Automatic Integration and Differentiation of Probabilistic Programs

by

Alexander K. Lew

B.S., Yale University, 2015
S.M., Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Alexander K. Lew
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Vikash K. Mansinghka
Principal Research Scientist, Department of Brain and Cognitive Sciences
Thesis Supervisor

Certified by: Joshua B. Tenenbaum
Professor of Brain and Cognitive Sciences
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Theses

Automatic Integration and Differentiation of Probabilistic Programs

by

Alexander K. Lew

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis addresses the challenge of automating fundamental operations from probability theory and calculus on probability distributions defined by higher-order probabilistic programs. It does this by developing a suite of composable program transformations for an expressive core calculus for probabilistic programming:

- Integration: Compiling a probabilistic program into a deterministic representation of its expectation operator, handling potentially intractable integrals symbolically.
- Unbiased estimation: Transforming programs involving intractable operations (like integration) into runnable probabilistic programs that yield provably unbiased estimates of the original value, with flexible levers for users to navigate cost-variance trade-offs.
- Radon-Nikodym differentiation: Compiling probabilistic programs into implementations of a novel interface for the unbiased estimation of density ratios, of the sort that arise in Monte Carlo and variational inference.
- Differentiation: Extending automatic differentiation (AD) to compose with the above transformations, enabling the optimization of expected values and density ratios of probabilistic programs.

These transformations operate on an expressive higher-order probabilistic programming language and are proven correct using denotational semantics and logical relations. The resulting framework enables the sound and automated implementation of a wide range of algorithms for probabilistic inference and learning.

To demonstrate the practical value of these techniques, we use them to implement three systems for scalable probabilistic inference in different domains: (1) extensions to the Gen probabilistic programming system that accelerate and automate a broad range of Monte Carlo and variational inference algorithms, (2) the PClean system for automated Bayesian reasoning about relational data, and (3) the GenLM system for controllable generation from language models. We find that our techniques enable these systems to scale to a variety of complex, real-world problems, and to achieve state-of-the-art performance on a range of benchmarks.

Thesis Supervisor: Vikash K. Mansinghka
Title: Principal Research Scientist, Department of Brain and Cognitive Sciences

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor of Brain and Cognitive Sciences

Acknowledgments

Vikash: I still remember our first Skype call. I was preparing to be grilled on my (thin) research record, but the first thing you asked was whether I had read Impro, by Keith Johnstone—which somehow led to a three-hour conversation about improvisation, pedagogy, graduate school, probability, and programming. As I suspected might be the case after that call, I've been very lucky to have you as a mentor. The scope and depth and courage of your vision for the field has shaped almost every aspect of my PhD. Thank you for pushing me when I needed pushes, and supporting me when I needed support. I'm grateful to have gone on this journey in your lab, and that I still get to come to you mentorship, hopefully for many years to come.

Josh: Thank you for bringing me into the sprawling, loving, inspiring CoCoSci family that you have created and nurtured over the years. The connection your work draws between probability and programming and cognition has inspired (and continues to inspire) all the research I do. I have learned so much from working with you, and I'm excited to continue learning from you in the years ahead.

I have also been fortunate to have many mentors and teachers beyond my advisors: Armando Solar-Lezama, who has been a source of both fun, high-bandwidth technical collaboration and thoughtful personal advice; Martin Rinard, whose suggestions have improved much of the work in this thesis and whose aphorisms (e.g., "in a PL paper, you build an atom bomb to kill a chicken") always have a point; Mike Carbin, who showed me how to write a PL paper (without harming any chickens); Tim O'Donnell, who has made the gigantic collaboration that is GenLM one of my most rewarding academic experiences; Laura Schulz, whose classes I always looked forward to (and to whom I apologize for every acronym in this thesis); Sam Staton and Ohad Kammar, whose work laid the theoretical foundations on which this thesis builds, and who have patiently answered countless beginner's questions about quasi-Borel spaces; and so many more (Jacob Andreas, Ryan Cotterell, Tim Vieira, Jason Eisner, Tamara Broderick, Steven Holtzen, Stuart Russell, Polina Golland—and before graduate school, the people who introduced me to programming languages and to Bayes: Larry Carin, Zhong Shao, David Carlson).

I was lucky to begin graduate school in a lab stacked with brilliant colleagues. Marco: thank you for teaching me most of what I know about Monte Carlo inference—and a whole lot of what I know about how to do research in general. Feras: the exceptional standard of rigor and excellence to which you hold your research was and continues to be an inspiration. Jar: thank you for being my first collaborator—many of the ideas in this thesis were first conceived playing with your Metaprob prototype. Cameron's feedback and mentorship have sharpened many of the ideas in this thesis. Many students from other labs also took me under their wings, teaching me all sorts of important lessons: the Zenna, Ben Sherman, Jon, MH, Luke, Maddie, the program inductors (Kevin Ellis, Max Nye, Evan Pu). Of course, I have also learned so much from the people who came to MIT after me. Xuan, Mathieu,

Tiffany, João, Nishad, McCoy, George, Matin, Xiaoyan, Yoni, Fabian, Ria, Eric, Arijit, Karen, Mau, Elias: it has been a pleasure to work with all of you. To Rif and the Googlers (Tuan-Anh, Matt, Dieterich, Emily, Thomas, Srinivas): thank you for being so generous with your time and expertise. Gaurav: it has been a joy getting to learn from you. And I've been incredibly grateful to add many more collaborators through GenLM: Gabe, Ben LeBrun, Ben Lipkin, Jacob Hoover, and the many at ETH—I'm so excited we get to keep working together!

To Amanda, Rachel, Lisa, Julian, Lucy, and the rest of the admin team: thank you for making life a million times easier than it might have been. Every time I ever heard anyone complain about the bureaucracy of grad school, I privately thanked you all that I had no idea what they were on about.

To my closest collaborators who I have been lucky to count also as among my closest friends: thank you for making these last seven years as fun and rewarding as they were. Lio: if a thought enters my head, and I don't text it to you, has it been thought at all? Monica: thanks for being there if I am ever in a PICKLE DB. Maddy: [high-pitched wailing sound]. Xuan: your future students don't know how lucky they are (admittedly, they don't know they're your future students yet). Tyler: I'm so excited I get to start the next chapter alongside you at Yale.

So many friends have also made my time in Cambridge special. The original River St. crew: Mike and Christine, Eric, Jess, Kavi, Brian, JMT, Lee. TV club: David, Sterling, Aaron. Ex!t: Zoe, Devin, Ava, Jacob. SUN: Avani, Sal, Ella, Jesús. The long room: Yoni, Tracey, Tony, Kartik, Sam, Brian, Alicia, Gabe. The Solving AI cohort: João, Tom, Sam, Lio. Kevin: I think of you every time I write $k\mu$ (which is a lot). George and Victor, Max and Miguel, Wen-Wen and Tony, Alena: I'm so lucky Monica and Lio invited me to that first BLV party (where Max lied about his babka).

Finally, to my family. Mom, Dad, Joey, Papa Arieh, Granny Clara, Teresa, Papa Myra, Papa George, Dina, Lilly, Elena, Ben, Will: I feel so fortunate to have gotten to grow up surrounded by so much love—and by so many great models of what it looks like to live with genuine curiosity about everything from cell biology to poetry to Spanish literature, and to do it all with a sense of joy.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

1.1 Probabilistic computation

At first glance, it may seem counterintuitive that we would want a computation to be probabilistic. We typically want our computers to be logical, systematic, reliable—not random and capricious. But probability is not just a theory of gambling: it is also a theory of rationality, explaining how to derive reasonable conclusions from multiple imperfect sources of information. A broad range of tasks that we routinely ask our computers to perform—from high-level tasks like transcribing speech into text, recommending a new TV show, or estimating the traffic on the way to work, to low-level tasks like CPU branch prediction, selectivity estimation in databases, or robust touch sensing on smartphones—are inherently probabilistic, in two senses:

- First, the **specifications** of many tasks are probabilistic, rather than logical. That is, the correctness of software that performs such tasks can only be established with respect to some probabilistic model. For example, a branch predictor is accurate or inaccurate relative to some probability distribution over programs and data that are processed by the CPU. Similarly, software that maps anonymous GPS pings from millions of smartphones to real-time estimates of traffic conditions is correct or incorrect relative to some probabilistic model over traffic conditions and smartphone usage.

- Second, the **methods** we use to solve these tasks are often also probabilistic. This does not mean that the final software artifact is stochastic, though in many settings, it is. (Consider the interface that most leading AI companies expose to their users: a prominent *Retry* button that allows us to draw multiple, independently sampled answers until we are satisfied with the result.) Rather, during the development of the software (if not at runtime), it is often necessary to compute or estimate probabilistic quantities, such as expected values, event probabilities, gradients of

densities, and so on. For example, modern reinforcement learning methods for large language models require the stochastic estimation of gradients of expected reward signals, as well as of a particular divergence between distributions, which serves as a regularizer during training [Shao et al., 2024]. As another example, Waymo uses sophisticated importance sampling methods to integrate billions of miles of simulated driving data with millions of miles of real-world driving data to estimate the probable rate of crashes; the estimates must be acceptably low before software updates are deployed on the road [Terres et al., 2023].

Despite the ubiquity of inherently probabilistic tasks, the programming languages that we use to specify and solve them generally do not let us express probabilistic specifications or methods at a high level of abstraction. Rather, the probabilistic thinking that goes into solving these tasks evolves independently from the code, in practitioners' notebooks, in research papers, and in the internal documentation of industry software. Conceptually small changes to a probabilistic model or algorithm may require laborious, error-prone pen-and-paper derivations and multiple, non-local code changes, slowing the pace of iteration.[1]

This thesis advances a particular hypothesis for what *higher-level* languages for probabilistic computation could look like. It builds on decades of work in **probabilistic programming**, a programming paradigm in which probability distributions are encoded as programs that make random choices during their execution. To this literature, it contributes new methods for automating fundamental operations on probability distributions, when they are encoded as programs. I argue that these advances open the way to a new architecture for probabilistic programming systems that could make them more generally useful for designing and implementing probabilistic software.

1.2 Programming with probability distributions

A high-level programming language for probabilistic computation must have in the first place a way of representing and working with probability distributions. In mathematics, probability distributions have a wide variety of representations: density functions, cumulative distribution functions, generating functions, measures, random variables, Bayesian networks, factor graphs, random fields, energy-based models, and so on. A central idea in probabilistic programming is that probability distributions can also be represented as *programs that make random choices* [Koller

[1] The breakneck pace of new developments in machine learning circa 2025 may seem to belie this claim. But a key contributor to this rapid rate of progress has been the development of high-level tools for specifying differentiable loss functions and automating the math of optimizing them [Paszke et al., 2017, e.g.]. Techniques that rely only on the standard deep learning toolkit have benefited the most from these tools. My hope is that new tools that provide high-level automation for a broader set of mathematical operations, from both calculus and probability theory, will spur research into a broader range of techniques, which may help to address some of the key drawbacks of pure deep learning. My collaborators and I have begun to explore some of these directions, which are discussed in detail in Part III of this thesis.

Probabilistic program

$p = \lambda\theta.$ **do**
  $b \leftarrow$ **flip**$(1-\theta)$
  if b **then**
    return 0
  else
    return $(\theta \div 2)$

Conditional probability distribution

$$p(x;\theta) = \begin{cases} 1-\theta & \text{if } x = 0 \\ \theta & \text{if } \theta > 0 \text{ and } x = \frac{\theta}{2} \\ 0 & \text{otherwise} \end{cases}$$

Figure 1-1: A program that makes random choices during its execution implicitly defines a conditional probability distribution: given the program's inputs, what is the distribution over its possible outputs?

et al., 1997, Milch et al., 2004, Goodman et al., 2008b, Mansinghka et al., 2009b], and that this representation is both a natural and a powerful one.

To understand the sense in which this is meant, consider the example in Fig. 1-1. On the left is a program. It takes a number θ as input, then flips a coin, with probability $1-\theta$ of heads. If the coin flip succeeds, the program returns 0. Otherwise, it returns $\frac{\theta}{2}$. We can view this program as a recipe for generating outputs, given inputs. But we can equivalently view it as an unambiguous specification of a particular conditional probability distribution, shown on the right of the figure. This is the distribution $p(x;\theta)$ that it induces on its output x, given its input θ.

If our programming language is sufficiently expressive, this representation can subsume many others. Table 1.1 gives several examples: with sequential imperative programs, we can represent Bayesian networks; with **if** statements, we can represent mixture models; with recursion, we can represent Markov chains, probabilistic grammars, and state space models; and with higher-order functions, we can represent non-parametric models and random measures [Roy et al., 2008, Dash et al., 2023]. Furthermore, many distributions that would be too cumbersome to work with using these existing representations become manageable with the machinery that programming languages provide for abstraction, composition, and the management of complexity. For example, Baydin et al. [2019] connect a stochastic simulator of particle collisions in high-energy colliders, originally implemented in a million lines of C++, to a probabilistic programming backend, enabling them to use it as a probabilistic model in which to perform inference.

