof. The tail asymptotics are immediate from the definition of EGDs. For the k-th moment, we find:

$$\begin{aligned}
\mathbb{E}_{X\sim\mathsf{EGD}(\mathbf{P},\alpha)}[X^k] &= \sum_{j=0}^{\infty} \mathsf{EGD}(\mathbf{P},\alpha)(j)\cdot j^k \\
&= \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \sum_{j=d}^{\infty} \mathbf{P}_d \cdot \alpha^{j-d}\cdot j^k \\
&= \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \frac{\mathbf{P}_d}{1-\alpha}\sum_{j=0}^{\infty}(1-\alpha)\alpha^j\cdot(j+d)^k \\
&= \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \frac{\mathbf{P}_d}{1-\alpha}\mathbb{E}_{Y\sim\mathsf{Geometric}(1-\alpha)}[(Y+d)^k] \\
&= \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \frac{\mathbf{P}_d}{1-\alpha}\mathbb{E}_{Y\sim\mathsf{Geometric}(1-\alpha)}\left[\sum_{i=0}^{k}\binom{k}{i}d^{k-i}Y^i\right] \\
&= \sum_{j=0}^{d-1} \mathbf{P}_j \cdot j^k + \frac{\mathbf{P}_d}{1-\alpha}\sum_{i=0}^{k}\binom{k}{i}d^{k-i}\mathbb{E}_{Y\sim\mathsf{Geometric}(1-\alpha)}[Y^i]
\end{aligned}$$

□

Expansion of EGDs When performing binary operations (e.g. comparison or addition) on EGDs, it will often be convenient to assume that their initial blocks have the same size. In fact, one can always expand the size of the initial block of an EGD without changing the distribution. For example, the distribution from Example 6.3.2 can equivalently be represented as:

$$\begin{aligned}
\mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} & \mathbf{P}_{0,1}\\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1}\end{pmatrix}, \boldsymbol{\alpha}\right) &= \mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} & \mathbf{P}_{0,1}\\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1}\\ \mathbf{P}_{1,0}\alpha_1 & \mathbf{P}_{1,1}\alpha_1\end{pmatrix}, \boldsymbol{\alpha}\right) \\
&= \mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} & \mathbf{P}_{0,1} & \mathbf{P}_{0,1}\alpha_2 & \mathbf{P}_{0,1}\alpha_2^2\\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & \mathbf{P}_{1,1}\alpha_2 & \mathbf{P}_{1,1}\alpha_2^2\end{pmatrix}, \boldsymbol{\alpha}\right)
\end{aligned}$$

More generally, the *expansion* $\mathsf{EGD}(\mathbf{Q},\boldsymbol{\alpha})$ of $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})$ is obtained by adding rows and columns with the appropriate factors of α_i.

Definition 6.3.5 (Expansion of EGDs)**.** The EGD $\mathsf{EGD}(\mathbf{Q},\boldsymbol{\alpha})$ is an ***expansion*** of $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})$ with $|\mathbf{P}| \le |\mathbf{Q}|$ (meaning $|\mathbf{P}|_i \le |\mathbf{Q}|_i$ for all $i = 1,\dots,n$) if

$$\begin{aligned}
\mathbf{Q}_{\boldsymbol{i}} &= \mathbf{P}_{\min(\boldsymbol{i},|\mathbf{P}|-\mathbf{1}_n)}\boldsymbol{\alpha}^{\max(\boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n,\mathbf{0}_n)} \\
&= \mathbf{P}_{\min(i_1,|\mathbf{P}|_1-1),\dots,\min(i_n,|\mathbf{P}|_n-1)}\alpha_1^{\max(i_1-|\mathbf{P}|_1+1,0)}\cdots\alpha_n^{\max(i_n-|\mathbf{P}|_n+1,0)}
\end{aligned}$$

Lemma 6.3.6 (Expansion preserves the measure)**.** *The expansion* $\mathsf{EGD}(\mathbf{Q},\boldsymbol{\alpha})$ *of* $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})$ *with* $|\mathbf{P}| \le |\mathbf{Q}|$ *describes the same measure as* $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})$.

Proof. We expand the definitions:

$$\mathsf{EGD}(\mathbf{Q},\boldsymbol{\alpha})(\boldsymbol{x}) = \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)}\boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)}$$

$$= \mathbf{P}_{\min(\boldsymbol{x},|\mathbf{P}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)-|\mathbf{P}|+\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)}$$
$$= \mathbf{P}_{\min(\boldsymbol{x},|\mathbf{P}|-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{P}|+\mathbf{1}_n)}$$

because

$$\begin{aligned}
&\max(\mathbf{0}_n, \min(\boldsymbol{x}, |\mathbf{Q}| - \mathbf{1}_n) - |\mathbf{P}| + \mathbf{1}_n) + \max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n) \\
&= \max(\mathbf{0}_n, \min(\boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n, \mathbf{0}_n) + |\mathbf{Q}| - |\mathbf{P}|) + \max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n) \\
&= \max(\max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n), \max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n) \\
&\quad + \min(\boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n, \mathbf{0}_n) + |\mathbf{Q}| - |\mathbf{P}|) \\
&= \max(\max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n), \boldsymbol{x} - |\mathbf{P}| + \mathbf{1}_n) \\
&= \max(\mathbf{0}_n, \boldsymbol{x} - |\mathbf{P}| + \mathbf{1}_n)
\end{aligned}$$

since $|\mathbf{P}| \leq |\mathbf{Q}|$ implies $\boldsymbol{x} - |\mathbf{Q}| + \mathbf{1}_n \leq \boldsymbol{x} - |\mathbf{P}| + \mathbf{1}_n$. □

Order of EGDs To decide $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ for $\mathbf{P}$ and $\mathbf{Q}$ of the same size $|\mathbf{P}| = |\mathbf{Q}|$, one might hope that this would be equivalent to $\mathbf{P} \leq \mathbf{Q}$ and $\boldsymbol{\alpha} \leq \boldsymbol{\beta}$ (both elementwise), because such a simple conjunction of inequalities would be easy to check. To extend this idea to EGDs $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ of different sizes, we first have to expand them to the same size $\max(|\mathbf{P}|, |\mathbf{Q}|)$. Inlining the definition of expansion, we obtain the following order.

Definition 6.3.7 (Order of EGDs)**.** The order $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is defined to hold if and only if for all $\boldsymbol{i} < \max(|\mathbf{P}|, |\mathbf{Q}|)$ (using the notation explained in Section 1.9):

$$\boldsymbol{\alpha} \leq \boldsymbol{\beta} \;\wedge\; \mathbf{P}_{\min(\boldsymbol{i},|\mathbf{P}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n,\mathbf{0}_n)} \leq \mathbf{Q}_{\min(\boldsymbol{i},|\mathbf{Q}|-\mathbf{1}_n)} \boldsymbol{\beta}^{\max(\boldsymbol{i}-|\mathbf{Q}|+\mathbf{1}_n,\mathbf{0}_n)}$$

Strictly speaking, $\preceq_{\mathrm{EGD}}$ is an order on the *representation* of EGDs, i.e. the parameters $\mathbf{P}$ and $\boldsymbol{\alpha}$ of an EGD $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, not on the measure itself. So to be precise, we should write $(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} (\mathbf{Q}, \boldsymbol{\alpha})$ instead of $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$. However, we went with the slightly imprecise notation because it provides the intuition that $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ is bounded by $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$, see Lemma 6.3.9 below.

Example 6.3.8. Consider the $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ from Example 6.3.2 with $|\mathbf{P}| = (2, 2)$ and the two-dimensional $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ with $|\mathbf{Q}| = (1, 4)$ and $\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & \mathbf{Q}_{0,2} & \mathbf{Q}_{0,3} \end{pmatrix}$. Then $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is equivalent to the following system of inequalities:

$$\begin{array}{llll}
\alpha_1 \leq \beta_1 & \mathbf{P}_{0,0} \leq \mathbf{Q}_{0,0} & \mathbf{P}_{0,1} \leq \mathbf{Q}_{0,1} & \mathbf{P}_{0,1} \cdot \alpha_2 \leq \mathbf{Q}_{0,2} \qquad \mathbf{P}_{0,1} \cdot \alpha_2^2 \leq \mathbf{Q}_{0,3} \\
\alpha_2 \leq \beta_2 & \mathbf{P}_{1,0} \leq \mathbf{Q}_{0,0} \cdot \beta_1 & \mathbf{P}_{1,1} \leq \mathbf{Q}_{0,1} \cdot \beta_1 & \mathbf{P}_{1,1} \cdot \alpha_2 \leq \mathbf{Q}_{0,2} \cdot \beta_1 \qquad \mathbf{P}_{1,1} \cdot \alpha_2^2 \leq \mathbf{Q}_{0,3} \cdot \beta_1
\end{array}$$

Unfortunately, this order $\preceq_{\mathrm{EGD}}$ is not exactly the same as $\preceq$ because zero coefficients at the edge of $\mathbf{P}$ can make $\boldsymbol{\alpha}$ irrelevant, e.g. $\mathrm{EGD}(0, \alpha) = \mathbf{0}$ for any $\alpha \in [0, 1)$. Thus $\mathbf{0} = \mathrm{EGD}(0, \alpha) \preceq \mathrm{EGD}(1, 0) = \mathrm{Dirac}(0)$ even though $\mathrm{EGD}(0, \alpha) \not\preceq_{\mathrm{EGD}} \mathrm{EGD}(1, 0)$ for $\alpha > 0$. However, such counterexamples are rare in practice, and $\preceq_{\mathrm{EGD}}$ does imply $\preceq$, as the following lemma shows. As a consequence, working with the simpler order $\preceq_{\mathrm{EGD}}$ instead of $\preceq$ is sufficient to establish bounds.

Lemma 6.3.9. *If* $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, *then* $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$.

Proof. Let $\mathrm{EGD}(\mathbf{P}', \boldsymbol{\alpha})$ and $\mathrm{EGD}(\mathbf{Q}', \boldsymbol{\beta})$ be expansions of sizes $|\mathbf{P}'| = |\mathbf{Q}'| = \max(|\mathbf{P}|, |\mathbf{Q}|)$. By Lemma 6.3.6, we have $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ if and only if $\mathrm{EGD}(\mathbf{P}', \boldsymbol{\alpha}) \preceq \mathrm{EGD}(\mathbf{Q}', \boldsymbol{\beta})$.

By the definition of expansions, $\mathbf{P}'_{\boldsymbol{i}} \preceq \mathbf{Q}'_{\boldsymbol{i}}$ is equivalent to the condition from the definition of $\preceq_{\mathrm{EGD}}$:

$$\mathbf{P}_{\min(\boldsymbol{i}, |\mathbf{P}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n)} \leq \mathbf{Q}_{\min(\boldsymbol{i}, |\mathbf{Q}|-\mathbf{1}_n)} \boldsymbol{\beta}^{\max(\boldsymbol{i}-|\mathbf{Q}|+\mathbf{1}_n)}$$

Finally, the conditions $\mathbf{P}'_{\boldsymbol{i}} \leq \mathbf{Q}'_{\boldsymbol{i}}$ for $\boldsymbol{i} < |\mathbf{P}'| = |\mathbf{Q}'|$ and $\boldsymbol{\alpha} \leq \boldsymbol{\beta}$ imply

$$\begin{aligned} \mathrm{EGD}(\mathbf{P}', \boldsymbol{\alpha})(\boldsymbol{i}) &= \mathbf{P}'_{\min(\boldsymbol{i}, |\mathbf{P}'|-1)} \boldsymbol{\alpha}^{\max(\boldsymbol{i}-|\mathbf{P}'|+\mathbf{1}_n)} \\ &\leq \mathbf{Q}'_{\min(\boldsymbol{i}, |\mathbf{Q}'|-1)} \boldsymbol{\beta}^{\max(\boldsymbol{i}-|\mathbf{Q}'|+\mathbf{1}_n)} = \mathrm{EGD}(\mathbf{Q}', \boldsymbol{\beta})(\boldsymbol{i}) \end{aligned}$$

for all $\boldsymbol{i} \in \mathbb{N}^n$, as desired. □

6.4 Geometric Bound Semantics

Using the ideas from the previous sections, we define a compositional semantics $[\![P]\!]^{\mathrm{geo}}$ for upper bounds on the unnormalized distribution of P. It is called ***geometric bound semantics*** because it operates on eventually geometric distribution bounds due to their desirable properties mentioned in Section 6.3. It also turns out that EGDs are closed under many operations we require: restricting to events, marginalizing, and adding or subtracting constants from variables.

The semantics $[\![P]\!]^{\mathrm{geo}}$ operates on distributions on $\mathbb{N}^n$, not $\mathbb{N}^n_{\lightning}$. We do not track the failure state $\lightning$ in this semantics, because the geometric bound semantics is only applicable to almost surely terminating programs (see Theorem 6.5.9), so there is no need to distinguish between observation failure and nontermination. Like in the residual mass semantics, we can bound the normalizing constant with the help of the lower bound semantics (see Theorem 6.5.5).

Relational semantics Since there may be more than one upper bound, $[\![P]\!]^{\mathrm{geo}}$ is not a function, but a binary *relation* on EGDs. The idea is that $(\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}), \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})) \in [\![P]\!]^{\mathrm{geo}}$, also written $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \; [\![P]\!]^{\mathrm{geo}} \; \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, ensures that $[\![P]\!](\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. If this $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is unique for a given $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, we will use function notation: $[\![P]\!]^{\mathrm{geo}}(\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})) = \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. Strictly speaking, the relation $[\![P]\!]^{\mathrm{geo}}$ (like $\preceq_{\mathrm{EGD}}$) operates on the *representations* $(\mathbf{P}, \boldsymbol{\alpha}), (\mathbf{Q}, \boldsymbol{\beta})$ of the involved EGDs $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ and $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, not just their measures.

Event semantics The event semantics $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})|^{\mathrm{geo}}_E$ computes an EGD representation of the standard event semantics $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E$ (see Table 6.2). The restriction $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) := \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})|^{\mathrm{geo}}_E$ is generally computed by expanding the initial block $\mathbf{P}$ (if necessary) and then zeroing its entries not corresponding to the event E. An important property is that $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$.

For the event $X_k = a$, we have to set the coefficients of $\mathbf{P}_{k:j}$ to zero for $j \neq a$. For this, we first have to expand $\mathbf{P}$ to $\mathbf{Q}$ with $|\mathbf{Q}|_k = a + 2$ if necessary. Then we set $\mathbf{Q}_{k:j} = \mathbf{0}$

Table 6.2: Geometric bound semantics for events

Event E	Restriction $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_E^{\mathrm{geo}}$ of $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ to the event E
$X_k = a$	$\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha}[k \mapsto 0])$ where $\vert\mathbf{Q}\vert = \vert\mathbf{P}\vert[k \mapsto \max(\vert\mathbf{P}\vert_k, a+2)]$ with $\mathbf{Q}_{k:j} = \mathbf{P}_{k:\min(j,\vert\mathbf{P}\vert_k-1)} \cdot \alpha_k^{\max(0,j-\vert\mathbf{P}\vert_k+1)} \cdot [j = a]$ for $j = 0, \ldots, \vert\mathbf{Q}\vert_k - 1$
$\mathsf{flip}(\rho)$	$\mathsf{EGD}(\rho \cdot \mathbf{P}, \boldsymbol{\alpha})$
$\neg E_1$	$\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$ where $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{E_1}^{\mathrm{geo}} = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ $\vert\mathbf{Q}\vert = \vert\mathbf{R}\vert$ and $\mathbf{Q}_{\boldsymbol{i}} = \mathbf{P}_{\min(\boldsymbol{i},\vert\mathbf{P}\vert-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}^{\max(\boldsymbol{i}-\vert\mathbf{P}\vert+\mathbf{1}_n,\mathbf{0}_n)} - \mathbf{R}_{\boldsymbol{i}} \quad \forall \boldsymbol{i} \leq \vert\mathbf{Q}\vert$
$E_1 \wedge E_2$	$\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{E_1}^{\mathrm{geo}}\right)\Big\vert_{E_2}^{\mathrm{geo}}$

for $j \neq a$ to ensure that all coefficients different from a are zero, and also set $\alpha_k = 0$. For instance, the EGD from Example 6.3.2 restricted to the event $X_2 = 2$ is

$$\mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} & \mathbf{P}_{0,1}\\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1}\end{pmatrix}, \boldsymbol{\alpha}\right)\Big\vert_{X_2=2}^{\mathrm{geo}} = \mathsf{EGD}\left(\begin{pmatrix}0 & 0 & \mathbf{P}_{0,1}\cdot\alpha_2 & 0\\ 0 & 0 & \mathbf{P}_{1,1}\cdot\alpha_2 & 0\end{pmatrix}, (\alpha_1, 0)\right)$$

Next, the event $\mathsf{flip}(\rho)$ is independent of the current distribution, so we can simply multiply the initial block $\mathbf{P}$ by the probability ρ. For the complement $\neg E_1$, we first compute the restriction $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{E_1}^{\mathrm{geo}} = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$. Since $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\alpha})$ by the property mentioned above, we can compute its complement $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha}) := \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathsf{EGD}(\mathbf{R}, \boldsymbol{\alpha})$ by first expanding $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ to $\mathsf{EGD}(\mathbf{P}', \boldsymbol{\alpha})$ with $\mathbf{P}'$ of the same size as $\mathbf{R}$, and then subtracting $\mathbf{Q} := \mathbf{P}' - \mathbf{R}$. So the result is $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$. Continuing the above example with the event $\neg(X_2 = 2)$, we get:

$$\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{\neg(X_2=2)}^{\mathrm{geo}} = \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{X_2=2}^{\mathrm{geo}} = \mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} & \mathbf{P}_{0,1} & 0 & \mathbf{P}_{0,1}\alpha_2^2\\ \mathbf{P}_{1,0} & \mathbf{P}_{1,1} & 0 & \mathbf{P}_{1,1}\alpha_2^2\end{pmatrix}, \boldsymbol{\alpha}\right)$$

Finally, the restriction to an intersection $E_1 \wedge E_2$ is computed by restricting to E_1 and then to E_2.

Statement semantics The statement semantics is a binary relation $[\![P]\!]^{\mathrm{geo}}$ on EGDs. It defines $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ by induction on the structure of P via constraints on $\mathbf{P}, \boldsymbol{\alpha}, \mathbf{Q}, \boldsymbol{\beta}$ (Table 6.3) and ensures $[\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. The semantics of skip is the identity relation and $P_1; P_2$ is relational composition. Since fail corresponds to observing a zero-probability event, its right-hand side is a zero distribution. For $X_k := 0$, we essentially have to marginalize out the k-th dimension (see Lemma 6.3.3) and then put all the probability mass on $X_k = 0$. For instance, applying $[\![X_2 := 0]\!]^{\mathrm{geo}}$ to the $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ from Example 6.3.2, yields the unique right-hand side

$$\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0} + \frac{\mathbf{P}_{0,1}}{1-\alpha_2} & 0\\ \mathbf{P}_{1,0} + \frac{\mathbf{P}_{1,1}}{1-\alpha_2} & 0\end{pmatrix}, (\alpha_1, 0)\right)$$

Table 6.3: Geometric bound semantics for statements

Statement P	Constraints for $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$
skip	$\boldsymbol{\beta} = \boldsymbol{\alpha} \ \wedge\ \mathbf{Q} = \mathbf{P}$
$P_1; P_2$	$\exists \mathbf{R}, \boldsymbol{\gamma}.\ \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P_1]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma}) \ \wedge\ \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})\ [\![P_2]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$
$X_k := 0$	$\boldsymbol{\beta} = \boldsymbol{\alpha}[k \mapsto 0] \ \wedge\ \mathbf{Q}_{k:0} = \dfrac{\mathbf{P}_{k:|\mathbf{P}|_k - 1}}{1 - \alpha_k} + \sum_{j=0}^{|\mathbf{P}|_k - 2} \mathbf{P}_{k:j} \ \wedge\ \mathbf{Q}_{k:1} = \mathbf{0}$ where $\lvert\mathbf{Q}\rvert = \lvert\mathbf{P}\rvert[k \mapsto 2]$
$X_k \mathrel{+}= a$	$\boldsymbol{\beta} = \boldsymbol{\alpha} \ \wedge\ \bigwedge_{j=0}^{|\mathbf{Q}|_k - 1} \mathbf{Q}_{k:j} = \begin{cases} 0 & \text{if } j < a \\ \mathbf{P}_{k:j-a} & \text{else} \end{cases}$ where $\lvert\mathbf{Q}\rvert = \lvert\mathbf{P}\rvert[k \mapsto \lvert\mathbf{P}\rvert_k + a]$
$X_k \dot{-}= 1$	$\boldsymbol{\beta} = \boldsymbol{\alpha} \ \wedge\ \bigwedge_{j=0}^{|\mathbf{Q}|_k - 1} \mathbf{Q}_{k:j} = \begin{cases} \mathbf{P}_{k:0} + \mathbf{P}_{k:\min(|\mathbf{P}|_k - 1, 1)} \cdot \alpha^{\max(2 - |\mathbf{P}|_k, 0)} & \text{if } j = 0 \\ \mathbf{P}_{k:\min(|\mathbf{P}|_k - 1, j+1)} \cdot \alpha^{\max(j+2-|\mathbf{P}|_k, 0)} & \text{else} \end{cases}$ where $\lvert\mathbf{Q}\rvert = \lvert\mathbf{P}\rvert[k \mapsto \max(\lvert\mathbf{P}\rvert_k - 1, 2)]$
if E $\{P_1\}$ else $\{P_2\}$	$\exists \mathbf{R}, \mathbf{S}, \boldsymbol{\gamma}, \boldsymbol{\delta}.\ \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_E^{\mathrm{geo}}\ [\![P_1]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma}) \ \wedge\ \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{\neg E}^{\mathrm{geo}}\ [\![P_2]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta})$ $\wedge\ \big(\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma}), \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})\big) \in \mathrm{Join}$ (see Definition 6.4.1)
while E $\{P\}$	$\exists c, \mathbf{R}, \mathbf{S}, \boldsymbol{\gamma}, \boldsymbol{\delta}.\ 0 \le c < 1 \ \wedge\ \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ $\wedge\ \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})\vert_E^{\mathrm{geo}}\ [\![P]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}) \ \wedge\ \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$ $\wedge\ \mathrm{EGD}\big(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma}\big)\vert_{\neg E}^{\mathrm{geo}} = \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$
fail	$\boldsymbol{\beta} = \mathbf{0}_n \ \wedge\ \mathbf{Q} = \mathbf{0}_{[1]\times\cdots\times[1]}$

For $X_k \mathrel{+}= c$, we shift the coefficients to the right by c in the k-th dimension, and fill up with zeros. For instance, applying $[\![X_2 \mathrel{+}= 2]\!]^{\mathrm{geo}}$ to the above EGD yields the unique right-hand side

$$\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathrm{EGD}\left(\begin{pmatrix} 0 & 0 & \mathbf{P}_{0,0} & \mathbf{P}_{0,1} \\ 0 & 0 & \mathbf{P}_{1,0} & \mathbf{P}_{1,1} \end{pmatrix}, \boldsymbol{\alpha}\right)$$

For $X_k \dot{-}= 1$, we shift the coefficients to the left by 1 in the k-th dimension, except for $\mathbf{P}_{k:0}$, which stays at index 0, so we get the sum of $\mathbf{P}_{k:0}$ and $\mathbf{P}_{k:1}$ at index 0. This special case requires $|\mathbf{Q}|_k \ge 2$, so we may first have to expand $\mathbf{P}$ to ensure $|\mathbf{P}|_k \ge 3$, which is done implicitly in Table 6.3. As an example, applying $[\![X_2 \dot{-}= 1]\!]^{\mathrm{geo}}$ to the above EGD yields the unique right-hand side

$$\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathrm{EGD}\left(\begin{pmatrix} \mathbf{P}_{0,0} + \mathbf{P}_{0,1} & \mathbf{P}_{0,1} \cdot \alpha_2 \\ \mathbf{P}_{1,0} + \mathbf{P}_{1,1} & \mathbf{P}_{1,1} \cdot \alpha_2 \end{pmatrix}, \boldsymbol{\alpha}\right)$$

Conditionals The semantics of if E $\{P_1\}$ else $\{P_2\}$ is more complex. We first require the existence of bounds on both branches ($\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta})$), whose constraints are given by: $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_E^{\mathrm{geo}}\ [\![P_1]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\vert_{\neg E}^{\mathrm{geo}}\ [\![P_2]\!]^{\mathrm{geo}}\ \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta})$. Next, we would like to sum the bounds on the branches to obtain a bound on the whole conditional. However the sum of two EGDs may not be an EGD. Instead, we define a *join* $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ as an upper bound on the sum $\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta})$ of a certain shape.

Definition 6.4.1 (Join relation)**.** We say $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a ***join*** of $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$, and write $(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}), \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})) \in \mathsf{Join}$, if $\boldsymbol{\gamma} \leq \boldsymbol{\beta}, \boldsymbol{\delta} \leq \boldsymbol{\beta}$ and there are expansions $\mathsf{EGD}(\mathbf{R}', \boldsymbol{\gamma})$ of $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{S}', \boldsymbol{\delta})$ of $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$, of the same size, such that $\mathbf{Q} = \mathbf{R}' + \mathbf{S}'$. We also define the ***strict join*** relation $\mathsf{Join}^* \subset \mathsf{Join}$ that strengthens the condition $\boldsymbol{\beta} \geq \max(\boldsymbol{\gamma}, \boldsymbol{\delta})$ to $\boldsymbol{\beta} = \max(\boldsymbol{\gamma}, \boldsymbol{\delta})$.

As an example, the minimal-size join of $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ and $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ from Example 6.3.8 is given by

$$\mathsf{EGD}\left(\begin{pmatrix} \mathbf{P}_{0,0} + \mathbf{Q}_{0,0} & \mathbf{P}_{0,1} + \mathbf{Q}_{0,1} & \mathbf{P}_{0,1}\alpha_2 + \mathbf{Q}_{0,2} & \mathbf{P}_{0,1}\alpha_2^2 + \mathbf{Q}_{0,3} \\ \mathbf{P}_{1,0} + \mathbf{Q}_{0,0}\beta_1 & \mathbf{P}_{1,1} + \mathbf{Q}_{0,1}\beta_1 & \mathbf{P}_{1,1}\alpha_2 + \mathbf{Q}_{0,2}\beta_1 & \mathbf{P}_{1,1}\alpha_2^2 + \mathbf{Q}_{0,3}\beta_1 \end{pmatrix}, (\gamma_1, \gamma_2)\right)$$

with $\alpha_1, \beta_1 \leq \gamma_1 < 1$ and $\alpha_2, \beta_2 \leq \gamma_2 < 1$. It is a strict join if $\gamma_1 = \max(\alpha_1, \beta_1)$ and $\gamma_2 = \max(\alpha_2, \beta_2)$. We use normal joins in the semantics because strict joins would introduce maxima in the constraints, making them harder to solve. Strict joins are useful for theoretical analysis, however.

Loops Bounding a loop $\mathsf{while}\, E\, \{B\}$ requires the existence of a contraction invariant $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and a contraction factor $c \in [0, 1)$ (see Section 6.2), satisfying the following conditions. First, the initial distribution has to be bounded by the invariant: $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$. Second, the invariant has to decrease by a factor of c in each loop iteration, which is encoded as: there exists an $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ such that

$$\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\big|_E^{\mathsf{geo}} \; [\![B]\!]^{\mathsf{geo}} \; \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \quad \text{and} \quad \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$$

Then $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) := \mathsf{EGD}(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma})\big|_{\neg E}^{\mathsf{geo}}$ is an upper bound on $[\![\mathsf{while}\, E\, \{B\}]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))$, as discussed in Section 6.2.

Nondeterminism Note that there are two places in the semantics where choices have to be made: conditionals and loops. This nondeterminism is the reason why the semantics $[\![-]\!]^{\mathsf{geo}}$ is a relation instead of a function. For $\mathsf{if}\, E\, \{P_1\}\, \mathsf{else}\, \{P_2\}$, the choice is in the join operation: concretely, the size of the expansion of the two distributions. For $\mathsf{while}\, E\, \{P\}$, the choice is in the contraction invariant $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and the factor c. First of all, the dimensions $|\mathbf{R}|$ have to be chosen. How we do this in practice is discussed in Section 6.6. Once the size of $\mathbf{R}$ is chosen, the conditions on $\mathbf{R}$ reduce to polynomial inequality constraints (a decidable problem). Of course, we typically want to find a "good" solution to these constraints by optimizing some objective (see Section 6.6.2).

6.4.1 Examples

We illustrate the geometric bound semantics with two examples of simple probabilistic programs.

Example 6.4.2 (Simple counter)**.** Consider a simple program P representing a geometric distribution:

$$\mathsf{while}\, \mathsf{flip}(1/2)\, \{X_1 \mathrel{+}= 1\}$$

The starting state of the program is described by $\mathsf{EGD}(1,0)$. Assume a c-contraction invariant $\mathsf{EGD}(p,\alpha)$ exists with $p \in \mathbb{R}_{\geq 0} \cong \mathbb{R}^{[1]}_{\geq 0}$ and $\alpha \in [0,1)$. Then the constraint $\mathsf{EGD}(1,0) \preceq_{\mathsf{EGD}} \mathsf{EGD}(p,\alpha)$ yields the inequalities $p \geq 1, \alpha \geq \mathbf{0}$.
Let the loop body be $B := (X_1 \mathrel{+}= 1)$. We find

$$\mathsf{EGD}(p,\alpha)|^{\mathsf{geo}}_{\mathsf{flip}(1/2)} [\![B]\!]^{\mathsf{geo}} \mathsf{EGD}\left(\begin{pmatrix} 0 & \frac{p}{2} \end{pmatrix}, \alpha\right)$$

The constraint for the contraction invariant is

$$\mathsf{EGD}\left(\begin{pmatrix} 0 & \frac{p}{2} \end{pmatrix}, \alpha\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot p, \alpha) = \mathsf{EGD}\left(\begin{pmatrix} cp & cp\alpha \end{pmatrix}, \alpha\right)$$

and amounts to $p \geq 1$ (from above) and $\frac{1}{2} \leq c \cdot \alpha$ with $c < 1$. The bound on the whole loop is then given by

$$\frac{1}{1-c} \mathsf{EGD}(p,\alpha)|^{\mathsf{geo}}_{\neg\mathsf{flip}(1/2)} = \mathsf{EGD}\left(\frac{p}{2(1-c)}, \alpha\right)$$

To get a low upper bound, it is best to set $p = 1$. How to choose α and c is not clear because decreasing α will increase $\frac{1}{1-c}$ and vice versa.
If we want to optimize the asymptotics of the tail probabilities $\mathbb{P}[X_1 = n]$ as $n \to \infty$, we want to choose α as small as possible, i.e. very close to $\frac{1}{2}$, accepting that this will make $\frac{1}{1-c} \geq \frac{2\alpha}{2\alpha-1}$ large. Setting $\alpha = \frac{1}{2} + \varepsilon$ yields $\frac{1}{2(1-c)} \geq \frac{\alpha}{2\alpha-1} = \frac{1}{4\varepsilon} + \frac{1}{2}$ and thus the bound

$$[\![P]\!](\mathsf{EGD}(1,0)) \preceq \mathsf{EGD}\left(\frac{1}{4\varepsilon} + \frac{1}{2}, \frac{1}{2} + \varepsilon\right)$$

which implies $\mathbb{P}[X_1 = n] \leq \left(\frac{1}{4\varepsilon} + \frac{1}{2}\right)\left(\frac{1}{2} + \varepsilon\right)^n$.
If we want to optimize the bound on the expected value of X_1, we want to minimize (by Lemma 6.3.4)

$$\mathbb{E}_{X \sim [\![P]\!](\mathsf{EGD}(1,0))}[X_1] \leq \mathbb{E}_{X \sim \mathsf{EGD}\left(\frac{1}{2(1-c)}, \alpha\right)}[X_1] = \frac{\alpha}{2(1-c)(1-\alpha)^2}$$

This is minimized under the constraints $0 \leq \alpha, c < 1$ and $\frac{1}{2} \leq \alpha c$ for $\alpha = \frac{\sqrt{5}-1}{2} \approx 0.618$ and $c = \frac{\sqrt{5}+1}{4} \approx 0.809$. At this point, the bound on the expected value is

$$\mathbb{E}[X_1] \leq \frac{11 + 5\sqrt{5}}{2} \approx 11.09$$

This is quite a bit more than the true value of 1, but it is a finite bound that can be found fully automatically. It can be improved by unrolling the loop a few times before applying the above procedure. This way, our implementation finds much better bounds (see Table 6.7).

Example 6.4.3 (asymmetric random walk)**.** Consider again the program P from Example 5.1.1 representing a biased random walk on $\mathbb{N}$, starting at 1, and stopping

when reaching 0:

$$
\begin{aligned}
&X_1 := 1; X_2 := 0;\\
&\textsf{while}\, X_1 > 0\,\{\\
&\quad X_2 \mathrel{+}= 1;\\
&\quad \{X_1 \mathrel{+}= 1\}\ [r]\ \{X_1 \mathrel{\dot{-}}= 1\}\\
&\}
\end{aligned}
$$

where X_1 is the current position, X_2 is the number of steps taken, and the bias $r < \frac{1}{2}$ is the probability of going right. We find

$$
[\![X_1 := 1; X_2 := 0]\!]^{\text{geo}}\ (\text{EGD}(1, (0,0))) = \text{EGD}\left(\begin{pmatrix}0\\1\end{pmatrix}, (0,0)\right)
$$

as the distribution at the start of the loop. Assume a c-contraction invariant $\text{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ with $\mathbf{P} \in \mathbb{R}_{\geq 0}^{[2]\times[1]}$ exists. Then the bound on the distribution of the program is

$$
[\![P]\!](\text{EGD}(1,(0,0))) \preceq \text{EGD}\left(\frac{\mathbf{P}}{1-c}, \boldsymbol{\alpha}\right)\Big|^{\text{geo}}_{\neg(X_1>0)} = \text{EGD}\left(\begin{pmatrix}\frac{\mathbf{P}_{0,0}}{1-c}\\0\end{pmatrix}, (0, \alpha_2)\right)
$$

One loop iteration transforms the contraction invariant as follows (using the strict join relation for the probabilistic branching):

$$
\begin{aligned}
&[\![X_2 \mathrel{+}= 1; \{X_1 \mathrel{+}= 1\}\ [r]\ \{X_1 \mathrel{\dot{-}}= 1\}]\!]^{\text{geo}} \left(\text{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0}\\ \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\Big|^{\text{geo}}_{X_1>0}\right)\\
&= [\![X_2 \mathrel{+}= 1; \{X_1 \mathrel{+}= 1\}\ [r]\ \{X_1 \mathrel{\dot{-}}= 1\}]\!]^{\text{geo}} \left(\text{EGD}\left(\begin{pmatrix}0\\ \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\right)\\
&= [\![\{X_1 \mathrel{+}= 1\}\ [r]\ \{X_1 \mathrel{\dot{-}}= 1\}]\!]^{\text{geo}} \left([\![X_2 \mathrel{+}= 1]\!]^{\text{geo}}\ (\text{EGD}\left(\begin{pmatrix}0\\ \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right))\right)\\
&= [\![\textsf{if flip}(r)\ \{X_1 \mathrel{+}= 1\}\ \textsf{else}\ \{X_1 \mathrel{\dot{-}}= 1\}]\!]^{\text{geo}} \left(\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\right)\\
&= \text{EGD}\left(\begin{pmatrix}0 & (1-r)\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1^2\mathbf{P}_{1,0} + r\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)
\end{aligned}
$$

because the two branches transform the input distribution as follows:

$$
\begin{aligned}
[\![X_1 \mathrel{+}= 1]\!]^{\text{geo}} \left(\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\Big|^{\text{geo}}_{\text{flip}(r)}\right) &= [\![X_1 \mathrel{+}= 1]\!]^{\text{geo}} \left(\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & r\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\right)\\
&= \text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & 0\\ 0 & r\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)
\end{aligned}
$$

$$[\![X_1 \dot{-}= 1]\!]^{\text{geo}}\left(\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\Big|^{\text{geo}}_{\neg\text{flip}(r)}\right) = [\![X_1 \dot{-}= 1]\!]^{\text{geo}}\left(\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & (1-r)\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)\right)$$
$$= \text{EGD}\left(\begin{pmatrix}0 & (1-r)\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)$$

so their strict join is

$$\text{EGD}\left(\begin{pmatrix}0 & 0\\ 0 & 0\\ 0 & r\mathbf{P}_{1,0}\end{pmatrix} + \begin{pmatrix}0 & (1-r)\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1^2\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right) = \text{EGD}\left(\begin{pmatrix}0 & (1-r)\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1^2\mathbf{P}_{1,0} + r\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)$$

Hence the contraction invariant has to satisfy the following requirements:

$$\text{EGD}\left(\begin{pmatrix}0\\ 1\end{pmatrix}, (0,0)\right) \preceq_{\text{EGD}} \text{EGD}\left(\begin{pmatrix}\mathbf{P}_{0,0}\\ \mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)$$
$$\text{EGD}\left(\begin{pmatrix}0 & (1-r)\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1\mathbf{P}_{1,0}\\ 0 & (1-r)\alpha_1^2\mathbf{P}_{1,0} + r\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right) \preceq_{\text{EGD}} \text{EGD}\left(\begin{pmatrix}c\mathbf{P}_{0,0}\\ c\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)$$
$$= \text{EGD}\left(\begin{pmatrix}c\mathbf{P}_{0,0} & c\alpha_2\mathbf{P}_{0,0}\\ c\mathbf{P}_{1,0} & c\alpha_2\mathbf{P}_{1,0}\\ c\alpha_1\mathbf{P}_{1,0} & c\alpha_1\alpha_2\mathbf{P}_{1,0}\end{pmatrix}, \boldsymbol{\alpha}\right)$$

which reduce to the following polynomial constraints:

$$0 \le \mathbf{P}_{0,0} \qquad 1 \le \mathbf{P}_{1,0} \qquad (1-r)\mathbf{P}_{1,0} \le c\alpha_2\mathbf{P}_{0,0}$$
$$(1-r)\alpha_1\mathbf{P}_{1,0} \le c\alpha_2\mathbf{P}_{1,0} \qquad (1-r)\alpha_1^2\mathbf{P}_{1,0} + r\mathbf{P}_{1,0} \le c\alpha_1\alpha_2\mathbf{P}_{1,0}$$

besides the obvious ones (every variable is nonnegative and $\alpha_1, \alpha_2, c \in [0,1)$). The projection of the solution set to α_1, α_2 is shown in Fig. 6.1a for $r = \frac{1}{4}$. The most interesting constraint is the last one, which is equivalent to

$$(1-r)\cdot\alpha_1^2 - c\alpha_2\cdot\alpha_1 + r \le 0$$

Solving this for α_1 yields $\alpha_1 \in \left[\frac{c\alpha_2 - \sqrt{c^2\alpha_2^2 - 4r(1-r)}}{2(1-r)}, \frac{c\alpha_2 + \sqrt{c^2\alpha_2^2 - 4r(1-r)}}{2(1-r)}\right]$. For α_1 to exist, we need to have $1 > c\alpha_2 \ge \sqrt{4r(1-r)}$. For α_1 to be satisfiable is equivalent to the lower bound of the interval being less than 1, i.e. $c\alpha_2 - 2(1-r) < \sqrt{c^2\alpha_2^2 - 4r(1-r)}$. This can only hold if $c\alpha_2 - 2(1-r) \le 0$ or

$$\begin{aligned} & (c\alpha_2 - 2(1-r))^2 < c^2\alpha_2^2 - 4r(1-r)\\ \iff\quad & -2c\alpha_2 + 4(1-r) < -4r\\ \iff\quad & c\alpha_2 > 2 \end{aligned}$$

a contradiction. Hence $c\alpha_2 \leq 2(1-r)$ must hold, and together with $c\alpha_2 \geq \sqrt{4r(1-r)}$ we obtain:

$$\sqrt{4r(1-r)} \leq 2(1-r) \iff 4r \leq 4(1-r) \iff r \leq \frac{1}{2}$$

For $r = \frac{1}{2}$, the constraint $\sqrt{4r(1-r)} \leq c\alpha_2 < 1$ is violated. So $r < \frac{1}{2}$ is a necessary condition for the existence of a solution. It turns out that this is sufficient: we can pick $c, \alpha_2 \in [0,1)$ with $c\alpha_2 = \sqrt{4r(1-r)}$. Then $\alpha_1 = \frac{c\alpha_2}{2(1-r)} < 1$ is a valid solution. This case split on r makes intuitive sense because for $r \geq \frac{1}{2}$, the program has infinite expected running time.
In the satisfiable case $r < \frac{1}{2}$, we get the following bound on the distribution of the program:

$$\mathsf{EGD}(1,(0,0))\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\begin{pmatrix}\frac{\mathbf{P}_{0,0}}{1-c}\\ \frac{\mathbf{P}_{1,0}}{1-c}\end{pmatrix}, \boldsymbol{\alpha}\right)\Bigg|^{\mathrm{geo}}_{\neg(X_1>0)} = \mathsf{EGD}\left(\begin{pmatrix}\frac{\mathbf{P}_{0,0}}{1-c}\\ 0\end{pmatrix}, (0,\alpha_2)\right)$$

The asymptotic bound is $\mathbb{P}[X_2 = n] = \frac{\mathbf{P}_{0,0}}{1-c}\alpha_2^n$, so it is best for α_2 as small as possible. Since $\alpha_2 \geq \frac{\sqrt{4r(1-r)}}{c} > 2\sqrt{r(1-r)}$, the best possible geometric bound for $\Pr[X_2 = n]$ is $O((2\sqrt{r(1-r)}+\varepsilon)^n)$. We could find a bound on $\mathbb{E}[X_2]$ in the same way as in the previous example, but this would be very tedious to do manually. Instead we will automate the semantics in Section 6.6.