The representational power of probabilistic programs is sometimes assumed to come at a cost, diminishing the effectiveness of the algorithms available for analyzing and manipulating them. But this need not be the case, and in fact, probabilistic programs have many advantages beyond their expressive power:

- First, just as Bayesian networks expose more structure in a distribution than does a simple joint probability table, the source code of a probabilistic program exposes a lot of structure that is not necessarily apparent in graphical model or density function representations. This structure can be exploited for more

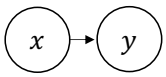
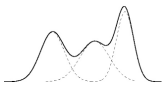
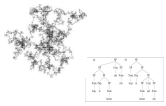
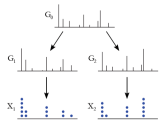
Class of distribution		Programming language feature	
	Bayesian networks	*Sequencing*	$x \leftarrow \textbf{normal}(0,1)$ $y \leftarrow \textbf{normal}(x,1)$
	Mixture models	*Conditionals*	$b \leftarrow \textbf{flip}(p)$ $x \leftarrow \textbf{if } b \textbf{ then}$ $\quad \textbf{normal}(0,1)$ $\quad \textbf{else}$ $\quad \textbf{normal}(4,2)$
	Markov chains State-space models Probabilistic grammars	*Recursion*	$walk = \lambda x.$ $\quad \textbf{if } x > 0 \textbf{ then do}$ $\quad\quad x' \leftarrow \textbf{normal}(x,1)$ $\quad\quad walk(x')$ $\quad \textbf{else}$ $\quad\quad \textbf{return } x$
	Random measures Non-parametric models	*Higher-order functions, laziness, memoization*	$dp = \lambda(\alpha, G_0).\ \textbf{do}$ $\quad sticks \leftarrow \textbf{mem}(\lambda i.\textbf{beta}(1,\alpha))$ $\quad atoms \leftarrow \textbf{mem}(\lambda i.G_0)$ $\quad \textbf{return } (\textbf{do}$ $\quad\quad u \leftarrow \textbf{uniform}(0,1)$ $\quad\quad \textbf{let } (i,\_) = \textbf{until}$ $\quad\quad\quad (\lambda(i,u).u < sticks(i))$ $\quad\quad\quad (\lambda(i,u).(i+1, \frac{u-sticks(i)}{1-sticks(i)}))$ $\quad\quad\quad (0,u)$ $\quad\quad \textbf{return } (atoms(i)))$

Table 1.1: Programming language features and example classes of probability distribution that each feature can be used to express.

powerful automation by probabilistic program compilers.

- Second, as a representation, probabilistic programs are *closed* under many operations that cannot be treated as first-class in other representations. For example, a Monte Carlo gradient estimator for a stochastic computation graph [Schulman et al., 2015] may not itself be a stochastic computation graph, but a gradient estimator for a probabilistic program is always itself a probabilistic program.

- Third, just as the study of graph theory and graph algorithms led to algorithmic advances for graphical representations of models, an extensive arsenal of techniques from the programming languages community—including program analyses, program transformations, synthesis algorithms, and optimization techniques—can

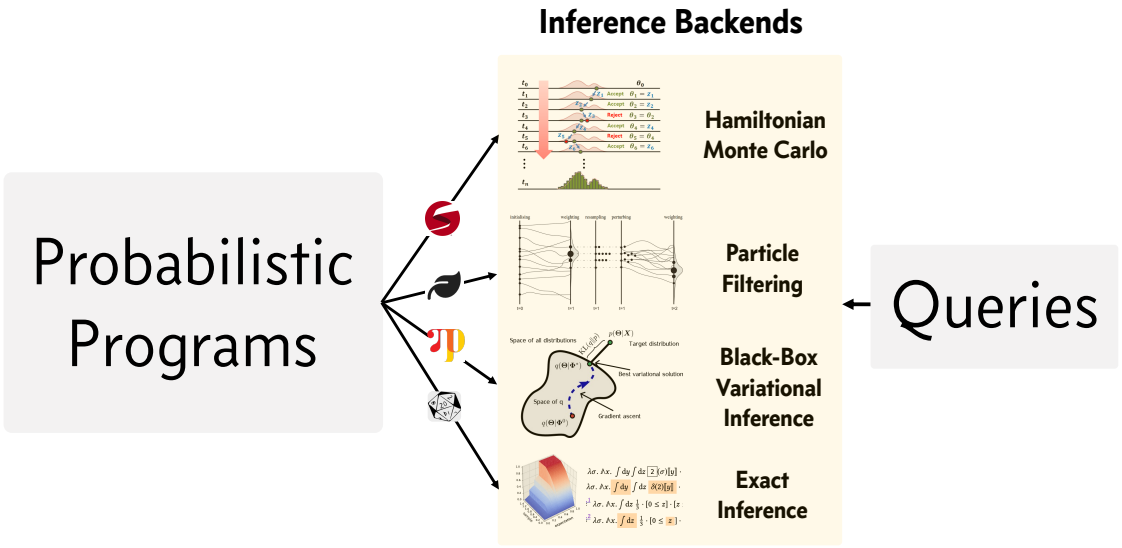


Figure 1-2: **A standard architecture for probabilistic programming systems.** In the standard architecture for PPLs, a *model* (specified as a probabilistic program) and a *query* (e.g., to find the parameters that maximize the likelihood of some dataset) are fed to an *inference engine*, which returns an answer.

be brought to bear on probabilistic programs.

Next, we will see how these properties can be exploited to develop new forms of automation for working with probabilistic programs.

1.3 Automatic integration and differentiation of probabilistic programs

Given a probabilistic program, what can we do with it? A standard architecture for a probabilistic programming system is illustrated in Fig. 1-2. In this architecture, the user provides a single probabilistic program (the *model*) and one or more *queries* (e.g., to find the parameters of the model distribution that maximize the likelihood of some dataset). An *inference engine* then uses one of a handful of supported algorithms to compute or estimate answers to the user's queries. In practice, different systems support different algorithms, and the modeling language is typically designed with a particular approach to inference in mind.

This thesis proposes a different architecture, depicted in Fig. 1-3. In this architecture, we augment an expressive core language for probabilistic programming with a set of *composable program transformations* that automate key operations on probability distributions. Our approach is inspired by the success of systems like TensorFlow and PyTorch, which provide users not with a black-box optimization engine, but with automation for computing the *gradients* of (deterministic) models encoded as programs. Relative to those systems, this thesis can be understood as a proposal to

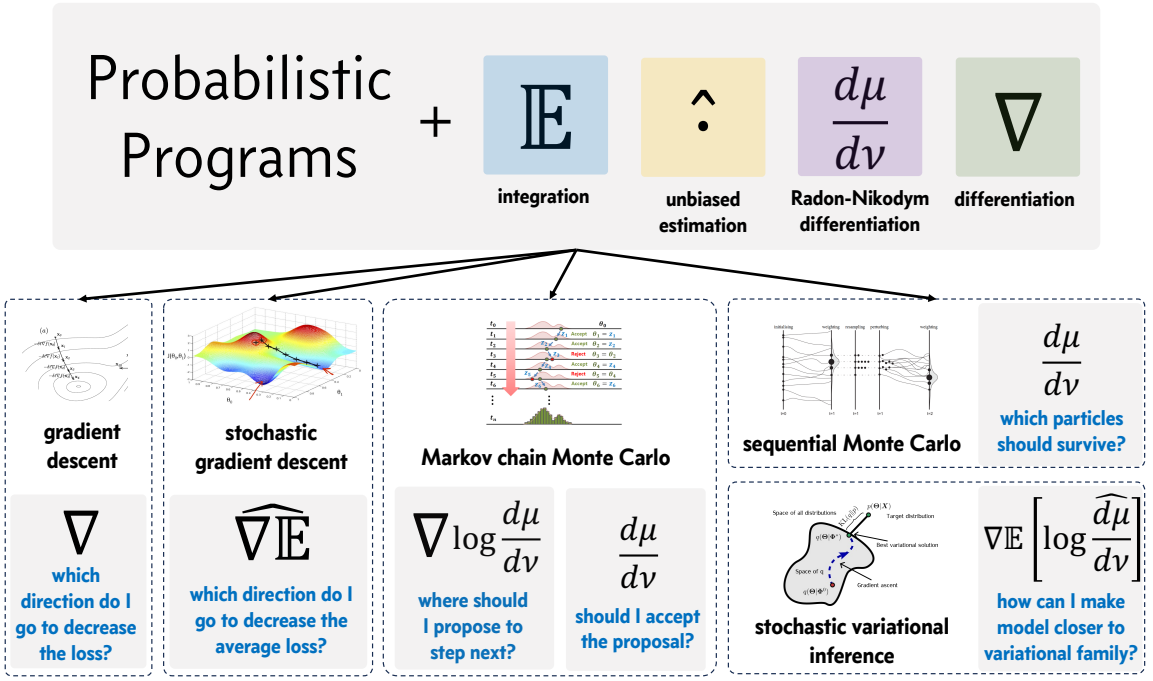


Figure 1-3: **The architecture proposed by this thesis.** We extend our language of probabilistic programs with constructs for key operations from calculus and probability theory: integration with respect to a distribution, unbiased estimation of intractable quantities, Radon-Nikodym differentiation, and standard differentiation.

both *expand the class of supported models*, to include recursive, higher-order programs that freely combine neural networks, symbolic control flow, and probabilistic computation, and *extend the automation*, to support not just gradients, but also key operations from probability theory, such as integrals and probability density ratios.

These operations can be composed to concisely and soundly implement an open-ended set of algorithms for probabilistic inference, prediction, and learning. Just as gradients answer the question *which direction in parameter space should we step to decrease the loss?* in hill-climbing optimization algorithms, this broader set of operations answers various questions that arise in the probabilistic setting. For example, Radon-Nikodym derivatives are used to decide whether to accept proposed samples in Markov chain Monte Carlo methods, as well as to identify promising particles in importance sampling and sequential Monte Carlo. As another example, gradients of integrals of log Radon-Nikodym derivatives are used to optimize the parameters of posterior approximations in variational Bayesian inference. Although these operations are typically intractable to compute exactly, all of these algorithms can make do with cheaper estimates, motivating another key aspect of our design: program transformations that automatically derive provably unbiased estimators for intractable quantities, with levers for controlling the cost-variance trade-off.

1.4 Application to scalable probabilistic inference

To demonstrate the practical value of these automated operations, the thesis develops three systems for scalable probabilistic inference in different domains:

- Extensions to the Gen probabilistic programming system [Cusumano-Towner et al., 2019b] that accelerate and automate a broad range of Monte Carlo and variational inference algorithms, handling the case when models, variational families, or proposals do not necessarily admit tractable densities, and when variational objectives do not necessarily admit tractable gradients (Chapter 6).
- PClean, a domain-specific probabilistic programming system for reasoning about and cleaning real-world tabular datasets of up to millions of rows, with an automated sequential Monte Carlo algorithm based on automated Radon-Nikodym derivatives (Chapter 7).
- GenLM, a system for controllable generation from language models, which uses estimated Radon-Nikodym derivatives to guide generation from a language model toward completions that satisfy user-specified constraints (Chapter 8).

In extensive empirical evaluations, we find that our techniques enable these systems to scale to a variety of complex, real-world problems, and to achieve state-of-the-art performance on a range of benchmarks.

1.5 Outline of the thesis

This thesis is structured as follows:

- **Part I: Preliminaries** lays the groundwork for the rest of the thesis.
  - **Chapter 2 (A Higher-Order Probabilistic Programming Language)** introduces the syntax and semantics of a higher-order probabilistic programming language. This language serves as the core calculus on which the program transformations developed in subsequent chapters operate. We pay especially close attention to developing its denotational semantics. In the rest of the thesis, our program transformations will be proven correct with respect to this semantics. Although our semantics features several minor departures from prior work on the theory of PPLs, for the most part, this chapter draws heavily on the existing literature, and is included to provide a thorough and self-contained reference for the notation and concepts used in the rest of the thesis.
- **Part II: Transforming Probabilistic Programs** describes the key technical contributions of the thesis: program transformations that automate the integration and differentiation of probabilistic programs.
  - **Chapter 3 (Integrals and Unbiased Estimates)** presents a framework for automatically deriving provably unbiased estimators for integrals and functions of integrals with respect to measures defined by probabilistic programs. It defines

and proves correct two key program transformations: one that compiles a probabilistic program into a higher-order deterministic program representing its *expectation operator*, and another that transforms a deterministic but intractable program into a tractable probabilistic program that generates unbiased estimates of the original program's intractable return value. This chapter also develops novel techniques for reasoning about probabilistic program transformations, culminating in very general correctness theorems. Furthermore, it establishes a foundation for the remaining chapters of the thesis, in two senses. First, it introduces the central mechanism of *estimation strategy annotations*, with which users can customize the estimates generated by our system in correct-by-construction ways. We use this technique throughout the thesis to afford users control over cost/variance trade-offs when estimating intractable quantities. Second, the remaining chapters sometimes represent other quantities of interest—density ratios and gradients—*as* intractable integrals that can be automatically and soundly estimated using the techniques developed in this chapter.

  - **Chapter 4 (Radon-Nikodym Derivatives)** tackles the problem of computing or estimating density ratios, or Radon-Nikodym derivatives, which are crucial for many inference algorithms. First, we introduce the *stochastic probability interface (SPI)*, a computational interface for densities that permits certain forms of unbiased estimation. We show that a broad variety of popular algorithms for inference can be soundly implemented against this interface. We then present a new program transformation for automatically compiling probabilistic programs into implementations of this interface.

  - **Chapter 5 (Derivatives)** develops an automatic differentiation (AD) algorithm for our probabilistic programming language. This algorithm can differentiate the intractable functions generated by the transformations from Chapters 3 and 4, enabling the optimization of expected values of stochastic processes, as well as of density ratios. We demonstrate how this AD algorithm, in conjunction with the transformations from previous chapters, allows for the derivation of unbiased estimators for gradients, which are essential for variational inference and stochastic optimization more generally.

- **Part III: Scaling Probabilistic Inference** demonstrates the practical value of our probabilistic programming techniques, by showing that our approach can be used to scale several applications of probabilistic inference.

  - **Chapter 6 (Programmable Bayesian Inference)** details how the program transformations developed in Part II can be used to construct flexible and modular libraries for *programmable inference*, to help users implement sophisticated Monte Carlo and variational inference algorithms. Empirical evaluations establish that: (1) the stochastic Radon-Nikodym estimators we automate can deliver orders-of-magnitude speedups over exact but slow densities; (2) the variance of the estimated densities is low enough that convergence of inference is not significantly impacted; and (3) in the context of variational inference,

when replicating certain models from the literature, our automation can build unbiased gradient estimators that lead to significantly faster convergence than achieved by the training algorithms hand-coded by the papers' original authors.