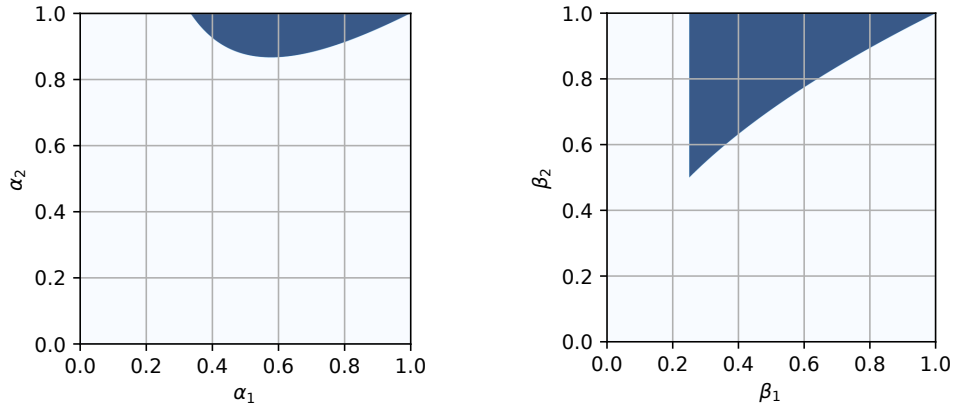
Interestingly, our method's tail bound can get arbitrarily close (asymptotically) to the best possible geometric bound for this example. As mentioned in Example 5.1.1, the exact probabilities for the random walk are given by

$$\mathbb{P}[X_2 = n] = \frac{1}{n}\binom{n}{(n-1)/2} \cdot r^{(n-1)/2}(1-r)^{(n+1)/2}$$

Using well-known approximations for Catalan numbers $\frac{1}{2n+1}\binom{2n+1}{n} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$, we can compute their asymptotics for the exact probabilities (see Example 5.1.1) as follows:

$$\begin{aligned}
\mathbb{P}[X_2 = n] &= \frac{1}{n}\binom{n}{(n-1)/2} \cdot r^{(n-1)/2}(1-r)^{(n+1)/2} \\
&\sim \frac{4^{(n-1)/2}}{((n-1)/2)^{3/2}\sqrt{\pi}} \cdot \frac{\sqrt{1-r}}{\sqrt{r}} \cdot (\sqrt{r(1-r)})^n \\
&= (2\sqrt{r(1-r)})^n \cdot \frac{\sqrt{1-r}}{2\sqrt{\pi r}((n-1)/2)^{3/2}} \\
&= \Theta((2\sqrt{r(1-r)})^n \cdot n^{-3/2})
\end{aligned}$$

Thus the best possible geometric bound is $O((2\sqrt{r(1-r)})^n)$, and our method's bound can get arbitrarily close. Our implementation (again using loop unrolling before applying the above reasoning) finds the bounds shown in Table 6.7 (for $r = \frac{1}{4}$).



(a) Solution set for Example 5.1.1, projected onto α_1, α_2 for $r = \frac{1}{4}$.

(b) Solution set for Example 6.5.2, projected onto β_1, β_2.

Figure 6.1: Visualizations of the solution sets of the constraints arising from two examples.

6.5 Theoretical Analysis

6.5.1 Decidability

For any $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ occurring in the semantics, each entry of $\mathbf{P}$ and $\boldsymbol{\alpha}$ is a rational function of entries of initial blocks, decay rates, and contraction factors. (In fact, it is linear in initial block entries.) Hence the $\preceq_{\mathrm{EGD}}$ constraints reduce to polynomial inequality constraints if the sizes of the EGDs are known. If we want to find an upper bound, these constraints will contain unknowns, which we have to solve for. Since the existential first-order theory of the reals is decidable, solving these constraints is indeed possible.

Theorem 6.5.1. *Given a program P and an $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, it is decidable whether there is an EGD $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ such that $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \ [\![P]\!]^{\mathrm{geo}} \ \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, assuming the sizes of the intermediate EGDs for joins and contraction invariants in the semantics are fixed.*

Proof. First, note that the sizes of all intermediate EGDs in each rule are determined by the input $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, except for contraction invariants and joins, whose size needs to be fixed for this reason. Next, we note that for $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) := \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}$, the elements of $\mathbf{Q}$ and $\boldsymbol{\beta}$ are polynomials of elements of $\mathbf{P}$ and $\boldsymbol{\alpha}$. In fact, only $\boldsymbol{\alpha}$ occurs nonlinearly. All this follows easily from a structural induction on E.

We claim that existence of $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ with $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \ [\![P]\!]^{\mathrm{geo}} \ \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is equivalent to a formula of the form

$$\exists x_1, \ldots, x_m. \bigwedge_{i=1}^{k} p_i(x_1, \ldots, x_m) \bowtie_i 0$$

where $x_1, \ldots, x_m \in \mathbb{R}_{\geq 0}$, $\bowtie_i \in \{\leq, <\}$ and each p_i is a polynomial. The proof is again by structural induction.

For simple statements skip, fail, $X_k \mathrel{+}= a, X_k \mathrel{\dot{-}}= 1, X_k := 0$, the claim is clear because the elements of $\mathbf{Q}$ and $\boldsymbol{\beta}$ are specified exactly as rational functions of elements of $\mathbf{P}$ and

$\boldsymbol{\alpha}$, so they can be encoded as

$$\exists \mathbf{Q}, \boldsymbol{\beta}. \bigwedge_{i=1}^{n} \beta_i = f_i(\boldsymbol{\alpha}) \wedge \bigwedge_{\boldsymbol{i}<|\mathbf{Q}|} \mathbf{Q}_{\boldsymbol{i}} = g_{\boldsymbol{i}}(\mathbf{P}, \boldsymbol{\alpha})$$

where f_i and $g_{\boldsymbol{i}}$ are rational functions. This formula can clearly be transformed into the desired shape. For $P = P_1; P_2$, the claim is clear by induction hypothesis.

For conditionals, we assume that the size $|\mathbf{Q}|$ of the join $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is fixed. The constraints for the two branches can be expressed in the desired form by the induction hypothesis. The constraint $(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}), \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})) \in \mathrm{Join}$ can be encoded as polynomial constraints in $\mathbf{R}, \mathbf{S}, \mathbf{Q}, \boldsymbol{\gamma}, \boldsymbol{\delta}, \boldsymbol{\beta}$, using the definition of Join and expansions:

$$\bigwedge_{i=1}^{n} (\gamma_i \leq \beta_i \wedge \delta_i \leq \beta_i \wedge \beta_i < 1)$$
$$\wedge \bigwedge_{\boldsymbol{i}<|\mathbf{Q}|} \mathbf{Q}_{\boldsymbol{i}} = \mathbf{R}_{\min(\boldsymbol{i},|\mathbf{R}|-\mathbf{1}_n)} \boldsymbol{\gamma}^{\max(\mathbf{0}_n, \boldsymbol{i}-|\mathbf{R}|+\mathbf{1}_n)} + \mathbf{S}_{\min(\boldsymbol{i},|\mathbf{S}|-\mathbf{1}_n)} \boldsymbol{\delta}^{\max(\mathbf{0}_n, \boldsymbol{i}-|\mathbf{S}|+\mathbf{1}_n)}$$

which is a system of polynomial equalities.

For loop invariants, we assume that the size $|\mathbf{R}|$ of the invariant $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ is fixed. Let $\boldsymbol{s} = \max(|\mathbf{P}|, |\mathbf{R}|)$. Then the constraint $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ from the semantics corresponds to the following polynomial constraints:

$$\bigwedge_{i=1}^{n} \alpha_i \leq \gamma_i \wedge \bigwedge_{\boldsymbol{i}<\boldsymbol{s}} \mathbf{P}_{\min(\boldsymbol{i},|\mathbf{P}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n)} \leq \mathbf{R}_{\min(\boldsymbol{i},|\mathbf{R}|-\mathbf{1}_n)} \boldsymbol{\gamma}^{\max(\mathbf{0}_n, \boldsymbol{i}-|\mathbf{R}|+\mathbf{1}_n)}$$

The constraint $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$ can be encoded analogously. The remaining constraints can be encoded by the inductive hypothesis and composed into a big existential conjunction. This proves the claim. Note that in all the constraints, elements of initial blocks occur linearly, which is exploited in the implementation (see Section 6.6.1). The decidability follows from the well-known fact that the existential theory of the reals is decidable. □

This is yet another reason to prefer EGDs over more general classes of distributions: while it may be possible to check $\preceq$ for slightly more complicated classes of distributions with concrete values, solving such a system of $\preceq$-inequalities with unknown values will be much harder. Despite the decidability results, solving our constraints is still not easy because the existential theory of the reals is NP-hard and algorithms only work for small instances in practice. We address the problem of practical constraint solving in Section 6.6.1.

6.5.2 Properties of the Solution Set

One might hope that the constraints arising from the geometric bound semantics have nice properties. For example, one might expect the solution set $(\mathbf{Q}, \boldsymbol{\beta})$ to be convex and $\boldsymbol{\beta}$ to be right-closed, i.e. that for any solution $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ and any $\boldsymbol{\gamma} \geq \boldsymbol{\beta}$, there is a solution $\mathsf{EGD}(\mathbf{Q}', \boldsymbol{\gamma})$, because this weakens the bound. Both properties would be desirable because they would make the constraint solving and optimization easier. However, we found a simple counterexample to both properties.

Example 6.5.2 (Doubling a geometric variable)**.** Consider the doubling of a geometric variable with initial distribution $X_1 \sim \mathsf{Geometric}(\frac{3}{4})$ and $X_2 = 0$, in other words $\boldsymbol{X} \sim \mathsf{EGD}\big(1, (\frac{1}{4}, 0)\big)$:

$$\mathsf{while}\, X_1 > 0\, \{X_2 \mathrel{+}= 2; X_1 \mathrel{\dot{-}}= 1\}$$

Assuming a contraction invariant $\mathsf{EGD}(p, \boldsymbol{\beta})$ with $p \in \mathbb{R} \cong \mathbb{R}^{[1]\times[1]}$ exists, we find:

$$\begin{aligned}
[\![X_2 \mathrel{+}= 2; X_1 \mathrel{\dot{-}}= 1]\!]^{\mathrm{geo}} \left(\mathsf{EGD}(p, \boldsymbol{\beta})|^{\mathrm{geo}}_{X_1>0}\right) &= [\![X_2 \mathrel{+}= 2; X_1 \mathrel{\dot{-}}= 1]\!]^{\mathrm{geo}} \left(\mathsf{EGD}\left(\begin{pmatrix} 0 \\ p\beta_1 \end{pmatrix}, \boldsymbol{\beta}\right)\right) \\
&= [\![X_1 \mathrel{\dot{-}}= 1]\!]^{\mathrm{geo}} \left(\mathsf{EGD}\left(\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & p\beta_1 \end{pmatrix}, \boldsymbol{\beta}\right)\right) \\
&= \mathsf{EGD}\left(\begin{pmatrix} 0 & 0 & p\beta_1 \end{pmatrix}, \boldsymbol{\beta}\right)
\end{aligned}$$

The conditions for the contraction invariant are:

$$\begin{aligned}
\mathsf{EGD}\big(1, (\tfrac{1}{4}, 0)\big) &\preceq_{\mathsf{EGD}} \mathsf{EGD}(p, \boldsymbol{\beta}) \\
\mathsf{EGD}\left(\begin{pmatrix} 0 & 0 & p\beta_1 \end{pmatrix}, \boldsymbol{\beta}\right) &\preceq_{\mathsf{EGD}} \mathsf{EGD}(cp, \boldsymbol{\beta}) = \mathsf{EGD}\left(\begin{pmatrix} cp & cp\beta_2 & cp\beta_2^2 \end{pmatrix}, \boldsymbol{\beta}\right)
\end{aligned}$$

which reduce to the following polynomial inequalities:

$$p \geq 1 \qquad \beta_1 \geq \tfrac{1}{4} \qquad \beta_1 < 1 \qquad \beta_2 \geq 0 \qquad \beta_2 < 1 \qquad cp\beta_1 \leq cp\beta_2^2 \qquad c \geq 0 \wedge c < 1$$

The constraints on $\boldsymbol{\beta}$, after eliminating all other unknowns are equivalent to:

$$\beta_1 \geq \tfrac{1}{4} \qquad \beta_1 < 1 \qquad \beta_2 > 0 \qquad \beta_2 < 1 \qquad \beta_1 < \beta_2^2$$

The feasible values of β are plotted in Fig. 6.1b. Clearly the solution set is neither convex nor right-closed.

6.5.3 Soundness

We prove that the geometric bound semantics is correct with respect to the standard semantics. But first, we show that the join relations Join and Join* and the geometric bound semantics enjoy nice properties. In particular they are monotone and multiplicative, as stated in the following lemmas.

Lemma 6.5.3 (Well-behavedness of the join relation)**.** *Suppose that we have a join* $\big(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}), \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})\big) \in \mathsf{Join}$. *Then*

1. *if* $\mathsf{EGD}\left(\tilde{\mathbf{R}}, \tilde{\boldsymbol{\gamma}}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ *and* $\mathsf{EGD}\left(\tilde{\mathbf{S}}, \tilde{\boldsymbol{\delta}}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ *then there is a* $\mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right)$ *such that* $\left(\mathsf{EGD}\left(\tilde{\mathbf{R}}, \tilde{\boldsymbol{\gamma}}\right), \mathsf{EGD}\left(\tilde{\mathbf{S}}, \tilde{\boldsymbol{\delta}}\right), \mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right)\right) \in \mathsf{Join}$ *and* $\mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$.
2. *for any* $c > 0$, *we have* $\big(\mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma}), \mathsf{EGD}(c \cdot \mathbf{S}, \boldsymbol{\delta}), \mathsf{EGD}(c \cdot \mathbf{Q}, \boldsymbol{\beta})\big) \in \mathsf{Join}$.

Analogous properties hold for the strict join relation Join*.

Proof. The proof is immediate from the definition of the join relations. □

Lemma 6.5.4 (Well-behavedness of the geometric bound semantics)**.** *Suppose we have* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. *Then*

1. *if* $\mathsf{EGD}\left(\tilde{\mathbf{P}}, \tilde{\boldsymbol{\alpha}}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ *holds then there exists an* $\mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right)$ *such that* $\mathsf{EGD}\left(\tilde{\mathbf{P}}, \tilde{\boldsymbol{\alpha}}\right)\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right)$ *and* $\mathsf{EGD}\left(\tilde{\mathbf{Q}}, \tilde{\boldsymbol{\beta}}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$.
2. *for any* $c > 0$, *we have* $\mathsf{EGD}(c \cdot \mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(c \cdot \mathbf{Q}, \boldsymbol{\beta})$.

Proof. The proof is a straightforward induction on the structure of P. □

Theorem 6.5.5 (Soundness)**.** *The rules for the EGD event semantics (Table 6.2) agree with the standard event semantics:* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}} = \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E$. *The rules for the EGD statement semantics* $[\![-]\!]^{\mathrm{geo}}$ *(Table 6.3) are sound: if* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ *then* $[\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. *Furthermore, if* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ *is a probability distribution, then we can bound the normalized distribution:*

$$\frac{[\![P]\!]_{\mathsf{lo}}(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))|_{\mathbb{N}^n}}{\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})(\mathbb{N}^n)} \preceq \mathsf{normalize}([\![P]\!]_{\lightning}(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))) \preceq \frac{\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})}{[\![P]\!]_{\mathsf{lo}}(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))(\mathbb{N}^n)}$$

Proof. We first prove soundness of the event semantics, that is $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}(\mathbf{P}, \alpha)|_E$ where $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) := \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}$. First, we note that $|\mathbf{Q}| \geq |\mathbf{P}|$ because no rule reduces the size of the initial block and $\beta_i \in \{0, \alpha_i\}$ because rules can only change the decay rates by setting them to 0. We strengthen the inductive hypothesis to include $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$.

For $E = (X_k = a)$, we note that $\mu|_E(\boldsymbol{x}) = \mu(\boldsymbol{x}) \cdot [x_k = a]$. Let $\mathbf{P}'$ be the expansion of $\mathbf{P}$ to size $\max(|\mathbf{P}|_k, a + 2)$ in dimension k. In particular, $|\mathbf{P}'| = |\mathbf{Q}|$. Then we find:

$$\begin{aligned}
\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E(\boldsymbol{x}) &= \mathsf{EGD}(\mathbf{P}', \boldsymbol{\alpha})|_E(\boldsymbol{x}) \\
&= \mathbf{P}'_{\min(\boldsymbol{x}, |\mathbf{P}'|-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{P}'|+\mathbf{1}_n)}[x_k = a] \\
&= \left([x_k = a]\mathbf{P}'_{\boldsymbol{x}[k \mapsto \min(x_k, |\mathbf{P}'|_k - 1)]}\right) \cdot \boldsymbol{\alpha}[k \mapsto 0]^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{P}'|+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x}, |\mathbf{Q}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)} \\
&= \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})(\boldsymbol{x})
\end{aligned}$$

because $x_k = a$ implies $x_k < |\mathbf{P}'|_k$ and $\mathbf{Q}_{\boldsymbol{i}}$ equals $[i_k = a]\mathbf{P}'_{\boldsymbol{i}} = [i_k = a]\mathbf{P}_{\min(\boldsymbol{i}, |\mathbf{P}|-\mathbf{1}_n)} \cdot \alpha_k^{\max(0, i_k - |\mathbf{P}|_k + 1)}$ by definition. Furthermore, $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$ follows from $\mathbf{Q}_{k:|\mathbf{Q}|_k - 1} = \mathbf{0}$.

For $E = \neg E_1$, let $\mu = \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ and $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) := \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{E_1}^{\mathrm{geo}} = \mu|_{E_1}$ by induction hypothesis. By the inductive hypothesis, we have $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\alpha})$. Let $\mathbf{P}'$ be the expansion of $\mathbf{P}$ to size $|\mathbf{Q}| = |\mathbf{R}|$. We find:

$$\begin{aligned}
\mu|_{\neg E_1}(\boldsymbol{x}) &= \mu - \mu|_{E_1}(\boldsymbol{x}) \\
&= \mathsf{EGD}(\mathbf{P}', \boldsymbol{\alpha})(\boldsymbol{x}) - \mathsf{EGD}(\mathbf{R}, \boldsymbol{\alpha})(\boldsymbol{x}) \\
&= \mathbf{P}'_{\min(\boldsymbol{x}, |\mathbf{P}'|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{P}'|+\mathbf{1}_n)} - \mathbf{R}_{\min(\boldsymbol{x}, |\mathbf{R}|-1)} \cdot \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{R}|+\mathbf{1}_n)} \\
&= \left(\mathbf{P}'_{\min(\boldsymbol{x}, |\mathbf{Q}|-\mathbf{1}_n)} - \mathbf{R}_{\min(\boldsymbol{x}, |\mathbf{Q}|-\mathbf{1}_n)}\right) \cdot \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x}, |\mathbf{Q}|-\mathbf{1}_n)} \boldsymbol{\alpha}^{\max(\mathbf{0}_n, \boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)}
\end{aligned}$$

$$= \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)} \boldsymbol{\beta}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)}$$
$$= \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})(\boldsymbol{x})$$

since $\mathbf{Q}_{\boldsymbol{i}} := \mathbf{P}'_{\boldsymbol{i}} - \mathbf{R}_{\boldsymbol{i}} = \mathbf{P}_{\min(\boldsymbol{i},|\mathbf{P}|-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{i}-|\mathbf{P}|+\mathbf{1}_n)} - \mathbf{R}_{\boldsymbol{i}}$ by the definition of the expansion.

For $E = \mathsf{flip}(\rho)$, the claim is clear by linearity. For $E = E_1 \wedge E_2$, the correctness follows immediately from $\mu|_{E_1 \wedge E_2} = \mu|_{E_1}|_{E_2}$.

As for the statement semantics, we prove $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})$ implies $[\![P]\!](\mathsf{EGD}(\mathbf{P},\boldsymbol{\beta})) \preceq \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})$. The cases skip, $P_1; P_2$, and fail are obvious. For skip and fail, even equality holds.

The case $X_k := 0$ follows from Lemma 6.3.3, except that the variable X_k is not removed, but set to 0. We have $[\![P]\!](\mu)(\boldsymbol{x}) = \sum_{n\in\mathbb{N}} \mu(\boldsymbol{x}[k \mapsto n]) \cdot [x_k = 0]$. Hence we find

$$\begin{aligned}
&[\![X_k := 0]\!](\mu)(\boldsymbol{x}) \\
&= \sum_{n\in\mathbb{N}} \mu(\boldsymbol{x}[k \mapsto n]) \cdot [x_k = 0] \\
&= [x_k = 0] \left(\frac{\mathbf{P}_{\min(\boldsymbol{x}[k\mapsto|\mathbf{P}|_k-1],|\mathbf{P}|-\mathbf{1}_n)}}{1-\alpha_k} + \sum_{j=0}^{|\mathbf{P}|_k-2} \mathbf{P}_{\min(\boldsymbol{x}[k\mapsto j],|\mathbf{P}|-\mathbf{1}_n)} \right) \cdot [x_k = 0]\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{P}|+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)} \cdot \boldsymbol{\alpha}[k \mapsto 0]^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{P}|[k\mapsto 2]+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)} \cdot \boldsymbol{\beta}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)}
\end{aligned}$$

by the definition of $\mathbf{Q}$.

The case $X_k \mathrel{+}= a$ is easier because $[\![P]\!](\mu)(\boldsymbol{x}) = [x_k \geq a]\mu(\boldsymbol{x}[k \mapsto x_k - a])$. We find:

$$\begin{aligned}
[\![X_k \mathrel{+}= a]\!](\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha}))(\boldsymbol{x}) &= [x_k \geq a]\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})(\boldsymbol{x}[k \mapsto x_k - a]) \\
&= [x_k \geq a]\mathbf{P}_{\min(\boldsymbol{x}[k\mapsto x_k-a],|\mathbf{P}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}[k\mapsto x_k-a]-|\mathbf{P}|+\mathbf{1}_n)} \\
&= [x_k \geq a]\mathbf{P}_{\min(\boldsymbol{x}[k\mapsto x_k-a],|\mathbf{P}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{P}|[k\mapsto \mathbf{P}_k+a]+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)} \\
&= \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})(\boldsymbol{x})
\end{aligned}$$

The case $X_k \mathrel{\dot{-}}= 1$ is similar: $[\![P]\!](\mu)(\boldsymbol{x}) = \mu(\boldsymbol{x}[k \mapsto x_k + 1]) + [x_k = 0]\mu(\boldsymbol{x})$. First we consider the case $|\mathbf{P}|_k \geq 3$. Then $|\mathbf{Q}|_k = \max(|\mathbf{P}|_k - 1, 2) = |\mathbf{P}|_k - 1$ and we find:

$$\begin{aligned}
&[\![X_k \mathrel{\dot{-}}= 1]\!](\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha}))(\boldsymbol{x}) \\
&= [x_k = 0]\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})(\boldsymbol{x}) + \mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})(\boldsymbol{x}[k \mapsto x_k + 1]) \\
&= [x_k = 0]\mathbf{P}_{\min(\boldsymbol{x},|\mathbf{P}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{P}|+\mathbf{1}_n)} + \mathbf{P}_{\min(\boldsymbol{x}[k\mapsto x_k+1],|\mathbf{P}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}[k\mapsto x_k+1]-|\mathbf{P}|+\mathbf{1}_n)} \\
&= \left([x_k = 0]\mathbf{P}_{\min(\boldsymbol{x}[k\mapsto 0],|\mathbf{P}|-\mathbf{1}_n)} + \mathbf{P}_{\min(\boldsymbol{x}[k\mapsto x_k+1],|\mathbf{P}|-\mathbf{1}_n)}\right) \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)} \\
&= \mathbf{Q}_{\min(\boldsymbol{x},|\mathbf{Q}|-\mathbf{1}_n)}\, \boldsymbol{\alpha}^{\max(\mathbf{0}_n,\boldsymbol{x}-|\mathbf{Q}|+\mathbf{1}_n)} \\
&= \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})(\boldsymbol{x})
\end{aligned}$$

For $|\mathbf{P}|_k \leq 2$, we first extend $\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha})$ to $\mathsf{EGD}(\mathbf{P}',\boldsymbol{\alpha})$ with $|\mathbf{P}'|_k = 3$. By the above argument, we find $[\![X_k \mathrel{\dot{-}}= 1]\!](\mathsf{EGD}(\mathbf{P}',\boldsymbol{\alpha})) = \mathsf{EGD}(\mathbf{Q},\boldsymbol{\beta})$ with

$$\mathbf{Q}_{\boldsymbol{i}} = [i_k = 0]\mathbf{P}'_{\boldsymbol{i}[k\mapsto 0]} + \mathbf{P}'_{\boldsymbol{i}[k\mapsto i_k+1]}$$

$$= [i_k = 0]\mathbf{P}_{\boldsymbol{i}[k\mapsto 0]} + \mathbf{P}_{\boldsymbol{i}[k\mapsto i_k+1]}\alpha_k^{\max(0,i_k+1-|\mathbf{P}|_k+1)}$$
$$= [i_k = 0]\mathbf{P}_{\boldsymbol{i}[k\mapsto 0]} + \mathbf{P}_{\boldsymbol{i}[k\mapsto i_k+1]}\alpha_k^{\max(0,2+i_k-|\mathbf{P}|_k)}$$

for all $\boldsymbol{i} < |\mathbf{Q}|$. It turns out that for $|\mathbf{P}|_k \geq 3$, we have $i_k \leq |\mathbf{Q}|_k - 1 \leq |\mathbf{P}|_k - 2$ and thus $\max(0, 2 + i_k - |\mathbf{P}|_k) = 0$. Hence the above equation for $\mathbf{Q}_{\boldsymbol{i}}$ holds generally, as desired. Note that we have to ensure $|\mathbf{Q}|_k \geq 2$ due to the special case for $i_k = 0$, which necessitates the case split on $|\mathbf{P}|_k \geq 3$.

For the case $\mathsf{if}\, E\, \{P_1\}\, \mathsf{else}\, \{P_2\}$, we have $[\![P]\!](\mu)(\boldsymbol{x}) = [\![P_1]\!](\mu|_E)(\boldsymbol{x}) + [\![P_2]\!](\mu|_{\neg E})(\boldsymbol{x})$. Suppose $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathsf{geo}}\ [\![P_1]\!]^{\mathsf{geo}}\ \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathsf{geo}}\ [\![P_2]\!]^{\mathsf{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ and $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a join of $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$. By the definition of the join relation, we have $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. Then we find with the inductive hypothesis:

$$\begin{aligned}[\![\mathsf{if}\, E\, \{P_1\}\, \mathsf{else}\, \{P_2\}]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) &= [\![P_1]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathsf{geo}}\right) + [\![P_2]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathsf{geo}}\right)\\ &\preceq \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})\\ &\preceq \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})\end{aligned}$$

Finally, consider loops $\mathsf{while}\, E\, \{B\}$. Suppose that there is a c-contraction invariant $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ such that $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E^{\mathsf{geo}}\ [\![B]\!]^{\mathsf{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ with $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$. We know $[\![B]\!](\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E) \preceq \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ by the induction hypothesis. The fixed point equation of $\mathsf{while}\, E\, \{B\}$ yields by monotonicity of the standard semantics:

$$\begin{aligned}[\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right) &= \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E + [\![\mathsf{while}\, E\, \{B\}]\!]\left([\![B]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E\right)\right)\\ &\preceq \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathsf{geo}} + [\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})\right)\\ &\preceq \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathsf{geo}} + [\![\mathsf{while}\, E\, \{B\}]\!]\left(c \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right)\\ &= \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathsf{geo}} + c \cdot [\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right)\end{aligned}$$

which implies that $[\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right) \preceq \frac{1}{1-c}\, \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathsf{geo}}$. By monotonicity, we also get

$$\begin{aligned}[\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\right) &\preceq [\![\mathsf{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right)\\ &\preceq \frac{1}{1-c}\, \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathsf{geo}} = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})\end{aligned}$$

For bounds on the normalized distribution, note that the normalizing constant equals

$$1 - [\![P]\!]_{\lightning}(\mu)(\lightning) = [\![P]\!]_{\lightning}(\mu)(\mathbb{N}^n_{\lightning}) - [\![P]\!]_{\lightning}(\mu)(\lightning) = [\![P]\!]_{\lightning}(\mu)(\mathbb{N}^n) = [\![P]\!](\mu)(\mathbb{N}^n)$$

by almost sure termination, the fact that μ is a probability distribution, and the definition of $[\![P]\!]$. The lower bound on $[\![P]\!](\mu)(\mathbb{N}^n)$ follows from Theorem 5.2.2 and the upper bound $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})(\mathbb{N}^n)$ from the above soundness proof. As a consequence, we obtain the desired bounds on the normalized distribution. □

6.5.4 Loop-free Fragment

It is instructive to look at when $[\![-]\!]^{\mathsf{geo}}$ is precise, i.e. when the right-hand side of the relation is unique and equals the $[\![-]\!]$ semantics. It turns out that this is the case for programs without loops, which is easy to see from the soundness proof.

Theorem 6.5.6 (Precision for loop-free fragment)**.** *Let P be a loop-free program. Assume that the strict* Join* *relation is used in the $[\![-]\!]^{\text{geo}}$ semantics of* if $-$ {$-$} else {$-$}*. Then for all* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$*, there is a unique* $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ *such that* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\text{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ *and* $[\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$*. Furthermore, each* $\beta_i \in \{0, \alpha_i\}$*.*

Proof. The proof is by induction on the structure of P. We strengthen the inductive hypothesis to state that $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha})$ and each $\beta_i \in \{0, \alpha_i\}$. In the soundness proof, it was shown that the restriction $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\text{geo}}$ is correct and that the semantics of skip, $X_k := 0$, $X_k \mathrel{+}= c$, $X_k \mathrel{\dot{-}}= 1$, and fail are exact. Only for fail and $X_k := 0$ can it happen that $\boldsymbol{\beta} \neq \boldsymbol{\alpha}$ (because some β_k's are set to zero). In both cases, $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ because $\mathbf{Q}_{k:|\mathbf{Q}|_k - 1}$ for all k where $\beta_k = 0 < \alpha_k$.

For $P = P_1; P_2$, the inductive hypothesis ensures that $[\![P_1]\!]^{\text{geo}}$ and $[\![P_2]\!]^{\text{geo}}$ are precise and thus $[\![P]\!]^{\text{geo}}$ is precise as well. The stronger claim $\beta_i \in \{0, \alpha_i\}$ also clearly holds for them because they can only ever set a component of $\boldsymbol{\alpha}$ to zero.

The only construct that is generally not precise is the conditional if E {P_1} else {P_2}. Suppose that $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\text{geo}}\ [\![P_1]\!]^{\text{geo}}\ \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\text{geo}}\ [\![P_2]\!]^{\text{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$, which are precise by inductive hypothesis and $\gamma_i, \delta_i \in \{0, \alpha_i\}$ for all i.

We claim that $\big(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}), \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})\big) \in$ Join* implies $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. By the definition of the strict join relation, $\boldsymbol{\beta} = \max(\boldsymbol{\gamma}, \boldsymbol{\delta}) \leq \boldsymbol{\alpha}$. Thus each $\gamma_i, \delta_i \in \{0, \beta_i\} \subseteq \{0, \alpha_i\}$. If either of them is zero, then our strengthened induction hypothesis ensures that $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\beta})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) = \mathsf{EGD}(\mathbf{S}, \boldsymbol{\beta})$ and similarly for any expansions $\mathbf{R}', \mathbf{S}'$ of them. Then by the definition of the strict join relation, $\mathbf{Q} = \mathbf{R}' + \mathbf{S}'$ for some expansions $\mathbf{R}', \mathbf{S}'$. As a consequence:

$$\begin{aligned}
\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) &= \mathsf{EGD}(\mathbf{R}', \boldsymbol{\beta}) + \mathsf{EGD}(\mathbf{S}', \boldsymbol{\beta}) \\
&= \mathsf{EGD}(\mathbf{R}', \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}', \boldsymbol{\delta}) \\
&= \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \\
&= [\![P_1]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\text{geo}}\right) + [\![P_2]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\text{geo}}\right) \\
&= [\![\text{if } E \text{ } \{P_1\} \text{ else } \{P_2\}]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))
\end{aligned}$$

Furthermore $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\alpha})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) = \mathsf{EGD}(\mathbf{S}, \boldsymbol{\alpha})$ by the inductive hypothesis. As a consequence, $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha}) = \mathsf{EGD}(\mathbf{R}', \boldsymbol{\alpha}) + \mathsf{EGD}(\mathbf{S}', \boldsymbol{\alpha}) = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. □

6.5.5 Sufficient Conditions for the Existence of Bounds

When does the constraint problem arising from our semantics actually have a solution? This is hard to characterize in general because the feasible region can be complex, as discussed above. Furthermore, a useful purely syntactic criterion is unlikely to exist because even for the simple asymmetric random walk

$$\begin{array}{l}
\text{while } X_1 > 0 \text{ } \{ \\
\quad \{X_1 \mathrel{+}= 1\}\ [\rho]\ \{X_1 \mathrel{\dot{-}}= 1\} \\
\}
\end{array}$$

a bound can be found if and only if $\rho < \frac{1}{2}$ (see Example 6.4.3).

Nevertheless, we have succeeded in identifying a sufficient criterion for the existence of an EGD loop bound: roughly speaking, if the difference between new and old values of the variables is the same in each loop iteration, and if there is a linear ranking supermartingale [Chatterjee et al. 2016], then an EGD bound exists. The precise conditions are stated in the following theorem.

Theorem 6.5.7. *Suppose* $L = \mathsf{while}\, E\, \{B\}$ *satisfies the following properties:*

1. *B only contains increment and decrement statements* ($X_k \mathrel{+}= 1$, $X_k \mathrel{\dot{-}}= 1$)*, probabilistic branching* ($\mathsf{if}\, \mathsf{flip}(\dots)\, \{\dots\}\, \mathsf{else}\, \{\dots\}$)*, and* fail*,*
2. *E occurring guarantees that* $X_1 \geq a_1 \wedge \cdots \wedge X_n \geq a_n$ *where* $a_i \in \mathbb{N}$ *is the maximum number of decrements of* X_i *in any program path through B, and*
3. *there is a conical (linear with nonnegative coefficients) combination* $\sum_{i=1}^{n} \lambda_i X_i$ *whose expected value decreases after every loop iteration for any initial assignment* $\boldsymbol{x}$ *that can enter the loop:*

$$\mathbb{E}_{X \sim [\![B]\!](\mathrm{Dirac}(\boldsymbol{x})|_E)} \left[\sum_{i=1}^{n} \lambda_i X_i \right] < \sum_{i=1}^{n} \lambda_i x_i \quad \text{for all } \boldsymbol{x} \in \mathbb{N}^n \text{ with } x_i \geq a_i$$

Then, for any initial $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$*, there is a solution* $(\mathbf{Q}, \boldsymbol{\beta})$ *to* $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ $[\![L]\!]^{\mathrm{geo}}$ $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$*.*

Proof. Since the loop body is loop-free, we have $[\![B]\!]^{\mathrm{geo}} = [\![B]\!]$ Theorem 6.5.6 (if the strict join relation is used in the semantics of $\mathsf{if} - \{-\}\, \mathsf{else}\, \{-\}$). Conditions (1) and (2) ensure that the distribution of the change is the same in every loop iteration. (The technical condition (2) is needed to deal with the fact that decrements are clamped at zero.) We will show that a contraction invariant of the form $\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ exists, with $\mathbf{R} \in \mathbb{R}$ viewed as a $(1, \dots, 1)$-tensor.

We will see that $\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ with $\boldsymbol{\gamma} = \mathbf{1}_n$ is "almost" a solution to the constraints of a c-contraction invariant. "Almost" in the sense that we relax some $<$-constraints to $\leq$-constraints, for example $\gamma_i < 1$ and $c < 1$. Via a gradient argument, we can show that taking an ε-small step from $\boldsymbol{\gamma} = \mathbf{1}_n$ in the direction of $-\boldsymbol{\lambda} = (-\lambda_1, \dots, -\lambda_n)$, i.e. the negated coefficients from the linear combination, improves all of the $\leq$-constraints. The full proof works as follows.

Let b_i be the maximum number of increments of X_i in any program path through B. We claim that there is a finite subprobability distribution D on $\Delta := \bigtimes_{i=1}^{n} \{-a_i, \dots, b_i\}$ such that $[\![B]\!](\mathrm{Dirac}(\boldsymbol{x})) = \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \mathrm{Dirac}(\boldsymbol{x} + \boldsymbol{\delta})$ for all $\boldsymbol{x}$ with $x_i \geq a_i$. The latter restriction is necessary because for $X_i = 0$, the decrement $X_i \mathrel{\dot{-}}= 1$ has no effect. Since we assume that E guarantees that $X_i \geq a_i$, we find $[\![B]\!](\mathrm{Dirac}(\boldsymbol{x})|_E)(\boldsymbol{w}) \leq [\boldsymbol{w} - \boldsymbol{x} \in \Delta] \cdot D(\boldsymbol{w} - \boldsymbol{x})$ for all $\boldsymbol{w} \in \mathbb{N}^n$. By a weighted sum of all $\boldsymbol{x}$, we can construct any measure μ on $\mathbb{N}^n$ and obtain

$$\begin{aligned}
[\![B]\!](\mu|_E)(\boldsymbol{w}) &\leq \sum_{\boldsymbol{x} \in \mathbb{N}^n} \mu|_E(\boldsymbol{x}) \cdot [\boldsymbol{w} - \boldsymbol{x} \in \Delta] \cdot D(\boldsymbol{w} - \boldsymbol{x}) \\
&= \sum_{\boldsymbol{\delta} \in \mathbb{N}^n} \mu|_E(\boldsymbol{w} - \boldsymbol{\delta}) \cdot [\boldsymbol{\delta} \in \Delta] \cdot D(\boldsymbol{\delta}) \\
&= \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \mu|_E(\boldsymbol{w} - \boldsymbol{\delta})
\end{aligned}$$

Let's see what one loop iteration on $\mu = \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ does, where $\mathbf{R} \in \mathbb{R}$ is viewed as a $(1, \dots, 1)$-tensor. The restriction $\mu|_E$ is represented by $\mathsf{EGD}(\mathbf{R}', \boldsymbol{\gamma})$ with $\mathbf{R}'_{\boldsymbol{x}} \leq [x_1 \geq a_1] \cdots [x_n \geq a_n] \cdot \mathbf{R}\boldsymbol{\gamma}^{\boldsymbol{x}}$ since E implies $\boldsymbol{x} \geq \boldsymbol{a}$. The semantics of B on $\mu|_E$ is then $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}') := [\![B]\!]^{\mathrm{geo}}\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\big|_E^{\mathrm{geo}}\right)$. By Theorem 6.5.6, $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}')$ is precise and we find for $\boldsymbol{x} < |\mathbf{S}|$:

$$\begin{aligned}
\mathbf{S}_{\boldsymbol{x}} &:= \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \mathsf{EGD}(\mathbf{R}', \boldsymbol{\gamma})(\boldsymbol{x} - \boldsymbol{\delta}) \\
&\leq \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot [\boldsymbol{x} - \boldsymbol{\delta} \geq \boldsymbol{a}] \cdot \mathbf{R} \cdot \boldsymbol{\gamma}^{\boldsymbol{x} - \boldsymbol{\delta}} \\
&\leq \mathbf{R} \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \boldsymbol{\gamma}^{\boldsymbol{x} - \boldsymbol{\delta}}
\end{aligned}$$

By the invariant of the proof of Theorem 6.5.6, we have $\boldsymbol{\gamma}' \leq \boldsymbol{\gamma}$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}') = \mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma})$. The constraint $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}') \preceq_{\mathsf{EGD}} c \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ is thus implied by the following inequality:

$$\begin{aligned}
& \mathbf{R} \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \boldsymbol{\gamma}^{\boldsymbol{x} - \boldsymbol{\delta}} \leq c \cdot \mathbf{R} \cdot \boldsymbol{\gamma}^{\boldsymbol{x}} \\
\iff\ & L(\boldsymbol{\gamma}) := \sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \boldsymbol{\gamma}^{-\boldsymbol{\delta}} \leq c < 1
\end{aligned}$$

The left-hand side is ≤ 1 at the point $\boldsymbol{\gamma} = \mathbf{1}_n$ because D is a subprobability distribution. We will show that in any neighborhood of $\mathbf{1}_n$, there is a $\boldsymbol{\gamma} < \mathbf{1}_n$ such that the LHS is $L(\boldsymbol{\gamma}) < 1$. Then choosing $c := L(\boldsymbol{\gamma})$ ensures $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}') \preceq_{\mathsf{EGD}} c \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$.

To show that such a point $\boldsymbol{\gamma} < \mathbf{1}_n$ exists, we exhibit a direction in which $L(\boldsymbol{\gamma})$ decreases, starting at $\boldsymbol{\gamma} = \mathbf{1}_n$. For this it is enough to find $\boldsymbol{\lambda} \in \mathbb{R}^n_{\geq 0}$ such that $\boldsymbol{\lambda}^\top \nabla L(\mathbf{1}_n) > 0$. We compute the derivatives:

$$\begin{aligned}
\frac{\partial}{\partial \gamma_k} L(\boldsymbol{\gamma})\big|_{\boldsymbol{\gamma} = \mathbf{1}_n} &= -\sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \delta_k \cdot \gamma_1^{-\delta_1} \cdots \gamma_k^{-\delta_k - 1} \cdots \gamma_n^{-\delta_n}\Big|_{\boldsymbol{\gamma} = \mathbf{1}_n} \\
&= -\sum_{\boldsymbol{\delta} \in \Delta} D(\boldsymbol{\delta}) \cdot \delta_k \\
&= -\sum_{\boldsymbol{X} \in \mathbb{N}^n} [\boldsymbol{X} - \boldsymbol{x} \in \Delta] \cdot D(\boldsymbol{X} - \boldsymbol{x}) \cdot (X_k - x_k) \\
&= -\sum_{\boldsymbol{X} \in \mathbb{N}^n} [\![B]\!](\mathsf{Dirac}(\boldsymbol{x})|_E)(\boldsymbol{X}) \cdot (X_k - x_k) \\
&= -\mathbb{E}_{\boldsymbol{X} \sim [\![B]\!](\mathsf{Dirac}(\boldsymbol{x})|_E)}[X_k - x_k]
\end{aligned}$$

where we picked $\boldsymbol{x} \geq \boldsymbol{a}$. In other words, this is the negative expected change of X_k after running one loop iteration on $\boldsymbol{x} \geq \boldsymbol{a}$. Therefore, we have

$$\begin{aligned}
\boldsymbol{\lambda}^\top \nabla L(\mathbf{1}_n) &\geq -\sum_{k=1}^n \lambda_k \mathbb{E}_{\boldsymbol{X} \sim [\![B]\!](\mathsf{Dirac}(\boldsymbol{x})|_E)}[X_k - x_k] \\
&= -\mathbb{E}_{\boldsymbol{X} \sim [\![B]\!](\mathsf{Dirac}(\boldsymbol{x})|_E)}\left[\sum_{k=1}^n \lambda_k (X_k - x_k)\right]
\end{aligned}$$

$$> 0$$

Without loss of generality, we can assume that $\lambda_k > 0$ for all k because increasing each λ_k by a small enough amount will not invalidate the above inequality since $X_k - x_k$ is bounded. This means that there is an $\varepsilon > 0$ such that for all $t \in (0, \varepsilon)$, we have $L(\mathbf{1}_n - t\boldsymbol{\lambda}) < 1$. Then all the constraints $L(\boldsymbol{\gamma}) < 1$ are satisfied at $\boldsymbol{\gamma} = 1 - t\boldsymbol{\lambda} < \mathbf{1}_n$ for all $t \in (0, \varepsilon)$. Having found $\boldsymbol{\gamma}$, we can choose $c := L(\boldsymbol{\gamma})$ and all the constraints arising from $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\gamma}') \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$ are satisfied.

The only remaining constraint is $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$. Concretely, this gives rise to the constraints $\boldsymbol{\alpha} \leq \boldsymbol{\gamma}$ and to the constraints of the form (for all $\boldsymbol{x} \leq |\mathbf{P}|$):

$$\mathbf{P}_{\boldsymbol{x}} \leq \mathbf{R} \cdot \gamma_1^{x_1} \cdots \gamma_n^{x_n}$$

Since $\boldsymbol{\gamma}$ can be chosen to be strictly positive, this can be satisfied by choosing $\mathbf{R} \geq \max_{\boldsymbol{x} \leq |\mathbf{P}|} \frac{\mathbf{P}_{\boldsymbol{x}}}{\gamma_1^{x_1} \cdots \gamma_n^{x_n}}$. The constraints $\boldsymbol{\alpha} \leq \boldsymbol{\gamma}$ can clearly be satisfied by choosing $\boldsymbol{\gamma}$ close enough to $\mathbf{1}_n$.

From the existence of a c-contraction invariant $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$, it directly follows that $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \; [\![L]\!]^{\mathrm{geo}} \; \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) := \mathsf{EGD}\left(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma}\right)\Big|^{\mathrm{geo}}_{\neg E}$. □

Although the assumptions of this theorem may seem restrictive, they are sufficient to prove the existence of bounds for the random walk (Example 5.1.1), which is nontrivial to analyze. In that case, $a_1 = 1, a_2 = 0$ and the expected value of X_1 decreases by $1 - 2r > 0$ in each loop iteration.

6.5.6 Necessary Conditions for the Existence of Bounds

We already mentioned that the existence of bounds is not guaranteed in general. An obvious necessary condition is that the true program distribution actually has tails that decay exponentially fast, i.e. can be bounded by an EGD. In particular, the moments of the program variables must be finite. Even simple programs can fail this requirement, as the following example demonstrates.

Example 6.5.8 (looping program with infinite moments)**.** The moments of the distribution of the following program are infinite:

```
X := 0; Y := 1;
while flip(1/2) {
    while Y > 0 {
        X += 1; Y -= 1
    };
    while X > 0 {
        X -= 1; Y += 2
    }
}
```

The loop body multiplies Y by 2 using the auxiliary variable X. This program is almost surely terminating and its probability mass function is given by $\mathbb{P}[X = 0, Y = 2^m] = 2^{-m}$

for $m > 0$ and zero elsewhere. Hence the k-th moment ($k \geq 1$) of Y is

$$\mathbb{E}[Y^k] = \sum_{m=1}^{\infty} \mathbb{P}[X = 0, Y = 2^m] \cdot 2^{km} \geq \sum_{m=1}^{\infty} 1 = \infty$$

The following theorem shows that the running time must satisfy the same requirements as the program variables regarding exponentially decaying tails and finite moments.

Theorem 6.5.9 (Necessary conditions on the running time). *If* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ $[\![P]\!]^{\mathrm{geo}}$ $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ *for an* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ *then the running time* T *of* P *on* $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ *can be bounded by an EGD as well. In particular, all its moments* $\mathbb{E}[T^k]$ *must be finite.*

Proof. We introduce an extra program variable $T := X_{n+1}$ to keep track of the running time. We transform the original program P to a program P^T which adds the statement T += 1 after each statement.

We claim that whenever $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ $[\![P]\!]^{\mathrm{geo}}$ $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, then there is a $\mathbf{Q}^T$ and $\beta^* \in (0, 1)$ such that for all $\beta' \in [\beta^*, 1)$, there is a $C(\beta') \in \mathbb{R}_{\geq 0}$ such that $C(\beta')$ tends to 1 as $\beta' \to 1$, and we have

$$\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \beta'))\ [\![P^T]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{Q}^T, (\boldsymbol{\beta}, \beta')\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(C(\beta') \cdot [\mathbf{Q}], (\boldsymbol{\beta}, \beta'))$$

where the notation $[\mathbf{P}]$ denotes the $(|\mathbf{P}|, 1)$-tensor with entry $[\mathbf{P}]_{n+1:0} = \mathbf{P}$. This claim implies the theorem because then $\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, 0)) \preceq_{\mathsf{EGD}} \mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \beta'))$ $[\![P^T]\!]^{\mathrm{geo}}$ $\mathsf{EGD}\left(\mathbf{Q}^T, (\boldsymbol{\beta}, \beta')\right)$ for all $\beta' \in [\beta^*, 1)$. By Lemma 6.5.4, there is also an $\mathsf{EGD}(\mathbf{Q}', (\boldsymbol{\beta}, \beta'))$ such that $\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, 0))$ $[\![P^T]\!]^{\mathrm{geo}}$ $\mathsf{EGD}(\mathbf{Q}', (\boldsymbol{\beta}, \beta'))$.

We observe that increasing $T = X_{n+1}$ transforms $\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \alpha'))$ to

$$\mathsf{EGD}([\mathbf{0}, \mathbf{P}], (\boldsymbol{\alpha}, \alpha')) \preceq_{\mathsf{EGD}} \mathsf{EGD}\left(\frac{1}{\alpha'}[\mathbf{P}], (\boldsymbol{\alpha}, \alpha')\right)$$

where $[\mathbf{0}, \mathbf{P}]$ is the $(|\mathbf{P}|, 2)$-tensor with $[\mathbf{0}, \mathbf{P}]_0 = \mathbf{0}$ and $[\mathbf{0}, \mathbf{P}]_1 = \mathbf{P}$. Using this observation, we prove the above claim by induction on the structure of P.

For simple statements P (i.e. not containing other statements), the claim is proven as follows. Since $P^T = (P; T \mathrel{+}= 1)$, we have

$$\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \beta'))\ [\![P^T]\!]^{\mathrm{geo}}\ \mathsf{EGD}([\mathbf{0}, \mathbf{Q}], (\boldsymbol{\beta}, \beta')) \preceq_{\mathsf{EGD}} \mathsf{EGD}\left(\frac{1}{\beta'}[\mathbf{Q}], (\boldsymbol{\beta}, \beta')\right)$$

and by setting $C(\beta') = \frac{1}{\beta'}$, the claim is proven.

For statements $P = P_1; P_2$, there is an $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ such that $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ $[\![P_1]\!]^{\mathrm{geo}}$ $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ $[\![P_2]\!]^{\mathrm{geo}}$ $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. By inductive hypothesis, there is a $\mathbf{R}^T$ and $\gamma^* \in (0, 1)$ with $\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \gamma'))$ $[\![P_1^T]\!]^{\mathrm{geo}}$ $\mathsf{EGD}\left(\mathbf{R}^T, (\boldsymbol{\gamma}, \gamma')\right)$ and $\mathsf{EGD}\left(\mathbf{R}^T, (\boldsymbol{\gamma}, \gamma')\right) \preceq_{\mathsf{EGD}}$ $\mathsf{EGD}\left(C_1(\gamma') \cdot [\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\right)$ for all $\gamma' \in [\gamma^*, 1)$. By the inductive hypothesis, there is also a tensor $\mathbf{Q}^T$ and $\beta^* \in [\gamma', 1)$ such that $\mathsf{EGD}\left([\mathbf{R}], (\boldsymbol{\gamma}, \beta')\right)$ $[\![P_2^T]\!]^{\mathrm{geo}}$ $\mathsf{EGD}\left(\mathbf{Q}^T, (\boldsymbol{\beta}, \beta')\right)$ and $\mathsf{EGD}\left(\mathbf{Q}^T, (\boldsymbol{\beta}, \beta')\right) \preceq_{\mathsf{EGD}}$ $\mathsf{EGD}(C_2(\beta')[\mathbf{Q}], (\boldsymbol{\beta}, \beta'))$ for all $\beta' \in [\beta^*, 1)$. Then the claim follows by Lemma 6.5.4 with $C(\beta') := C_1(\beta') \cdot C_2(\beta')$ by monotonicity of C.

For conditionals $P = \texttt{if}\ E\ \{P_1\}\ \texttt{else}\ \{P_2\}$, we have $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}\ [\![P_1]\!]^{\mathrm{geo}}\ \mathsf{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)$ and $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathrm{geo}}\ [\![P_2]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$. By inductive hypothesis, there are $\mathbf{R}^T, \mathbf{S}^T$ and $\gamma^* \in (0,1), \delta^* \in (0,1)$ such that

$$\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \gamma'))|_E^{\mathrm{geo}}\ [\![P_1^T]\!]^{\mathrm{geo}}\ \mathsf{EGD}\Big(\mathbf{R}^T, (\boldsymbol{\gamma}, \gamma')\Big) \preceq_{\mathsf{EGD}} \mathsf{EGD}\big(C_1(\gamma') \cdot [\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$$

for all $\gamma' \in [\gamma^*, 1)$ and

$$\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \delta'))|_E^{\mathrm{geo}}\ [\![P_2^T]\!]^{\mathrm{geo}}\ \mathsf{EGD}\Big(\mathbf{S}^T, (\boldsymbol{\delta}, \delta')\Big) \preceq_{\mathsf{EGD}} \mathsf{EGD}(C_2(\delta') \cdot [\mathbf{S}], (\boldsymbol{\delta}, \delta'))$$

for all $\delta' \in [\delta^*, 1)$. Set $\beta^* = \max(\gamma^*, \delta^*)$ and $C(\beta') = \max(C_1(\beta'), C_2(\beta'))$. Then $\mathsf{EGD}([\mathbf{Q}], (\boldsymbol{\beta}, \beta'))$ is a join of $\mathsf{EGD}\big([\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$ and $\mathsf{EGD}([\mathbf{S}], (\boldsymbol{\delta}, \delta'))$ and the claim follows from Lemmas 6.5.3 and 6.5.4.

For loops $P = \texttt{while}\ E\ \{B\}$, we know that there is a c-contraction invariant $\mathsf{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)$ with $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)$ and $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) = \mathsf{EGD}\big(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma}\big)|_{\neg E}^{\mathrm{geo}}$. We claim that $\mathsf{EGD}\big([\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$ is a contraction invariant for P^T for all $\gamma' \in [\gamma^*, 1)$ where γ^* will be determined later. By the inductive hypothesis, we have

$$\mathsf{EGD}\big([\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)|_E^{\mathrm{geo}}\ [\![B^T]\!]^{\mathrm{geo}}\ \mathsf{EGD}([\mathbf{S}'], (\boldsymbol{\delta}, \delta')) \preceq_{\mathsf{EGD}} \mathsf{EGD}\big(c \cdot C(\gamma') \cdot [\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$$

Since $\lim_{\gamma' \to 1} C(\gamma') = 1$, we can choose γ^* large enough such that $C(\gamma') < \frac{1}{c}$ for all $\gamma' \in [\gamma^*, 1)$ and thus $c'(\gamma') := c \cdot C(\gamma') < 1$ for all $\gamma' \in [\gamma^*, 1)$. Furthermore, we have $c'(\gamma') \to c$ as $\gamma' \to 1$. Clearly, $\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \gamma')) \preceq_{\mathsf{EGD}} \mathsf{EGD}\big([\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$ holds too. Thus $\mathsf{EGD}\big([\mathbf{R}], (\boldsymbol{\gamma}, \gamma')\big)$ is a $c'(\gamma')$-contraction invariant for all $\gamma' \in [\gamma^*, 1)$. So we obtain the bound:

$$\mathsf{EGD}([\mathbf{P}], (\boldsymbol{\alpha}, \gamma'))\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\left[\frac{\mathbf{R}}{1 - c'(\gamma')}\right], (\boldsymbol{\gamma}, \gamma')\right)\Big|_{\neg E}^{\mathrm{geo}}$$

The ratio between $\frac{1}{1-c'(\gamma')}$ and $\frac{1}{1-c}$ tends to 1 as $\gamma' \to 1$, as well. This finishes the proof. □

6.5.7 Convergence

Just like for the residual mass semantics, we would expect the geometric bound semantics to yield tighter and tighter bounds as we unroll loops further and further. It turns out that for the u-fold unrolling $P^{(u)}$ of a program P, we can find upper bounds whose distance from the true distribution decreases exponentially in u.

Theorem 6.5.10 ("Convergence of upper bounds"). *Let P be a program containing potentially nested loops and $P^{(u)}$ its u-fold unrolling. Suppose $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. Then there exist $A \in \mathbb{R}_{\geq 0}, C \in [0,1)$ and $\mathbf{Q}^{(u)}$ such that*

$$\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P^{(u)}]\!]^{\mathrm{geo}}\ \mathsf{EGD}\Big(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\Big) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \quad \textit{and}$$

$$\mathsf{EGD}\Big(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\Big) - [\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq A \cdot C^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

In particular, the distribution bound $\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right)$ *converges in total variation distance to the true distribution* $[\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))$*, as* $u \to \infty$*. Similarly, the* k*-th moment bound converges to the true moment for any* k*:*

$$\lim_{u\to\infty} \mathbb{E}_{X\sim\mathsf{EGD}\left(\mathbf{Q}^{(u)},\boldsymbol{\beta}\right)}[X_i^k] = \mathbb{E}_{X\sim[\![P]\!](\mathsf{EGD}(\mathbf{P},\boldsymbol{\alpha}))}[X_i^k]$$

Proof. Recall that there are two sources of imprecision in the geometric bound semantics: overapproximating loops and joining branches of conditionals. The imprecision of loop approximations decreases exponentially in u because by the definition of a contraction invariant, the bound on the probability distribution entering the loop decreases by the contraction factor $c < 1$ in each iteration. So after u unrollings, the imprecision is reduced by a factor of c^u. The imprecision of joins can also be reduced by picking larger and larger expansions of the EGDs involved. The complete proof that follows is technical, because we have to ensure that all these approximations compose in the right way, but these two ideas are the key ingredients.

First, note that $[\![P^{(u)}]\!] = [\![P]\!]$ since unrolling does not affect the semantics of a program. The proof works by induction on the structure of P. We strengthen the inductive hypothesis by asserting that A and C are monotonic functions of $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$.

Exactly representable statements. The statements $P = \mathsf{skip}$, $X_k \mathrel{+}= c$, $X_k \mathrel{\dot{-}}= 1$, $X_k := 0$, we have $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ [\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}))$. Since $[\![P]\!]^{\mathrm{geo}}$ is exact here, the claim holds trivially by setting $A = 0$ and $C = 0$.

Sequential composition. If $P = P_1; P_2$, we argue as follows: Since $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, there are $\mathbf{R}, \boldsymbol{\gamma}$ with $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P_1]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)\ [\![P_2]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. By the induction hypothesis, there are $A_1, C_1, \mathbf{R}^{(u)}$ with $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P_1^{(u)}]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - [\![P_1]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq A_1 \cdot C_1^u \cdot \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)$. Also by the induction hypothesis, there exist $A_2, C_2, \mathbf{Q}^{(u)}, \mathbf{Q}'$ such that $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\ [\![P_2]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ (by Lemma 6.5.4 since $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)$) and $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\ [\![P_2^{(u)}]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right)$ and $\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P_2]\!]\left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\right) \preceq A_2 \cdot C_2^u \cdot \mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta})$. Together, we find:

$$\begin{aligned}
&\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \\
&= \left(\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P_2]\!]\left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\right)\right) + \left([\![P_2]\!]\left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\right) - [\![P]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\right)\right) \\
&= \left(\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P_2]\!]\left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)\right)\right) + [\![P_2]\!]\left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - [\![P_1]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\right)\right) \\
&\preceq A_2 \cdot C_2^u \cdot \mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta}) + [\![P_2]\!]\left(A_1 \cdot C_1^u \cdot \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)\right) \\
&\preceq A_2 \cdot C_2^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) + A_1 \cdot C_1^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\
&\preceq (A_1 + A_2) \cdot \max(C_1, C_2)^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})
\end{aligned}$$

Furthermore, monotonicity of $A := A_1 + A_2$ and $C := \max(C_1, C_2)$ in $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ is preserved.

Conditionals. The case $\mathsf{if}\ E\ \{P_1\}\ \mathsf{else}\ \{P_2\}$ is similar. First, let $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}\ [\![P_1]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathrm{geo}}\ [\![P_2]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$. Next, let $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}\ [\![P_1^{(u)}]\!]^{\mathrm{geo}}$

$\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathrm{geo}}\ [\![P_2^{(u)}]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right)$ with

$$\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - [\![P_1]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}\right) \preceq A_1 \cdot C_1^u \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$$
$$\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right) - [\![P_2]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathrm{geo}}\right) \preceq A_2 \cdot C_2^u \cdot \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$$

Now $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a join of $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ via expansions $\mathbf{R}_*, \mathbf{S}_*$ and moreover $\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right)$ is a join of $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right)$ via expansions $\mathbf{R}_*^{(u)}$ and $\mathbf{S}_*^{(u)}$. We know that $\mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a join of $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$. By Lemma 6.5.3, there is a join $\mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta})$ of $\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right)$ and $\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right)$ with $\mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. By Lemma 6.5.11, we can find such a $\mathbf{Q}^{(u)}$ with

$$\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - \mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - \mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right) \preceq A_0 \cdot C_0^u \cdot \mathsf{EGD}(\mathbf{Q}', \boldsymbol{\beta}) \preceq A_0 \cdot C_0^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

where A_0 and C_0 will be chosen very shortly. Overall, we find

$$\begin{aligned}
&\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P]\!](\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \\
&= \left(\mathsf{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - \mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - \mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right)\right) \\
&\quad + \left(\mathsf{EGD}\left(\mathbf{R}^{(u)}, \boldsymbol{\gamma}\right) - [\![P_1]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_E^{\mathrm{geo}}\right)\right) + \left(\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\delta}\right) - [\![P_2]\!]\left(\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})|_{\neg E}^{\mathrm{geo}}\right)\right) \\
&\preceq A_0 \cdot C_0^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) + A_1 \cdot C_1^u \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma}) + A_2 \cdot C_2^u \cdot \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \\
&\preceq A_0 \cdot C_0^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) + \max(A_1, A_2) \cdot \max(C_1, C_2)^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\
&\preceq 2(A_1 + A_2) \cdot \max(C_1, C_2)^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})
\end{aligned}$$

by choosing $A_0 := \max(A_1, A_2)$ and $C_0 := \max(C_1, C_2)$. Monotonicity of $A := 2(A_1 + A_2)$ and $C := \max(C_1, C_2)$ in $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ also holds by induction hypothesis.

Loops. The crucial case is $P = \mathsf{while}\, E\, \{B\}$. Since $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})\ [\![P]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$, there exist $c, \mathbf{R}, \mathbf{S}, \boldsymbol{\gamma}, \boldsymbol{\delta}$ such that $\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$, and $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E^{\mathrm{geo}}\ [\![B]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta})$ and $\mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathsf{EGD}} \mathsf{EGD}(c \cdot \mathbf{R}, \boldsymbol{\gamma})$ and $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_{\neg E}^{\mathrm{geo}} = \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. By the induction hypothesis, $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E^{\mathrm{geo}}\ [\![B^{(u)}]\!]^{\mathrm{geo}}\ \mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\gamma}\right)$ with $\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\gamma}\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq_{\mathsf{EGD}} c \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$, so $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ is a c-contraction invariant for $\mathsf{while}\, E\, \{B^{(u)}\}$ too. Furthermore, the inductive hypothesis yields

$$\mathsf{EGD}\left(\mathbf{S}^{(u)}, \boldsymbol{\gamma}\right) - [\![B]\!]\left(\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})|_E^{\mathrm{geo}}\right) \preceq A_r \cdot C_r^u \cdot \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq A_r \cdot C_r^u \cdot c \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$$

Definition of $\mathbf{P}_v$ and $\mathbf{P}_v^{(u)}$. We define intermediate distributions within the unrolling, illustrated in Fig. 6.2. Define $\mathsf{EGD}(\mathbf{P}_0, \boldsymbol{\alpha}_0) = \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ and $\mathsf{EGD}(\mathbf{P}_{v-1}, \boldsymbol{\alpha}_{v-1})|_E^{\mathrm{geo}}\ [\![B]\!]^{\mathrm{geo}}\ \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v)$. This satisfies the inequality $\mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v) \preceq_{\mathsf{EGD}} c^v \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ by a simple inductive argument using the definition of contraction invariants. Define $\mathsf{EGD}\left(\mathbf{P}_0^{(u)}, \boldsymbol{\alpha}_0\right) = \mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha})$ and choose $\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)$ to satisfy the following conditions:

```
// EGD(P_0, α_0) = EGD(P, α)
if E {
  B
  // EGD(P_1, α_1)
  if E {
    B
    // EGD(P_2, α_2)
    ...
    // EGD(P_u, α_u)
    while E {B}
    // EGD(Q_u, β_u)
    ...
    // EGD(Q_2, β_2)
  } else {skip}
  // EGD(Q_1, β_1)
} else {skip}
// EGD(Q_0, β_0) = EGD(Q, β)
```

```
// EGD(P_0^(u), α_0) = EGD(P, α)
if E {
  B^(u)
  // EGD(P_1^(u), α_1)
  if E {
    B^(u)
    // EGD(P_2^(u), α_2)
    ...
    // EGD(P_u^(u), α_u)
    while E {B^(u)}
    // EGD(Q_u^(u), β_u)
    ...
    // EGD(Q_2^(u), β_2)
  } else {skip}
  // EGD(Q_1^(u), β_1)
} else {skip}
// EGD(Q_0^(u), β_0) = EGD(Q^(u), β)
```

Figure 6.2: Notation for the loop case in the proof of Theorem 6.5.10

1. $\mathsf{EGD}\left(\mathbf{P}_{v-1}^{(u)}, \boldsymbol{\alpha}_{v-1}\right)\Big|_E^{\text{geo}} \; [\![B^{(u)}]\!]^{\text{geo}} \; \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)$, which is possible by induction hypothesis.
2. $\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v)$, which is satisfiable by a simple inductive argument using monotonicity of the semantics.
3. $\mathsf{EGD}\left(\mathbf{P}_{v-1}^{(u)}, \boldsymbol{\alpha}_{v-1}\right)\Big|_E^{\text{geo}} \; [\![B]\!]^{\text{geo}} \; G_v^{(u)}$ with $G_v^{(u)} \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v)$, which can be satisfied by Lemma 6.5.4 because $\mathsf{EGD}\left(\mathbf{P}_{v-1}^{(u)}, \boldsymbol{\alpha}_{v-1}\right)\Big|_E^{\text{geo}} \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_{v-1}, \boldsymbol{\alpha}_{v-1})|_E^{\text{geo}}$ and because $\mathsf{EGD}(\mathbf{P}_{v-1}, \boldsymbol{\alpha}_{v-1})|_E^{\text{geo}} \; [\![B]\!]^{\text{geo}} \; \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v)$.

As a consequence of these conditions and the induction hypothesis, we find that

$$\begin{aligned}
\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right) - [\![B]\!] \left(\mathsf{EGD}\left(\mathbf{P}_{v-1}^{(u)}, \boldsymbol{\alpha}_{v-1}\right)\Big|_E^{\text{geo}}\right) &\preceq A_v \cdot C_v^u \cdot G_v^{(u)} \\
&\preceq A_v \cdot C_v^u \cdot \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v) \\
&\preceq A_v \cdot C_v^u \cdot c^v \cdot \mathsf{EGD}(\mathbf{R}, \gamma) \\
&\preceq A_r \cdot C_r^u \cdot c^v \cdot \mathsf{EGD}(\mathbf{R}, \gamma)
\end{aligned}$$

where the last steps follows from monotonicity of A and C.

Definition of $\mathbf{Q}_v^{(u)}$. Similarly, we want to define $\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right)$ such that

$$\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right) \llbracket (\mathsf{while}\, E\, \{B\})_v^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) \preceq_{\mathsf{EGD}} \frac{c^u}{1-c} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

We achieve this by the following inductive definition: For $v = u$, choose it so that

$$\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right) \llbracket \mathsf{while}\, E\, \{B^{(u)}\} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_u^{(u)}, \boldsymbol{\beta}\right)$$

Since $\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_u, \boldsymbol{\alpha}_u) \preceq_{\mathsf{EGD}} c^u \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$, we can choose $\mathbf{Q}_u^{(u)}$ such that

$$\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right) \llbracket \mathsf{while}\, E\, \{B^{(u)}\} \rrbracket^{\mathsf{geo}} c^u \cdot \mathsf{EGD}\left(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma}\right)\Big|_{\neg E}^{\mathsf{geo}} \preceq_{\mathsf{EGD}} \frac{c^u}{1-c} \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

by Lemma 6.5.4. For $v < u$, use Lemma 6.5.11 to choose $\mathbf{Q}_v^{(u)}$ such that $\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right)$ is a join of $\mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right)$ and $\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}\right)\Big|_{\neg E}^{\mathsf{geo}}$ such that

$$\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\mathsf{geo}} - \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) \preceq \frac{B_v}{1-c} \cdot c^v \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

where $B_v = A_r \cdot C_r^u \cdot c(1-c)$. The lemma can be applied because $\frac{c^v}{1-c} \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a bound on the join of $\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\mathsf{geo}}$ and $\mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right)$. This is because

$$\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\mathsf{geo}} \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_v, \boldsymbol{\alpha}_v)|_{\neg E}^{\mathsf{geo}} \preceq_{\mathsf{EGD}} c^v \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\Big|_{\neg E}^{\mathsf{geo}} = c^v \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$$

and $\mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) \preceq_{\mathsf{EGD}} \frac{c^{v+1}}{1-c} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ by inductive hypothesis, which implies that the sum $\left(c^v + \frac{c^{v+1}}{1-c}\right) \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ is a valid join and equals $\frac{c^v}{1-c} \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. Hence a $\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) \preceq_{\mathsf{EGD}} \frac{c^v}{1-c} \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ with the desired properties exists. Finally, define $\mathbf{Q}^{(u)} := \mathbf{Q}_0^{(u)}$.

Inductively, it is easy to see that

$$\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_u\right) \llbracket (\mathsf{while}\, E\, \{B\})_v^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right)$$

In the base case $v = u$, this is clear from the definition. For the inductive step, we can assume $\mathsf{EGD}\left(\mathbf{P}_{v+1}^{(u)}, \boldsymbol{\alpha}_{v+1}\right) \llbracket (\mathsf{while}\, E\, \{B\})_{v+1}^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right)$. By definition, we have $\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\mathsf{geo}} \llbracket B^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{P}_{v+1}^{(u)}, \boldsymbol{\alpha}_{v+1}\right)$ and thus

$$\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\mathsf{geo}} \llbracket B^{(u)}; (\mathsf{while}\, E\, \{B\})_{v+1}^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right)$$

Since $\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right)$ is a join of $\mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right)$ and $\mathsf{EGD}(\mathbf{P}_v^u, \boldsymbol{\alpha})\Big|_{\neg E}^{\mathsf{geo}}$, the semantics of $\mathsf{if} - \{-\}\, \mathsf{else}\, \{-\}$ implies

$$\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right) \llbracket (\mathsf{while}\, E\, \{B\})_v^{(u)} \rrbracket^{\mathsf{geo}} \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right)$$

because $(\text{while}\, E\, \{B\})_v^{(u)} = \text{if}\, E\, \{B^{(u)}; (\text{while}\, E\, \{B\})_{v+1}^{(u)}\}\, \text{else}\, \{\text{skip}\}$.