  - **Chapter 7 (Scalable Automated Reasoning about Relational Data)** introduces PClean, a domain-specific probabilistic programming system designed for modeling and cleaning tabular data about interrelated entities. PClean infers a latent clean database from potentially noisy observations by employing a specialized sequential Monte Carlo (SMC) algorithm with MCMC rejuvenation. This chapter highlights how the automated computation of Radon-Nikodym derivatives (for SMC weights and MCMC acceptance probabilities), as developed in Chapter 4, is specialized and applied within PClean to handle large-scale, real-world tabular datasets, e.g. to detect thousands of errors and impute millions of missing values in Medicare's national database of practicing physicians.
  - **Chapter 8 (Controllable Generation from Language Models)** presents novel techniques for guiding the output of language models (LMs) using sequential Monte Carlo methods. These techniques help users generate samples from programmatically defined target distributions that combine a base LM distribution with domain-specific syntactic or semantic constraints. The fast proposals employed during inference do not admit tractable density functions, but the necessary Radon-Nikodym derivatives can be tractably estimated using the techniques developed in this thesis. Extensive empirical evaluations show that controlling generation with these techniques can lead to significant improvements in downstream accuracy on a variety of structured generation tasks, from molecule synthesis to code generation.

- **Part IV: Conclusion** concludes the thesis.
  - **Chapter 9 (Future Directions)** discusses potential avenues for future research.

Part I

Preliminaries

2

A HIGHER-ORDER PROBABILISTIC PROGRAMMING LANGUAGE

This chapter reviews the syntax and semantics of a standard higher-order language for probabilistic programming, similar to that found in Ścibior et al. [2018]. This will serve as the language of study for the remainder of this thesis.

2.1 Background: The simply typed λ-calculus

Chapter 1 asserted that *programs* could be interpreted as *mathematical objects*, such as functions and probability distributions. In this chapter, we make this notion precise. To do so, we must fix a particular programming language, and then define a mapping from programs in this language to the mathematical objects they encode. Such a mapping is called a **denotational semantics**.

We will define our programming language in stages, starting in this section with a variant of the **simply typed λ-calculus** [Church, 1940], a stripped-down functional programming language. In the next section, we will add constructs for probabilistic programming, and in the sections after that, we will add support for recursion and recursive types, bringing the language's expressiveness closer to real-world languages such as Haskell and OCaml. But one key difference between this language and real-world functional programming languages is that we will assume *exact real numbers* are available as a primitive type, so that, e.g., $x + y$ computes the exact sum of two real-valued variables, rather than a floating-point approximation.

The material in this section is entirely standard, and can be safely skipped by readers already familiar with functional programming and denotational semantics. For readers with machine learning or probabilistic inference backgrounds who have never studied programming language theory, this section can serve as a self-contained introduction to just those concepts most relevant to this thesis.

Listing 2.1 Grammar of the simply typed λ-calculus, its types, and its judgments.

Syntactic category		**Productions**
Types τ	::=	$1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \tau_1 \times \tau_2 \mid \ell_1 \tau_1 + \cdots + \ell_n \tau_n \mid \tau_1 \to \tau_2$
Expressions e	::=	$x \mid c \mid f \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e \mid \lambda x.\, e \mid e_1\ e_2 \mid$
		$\ell_i\ e \mid \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\}$
Constants c	::=	$() \mid n \mid r$
Built-ins f	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \ldots$
Variable names x	$\in$	$\Sigma^* \setminus \{match, with, exp, \ldots\}$
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$
Typing contexts Γ	::=	$\cdot \mid \Gamma, x : \tau$
Typing judgments J	::=	$\Gamma \vdash e : \tau$
Syntactic sugar		**Expanded form**
$\mathbb{B}$	=	$true\ 1 + false\ 1$
true	=	$true()$
false	=	$false()$
if e_1 **then** e_2 **else** e_3	=	**match** e_1 **with** $\{true\ \_ \mapsto e_2 \mid false\ \_ \mapsto e_3\}$
let $x = e_1$ **in** e_2	=	$(\lambda x.\, e_2)\ e_1$

2.1.1. Types

Every program in the simply typed λ-calculus is assigned a **type** τ, which describes the kind of value that the program computes. (Variables within a program are also assigned types, which represent the kind of value that the variable can hold.) We include in our language small collection of **base types**:

- 1: the unit type, for programs that produce no meaningful output.
- $\mathbb{N}$: the type of natural numbers, for programs that output a natural number.
- $\mathbb{R}$: the type of real numbers, for programs that output a real number. We also include subtypes $\mathbb{R}_{[0,1]}$ for the reals on the unit interval, and $\mathbb{R}_{>0}$ for the (strictly) positive reals.

Given two types τ_1 and τ_2, we can form the **product type** $\tau_1 \times \tau_2$, for programs that output a pair of values, one of type τ_1 and one of type τ_2, as well as the **function type** $\tau_1 \to \tau_2$, for programs that represent functions with inputs of type τ_1 and outputs of type τ_2. We can also form **sum types** $\ell_1 \tau_1 + \cdots + \ell_n \tau_n$, for programs that output one of several alternatives. Each alternative outcome ℓ_i has an associated type τ_i of data that can be returned alongside the label ℓ_i (if there is no such auxiliary data, we can use $\tau_i = 1$, the unit type). For example, we can define a type for Boolean values as $true\ 1 + false\ 1$, representing values that are either *true* or *false*.

The grammar of types is given in Listing 2.1.

Remark 1 (Grammars)**.** A **grammar** is a way of formally specifying the syntax of

a programming language. It consists of a set of **syntactic categories**, which we mark in teal font, each of which can be *expanded* in one or more ways, according to the *productions* listed in the grammar. We write category ::= *production 1* | ⋯ | *production n*, where each production is a sequence of symbols, some of which may themselves be syntactic categories (with natural-number indices to disambiguate multiple occurrences of the same category). A string *belongs to* a syntactic category if it can be derived by repeatedly expanding syntactic category names according to the productions of the grammar. For example, τ can be expanded into $\tau_1 \to \tau_2$, which can be further expanded into $(\tau_3 \times \tau_4) \to \tau_2$, which can be futher expanded into $(\mathbb{R} \times \mathbb{N}) \to \mathbb{B}$. Thus, $(\mathbb{R} \times \mathbb{N}) \to \mathbb{B}$ belongs to the syntactic category of *Types* τ.

Throughout this thesis, we use symbols in teal font as (meta-language) variables that range over the possible items belonging to the corresponding syntactic category.

2.1.2. Expressions

Programs themselves are written as **expressions** e, whose grammar is also given in Listing 2.1. Intuitively, an expression e is a program that can be executed in an **environment** storing values for all the program's **free variables**—variables not declared within the program itself. For example, the program $x_1 + x_2$ has two free variables, x_1 and x_2, and its effect is to compute the sum of the numbers they are assigned to in the current environment. Our language supports the following constructs for building expressions:

- **Variables** x are expressions. A variable evaluates to the value that is currently assigned to that variable in the environment.
- **Constants** c are expressions that evaluate to some fixed value, no matter the environment. For each base type in our language, we have a corresponding set of constants. The constant of unit type is written (), similar to `None` in Python. All natural numbers n are available as constants of type $\mathbb{N}$, and all real numbers r are available as constants of type $\mathbb{R}$.
- The expression (e_1, e_2) evaluates to a **tuple**, or pair of values, the first of which is the result of evaluating e_1 and the second of which is the result of evaluating e_2.
- The expressions $\pi_1\ e$ and $\pi_2\ e$ evaluate to the first and second components of a tuple, respectively.
- The expression $\lambda x.\, e$ evaluates to a **function** that, when applied to an argument, evaluates e with x assigned to that argument in the environment. For example, the expression $\lambda x.\, x + 3$ evaluates to a function that, when applied to an argument, adds that argument to 3.
- The expression $e_1\ e_2$ evaluates to the result of applying the function e_1 to the argument e_2.
- The expression $\ell_i\ e$ evaluates to a labeled value, where e is evaluated and then tagged with the label ℓ_i. This is a constructor for sum types.

- The expression **match** e **with** $\{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\}$ evaluates e, which must yield a labeled value $\ell_i\ v$ for some i. It then evaluates e_i with x_i bound to v. This construct allows case analysis on sum types.
- **Built-in functions** f are functions that are predefined in our language. They include, for example, the standard arithmetic operations $+$, $-$, $\times$, $\div$, and exp. The built-ins also include the **inclusions** $\iota_{\tau_1 \hookrightarrow \tau_2}$ when τ_1 represents a subset of τ_2 (e.g. $\iota_{\mathbb{N} \hookrightarrow \mathbb{R}}$, $\iota_{\mathbb{R}_{[0,1]} \hookrightarrow \mathbb{R}}$, and so on.)

Remark 2 (Infix notation)**.** Strictly speaking, the syntax of our language suggests that to add two numbers, we should write $+\ (e_1, e_2)$—the application of the built-in $+$ function to the pair (e_1, e_2). We will abuse notation by writing $e_1 + e_2$ instead, and do similarly for other common binary operations. We will also freely use parentheses to group sub-expressions in the usual way, e.g. $e_1(e_2\ e_3)$ is the program that applies e_1 to the result of applying e_2 to e_3.

Remark 3 (Higher-order functions)**.** The simply typed λ-calculus is a **higher-order language**, which means that functions are first-class citizens that can be stored in variables and passed as arguments to other functions. For example, the expression $\lambda x.\ x\ 3$ represents a function that takes an argument x—which we assume is *itself* a function—and applies it to 3.

Remark 4 (Functions with multiple arguments)**.** The expression $\lambda x.\ e$ defines a function of one argument x. Functions of multiple arguments can be defined either by taking the single argument to be a tuple (whose individual values can then be accessed using π_1 and π_2), or by **currying**: we write $\lambda x_1.\ \lambda x_2.\ e$ to define a function of two arguments, the first of which is x_1 and the second of which is x_2. When applied to just one argument, the result is a function that is still waiting for the second argument. For example, the expression $((\lambda x_1.\ \lambda x_2.\ x_1 + x_2)\ 3)\ 2$ evaluates to 5.

Remark 5 (Syntactic sugar)**.** We introduce several forms of syntactic sugar to make it easier to read and write programs in our language:

- The type $\mathbb{B}$ is sugar for $\mathit{true}\ 1 + \mathit{false}\ 1$. It represents the type of Booleans. We also use **true** and **false** as constants of type $\mathbb{B}$, instead of $\mathit{true}()$ and $\mathit{false}()$.
- Similarly, we use **if** e_1 **then** e_2 **else** e_3 as syntactic sugar for **match** e_1 **with** $\{\mathit{true}\ \_ \mapsto e_2 \mid \mathit{false}\ \_ \mapsto e_3\}$.
- We use **let** $x = e_1$ **in** e_2 as syntactic sugar for $(\lambda x.\ e_2)\ e_1$.

2.1.3. Assigning types to expressions

A **context** Γ is a list of **type assumptions** $x_1 : \tau_1, \ldots, x_n : \tau_n$ for some variables $x_1, \ldots, x_n$ and types $\tau_1, \ldots, \tau_n$. A **typing judgment** $\Gamma \vdash e : \tau$ asserts that the expression e has type τ in the context Γ. Intuitively, this means that: (1) Γ contains a type assumption for every free variable occurring in e, and (2) assuming that each variable in Γ is assigned a value of the assumed type, e evaluates to a value of type τ.

A typing judgment is **valid** if it can be derived using the **typing rules** given in Listing 2.2. If $\Gamma \vdash e : \tau$ is valid for some Γ and some τ, we say e is **well typed** in context Γ. An expression that is well typed in the empty context is said to be **closed**.

Each typing rule in Listing 2.2 includes a list of *premises* (above the line) and a *conclusion* (below the line). Implicitly, all meta-variables (Γ, e, x, τ, etc.) are universally quantified, and the dividing line between the premises and a conclusion can be read as an implication ("for all Γ, e, x, ... , if *premises*, then *conclusion*"). For example, the rule Pair has two premises: $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. When both are valid, so is the conclusion: $\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$. That is, the expression (e_1, e_2) is well typed in Γ if each sub-expression is individually well typed in Γ, and in that case, the type of the overall expression is the product of the types of the sub-expressions.

The rule BuiltIn depends on pre-defined *argument types* $\mathsf{ArgType}(f)$ and *return types* $\mathsf{RetType}(f)$ for each f. For example, $\mathsf{ArgType}(+) = \mathbb{R} \times \mathbb{R}$ and $\mathsf{RetType}(+) = \mathbb{R}$, because + operates on a pair of real numbers and yields one real number.[1]

Example 1. Consider the expression $e := \lambda x.\, exp(x)$. To derive the typing judgment $\Gamma \vdash e : \mathbb{R} \to \mathbb{R}$, we apply the rules BuiltIn, Var, App, and Abs, as shown in the **derivation tree** below:

$$\textsc{Abs}\ \dfrac{\textsc{App}\ \dfrac{\textsc{BuiltIn}\ \dfrac{}{\Gamma, x : \mathbb{R} \vdash exp : \mathbb{R} \to \mathbb{R}} \qquad \textsc{Var}\ \dfrac{}{\Gamma, x : \mathbb{R} \vdash x : \mathbb{R}}}{\Gamma, x : \mathbb{R} \vdash exp(x) : \mathbb{R}}}{\Gamma \vdash \lambda x.\, exp(x) : \mathbb{R} \to \mathbb{R}}$$

A derivation tree is a tree where each node is a rule application, and the premises of a parent node are the conclusions of its children. The tree is pictured "upside down" with the root node—whose conclusion is the judgment we are ultimately trying to derive—at the bottom. Leaf nodes (at the top) are rules with no premises.

2.1.4. Denotational semantics

The **denotational semantics** of a programming language assign a mathematical *meaning* to each program in the language. The denotational semantics of the simply typed λ-calculus are summarized in Listing 2.3, and explained in detail below.

Semantics of types. First, for each type τ in our language, we define a set of values $[\![\tau]\!]$, the **denotation of** τ. Because we have infinitely many types, we define $[\![\cdot]\!]$ by induction on the grammar of types. For numeric types, we choose the corresponding sets of numbers. For the unit type, we can choose any singleton set; we write () for the single value of unit type, so that $[\![1]\!] = \{()\}$.

Product and function types are defined using the cartesian product and function

[1] Formally, we assume that numeric operations such as + and × have variants such as $+_{\mathbb{N}}$ and $\times_{\mathbb{N}}$ that operate on natural numbers; when clear from context, we omit the subscript.