Actual proof for the loop case. Consider $P^{(u)} = (\text{while}\, E\, \{B\})_0^{(u)}$. The proof for the while case is by induction on v in $(\text{while}\, E\, \{B\})_v^{(u)}$. We need to strengthen our induction hypothesis to the following:

$$\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - [\![P]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\right) \preceq \left(c^u + 2A_r C_r^u \cdot \frac{c^{v+1}}{1-c}\right) \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \tag{6.2}$$

We first consider the base case $v = u$: We have $(\text{while}\, E\, \{B\})_u^{(u)} = \text{while}\, E\, \{B^{(u)}\}$. We remarked above that $\mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ is a c-contraction invariant for $\text{while}\, E\, \{B^{(u)}\}$. Furthermore we have $\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right) \preceq_{\mathsf{EGD}} \mathsf{EGD}(\mathbf{P}_u, \boldsymbol{\alpha}_u) \preceq_{\mathsf{EGD}} c^u \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})$ and thus:

$$\begin{aligned}\mathsf{EGD}\left(\mathbf{P}_u^{(u)}, \boldsymbol{\alpha}_u\right)\, [\![(\text{while}\, E\, \{B\})_u^{(u)}]\!]^{\text{geo}}\, c^u \cdot \mathsf{EGD}\left(\frac{\mathbf{R}}{1-c}, \boldsymbol{\gamma}\right)\Big|_{\neg E}^{\text{geo}} \\ = c^u \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \preceq \left(c^u + 2A_r C_r^u \frac{c^{u+1}}{1-c}\right) \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta})\end{aligned}$$

Now for the inductive step $v < u$: By the loop unrolling property, we have:

$$\begin{aligned}&[\![\text{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\right) \\ &= [\![\text{while}\, E\, \{B\}]\!]\left([\![B]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\text{geo}}\right)\right) + \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\text{geo}}\end{aligned}$$

Furthermore, since $\text{while}\, E\, \{B\}_v^{(u)}$ has the same semantics as $\text{while}\, E\, \{B\}$, we find:

$$\begin{aligned}&\mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - [\![\text{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\right) \\ &= \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - \left([\![\text{while}\, E\, \{B\}]\!]\left([\![B]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\text{geo}}\right)\right) + \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\text{geo}}\right) \\ &= \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\text{geo}} - \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) \\ &\quad + \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) - [\![\text{while}\, E\, \{B\}]\!]\left([\![B]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\text{geo}}\right)\right) \\ &= \mathsf{EGD}\left(\mathbf{Q}_v^{(u)}, \boldsymbol{\beta}\right) - \mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_{\neg E}^{\text{geo}} - \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) \\ &\quad + \mathsf{EGD}\left(\mathbf{Q}_{v+1}^{(u)}, \boldsymbol{\beta}\right) - [\![\text{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}\left(\mathbf{P}_{v+1}^{(u)}, \boldsymbol{\alpha}_{v+1}\right)\right) \\ &\quad + [\![\text{while}\, E\, \{B\}]\!]\left(\mathsf{EGD}\left(\mathbf{P}_{v+1}^{(u)}, \boldsymbol{\alpha}_{v+1}\right) - [\![B]\!]\left(\mathsf{EGD}\left(\mathbf{P}_v^{(u)}, \boldsymbol{\alpha}_v\right)\Big|_E^{\text{geo}}\right)\right) \\ &\preceq \frac{B_v}{1-c} \cdot c^v \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\ &\quad + \left(c^u + 2A_r C_r^u \cdot \frac{c^{v+2}}{1-c}\right) \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\ &\quad + [\![\text{while}\, E\, \{B\}]\!]\left(A_r \cdot C_r^u \cdot c^{v+1} \cdot \mathsf{EGD}(\mathbf{R}, \boldsymbol{\gamma})\right) \\ &\preceq A_r \cdot C_r^u \cdot c^{v+1} \cdot \mathsf{EGD}(\mathbf{Q}, \beta)\end{aligned}$$

$$
\begin{aligned}
&+ \left(c^u + 2A_r C_r^u \cdot \frac{c^{v+2}}{1-c}\right) \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\
&+ A_r \cdot C_r^u \cdot c^{v+1} \cdot \mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)\Big|_{\neg E}^{\mathrm{geo}} \\
&\preceq \left(c^u + 2A_r C_r^u \cdot \frac{c^{v+2}}{1-c}\right) \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) + 2 \cdot A_r \cdot C_r^u \cdot c^{v+1} \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \\
&\preceq \left(c^u + 2A_r C_r^u \cdot \frac{c^{v+1}}{1-c}\right) \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})
\end{aligned}
$$

This proves Eq. (6.2). As a consequence, we have (since $\mathbf{P} = \mathbf{P}_0^{(u)}$ and $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0$):

$$
\mathrm{EGD}\left(\mathbf{Q}^{(u)}, \boldsymbol{\beta}\right) - [\![P]\!](\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})) \preceq \left(c^u + 2A_r C_r^u \frac{c}{1-c}\right) \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta}) \preceq AC^u \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})
$$

with $A := 1 + \frac{2A_r c}{1-c}$ and $C := \max(c, C_r)$. Furthermore, A, C can be chosen to be monotonic functions of $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$, by choosing $\mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)$ as a monotonic function of $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})$. This finishes the induction and the proof. □

The following lemma was needed in the proof.

Lemma 6.5.11. *Let $A > 0$ and suppose we have $\big(\mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big), \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})\big) \in$ Join. Then there is a tensor $\mathbf{P}$ and $\boldsymbol{\alpha} \in [0,1)^n$ such that $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ and $\big(\mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big), \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}), \mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha})\big) \in$ Join and $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big) - \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq A \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$.*

Proof. Let $\mathrm{EGD}\big(\mathbf{R}^*, \boldsymbol{\gamma}\big), \mathrm{EGD}(\mathbf{S}^*, \boldsymbol{\delta})$ be the expansions of $\mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big)$ and $\mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta})$ such that $\mathbf{R}^* + \mathbf{S}^* = \mathbf{Q}$. We set $\boldsymbol{\alpha} = \max(\boldsymbol{\gamma}, \boldsymbol{\delta})$. Clearly, $\boldsymbol{\alpha} \leq \boldsymbol{\beta}$ and thus $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\alpha}) \preceq_{\mathrm{EGD}} \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$.

The proof is by induction on $n = |\boldsymbol{\alpha}|$. If $n = 0$ then $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) = \mathbf{P} = \mathbf{R} + \mathbf{S} \in \mathbb{R}$ and $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big) - \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}) = \mathbf{P} - \mathbf{R} - \mathbf{S} = 0$.

If $n > 0$, we assume without loss of generality that $\gamma_n \leq \delta_n$ and thus $\alpha_n = \delta_n$. Let $\boldsymbol{\alpha}' := (\alpha_1, \ldots, \alpha_{n-1})$, $\boldsymbol{\beta}' := (\beta_1, \ldots, \beta_{n-1})$, $\boldsymbol{\gamma}' := (\gamma_1, \ldots, \gamma_{n-1})$, and $\boldsymbol{\delta}' := (\delta_1, \ldots, \delta_{n-1})$. In this proof we will write $\mathbf{T}_j$ for $\mathbf{T}_{n:j}$, i.e. the j-the component in the n-th dimension of a tensor t.

Let $m > |\mathbf{Q}|_n$ to be chosen later and let $\mathrm{EGD}\left(\mathbf{R}^{(m)}, \boldsymbol{\gamma}\right), \mathrm{EGD}\left(\mathbf{S}^{(m)}, \boldsymbol{\delta}\right), \mathrm{EGD}\left(\mathbf{Q}^{(m)}, \boldsymbol{\beta}\right)$ be the expansions of $\mathrm{EGD}\big(\mathbf{R}^*, \boldsymbol{\gamma}\big), \mathrm{EGD}(\mathbf{S}^*, \boldsymbol{\delta})$, and $\mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$ to size m in dimension n. Then for $j < m$, we can find by the inductive hypothesis some tensor $\mathbf{P}_j$ such that $\left(\mathrm{EGD}\left(\mathbf{R}_j^{(m)}, \boldsymbol{\gamma}'\right), \mathrm{EGD}\left(\mathbf{S}_j^{(m)}, \boldsymbol{\delta}'\right), \mathrm{EGD}\left(\mathbf{P}_j, \boldsymbol{\alpha}'\right)\right) \in$ Join and

$$
\mathrm{EGD}\left(\mathbf{P}_j^{(m)}, \boldsymbol{\alpha}'\right) - \mathrm{EGD}\left(\mathbf{R}_j^{(m)}, \boldsymbol{\gamma}'\right) - \mathrm{EGD}\left(\mathbf{S}_j^{(m)}, \boldsymbol{\delta}'\right) \preceq A/2 \cdot \mathrm{EGD}\left(\mathbf{Q}_j^{(m)}, \boldsymbol{\beta}'\right)
$$

Let $[m_1] \times \cdots \times [m_{n-1}]$ be the maximum dimensions of $\mathbf{P}_j$. Without loss of generality, we can assume that all $\mathbf{R}_j^{(m)}, \mathbf{S}_j^{(m)}, \mathbf{P}_j$ are expanded to these maximum dimensions. Collecting the $\mathbf{P}_j$'s into a big vector yields $\mathbf{P} = (\mathbf{P}_0, \ldots, \mathbf{P}_{|\mathbf{Q}|_n})$. We claim $\mathrm{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathrm{EGD}\big(\mathbf{R}, \boldsymbol{\gamma}\big) - \mathrm{EGD}(\mathbf{S}, \boldsymbol{\delta}) \preceq A \cdot \mathrm{EGD}(\mathbf{Q}, \boldsymbol{\beta})$. We show this by looking at the j-th component, i.e.

$\mu_j(\boldsymbol{x}') := \mu(\boldsymbol{x}', j)$ for $\boldsymbol{x}' \in \mathbb{N}^{n-1}$. For $j < m$, this follows from the definition. For $j \geq m$, we analyze two cases.

If $\gamma_n = \delta_n$, then $\alpha_n = \gamma_n = \delta_n$ by definition. We choose $m = |\mathbf{Q}|_n$ and find for $j \geq m = |\mathbf{Q}|_n$:

$$\begin{aligned}
&\mathsf{EGD}\left(\mathbf{P}_{m-1} \cdot \alpha_n^{j-m+1}, \boldsymbol{\alpha}'\right) - \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1} \cdot \gamma_n^{j-m+1}, \boldsymbol{\gamma}'\right) - \mathsf{EGD}\left(\mathbf{S}^{(m)}_{m-1} \cdot \delta_n^{j-m+1}, \boldsymbol{\delta}'\right) \\
&= \left(\mathsf{EGD}(\mathbf{P}_{m-1}, \boldsymbol{\alpha}') - \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\gamma}'\right) - \mathsf{EGD}\left(\mathbf{S}^{(m)}_{m-1}, \boldsymbol{\delta}'\right)\right) \cdot \alpha_n^{j-m+1} \\
&\leq A \cdot \mathsf{EGD}(\mathbf{Q}_{m-1}, \boldsymbol{\beta}') \cdot \beta_n^{j-m+1}
\end{aligned}$$

If $\gamma_n < \delta_n = \alpha_n$, we find for $j \geq m$:

$$\begin{aligned}
&\mathsf{EGD}(\mathbf{P}_{m-1}, \boldsymbol{\alpha}') \cdot \alpha_n^{j-m+1} - \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\gamma}'\right) \cdot \gamma_n^{j-m+1} - \mathsf{EGD}\left(\mathbf{S}^{(m)}_{m-1}, \boldsymbol{\delta}'\right) \cdot \delta_n^{j-m+1} \\
&= \left(\mathsf{EGD}(\mathbf{P}_{m-1}, \boldsymbol{\alpha}') - \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\gamma}'\right) - \mathsf{EGD}\left(\mathbf{S}^{(m)}_{m-1}, \boldsymbol{\delta}'\right)\right) \cdot \alpha_n^{j-m+1} \\
&\quad + \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\alpha}'\right) \cdot \alpha_n^{j-m+1} - \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\gamma}'\right) \cdot \gamma_n^{j-m+1} \\
&\leq A/2 \cdot \mathsf{EGD}(\mathbf{Q}_{m-1}, \boldsymbol{\beta}') \cdot \beta_n^{j-m+1} + \mathsf{EGD}\left(\mathbf{R}^{(m)}_{m-1}, \boldsymbol{\alpha}'\right) \cdot \alpha_n^{j-m+1} \\
&= A/2 \cdot \mathsf{EGD}\left(\mathbf{Q}_{|\mathbf{Q}|_n-1} \cdot \beta_n^{m-|\mathbf{Q}|_n}, \boldsymbol{\beta}'\right) \cdot \beta_n^{j-m+1} + \mathsf{EGD}\left(\mathbf{R}^{*}_{|\mathbf{Q}|_n-1} \cdot \gamma_n^{m-|\mathbf{Q}|_n}, \boldsymbol{\alpha}'\right) \cdot \alpha_n^{j-m+1} \\
&\leq A/2 \cdot \mathsf{EGD}\left(\mathbf{Q}_{|\mathbf{Q}|_n-1}, \boldsymbol{\beta}'\right) \cdot \beta_n^{j-|\mathbf{Q}|_n+1} + \mathsf{EGD}\left(\mathbf{R}^{*}_{|\mathbf{Q}|_n-1}, \boldsymbol{\beta}'\right) \cdot \alpha_n^{j-|\mathbf{Q}|_n+1} \cdot \left(\frac{\gamma_n}{\alpha_n}\right)^{m-|\mathbf{Q}|_n} \\
&\leq \left(A/2 + \left(\frac{\gamma_n}{\alpha_n}\right)^{m-|\mathbf{Q}|_n}\right) \cdot \mathsf{EGD}\left(\mathbf{Q}_{|\mathbf{Q}|_n-1}, \boldsymbol{\beta}'\right) \cdot \beta_n^{j-|\mathbf{Q}|_n+1} \\
&\leq A \cdot \mathsf{EGD}\left(\mathbf{Q}_{|\mathbf{Q}|_n-1}, \boldsymbol{\beta}'\right) \cdot \beta_n^{j-|\mathbf{Q}|_n+1}
\end{aligned}$$

where the last step follows from $\left(\frac{\gamma_n}{\alpha_n}\right)^{m-|\mathbf{Q}|_n} \to 0$ as $m \to \infty$, so we can choose m large enough such that the bound holds for all $j \geq m$.

So the bound holds for $j < m$ and $j \geq m$, and thus we find, as desired:

$$\mathsf{EGD}(\mathbf{P}, \boldsymbol{\alpha}) - \mathsf{EGD}\left(\mathbf{R}, \boldsymbol{\gamma}\right) - \mathsf{EGD}(\mathbf{S}, \boldsymbol{\delta}) \leq A \cdot \mathsf{EGD}(\mathbf{Q}, \boldsymbol{\alpha}) \qquad \square$$

Note that this theorem does *not* prove that the parameters $\boldsymbol{\beta}$ of the geometric tails converge to the best possible ones. In fact, unrolling does not improve the decay rate at all, in general. In specific circumstances, it can improve the tail bounds, for example, if the loop is always finite. Then no probability mass will enter the loop after sufficiently many unrollings, and the loop contraction invariant will be zero. But in general, the derived tail bounds may differ arbitrarily from the true tails, as shown in the next example.

Example 6.5.12 (Nonoptimal tail bounds)**.** As an example of nonoptimal tail bounds, consider the following program where $\rho \in (0, 1)$:

$$\begin{aligned}
&X_1 := 0; X_2 := 0; \\
&\mathsf{while}\ \mathsf{flip}(\rho)\ \{
\end{aligned}$$

$$
\begin{aligned}
&\quad X_1 \mathrel{+}= 1; X_2 \mathrel{+}= 1\\
&\};\\
&\textsf{while}\, X_1 > 0 \wedge X_2 > 0 \,\{\\
&\quad X_1 \mathrel{\dot{-}}= 1; X_2 \mathrel{\dot{-}}= 1\\
&\}
\end{aligned}
$$

It is clear that at the end of the program, $X_1 = 0$ and $X_2 = 0$ almost surely. Throughout the program, the same operations are applied to them. However, the geometric bound semantics cannot keep track of this correlation completely because the structure of EGDs can only express correlations in the finite initial block and not in the tails. Hence the best tail bounds we get for X_1 and X_2 are of the form $O((\rho + \delta)^n)$ for any $\delta > 0$, whereas the true tails are zero. So the tail bounds can be arbitrarily far from the true tails. Note, however, that the existence of geometric tails is already useful knowledge: it tells us that the distribution is not heavy-tailed and that all moments are finite.

This program can easily be extended to an example where the geometric bound semantics completely fails even though an EGD bound for the whole program exists.

Example 6.5.13 (Failure of geometric bound semantics)**.** Recall that at the end of the program from Example 6.5.12, the true distribution is $\text{Dirac}(0, 0)$, but the geometric bound semantics finds a bound with nonzero tails. If we append $\textsf{if}\, X_1 = 0 \,\{\textsf{skip}\}\, \textsf{else}\, \{\textsf{diverge}\}$ to the program, this does not affect the true distribution (because $X_1 = 0$ almost surely), but the geometric bound semantics will now fail because $\textsf{diverge} := \textsf{while true}\, \{\textsf{skip}\}$ does not admit a geometric bound since it has infinite expected running time.

6.6 Implementation

The geometric bound semantics is implemented in the same tool ***Diabolo*** ("**Di**screte **Di**stribution **A**nalysis via **Bo**unds, supporting **L**oops and **O**bservations") as the residual mass semantics for discrete SGCL (see Section 5.4.1). The code is archived on Zenodo [Zaiser 2024] and the latest version is available on Github: `https://github.com/fzaiser/diabolo`.

Diabolo takes a discrete SGCL program as input and outputs bounds on probability masses, moments, and tail asymptotics of the posterior distribution of a specified program variable. The output and computational effort can be controlled with various options: the loop unrolling limit u, the dimension d of the contraction invariants to be synthesized, an optimization objective, as well as the solver and optimizer to be used for the polynomial constraints. The optimization objective is the quantity whose bound should be optimized: the probability mass, the expected value, or the asymptotic decay rates. Depending on which of these is chosen, the best EGD bound can vary significantly (see Section 6.4.1). The tool is implemented in Rust [Matsakis and Klock 2014] for performance and uses exact rational number computations instead of floating point to ensure the correctness of the results. We describe a few practical aspects of the implementation below, most of them pertaining to the geometric bound semantics.

Nondeterminism in the semantics As pointed out in Section 6.4, there are two places in the geometric bound semantics where choices have to be made: branching

and loops. To generate the polynomial constraints arising from while loops, we need to choose the size of the contraction invariants $\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$. It is usually best to choose the dimensions of $\mathbf{R}$ to be as small as possible (often 1): this reduces both the number of constraints and constraint variables, which facilitates solving. Some programs only have larger contraction invariants, but the dimension rarely needs to be greater than 2. In terms of quality of the bounds, increasing the unrolling limit has a much greater effect (cf. Theorem 6.5.10). In the semantics of if statements, the choice is in the size of the expansion of the EGDs in the Join relation. In Diabolo, we choose the smallest expansion for speed and memory usage reasons. The size of the EGDs in a loop depends mostly on the size of the contraction invariant, so an increase there automatically reduces the imprecision in Join.

Approximating the support In the constraint problem arising from a c-contraction invariant $\mathrm{EGD}(\mathbf{R}, \boldsymbol{\gamma})$, the constraint variables corresponding to the entries of $\mathbf{R}$ are "easy" because they only occur linearly, whereas the constraints typically contain higher-degree polynomials in $\boldsymbol{\gamma}$. For this reason, it is desirable to determine the decay rates $\boldsymbol{\gamma}$ in advance, if possible.

If the program variable X_i has finite support $\{a_i, a_i + 1, \ldots, b_i\} \subset \mathbb{N}$, we can set the decay rate γ_i to zero since the tails of X_i's distribution are zero. But to be able to represent its distribution with $\gamma_i = 0$, we have to choose at least $|\mathbf{R}|_i = b_i + 1$. (In this case, the user-provided dimension d for the contraction invariant is overridden.) Furthermore, we can infer from the support that $\mathbf{R}_{i:j} = \mathbf{0}$ for $j < a_i$. To reap these benefits, we need to analyze the support of the random variables occurring in the program, as described in Section 5.4.3.

6.6.1 Constraint Solving

How can we solve the polynomial constraints arising from the geometric bound semantics? As this is a decidable problem known as the *existential theory of the reals*, we first tried to use SMT solvers, such as Z3 [de Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022], and the dedicated tool QEPCAD [Brown 2003]. Out of these, Z3 performed the best, but it was only able to solve the simplest problems and scaled badly.

Numerical solvers Given that even simple programs lead to dozens and sometimes hundreds of constraints, we resorted to numerical solutions instead. The best solver for our purposes turned out to be IPOPT: a library for large-scale nonlinear optimization [Wächter and Biegler 2006]. One issue with IPOPT are rounding errors, so it sometimes returns solutions that are not exactly feasible. To address this, we tighten the constraints by a small margin before handing them over to IPOPT and check the feasibility of the returned solution with exact rational arithmetic. Another disadvantage of IPOPT is that it cannot prove infeasibility – only exact solvers like Z3 and QEPCAD can do so. However, its impressive scalability makes up for these shortcomings.

We also developed a custom solver that transforms the constraint problem into an unconstrained optimization problem (details in Section 6.6.3) and applies the ADAM optimizer [Kingma and Ba 2015], a popular gradient descent method. We also explored

other off-the-shelf solvers for polynomial constraints: IBEX, a rigorous solver based on interval arithmetic, and constrained optimization methods in scipy, the Python scientific computing library; but neither of them performed well. Diabolo includes the solvers IPOPT (the default), our ADAM-based solver, and Z3.

Nested loops Nested loops lead to cyclic $\leq$-constraints on the decay rates, e.g. $\alpha \leq \beta \wedge \beta \leq \alpha$. Numerical solvers struggle with such indirect equality constraints, so we added a preprocessing step to detect them and replace the equal variables with a single representative.

Reducing the cost of unrolling Unrolling loops is essential to obtain tight bounds on posterior masses and moments (it does not affect tail bounds). On the other hand, unrolling increases the complexity of the constraint problem considerably, both in terms of number of variables and constraints. A key observation from Theorem 6.5.10 can mitigate this problem: the variables that occur nonlinearly, i.e. decay rates and contraction factors, need not be changed as the unrolling count increases. As a consequence, we first solve the constraint problem without unrolling and need only solve a *linear* constraint problem for higher unrolling limits. Since linear programming solvers are much faster and more robust than nonlinear solvers, this approach significantly reduces numerical issues and computation time. In fact, without this technique, several benchmarks in Section 6.7.2 would not be solvable at all.

6.6.2 Optimization

Since the geometric bound semantics is nondeterministic, there are many EGD bounds for a given program. Which one is the best depends on what quantity we want to optimize. In Diabolo, the user can specify one of the following optimization objectives to minimize: the bound on the total probability mass, on the expected value, or on the tail asymptotics (i.e. the decay rate).

IPOPT and the ADAM-based solver can also be used for optimization. Moreover, we apply the linear programming solver CBC by the COIN-OR project to optimize the linear variables (keeping the nonlinear ones fixed), which is very fast. By default, Diabolo runs IPOPT, the ADAM-based optimizer, and the linear solver in this order, each improving the solution of the previous one. The ADAM-based optimizer can be slow for larger programs, in which case the user may decide to skip it. However, it often finds better tail bounds than IPOPT, which is why it is included by default.

The constraints arising from the geometric bound semantics are linear in the entries of initial blocks of EGDs, which is typically the majority of the constraint variables. Hence, once a feasible solution has been found, it makes sense to optimize the linear variables using a linear programming (LP) solver, while keeping the nonlinear variables fixed. Since LP solvers are much faster than general nonlinear optimization solvers, this is a cheap and effective postprocessing step.

6.6.3 Our ADAM-based Solver and Optimizer

The success of the (gradient-based) ADAM optimizer in high-dimensional problems in machine learning inspired us to apply it to optimizing our bounds as well. Since ADAM works on unconstrained optimization problems, we first need to reduce the constrained optimization problem to an unconstrained one. Our idea is to add penalty terms to the objective function for constraints that are violated or almost violated, to steer the solver towards feasible points. Specifically, for each constraint $f(\boldsymbol{x}) \leq 0$, we add a penalty term to the objective function: $\exp(\lambda \mathrm{dist}_f(\boldsymbol{x}))$, where $\mathrm{dist}_f(\boldsymbol{x}) = \frac{f(\boldsymbol{x})}{\|\nabla f(\boldsymbol{x})\|}$ is an approximation of the (signed) distance to the constraint boundary. This term is near zero if the constraint is comfortably satisfied, but large if the constraint is violated. In that sense it is similar to *penalty function methods* used in constrained optimization, except penalty functions are *exactly* zero when the constraint is satisfied. The parameter λ is increased during the optimization: we take it to be the current iteration number. A higher λ leads to a "sharper" transition from near zero to a very high penalty at the constraint boundary.

How do we find an initial point for the optimization? In many cases (see proof of Theorem 6.5.7), there is a point with $\boldsymbol{\beta} = \mathbf{1}_n$ that is close to a feasible point. Thus we simply initialize $\boldsymbol{\beta}$ at the point $0.999 \cdot \mathbf{1}_n$ (i.e. very close to one) in our implementation. This initialization is typically good enough for our optimizer to quickly find a feasible point. However, as mentioned in the main text, IPOPT is better at finding feasible points than our ADAM-based approach. The primary strength of the ADAM-based optimizer is to improve the geometric bounds, particularly the tail bounds.

6.7 Empirical Evaluation

In this section, we evaluate our two methods in practice to answer the following four questions:

1. How often is the geometric bound semantics applicable in practice? (Section 6.7.1)
2. How tight are the bounds in practice? (Section 6.7.2)
3. How do our methods perform compared to previous work? (Section 6.7.3)
4. How do our two semantics perform compared to each other? (Section 6.7.4)

All benchmarks and code to reproduce the results are available [Zaiser 2024].

6.7.1 Applicability of the Geometric Bound Semantics

In this section, we empirically investigate the *existence* of geometric bounds, i.e. how often the constraints arising from the geometric bound semantics can be solved. If some bound can be found at all, Theorem 6.5.10 ensures that it can be made arbitrarily tight by increasing the unrolling limit. (This aspect of the *quality* of bounds is studied in Sections 6.7.2 and 6.7.4.) While Theorem 6.5.7 provides sufficient conditions for existence, we want to test the applicability of the geometric bound semantics in practice in a *systematic* way.

Benchmark selection For this purpose, we collected benchmarks from the Github repositories of several probabilistic programming languages: Polar [Moosbrugger et al. 2022], Prodigy [Klinkenberg et al. 2024], and PSI [Gehr et al. 2016]. Note that a benchmark being available in a tool's repository does not mean that the tool can solve it (we mark such benchmarks with an asterisk in Table 6.4).

We searched all benchmarks from these repositories for the keyword `while`, in order to find benchmarks with loops. We manually filtered out benchmarks whose loops are actually bounded or that make essential use of continuous distributions (15), negative integers (4), comparisons of two variables (7), or multiplication (1), because our language cannot express these. Some benchmarks using these features could still be translated in other ways to an equivalent program in our language.

The Polar benchmarks are all unnested loops without conditioning. Since Polar is also used for loop invariant synthesis, most of their operate on loops of the form while true $\{\ldots\}$. We excluded benchmarks with such infinite loops. Since PSI cannot handle unbounded loops, the loops in its repository were all either artificially bounded or could not be solved by PSI. We removed the artificial bounds as we are interested in unbounded loops, which explains why PSI cannot solve any of "its own" benchmarks.

Polar and Prodigy can handle symbolic inputs and symbolic parameters to some extent, which our techniques cannot. One benchmark (`polar/fair_biased_coin`) used symbolic parameters, which we replaced with concrete values. Several benchmarks from Polar and Prodigy have symbolic inputs, i.e. they are parametric in the initial values in $\mathbb{N}$ of the variables. Since our method cannot reason parametrically about the input, we instead put a $\mathrm{Geometric}(1/2)$ distribution on such inputs, to cover all possible values. Of course, this yields less information than a method computing a symbolic result that is valid for all input values. But if an EGD bound can be found for this input distribution, then an EGD bound can be found for each possible input value because any Dirac distribution on the input can be bounded by a scaled version of the geometric distribution: $\mathrm{Dirac}(m) \preceq 2^m \cdot \mathrm{Geometric}(1/2)$. Thus, for the question of the existence of bounds, we consider this a valid approach. Note that reasoning about such a geometric distribution on the input is nontrivial (and much harder than reasoning about fixed input values). For instance, Polar cannot always handle it even for benchmarks where it supports the version with symbolic input values.

We ended up with 43 benchmarks: 9 from Polar, 11 from Prodigy, 9 from PSI, and we added 14 of our own. They include standard probabilistic loop examples (in particular, all examples from this chapter, and variations thereof), nested loops, and real-world algorithms, such as probabilistic self-stabilization protocols [Herman 1990; Israeli and Jalfon 1990; Beauquier et al. 1999].

Methodology We ran our tool for the geometric bound semantics on all of them with a timeout of 10 seconds, an unrolling limit of zero, and no optimization (since we are only concerned with existence of bounds, not their quality, for now). For each benchmark, we recorded the time it took to compute the bounds, or any errors.

Results The results of the geometric bound semantics, as implemented in Diabolo, on all benchmarks are shown in Table 6.4, along with statistics about the input program and

Table 6.4: Existence of geometric bounds: for each benchmark, we present some statistics about the program, the constraint problem arising from the geometric bound semantics, and the solving time, if successful. (*: cannot be solved by the respective tool; #Var: number of program variables; #Stmt: number of statements; Obs?: presence of observations; #Constr: number of constraints; #ConstrVar: number of constraint variables; Time: time to compute bounds (or error); timeout: 5 minutes exceeded; infeasible: no solution to the constraints exists; "inf. runtime": expected runtime is infinite, implies infeasible.)

Benchmark	#Var	#Stmt	Obs?	#Constr	#ConstrVar	Time
polar/c4B_t303*	3	21	✗	6	13	0.02 s
polar/coupon_collector2	5	12	✗	2	10	0.02 s
polar/fair_biased_coin	2	5	✗	2	5	0.01 s
polar/geometric (Ex. 6.5.8)	4	13	✗	21	18	✗ (inf. runtime)
polar/las_vegas_search	3	7	✗	20	23	0.02 s
polar/linear01*	1	5	✗	2	3	0.02 s
polar/rabin*	4	19	✗	45	34	✗ (infeasible)
polar/random_walk_2d	3	11	✗	11	7	✗ (inf. runtime)
polar/simple_loop	2	8	✗	1	3	0.01 s
prodigy/bit_flip_conditioning	5	13	✓	3	18	0.02 s
prodigy/brp_obs	2	9	✓	93	8	2.79 s
prodigy/condand*	2	6	✗	4	6	0.02 s
prodigy/dep_bern	3	8	✗	4	5	0.05 s
prodigy/endless_conditioning	1	7	✓	1	2	0.09 s
prodigy/geometric	2	6	✗	3	4	0.11 s
prodigy/ky_die	2	37	✗	42	43	0.03 s
prodigy/n_geometric	2	6	✗	3	4	0.01 s
prodigy/nested_while	3	11	✗	12	10	✗ (infeasible)
prodigy/random_walk	2	7	✗	2	4	✗ (inf. runtime)
prodigy/trivial_iid	2	5	✗	12	4	0.03 s
psi/beauquier-etal3*	14	39	✗	64	8194	2.43 s
psi/cav-example7*	2	16	✗	4	6	0.15 s
psi/dieCond*	2	7	✓	2	8	0.02 s
psi/ex3*	1	5	✗	2	3	0.02 s
psi/ex4*	2	5	✗	1	2	0.06 s
psi/fourcards*	7	21	✗	112	257	0.09 s
psi/herman3*	8	33	✗	16	258	0.04 s
psi/israeli-jalfon3*	8	30	✗	4	130	0.03 s
psi/israeli-jalfon5*	12	48	✗	116	4098	2.99 s
ours/1d-asym-rw (Ex. 5.1.1)	2	7	✗	2	4	0.14 s
ours/2d-asym-rw	3	12	✗	4	5	0.03 s
ours/3d-asym-rw	4	17	✗	126	59	0.18 s
ours/asym-rw-conditioning	3	10	✓	6	6	0.02 s
ours/coupon-collector5	7	18	✗	75	162	0.04 s
ours/double-geo (Ex. 6.5.2)	2	5	✗	2	4	0.01 s
ours/geometric (Ex. 6.4.2)	1	3	✗	1	3	0.02 s
ours/grid	2	6	✗	11	12	0.22 s
ours/herman5	12	38	✗	214	6146	✗ (timeout)
ours/imprecise_tails (Ex. 6.5.12)	2	8	✗	14	13	0.10 s
ours/israeli-jalfon4	10	39	✗	25	770	0.09 s
ours/nested	2	7	✗	12	10	0.02 s
ours/sub-geom	1	3	✗	3	4	0.01 s
ours/sum-geos	2	5	✗	3	5	0.03 s

the constraint problem arising from the geometric bound semantics. Diabolo was able to solve 37 out of the 43 benchmarks (86%); 5 failed because no EGD bound exists; and 1 timed out due to the complexity of the constraint problem. For the 5 failing benchmarks, Z3 could prove that no EGD bound exists (for invariant sizes $d = 1, 2, 3, 4$), these are marked as "infeasible". For 3 of them, we manually determined that the expected runtime is infinite, which implies no EGD bound of any size exists (Theorem 6.5.9). We suspect that no EGD bound (for any invariant size) exists for the other 2 benchmarks either. Among the solved benchmarks, the solution time was at most 3 seconds and typically much less. This demonstrates that our geometric bound semantics is also of practical relevance, in addition to its nice theoretical properties.

6.7.2 Quality of the Geometric Bounds

Methodology Beyond the mere *existence*, we are also interested in the *quality* of the bounds. To this end, we ran Diabolo again on each benchmark where a bound could be found, once optimizing the bound on the expected value, once optimizing the tail bound. The configuration options were left at the default settings with an unrolling limit of 30 for most benchmarks. For expectation bounds, we modified the settings for 23 benchmarks, to adjust the unrolling limit (reported in Table 6.5), invariant size to 2, 3, or 4 (if smaller values failed or yielded bad bounds), and to skip the ADAM-based optimizer (due to it being slow for some benchmarks). For tail bounds, we set the unrolling limit to 1 and modified the default settings for 15 benchmarks to adjust the invariant size to 2, 3, or 4 and to skip the ADAM-based optimizer (due to it being slow for some benchmarks).

Results The results are shown in Table 6.5. We mark with "†" all examples that, to our knowledge, could not be solved automatically before, in the sense of bounding the distribution's moments and tails without user intervention. Note that no other existing tool is able to obtain *exponential* tail bounds like ours on any benchmark (unless the tails are zero). One can see that all bounds are nontrivial and the upper and lower bounds on the expected value are usually close together. However, it seems to be harder to find good bounds for benchmarks where the tails of the distribution decay slowly, such as the asymmetric random walks (ours/*-asym-rw). Most of the tail bounds are also very close to the theoretical optimum where the exact tail bound could be manually determined. This demonstrates that our geometric bound semantics generally yields useful bounds.

6.7.3 Comparison with Previous Work

Comparison with Polar We compare the geometric bound semantics (as implemented in Diabolo) with Polar [Moosbrugger et al. 2022], the only other tool that can be used to bound moments. In fact, Polar can compute moments exactly, but does not bound the tail asymptotics.[2] We only included benchmarks from Table 6.5 where Polar can compute the expected value (EV), at least in theory. (We had to extend Polar slightly to support a geometric distribution as the initial distribution.) Polar was run with a

[2] One could derive the tail asymptotic bound $O(n^{-k})$ from the k-th moment via Markov's inequality. But this asymptotic bound is very weak as it applies to any distribution with a finite k-th moment.

Table 6.5: Diabolo's bounds on the expected value and the tail asymptotics for the benchmarks from Section 6.7.1 where bounds exist. Results are rounded to 4 significant digits. (#U: unrolling limit; EV: expected value; "?": ground truth could not be determined; "†": could not be solved by an automatic tool before, in the sense of bounding the distribution's moments and tails without user intervention; timeout: 5 minutes exceeded.)

Benchmark	#U	True EV	EV bound	Time	True tail	Tail bound	Time
polar/c4B_t303 †	30	0.1787...	[0.1787, 0.1788]	0.24 s	?	$O(0.5001^n)$	0.12 s
polar/coupon_collector2	30	2	[1.999, 2.001]	0.16 s	$\Theta(0.5^n)$	$O(0.5071^n)$	0.11 s
polar/fair_biased_coin	30	0.5	[0.4999, 0.5001]	0.03 s	0	0	0.02 s
polar/las_vegas_search	200	20	[19.98, 26.79]	2.29 s	$\Theta(0.9523...^n)$	$O(0.9524^n)$	0.72 s
polar/linear01 †	30	0.375	[0.3749, 0.3751]	0.03 s	0	0	0.03 s
polar/simple_loop	30	1.3	[1.299, 1.301]	0.02 s	0	0	0.02 s
prodigy/bit_flip_conditioning	30	1.254...	[1.254, 1.255]	0.28 s	?	$O(0.6254^n)$	0.19 s
prodigy/brp_obs †	30	4.989...e-10	[4.989e-10, 1.489e-09]	0.61 s	0	0	3.83 s
prodigy/condand †	30	0.75	[0.7499, 0.7501]	0.15 s	?	$O(0.5001^n)$	0.07 s
prodigy/dep_bern †	30	0.5	[0.4999, 0.5124]	1.00 s	$\Theta(0.3333...^n)$	$O(0.34^n)$	0.17 s
prodigy/endless_conditioning	30	undef	undef	0.15 s	0	0	0.13 s
prodigy/geometric	30	2	[1.999, 2.007]	1.09 s	$\Theta(0.5^n)$	$O(0.5066^n)$	0.60 s
prodigy/ky_die	30	3.5	[3.499, 3.501]	0.18 s	0	0	0.08 s
prodigy/n_geometric †	30	1	[0.998, 1.068]	0.08 s	$\Theta(0.6666...^n)$	$O(0.673^n)$	0.07 s
prodigy/trivial_iid †	30	3.5	[3.499, 3.503]	2.60 s	$\Theta(0.8318...^n)$	$O(0.836^n)$	0.16 s
psi/beauquier-etal3	30	?	✗ (timeout)	t/o	?	$O(0.7952^n)$	17.64 s
psi/cav-example7	80	10.41...	[10.41, 10.51]	1.71 s	?	$O(0.8927^n)$	0.12 s
psi/dieCond (Ex. 2.2.1) †	40	1.5	[1.499, 1.501]	0.15 s	$\Theta(0.3333...^n)$	$O(0.3395^n)$	0.13 s
psi/ex3	30	0.6666...	[0.6666, 0.6667]	0.04 s	0	0	0.06 s
psi/ex4	30	0.6666...	[0.6666, 0.6667]	0.23 s	0	0	0.27 s
psi/fourcards	30	0.2642...	[0.264, 0.2648]	0.46 s	0	0	0.34 s
psi/herman3	30	1.333...	[1.333, 1.334]	62.08 s	?	$O(0.5002^n)$	0.48 s
psi/israeli-jalfon3	30	0.6666...	[0.6666, 0.6668]	1.71 s	?	$O(0.2501^n)$	0.15 s
psi/israeli-jalfon5	30	?	✗ (timeout)	t/o	?	$O(0.6583^n)$	7.13 s
ours/1d-asym-rw (Ex. 5.1.1) †	70	2	[1.999, 2.542]	1.47 s	$\Theta(\frac{0.8660...^n}{n^{1.5}})$	$O(0.8682^n)$	0.12 s
ours/2d-asym-rw †	70	?	[7.394, 100.4]	79.60 s	?	$O(0.9385^n)$	4.88 s
ours/3d-asym-rw †	30	?	[9.443, 6.85e+05]	289.94 s	?	$O(0.9812^n)$	6.35 s
ours/asym-rw-conditioning †	70	2.444...	[2.316, 2.588]	4.85 s	0	0	0.05 s
ours/coupon-collector5	80	11.41...	[11.41, 11.56]	47.93 s	$\Theta(0.8^n)$	$O(0.8002^n)$	15.56 s
ours/double-geo (Ex. 6.5.2) †	30	2	[1.999, 2.01]	0.32 s	$\Theta(0.7071...^n)$	$O(0.711^n)$	0.12 s
ours/geometric (Ex. 6.4.2) †	30	1	[0.9999, 1.006]	0.12 s	$\Theta(0.5^n)$	$O(0.5037^n)$	0.10 s
ours/grid †	30	2.75	[2.749, 2.766]	2.39 s	?	$O(0.5285^n)$	0.30 s
ours/imprecise_tails (Ex. 6.5.12) †	30	0	[0, 0.007054]	2.36 s	0	$O(0.5001^n)$	0.18 s
ours/israeli-jalfon4	30	?	[1.457, 1.458]	121.04 s	?	$O(0.5011^n)$	0.44 s
ours/nested †	5	?	[0.9152, 10.57]	12.59 s	?	$O(0.4998^n)$	0.40 s
ours/sub-geom †	30	0.6666...	[0.6666, 0.6667]	0.06 s	?	$O(0.5001^n)$	0.10 s
ours/sum-geos	80	8	[7.998, 8.001]	2.91 s	$\Theta(0.875^n)$	$O(0.8751^n)$	0.30 s

timeout of 5 minutes, like Diabolo. The results (Table 6.6) demonstrate that Diabolo is often faster, and sometimes much faster, than Polar and applicable to more benchmarks.

Comparison of tail bounds There are existing program analyses that can bound the tail distribution of the running time of a probabilistic program (see Section 2.6), but only one [Chatterjee et al. 2016] achieves exponential bounds $O(c^n)$ for $c < 1$ like our bounds. This is particularly interesting as their assumptions (bounded differences and the existence of a linear ranking supermartingale) are remarkably similar to the conditions of our Theorem 6.5.7. Since the code for the experiments by Chatterjee et al. [2016] is not available, we do a manual comparison on Examples 5.1.1 and 6.4.2.

Table 6.6: Comparison of Diabolo with Polar [Moosbrugger et al. 2022] on the benchmarks from Table 6.5 where Polar can compute the expected value (EV). Diabolo's results are the same as in Table 6.5, but listed again for an easier comparison. (t/o: timeout of 5 minutes exceeded.)

Benchmark	Polar		Diabolo (ours)	
	Exact EV	Time	EV bound	Time
polar/coupon_collector2	2	0.26 s	[1.999, 2.001]	0.16 s
polar/fair_biased_coin	1/2	0.41 s	[0.4999, 0.5001]	0.03 s
polar/las_vegas_search	✗	t/o	[19.98, 26.79]	2.29 s
polar/simple_loop	13/10	0.20 s	[1.299, 1.301]	0.02 s
prodigy/geometric	2	0.46 s	[1.999, 2.007]	1.09 s
prodigy/ky_die	✗	t/o	[3.499, 3.501]	0.18 s
psi/beauquier-etal3	✗	t/o	✗	t/o
psi/cav-example7	✗	t/o	[10.41, 10.51]	1.71 s
psi/ex3	2/3	0.58 s	[0.6666, 0.6667]	0.04 s
psi/ex4	2/3	0.21 s	[0.6666, 0.6667]	0.23 s
psi/fourcards	✗	t/o	[0.264, 0.2648]	0.46 s
psi/herman3	4/3	19.53 s	[1.333, 1.334]	62.08 s
psi/israeli-jalfon3	2/3	19.40 s	[0.6666, 0.6668]	1.71 s
psi/israeli-jalfon5	✗	t/o	✗	t/o
ours/coupon-collector5	137/12	74.86 s	[11.41, 11.56]	47.93 s
ours/geometric	1	0.59 s	[0.9999, 1.006]	0.12 s
ours/israeli-jalfon4	✗	t/o	[1.457, 1.458]	121.04 s
ours/sum-geos	8	0.19 s	[7.998, 8.001]	2.91 s

For the example of the asymmetric random walk (Example 5.1.1), their method yields the tail bound $O\left(\left(\sqrt{\exp(-(1-2r)^2)}\right)^n\right)$, where r is the bias of the random walk. This is derived as follows: we take X_2 as the ranking supermartingale M and find that $M_{i+1} = M_i + r \cdot 1 + (1-r) \cdot (-1) = M_i - (1-2r)$. As discussed in Section 2.6, for a ranking supermartingale M with $M_{i+1} \leq M_i - \varepsilon$, the tail of the termination time T is bounded by $\mathbb{P}[T > n] \leq \exp\left(-\frac{2(\varepsilon((n-1)-M_1))^2}{(n-1)(b-a)^2}\right)$. In our case, we have $\varepsilon = 1-2r$ and $b - a = 2$. Thus we find

$$\mathbb{P}[T > n] \leq \exp\left(-\frac{2(\varepsilon((n-1)-M_1))^2}{2(n-1)(b-a)^2}\right) = O\left(\exp\left(-\frac{((1-2r)(n-1))^2}{2(n-1)}\right)\right)$$
$$= O\left(\exp\left(-\frac{(1-2r)^2(n-1)}{2}\right)\right) = O\left(\left(\sqrt{\exp(-(1-2r)^2)}\right)^n\right)$$

Note that $\exp(-(1-2r)^2) = \exp(-1+4r-4r^2) > 4r(1-r)$ for $r \neq 0$ by the inequality $\exp(x) > 1 + x$ for $x \neq 0$. By contrast, our method yields $\mathbb{P}[T > n] \leq c^{n-1}(1-c) = O(c^n) = O((2\sqrt{r(1-r)}+\delta)^n)$ for any $\delta > 0$. Hence our method yields better asymptotic tail bounds than the martingale-based method. In contrast, our method can only yield arbitrarily close bounds (Theorem 6.5.10).

Similarly, for the geometric example (Example 6.4.2), our method yields the tail bound $O((\frac{1}{2}+\delta)^n)$. The martingale is given by the variable $M_i := \mathrm{flip}(\frac{1}{2})$ implicit in

the loop condition. It satisfies

$$\mathbb{E}[M_{i+1} \mid M_i] = \begin{cases} \frac{1}{2} & \text{if } M_i = 1 \\ 0 & \text{if } M_i = 0 \end{cases} \leq M_i - \varepsilon \cdot [M_i > 0]$$

for $\varepsilon = \frac{1}{2}$. Since the martingale is bounded between 0 and 1, we have $b - a = 1$ and find:

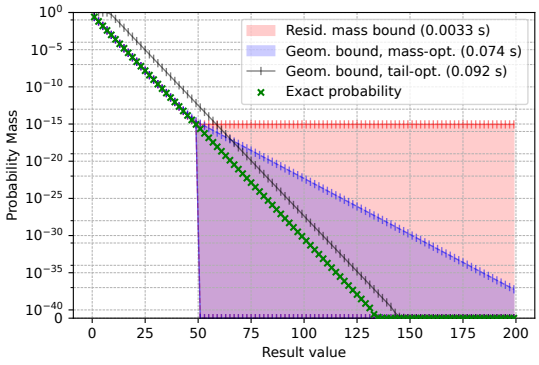
$$\mathbb{P}[T > n] \leq \exp\left(-\frac{2(\varepsilon((n-1) - M_1))^2}{(n-1)(b-a)^2}\right) = O\left(\exp\left(-\frac{2(\frac{1}{2}(n-1))^2}{(n-1)}\right)\right) = O\left(\left(\frac{1}{\sqrt{e}}\right)^n\right)$$

Thus our geometric bound semantics finds better asymptotic tail bounds in both cases. It would be interesting to investigate whether this is always the case, assuming EGD bounds exist.

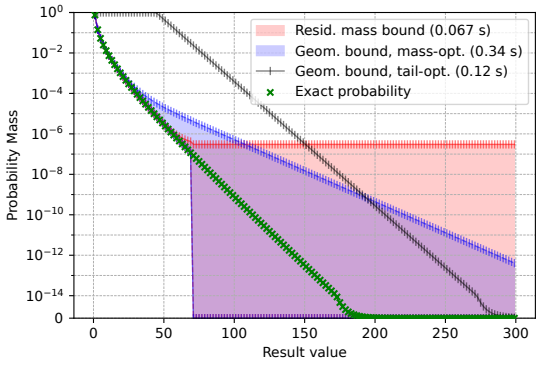
6.7.4 Comparison with Residual Mass Bounds

We compare the *quality* of the residual mass bounds and the EGD bounds on 5 examples: the simple counter (Example 6.4.2), the asymmetric random walk (Example 5.1.1), the introductory example (Example 2.2.1), the coupon collector problem with 5 coupons (ours/coupon-collector5), and Herman's self-stabilization protocol with 3 processes (adapted from psi/herman3). They were run with a loop unrolling limit of 50, 70, 40, 80, and 30 respectively, except for the geometric tail bounds where the limit was set to 1. We report bounds on the first two (raw) moments and the tails (Table 6.7). Note that the residual mass semantics (not just the lower bound semantics) is needed for lower bounds on moments because they require upper bounds on the normalizing constant. The optimization objective for the EGD bounds was the expected value for the moment bounds and the decay rate for the tail bound. The residual mass semantics is faster and finds good lower bounds on the moments, but cannot find upper bounds or tail bounds. The geometric bound semantics can find upper bounds on both, but takes much more time than the residual mass semantics.

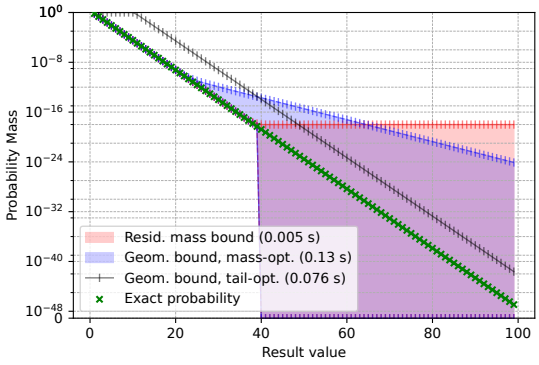
The bounds on the probability masses can be found in Fig. 6.3. We report EGD bounds for two optimization objectives: the total probability mass ("mass-optimized") and the decay rate ("tail-optimized"). The residual mass semantics is faster and yields tighter bounds for the probability masses on small values, but the bound is flat, i.e. the difference between upper and lower bounds is constant. In contrast, the geometric bound semantics finds upper bounds that keep decreasing like a geometric distribution even for large numbers beyond the unrolling limit. On the other hand, its upper bound is somewhat worse for small values than the residual mass semantics. This is because the geometric bounds arise from the contraction invariant, which requires a uniform decrease of the distribution. So the geometric bounds need to lose precision either for large values (when minimizing the total mass bound), or small values (when minimizing the tail bounds). In fact, the tail-optimized bound is much worse for small values, but eventually becomes almost parallel to the exact solution, since their asymptotics are almost equal up to a constant factor.



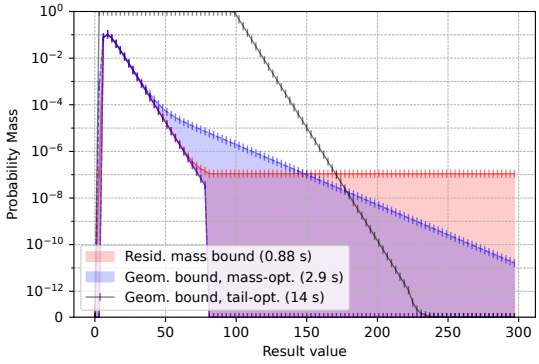
(a) Geom. counter (Example 6.4.2)



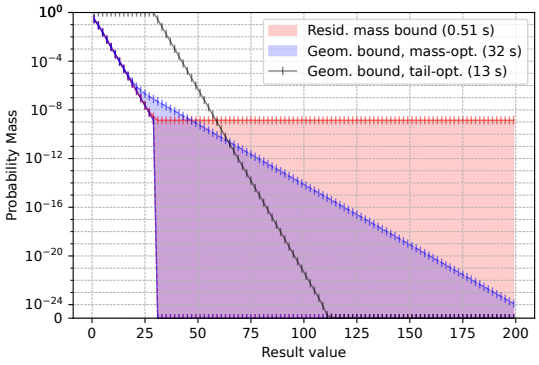
(b) Asym. rand. walk (Example 5.1.1)



(c) Die paradox (Example 2.2.1)



(d) Coupon collector problem with 5 coupons



(e) Herman's self-stabilization with 3 processes

Figure 6.3: Comparison of the residual mass and geometric bound semantics. Note that the probability masses (y-axis) are on a logarithmic scale, except for the lowest part, which is linear so as to include 0.

Table 6.7: Comparison of the quality of moment and tail bounds for various examples between the residual mass and geometric bound semantics.

Example	Method	Expected value	2nd moment	Tail	Time
Simple counter (Example 6.4.2)	exact	1	3	$\Theta(0.5^n)$	
	res. mass sem.	$\geq 1 - 5 \cdot 10^{-14}$	$\geq 3 - 3 \cdot 10^{-12}$	n/a	0.003 s
	geo. bound sem.	≤ 1.000037	≤ 3.00017	$O(0.5037^n)$	0.092 s
Random walk (Example 5.1.1)	exact	2	10	$\Theta(n^{-3/2}0.8660...^n)$	
	res. mass sem.	≥ 1.99997	≥ 9.998	n/a	0.067 s
	geo. bound sem.	≤ 2.55	≤ 21.8	$O(0.8682^n)$	1.47 s
Introductory "Die paradox" (Example 2.2.1)	exact	1.5	3	$\Theta(0.3333...^n)$	
	res. mass sem.	$\geq 1.5 - 2 \cdot 10^{-16}$	$\geq 3 - 4 \cdot 10^{-16}$	n/a	0.005 s
	geo. bound sem.	$\leq 1.5 + 8 \cdot 10^{-7}$	$\leq 3 + 3 \cdot 10^{-6}$	$O(0.3395^n)$	0.097 s
Coupon collector (5 coupons)	exact	11.41666...	155.513888...	$\Theta(0.8^n)$	
	res. mass sem.	≥ 11.41665	≥ 155.5131	n/a	0.88 s
	geo. bound sem.	≤ 11.56	≤ 158.5	$O(0.8002^n)$	22.6 s
Herman's self-stabilization (3 processes)	exact	1.333...	4.222...	?	
	res. mass sem.	≥ 1.3333332	≥ 4.222220	n/a	0.509 s
	geo. bound sem.	≤ 1.3339	≤ 4.226	$O(0.5002^n)$	34.8 s

6.8 Related Work

A summary of the most relevant work is given in Table 6.1: it includes work on guaranteed bounds [Beutner et al. 2022a; P. Wang et al. 2024], exact Bayesian inference with loops [Klinkenberg et al. 2024], and exact loop analysis with moments [Moosbrugger et al. 2022]. An extensive overview of formal analysis techniques for probabilistic programs was already given in Section 2.6. In this section, we highlight their trade-offs compared to our work.

Guaranteed Bounds The most closely related approaches to this chapter are methods for computing guaranteed bounds for Bayesian inference, such as GuBPI (Chapter 3), work by P. Wang et al. [2024], and the residual mass bounds (Chapter 5). A comparison of these three methods was given in Section 5.5. In Section 6.7.4, we compare geometric bounds with residual mass bounds, the strongest competitor in our setting. It is worth reiterating that previous approaches are unable to bound the moments or tail probability asymptotics. Our geometric bound semantics bounds the *whole distribution* and therefore does not suffer from these restrictions. On the other hand, our programming language is more restricted, e.g. it does not support continuous distributions, and geometric bounds do not always exist.

Exact Bayesian inference The exact inference system Prodigy [Klinkenberg et al. 2024] can perform exact Bayesian inference on a *single probabilistic loop without nesting* parameterized in the program inputs, assuming the user provides a loop invariant

template, which is often difficult to find (see Section 2.4). In contrast, our method is fully automatic. As a trade-off, it is not exact and only works on fixed program inputs.

Exact Loop Analysis with Moments There has been some work on studying probabilistic loop behavior by analyzing the moments of the random variables, in particular, invariant synthesis [Bartocci et al. 2019; Amrollahi et al. 2022; Moosbrugger et al. 2022; Kofnov et al. 2024]. Our approach differs in several regards. On the one hand, these approaches yield exact moments and can handle continuous distributions and more primitive operations (e.g. multiplication). On the other hand, they do not support branching on variables with infinite support, nested loops, or conditioning. Our method can handle these constructs, and bounds the whole distribution, not just the moments. As a consequence, we get bounds on probability masses and exponential bounds on the tails, which are better than the polynomial bounds that can be derived from the moments in previous works.

Cost Analysis & Martingales Martingales are a common technique for cost analysis of probabilistic programs, i.e. computing bounds on the total cost of a program [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2016; Kura et al. 2019; D. Wang et al. 2021; P. Wang et al. 2019; Chatterjee et al. 2024]. Our method makes different trade-offs. Martingale-based methods can handle continuous distributions and more primitive program operations than discrete SGCL, such as multiplication. Their bounds are also typically parameterized in the program inputs, whereas our bounds only apply to fixed inputs. Conversely, our method yields bounds not just on one program variable (e.g. the running time), but on the whole distribution of all program variables; we support conditioning; and our bounds can be made arbitrarily tight (Theorem 6.5.10).

6.9 Conclusion

In this chapter, we advance the concept of guaranteed bounds for discrete probabilistic programs. Our geometric bound semantics adds two novel concepts (contraction invariants and EGDs) to the toolbox of automated probabilistic program analysis, which currently relies predominantly on martingales, moment analysis, and generating functions. Since this semantics overapproximates the *whole distribution* (as opposed to individual probabilities or moments), it can bound probability masses, moments, and tails at the same time. It has desirable theoretical properties, such as soundness and convergence, and our empirical studies demonstrate their applicability and effectiveness on a variety of benchmarks.

Future work The geometric bound semantics can likely be extended to handle (demonic) nondeterminism, as this corresponds to finding a bound ν on the maximum of two EGD bounds μ_1, μ_2, which can be specified using inequalities $\mu_1 \preceq \nu$ and $\mu_2 \preceq \nu$. One could also try to generalize the shape of upper bounds from EGDs to a more expressive subclass of discrete phase-type distributions. Furthermore, it would be interesting to illuminate the theoretical connections with martingale analysis and the quality of the derivable bounds, which was investigated empirically in Section 6.7.3.

It should also be possible to modify the geometric bound semantics to handle variables in $\mathbb{Z}$ instead of $\mathbb{N}$. For a single variable, we need two EGDs: one for the negative part and one for the nonnegative part. In the case of n variables, we need one EGD$(\mathbf{P}^{\boldsymbol{o}}, \boldsymbol{\alpha}^{\boldsymbol{o}})$ for each orthant $\boldsymbol{o} \in \{+, -\}^n$. We do not foresee any serious conceptual difficulties extending the geometric bound semantics to operate on such a family of orthant EGDs, but the notation and proofs become quite cumbersome because $\mathbf{P}$ takes *two* multi-indices, not just one. In the interest of clarity and conceptual simplicity, we decided not to pursue this idea further.

Variables with values in $\mathbb{R}$ that can be sampled from continuous distributions are much harder to support. Analogously to EGDs extending geometric distributions, it may be possible to handle continuous distributions by extending the exponential distribution to "eventually exponential distributions", which would consist of an initial block using, e.g. a polynomial bound on the probability density function on a compact interval $[-a, a]$ of $\mathbb{R}$, and exponential distributions on $(-\infty, -a]$ and $[a, \infty)$. Whether and how this could work in detail is unclear and requires further investigation.

7 Conclusion

Summary Over the course of this thesis, I have introduced several methods for computing guaranteed bounds on the posterior distribution of probabilistic programs. First, I discussed the most widely applicable approach based on abstract interpretation on the trace space with interval arithmetic for SPCF programs (Chapter 3). Next, I presented a more efficient technique to obtain guaranteed bounds for SGCL (Chapter 5), extending an exact inference method for loop-free programs leveraging generating functions (Chapter 4). Finally, I described an approach to compute more informative bounds for discrete SGCL programs (Chapter 6) based on the family of eventually geometric distributions as an abstract domain for distributions. There are a number of trends to observe here, some of which were summarized in Table 1.1 (page 15).

First, the class of supported probabilistic programs goes from very expressive to more restricted: SPCF in Chapter 3, which is a universal PPL with higher-order functions, continuous distributions, soft conditioning and recursion; then SGCL in Chapter 5, which is restricted to a fixed number of variables, linear operations, and discrete observations; and finally discrete SGCL in Chapter 6, which disallows continuous distributions completely.

At the same time, the efficiency of the techniques, as well as the quality of the bounds, increases. For instance, the residual mass semantics (Chapter 5) obtains better bounds much faster than GuBPI (Chapter 3) and does not suffer from the path explosion problem. Furthermore, the geometric bounds (Chapter 6) are strong enough to bound moments and the tail asymptotics, not just posterior probability masses like the residual mass bounds (Chapter 5).

Finally, the theoretical guarantees become stronger in the later chapters as well. For instance, the convergence result in Chapter 3 is a mere existence result ("assuming good choices are made") whereas in Chapters 5 and 6, the bounds tighten automatically with increased computational effort.

Evaluation of the contributions In Section 1.5, I set out to solve the problem of obtaining "*hard guarantees* on the posterior distribution of a probabilistic program, without the restrictions of exact inference". This goal has largely been achieved in terms of automation and expressivity: both SPCF and SGCL can express complex models with unbounded recursion or loops, which are beyond the reach of exact inference

algorithms. On the other hand, there are notable limitations, as detailed in Section 1.8. In particular, efficiency remains a major challenge: the computational cost of the bounding techniques grows exponentially with the dimensionality of the inference problem. Monte Carlo methods are simply hard to beat in terms of scalability: their sampling-based approach alleviates the curse of dimensionality. Nevertheless, guaranteed bounds are worth the extra effort when debugging an inference algorithm or to detect issues with a model, possibly in a scaled-down form.

Role in Bayesian statistics While I focused on the verification of the posterior distribution in this thesis, probabilistic models themselves merely approximate reality and model misspecifications are a major concern in practice. For these reasons, a *Bayesian workflow* [Gelman et al. 2020] incorporates three components: iterative model development, inference, and model checking. It is important to remember that Bayesian inference is but one source of error in Bayesian statistical modeling. The aim of the model checking step is to validate the model with the inferred parameters, for instance via posterior predictive checks. This step also subsumes statistical inference checks, such as convergence diagnostics and simulation-based calibration (see Section 2.5.6). Guaranteed bounds can serve as a valuable addition to this aspect of the Bayesian workflow, by pinpointing problems with approximate inference, especially in the early stages of model development where the models are smaller and simpler.

Role in formal verification In this thesis, I introduced the new concept of guaranteed bounds on the program distribution, which can be viewed as partial correctness properties of probabilistic programs, to the field of quantitative program verification. Moreover, I extended the toolbox of automated reasoning for probabilistic programs with techniques such as abstract interpretation on the trace space (Chapter 3), automatic differentiation of generating functions (Chapter 4), residual mass bounds (Chapter 5), contraction invariants, and eventually geometric distributions (Chapter 6). These ideas also have the potential to be applied to other problems in quantitative program analysis.

7.1 Future Directions

In the following, I outline potential future directions for the verification of Bayesian inference in probabilistic programming, including but not limited to guaranteed bounds.

Verification of inference Advanced inference algorithms such as NUTS (see Section 2.5.3) are notoriously complex, making their implementations prone to subtle bugs that can be hard to detect. Therefore, I find the idea of mechanizing their correctness proofs in a theorem prover appealing, as it would enable the extraction of fully verified implementations. Tassarotti and Tristan [2023] take a step in this direction by verifying Stan's compiler in Coq, but stop short of verifying its NUTS implementation.

A more lightweight alternative is the idea of *correctness by construction* in the context of inference programming. In this setting, the user writes their own inference algorithm as a program using sound building blocks and the PPS ensures correctness,

e.g. with a type system [Lew et al. 2020]. Current techniques can statically verify several necessary soundness properties, but cannot prove full correctness and only apply to easily decomposable inference algorithms. It would be interesting to investigate how much further one can push this idea.

Guaranteed bounds Guaranteed bounds on the inference results go beyond the mathematical correctness of the inference algorithm in that they can detect convergence issues and other practical problems. While I already discussed future directions specific to each bounding method at the end of their respective chapters, here I want to highlight the overarching themes of future work on guaranteed bounds with the greatest potential for impact.

As in exact inference, the efficiency of the bounding techniques is a major concern. One can imagine a number of ways to improve performance in practice: factorizing the joint probability distribution, or more generally exploiting the typically rich dependency structure in probabilistic programs. The use of probabilistic circuits as in the exact inference systems Dice [Holtzen et al. 2020] and SPPL [Saad et al. 2021] could be another promising direction.

It would also be desirable to improve the precision of the bounds. One way of achieving this could be more sophisticated abstract domains. Eventually geometric distributions were used in Chapter 6 and (truncated) normal distributions in Smith [2008]. A promising candidate for high-dimensional continuous distributions could be mixtures of Gaussian distributions, for two reasons: mixtures can express multimodal distributions and the family of Gaussian distributions is closed under many operations such as conditioning, marginalization, and linear operations.

The bounding techniques presented in this thesis work on a complete program and its posterior distribution. An interesting future direction is to modularize the approaches, so that they can be applied to subprograms and possibly be integrated with other inference algorithms. For instance, one could imagine the guaranteed bounds informing the choice of the proposal distributions of Monte Carlo algorithms.

All these directions have the potential to strengthen the impact of guaranteed bounds on the formal verification of Bayesian inference results in probabilistic programming.

Appendices

A

Supplementary Material for Chapter 3

Contents

A.1 Supplementary Material for Section 3.3 194
A.1.1 Primitive Functions 194
A.2 Supplementary Material for Section 3.4 196
A.2.1 Soundness 196
A.2.2 Weak Completeness 200
A.2.3 Constraint-based Type Inference 201

A.1 Supplementary Material for Section 3.3

A.1.1 Primitive Functions

It is not immediately obvious that a set of primitive functions satisfying Assumption 3.3.12 exists. We first consider an example of such a set that *almost* satisfies this assumption. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called a ***submersion*** if it is continuously differentiable and its gradient is nonzero everywhere.

Lemma A.1.1. *The set $\mathcal{F}_{\mathsf{subm}}$ of submersions is closed under composition and each of its functions is boxwise continuous and interval separable.*

Proof. Boxwise continuity is obvious given that the functions are even continuously differentiable. For interval separability, we use Lemma 3.3.11. Let $f : \mathbb{R}^n \to \mathbb{R} \in \mathcal{F}_{\mathsf{subm}}$. Since f is a submersion, the preimage $f^{-1}(x)$ of any point $x \in \mathbb{R}$ is an $(n-1)$-dimensional submanifold of $\mathbb{R}^n$ by the preimage theorem (a variation of the implicit function theorem). Submanifolds of codimension > 1 have measure zero. (This well-known fact can be shown by writing the submanifold as a countable union of graphs and applying Fubini's theorem to each of them.) Therefore, the lemma applies.

For closure under composition, let $f : \mathbb{R}^m \to \mathbb{R}$ and $f_i : \mathbb{R}^{n_i} \to \mathbb{R}$ for $i \in \{1, \dots, m\}$, all in $\mathcal{F}_{\mathsf{subm}}$. The composition $g := f \circ (f_1 \times \cdots \times f_m)$ is clearly C^1 again, so we just have

to check the submersion property. By the chain rule, we find that the gradient of the composition

$$\nabla g(x_1, \dots, x_m) = \begin{pmatrix} \partial_1 f(f_1(x_1), \dots, f_m(x_m)) \cdot \nabla f_1(x_1) \\ \vdots \\ \partial_m f(f_1(x_1), \dots, f_m(x_m)) \cdot \nabla f_m(x_m) \end{pmatrix}$$

is nonzero because at least one of the $\partial_i f$ is nonzero and $\nabla f_i(x_i) \neq 0$ by assumption. Hence the composition is a submersion again. □

Unfortunately, the set of submersions does not contain constant functions. This is a problem because then it is not guaranteed that partially applying a primitive function to a constant is still an admissible primitive function. (For example, this would break Lemma 3.3.17.) Hence we need to assume that *all constant functions be primitive functions.* Luckily, the set $\mathcal{F}_{\text{subm}}$ of submersions can be easily extended to accommodate this.

Lemma A.1.2. *Let $\mathcal{F}^*_{\text{subm}}$ be the set of functions $f : \mathbb{R}^n \to \mathbb{R}$ (for all $n \in \mathbb{N}$) such that whenever the partial derivative $\partial_i f(x)$ is zero for some $i \in \{1, \dots, n\}$ and $x \in \mathbb{R}^n$ then f is constant in its i-th argument, i.e. there is a function $f^* : \mathbb{R}^{n-1} \to \mathbb{R}$ such that $f(x_1, \dots, x_n) = f^*(x_1, \dots, x_{i-1}, x_{i+1}, \dots x_n)$. This set satisfies all the assumptions about sets of primitive functions: it is closed under composition, contains all constant functions, and all its functions are boxwise continuous and interval separable.*

Proof. Boxwise continuity is obvious given that the functions are even continuously differentiable. Similarly, it is clear that $\mathcal{F}^*_{\text{subm}}$ contains all constant functions.

For interval separability, let $f : \mathbb{R}^n \to \mathbb{R} \in \mathcal{F}^*_{\text{subm}}$ and $J \subseteq \{1, \dots, n\}$ be the set of indices in which f is not constant, and J' its complement. Hence there is a submersion $f_J : \mathbb{R}^{|J|} \to \mathbb{R}$ such that $f(x) = f_J(x_J)$ where x_J stands for the vector of coordinates of x with index in J. The preimage of $f^{-1}(U) \subseteq \mathbb{R}^{|J|}$ of any set $U \subseteq \mathbb{R}$ can be tightly approximated by boxes if and only if $f_J^{-1}(U)$ can because $f^{-1}(U)$ is a Cartesian product of $f_J^{-1}(U)$ and $\mathbb{R}^{|J'|}$. Since f_J is interval separable by the previous lemma, this shows that f is as well.

For closure under composition, let $f : \mathbb{R}^m \to \mathbb{R}$ and $f_i : \mathbb{R}^{n_i} \to \mathbb{R}$ for $i \in \{1, \dots, m\}$, all in $\mathcal{F}^*_{\text{subm}}$. The composition $g := f \circ (f_1 \times \cdots \times f_m)$ is clearly C^1 again, so we just have to check the property of the partial derivatives. By the chain rule, the partial derivatives of the composition are

$$\partial_i g(x_1, \dots, x_m) = \partial_j f(f_1(x_1), \dots, f_m(x_m)) \partial_k f_j(x_{j1}, \dots, x_{jn_j})$$

for some $j \in \{1, \dots m\}$ and $k \in \{1, \dots n_j\}$, and for all $x_1 \in \mathbb{R}^{n_1}, \dots, x_m \in \mathbb{R}^{n_m}$. So if this partial derivative is zero, there are two cases. First, if $\partial_j f(f_1(x_1), \dots, f_m(x_m)) = 0$ then f must be constant in its j-th argument (because $f \in \mathcal{F}^*_{\text{subm}}$) and thus g is constant in x_j, and in particular the i-th argument (which is an entry of x_j). Second, if $\partial_k f_j(x_{j1}, \dots, x_{jn_j}) = 0$ then f_j must be constant in its k-th argument (because $f_j \in \mathcal{F}^*_{\text{subm}}$) and thus g is constant in its corresponding i-th argument as well. This proves $g \in \mathcal{F}^*_{\text{subm}}$, as desired. □

Note that exp, sinh, arctan, n-th roots for n odd, and all linear functions are in $\mathcal{F}^*_{\text{subm}}$. So this is a useful set of primitive functions already. Unfortunately, it does not

include multiplication because the gradient of $(x, y) \mapsto xy$ is zero at $(0, 0)$. To fix this issue, we need to restrict the domain.[1]

For simplicity, we required primitive functions to be defined on all of $\mathbb{R}^n$. Suppose we allow for primitive functions to be defined only on an open subset of $\mathbb{R}^n$, and applying them to a value outside their domain is disallowed in SPCF programs. Then we can also include multiplication (on $\mathbb{R}^2 \setminus \{(0, 0)\}$), logarithms (on $(0, \infty)$), non-constant univariate polynomials (on the complement of their stationary points), quantile functions of continuous distributions with nonzero density (on $(0, 1)$), and probability density functions (on the complement of their stationary points). The fact that some points in the domain are missing is inconvenient for functions that can be continuously extended to the these points, but one can work around this in a program by checking for the points that are not in the domain and returning the function values as constants for those cases.

A.2 Supplementary Material for Section 3.4

We provide additional proofs and material for Section 3.4.

A.2.1 Soundness

To show soundness (Theorem 3.4.3), we establish a (weight-aware) subject reduction property for our type system as follows. For an interval $[a, b] \in \mathbb{I}$ and $r \in \mathbb{R}_{\geq 0}$, we define $r \cdot [a, b] := [r \cdot a, r \cdot b]$. To simplify notation, we use a modified transition relation that omits the concrete trace (which is irrelevant in Theorem 3.4.3). We write $P \to_w P'$ if $(P, \boldsymbol{s}, 1) \to (P', \boldsymbol{s}', w)$ for some $w \in \mathbb{R}$ and $\boldsymbol{s}, \boldsymbol{s}' \in \mathbb{T}$. Note that we could define $\to_w$ as a dedicated reduction system by adapting the rules from $\to$ in Fig. 2.2.

Lemma A.2.1 (Substitution). *If* $\Gamma; \{x_i : \sigma_i\}_{i=1}^n \vdash P : \mathcal{A}$ *for distinct variables* x_i *and if* $\Gamma \vdash M_i : \begin{Bmatrix} \sigma_i \\ \mathbf{1} \end{Bmatrix}$ *for all* $i \in \{1, \dots, n\}$ *then* $\Gamma \vdash P[M_i/x_i]_{i=1}^n : \mathcal{A}$.

Proof. By a standard induction on M. □

Lemma A.2.2 (Weighted Subject Reduction). *Let* P *be any program such that* $\vdash P : \begin{Bmatrix} \sigma \\ J \end{Bmatrix}$ *and* $P \to_w P'$ *for some* $w > 0$. *Then* $\vdash P' : \left\{ \begin{smallmatrix} \sigma \\ \frac{1}{w} \cdot J \end{smallmatrix} \right\}$.

Proof. We prove this by induction on the structure of P.

Case $P =$ sample**:** then $P \to_1 \underline{r}$ for some $r \in [0, 1]$. As multiple consecutive application of (Sub) can be replaced by a single one since (as $\sqsubseteq_{\mathcal{A}}$ is transitive), we can

[1] Handling these issues at the level of primitive functions directly (without restricting the domain) seems challenging: even if a function has only one point with zero gradient, e.g. multiplication, its preimage under other primitive functions can become very complicated. We tried to handle this by allowing the primitive functions to be submersions except on a null set given by a union of lower-dimensional manifolds. However, the preimages of such manifolds need not be manifolds again. Hence it seems difficult to come up with a broader class of primitive functions satisfying the assumptions without restricting the domain.

assume w.l.o.g. that the last step in $\vdash P : \left\{\begin{matrix}\sigma\\ J\end{matrix}\right\}$ was:

$$\dfrac{\dfrac{}{\vdash \mathsf{sample} : \left\{\begin{matrix}[0,1]\\ \mathbf{1}\end{matrix}\right\}}\ \textsc{Sample}}{\vdash \mathsf{sample} : \left\{\begin{matrix}I\\ J\end{matrix}\right\}}\ \textsc{Sub}$$

By subtyping we have $[0,1] \sqsubseteq I$ and $\mathbf{1} \sqsubseteq J$. So $[r,r] \sqsubseteq I$ and we can type $\vdash \underline{r} : \left\{\begin{matrix}I\\ J\end{matrix}\right\}$ using (Lit) and (Sub) as required.