Listing 2.2 Standard typing rules for the simply typed λ-calculus.

$$\textsc{Var}\ \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}\ (\text{if } x \notin \Gamma') \qquad \textsc{ConstR}\ \frac{}{\Gamma \vdash r : \mathbb{R}} \qquad \textsc{ConstN}\ \frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1}$$

$$\textsc{BuiltIn}\ \frac{}{\Gamma \vdash f : \mathsf{ArgType}(f) \to \mathsf{RetType}(f)} \qquad \textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{Proj}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i}$$

$$\textsc{Match}\ \frac{\Gamma \vdash e : \ell_1\tau_1 + \cdots + \ell_n\tau_n \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \ldots \qquad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} : \tau}$$

$$\textsc{Inj}\ \frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \ell_i\, e : \ell_1\tau_1 + \cdots + \ell_n\tau_n} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

space constructions from set theory: for products, $[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]$, and for functions, $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$, where $X \Rightarrow Y$ is the set of all functions from a set X to a set Y. Sum types denote disjoint unions with elements $\ell_i(v)$ for $v \in [\![\tau_i]\!]$.

Semantics of contexts. Given a context Γ, we write $[\![\Gamma]\!]$ for the set of all *environments* γ that satisfy the assumptions in Γ. Formally, an environment γ is a finite map from variable names to values. Given an environment γ, if x is in its domain, we write $\gamma[x]$ for the value assigned to x in γ. We write [] for the empty environment, and $\gamma[x \mapsto v]$ for the environment that extends γ with a mapping of x to v (overwriting any existing mapping of x in γ if necessary).

For the empty context $\cdot$, we have $[\![\cdot]\!] = \{[]\}$; that is, only the empty environment satisfies the assumptions in $\Gamma = \cdot$. For a non-empty context $\Gamma, x : \tau$, the set of satisfying environments is defined inductively, as $\{\gamma[x \mapsto v] \mid \gamma \in [\![\Gamma]\!], v \in [\![\tau]\!]\}$.

Semantics of well-typed expressions. Suppose $\Gamma \vdash e : \tau$ is a valid typing judgment. Then we write $[\![\Gamma \vdash e : \tau]\!]$, or $[\![e]\!]$ for short (when Γ and τ are clear), for the **denotation of** e, which is a function $[\![\Gamma]\!] \to [\![\tau]\!]$ mapping each environment γ satisfying the assumptions of Γ to the value $[\![e]\!](\gamma) \in [\![\tau]\!]$ that the expression takes on in that environment. That is, $[\![e]\!]$ tells us how to *run* the program e in a given environment.

The definition of $[\![e]\!]$ (the bottom panel of Listing 2.3) is given by structural induction on the grammar of expressions. For example, the denotation of a variable x is the function $[\![x]\!](\gamma) = \gamma[x]$ that maps each environment γ to the value assigned to x in γ. The denotation of a λ-abstraction $\lambda x.\, e$ in environment γ is the function $[\![\lambda x.\, e]\!](\gamma) = v \mapsto [\![e]\!](\gamma[x \mapsto v])$ that maps each value v to the result of running

Listing 2.3 Standard denotational semantics for the simply typed λ-calculus.

Semantics of Types

Type		*Set of values*
$[\![1]\!]$	$=$	$\{()\}$
$[\![\mathbb{N}]\!]$	$=$	$\mathbb{N}$
$[\![\mathbb{R}]\!]$	$=$	$\mathbb{R}$
$[\![\mathbb{R}_{[0,1]}]\!]$	$=$	$[0,1]$
$[\![\mathbb{R}_{>0}]\!]$	$=$	$(0,\infty)$
$[\![\tau_1 \times \tau_2]\!]$	$=$	$[\![\tau_1]\!] \times [\![\tau_2]\!]$
$[\![\tau_1 \to \tau_2]\!]$	$=$	$[\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$
$[\![\ell_1\tau_1 + \cdots + \ell_n\tau_n]\!]$	$=$	$\{\ell_i(v) \mid v \in [\![\tau_i]\!]\}$

Semantics of Contexts

Context		*Set of environments*
$[\![\cdot]\!]$	$=$	$\{[]\}$
$[\![\Gamma, x : \tau]\!]$	$=$	$\{\gamma[x \mapsto v] \mid \gamma \in [\![\Gamma]\!], v \in [\![\tau]\!]\}$

Semantics of Well-Typed Expressions

Term		*Value*
$[\![x]\!](\gamma)$	$=$	$\gamma[x]$
$[\![()]\!](\gamma)$	$=$	$()$
$[\![n]\!](\gamma)$	$=$	n
$[\![r]\!](\gamma)$	$=$	r
$[\![f]\!](\gamma)$	$=$	$\mathsf{Operation}(f)$
$[\![\ell_i\ e]\!](\gamma)$	$=$	$\ell_i([\![e]\!](\gamma))$

Term		*Value*
$[\![\textbf{match}\ e\ \textbf{with}\ \{\ell_i x_i \mapsto e_i\}_{1\le i\le n}]\!](\gamma)$	$=$	$[\![e_i]\!](\gamma[x_i \mapsto v])$ *where* $[\![e]\!](\gamma) = \ell_i(v)$
$[\![(e_1, e_2)]\!](\gamma)$	$=$	$([\![e_1]\!](\gamma), [\![e_2]\!](\gamma))$
$[\![\pi_i e]\!](\gamma)$	$=$	$\pi_i([\![e]\!](\gamma))$
$[\![e_1\ e_2]\!](\gamma)$	$=$	$[\![e_1]\!](\gamma)([\![e_2]\!](\gamma))$
$[\![\lambda x.\ e]\!](\gamma)$	$=$	$v \mapsto [\![e]\!](\gamma[x \mapsto v])$

e in the environment that maps x to v and is otherwise identical to γ. This rule implements *lexical scoping*, because the environment in which e is evaluated is the one in which $\lambda x.\ e$ is *written*, not the one in which it is *applied*.

Example 2. Consider the expression $e := \lambda x_1.\ \lambda x_2.\ x_1 + x_2$, a curried addition function. In any environment γ, the denotation $[\![e]\!](\gamma)$ is the function that maps each real number v_1 to the function that maps each real number v_2 to their sum $v_1 + v_2$:

$$\begin{aligned}
[\![e]\!](\gamma) &= v_1 \mapsto [\![\lambda x_2.\ x_1 + x_2]\!](\gamma[x_1 \mapsto v_1]) \\
&= v_1 \mapsto (v_2 \mapsto [\![x_1 + x_2]\!](\gamma[x_1 \mapsto v_1][x_2 \mapsto v_2])) \\
&= v_1 \mapsto (v_2 \mapsto [\![x_1]\!](\gamma[x_1 \mapsto v_1][x_2 \mapsto v_2]) + [\![x_2]\!](\gamma[x_1 \mapsto v_1][x_2 \mapsto v_2])) \\
&= v_1 \mapsto (v_2 \mapsto \gamma[x_1 \mapsto v_1][x_2 \mapsto v_2][x_1] + \gamma[x_1 \mapsto v_1][x_2 \mapsto v_2][x_2]) \\
&= v_1 \mapsto (v_2 \mapsto v_1 + v_2)
\end{aligned}$$

Note that this is the denotation of the expression even in environments that map x_1 or x_2 to specific (pre-existing) values. This is because $\gamma[x_1 \to v_1][x_2 \to v_2]$ overwrites any pre-existing mappings. For example, this means that $[\![\textbf{let}\ x_1 = 2\ \textbf{in}\ (e\ 3)\ 4]\!](\gamma) = 7$ (in any environment γ). We say that the instance of x_1 bound by λ in e **shadows** the outer binding of x_1 to the constant 2.

Denotations of closed expressions. When an expression e is well typed in the empty context (i.e., is closed), its meaning cannot depend on the environment, because it has no free variables. In this case, we sometimes write $[\![e]\!]$ to refer directly to its value (rather than $[\![e]\!](\gamma)$). We say that the closed expression e **denotes** $[\![e]\!]$. For example, $2 + 5$ denotes 7 and $\lambda x.exp(exp(x))$ denotes the function $x \mapsto e^{e^x}$.

2.2 Background: The probabilistic λ-calculus

So far, our programs have been **deterministic**: in a given environment γ, there is exactly one value $[\![e]\!](\gamma)$ to which a program e evaluates. This has made it a good language in which to define mathematical functions, by writing programs whose denotations are the functions of interest. In this section, we add new constructs to our language for sampling random outcomes. As a result, we will have to revise our semantics so that programs can denote not just values and functions, but also *probability distributions* and *Markov kernels*.

Remark 6. (Pseudo-randomness) Just as we have papered over the distinction between floating-point numbers (used by actual implementations) and exact reals (considered by our semantics), we will similarly consider an idealized language where random sampling is truly random, even though practical implementations will rely on imperfect **pseudo-random number generators** (PRNGs). This is because, in this thesis, we think of programs as compositional *definitions* of functions and probability distributions that the user is interested in, and develop program transformations that faithfully transform these definitions into other definitions that have related denotations (e.g., that denote the derivative of the function that the original program denoted). These program transformations are implementable faithfully on practical hardware, and are exact. The thing that is approximate is the *execution* of these programs, which does not faithfully compute their semantics, due to floating-point error and pseudo-randomness. But such approximations can be studied independently from the program transformations themselves, and are outside the scope of this thesis.

2.2.1. Syntax for probabilistic programming

In Listing 2.4, we extend the grammar of our language with a new production $\tau ::= \cdots \mid P\ \tau$. For each type τ, the type $P\ \tau$ represents the space of all *probability distributions* on $[\![\tau]\!]$. When an expression e has type $P\ \tau$, we think of the expression as *randomly generating* a value of type τ when it is run. We call a program of type $P\ \tau$ a **probabilistic program**.

Remark 7 (Samplers or distributions?)**.** The code of a probabilistic program of type $P\ \tau$ can be read in two ways. The most natural reading of the code is as a recipe for generating a random value of type τ. When we run a probabilistic program, this is the appropriate reading: we execute each line of code, generating and transforming random numbers to eventually obtain a random output of the

Listing 2.4 Grammar of the probabilistic λ-calculus, extending the simply-typed λ-calculus from Listing 2.1. New productions are highlighted.

Syntactic category		Productions
Types τ	::=	$1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \tau_1 \times \tau_2 \mid \ell_1 \tau_1 + \cdots + \ell_n \tau_n \mid \tau_1 \to \tau_2 \mid P\ \tau$
Expressions e	::=	$x \mid c \mid f \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e \mid \lambda x.\, e \mid e_1\ e_2 \mid$ $\ell_i\ e \mid \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} \mid$ $\textbf{return}\ e \mid \textbf{do}\ \{m\}$
Blocks m	::=	$e \mid x \leftarrow e; m$
Constants c	::=	$() \mid n \mid r$
Built-ins f_{det}	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \ldots$
f_{prb}	::=	$uniform \mid flip \mid normal \mid \ldots$
Variable names x	$\in$	$\Sigma^* \setminus \{match, with, exp, \ldots\}$
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$

Listing 2.5 Typing rules for the probabilistic constructs introduced in Listing 2.4, extending the standard typing rules from Listing 2.2.

$$\textsc{Return}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{return}\ e : P\ \tau} \qquad \textsc{Do}\ \frac{\Gamma \vdash e : P\ \tau}{\Gamma \vdash \textbf{do}\ \{e\} : P\ \tau} \qquad \textsc{Bind}\ \frac{\Gamma \vdash e : P\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash \textbf{do}\ \{m\} : P\ \tau_2}{\Gamma \vdash \textbf{do}\ \{x \leftarrow e; m\} : P\ \tau_2}$$

program. However, mathematically speaking, the code is also an unambiguous definition of a *probability distribution* on the space $[\![\tau]\!]$ of values of type τ. Our denotational semantics, which we develop in the next section, will give a precise meaning to the code of a probabilistic program as a probability distribution. But running the program does not compute this distribution; it just yields a *sample* from the distribution the program denotes.

Probabilistic built-ins. We extend the grammar of expressions with new constructs for producing and consuming random values. The simplest of these constructs are new builtins f_{prb}, such as $flip : \mathbb{R}_{[0,1]} \to P\ \mathbb{B}$, which takes as input a number $p \in [0, 1]$ and returns a random Boolean: *true* with probability p and *false* with probability $1 - p$. As another example, $normal : \mathbb{R} \times \mathbb{R}_{>0} \to P\ \mathbb{R}$ takes in a mean and standard deviation and returns a number generated from a Gaussian distribution with the given parameters.

Deterministic computations as probabilistic programs. Deterministic computations are special cases of probabilistic computations; they just happen to allocate all probability to a single outcome. The expression **return** e has type $P\ \tau$ when e has type τ, and represents the probability distribution that, with probability 1, yields

the result of e.

Sequencing probabilistic computations. A common pattern in probabilistic programs is to generate a value from one distribution, then use it to decide what (probabilistic) code to run next. This sequence of computations can be viewed itself as a probabilistic computation. Syntactically, we introduce the notion of a *block* m, which is either an expression e of probabilistic type, or a *sequencing* $x \leftarrow e; m$ of a *random assignment* $x \leftarrow e$ (where e is probabilistic) with a block m representing the remainder of the computation. Given a block m, **do** $\{m\}$ represents the probabilsitic computation that performs each random assignment in the block and finally returns the outcome of the probabilistic expression at the end of the block.

The typing rules for **return** and **do** are given in Listing 2.5.

Remark 8 (Probability monad)**.** P is a monad, and we have adopted Haskell's **return** and **do** notation for the monadic unit and bind.

Remark 9 (Syntactic sugar for deterministic assignment within a block)**.** Within a block m, we allow $x = e$ as syntactic sugar for $x \leftarrow$ **return** e. This allows for making deterministic assignments within a larger probabilistic computation.

Example 3 (Mixture of Gaussians)**.** The following program generates a sample from a mixture of two Gaussian distributions. It first flips a fair coin, then based on the outcome, samples from either $\mathcal{N}(0,1)$ or $\mathcal{N}(5,2)$:

$$\begin{aligned}\texttt{mixtureModel} \coloneqq \textbf{do}\ \{&b \leftarrow \mathit{flip}(0.5);\\ &\textbf{if}\ b\ \textbf{then}\ \mathit{normal}(0,1)\ \textbf{else}\ \mathit{normal}(5,2)\}\end{aligned}$$

This expression has type $P\ \mathbb{R}$, representing a probability distribution over real numbers. It demonstrates how we can use random assignments and conditional expressions to create more complex distributions from simpler ones.

Example 4 (Prior over linear functions)**.** The following program implements a simple prior distribution over linear functions, of the sort used in Bayesian linear regression. It generates a slope and intercept from prior distributions, then returns the linear map they induce.

$$\begin{aligned}\texttt{randomLinearFunc} \coloneqq \textbf{do}\ \{&m \leftarrow \mathit{normal}(0,1);\\ &b \leftarrow \mathit{normal}(0,1);\\ &\textbf{return}\ (\lambda x.\ m \times x + b)\}\end{aligned}$$

This program has type $P\ (\mathbb{R} \to \mathbb{R})$, and represents a probability distribution over real-valued functions.

Example 5 (Bayesian Network for flu, allergies, and symptoms)**.** The following program models a simple Bayesian network where having the flu or allergies influences the probability of observing fever and cough. It generates a sample representing the state of a patient (presence/absence of flu, allergies, fever, and

cough).

$$
\begin{aligned}
\texttt{genPatientState} \coloneqq \textbf{do}\ \{&\mathit{hasFlu} \leftarrow \mathit{flip}(0.05);\\
&\mathit{hasAllergy} \leftarrow \mathit{flip}(0.2);\\
&\mathit{hasFever} \leftarrow \textbf{if}\ \mathit{hasFlu}\ \textbf{then}\ \mathit{flip}(0.8)\ \textbf{else}\ \mathit{flip}(0.05);\\
&\mathit{hasCough} \leftarrow \textbf{if}\ \mathit{hasFlu}\ \textbf{then}\\
&\quad (\textbf{if}\ \mathit{hasAllergy}\ \textbf{then}\ \mathit{flip}(0.9)\ \textbf{else}\ \mathit{flip}(0.7))\\
&\quad \textbf{else}\\
&\quad (\textbf{if}\ \mathit{hasAllergy}\ \textbf{then}\ \mathit{flip}(0.6)\ \textbf{else}\ \mathit{flip}(0.1));\\
&\textbf{return}\ ((\mathit{hasFlu}, \mathit{hasAllergy}), (\mathit{hasFever}, \mathit{hasCough}))\}
\end{aligned}
$$

This program has type $P\,((\mathbb{B}\times\mathbb{B})\times(\mathbb{B}\times\mathbb{B}))$, representing a joint probability distribution over the presence of underlying conditions (flu and allergies) and of symptoms (fever and cough), according to the specified conditional probabilities.

Example 6 (Randomized 3D scene generation)**.** The following program generates a randomized 3D scene by sampling object properties and lighting conditions, then renders an image of the scene. We assume a function *render* that takes a scene description (e.g., object position, color, and light intensity) and returns an image represented as a flat vector of pixel values ($\mathbb{R}^{3\times16,384}$ for a 128x128 image with RGB color data).

$$
\begin{aligned}
\texttt{randomScene} \coloneqq \textbf{do}\ \{&\mathit{objX} \leftarrow \mathit{normal}(0,1);\\
&\mathit{objY} \leftarrow \mathit{normal}(0,1);\\
&\mathit{objZ} \leftarrow \mathit{normal}(0,1);\\
&\mathit{objR} \leftarrow \mathit{uniform}(0,1);\\
&\mathit{objG} \leftarrow \mathit{uniform}(0,1);\\
&\mathit{objB} \leftarrow \mathit{uniform}(0,1);\\
&\mathit{lightIntensity} \leftarrow \mathit{normal}(10,2);\\
&\mathit{sceneDesc} = ((\mathit{objX}, \mathit{objY}, \mathit{objZ}), (\mathit{objR}, \mathit{objG}, \mathit{objB}), \mathit{lightIntensity});\\
&\textbf{return}\ (\mathit{render}(\mathit{sceneDesc}))\}
\end{aligned}
$$

This program has type $P\,\mathbb{R}^{3\times16,384}$ (shorthand for a large tuple of scalars), representing a probability distribution over rendered images, where the variation comes from the probabilistically sampled scene parameters like object position, color, and lighting.

2.2.2. Finite semantics

To define the semantics of the probabilistic λ-calculus, we need to extend our definitions of $[\![\cdot]\!]$ to cover our new types $P\ \tau$ and our new language constructs (the primitives f_{prb}, **return**, and **do** blocks).

The first question to answer is what space of values $P\ \tau$ ought to denote. Intuitively,

Listing 2.6 Semantics of the probabilistic constructs introduced in Listing 2.4, specialized for a language where every distribution is supported on a finite number of points. These definitions extend the standard semantics from Listing 2.3.

$$
\begin{array}{rcl}
\textit{Type} & & \textit{Set of values} \\
[\![P\ \tau]\!] & = & \{\, p : [\![\tau]\!] \to \mathbb{R}_{\geq 0} \mid p(x) > 0 \text{ for finitely many } x, \textstyle\sum_X p(x) = 1 \,\} \\
\\
\textit{Expression} & & \textit{Value in } \gamma \\
[\![\textbf{return}\ e]\!](\gamma) & = & \lambda x. \begin{cases} 1 & \text{if } x = [\![e]\!](\gamma) \\ 0 & \text{otherwise} \end{cases} \\
[\![\textbf{do}\ \{e\}]\!](\gamma) & = & [\![e]\!](\gamma) \\
[\![\textbf{do}\ \{x \leftarrow e; m\}]\!](\gamma) & = & \lambda y. \textstyle\sum_{v \in \mathrm{supp}([\![e]\!](\gamma))} [\![e]\!](\gamma)(v) \cdot [\![\textbf{do}\ \{m\}]\!](\gamma[x \mapsto v])(y)
\end{array}
$$

it should denote the space of all *probability distributions* on $[\![\tau]\!]$, but for general sets X, there is no canonical construction of the probability distributions on X; in general, probability distributions are defined over *measurable spaces*, not arbitrary sets.

That said, there is a coherent definition of the set of all *finitely supported* probability distributions on a set X. The probabilistic programs we are interested in writing are generally not finitely supported; even many of our primitives, such as *normal* and *uniform*, are supported on infinite sets. But working through the semantics in the simple setting of finitely supported distributions may help the reader better digest the full semantics we present in the next section.

Finitely supported distributions. Let X be a set. Then we can define the set

$$\textbf{FinProb}(X) := \left\{ p : X \to \mathbb{R}_{\geq 0} \;\middle|\; p(x) > 0 \text{ for only finitely many } x \wedge \sum_{x \in X} p(x) = 1 \right\}$$

of **finitely supported probability distributions on** X. Intuitively, a distribution $p \in \textbf{FinProb}(X)$ allocates total probability mass 1 among finitely many values $x \in X$, called the **support of** p ($\mathrm{supp}(p)$).

In Listing 2.6, we show how finitely supported probability distributions can be used to give a semantics to our probabilistic constructs, so long as we remove primitives with infinite support (such as *normal*). We set the type $P\ \tau$ to denote $[\![P\ \tau]\!] = \textbf{FinProb}([\![\tau]\!])$. The primitive *flip* denotes the function

$$[\![\mathit{flip}]\!] = p \mapsto \left(b \mapsto \begin{cases} p & b = \mathit{true}() \\ 1 - p & b = \mathit{false}() \end{cases} \right),$$

mapping a parameter p to a finitely supported probability distribution on $\mathbb{B}$.

To interpret composite probabilistic programs, we need a semantics for **return** and

do. For **return**, we use the Dirac delta distribution:

$$[\![\textbf{return } e]\!](\gamma) = v \mapsto \begin{cases} 1 & v = [\![e]\!](\gamma) \\ 0 & \text{otherwise} \end{cases}.$$

For **do**, we use *marginalization* to sum over the possible values of the temporary variable x. For $\Gamma \vdash e : \tau_1$ and $\Gamma, x : \tau_1 \vdash \textbf{do}\{m\} : P\ \tau_2$, we have

$$\begin{aligned}[\![\textbf{do } \{x \leftarrow e; m\}]\!](\gamma) &= y \mapsto \sum_{v \in \text{supp}([\![e]\!](\gamma))} [\![e]\!](\gamma)(v) \cdot [\![\textbf{do } \{m\}]\!](\gamma[x \mapsto v])(y) \\ &= y \mapsto \mathbb{E}_{v \sim [\![e]\!](\gamma)} \left[[\![\textbf{do } \{m\}]\!](\gamma[x \mapsto v]) \right].\end{aligned}$$

That is, the probability of generating an outcome y is equal to the sum, over all possible outcomes v of e, of the probability of generating v for x and then generating y from the rest of the program (run with x set to v).

Using these extensions to $[\![\cdot]\!]$, we can assign to any expression of type $P\ \tau$ (relying on only the *flip* primitive, and not continuous or other infinitely supported distributions) a finitely supported probability distribution (over final outputs) that it denotes.

Remark 10 (Samplers or mass functions?)**.** In the semantics we have just defined, the meaning of a probabilistic program is a measure μ that can integrate any function. As discussed in Remarks 7 and 10, *this does not mean that when we run a probabilistic program, we compute an integral*. Running a probabilistic program just yields a sample: our new semantics, like our old semantics, is just a way to *analyze* which probability distribution is sampled from when we run a program.

2.2.3. Measure-theoretic preliminaries and quasi-Borel spaces

In this section, we develop a semantics that can handle probabilistic programs with infinite support. The key difficulty is that our naïve definition of a probability distribution, as a function assigning probabilities to individual elements of a set X, no longer suffices. Instead, we need a more sophisticated definition for probability distributions, as coherent assignments of probabilities to **events**, i.e. *subsets* of possible outcomes. This more sophisticated definition comes from measure theory.

Measures and measurable spaces. We first review several standard definitions from measure theory:

- A **measurable space** X is a pair (E_X, Σ_X) where E_X is a set and $\Sigma_X \subseteq \mathcal{P}(E_X)$ is a **σ-algebra** on E_X, i.e., a collection of subsets of E_X closed under countable unions and complements.
- A **measurable map** $f : X \to Y$ between measurable spaces $X = (E_X, \Sigma_X)$ and $Y = (E_Y, \Sigma_Y)$ is a function from E_X to E_Y such that $f^{-1}(A) \in \Sigma_X$ for all $A \in \Sigma_Y$.

- A **measure** on a measurable space (E_X, Σ_X) is a function $\mu : \Sigma_X \to \overline{\mathbb{R}}_{\geq 0}$ (where $\overline{\mathbb{R}}_{\geq 0} \coloneqq [0, \infty]$) such that $\mu(\emptyset) = 0$ and $\mu(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \mu(A_i)$ for any countable collection of pairwise disjoint sets $A_i \in \Sigma_X$. A **probability measure** is a measure μ such that $\mu(E_X) = 1$.

Example 7 (Measurable spaces with countable domains)**.** When E_X is a countable set (e.g., $\mathbb{N}$), it is common to take Σ_X to be the full powerset $\mathcal{P}(E_X)$.

Example 8 (Borel σ-algebras)**.** More generally, when E_X is the underlying set of a topological space (e.g., $\mathbb{R}$ or $\mathbb{R}^n$), it is common to take Σ_X to be the Borel σ-algebra $\mathcal{B}(E_X)$, which is the smallest σ-algebra containing all open subsets of E_X.

Example 9 (Measurable space of probability measures)**.** Given a measurable space $X = (E_X, \Sigma_X)$, we can define the measurable space Prob X of probability measures on X. The underlying set $E_{\text{Prob } X}$ consists of all probability measures on X. The σ-algebra $\Sigma_{\text{Prob } X}$ is the smallest σ-algebra making all evaluation maps $\mu \mapsto \mu(A)$ measurable for $A \in \Sigma_X$. More precisely, $\Sigma_{\text{Prob } X}$ is generated by sets of the form

$$\{\mu \in E_{\text{Prob } X} \mid \mu(A) \in B\}$$

for all $A \in \Sigma_X$ and all Borel sets $B \subseteq [0, 1]$.

Integration of measurable functions. Given a measure μ and a measurable function $f : X \to \overline{\mathbb{R}}_{\geq 0}$, we can *integrate f with respect to μ* to obtain a real number $\int f(x)\, \mu(dx)$. The integral is defined as follows:

- if f is the indicator function 1_A of a set $A \in \Sigma_X$, then $\int f(x)\, \mu(dx) \coloneqq \mu(A)$.
- If f is a simple function (a finite linear combination of indicator functions, $f(x) = \sum_{i=1}^{n} a_i \cdot 1_{A_i}(x)$ for positive coefficients a_i), then the integral is the linear combination of the integrals of its components $\left(\int f(x)\, \mu(dx) \coloneqq \sum_{i=1}^{n} a_i \cdot \mu(A_i)\right)$.
- Finally, if f is an arbitrary nonnegative function, then the integral is the supremum of the integrals of simple functions that are pointwise bounded above by f:

$$\int f(x)\, \mu(dx) \coloneqq \sup \left\{ \int g(x)\, \mu(dx) \;\middle|\; g \text{ simple } \wedge \forall x \in E_X, g(x) \leq f(x) \right\}.$$

For $A \in \Sigma_X$, we write $\int_A f(x)\, \mu(dx) \coloneqq \int 1_A(x) \cdot f(x)\, \mu(dx)$ for the integral of f over A with respect to μ.

Example 10 (Dirac measure)**.** Given a measurable space X and a point $x \in E_X$, the *Dirac measure at x* is the measure $\delta_x : \Sigma_X \to \overline{\mathbb{R}}_{\geq 0}$ that maps a set $A \in \Sigma_X$ to the value $1_A(x)$. The integral of a function $f : X \to \overline{\mathbb{R}}_{\geq 0}$ with respect to δ_x is $\int f(y)\, \delta_x(dy) = f(x)$.