Case $P = f(\underline{r_1}, \ldots, \underline{r_{|f|}})$**:** then $P \to_1 \underline{f(r_1, \ldots, r_{|f|})}$. W.l.o.g., we can assume that the last step in $\vdash P : \left\{\begin{matrix}\sigma\\ J\end{matrix}\right\}$ was

$$\dfrac{\dfrac{\dfrac{}{\vdash \underline{r_1} : \left\{\begin{matrix}[r_1,r_1]\\ \mathbf{1}\end{matrix}\right\}}\ \textsc{Lit}}{\vdash \underline{r_1} : \left\{\begin{matrix}I_1\\ J_1\end{matrix}\right\}}\ \textsc{Sub} \quad \ldots \quad \dfrac{\dfrac{}{\vdash \underline{r_{|f|}} : \left\{\begin{matrix}[r_{|f|},r_{|f|}]\\ \mathbf{1}\end{matrix}\right\}}\ \textsc{Lit}}{\vdash \underline{r_{|f|}} : \left\{\begin{matrix}I_{|f|}\\ J_{|f|}\end{matrix}\right\}}\ \textsc{Sub}}{\vdash f(\underline{r_1}, \ldots, \underline{r_{|f|}}) : \left\{\begin{matrix}f^{\mathbb{I}}(I_1, \ldots, I_{|f|})\\ (\times^{\mathbb{I}})_{i=1}^{|f|} J_i\end{matrix}\right\}}\ \textsc{Prim}}{\vdash f(\underline{r_1}, \ldots, \underline{r_{|f|}}) : \left\{\begin{matrix}I\\ J\end{matrix}\right\}}\ \textsc{Sub}$$

So by subtyping $r_i \in I_i$ and $1 \in J_i$ for all i. By definition of $f^{\mathbb{I}}$ we thus have $f(r_1, \ldots, r_{|f|}) \in f^{\mathbb{I}}(I_1, \ldots, I_{|f|})$ and (again by subtyping) we have $f(r_1, \ldots, r_{|f|}) \in I$. Similarly, by definition of $\times^{\mathbb{I}}$ we have $1 \in (\times^{\mathbb{I}})_{i=1}^{|f|} J_i$ and thus $1 \in J$. We can type $\vdash \underline{f(r_1, \ldots, r_{|f|})} : \left\{\begin{matrix}I\\ J\end{matrix}\right\}$ using (Lit) and (Sub) as required.

Case $P = \mathsf{if}\ \underline{r} \le 0\ \{M\}\ \mathsf{else}\ \{N\}$ **and** $r \le 0$**:** then $P \to_1 M$. W.l.o.g., we can assume that the last step in $\vdash P : \left\{\begin{matrix}\sigma\\ J\end{matrix}\right\}$ is

$$\dfrac{\dfrac{\dfrac{}{\vdash \underline{r} : \left\{\begin{matrix}[r,r]\\ \mathbf{1}\end{matrix}\right\}}\ \textsc{Lit}}{\vdash \underline{r} : \left\{\begin{matrix}I\\ J''\end{matrix}\right\}}\ \textsc{Sub} \quad \vdash M : \left\{\begin{matrix}\sigma'\\ J'\end{matrix}\right\} \quad \vdash N : \left\{\begin{matrix}\sigma'\\ J'\end{matrix}\right\}}{\vdash \mathsf{if}\ \underline{r} \le 0\ \{M\}\ \mathsf{else}\ \{N\} : \left\{\begin{matrix}\sigma'\\ J' \times^{\mathbb{I}} J''\end{matrix}\right\}}\ \textsc{If}}{\vdash \mathsf{if}\ \underline{r} \le 0\ \{M\}\ \mathsf{else}\ \{N\} : \left\{\begin{matrix}\sigma\\ J\end{matrix}\right\}}\ \textsc{Sub}$$

and we have $1 \in J''$, $\sigma' \sqsubseteq_\sigma \sigma$ and $J' \times^{\mathbb{I}} J'' \sqsubseteq J$ by subtyping. As $1 \in J''$, we get $J' \sqsubseteq J' \times^{\mathbb{I}} J''$. Thus we obtain $J' \sqsubseteq J$ and $\sigma' \sqsubseteq_\sigma \sigma$, and we can type $\vdash M : \left\{\begin{matrix}\sigma\\ J\end{matrix}\right\}$ using (Sub).

Case $P = \mathsf{if}\ \underline{r} \le 0\ \{M\}\ \mathsf{else}\ \{N\}$ **and** $r > 0$**:** then $P \to_1 N$. Analogous to the previous case.

Case $P = \mathsf{score}(\underline{r})$ **and** $r \geq 0$**:** then $P \to_r \underline{r}$. W.l.o.g., we can assume that the last step in $\vdash P : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}$ is

$$\dfrac{\dfrac{\dfrac{}{\vdash \underline{r} : \left\{\begin{smallmatrix}[r,r]\\ \mathbf{1}\end{smallmatrix}\right\}}\textsc{Lit}}{\vdash \underline{r} : \left\{\begin{smallmatrix}I'\\ J'\end{smallmatrix}\right\}}\textsc{Sub}}{\vdash \mathsf{score}(\underline{r}) : \left\{\begin{smallmatrix}I' \sqcap [0,\infty]\\ J' \times^{\mathbb{I}} (I' \sqcap [0,\infty])\end{smallmatrix}\right\}}\textsc{Score}}{\vdash \mathsf{score}(\underline{r}) : \left\{\begin{smallmatrix}I\\ J\end{smallmatrix}\right\}}\textsc{Sub}$$

By subtyping we have $r \in I'$ and even $r \in I' \sqcap [0,\infty]$ because $r \geq 0$. Thus $r \in I$, again by subtyping. Similarly, $1 \in J'$ and by definition of $\times^{\mathbb{I}}$, we have $r \in J' \times^{\mathbb{I}} (I' \sqcap [0,\infty])$. Thus $r \in J$ by subtyping. This already implies $1 \in \frac{1}{r} \cdot J$ and we can thus type $\vdash \underline{r} : \left\{\begin{smallmatrix}I\\ \frac{1}{r} \cdot J\end{smallmatrix}\right\}$ by using (Lit) and (Sub).

Case $P = (\lambda x.M)V$**:** then $P \to_1 M[V/x]$. W.l.o.g., the last step in $\vdash P : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x : \tilde{\sigma}'\} \vdash M : \left\{\begin{smallmatrix}\tilde{\sigma}''\\ \tilde{J}''\end{smallmatrix}\right\}}{\vdash \lambda x.M : \left\{\begin{smallmatrix}\tilde{\sigma}' \to \left\{\begin{smallmatrix}\tilde{\sigma}''\\ \tilde{J}''\end{smallmatrix}\right\}\\ \mathbf{1}\end{smallmatrix}\right\}}\textsc{Abs}}{\vdash \lambda x.M : \left\{\begin{smallmatrix}\sigma' \to \left\{\begin{smallmatrix}\sigma''\\ J''\end{smallmatrix}\right\}\\ J'\end{smallmatrix}\right\}}\textsc{Sub} \qquad \vdash V : \left\{\begin{smallmatrix}\sigma'\\ J'''\end{smallmatrix}\right\}}{\vdash (\lambda x.M)V : \left\{\begin{smallmatrix}\sigma''\\ J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J'''\end{smallmatrix}\right\}}\textsc{App}}{\vdash (\lambda x.M)V : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}}\textsc{Sub}$$

By subtyping we get that $\sigma' \sqsubseteq_\sigma \tilde{\sigma}'$. It is easy to see that we can also type $\vdash V : \left\{\begin{smallmatrix}\sigma'\\ \mathbf{1}\end{smallmatrix}\right\}$ because V is a value and we get $\vdash V : \left\{\begin{smallmatrix}\tilde{\sigma}'\\ \mathbf{1}\end{smallmatrix}\right\}$ by (Sub). Using Lemma A.2.1, we can thus type $\vdash M[V/x] : \left\{\begin{smallmatrix}\tilde{\sigma}''\\ \tilde{J}''\end{smallmatrix}\right\}$.

We have $1 \in J'$ by subtyping and as V is a value, it is easy to see that $1 \in J'''$. Hence $J'' \sqsubseteq J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J'''$. Also by subtyping, we find $\tilde{\sigma}'' \sqsubseteq_\sigma \sigma'' \sqsubseteq_\sigma \sigma$, $\tilde{J}'' \sqsubseteq J''$, and $J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J''' \sqsubseteq J$. This implies $\tilde{\sigma}'' \sqsubseteq_\sigma \sigma$ and $\tilde{J}'' \sqsubseteq J$. By subtyping $\vdash M[V/x] : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}$, as required.

Case $P = (\mathsf{fix}^{\varphi}_x.\, M)V$**:** then $P \to_1 M[V/x, (\mathsf{fix}^{\varphi}_x.\, M)/\varphi]$. Analogous to the previous case for abstractions.

Case $P = E[P']$ **for an evaluation context** $E \neq [\cdot]$**:** then $P' \to_r P''$ and $P \to_r E[P'']$. All such cases follow easily by case analysis on E. As an example, consider the context

$E = [\cdot]N$. In this situation, we have $P = E[P'] = P'N$ with $P' \to_r P''$, so $P \to_r P''N$. W.l.o.g., the last step in $\vdash P : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}$ is

$$\dfrac{\dfrac{\vdash P' : \left\{\begin{smallmatrix}\sigma' \to \left\{\begin{smallmatrix}\sigma''\\ J''\end{smallmatrix}\right\}\\ J'\end{smallmatrix}\right\} \qquad \vdash N : \left\{\begin{smallmatrix}\sigma'\\ J'''\end{smallmatrix}\right\}}{\vdash P'N : \left\{\begin{smallmatrix}\sigma''\\ J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J'''\end{smallmatrix}\right\}}\,\textsc{App}}{\vdash P'N : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}}\,\textsc{Sub}$$

By the inductive assumption for $P' \to_r P''$, we get $\vdash P'' : \left\{\begin{smallmatrix}\sigma' \to \left\{\begin{smallmatrix}\sigma''\\ J''\end{smallmatrix}\right\}\\ \frac{1}{r} \cdot J'\end{smallmatrix}\right\}$ and can then type

$$\dfrac{\dfrac{\vdash P'' : \left\{\begin{smallmatrix}\sigma' \to \left\{\begin{smallmatrix}\sigma''\\ J''\end{smallmatrix}\right\}\\ \frac{1}{r} \cdot J'\end{smallmatrix}\right\} \qquad \vdash N : \left\{\begin{smallmatrix}\sigma'\\ J'''\end{smallmatrix}\right\}}{\vdash P'N : \left\{\begin{smallmatrix}\sigma''\\ \frac{1}{r} \cdot J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J'''\end{smallmatrix}\right\}}\,\textsc{App}}{\vdash P'N : \left\{\begin{smallmatrix}\sigma\\ \frac{1}{r} \cdot J\end{smallmatrix}\right\}}\,\textsc{Sub}$$

because if $J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J''' \sqsubseteq J$ then $\frac{1}{r} \cdot J' \times^{\mathbb{I}} J'' \times^{\mathbb{I}} J''' \sqsubseteq \frac{1}{r} \cdot J$.

The proof for the other evaluation contexts, i.e., where $P = \text{if } P' \leq 0 \{M\} \text{ else } \{N\}$, $P = VP'$, $P = \text{score}(P')$, or $P = \underline{f}(r_1, \ldots, r_{i-1}, P', N_{i+1}, \ldots, N_{|f|})$ for some $P' \to_r P''$, are all analogous to the above. □

Lemma A.2.3 (Zero-Weighted Subject Reduction)**.** *Let P be any program such that $\vdash P : \left\{\begin{smallmatrix}\sigma\\ J\end{smallmatrix}\right\}$ and $P \to_0 P'$. Then*

1. *$\vdash P' : \left\{\begin{smallmatrix}\sigma\\ J'\end{smallmatrix}\right\}$ for some J', and*
2. *$0 \in J$*

Proof. The proof that $\vdash P' : \left\{\begin{smallmatrix}\sigma\\ J'\end{smallmatrix}\right\}$ for some J' is analogous to the proof of Lemma A.2.2 with fewer restrictions on the weight. The claim $0 \in J$ follows by observing that $P \to_0 P'$ is only possible if the redex in P is $\text{score}(\underline{0})$. In case $P = \text{score}(\underline{0})$, the claim follows directly from (Score). If $P = E[\text{score}(\underline{0})]$, it is a simple induction on the structure of the evaluation context E. □

Theorem 3.4.3. *Let $\vdash P : \mathbf{R}$ be a simply-typed program. If $\vdash P : \left\{\begin{smallmatrix}[a,b]\\ [c,d]\end{smallmatrix}\right\}$ and $(P, \boldsymbol{s}, 1) \to^* (\underline{r}, \langle\rangle, w)$ for some $\boldsymbol{s} \in \mathbb{T}$ and $r, w \in \mathbb{R}$, then $r \in [a,b]$ and $w \in [c,d]$.*

Proof. Let

$$(P, \boldsymbol{s}, 1) = (P_0, \boldsymbol{s}_0, w_0) \to \cdots \to (P_n, \boldsymbol{s}_n, w_n) = (\underline{r}, \langle\rangle, w)$$

be the reduction sequence of $(P, \boldsymbol{s}, 1)$. By definition of $\to_w$ it is easy to see that we get

$$P = P_0 \to_{\tilde{w}_1} P_1 \to_{\tilde{w}_2} \cdots \to_{\tilde{w}_n} P_n = \underline{r}$$

for unique $\tilde{w}_1, \ldots, \tilde{w}_n$. Note that $w_i = \prod_{j=1}^{i} \tilde{w}_j$.

We first assume that $w \neq 0$ (so $\tilde{w}_i \neq 0$ for all i). We claim that for each $0 \leq i \leq n$, we can type

$$\vdash P_i : \left\{ \begin{matrix} [a,b] \\ \frac{1}{w_i} \cdot [c,d] \end{matrix} \right\} \tag{A.1}$$

Equation (A.1) follows by simple induction: the base case $i = 0$ holds by the assumption $\vdash P_0 : \left\{ \begin{matrix} [a,b] \\ [c,d] \end{matrix} \right\}$ and because $w_0 = 1$. For the inductive case, we assume that $\vdash P_i : \left\{ \begin{matrix} [a,b] \\ \frac{1}{w_i} \cdot [c,d] \end{matrix} \right\}$. We apply Lemma A.2.2 to $P_i \to_{\tilde{w}_i} P_{i+1}$ and finish the induction step using $w_{i+1} = w_i \tilde{w}_{i+1}$:

$$\vdash P_{i+1} : \left\{ \begin{matrix} [a,b] \\ \frac{1}{\tilde{w}_{i+1}} \cdot \frac{1}{w_i} \cdot [c,d] \end{matrix} \right\} = \left\{ \begin{matrix} [a,b] \\ \frac{1}{w_{i+1}} \cdot [c,d] \end{matrix} \right\}$$

Equation (A.1) thus implies $\vdash \underline{r} : \left\{ \begin{matrix} [a,b] \\ \frac{1}{w} \cdot [c,d] \end{matrix} \right\}$. W.l.o.g., we can assume that this type judgment has the form

$$\dfrac{\dfrac{}{\vdash \underline{r} : \left\{ \begin{matrix} [r,r] \\ \mathbf{1} \end{matrix} \right\}}\ \textsc{Lit}}{\vdash \underline{r} : \left\{ \begin{matrix} [a,b] \\ \frac{1}{w} \cdot [c,d] \end{matrix} \right\}}\ \textsc{Sub}$$

This implies $r \in [a,b]$ and $w \in [c,d]$ (because $1 \in \frac{1}{w} \cdot [c,d]$), as required.

Now consider the case $w = 0$, so at least one $\tilde{w}_i = 0$. The claim that $r \in [a,b]$ follows easily as above (now using Item 1 of Lemma A.2.3 to handle the weight 0 reduction steps). Let i^* be the smallest index such that $\tilde{w}_{i^*} = 0$. Using the same argument as above on the (possibly empty) prefix up to index $i^* - 1$ (where all $\tilde{w}_j$ are non-zero) we find

$$\vdash P_{i^*-1} : \left\{ \begin{matrix} [a,b] \\ \frac{1}{w_{i^*-1}} \cdot [c,d] \end{matrix} \right\}$$

Note that this is well defined because $\tilde{w}_j > 0$ for $j < i^*$. Item 2 of Lemma A.2.3 shows $0 \in \frac{1}{w_{i^*-1}} \cdot [c,d]$, which already implies $w = 0 \in [c,d]$, as required. □

A.2.2 Weak Completeness

Theorem 3.4.4. *Let $\vdash P : \alpha$ be a simply-typed program. There exists a weighted interval type $\mathcal{A}$ such that $\vdash P : \mathcal{A}$.*

Proof. For every *simple* type α, we define a weighted type $\mathcal{A}_\alpha$ and weightless type σ_α by mutual recursion as follows.

$$\sigma_{\mathbf{R}} := [-\infty, \infty]$$

$$\sigma_{\alpha\to\beta} := \sigma_\alpha \to \mathcal{A}_\beta$$
$$\mathcal{A}_\alpha := \left\{ \begin{matrix} \sigma_\alpha \\ [0,\infty] \end{matrix} \right\}$$

That is, we insert $[-\infty,\infty]$ for values and $[0,\infty]$ for weights in all locations. We claim that if $\vdash P : \alpha$ in the simple type system, then $\vdash P : \mathcal{A}_\alpha$ in the weight-aware interval type system.

For the proof, we strengthen the induction hypothesis by claiming: if $\Gamma \vdash P : \alpha$ in the simple type system then $\Gamma^{\uparrow\sigma} \vdash P : \mathcal{A}_\alpha$ where

$$\Gamma^{\uparrow\sigma} := \{x : \sigma_\alpha \mid x : \alpha \in \Gamma\}$$

The claim can be proved by simple induction on the derivation of $\Gamma \vdash P : \alpha$ using the respective typing rule for the interval type system possibly followed by (Sub) for typing rules that yield proper subtypes of $\mathcal{A}_\alpha$.

Note that this typing derivation does not contain any useful information to improve the precision of GuBPI. □

A.2.3 Constraint-based Type Inference

In this section we formalize the constraint-based type inference algorithm and sketch our constraints-solving method based on worklist and widening. The overarching idea is to substitute intervals with variables v_i, called interval variables, and to encode typability as a constraint system. As we work in the restricted interval domain (as opposed to e.g. full first-order refinements), the resulting constraints can be solved very efficiently, which is crucial to the practicality of our tool.

Symbolic types Symbolic types are defined by the following grammar:

$$\kappa := v_i \mid \kappa \to \aleph \qquad \aleph := \left\{ \begin{matrix} \kappa \\ v_j \end{matrix} \right\}$$

where v_i, v_j are interval variables. Symbolic types are identical to interval types but use interval variables instead of intervals as first-order types and in the weight bound.

Constraints Constraints on interval variables come in three forms:

$$c := \; v_n \equiv [a,b] \;\Big|\; \kappa_1 \sqsubseteq \kappa_2 \;\Big|\; v_n \equiv \underline{f}(v_{n_1}, \ldots, v_{n_{|f|}})$$

where f is a primitive function and $[a,b]$ an interval. That is, a constraint can either equate an interval variable to a particular interval, require a symbolic type κ_1 to be a subtype of a type κ_2, or equate an interval variable to the result of a function applied to interval variables. Note that due to the compositional nature of our subtype relation $\sqsubseteq$ (which extends to symbolic types) we can restrict ourself to constraints of the form $v_1 \sqsubseteq v_2$ because each constraint of the form $\kappa_1 \sqsubseteq \kappa_2$ or $\aleph_1 \sqsubseteq \aleph_2$ (with identical base types) can be reduced to an equivalent set of constraints on interval variables by the definition of the subtype relation.

Symbolic type system In the presentation of our symbolic type inference system, we aim to stay as close as possible to the implementation. Thus we describe it as an impure type system, meaning that our typing rules have side effects. In our case, typing rules can generate fresh interval variables.

For a simple type α, we write $\mathit{fresh}(\alpha)$ for the symbolic weightless type obtained by replacing every base type $\mathbf{R}$ with a fresh interval variable v_n (and adding weights given by fresh interval variables where needed). We write $\mathit{fresh}()$ for $\mathit{fresh}(\mathbf{R})$. For a symbolic type κ we write $\mathit{base}(\kappa)$ for the underlying simple type (defined in the obvious way).

Our constraint generation system is given in Fig. A.1. Judgments have the form $\Gamma \vdash M : \aleph, C$ where Γ maps variables to weightless symbolic types, $\aleph$ is a weighted symbolic type and C a list of constraints on the interval variables. The rules follow the structure of the system in Fig. 3.5 but replace all operations on intervals with interval variables and constraints. The term structure directly determines the symbolic typing derivation; there are no choices to be made, contrary to Fig. 3.5, which requires "cleverness", for example to find a suitable interval for an argument in the fixpoint rule. Note that in our system, we assume that the *simple* types of arguments of abstractions and fixpoints are given. These types can be determined by a simple prior run of any standard type inference algorithm.

From symbolic to concrete types An assignment A is a mapping from interval variables to concrete intervals. Given a symbolic type κ, we define the interval type κ^A by replacing every interval variable in κ with the concrete interval assigned to it in A. For a weighted symbolic type $\aleph$, we define $\aleph^A$ in the same way. Given a set of constraints C, we say that A satisfies C, written $A \models C$ if all constraints in C are satisfied (defined in the obvious way).

Theorem A.2.4. *If* $\vdash M : \aleph, C$ *and* $A \models C$ *then* $\vdash M : \aleph^A$.

Proof. This can be shown by induction on the structure of the term, which also determines the symbolic typing derivation. From this, we obtain a valid interval typing derivation by replacing interval variables in the derivation with the concrete intervals assigned to them in A, and by applying the (Sub) rule in places where subtyping constraints are introduced. □

This theorem states that solutions to our constraints directly give us valid judgments in our interval type system, which allows us to invoke Theorem 3.4.3.

Solving Constraints To solve the resulting constraints, we employ known techniques from abstract interpretation [P. Cousot and R. Cousot 1977]. Again, note that due to the simplicity of our constraints, our approach avoids expensive calls to a theorem prover. When solving a set of constraints C, we are interested in the smallest solution, i.e. an assignment A with $A \models C$ where the intervals in $\mathcal{A}$ are smallest possible w.r.t. $\sqsubseteq$.

Naïve algorithm A naïve attempt to find a satisfying assignment for a set of constraints would be to iterate over the constraints and to extend the current assignment (initially chosen to map all elements to the bottom element $\bot$ in the interval domain, i.e. an empty interval) whenever needed. For example, if $v_i \sqsubseteq v_j$ is not satisfied by

$$\frac{x : \kappa \in \Gamma \qquad \nu = \mathit{fresh}()}{\Gamma \vdash x : \begin{Bmatrix}\kappa\\ \nu\end{Bmatrix}, \{\nu \equiv \mathbf{1}\}} \qquad \frac{\kappa = \mathit{fresh}(\alpha) \qquad \nu = \mathit{fresh}() \qquad \Gamma; x : \kappa \vdash M : \aleph, C}{\Gamma \vdash \lambda x^{\alpha}.M : \begin{Bmatrix}\kappa \to \aleph\\ \nu\end{Bmatrix}, C \cup \{\nu \equiv \mathbf{1}\}}$$

$$\frac{\Gamma \vdash M : \begin{Bmatrix}\nu_1\\ \nu_2\end{Bmatrix}, C \qquad \nu = \mathit{fresh}()}{\Gamma \vdash \mathsf{score}(M) : \begin{Bmatrix}\nu_1\\ \nu\end{Bmatrix}, C \cup \{\nu \equiv \nu_1 \times \nu_2\}} \qquad \frac{\nu_1 = \mathit{fresh}() \qquad \nu_2 = \mathit{fresh}()}{\Gamma \vdash \underline{r} : \begin{Bmatrix}\nu_1\\ \nu_2\end{Bmatrix}, \{\nu_1 \equiv [r, r], \nu_2 \equiv \mathbf{1}\}}$$

$$\frac{\kappa = \mathit{fresh}(\alpha) \qquad \kappa_1 = \mathit{fresh}(\beta) \qquad \nu, \nu_1 = \mathit{fresh}() \qquad \Gamma; \varphi : \kappa \to \begin{Bmatrix}\kappa_1\\ \nu_1\end{Bmatrix}; x : \kappa \vdash M : \begin{Bmatrix}\kappa_2\\ \nu_2\end{Bmatrix}, C}{\Gamma \vdash \mathsf{fix}_x^{\varphi:\alpha\to\beta}.M : \left\{\begin{matrix}\kappa \to \begin{Bmatrix}\kappa_2\\ \nu_2\end{Bmatrix}\\ \nu\end{matrix}\right\}, C \cup \{\nu \equiv \mathbf{1}, \kappa_2 \sqsubseteq \kappa_1, \nu_2 \sqsubseteq \nu_1\}}$$

$$\frac{\nu, \nu' = \mathit{fresh}()}{\Gamma \vdash \mathsf{sample} : \begin{Bmatrix}\nu\\ \nu'\end{Bmatrix}, \{\nu \equiv [0, 1], \nu' \equiv \mathbf{1}\}}$$

$$\frac{\Gamma \vdash M : \left\{\begin{matrix}\kappa_1 \to \begin{Bmatrix}\kappa_2\\ \nu_2\end{Bmatrix}\\ \nu_1\end{matrix}\right\}, C_1 \qquad \Gamma \vdash N : \begin{Bmatrix}\kappa_3\\ \nu_3\end{Bmatrix}, C_2 \qquad \nu = \mathit{fresh}()}{\Gamma \vdash MN : \begin{Bmatrix}\kappa_2\\ \nu\end{Bmatrix}, C_1 \cup C_2 \cup \{\kappa_3 \sqsubseteq \kappa_1, \nu \equiv \nu_1 \times \nu_2 \times \nu_3\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \begin{Bmatrix}\nu_1\\ \nu_2\end{Bmatrix}, C_M \qquad \Gamma \vdash N : \begin{Bmatrix}\kappa_1\\ \nu_3\end{Bmatrix}, C_N \qquad \Gamma \vdash P : \begin{Bmatrix}\kappa_2\\ \nu_4\end{Bmatrix}, C_P\\ \kappa = \mathit{fresh}(\mathit{base}(\kappa_1)) \qquad \nu, \nu' = \mathit{fresh}()\end{array}}{\Gamma \vdash \mathsf{if}\ M \le 0\ \{N\}\ \mathsf{else}\ \{P\} : \begin{Bmatrix}\kappa\\ \nu\end{Bmatrix}, C_M \cup C_N \cup C_P \cup \{\kappa_1 \sqsubseteq \kappa, \kappa_2 \sqsubseteq \kappa, \nu \equiv \nu_2 \times \nu', \nu_3 \sqsubseteq \nu', \nu_4 \sqsubseteq \nu'\}}$$

$$\frac{\Gamma \vdash M_1 : \begin{Bmatrix}\nu_1\\ \nu_1'\end{Bmatrix}, C_1 \qquad \cdots \qquad \Gamma \vdash M_{|f|} : \begin{Bmatrix}\nu_{|f|}\\ \nu_{|f|}'\end{Bmatrix}, C_{|f|} \qquad \nu, \nu' = \mathit{fresh}()}{\Gamma \vdash f(M_1, \ldots, M_{|f|}) : \begin{Bmatrix}\nu\\ \nu'\end{Bmatrix}, \bigcup_{i=1}^{|f|} C_i \cup \{\nu \equiv \underline{f}(\nu_1, \ldots, \nu_{|f|}), \nu' \equiv \prod_{i=1}^{|f|} \nu_i'\}}$$

Figure A.1: Symbolic weight-aware type system.

assignment A, we can update A by mapping v_j to $A(v_j) \sqcup A(v_i)$. As is well known from abstract interpretation, this naïve approach may not terminate because the interval domain is not chain-complete. For instance, consider the constraints $C = \{v_1 = [0,0], v_2 = [1,1], v_1 \sqsubseteq v_3, v_3 \equiv v_3 + v_2\}$. The minimal solution is $\{v_1 \mapsto [0,0], v_2 \mapsto [1,1], v_3 \mapsto [0,\infty]\}$, but the algorithm never terminates and instead assigns the ascending chain $[0,0], [0,1], [0,2], \ldots$ to v_3.

Widening To remedy the above problem, we use ***widening***, a standard approach to ensure termination of abstract interpretation on domains with infinite chains [P. Cousot and R. Cousot 1977]. Let ∇ be a *widening operator* for intervals. This means that $I_1 \sqcup I_2 \sqsubseteq I_1 \nabla I_2$ for all intervals I_1, I_2 and for every chain $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \cdots$, the chain $(I_i^\nabla)_{i \in \mathbb{N}}$ defined by $I_0^\nabla := I_0$ and $I_i^\nabla := I_{i-1}^\nabla \nabla I_i$ for $i \geq 1$ stabilizes eventually. A trivial widening operator is given by only allowing intervals to extend to infinity, defined as follows:

$$
\begin{aligned}
\bot \nabla I := I \nabla \bot &:= I \\
[a,b] \nabla [c,d] &:= [a,b] && \text{if } a \leq c \wedge d \leq b \\
[a,b] \nabla [c,d] &:= [a,\infty] && \text{if } a \leq c \wedge d > b \\
[a,b] \nabla [c,d] &:= [-\infty,b] && \text{if } d \leq b \wedge c < a \\
[a,b] \nabla [c,d] &:= [-\infty,\infty] && \text{if } c < a \wedge d > b
\end{aligned}
$$

As soon as the upper bound increases or lower bound decreases, the bound is directly set to $= \infty$ or $-\infty$ respectively. By using the widening operator in each update step of our naive algorithm, we break infinite increasing chains and the resulting algorithm is guaranteed to converge to a satisfying assignment (if one exists).

GuBPI solves constraints by using a standard worklist algorithm, combined with the previous widening operator.

B

Supplementary Material for Chapter 4

Contents

B.1 Details on the Empirical Evaluation 205
B.1.1 Details on the Benchmarks 205
B.1.2 Additional Results 207

B.1 Details on the Empirical Evaluation

B.1.1 Details on the Benchmarks

Population models The probabilistic program for the original population model from Winner and Sheldon [2016] is shown in Fig. B.1a. This program can also be written without the extra variable *New* by writing the two statements involving it as $N +\sim \text{Poisson}(\lambda_1)$. Here we used the same parameter values as Winner and Sheldon [2016]: $\delta = 0.2636$, $\lambda = \Lambda \cdot (0.0257, 0.1163, 0.2104, 0.1504, 0.0428)$, and $\Lambda = 2000$ is the population size parameter. Note that the largest population size considered by Winner and Sheldon [2016] is $\Lambda = 500$. For the observation rate ρ, Winner and Sheldon [2016] consider values from 0.05 to 0.95. For simplicity, we set it to $\rho = 0.2$. The y_k are $k = 4$ simulated data points: $(45, 98, 73, 38)$ for $\Lambda = 2000$.

The modified population example is shown in Fig. B.1b, where the arrival rate is reduced to $\lambda' := 0.1\lambda$ in the case of a natural disaster, which happens with probability 0.1. The rest of the model is the same.

The model of two interacting populations (multitype branching process) is programmed as shown in Fig. B.1c, where $\lambda^{(1)} := 0.9\lambda$, $\lambda^{(2)} := 0.1\lambda$, $\gamma := 0.1$. In this model, some individuals of the first kind (i.e. in N_1) can turn into individuals of the second kind and are added to N_2. The other parameters of this model are the same as before. The data points are $y^{(1)} = (35, 83, 78, 58)$ and $y^{(2)} = (3, 6, 10, 4)$.

$N := \text{Poisson}(\lambda_0);$

$N \sim \text{Binomial}(N, \delta);$
$New \sim \text{Poisson}(\lambda_1);$
$N := New + N;$
observe $y_1 \sim \text{Binomial}(N, \rho);$

$\vdots$

(a) Original model from Winner and Sheldon [2016].

$N := \text{Poisson}(\lambda_0);$

$N \sim \text{Binomial}(N, \delta);$
$Disaster \sim \text{Bernoulli}(0.1);$
if $Disaster = 1$ {
  $N +\sim \text{Poisson}(\lambda_1')$
} else {
  $N +\sim \text{Poisson}(\lambda_1)$
}
observe $y_1 \sim \text{Binomial}(N, \rho);$

$\vdots$

(b) Model with randomly modified arrival rate.

$N_1 \sim \text{Poisson}(\lambda_0^{(1)});$
$N_2 \sim \text{Poisson}(\lambda_0^{(2)});$

$N_2 +\sim \text{Binomial}(N_1, \gamma);$
$N_1 \sim \text{Binomial}(N_1, \delta_1);$
$N_2 \sim \text{Binomial}(N_2, \delta_2);$
$N_1 +\sim \text{Poisson}(\lambda_1^1);$
$N_2 +\sim \text{Poisson}(\lambda_1^2);$
observe $y_1^{(1)} \sim \text{Binomial}(N_1, \rho);$
observe $y_1^{(2)} \sim \text{Binomial}(N_2, \rho);$

$\vdots$

(c) Two interacting populations.

Figure B.1: The program code for the population model variations.

Bayesian change point analysis Bayesian change point analysis is about detecting a change in the frequency of certain events over time. An example is the frequency of coal mining disasters in the United States from 1851 to 1962, as discussed in the PyMC3 tutorial [Salvatier et al. 2016]. We use the same model as in Salvatier et al. [2016], and its probabilistic program is shown in Fig. B.2a. This situation can be modeled with two Poisson distributions with parameters Λ_1 (before the change) and Λ_2 (after the change). Suppose we have observations $y_1, \ldots, y_n$, with $y_t \sim \text{Poisson}(\Lambda_1)$ if $t < T$ and $y_t \sim \text{Poisson}(\Lambda_2)$ if $t \geq T$. The parameters Λ_1, Λ_2 are given exponential priors $\text{Exponential}(1)$. The change point T itself is assumed to be uniformly distributed: $T \sim \text{Uniform}\{1..n\}$. The probabilistic program code is shown in Fig. B.2a, where $n = 111$ and $y = (4, 5, 4, 0, 1, 4, 3, 4, 0, 6, 3, 3, 4, 0, 2, 6, 3, 3, 5, 4, 5, 3, 1, 4, 4, 1, 5, 5, 3, 4, 2, 5, 2, 2, 3, 4, 2, 1, 3, \text{n/a}, 2, 1, 1, 1, 1, 3, 0, 0, 1, 0, 1, 1, 0, 0, 3, 1, 0, 3, 2, 2, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 2, 1, 0, 0, 0, 1, 1, 0, 2, 3, 3, 1, \text{n/a}, 2, 1, 1, 1, 1, 2, 4, 2, 0, 0, 1, 4, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1)$ where "n/a" indicates missing data (which is omitted in the program).

In fact, this program can be rewritten to the more efficient form shown in Fig. B.2b, to minimize the number of variables needed.

Mixture model In the binary mixture model, the frequency events are observed from an equal-weight mixture of two Poisson distributions with different parameters Λ_1, Λ_2, which are in discrete steps of 0.1. This is implemented by imposing a geometric prior and multiplying the rate by 0.1 in the Poisson distribution: $\text{Poisson}(0.1 \cdot \Lambda_1)$. The data is the same as for the change point model. The task is to infer the first rate Λ_1. The probabilistic program is shown in Fig. B.3a.

```
T ~ Uniform{1..n};
Λ1 ~ Exponential(1);
Λ2 ~ Exponential(1);
if 1 < T {
    observe y1 ~ Poisson(Λ1);
} else {
    observe y1 ~ Poisson(Λ2);
}
⋮
if n < T {
    observe yn ~ Poisson(Λ1);
} else {
    observe yn ~ Poisson(Λ2);
}
```

(a) Change point model from Salvatier et al. [2016].

```
T ~ Uniform{1..n};
Λ ~ Exponential(1);
if 1 ~ Bernoulli(1/n) {
    observe y1 ~ Poisson(Λ);
    Λ ~ Exponential(1);
    observe y2 ~ Poisson(Λ);
    ⋮
    observe yn ~ Poisson(Λ);
    T := 1;
} else {if 1 ~ Bernoulli(1/(n − 1)) {
    observe y1 ~ Poisson(Λ);
    observe y2 ~ Poisson(Λ);
    Λ ~ Exponential(1);
    observe y3 ~ Poisson(Λ);
    ⋮
    observe yn ~ Poisson(Λ);
    T := 2;
} else {
    ⋱
}}
```

(b) More efficient version of the change point model.

Figure B.2: The program code for the Bayesian change point analysis.

Hidden Markov model The hidden Markov example is based on Saad et al. [2021, Section 2.2]. The hidden state Z can be 0 or 1 and transitions to the other state with probability 0.2. The rate of the Poisson-distributed number of observed events depends on the state Z and is $0.1 \cdot \Lambda_1$ or $0.1 \cdot \Lambda_2$, where we impose a geometric prior on Λ_1 and Λ_2. As for the mixture model, this discretizes the rates in steps of 0.1. The inference problem is to infer the first rate Λ_1. The probabilistic program is shown in Fig. B.3b, where y_k are $m = 30$ simulated data points from the true rates $\lambda_1 := 0.5$ and $\lambda_2 := 2.5$. Concretely, we have $y = (2, 2, 4, 0, 0, 0, 0, 0, 1, 1, 0, 2, 4, 3, 3, 5, 1, 2, 3, 1, 3, 3, 0, 0, 2, 0, 0, 2, 6, 1)$.