Example 11 (counting measure)**.** The *counting measure* on a measurable space X is the measure $\mathbf{counting}_X : \Sigma_X \to \overline{\mathbb{R}}_{\geq 0}$ that maps a set $A \in \Sigma_X$ to $|A|$, the number of

elements in A. The integral of a function $f : X \to \overline{\mathbb{R}}_{\geq 0}$ with respect to the counting measure is $\int f(x)\ \mathbf{counting}_X(dx) = \sum_{x \in E_X} f(x)$.

Example 12 (Lebesgue measure)**.** The *Lebesgue measure* on $\mathbb{R}^n$ is the measure $\Lambda_n : \mathcal{B}(\mathbb{R}^n) \to \overline{\mathbb{R}}_{\geq 0}$ that maps a Borel set $A \in \mathcal{B}(\mathbb{R}^n)$ to its n-dimensional volume. The integral of a function $f : \mathbb{R}^n \to \overline{\mathbb{R}}_{\geq 0}$ with respect to Λ_n is $\int f(x)\ \Lambda_n(dx) = \int f(x)\ dx$, where the right-hand side is the standard Lebesgue integral in Euclidean space.

Example 13 (Pushforward of a measure by a measurable map)**.** Let $f : X \to Y$ be a measurable map and μ a measure on X. Then $f_*\mu$, the *pushforward of μ by f*, is the measure on Y defined by $(f_*\mu)(A) \coloneqq \mu(f^{-1}(A))$. The integral of a function $g : Y \to \overline{\mathbb{R}}_{\geq 0}$ with respect to $f_*\mu$ is $\int g(y)\ f_*\mu(dy) = \int g(f(x))\ \mu(dx)$.

Obstacles to a measure-theoretic semantics. In the previous section, we interpreted types τ as *sets* $[\![\tau]\!]$, and so to interpret $P\ \tau$, we needed a general recipe for turning a set X into a set $\mathbf{FinProb}(X)$ of probability distributions on X. We would now like to interpret $P\ \tau$ more generally, as a space of arbitrary *probability measures* on X. To do so, we need to:

- *Interpret types as measurable spaces.* Given a set X, there is no "set of probability measures on a X"; we must choose a specific σ-algebra on the set. Since we want to be able to talk about probability measures over any type in our language, we need to assign to each type in our language not just a set of values but also a σ-algebra; that is, $[\![\tau]\!]$ should pick out a *measurable space* of values for the type τ.

- *Interpret expressions as measurable functions.* If $e_1 : P\ \tau_1$ is a closed probabilistic program, and $x : \tau_1 \vdash e_2 : \tau_2$ is a deterministic computation on a variable x of type τ_1, then $\mathbf{do}\ \{x \leftarrow e_1; \mathbf{return}\ e_2\}$ represents the pushforward of the measure $[\![e_1]\!]$ by the map $v \mapsto [\![e_2]\!](x \mapsto v)$. Because pushforwards are only defined for *measurable* functions, this implies that in order to come up with a coherent semantics, we need for expressions like e_2 to always denote *measurable* functions of their environment. Note that this also implies that function types $\tau_1 \to \tau_2$ should denote some space $[\![\tau_1 \to \tau_2]\!]$ containing only the *measurable* maps from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$.

Unfortunately, there is an important obstacle to actually defining such a measure-theoretic semantics for our language. The problem arises when we try to define $[\![\tau_1 \to \tau_2]\!]$. The underlying set should be the set of all measurable functions from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$. But how should we choose a σ-algebra on this set? For example, which subsets of all measurable functions $\mathbb{R} \to \mathbb{R}$ should we consider to be measurable? Aumann [1961] showed that there is no way to choose a σ-algebra that makes the *evaluation map* $\mathbf{apply} \coloneqq (f, x) \mapsto f(x)$ measurable. But $\mathbf{apply}$ is precisely what we want function application to denote in our semantics; that is, $[\![f : \mathbb{R} \to \mathbb{R}, x : \mathbb{R} \vdash f\ x : \mathbb{R}]\!]$ should be precisely $\mathbf{apply}$. The inability to choose a σ-algebra that makes $\mathbf{apply}$ measurable, then, prevents us from achieving the second goal above, of ensuring that every expression has a measurable semantics.

Quasi-Borel spaces. These difficulties motivated the devleopment of **quasi-Borel spaces**, an drop-in replacement for measurable spaces that *do* allow us to interpret function types [Heunen et al., 2017]. We now review the key definitions.

Definition 1 (Quasi-Borel space)**.** A **quasi-Borel space** X is a pair (E_X, M_X), where E_X is a set, and M_X is a collection of functions, called *random elements*, from $\mathbb{R}$ to E_X. The collection M_X is required to satisfy the following closure properties:

- (*pre-composition*) If $f \in M_X$ and $g : \mathbb{R} \to \mathbb{R}$ is measurable, then $f \circ g \in M_X$.
- (*all constants*) If $x \in E_X$, then the constant function $r \mapsto x \in M_X$.
- (*piecewise gluing*) If $(f_i)_{i\in\mathbb{N}}$ is a countable collection of functions in M_X and $(A_i)_{i\in\mathbb{N}}$ is a countable partition of $\mathbb{R}$ into disjoint Borel sets, then the function $f : \mathbb{R} \to E_X$ defined by $f(r) = f_i(r)$ when $r \in A_i$ is in M_X.

The analogue of a measurable map for quasi-Borel spaces is a **quasi-Borel map**.

Definition 2 (Quasi-Borel map)**.** A **quasi-Borel map** $f : X \to Y$ between quasi-Borel spaces X and Y is a function $f : E_X \to E_Y$ such that $f \circ \alpha \in M_Y$ for all $\alpha \in M_X$.

Example 14 (Standard Borel spaces)**.** When X is a **standard Borel space** (i.e., a measurable space isomorphic to $\{1, \dots, n\}$ for some n, to $\mathbb{N}$, or to $\mathbb{R}$), we can create an equivalent quasi-Borel space X by taking $M_X := \{f \mid f \text{ is measurable}\}$. This includes as special cases:

- The natural numbers $\mathbb{N}$, whose random elements are those functions f such that $f^{-1}(n) \in \mathcal{B}(\mathbb{R})$ for all $n \in \mathbb{N}$.
- The reals $\mathbb{R}$, whose random elements are those functions f such that $f^{-1}(A) \in \mathcal{B}(\mathbb{R})$ for all Borel sets $A \subseteq \mathbb{R}$.
- The n-dimensional vectors $\mathbb{R}^n$, whose random elements are those functions f such that $f^{-1}(A) \in \mathcal{B}(\mathbb{R}^n)$ for all Borel sets $A \subseteq \mathbb{R}^n$.
- The non-negative extended reals $\overline{\mathbb{R}}_{\geq 0}$, whose random elements are those functions f measurable with respect to the σ-algebra $\{A \subseteq \overline{\mathbb{R}}_{\geq 0} \mid A \cap \mathbb{R} \in \mathcal{B}(\mathbb{R})\}$.

Example 15 (Products of quasi-Borel spaces)**.** The product of two quasi-Borel spaces X and Y is the quasi-Borel space $X \times Y$ with underlying set $E_{X\times Y} := E_X \times E_Y$ and random elements $M_{X\times Y} := \{f \mid \pi_1 \circ f \in M_X \text{ and } \pi_2 \circ f \in M_Y\}$.

Example 16 (Sums of quasi-Borel spaces)**.** Given quasi-Borel spaces $X_1, \dots, X_n$, we can form their sum $\sum_{i=1}^n \ell_i X_i = \ell_1 X_1 + \cdots + \ell_n X_n$ as a quasi-Borel space with underlying set $E_{\sum_{i=1}^n \ell_i X_i} = \{\ell_1(x) \mid x \in E_{X_1}\} \cup \cdots \cup \{\ell_n(x) \mid x \in E_{X_n}\}$ and random elements

$$M_{\sum_{i=1}^n \ell_i X_i} = \left\{ \left(r \mapsto \begin{cases} \ell_1(\alpha_1(r)) & r \in A_1 \\ \cdots & \cdots \\ \ell_n(\alpha_n(r)) & r \in A_n \end{cases} \right) \;\middle|\; \alpha_i \in M_{X_i}, (A_i)_{i=1}^n \text{ measurably partition } \mathbb{R} \right\}$$

Intuitively, a random element of a sum is constructed by partitioning $\mathbb{R}$ and using a different random element α_i from each X_i on the corresponding region A_i, with each value tagged by the appropriate label ℓ_i.

Example 17 (Function spaces)**.** The space of quasi-Borel maps from a quasi-Borel space X to a quasi-Borel space Y forms a quasi-Borel space $X \Rightarrow Y$, with underlying set $E_{X\Rightarrow Y} \coloneqq \{f \mid \forall \alpha \in M_X.\ f \circ \alpha \in M_Y\}$ and random elements $M_{X\Rightarrow Y} \coloneqq \{f \mid \forall g \in M_X.\ (r \mapsto f(r)(g(r))) \in M_Y\}$. Note that **apply** $: (X \Rightarrow Y) \times X \to Y$, defined by $\mathbf{apply}(f, x) \coloneqq f(x)$, is a quasi-Borel map, because for any $\alpha \in M_{(X\Rightarrow Y)\times X}$ we have $\pi_2 \circ \alpha \in M_X$ and $\pi_1 \circ \alpha \in M_{X\Rightarrow Y}$, which together imply that

$$\mathbf{apply} \circ \alpha = (r \mapsto \pi_1(\alpha(r))(\pi_2(\alpha(r))))$$

is in M_Y.

Example 18 (Quasi-Borel subspaces)**.** If $E_Y \subseteq E_X$, then we can form a quasi-Borel space Y by equipping E_Y with random elements $M_Y \coloneqq \{f \mid f \in M_X \text{ and } \forall r \in \mathbb{R}.\ f(r) \in E_Y\}$. M_Y satisfies the necessary closure properties whenever M_X does.

Definition 3 (Quasi-Borel measure)**.** A **quasi-Borel measure** on a quasi-Borel space X is a quasi-Borel map $\mu : (X \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \to \overline{\mathbb{R}}_{\geq 0}$ such that there exists $\alpha \in M_X$ and $A \in \mathcal{B}(\mathbb{R})$ satisfying $\mu(f) = \int_A f(\alpha(r))\, dr$ for all quasi-Borel morphisms $f : X \to \overline{\mathbb{R}}_{\geq 0}$. The quasi-Borel measures on X form a quasi-Borel space Meas X, a subspace of the function space $(X \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \overline{\mathbb{R}}_{\geq 0}$.

Definition 4 (Quasi-Borel probability measure)**.** A **quasi-Borel probability measure** is a quasi-Borel measure μ such that $\mu(x \mapsto 1) = 1$. The quasi-Borel probability measures on X form a quasi-Borel space Prob X, a subspace of Meas X.

Definition 5 (Integration with respect to a quasi-Borel measure)**.** If μ is a quasi-Borel probability measure on a quasi-Borel space X, and $f : X \to \overline{\mathbb{R}}_{\geq 0}$ is a quasi-Borel function, then we can define the integral

$$\int f(x)\ \mu(dx) \coloneqq \mu(f).$$

We also write $\int f d\mu$ for the same quantity, and also $\mathbb{E}_\mu[f]$ or $\mathbb{E}_{x\sim\mu}[f(x)]$, especially (but not exclusively) when μ is a probability measure.

Note that because **apply** is a quasi-Borel map, integration with respect to a quasi-Borel measure (**integrate** : Meas $X \times (X \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \to \overline{\mathbb{R}}_{\geq 0}$) is also a quasi-Borel map. Indeed, **integrate** is just the restriction of **apply** to the domain Meas $X \times (X \Rightarrow \overline{\mathbb{R}}_{\geq 0})$.

Definition 6 (Quasi-Borel measure kernel)**.** A **quasi-Borel measure kernel** (resp. **probability kernel**) is a quasi-Borel map $k : X \to$ Meas Y (resp. $X \to$ Prob Y).

Definition 7 (Integration with respect to a quasi-Borel measure kernel)**.** Let $k : X \to$ Meas Y be a kernel, and $x \in E_X$ a point in the underlying set of X. Further, let

$f : Y \to \overline{\mathbb{R}}_{\geq 0}$ be a quasi-Borel map. Then we write

$$\int f(y)k(x, dy) \coloneqq \int f(y)k(x)(dy) = k(x)(f)$$

for the integral of f with respect to k at x. Note that by the notation introduced in Definition 5, we can also write $\mathbb{E}_{y\sim k(x)}[f(y)]$ or $\mathbb{E}_{k(x)}[f]$ for the same quantity.

Definition 8 (Composition of measures and kernels)**.** The composition of a quasi-Borel measure μ on X and a quasi-Borel measure kernel $k : X \to \text{Meas } Y$ is the quasi-Borel measure μk on Y, defined by $(\mu k)(g) \coloneqq \mu(x \mapsto k(x)(g))$. We can also write the composition using the **Kock integral** notation:

$$\iint k(x)\mu(dx) \coloneqq k\mu = \left(g \mapsto \iint g(y)k(x)(dy)\mu(dx)\right).$$

Just as with the integral notation in Definition 7, we can also use Kock integration with respect to kernels instead of measures. Given $x \in E_X$, $k_1 : X \to \text{Prob } Y$, and $k_2 : Y \to \text{Prob } Z$, we have:

$$\iint k_2(y)k_1(x, dy) \coloneqq \iint k_2(y)k_1(x)(dy) = \left(g \mapsto \iint g(z)k_2(y, dz)k_1(x, dy)\right).$$

The **dependent product** of μ with k is the quasi-Borel measure $\mu \otimes k$ on $X \times Y$,

$$\mu \otimes k \coloneqq \iint \delta_{(x,y)}\mu(dx)k(x, dy) = (g \mapsto \mu(x \mapsto k(x)(y \mapsto g(x, y)))).$$

Example 19 (Dirac measure)**.** For $x \in E_X$, $\delta_x(f) = f(x)$ is also a quasi-Borel probability measure. To see this, note that $\delta_x(f) = \int_A f(\alpha(r))\, dr$ for $A = [0, 1]$ and $\alpha(r) = x$.

Example 20 (Counting measure)**.** Let X be a quasi-Borel space whose underlying set E_X has countably many elements $(x_i)_{i\in\mathbb{N}}$. The counting measure on X is the quasi-Borel measure $\textbf{counting}_X(f) \coloneqq \sum_{i\in\mathbb{N}} f(x_i)$. To see that this is a quasi-Borel measure, note that it arises as $\int_A f(\alpha(r))\, dr$ for $A = \mathbb{R}_{\geq 0}$ and α the function $r \mapsto x_{\lfloor r \rfloor}$. When E_X has finitely many elements $(x_i)_{i\in\{1,\ldots,n\}}$, we can instead take $A = [0, n)$.

Example 21 (Lebesgue measure)**.** The quasi-Borel Lebesgue measure on $\mathbb{R}$ is $\Lambda(f) \coloneqq \int f(x)\, dx$, which arises as $\int_A f(\alpha(r))\, dr$ for $A = \mathbb{R}$ and α the identity function.

Example 22 (Pushforward of a quasi-Borel measure by a quasi-Borel map)**.** Let $f : X \to Y$ be a quasi-Borel map and μ a quasi-Borel measure on X. Then $f_*\mu$, the *pushforward of μ by f*, is the quasi-Borel measure on Y defined by $(f_*\mu)(g) \coloneqq \mu(g \circ f)$. If $\mu(g_X) = \int_A g_X(\alpha_X(r))\, dr$, then $(f_*\mu)(g_Y) = \int_A g_Y(\alpha_Y(r))\, dr$, where $\alpha_Y = f \circ \alpha_X$.

Example 23 (Product of quasi-Borel measures)**.** The product of two quasi-Borel measures μ and ν on X and Y is the quasi-Borel measure $\mu \otimes \nu$ on $X \times Y$, defined by $(\mu \times \nu)(f) \coloneqq \mu(x \mapsto \nu(y \mapsto f(x, y)))$. To see why this is a quasi-Borel measure, note

Listing 2.7 Semantics of the probabilistic λ-calculus in the general case (infinite-support and continuous distributions).

Qbs Semantics of Types

Type		*Qbs of values*
$[\![1]\!]$	$=$	1
$[\![\mathbb{N}]\!]$	$=$	$\mathbb{N}$
$[\![\mathbb{R}]\!]$	$=$	$\mathbb{R}$
$[\![\mathbb{R}_{[0,1]}]\!]$	$=$	$[0,1] \subseteq \mathbb{R}$
$[\![\mathbb{R}_{>0}]\!]$	$=$	$(0,\infty) \subseteq \mathbb{R}$
$[\![\tau_1 \times \tau_2]\!]$	$=$	$[\![\tau_1]\!] \times [\![\tau_2]\!]$
$[\![\sum_{i=1}^n \ell_i \tau_i]\!]$	$=$	$\sum_{i=1}^n \ell_i [\![\tau_i]\!]$
$[\![\tau_1 \to \tau_2]\!]$	$=$	$[\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$
$[\![P\ \tau]\!]$	$=$	$\mathrm{Prob}\ [\![\tau]\!]$

Qbs Semantics of Contexts

Context		*Qbs of environments*
$[\![\cdot]\!]$	$=$	$(E = \{[]\}, M = \{r \mapsto []\})$
$[\![\Gamma, x : \tau]\!]$	$=$	$(E = \{\gamma[x \mapsto v] \mid \gamma \in E_{[\![\Gamma]\!]}, v \in E_{[\![\tau]\!]}\},$ $M = \{r \mapsto \alpha_1(r)[x \mapsto \alpha_2(r)] \mid \alpha_1 \in M_{[\![\Gamma]\!]}, \alpha_2 \in M_{[\![\tau]\!]}\})$

Qbs Semantics of Probabilistic Constructs

Expression		*Value*
$[\![\mathbf{return}\ e]\!](\gamma)$	$=$	$\delta_{[\![e]\!](\gamma)}$
$[\![\mathbf{do}\ \{e\}]\!](\gamma)$	$=$	$[\![e]\!](\gamma)$
$[\![\mathbf{do}\ \{x \leftarrow e; m\}]\!](\gamma)$	$=$	$\int\!\!\!\!\!\!\int [\![\mathbf{do}\{m\}]\!](\gamma[x \mapsto v])\ [\![e]\!](\gamma, dv)$

that there exists a measurable bijection $\phi : \mathbb{R} \to \mathbb{R}^2$ such that $\phi_* \Lambda = \Lambda_2$. Supposing $\mu(f_X) = \int_{A_X} f_X(\alpha_X(r))\, dr$ and $\nu(f_Y) = \int_{A_Y} f_Y(\alpha_Y(r))\, dr$, we have

$$\begin{aligned}
(\mu \otimes \nu)(f) &= \int_{A_X} \int_{A_Y} f(\alpha_X(r_1), \alpha_Y(r_2))\, dr_1\, dr_2 \\
&= \int_{A_X \times A_Y} f(\alpha_X(r_1), \alpha_Y(r_2)) \Lambda_2(d(r_1, r_2)) \\
&= \int_{\phi^{-1}(A_X \times A_Y)} f(\alpha_X(\phi_1(r)), \alpha_Y(\phi_2(r)))\, dr \\
&= \int_A f(\alpha(r))\, dr,
\end{aligned}$$

where $\alpha(r) = (\alpha_X(\phi_1(r)), \alpha_Y(\phi_2(r)))$ and $A = \phi^{-1}(A_X \times A_Y)$.

2.2.4. Infinite and continuous semantics

We are finally ready to give a semantics to the probabilistic λ-calculus with infinitely supported distributions. To do so, we:

- *Interpret types as quasi-Borel spaces.* To each type τ in our language, we assign a *quasi-Borel space* $[\![\tau]\!]$ as its denotation. To our ground types (e.g., $\mathbb{B}, \mathbb{R}, \mathbb{R}_{[0,1]}$) we associate the obvious quasi-Borel counterparts of their previously defined

denotations, described in Example 14. To the product type $\tau_1 \times \tau_2$, we associate the product of the denotations: $[\![\tau_1 \times \tau_2]\!] \coloneqq [\![\tau_1]\!] \times [\![\tau_2]\!]$ (see Example 15). To the sum type $\ell_1\tau_1 + \cdots + \ell_n\tau_n$, we associate the sum of the denotations: $\ell_1[\![\tau_1]\!] + \cdots + \ell_n[\![\tau_n]\!]$ (see Example 16). To the function type $\tau_1 \to \tau_2$, we assign the quasi-Borel space of quasi-Borel maps $[\![\tau_1 \to \tau_2]\!] \coloneqq [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$ (see Example 17). To the type $P\ \tau$, we assign the quasi-Borel space of probability measures Prob $[\![\tau]\!]$ (see Definition 4).

- *Interpret contexts as quasi-Borel environments.* A typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is now associated with a *quasi-Borel space* $[\![\Gamma]\!]$ of environments. The underlying set $E_{[\![\Gamma]\!]}$ is the set of environments γ satisfying the assumptions in Γ, as previously defined. The corresponding set of random elements is

$$M_{[\![\Gamma]\!]} \coloneqq \{\alpha : \mathbb{R} \to E_{[\![\Gamma]\!]} \mid \text{for all assumptions } (x : \tau) \in \Gamma, (r \mapsto \alpha(r)[x]) \in M_{[\![\tau]\!]}\}.$$

- *Interpret expressions as quasi-Borel functions.* Given a well-typed expression $\Gamma \vdash e : \tau$, we assign it a denotation $[\![e]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$, a quasi-Borel map. For the non-probabilistic constructs in our language, the previously defined expression denotations (in Listing 2.7) are already quasi-Borel. For the new constructs, we assign new quasi-Borel denotations:

  1. *Probabilistic built-ins.* Primitives f_{prb}, such as *flip* and *normal*, now denote quasi-Borel probability kernels. For example, *flip* denotes the kernel

$$\begin{aligned}[\![\mathit{flip}]\!] &\coloneqq p \mapsto (f \mapsto p \cdot f(\mathit{true}) + (1-p) \cdot f(\mathit{false})) \\ &= p \mapsto \left(f \mapsto \int_{[0,1]} f(\alpha(r))\, dr\right),\end{aligned}$$

     where $\alpha(r) = \begin{cases} \mathit{true} & r \le p \\ \mathit{false} & r > p \end{cases}$. (This last equality establishes that it is in fact a quasi-Borel kernel.) But we can now also define semantics for continuous primitives such as *normal*:

$$\begin{aligned}[\![\mathit{normal}]\!] &\coloneqq (\mu, \sigma) \mapsto \left(f \mapsto \int f(x) \cdot \mathcal{N}(x; \mu, \sigma)\, dx\right) \\ &= (\mu, \sigma) \mapsto \left(f \mapsto \int_{[0,1]} f(\alpha(r))\, dr\right),\end{aligned}$$

     where α is the inverse-cdf function for the Gaussian distribution centered at μ with standard deviation σ.

  2. *The **return** construct.* We interpret **return** using the Dirac delta (Example 19). That is, $[\![\textbf{return}\ e]\!](\gamma) \coloneqq \delta_{[\![e]\!](\gamma)}$.

  3. *Composition with **do** blocks.* As in the finite semantics, we interpret sequencing $\textbf{do}\{x \leftarrow e; m\}$ by marginalizing the temporary variable x, but now instead of

Listing 2.8 Grammar of the probabilistic λ-calculus with recursion, extending the probabilistic λ-calculus without recursion from Listing 2.4. New productions are highlighted.

Syntactic category		Productions
Types τ	::=	$1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \tau_1 \times \tau_2 \mid \ell_1 \tau_1 + \cdots + \ell_n \tau_n \mid \tau_1 \to \tau_2 \mid P\ \tau \mid P_{lazy}\ \tau$
Expressions e	::=	$x \mid c \mid f \mid \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e \mid \lambda x.\, e \mid e_1\ e_2 \mid \textbf{return}\ e \mid \textbf{do}\ \{m\} \mid \textbf{return}_{lazy}\ e \mid \textbf{do}_{lazy}\{x \leftarrow e; m\} \mid \mu x.\, e$
Blocks m	::=	$e \mid x \leftarrow e; m$
Constants c	::=	$() \mid n \mid r$
Built-ins f_{det}	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \ldots$
f_{prb}	::=	$uniform \mid flip \mid normal \mid \ldots$
Variable names x	$\in$	$\Sigma^* \setminus \{if, then, else, true, false, exp, \ldots\}$
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$

a sum over the support of $[\![e]\!]$, we take an integral with respect to it:

$$\begin{aligned}
[\![\textbf{do}\ \{x \leftarrow e; m\}]\!](\gamma) &:= \int [\![\textbf{do}\ \{m\}]\!](\gamma[x \mapsto v])\ [\![e]\!](\gamma, dv) \\
&= f \mapsto \int \left(\int f(y)\ [\![\textbf{do}\ \{m\}]\!](\gamma[x \mapsto v])(dy) \right) [\![e]\!](\gamma)(dv) \\
&= f \mapsto \mathbb{E}_{v \sim [\![e]\!](\gamma)} \left[\mathbb{E}_{y \sim [\![\textbf{do}\ \{m\}]\!](\gamma[x \mapsto v])}[f(y)] \right] \\
&= k\mu,
\end{aligned}$$

where k is the kernel $v \mapsto [\![\textbf{do}\ \{m\}]\!](\gamma[x \mapsto v])$ and μ is the measure $[\![e]\!](\gamma)$.

Remark 11 (Samplers or measures?)**.** In the semantics we have just defined, the meaning of a probabilistic program is a measure μ that can integrate any function. As discussed in Remarks 7 and 10, *this does not mean that when we run a probabilistic program, we compute an integral*. Running a probabilistic program just yields a sample: our new semantics, like our old semantics, is just a way to *analyze* which probability distribution is sampled from when we run a program.

2.3 Probabilistic programming with recursion

In this section, we extend our probabilistic λ-calculus to support *recursive* definitions, by adding the construct $\mu x.\, e$ to the language (Listing 2.8). Intuitively, this expression runs e, but in an environment where x is itself bound to $\mu x.\, e$, allowing x to be used

within e to recursively reference the program being defined. The typing rule is

$$\textsc{Mu}\ \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x.\, e : \tau},$$

where τ is the type of the value being defined recursively. We give several examples for intuition on the flavor of recursion we implement, before defining the semantics.

Example 24 (Factorial function)**.** The factorial function can be defined recursively using the μ construct:

$$\begin{aligned}&\texttt{factorial} : \mathbb{N} \to \mathbb{N}\\ &\texttt{factorial} \coloneqq \mu f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times f(n-1)\end{aligned}$$

Here, $f : \mathbb{N} \to \mathbb{N}$ is bound by μ to refer to the function being defined. When $n = 0$, the base case returns 1. Otherwise, it multiplies n by the factorial of $n - 1$, computed recursively.

Example 25 (Geometric distribution)**.** The type of the value being defined recursively need not be a function type; we can also define probability distributions recursively. The geometric distribution models the number of Bernoulli trials needed to get one success. We can encode it in our probabilistic language using recursion:

$$\begin{aligned}&\texttt{geometric} : P\,\mathbb{N}\\ &\texttt{geometric} \coloneqq \mu g.\ \textbf{do}\{\\ &\qquad b \leftarrow \mathit{flip}(0.5);\\ &\qquad \textbf{if}\ b\ \textbf{then return}\ 0\ \textbf{else do}\{\\ &\qquad\quad n \leftarrow g;\\ &\qquad\quad \textbf{return}\ (n+1)\\ &\qquad \}\\ &\}\end{aligned}$$

Here, *geometric* represents a distribution over natural numbers where the probability of n is 0.5^{n+1}. If the first flip succeeds (with probability 0.5), we return 0 trials. Otherwise, we recursively sample from the same distribution and add 1 to the result. The name g is bound by μ to refer to the distribution being defined.