B.1.2 Additional Results

In the main text, the data we presented was mainly in form of the total variation distance (TVD) between the true posterior (computed by our method) and the approximated posterior (computed by Anglican's inference algorithms). In this section, we present additional evidence in two forms:

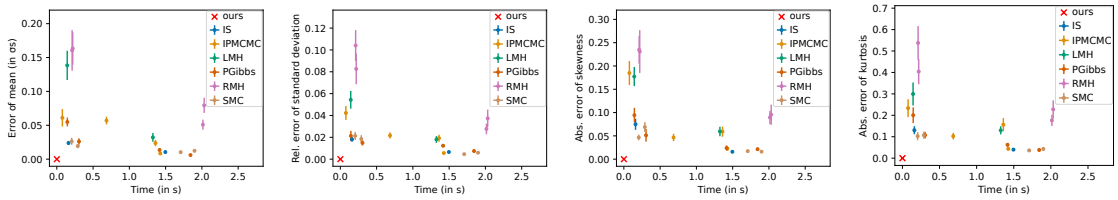
```
Λ1 ~ Geometric(0.1);
Λ2 ~ Geometric(0.1);
if 1 ~ Bernoulli(0.5) {
    observe y1 ~ Poisson(0.1 · Λ1)
} else {
    observe y1 ~ Poisson(0.1 · Λ2)
}
⋮
if 1 ~ Bernoulli(0.5) {
    observe ym ~ Poisson(0.1 · Λ1)
} else {
    observe ym ~ Poisson(0.1 · Λ2)
}
```

(a) Program code for the mixture model.

```
Z := 1;
Λ1 ~ Geometric(0.1);
Λ2 ~ Geometric(0.1);
if Z = 0 {
    observe y1 ~ Poisson(0.1 · Λ1);
    Z ~ Bernoulli(0.2);
} else {
    observe y1 ~ Poisson(0.1 · Λ2);
    Z ~ Bernoulli(0.8); }
⋮
if Z = 0 {
    observe ym ~ Poisson(0.1 · Λ1);
    Z ~ Bernoulli(0.2);
} else {
    observe ym ~ Poisson(0.1 · Λ2);
    Z ~ Bernoulli(0.8); }
```

(b) Program code for the HMM.

Figure B.3: The program code for the mixture and HMM model.

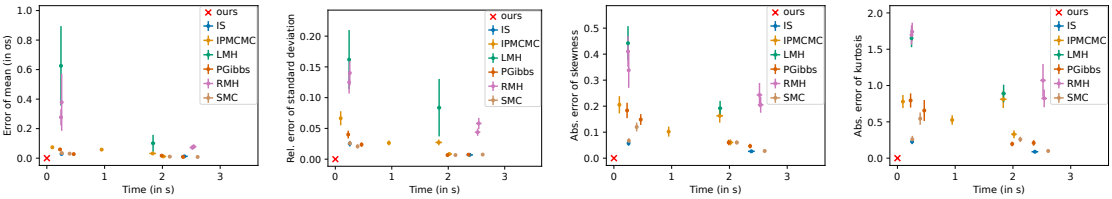


(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.4: Comparison of moments for the original population model.

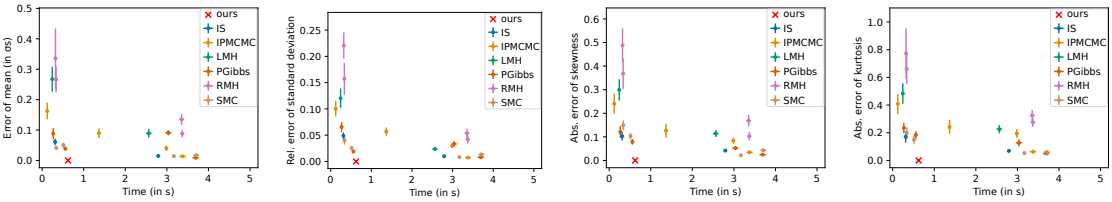
- errors of the posterior moments (mean, standard deviation, skewness, and kurtosis) in Figs. B.4 to B.9, and
- histograms (Fig. B.10) of the sampled distribution produced by the best MCMC algorithm, where the sampling budget is picked such that the running time is close to that of our method (e.g. 1000 for the (fast) population model and 10000 for the (slower) mixture model).

The additional plots confirm the conclusions from Section 4.4.



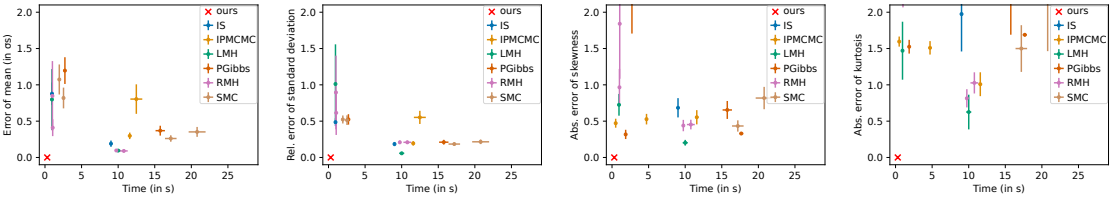
(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.5: Comparison of moments for the modified population model.



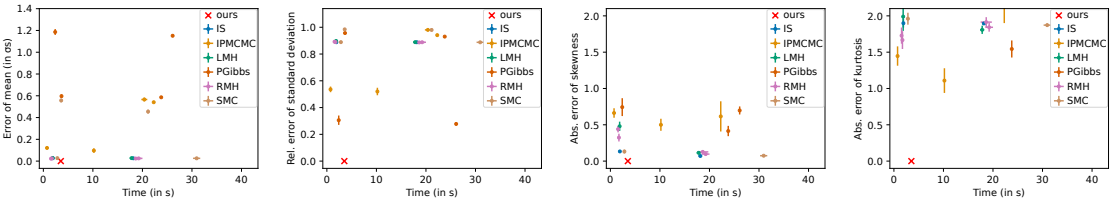
(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.6: Comparison of moments for the two-type population model.



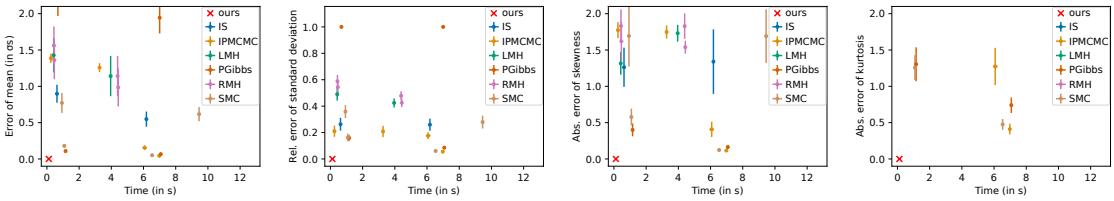
(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.7: Comparison of moments for the change point model.



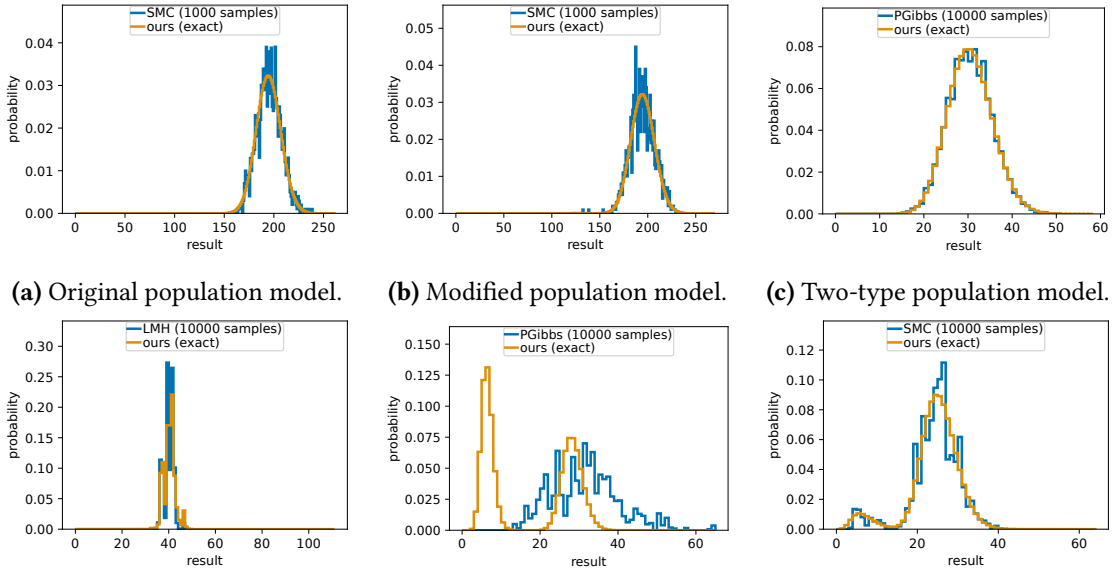
(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.8: Comparison of moments for the mixture model.



(a) Error of the mean **(b)** Error of the standard deviation **(c)** Error of the skewness **(d)** Error of the kurtosis

Figure B.9: Comparison of moments for the HMM.



(a) Original population model. **(b)** Modified population model. **(c)** Two-type population model.

(d) Change point model. **(e)** Mixture model. **(f)** Hidden Markov model.

Figure B.10: Histograms of the exact distribution and the MCMC samples with the lowest TVD and similar running time to our exact method.

C

Supplementary Material for Chapter 6

Contents

C.1 Relation of EGDs and Generating Functions 211

C.1 Relation of EGDs and Generating Functions

EGDs were originally inspired by the shape of their generating functions, which are certain kinds of rational functions. Here we describe and investigate this relationship.

Originally, rational functions seemed like a reasonable shape for the generating function of upper bounds as they consist of polynomials, which are easy to handle computationally. For instance, the existential theory of reals (i.e. polynomial inequalities with rational coefficients) is decidable. However, it turns out that deciding the order on generating functions (defined by $\preceq$ on the corresponding measures) is equivalent to the Positivity Problem of constant-recursive sequences. This equivalence is almost certainly a known result, but we were unable to find a self-contained reference, which is why we prove it here.

Theorem C.1.1. *Let p, q, r, s be polynomials in one variable with rational coefficients. Then deciding whether $\frac{p}{q} \preceq \frac{r}{s}$ is equivalent to the positivity problem of constant-recursive sequences ("Positivity Problem").*

Proof. The condition $\frac{p}{q} \preceq \frac{r}{s}$ is equivalent to $\mathbf{0} \preceq \frac{rq-ps}{qs}$. The coefficients of this power series follow a *constant-recursive sequence* $a_0, a_1, \ldots$ defined by

$$a_m = c_1 a_{m-1} + c_2 a_{m-2} + \cdots + c_d a_{m-d}$$

with rational coefficients $c_1, \ldots, c_d$ and rational initial values $a_0, \ldots, a_d$. So we want to decide whether $a_m \geq 0$ for all $m \in \mathbb{N}$. Let L be the least common multiple of the denominators of $a_0, \ldots, a_{d-1}, c_1, \ldots, c_d$. Then the sequence $b_m := L^{m+1} a_m$ is constant-recursive with

$$b_m = L^{m+1} a_m = c_1 L \cdot L^m a_{m-1} + \cdots + c_d L^d \cdot L^{m+1-d} a_{m-d} = (c_1 L) b_{m-1} + \cdots + (c_d L^d) b_{m-d}$$

and initial values $b_0 := a_0 L, \ldots, b_d := a_d L^{d+1}$. All initial values and coefficients are integers, so b_m is an integer sequence. Then the sequence $b_m + 1$ is also constant recursive (by known closure properties) and its positivity is equivalent to nonnegativity of b_m, which is equivalent to nonnegativity of a_m.

Conversely, the positivity of an integer sequence b_m is clearly equivalent to the nonnegativity of $b_m - 1$. One can find a rational function $f(x)$ whose power series coefficients are $b_m - 1$, reducing the problem to deciding $\mathbf{0} \preceq f$. □

The Positivity Problem has been open for decades and is related to the famous Skolem Problem. Decision procedures are only known for low orders and they rely on deep mathematical results [Ouaknine and Worrell 2014].

As a consequence of this, we have to restrict the shape of generating functions for upper bounds further. We consider this shape:

$$\frac{p(\boldsymbol{x})}{(1-\alpha_1 x_1)\cdots(1-\alpha_n x_n)}$$

where p is a polynomial in n variables with rational coefficients, and $\boldsymbol{\alpha}$ are rational numbers in $[0, 1)$. The generating functions of this shape turn out to correspond to EGDs.

It turns out that EGDs capture exactly the rational generating functions of the shape $\frac{p(\boldsymbol{x})}{(1-\alpha_1 x_1)\cdots(1-\alpha_n x_n)}$ that represent a valid distribution (i.e. whose power series coefficients are nonnegative).

Lemma C.1.2. *Let $G(\boldsymbol{x}) = \frac{p(\boldsymbol{x})}{(1-\alpha_1 x_1)\cdots(1-\alpha_n x_n)}$ be a rational function with p a polynomial in n variables and $\boldsymbol{\alpha} \in [0,1)^n$. Let $d_1, \ldots, d_n$ be the highest degrees of $x_1, \ldots, x_n$ occurring in $p(\boldsymbol{x})$ respectively. Then there is an n-dimensional tensor $\mathbf{Q} \in \mathbb{R}^{[d_1+1]\times\cdots\times[d_n+1]}$ such that*

$$\begin{aligned} G(\boldsymbol{x}) &= \sum_{(i_1,\ldots,i_n)\in[d_1+1]\times\cdots\times[d_n+1]} \mathbf{Q}_{i_1,\ldots,i_n} \frac{x_1^{i_1}\cdots x_n^{i_n}}{(1-\alpha_1 x_1)^{[i_1=d_1]}\cdots(1-\alpha_n x_n)^{[i_n=d_n]}} \\ &= \sum_{(i_1,\ldots,i_n)\in\mathbb{N}^n} \mathbf{Q}_{\min(i_1,d_1),\ldots,\min(i_n,d_n)} \alpha_1^{\max(i_1-d_1,0)}\cdots\alpha_n^{\max(i_n-d_n,0)} \cdot x_1^{i_1}\cdots x_n^{i_n} \end{aligned}$$

where the second equation is the power series expansion of G. In particular, G is a valid generating function if and only if all power series coefficients $\mathbf{Q}_{i_1,\ldots,i_n}$ are nonnegative. In that case, G is the generating function of EGD$(\mathbf{Q}, \boldsymbol{\alpha})$.

Proof. Let $\boldsymbol{x} = (x_1, \boldsymbol{x}')$ and write $p(\boldsymbol{x}) = p_0(x') + p_1(\boldsymbol{x}')x_1 + p_2(x')x_1^2 + \cdots + p_{d_1}(x')x_1^{d_1}$. Then define $p'_0(\boldsymbol{x}') := p_0(\boldsymbol{x}')$ and $p'_i(\boldsymbol{x}') := p_i(\boldsymbol{x}') + \alpha_1 p_{i-1}(\boldsymbol{x}')$ for $i = 1, \ldots, d_1$. We can rewrite $p(\boldsymbol{x})$ as follows:

$$\begin{aligned} &(1-\alpha_1 x_1)\cdot\left(p'_0(\boldsymbol{x}') + p'_1(\boldsymbol{x}')x_1 + p'_2(\boldsymbol{x}')x_1^2 + \cdots + p'_{d_1-1}(\boldsymbol{x}')x_1^{d_1-1} + \frac{p'_{d_1}(\boldsymbol{x}')}{1-\alpha_1 x_1}x_1^{d_1}\right) \\ &= p'_0(\boldsymbol{x}') + (p'_1(x') - \alpha_1 p'_0(\boldsymbol{x}'))x_1 + \cdots + (p'_{d_1}(\boldsymbol{x}') - \alpha_1 p'_{d_1-1}(\boldsymbol{x}'))x_1^{d_1} \\ &= p_0(\boldsymbol{x}') + p_1(\boldsymbol{x}')x_1 + p_2(\boldsymbol{x}')x_1^2 + \cdots + p_{d_1}(\boldsymbol{x}')x_1^{d_1} \\ &= p(\boldsymbol{x}) \end{aligned}$$

Then we can rewrite G as:

$$\begin{aligned} G(\boldsymbol{x}) &= \frac{p(\boldsymbol{x})}{(1-\alpha_1 x_1)\cdots(1-\alpha_n x_n)} \\ &= \frac{p'_0(\boldsymbol{x}')}{(1-\alpha_2 x_2)\cdots(1-\alpha_n x_n)} + \frac{p'_1(\boldsymbol{x}')}{(1-\alpha_2 x_2)\cdots(1-\alpha_n x_n)} x_1 + \cdots \\ &\quad + \frac{p'_{d_1-1}(\boldsymbol{x}')}{(1-\alpha_2 x_2)\cdots(1-\alpha_n x_n)} x_1^{d_1-1} + \frac{p'_{d_1}(\boldsymbol{x}')}{(1-\alpha_2 x_2)\cdots(1-\alpha_n x_n)} \cdot \frac{x_1^{d_1}}{1-\alpha_1 x_1} \\ &= \sum_{i\in[d_1+1]} \frac{p'_i(\boldsymbol{x}')}{(1-\alpha_2 x_2)\cdots(1-\alpha_n x_n)} \cdot \frac{x_1^i}{(1-\alpha_1 x_1)^{[i=d_1]}} \end{aligned}$$

where $[i=j]$ is the Iverson bracket, which is 1 if $i=j$ and 0 otherwise, so the term $a^{[i=j]}$ is a if $i=j$ and 1 otherwise. Iterating this procedure for the other variables $x_2,\ldots,x_n$, we obtain numbers $\mathbf{Q}_{i_1,\ldots,i_n} \in \mathbb{R}$ for $i_j \in [d_j+1]$ such that:

$$\begin{aligned} G(\boldsymbol{x}) &= \sum_{(i_1,\ldots,i_n)\in[d_1+1]\times\cdots\times[d_n+1]} \mathbf{Q}_{i_1,\ldots,i_n} \frac{x_1^{i_1}\cdots x_n^{i_n}}{(1-\alpha_1 x_1)^{[i_1=d_1]}\cdots(1-\alpha_n x_n)^{[i_n=d_n]}} \\ &= \sum_{(i_1,\ldots,i_n)\in\mathbb{N}^n} \mathbf{Q}_{\min(i_1,d_1),\ldots,\min(i_n,d_n)} \alpha_1^{\max(i_1-d_1,0)}\cdots\alpha_n^{\max(i_n-d_n,0)} x_1^{i_1}\cdots x_n^{i_n} \end{aligned}$$

where the second equation is the power series of $G(\boldsymbol{x})$. □

Example C.1.3. Consider the generating function: $G(\boldsymbol{x}) = \frac{0.4+0.3x_1+0.2x_2+0.1x_1x_2}{(1-\alpha_1 x_1)(1-\alpha_2 x_2)}$. We can rewrite the function as:

$$\begin{aligned} G(\boldsymbol{x}) &= \frac{0.4+0.3x_1+0.2x_2+0.1x_1x_2}{(1-\alpha_1 x_1)(1-\alpha_2 x_2)} \\ &= 0.4 + (0.3+0.4\alpha_1)\frac{x_1}{(1-\alpha_1 x_1)} + (0.2+0.4\alpha_2)\frac{x_2}{1-\alpha_2 x_2} \\ &\quad + (0.1+0.3\alpha_2+0.2\alpha_1+0.4\alpha_1\alpha_2)\frac{x_1x_2}{(1-\alpha_1 x_1)(1-\alpha_2 x_2)} \\ &= \mathbf{Q}_{0,0} + \mathbf{Q}_{1,0}\frac{x_1}{1-\alpha_1 x_1} + \mathbf{Q}_{0,1}\frac{x_2}{1-\alpha_2 x_2} + \mathbf{Q}_{1,1}\frac{x_1x_2}{(1-\alpha_1 x_1)(1-\alpha_2 x_2)} \\ &= \mathrm{gf}(\mathsf{EGD}(\mathbf{Q},(\alpha_1,\alpha_2)))(\boldsymbol{x}) \end{aligned}$$

where $\mathbf{Q} \in \mathbb{R}^{[2]\times[2]}$ is given by $\mathbf{Q}_{0,0} = 0.4$, $\mathbf{Q}_{1,0} = 0.3+0.4\alpha_1$, $\mathbf{Q}_{0,1} = 0.2+0.4\alpha_2$, and $\mathbf{Q}_{1,1} = 0.1+0.3\alpha_2+0.2\alpha_1+0.4\alpha_1\alpha_2$.

References

Agarwal, Sanyam and Markus Bläser (2024). "Probabilistic Generating Circuits - Demystified". In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research, pp. 329–342 (cit. on p. 127).

Aguirre, Alejandro, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg (2018). "Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Vol. 10801. Lecture Notes in Computer Science, pp. 214–241. DOI: 10.1007/978-3-319-89884-1_8 (cit. on p. 44).

Albarghouthi, Aws, Loris D'Antoni, Samuel Drews, and Aditya V. Nori (2017). "FairSquare: probabilistic verification of program fairness". In: *Proc. ACM Program. Lang.* 1.OOPSLA, 80:1–80:30. DOI: 10.1145/3133904 (cit. on pp. 46, 90).

Amrollahi, Daneshvar, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovic (2022). "Solving Invariant Generation for Unsolvable Loops". In: *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*. Vol. 13790. Lecture Notes in Computer Science, pp. 19–43. DOI: 10.1007/978-3-031-22308-2_3 (cit. on pp. 46, 188).

Andrieu, Christophe, Arnaud Doucet, and Roman Holenstein (2009). "Particle Markov chain Monte Carlo for efficient numerical simulation". In: *Monte Carlo and quasi-Monte Carlo methods 2008. Proceedings of the 8th international conference Monte Carlo and quasi-Monte Carlo methods in scientific computing, Montréal, Canada, July 6–11, 2008*. Pp. 45–60. DOI: 10.1007/978-3-642-04107-5_3 (cit. on p. 41).

Bagnall, Alexander, Gordon Stewart, and Anindya Banerjee (2023). "Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning". In: *Proc. ACM Program. Lang.* 7.PLDI, pp. 1–24. DOI: 10.1145/3591220 (cit. on p. 47).

Baldoni, Velleda, Nicole Berline, Jesús A. De Loera, Matthias Köppe, and Michèle Vergne (2011). "How to integrate a polynomial over a simplex". In: *Math. Comput.* 80.273, pp. 297–325. DOI: 10.1090/S0025-5718-2010-02378-6 (cit. on p. 80).

Barbosa, Haniel, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar (2022). "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Vol. 13243. Lecture Notes in Computer Science, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cit. on p. 177).

Barthe, Gilles, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub (2013). "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Vol. 8604. Lecture Notes in Computer Science, pp. 146–166. DOI: 10.1007/978-3-319-10082-1_6 (cit. on p. 47).

Barthe, Gilles, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub (2018). "An Assertion-Based Program Logic for Probabilistic Programs". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP*

2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Vol. 10801. Lecture Notes in Computer Science, pp. 117–144. DOI: 10.1007/978-3-319-89884-1_5 (cit. on pp. 44, 47).

Bartocci, Ezio, Laura Kovács, and Miroslav Stankovic (2019). "Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Vol. 11781. Lecture Notes in Computer Science, pp. 255–276. DOI: 10.1007/978-3-030-31784-3_15 (cit. on pp. 46, 188).

Baydin, Atilim Günes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (2017). "Automatic Differentiation in Machine Learning: a Survey". In: *J. Mach. Learn. Res.* 18, 153:1–153:43 (cit. on p. 109).

Baydin, Atilim Günes, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip H. S. Torr, Victor W. Lee, Kyle Cranmer, Prabhat, and Frank Wood (2019). "Etalumis: bringing probabilistic programming to scientific simulators at scale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, 29:1–29:24. DOI: 10.1145/3295500.3356180 (cit. on p. 10).

Beauquier, Joffroy, Maria Gradinariu, and Colette Johnen (1999). "Memory Space Requirements for Self-Stabilizing Leader Election Protocols". In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999*, pp. 199–207. DOI: 10.1145/301308.301358 (cit. on p. 180).

Ben-Or, Michael (1983). "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)". In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pp. 27–30. DOI: 10.1145/800221.806707 (cit. on p. 2).

Betancourt, Michael (2018). *A Conceptual Introduction to Hamiltonian Monte Carlo*. arXiv: 1701.02434 [stat.ME] (cit. on p. 39).

Beutner, Raven and Luke Ong (2021). "On probabilistic termination of functional programs with continuous distributions". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pp. 1312–1326. DOI: 10.1145/3453483.3454111 (cit. on pp. 33, 54, 55, 66, 67, 89).

Beutner, Raven, Luke Ong, and Fabian Zaiser (2022a). "Guaranteed bounds for posterior inference in universal probabilistic programming". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pp. 536–551. DOI: 10.1145/3519939.3523721 (cit. on pp. 15, 52, 138, 187).

Beutner, Raven, Luke Ong, and Fabian Zaiser (2022b). "Guaranteed Bounds for Posterior Inference in Universal Probabilistic Programmings". In: *CoRR* abs/2204.02948. DOI: 10.48550/arXiv.2204.02948 (cit. on pp. 52, 67).

Bhat, Sooraj, Ashish Agarwal, Richard W. Vuduc, and Alexander G. Gray (2012). "A type theory for probability density functions". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pp. 545–556. DOI: 10.1145/2103656.2103721 (cit. on pp. 36, 47).

Bhat, Sooraj, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo (2013). "Deriving Probability Density Functions from Probabilistic Functional Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of*

Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Vol. 7795. Lecture Notes in Computer Science, pp. 508–522. DOI: 10.1007/978-3-642-36742-7_35 (cit. on p. 36).

Bichsel, Benjamin, Timon Gehr, and Martin T. Vechev (2018). "Fine-Grained Semantics for Probabilistic Programs". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Vol. 10801. Lecture Notes in Computer Science, pp. 145–185. DOI: 10.1007/978-3-319-89884-1_6 (cit. on p. 30).

Bingham, Eli, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman (2019). "Pyro: Deep Universal Probabilistic Programming". In: *J. Mach. Learn. Res.* 20, 28:1–28:6 (cit. on pp. 13, 42, 50, 53, 86, 123).

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. ISBN: 0387310738 (cit. on pp. 3, 6, 41).

Bladt, Mogens and Bo Friis Nielsen (2017). *Matrix-Exponential Distributions in Applied Probability*. DOI: https://doi.org/10.1007/978-1-4939-7049-0 (cit. on p. 144).

Boonstoppel, Peter, Cristian Cadar, and Dawson R. Engler (2008). "RWset: Attacking Path Explosion in Constraint-Based Test Generation". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science, pp. 351–366. DOI: 10.1007/978-3-540-78800-3_27 (cit. on p. 83).

Borgström, Johannes, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael (2013). "Measure Transformer Semantics for Bayesian Machine Learning". In: *Log. Methods Comput. Sci.* 9.3. DOI: 10.2168/LMCS-9(3:11)2013 (cit. on p. 22).

Borgström, Johannes, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak (2016). "A lambda-calculus foundation for universal probabilistic programming". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pp. 33–46. DOI: 10.1145/2951913.2951942 (cit. on pp. 26, 33, 34).

Brent, Richard P. and H. T. Kung (1978). "Fast Algorithms for Manipulating Formal Power Series". In: *J. ACM* 25.4, pp. 581–595. DOI: 10.1145/322092.322099 (cit. on p. 113).

Brooks, Stephen P. and Andrew Gelman (1998). "General methods for monitoring convergence of iterative simulations". In: *J. Comput. Graph. Statist.* 7.4, pp. 434–455. DOI: 10.2307/1390675 (cit. on p. 42).

Brown, Christopher W. (2003). "QEPCAD B: a program for computing with semi-algebraic sets using CADs". In: *SIGSAM Bull.* 37.4, pp. 97–108. DOI: 10.1145/968708.968710 (cit. on p. 177).

Büeler, Benno, Andreas Enge, and Komei Fukuda (2000). "Exact Volume Computation for Polytopes: A Practical Study". In: *Polytopes—combinatorics and computation*. Vol. 29. DMV Sem. DOI: 10.1007/978-3-0348-8438-9_6 (cit. on p. 82).

Campillo Navarro, Azucena (2018). "Order statistics and multivariate discrete phase-type distributions". PhD thesis. DTU Compute (cit. on p. 144).

Carpenter, Bob, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell (2017). "Stan : A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1. DOI: 10.18637/jss.v076.i01 (cit. on pp. 7, 10, 42, 49, 123).

Celeux, Gilles, Merrilee Hurn, and Christian P. Robert (2000). "Computational and inferential difficulties with mixture posterior distributions". In: *J. Amer. Statist. Assoc.* 95.451, pp. 957–970. DOI: 10.2307/2669477 (cit. on p. 40).

Chaganty, Arun Tejasvi, Aditya V. Nori, and Sriram K. Rajamani (2013). "Efficiently Sampling Probabilistic Programs via Program Analysis". In: *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*. Vol. 31. JMLR Workshop and Conference Proceedings, pp. 153–160 (cit. on pp. 42, 46, 62).

Chakarov, Aleksandar and Sriram Sankaranarayanan (2013). "Probabilistic Program Analysis with Martingales". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Vol. 8044. Lecture Notes in Computer Science, pp. 511–526. DOI: 10.1007/978-3-642-39799-8_34 (cit. on pp. 46, 188).

Champion, Adrien, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato (2020). "ICE-Based Refinement Type Discovery for Higher-Order Functional Programs". In: *J. Autom. Reason.* 64.7, pp. 1393–1418. DOI: 10.1007/S10817-020-09571-Y (cit. on p. 77).

Chang, J. T. and D. Pollard (1997). "Conditioning as disintegration". In: *Statistica Neerlandica* 51.3, pp. 287–317. DOI: https://doi.org/10.1111/1467-9574.00056 (cit. on p. 27).

Chang, Kuo-Chu and Robert M. Fung (1995). "Symbolic probabilistic inference with both discrete and continuous variables". In: *IEEE Trans. Syst. Man Cybern.* 25.6, pp. 910–916. DOI: 10.1109/21.384253 (cit. on p. 35).

Chatterjee, Krishnendu, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad (2016). "Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 327–342. DOI: 10.1145/2837614.2837639 (cit. on pp. 46, 47, 164, 183, 188).

Chatterjee, Krishnendu, Amir Kafshdar Goharshady, Tobias Meggendorfer, and Dorde Zikelic (2024). "Quantitative Bounds on Resource Usage of Probabilistic Programs". In: *Proc. ACM Program. Lang.* 8.OOPSLA1, pp. 362–391. DOI: 10.1145/3649824 (cit. on pp. 47, 188).

Chen, Mingshuai, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler (2022). "Does a Program Yield the Right Distribution? - Verifying Probabilistic Programs via Generating Functions". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Vol. 13371. Lecture Notes in Computer Science, pp. 79–101. DOI: 10.1007/978-3-031-13185-1_5 (cit. on pp. 93, 98, 100, 108, 109, 127).

Choi, YooJung, Antonio Vergari, and Guy Van den Broeck (2020). *Probabilistic Circuits: A Unifying Framework for Tractable Probabilistic Models*. Accessed: 2024-08-06 (cit. on p. 35).

Claret, Guillaume, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström (2013). "Bayesian inference using data flow analysis". In: *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*, pp. 92–102. DOI: 10.1145/2491411.2491423 (cit. on p. 35).

Cock, David A. (2012). "Verifying Probabilistic Correctness in Isabelle with pGCL". In: *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012*. Vol. 102. EPTCS, pp. 167–178. DOI: 10.4204/EPTCS.102.15 (cit. on p. 47).

Cousot, Patrick and Radhia Cousot (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles,*

California, USA, January 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cit. on pp. 45, 77, 202, 204).

Cousot, Patrick and Michael Monerau (2012). "Probabilistic Abstract Interpretation". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Vol. 7211. Lecture Notes in Computer Science, pp. 169–193. DOI: 10.1007/978-3-642-28869-2_9 (cit. on p. 45).

Culpepper, Ryan and Andrew Cobb (2017). "Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Vol. 10201. Lecture Notes in Computer Science, pp. 368–392. DOI: 10.1007/978-3-662-54434-1_14 (cit. on p. 60).

Cusumano-Towner, Marco, Alexander K. Lew, and Vikash K. Mansinghka (2020). *Automating Involutive MCMC using Probabilistic and Differentiable Programming*. arXiv: 2007.09871 [stat.CO] (cit. on p. 40).

Cusumano-Towner, Marco F. and Vikash K. Mansinghka (2017). "AIDE: An algorithm for measuring the accuracy of probabilistic inference algorithms". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 3000–3010 (cit. on p. 43).

Cusumano-Towner, Marco F., Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka (2019). "Gen: a general-purpose probabilistic programming system with programmable inference". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pp. 221–236. DOI: 10.1145/3314221.3314642 (cit. on pp. 11, 50, 123).

Dagum, Paul and Michael Luby (1993). "Approximating Probabilistic Inference in Bayesian Belief Networks is NP-Hard". In: *Artif. Intell.* 60.1, pp. 141–153. DOI: 10.1016/0004-3702(93)90036-B (cit. on pp. 16, 143).

DeGroot, Morris H. and Mark J. Schervish (2018). *Probability and statistics (classic version)*. 4th ed. Pearson Modern Classics for Advanced Statistics (cit. on p. 2).

De Moura, Leonardo Mendonça and Nikolaj S. Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 177).

Doucet, Arnaud, Nando de Freitas, and Neil J. Gordon (2001). *Sequential Monte Carlo Methods in Practice*. Statistics for Engineering and Information Science. DOI: 10.1007/978-1-4757-3437-9 (cit. on p. 40).

Duane, Simon, Anthony D. Kennedy, Brian J. Pendleton, and Duncan Roweth (1987). "Hybrid Monte Carlo". In: *Physics letters B* 195.2. DOI: 10.1016/0370-2693(87)91197-X (cit. on p. 39).

Dutta, Saikat, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic (2018). "Testing probabilistic programming systems". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pp. 574–586. DOI: 10.1145/3236024.3236057 (cit. on p. 48).

Dyer, Martin E. and Alan M. Frieze (1988). "On the Complexity of Computing the Volume of a Polyhedron". In: *SIAM J. Comput.* 17.5, pp. 967–974. DOI: 10.1137/0217060 (cit. on p. 81).

Eberl, Manuel, Johannes Hölzl, and Tobias Nipkow (2015). "A Verified Compiler for Probability Density Functions". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9032. Lecture Notes in Computer Science, pp. 80–104. DOI: 10.1007/978-3-662-46669-8_4 (cit. on p. 47).

Feller, William (1968). *An introduction to probability theory and its applications, volume 1*. 3rd ed. Wiley Series in Probability and Statistics (cit. on p. 93).

Fierens, Daan, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt (2015). "Inference and learning in probabilistic logic programs using weighted Boolean formulas". In: *Theory Pract. Log. Program.* 15.3, pp. 358–401. DOI: 10.1017/S1471068414000076 (cit. on p. 49).

Fischer, Michael J., Nancy A. Lynch, and Mike Paterson (1983). "Impossibility of Distributed Consensus with One Faulty Process". In: *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, pp. 1–7. DOI: 10.1145/588058.588060 (cit. on p. 2).

Flajolet, P., É. Fusy, O. Gandouet, and F. Meunier (2007). "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *2007 Conference on analysis of algorithms, AofA 07. Papers from the 13th Conference held in Juan-les-Pins, France, June 17–22, 2007.* Pp. 127–146 (cit. on p. 11).

Fraser, D. A. S., Philip McDunnough, Amir Naderi, and André Plante (1995). "On the definition of probability densities and sufficiency of the likelihood map". In: *Probab. Math. Stat.* 15, pp. 301–310. ISSN: 0208-4147 (cit. on pp. 20, 22, 27).

Freeman, Timothy S. and Frank Pfenning (1991). "Refinement Types for ML". In: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pp. 268–277. DOI: 10.1145/113445.113468 (cit. on pp. 77, 90).

Freivalds, Rusins (1979). "Fast Probabilistic Algorithms". In: *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*. Vol. 74. Lecture Notes in Computer Science, pp. 57–69. DOI: 10.1007/3-540-09526-8_5 (cit. on p. 11).

Frey, Brendan J. and David J. C. MacKay (1997). "A Revolution: Belief Propagation in Graphs with Cycles". In: *Advances in Neural Information Processing Systems 10, NIPS Conference, Denver, Colorado, USA, 1997*, pp. 479–485 (cit. on p. 42).

Garfinkel, Simson L. and Philip Leclerc (2020). "Randomness Concerns when Deploying Differential Privacy". In: *WPES'20: Proceedings of the 19th Workshop on Privacy in the Electronic Society, Virtual Event, USA, November 9, 2020*, pp. 73–86. DOI: 10.1145/3411497.3420211 (cit. on p. 2).

Ge, Hong, Kai Xu, and Zoubin Ghahramani (2018). "Turing: A Language for Flexible Probabilistic Inference". In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Vol. 84. Proceedings of Machine Learning Research. 09–11 April, pp. 1682–1690 (cit. on pp. 50, 123).

Gehr, Timon, Sasa Misailovic, and Martin T. Vechev (2016). "PSI: Exact Symbolic Inference for Probabilistic Programs". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Vol. 9779. Lecture Notes in

Computer Science, pp. 62–83. DOI: 10.1007/978-3-319-41528-4_4 (cit. on pp. 35, 36, 82–84, 93, 121, 122, 180).

Gehr, Timon, Samuel Steffen, and Martin T. Vechev (2020). "λPSI: exact inference for higher-order probabilistic programs". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pp. 883–897. DOI: 10.1145/3385412.3386006 (cit. on pp. 36, 83, 84).

Geldenhuys, Jaco, Matthew B. Dwyer, and Willem Visser (2012). "Probabilistic symbolic execution". In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pp. 166–176. DOI: 10.1145/2338965.2336773 (cit. on pp. 46, 62).

Gelman, Andrew, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin (2013). *Bayesian Data Analysis*. DOI: 10.1201/b16018 (cit. on pp. 3, 6, 7, 13, 42, 43).

Gelman, Andrew and Donald B Rubin (1992). "Inference from iterative simulation using multiple sequences". In: *Statistical science* 7.4, pp. 457–472 (cit. on p. 42).

Gelman, Andrew, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák (2020). *Bayesian Workflow*. arXiv: 2011.01808 [stat.ME] (cit. on pp. 43, 191).

Ghahramani, Zoubin (2015). "Probabilistic machine learning and artificial intelligence". In: *Nat.* 521.7553, pp. 452–459. DOI: 10.1038/nature14541 (cit. on p. 6).

Gilks, W. R., A. Thomas, and D. J. Spiegelhalter (1994). "A Language and Program for Complex Bayesian Modelling". In: *Journal of the Royal Statistical Society: Series D (The Statistician)* 43.1, pp. 169–177. DOI: https://doi.org/10.2307/2348941 (cit. on p. 49).

Goodman, Noah D., Vikash K. Mansinghka, Daniel M. Roy, Kallista A. Bonawitz, and Joshua B. Tenenbaum (2008). "Church: a language for generative models". In: *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pp. 220–229 (cit. on p. 49).

Goodman, Noah D. and Andreas Stuhlmüller (2014). *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org. Accessed: 2024-08-06 (cit. on pp. 50, 123).

Gordon, Andrew D., Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani (2014). "Probabilistic programming". In: *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pp. 167–181. DOI: 10.1145/2593882.2593900 (cit. on p. 6).

Gorinova, Maria I., Andrew D. Gordon, Charles Sutton, and Matthijs Vákár (2022). "Conditional Independence by Typing". In: *ACM Trans. Program. Lang. Syst.* 44.1, 4:1–4:54. DOI: 10.1145/3490421 (cit. on p. 48).

Gorinova, Maria I., Dave Moore, and Matthew D. Hoffman (2020). "Automatic Reparameterisation of Probabilistic Programs". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 3648–3657 (cit. on pp. 40, 48, 84).

Griewank, Andreas and Andrea Walther (2008). *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition*. DOI: 10.1137/1.9780898717761 (cit. on pp. 110, 113).

Grosse, Roger B., Siddharth Ancha, and Daniel M. Roy (2016). "Measuring the reliability of MCMC inference with bidirectional Monte Carlo". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 2451–2459 (cit. on p. 43).

Grosse, Roger B., Zoubin Ghahramani, and Ryan P. Adams (2015). "Sandwiching the marginal likelihood using bidirectional Monte Carlo". In: *CoRR* abs/1511.02543. DOI: 10.48550/arXiv.1511.02543 (cit. on p. 43).

Gut, Allan (2013). *Probability: A Graduate Course*. 2nd ed. Springer Texts in Statistics, pp. xxvi+600. DOI: 10.1007/978-1-4614-4708-5 (cit. on p. 93).

Harviainen, Juha, Vaidyanathan Peruvemba Ramaswamy, and Mikko Koivisto (2023). "On inference and learning with probabilistic generating circuits". In: *Uncertainty in Artificial Intelligence, UAI 2023, July 31 - 4 August 2023, Pittsburgh, PA, USA*. Vol. 216. Proceedings of Machine Learning Research, pp. 829–838 (cit. on p. 127).

Hastings, W. K. (1970). "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57, pp. 97–109. DOI: 10.1093/biomet/57.1.97 (cit. on p. 39).

Herman, Ted (1990). "Probabilistic Self-Stabilization". In: *Inf. Process. Lett.* 35.2, pp. 63–67. DOI: 10.1016/0020-0190(90)90107-9 (cit. on p. 180).

Heule, Stefan, Marc Nunkesser, and Alexander Hall (2013). "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm". In: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pp. 683–692. DOI: 10.1145/2452376.2452456 (cit. on pp. 2, 11).

Heunen, Chris, Ohad Kammar, Sam Staton, and Hongseok Yang (2017). "A convenient category for higher-order probability theory". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pp. 1–12. DOI: 10.1109/LICS.2017.8005137 (cit. on pp. 10, 33).

Hirata, Michikazu, Yasuhiko Minamide, and Tetsuya Sato (2022). "Program Logic for Higher-Order Probabilistic Programs in Isabelle/HOL". In: *Functional and Logic Programming - 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10-12, 2022, Proceedings*. Vol. 13215. Lecture Notes in Computer Science, pp. 57–74. DOI: 10.1007/978-3-030-99461-7_4 (cit. on p. 47).

Hoffman, Matthew D., David M. Blei, Chong Wang, and John W. Paisley (2013). "Stochastic variational inference". In: *J. Mach. Learn. Res.* 14.1, pp. 1303–1347. DOI: 10.5555/2567709.2502622 (cit. on p. 41).

Hoffman, Matthew D. and Andrew Gelman (2014). "The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo". In: *J. Mach. Learn. Res.* 15.1, pp. 1593–1623. DOI: 10.5555/2627435.2638586 (cit. on p. 39).

Holtzen, Steven, Guy Van den Broeck, and Todd D. Millstein (2020). "Scaling exact inference for discrete probabilistic programs". In: *Proc. ACM Program. Lang.* 4.OOPSLA, 140:1–140:31. DOI: 10.1145/3428208 (cit. on pp. 35, 93, 94, 121, 122, 139, 192).

Hölzl, Johannes (2016). "Formalising Semantics for Expected Running Time of Probabilistic Programs". In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. Vol. 9807. Lecture Notes in Computer Science, pp. 475–482. DOI: 10.1007/978-3-319-43144-4_30 (cit. on p. 47).

Hougardy, Stefan, Fabian Zaiser, and Xianghui Zhong (2020). "The approximation ratio of the 2-Opt Heuristic for the metric Traveling Salesman Problem". In: *Oper. Res. Lett.* 48.4, pp. 401–404. DOI: 10.1016/J.ORL.2020.05.007 (cit. on p. 16).

Huang, Zixin, Saikat Dutta, and Sasa Misailovic (2021). "AQUA: Automated Quantized Inference for Probabilistic Programs". In: *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*. Vol. 12971. Lecture Notes in Computer Science, pp. 229–246. DOI: 10.1007/978-3-030-88885-5_16 (cit. on p. 90).

Hur, Chung-Kil, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel (2015). "A Provably Correct Sampler for Probabilistic Programs". In: *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*. Vol. 45. LIPIcs, pp. 475–488. DOI: 10.4230/LIPICS.FSTTCS.2015.475 (cit. on pp. 29, 30, 47).

Hurd, Joe (2003). *Formal Verification of Probabilistic Algorithms*. Tech. rep. UCAM-CL-TR-566. University of Cambridge, Computer Laboratory. DOI: 10.48456/tr-566 (cit. on p. 10).

Hurd, Joe, Annabelle McIver, and Carroll Morgan (2005). "Probabilistic guarded commands mechanized in *HOL*". In: *Theor. Comput. Sci.* 346.1, pp. 96–112. DOI: 10.1016/J.TCS.2005.08.005 (cit. on p. 47).

Israeli, Amos and Marc Jalfon (1990). "Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion". In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pp. 119–131. DOI: 10.1145/93385.93409 (cit. on p. 180).

Jacobs, Jules (2021). "Paradoxes of probabilistic programming: and how to condition on events of measure zero with infinitesimal probabilities". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–26. DOI: 10.1145/3434339 (cit. on p. 27).

Jansen, Nils, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen (2016). "Bounded Model Checking for Probabilistic Programs". In: *ATVA 2016: Automated Technology for Verification and Analysis - 14th International Symposium, Chiba, Japan, October 17-20, 2016, Proceedings*. Vol. 9938. Lecture Notes in Computer Science, pp. 68–85. DOI: 10.1007/978-3-319-46520-3_5 (cit. on p. 133).

Kaminski, Benjamin Lucien (2019). "Advanced weakest precondition calculi for probabilistic programs". PhD thesis. RWTH Aachen University, Germany (cit. on p. 45).

Kaminski, Benjamin Lucien, Joost-Pieter Katoen, and Christoph Matheja (2019). "On the hardness of analyzing probabilistic programs". In: *Acta Informatica* 56.3, pp. 255–285. DOI: 10.1007/S00236-018-0321-1 (cit. on pp. 11, 16, 44).

Kenyon-Roberts, Andrew and Luke Ong (2021). "Supermartingales, Ranking Functions and Probabilistic Lambda Calculus". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470550 (cit. on p. 60).

Khajwal, Basim, Luke Ong, and Dominik Wagner (2023). "Fast and Correct Gradient-Based Optimisation for Probabilistic Programming via Smoothing". In: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*. Vol. 13990. Lecture Notes in Computer Science, pp. 479–506. DOI: 10.1007/978-3-031-30044-8_18 (cit. on p. 48).

King, James C. (1976). "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7, pp. 385–394. DOI: 10.1145/360248.360252 (cit. on p. 45).

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (cit. on p. 177).

Klinkenberg, Lutz, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler (2020). "Generating Functions for Probabilistic Programs". In: *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*. Vol. 12561. Lecture Notes in Computer Science, pp. 231–248. DOI: 10.1007/978-3-030-68446-4_12 (cit. on pp. 23, 93, 100, 127).

Klinkenberg, Lutz, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen (2024). "Exact Bayesian Inference for Loopy Probabilistic Programs using Generating Functions". In: *Proc. ACM Program. Lang.* 8.OOPSLA1, pp. 923–953. DOI: 10.1145/3649844 (cit. on pp. 26, 30, 33, 36, 93, 108, 109, 121, 122, 127, 142, 180, 187).

Kofnov, Andrey, Marcel Moosbrugger, Miroslav Stankovic, Ezio Bartocci, and Efstathia Bura (2024). "Exact and Approximate Moment Derivation for Probabilistic Loops With Non-Polynomial Assignments". In: *ACM Trans. Model. Comput. Simul.* 34.3, 18:1–18:25. DOI: 10.1145/3641545 (cit. on pp. 46, 188).

Kozen, Dexter (1981). "Semantics of Probabilistic Programs". In: *J. Comput. Syst. Sci.* 22.3, pp. 328–350. DOI: 10.1016/0022-0000(81)90036-2 (cit. on pp. 11, 29, 30, 33).

Kozen, Dexter (1985). "A Probabilistic PDL". In: *J. Comput. Syst. Sci.* 30.2, pp. 162–178. DOI: 10.1016/0022-0000(85)90012-1 (cit. on p. 44).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 1106–1114 (cit. on p. 3).

Kucukelbir, Alp, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei (2017). "Automatic Differentiation Variational Inference". In: *J. Mach. Learn. Res.* 18, 14:1–14:45 (cit. on p. 41).

Kura, Satoshi, Natsuki Urabe, and Ichiro Hasuo (2019). "Tail Probabilities for Randomized Program Runtimes via Martingales for Higher Moments". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. Vol. 11428. Lecture Notes in Computer Science, pp. 135–153. DOI: 10.1007/978-3-030-17465-1_8 (cit. on pp. 47, 188).

Lee, Wonyeol, Xavier Rival, and Hongseok Yang (2023). "Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference". In: *Proc. ACM Program. Lang.* 7.POPL, pp. 335–366. DOI: 10.1145/3571205 (cit. on p. 48).

Lee, Wonyeol, Hangyeol Yu, Xavier Rival, and Hongseok Yang (2020). "Towards verified stochastic variational inference for probabilistic programs". In: *Proc. ACM Program. Lang.* 4.POPL, 16:1–16:33. DOI: 10.1145/3371084 (cit. on p. 48).

Lew, Alexander K., Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka (2020). "Trace types and denotational semantics for sound programmable inference in probabilistic languages". In: *Proc. ACM Program. Lang.* 4.POPL, 19:1–19:32. DOI: 10.1145/3371087 (cit. on pp. 48, 192).

Lew, Alexander K., Mathieu Huot, Sam Staton, and Vikash K. Mansinghka (2023). "ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs". In: *Proc. ACM Program. Lang.* 7.POPL, pp. 121–153. DOI: 10.1145/3571198 (cit. on p. 48).

Liang, Feynman T., Liam Hodgkinson, and Michael W. Mahoney (2023). "A Heavy-Tailed Algebra for Probabilistic Programming". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023* (cit. on p. 14).

Loera, Jesús A. De, Brandon E. Dutra, Matthias Köppe, S. Moreinis, G. Pinto, and J. Wu (2013). "Software for exact integration of polynomials over polyhedra". In: *Comput. Geom.* 46.3, pp. 232–252. DOI: 10.1016/J.COMGEO.2012.09.001 (cit. on p. 80).

Lunn, David, David Spiegelhalter, Andrew Thomas, and Nicky Best (2009). "The BUGS project: Evolution, critique and future directions". In: *Statistics in Medicine* 28.25, pp. 3049–3067. DOI: https://doi.org/10.1002/sim.3680 (cit. on p. 49).

Mak, Carol, Luke Ong, Hugo Paquet, and Dominik Wagner (2021a). "Densities of Almost Surely Terminating Probabilistic Programs are Differentiable Almost Everywhere". In: *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*. Vol. 12648. Lecture Notes in Computer Science, pp. 432–461. DOI: 10.1007/978-3-030-72019-3_16 (cit. on pp. 22, 26, 28, 29, 33, 34, 40, 62, 65).

Mak, Carol, Fabian Zaiser, and Luke Ong (2021b). "Nonparametric Hamiltonian Monte Carlo". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Vol. 139. Proceedings of Machine Learning Research, pp. 7336–7347 (cit. on pp. 16, 28).

Mak, Carol, Fabian Zaiser, and Luke Ong (2022). "Nonparametric Involutive Markov Chain Monte Carlo". In: *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*. Vol. 162. Proceedings of Machine Learning Research, pp. 14802–14859 (cit. on p. 16).

Mangoubi, Oren, Natesh S. Pillai, and Aaron Smith (2018). "Does Hamiltonian Monte Carlo mix faster than a random walk on multimodal densities?" In: *CoRR* abs/1808.03230. arXiv: 1808.03230 (cit. on p. 40).

Manning, Chris and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing* (cit. on p. 9).

Mansinghka, Vikash K., Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin C. Rinard (2018). "Probabilistic programming with programmable inference". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 603–616. DOI: 10.1145/3192366.3192409 (cit. on p. 50).

Mansinghka, Vikash K., Daniel Selsam, and Yura N. Perov (2014). "Venture: a higher-order probabilistic programming platform with programmable inference". In: *CoRR* abs/1404.0099. arXiv: 1404.0099 (cit. on p. 50).

Matsakis, Nicholas D. and Felix S. Klock (2014). "The rust language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pp. 103–104. DOI: 10.1145/2663171.2663188 (cit. on pp. 119, 136, 176).

McCallum, Andrew, Karl Schultz, and Sameer Singh (2009). "FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs". In: *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, pp. 1249–1257 (cit. on p. 49).

McIver, Annabelle and Carroll Morgan (2005). *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. DOI: 10.1007/B138392 (cit. on p. 45).

Metropolis, Nicholas, Arianna W. Rosenbluth, Marshall N. Rosenbluth, and Augusta H. Teller (1953). "Equation of state calculations by fast computing machines". In: *J. Chem. Phys.* 21, pp. 1087–1092. DOI: 10.1063/1.1699114 (cit. on p. 39).

Milch, Brian, Bhaskara Marthi, Stuart Russell, David A. Sontag, Daniel L. Ong, and Andrey Kolobov (2005). "BLOG: Probabilistic Models with Unknown Objects". In: *IJCAI-05,*

Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005, pp. 1352–1359 (cit. on p. 49).

Minka, T., J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill (2018). */Infer.NET 0.3*. Microsoft Research Cambridge. http://dotnet.github.io/infer (cit. on p. 49).

Minka, Thomas P. (2001). "Expectation Propagation for approximate Bayesian inference". In: *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*, pp. 362–369 (cit. on p. 42).

Monniaux, David (2000). "Abstract Interpretation of Probabilistic Semantics". In: *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*. Vol. 1824. Lecture Notes in Computer Science, pp. 322–339. DOI: 10.1007/978-3-540-45099-3_17 (cit. on pp. 45, 90).

Moore, Ramon E. (1966). *Interval Analysis* (cit. on p. 54).

Moosbrugger, Marcel, Miroslav Stankovic, Ezio Bartocci, and Laura Kovács (2022). "This is the moment for probabilistic loops". In: *Proc. ACM Program. Lang.* 6.OOPSLA2, pp. 1497–1525. DOI: 10.1145/3563341 (cit. on pp. 46, 142, 180, 182, 184, 187, 188).

Morgan, Carroll, Annabelle McIver, and Karen Seidel (1996). "Probabilistic Predicate Transformers". In: *ACM Trans. Program. Lang. Syst.* 18.3, pp. 325–353. DOI: 10.1145/229542.229547 (cit. on p. 44).

Narayanan, Praveen, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov (2016). "Probabilistic Inference by Program Transformation in Hakaru (System Description)". In: *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Vol. 9613. Lecture Notes in Computer Science, pp. 62–79. DOI: 10.1007/978-3-319-29604-3_5 (cit. on pp. 36, 121).

Neal, Radford M (2003). "Slice sampling". In: *The annals of statistics* 31.3. DOI: 10.1214/aos/1056562461 (cit. on p. 84).

Neklyudov, Kirill, Max Welling, Evgenii Egorov, and Dmitry P. Vetrov (2020). "Involutive MCMC: a Unifying Framework". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 7273–7282 (cit. on p. 40).

Neuts, Marcel F (1975). "Probability distributions of phase type". In: *Liber Amicorum Prof. Emeritus H. Florin* (cit. on p. 144).

Ng, Andrew Y. and Michael I. Jordan (2001). "On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes". In: *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, pp. 841–848 (cit. on p. 3).

Ngo, Van Chan, Quentin Carbonneaux, and Jan Hoffmann (2018). "Bounded expectations: resource analysis for probabilistic programs". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 496–512. DOI: 10.1145/3192366.3192394 (cit. on p. 47).

Nishimura, Akihiko, David B. Dunson, and Jianfeng Lu (2020). "Discontinuous Hamiltonian Monte Carlo for discrete parameters and discontinuous likelihoods". In: *Biometrika* 107.2, pp. 365–380. DOI: 10.1093/biomet/asz083 (cit. on p. 42).

Nori, Aditya V., Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel (2014). "R2: An Efficient MCMC Sampler for Probabilistic Programs". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pp. 2476–2482. DOI: 10.1609/AAAI.V28I1.9060 (cit. on p. 47).

Ouaknine, Joël and James Worrell (2014). "Positivity Problems for Low-Order Linear Recurrence Sequences". In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete*

Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014, pp. 366–379. DOI: 10.1137/1.9781611973402.27 (cit. on p. 212).

Pearl, Judea (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. ISBN: 1558604790 (cit. on p. 42).

Pfeffer, Avi (2001). "IBAL: A Probabilistic Rational Programming Language". In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pp. 733–740 (cit. on p. 49).

Piccioni, M. (1983). "Continuous versions of Radon-Nikodym derivatives as likelihood ratios". In: *Syst. Control Lett.* 2, pp. 369–374. DOI: 10.1016/0167-6911(83)90047-6 (cit. on pp. 20, 22, 27).

Pierro, Alessandra Di, Chris Hankin, and Herbert Wiklicky (2005). "Probabilistic λ-calculus and Quantitative Program Analysis". In: *J. Log. Comput.* 15.2, pp. 159–179. DOI: 10.1093/LOGCOM/EXI008 (cit. on p. 45).

Pla, Oriol Abril, Virgile Andréani, Colin Carroll, Larry Dong, Christopher Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas V. Wiecki, and Robert Zinkov (2023). "PyMC: a modern, and comprehensive probabilistic programming framework in Python". In: *PeerJ Comput. Sci.* 9, e1516. DOI: 10.7717/PEERJ-CS.1516 (cit. on p. 49).

Pollard, D. (2002). *A User's Guide to Measure-Theoretic Probability*. Cambridge Series in Statistical and Probabilistic Mathematics. DOI: 10.1017/CBO9780511811555 (cit. on pp. 19, 21, 27).

Rabin, Michael O (1980). "Probabilistic algorithm for testing primality". In: *Journal of Number Theory* 12.1, pp. 128–138. DOI: 10.1016/0022-314x(80)90084-0 (cit. on p. 11).

Radul, Alexey and Boris Alexeev (2021). "The Base Measure Problem and its Solution". In: *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*. Vol. 130. Proceedings of Machine Learning Research, pp. 3583–3591 (cit. on p. 27).

Raedt, Luc De and Angelika Kimmig (2015). "Probabilistic (logic) programming concepts". In: *Mach. Learn.* 100.1, pp. 5–47. DOI: 10.1007/S10994-015-5494-Z (cit. on p. 48).

Rainforth, Tom (2017). "Automating inference, learning, and design using probabilistic programming". PhD thesis. University of Oxford, UK (cit. on pp. 6, 8, 36, 39, 40).

Rainforth, Tom, Christian A. Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem van de Meent, Arnaud Doucet, and Frank D. Wood (2016). "Interacting Particle Markov Chain Monte Carlo". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Vol. 48. JMLR Workshop and Conference Proceedings, pp. 2616–2625 (cit. on p. 41).

Ranganath, Rajesh, Sean Gerrish, and David M. Blei (2014). "Black Box Variational Inference". In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*. Vol. 33. JMLR Workshop and Conference Proceedings, pp. 814–822 (cit. on p. 41).

Robert, Christian (2007). *The Bayesian choice*. 2nd ed. Springer Texts in Statistics (cit. on p. 5).

Ronquist, Fredrik, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B. Schön, and David Broman (2021). "Universal probabilistic programming offers a powerful approach to statistical phylogenetics". In: *Communications biology* 4.1. DOI: 10.1038/s42003-021-01753-7 (cit. on p. 10).

Roy, Vivekananda (2020). "Convergence Diagnostics for Markov Chain Monte Carlo". In: *Annual Review of Statistics and Its Application* 7. DOI: 10.1146/annurev-statistics-031219-041300 (cit. on p. 13).

Saad, Feras A., Martin C. Rinard, and Vikash K. Mansinghka (2021). "SPPL: probabilistic programming with fast exact symbolic inference". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pp. 804–819. DOI: 10.1145/3453483.3454078 (cit. on pp. 35, 36, 122, 126, 192, 207).

Salvatier, John, Thomas V. Wiecki, and Christopher Fonnesbeck (2016). "Probabilistic programming in Python using PyMC3". In: *PeerJ Comput. Sci.* 2, e55. DOI: 10.7717/peerj-cs.55 (cit. on pp. 126, 206, 207).

Sankaranarayanan, Sriram, Aleksandar Chakarov, and Sumit Gulwani (2013). "Static analysis for probabilistic programs: inferring whole program properties from finitely many paths". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pp. 447–458. DOI: 10.1145/2491956.2462179 (cit. on pp. 45, 46, 82, 83, 89–91).

Sato, Taisuke and Yoshitaka Kameya (1997). "PRISM: A Language for Symbolic-Statistical Modeling". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pp. 1330–1339 (cit. on p. 49).

Sato, Tetsuya, Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Justin Hsu (2019). "Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, Bayesian inference, and optimization". In: *Proc. ACM Program. Lang.* 3.POPL, 38:1–38:30. DOI: 10.1145/3290351 (cit. on pp. 44, 47).

Schröer, Philipp, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja (2023). "A Deductive Verification Infrastructure for Probabilistic Programs". In: *Proc. ACM Program. Lang.* 7.OOPSLA2, pp. 2052–2082. DOI: 10.1145/3622870 (cit. on p. 45).

Shachter, Ross D., Bruce D'Ambrosio, and Brendan Del Favero (1990). "Symbolic Probabilistic Inference in Belief Networks". In: *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes*, pp. 126–131 (cit. on p. 35).

Shan, Chung-chieh and Norman Ramsey (2017). "Exact Bayesian inference by symbolic disintegration". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 130–144. DOI: 10.1145/3009837.3009852 (cit. on pp. 27, 36).

Smith, Michael J. A. (2008). "Probabilistic Abstract Interpretation of Imperative Programs using Truncated Normal Distributions". In: *Proceedings of the Sixth Workshop on Quantitative Aspects of Programming Languages, QAPL 2008, Budapest, Hungary, March 29-30, 2008*. Vol. 220. Electronic Notes in Theoretical Computer Science 3, pp. 43–59. DOI: 10.1016/J.ENTCS.2008.11.018 (cit. on pp. 45, 90, 192).

Staton, Sam (2017). "Commutative Semantics for Probabilistic Programming". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Vol. 10201. Lecture Notes in Computer Science, pp. 855–879. DOI: 10.1007/978-3-662-54434-1_32 (cit. on p. 27).

Stein, Dario and Sam Staton (2021). "Compositional Semantics for Probabilistic Programs with Exact Conditioning". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470552 (cit. on p. 27).

Szymczak, Marcin and Joost-Pieter Katoen (2019). "Weakest Preexpectation Semantics for Bayesian Inference - Conditioning, Continuous Distributions and Divergence". In: *Engineering Trustworthy Software Systems - 5th International School, SETSS 2019, Chongqing, China, April 21-27, 2019, Tutorial Lectures*. Vol. 12154. Lecture Notes in Computer Science, pp. 44–121. DOI: 10.1007/978-3-030-55089-9_3 (cit. on p. 45).

Talts, Sean, Michael Betancourta, Daniel Simpson, Aki Vehtari, and Andrew Gelman (2018). "Validating Bayesian Inference Algorithms with Simulation-Based Calibration". In: *arXiv* 1804.06788. DOI: 10.48550/arXiv.1804.06788 (cit. on pp. 43, 87, 88).

Tassarotti, Joseph and Jean-Baptiste Tristan (2023). "Verified Density Compilation for a Probabilistic Programming Language". In: *Proc. ACM Program. Lang.* 7.PLDI, pp. 615–637. DOI: 10.1145/3591245 (cit. on pp. 47, 191).

Tolpin, David, Jan-Willem van de Meent, and Frank D. Wood (2015). "Probabilistic Programming in Anglican". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*. Vol. 9286. Lecture Notes in Computer Science, pp. 308–311. DOI: 10.1007/978-3-319-23461-8_36 (cit. on pp. 53, 83, 122).

Tran, Dustin, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei (2017). "Deep Probabilistic Programming". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (cit. on p. 50).

Vákár, Matthijs, Ohad Kammar, and Sam Staton (2019). "A domain theory for statistical probabilistic programming". In: *Proc. ACM Program. Lang.* 3.POPL, 36:1–36:29. DOI: 10.1145/3290349 (cit. on pp. 8, 33).

Van de Meent, Jan-Willem, Brooks Paige, Hongseok Yang, and Frank Wood (2018). "An Introduction to Probabilistic Programming". In: *CoRR* abs/1809.10756. arXiv: 1809.10756 (cit. on p. 6).

Van der Vaart, A. W. (1998). *Asymptotic statistics*. Vol. 3. Camb. Ser. Stat. Probab. Math. DOI: 10.1017/CBO9780511802256 (cit. on p. 4).

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008 (cit. on p. 3).

Wächter, Andreas and Lorenz T. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program.* 106.1, pp. 25–57. DOI: 10.1007/S10107-004-0559-Y (cit. on p. 177).

Wang, Di, Jan Hoffmann, and Thomas W. Reps (2018). "PMAF: an algebraic framework for static analysis of probabilistic programs". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 513–528. DOI: 10.1145/3192366.3192408 (cit. on p. 91).

Wang, Di, Jan Hoffmann, and Thomas W. Reps (2021). "Central moment analysis for cost accumulators in probabilistic programs". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pp. 559–573. DOI: 10.1145/3453483.3454062 (cit. on pp. 47, 188).

Wang, Peixin, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin, and Wenjun Shi (2019). "Cost analysis of nondeterministic probabilistic programs". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, pp. 204–220. DOI: 10.1145/3314221.3314581 (cit. on pp. 47, 188).

Wang, Peixin, Tengshun Yang, Hongfei Fu, Guanyan Li, and Luke Ong (2024). "Static Posterior Inference of Bayesian Probabilistic Programming via Polynomial Solving". In: *Proc. ACM Program. Lang.* 8.PLDI, pp. 1361–1386. DOI: 10.1145/3656432 (cit. on pp. 8, 14, 88, 138, 142, 187).

Wingate, David, Andreas Stuhlmüller, and Noah D. Goodman (2011). "Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*. Vol. 15. JMLR Proceedings, pp. 770–778 (cit. on p. 40).

Winn, John M. and Christopher M. Bishop (2005). "Variational Message Passing". In: *J. Mach. Learn. Res.* 6, pp. 661–694 (cit. on p. 42).

Winner, Kevin and Daniel Sheldon (2016). "Probabilistic Inference with Generating Functions for Poisson Latent Variable Models". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 2640–2648 (cit. on pp. 93, 94, 100, 122, 124–127, 205, 206).

Winner, Kevin, Debora Sujono, and Daniel Sheldon (2017). "Exact Inference for Integer Latent-Variable Models". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Vol. 70. Proceedings of Machine Learning Research, pp. 3761–3770 (cit. on pp. 93, 109, 115, 124, 125, 127).

Wood, Frank D., Jan-Willem van de Meent, and Vikash Mansinghka (2014). "A New Approach to Probabilistic Programming Inference". In: *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*. Vol. 33. JMLR Workshop and Conference Proceedings, pp. 1024–1032 (cit. on p. 50).

Wu, Yi, Siddharth Srivastava, Nicholas Hay, Simon S. Du, and Stuart Russell (2018). "Discrete-Continuous Mixtures in Probabilistic Programming: Generalized Semantics and Inference Algorithms". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Vol. 80. Proceedings of Machine Learning Research, pp. 5339–5348 (cit. on p. 27).

Yao, Yuling, Aki Vehtari, and Andrew Gelman (2022). "Stacking for Non-mixing Bayesian Computations: The Curse and Blessing of Multimodal Posteriors". In: *J. Mach. Learn. Res.* 23, 79:1–79:45 (cit. on p. 40).

Zaiser, Fabian (2024). *Artifact for: Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops (POPL 2025)*. Version 1.0. DOI: 10.5281/zenodo.14169507 (cit. on pp. 136, 138, 176, 179).

Zaiser, Fabian, Andrzej S. Murawski, and Luke Ong (2023). "Exact Bayesian Inference on Discrete Models via Probability Generating Functions: A Probabilistic Programming Approach". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023* (cit. on pp. 15, 22, 23, 29, 93, 112, 121).

Zaiser, Fabian, Andrzej S. Murawski, and Luke Ong (2024). *Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops*. arXiv: 2411.10393 [cs.PL] (cit. on pp. 129, 141).

Zaiser, Fabian, Andrzej S. Murawski, and Luke Ong (2025). "Guaranteed Bounds on Posterior Distributions of Discrete Probabilistic Programs with Loops". In: *Proc. ACM Program. Lang.* 9.POPL. DOI: 10.1145/3704874 (cit. on pp. 15, 22, 129, 141).

Zaiser, Fabian and Luke Ong (2020). "The Extended Theory of Trees and Algebraic (Co)datatypes". In: *Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and Synthesis, VPT/HCVS@ETAPS 2020, Dublin, Ireland, 25-26th April 2020*. Vol. 320. EPTCS, pp. 167–196. DOI: 10.4204/EPTCS.320.14 (cit. on p. 15).

Zhang, Cheng, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt (2019). "Advances in Variational Inference". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 41.8, pp. 2008–2026. DOI: 10.1109/TPAMI.2018.2889774 (cit. on p. 41).

Zhang, Honghua, Brendan Juba, and Guy Van den Broeck (2021). "Probabilistic Generating Circuits". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Vol. 139. Proceedings of Machine Learning Research, pp. 12447–12457 (cit. on p. 126).

Zhang, Nevin L. and David Poole (1994). "A simple approach to Bayesian network computations". In: *Proc. of the Tenth Canadian Conference on Artificial Intelligence* (cit. on p. 41).

Zhou, Yuan, Bradley J. Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood (2019). "LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models". In: *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*. Vol. 89. Proceedings of Machine Learning Research, pp. 148–157 (cit. on pp. 42, 84).

Zhou, Yuan, Hongseok Yang, Yee Whye Teh, and Tom Rainforth (2020). "Divide, Conquer, and Combine: a New Inference Strategy for Probabilistic Programs with Stochastic Support". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research, pp. 11534–11545 (cit. on p. 42).

Index

ω-complete partial order (ω-cpo), 30
ω-continuous, 33
c-contraction invariant, 144

absolutely continuous, 20
abstract domain, 45
abstract interpretation, 45
almost disjoint, 57
almost everywhere (a.e.), 19
almost never, 19
almost surely (a.s.), 19
almost surely terminating (AST), 32, 34
Anglican, 50
approximate inference, 12, 36
arity, 27
automatic differentiation (AD), 41, 109

Bayes' rule, 4, 21
Bayesian network, 7
Bayesian paradigm, 4
Bayesian probabilistic programming, 7
Bayesian statistical probabilistic programming, 8
BLOG, 49
Borel σ-algebra, 19
bottom element, 55
box, 55, 80
boxwise continuous, 66
BUGS, 49

call-by-value, 33
central moment, 20
Church, 49
closed form, 97
compatible, 57
compound distribution, 24
conditional density, 21
conditional probability, 21
configuration, 33
consistent, 12
continuous distribution, 20
contraction invariant, 144
counting measure, 19
cumulative distribution function (CDF), 20
curse of dimensionality, 12

decay rate, 144
denotational semantics, 29
density, 20
Diabolo, 136, 176
Dice, 35
discrete distribution, 20
discrete SGCL, 26
discriminative model, 3
disintegration, 21, 27
distribution, 19
distribution transformer, 29

Edward, 50
effective sample size (ESS), 42
evaluation context, 33
event, 19, 23
eventually geometric distribution (EGD), 144
evidence, 4
evidence lower bound (ELBO), 41
exact inference, 11
exhaustive, 58
expansion, 147
expected value, 20
extended GF semantics, 137
extended state space, 30

factorial moment, 97
factorial moment generating function, 96
Factorie, 49
failure state, 30
frequentist, 3

Gelman-Rubin statistic, 42
Gen, 50
generating function (GF), 96
generating function semantics, 98
generative model, 3
Genfer, 35, 94, 118
geometric bound semantics, 142, 149
guaranteed bound, 13
GuBPI, 54

Hakaru, 36
Hamiltonian Monte Carlo (HMC), 39
hard conditioning, 25

IBAL, 49
importance sampling (IS), 38
importance weight, 38
independent, 20
Infer.NET, 49
infinite trace, 58
initial block, 144
integrable, 35
interacting particle MCMC, 41
interval, 55
interval separable, 67
interval SPCF, 56

interval term, 56
interval trace, 55
inverse cumulative distribution function (ICDF), 20
involutive MCMC (iMCMC), 40

join, 55, 152
joint distribution, 19

Kullback-Leibler divergence, 41
kurtosis, 20

Las Vegas algorithm, 11
latent variable, 37
lattice, 55
Lebesgue integral, 19
Lebesgue measure, 19
lightweight MH (LMH), 40
likelihood, 4
lower bound semantics, 131

machine learning, 2
marginal density, 21
marginal distribution, 20
marginal likelihood, 4
marginalization, 97
marginalize out, 19
Markov chain, 39
Markov chain Monte Carlo (MCMC), 39
Markov kernel, 21
mass function, 20
mean, 20
measurable function, 19
measurable space, 19
measure, 19
measure kernel, 21
measure space, 19
meet, 55
message passing, 41
Metropolis Hastings (MH), 39
mode, 20
moment, 20
Monte Carlo algorithm, 11
monus, 23
multimodal, 40
multivariate Taylor polynomial, 110

nested inference, 8
No-U-Turn Sampler (NUTS), 39
non-statistical probabilistic programming, 8
normalize, 32
normalized distribution, 35
normalizing constant, 34
null set, 19

operational semantics, 29

particle, 40
particle filtering, 40
particle Gibbs (PGibbs), 41
particle MCMC (PMCMC), 41
posterior, 4
posterior distribution, 35
potential scale reduction factor, 42
power series, 96
prior, 4
PRISM, 49
probabilistic graphical model (PGM), 6
probabilistic guarded command language (pGCL), 23, 44
probabilistic programming, 7, 22
probabilistic programming language (PPL), 8, 22
probabilistic programming system (PPS), 8
probability, 19
probability density function (PDF), 20
probability distribution, 19
probability generating function (GF), 93
probability kernel, 21
probability mass function (PMF), 20
probability measure, 19
probability theory, 2
ProbLog, 49
Prodigy, 36
product σ-algebra, 19
product measure, 19
program, 23
PSI, 36
pushforward measure, 19
PyMC, 49
Pyro, 50

quantile function, 20

Radon-Nikodym derivative, 20
random variable (RV), 19
random-walk Metropolis Hastings (RMH), 39
randomized programming, 8
raw moment, 20
reduction, 33
refinement, 56
regular conditional probability distribution, 21
rejection sampling, 37
residual mass, 133
residual mass semantics, 130, 134
restriction of a measure, 19

sample space, 19
sample variable, 63
sequential Monte Carlo (SMC), 40
simplified standard semantics, 32
simulation-based calibration (SBC), 43
single-site Metropolis Hastings, 40

skewness, 20
soft conditioning, 25, 27
SPPL, 35
Stan, 49
standard deviation, 20
standard measure, 19
standard semantics, 30
standardized moment, 20
state space, 29
statistical GCL (SGCL), 23
statistical inference, 2
statistical PCF (SPCF), 26
statistical probabilistic programming, 8
statistics, 2
stochastic gradient descent (SGD), 41
stochastic variational inference (SVI), 41
strict join, 152
subadditive measure, 59
submersion, 194
subprobability distribution, 19
superadditive measure, 57
support, 20
symbolic configuration, 63
symbolic constraint, 63
symbolic interval configuration, 78
symbolic interval path, 78
symbolic interval term, 78
symbolic interval value, 78
symbolic path, 64
symbolic term, 63
symbolic value, 63

tail, 20
tail asymptotics, 20
Taylor coefficient, 97
Taylor polynomial, 110
tensor, 17
term, 27
terminating, 33
tight subset, 67
total mass, 19
total measure, 19
total variation distance (TVD), 20
trace, 33
trace space, 33
transformer semantics, 29
Turing, 50

universal probabilistic programming language, 22
unnormalized distribution, 34
unrolling, 130

value, 27, 33
variable elimination, 41
variance, 20
variational inference (VI), 41
Venture, 50
volume, 57

WebPPL, 50
weight, 33
weighted sample, 38
widening, 77, 204