Example 26 (Infinite loops)**.** General recursion opens the door to infinite loops. For example, consider the following expressions:

- $\mu f.\lambda x.f(x+1)$ of type $\mathbb{R} \to \mathbb{R}$. This expression evaluates to a function, which, given an input x, recursively calls itself with the argument $x + 1$. Running this function on any input will **diverge**, i.e., result in an infinite loop.
- $\mu p.\textbf{do}\{n \leftarrow p; \textbf{return}\ (n+1)\}$ of type $P\,\mathbb{N}$. If we try to run this probabilistic program,

it will immediately attempt to call itself recursively, to generate a natural number n. Since there is no base case, the recursion never terminates, and the program diverges.

- $\mu r.\ r \div 2 + 1$ of type $\mathbb{R}$. This expression attempts to compute a real number by immediately evaluating itself recursively, dividing the result by 2, and adding 1. Without a base case, it diverges. (This is the case even though there is in theory a fixed-point of the equation $r = r \div 2 + 1$, namely $r = 2$.)
- $\mu x.\ x$, which can be assigned any type τ. This expression attempts to compute a value of type τ by immediately evaluating itself recursively, with no base case, and thus diverges.

Lazy evaluation (deterministic programs). Consider the expression $\pi_1\ (2, \mu x.\ x)$. If $\pi_1\ e$ computes $[\![e]\!]$ before extracting the first element, then this expression diverges. But if we adopt a **lazy evaluation strategy**—as in Haskell—we can avoid the infinite loop. In lazy evaluation, a sub-expression is evaluated only when its value is needed by the rest of the program. On this example, lazy evaluation computes the result 2.

Without recursion, laziness changes how efficiently a result is computed, but not the result itself. With recursion, lazy evaluation can change the result of a program, as the example above demonstrates. Our semantics for recursion will be defined to correspond to a lazy evaluation strategy similar to Haskell's: divergence of a *deterministic* sub-expression will not cause the entire program to diverge unless the program needs to evaluate (or **force**) the sub-expression.

Lazy evaluation (probabilistic programs). Consider the probabilistic program $\mathbf{do}\{x \leftarrow \mu y.\ y; \mathbf{return}\ 5\}$, of type $P\ \mathbb{R}$. Does this program diverge, or does it (deterministically) return 5? In real-world languages like Haskell, the answer depends on the implementation of probabilistic sampling. For example, if we use Haskell's `IO` monad, the program does diverge. This is because, in `IO`, a value of type $P\ \tau$ is implemented as a function that transforms a PRNG state. In the expression $\mathbf{do}\ \{x \leftarrow e; m\}$, m does depend on e even if it does not reference x, because we must let e transform the PRNG state before passing it to m. Thus, if e diverges before it can yield an updated PRNG state, the overall program will diverge.[2] By contrast, if we use a backend based on *splittable* random number generators, as implemented by Dash et al. [2023], programs like $\mathbf{do}\{x \leftarrow \mu y.\ y; \mathbf{return}\ 5\}$ do not diverge. They handle $\mathbf{do}\ \{x \leftarrow e; m\}$ by *splitting* the PRNG into a substate for e and a substate for m, then immediately running m, forcing e only when x is used.

We will model both behaviors, using two probability monads: P and P_{lazy}. By default, we will use P, which models the behavior of the Haskell `IO` monad. However, in several parts of the thesis, it will be useful to consider fully lazy probabilistic programs, modeled by P_{lazy}.

[2] Note that $\mathbf{do}\ \{x \leftarrow \mathbf{return}\ e; m\}$ does not diverge, even if e does. That is because it is immediately obvious that $\mathbf{return}\ e$ does not change the PRNG state; thus, the PRNG state can be returned immediately and the actual sampled value can be handled lazily.

2.3.1. Quasi-Borel predomains

Above we gave an intuitive operational explanation of μ. We now make these intuitions formal by extending our semantics to support recursion.

Values to represent divergence. First, as explained in Example 26, with recursion our programs can have a new kind of behavior: they can diverge. To assign denotations to programs that diverge, we need to extend our semantic domains $[\![\tau]\!]$ to include values that represent divergence. For our numeric types $\tau_{num} \in \{\mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}\}$, we add a new **bottom element** $\bot$ to each domain $[\![\tau_{num}]\!]$. (We will revisit the question of how to represent divergence at other types shortly.) To extend these types with $\bot$, we can use the following construction. Given a quasi-Borel space $X = (E_X, M_X)$, we define $X_\bot$ as the quasi-Borel space whose underlying set is $E_X \cup \{\bot\}$, and whose random elements are

$$M_{X_\bot} := \left\{ \left(r \mapsto \begin{cases} \bot & \text{if } r \in A \\ \alpha(r) & \text{otherwise} \end{cases} \right) \middle| \, A \in \mathcal{B}(\mathbb{R}), \alpha \in M_X \right\}.$$

Scott continuity and the halting problem. Consider the function $\mathit{halts?} : \mathbb{R}_\bot \to \mathbb{B}_\bot$ that returns *true* if the input is a real number and *false* if the input is $\bot$. This is a quasi-Borel map, but we cannot allow it to be included in the semantic domain $[\![\mathbb{R} \to \mathbb{B}]\!]$; if we do, our semantic definitions will no longer be logically coherent. To see why, consider the program $\lambda f.\, \mu x.\textbf{if } f(x) \textbf{ then } \mu y.\, y \textbf{ else } 0$, of type $(\mathbb{R} \to \mathbb{B}) \to \mathbb{R}$. Our semantics should assign it some quasi-Borel map $(\mathbb{R}_\bot \Rightarrow \mathbb{B}_\bot) \to \mathbb{R}_\bot$. When we apply this map to *halts?*, what value $v \in E_{\mathbb{R}_\bot}$ should we get? Whatever the answer, since it is the result of evaluating a recursive expression $\mu x.e$, plugging it in for x in e must yield v again; that is, it must satisfy the *fixed-point equation*

$$v = \begin{cases} [\![\mu y.\, y]\!] & \text{if } \mathit{halts?}(v) \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \bot & \text{if } \mathit{halts?}(v) \\ 0 & \text{otherwise} \end{cases}.$$

But there can be no such v: if $v = \bot$, then $\mathit{halts?}(v)$ is *false*, and v must equal 0; if $v \in \mathbb{R}$, then $\mathit{halts?}(v)$ is *true*, and v must equal $\bot$. This is a version of the contradiction called the **halting problem**. In our setting, it is a sign that in order to have a well-defined semantics for recursion, we need to impose a requirement *beyond* measurability (or quasi-Borelness) on the denotations of our programs—a requirement that rules out denotations like *halts?*. That requirement is **Scott continuity**. Just as defining measurability required us to equip our spaces with σ-algebras and defining quasi-Borelness required us to equip our spaces with sets of random elements, defining Scott continuity also requires us to add some new structure to our semantic domains. Instead of quasi-Borel spaces, we will now interpret our types as **quasi-Borel predomains** (abbreviated as ω**-qbses**) [Vákár et al., 2019], which augment quasi-Borel spaces with **partial orders**.

Definition 9 (Quasi-Borel predomain)**.** A **quasi-Borel predomain** (ω**-qbs**) is a tuple

$X = (E_X, M_X, \leq_X)$, such that (E_X, M_X) is a quasi-Borel space, $\leq_X$ is a partial order on E_X, and the following properties are satisfied:

- **Closure of E_X under suprema.** Every increasing sequence $x_1 \leq_X x_2 \leq_X \ldots$ of elements of E_X has a **least upper bound** $\sup_{i\in\mathbb{N}} x_i \in E_X$.
- **Closure of M_X under suprema.** Every increasing sequence $\alpha_1 \leq_X \alpha_2 \leq_X \ldots$ of random elements in M_X, ordered pointwise (i.e., $\alpha_1 \leq_X \alpha_2$ iff $\alpha_1(r) \leq_X \alpha_2(r)$ for all $r \in \mathbb{R}$), has a least upper bound $\sup_{i\in\mathbb{N}} \alpha_i = (r \mapsto \sup_{i\in\mathbb{N}} \alpha_i(r)) \in M_X$.

Definition 10 (Pointed quasi-Borel predomain)**.** A **pointed quasi-Borel predomain** is a quasi-Borel predomain $X = (E_X, M_X, \leq_X)$ with a distinguished *bottom element* $\bot_X \in E_X$ such that $\forall x \in E_X, \bot_X \leq x$.

Definition 11 (Scott-continuous map)**.** For ω-qbses X and Y, a **Scott-continuous map** $f : X \to Y$ is a function $E_X \to E_Y$ that is **monotone** (for all $x_1 \leq_X x_2$, $f(x_1) \leq_Y f(x_2)$) and **preserves suprema** (for chains $x_1 \leq_X x_2 \leq_X \ldots$ in X, $f(\sup_{i\in\mathbb{N}} x_i) = \sup_{i\in\mathbb{N}} f(x_i)$).

Definition 12 (Scott-Borel map)**.** A **Scott-Borel map** $f : X \to Y$ between ω-qbses is a quasi-Borel map between (E_X, M_X) and (E_Y, M_Y) that is also Scott continuous.

Example 27 (Lifted quasi-Borel spaces)**.** Let $X = (E_X, M_X, \leq_X)$ be a quasi-Borel predomain. We define the pointed quasi-Borel predomain $X_\bot := (E_{X_\bot}, M_{X_\bot}, \leq_{X_\bot})$ by:

- $E_{X_\bot} = E_X \cup \{\bot\}$, where $\bot \notin E_X$ is a new element
- $M_{X_\bot} = \left\{ \left(r \mapsto \begin{cases} \bot & r \in A \\ \alpha(r) & r \notin A \end{cases} \right) \;\middle|\; A \in \mathcal{B}(\mathbb{R}), \alpha \in M_X \right\}$
- $x \leq_{X_\bot} y$ if and only if $x = \bot$ or $x \leq_X y$

Example 28 (Products of ω-qbses)**.** Consider two ω-qbses $X = (E_X, M_X, \leq_X)$ and $Y = (E_Y, M_Y, \leq_Y)$. We can then define the ω-qbs $X \times Y := (E_{X\times Y}, M_{X\times Y}, \leq_{X\times Y})$, where $(E_{X\times Y}, M_{X\times Y})$ is their product as qbses, and $(x, y) \leq_{X\times Y} (x', y')$ if and only if $x \leq_X x'$ and $y \leq_Y y'$. If X and Y are pointed, so is $X \times Y$, with bottom element $\bot_{X\times Y} = (\bot_X, \bot_Y)$.

Example 29 (Sums of ω-qbses)**.** Let $X_1, \ldots, X_n$ be ω-qbses. Then $\sum_{i=1}^n \ell_i X_i := (E_{\sum_i^n \ell_i X_i}, M_{\sum_i^n \ell_i X_i}, \leq_{\sum_i^n \ell_i X_n})$ is again an ω-qbs, where $(E_{\sum_i^n \ell_i X_i}, M_{\sum_i^n \ell_i X_i})$ is the quasi-Borel space sum (Example 16), and $\ell_j(v_1) \leq_{\sum_{i=1}^n \ell_i X_i} \ell_k(v_2)$ if and only if $j = k$ and $v_1 \leq_{X_j} v_2$.

Example 30 (Spaces of Scott-Borel maps)**.** Consider two ω-qbses $X = (E_X, M_X, \leq_X)$ and $Y = (E_Y, M_Y, \leq_Y)$. We define the ω-qbs $X \Rightarrow Y := (E_{X\Rightarrow Y}, M_{X\Rightarrow Y}, \leq_{X\Rightarrow Y})$, where:

- $E_{X\Rightarrow Y} = \{f \mid f : E_X \to E_Y \text{ and } f \text{ is Scott-Borel}\}$
- $M_{X\Rightarrow Y} = \{\alpha \mid \forall \alpha' \in M_X, (r \mapsto \alpha(r)(\alpha'(r))) \in M_Y\}$
- $f \leq_{X\Rightarrow Y} g$ if and only if $\forall x \in E_X, f(x) \leq_Y g(x)$

Note that if Y is pointed, so is $X \Rightarrow Y$, with bottom element $\bot_{X\Rightarrow Y} = (x \mapsto \bot_Y)$.

Definition 13 (ω-qbs of non-negative extended reals)**.** The **ω-qbs of non-negative extended reals** $\overline{\mathbb{R}}_{\geq 0}$ is ω-qbs $([0,\infty], M_{\overline{\mathbb{R}}_{\geq 0}}, \leq_{\overline{\mathbb{R}}_{\geq 0}})$, where:

- $M_{\overline{\mathbb{R}}_{\geq 0}} = \left\{ \left(r \mapsto \begin{cases} \infty & r \in A \\ \alpha(r) & r \notin A \end{cases} \right) \;\middle|\; A \in \mathcal{B}(\mathbb{R}), \alpha \in M_{\mathbb{R}_{\geq 0}} \right\}$
- $r \leq_{\overline{\mathbb{R}}_{\geq 0}} s$ if and only if $s = \infty$ or $r \leq s$.

This space is pointed, with bottom element $\bot_{\overline{\mathbb{R}}_{\geq 0}} = 0$.

Definition 14 (ω-quasi-Borel measures)**.** An **ω-quasi-Borel measure** on a quasi-Borel predomain X is either:

- a Scott-Borel map $\mu : (X \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \to \overline{\mathbb{R}}_{\geq 0}$, such that there exists $A \in \mathcal{B}(\mathbb{R})$ and $\alpha \in M_X$ with $\mu(f) = \int_A f(\alpha(r))\, dr$ for all Scott-Borel maps $f : X \to \overline{\mathbb{R}}_{\geq 0}$; or
- the least upper bound of a monotone chain of other ω-quasi-Borel measures, ordered pointwise.

ω-quasi-Borel probability measures and kernels are defined just as their quasi-Borel counterparts (Definitions 4 and 6). We adopt the same notation for integration with respect to μ as in Definitions 5, 7 and 8.

Example 31 (Subspaces of ω-qbses)**.** Let $X = (E_X, M_X, \leq_X)$ be a ω-qbs. If $E_Y \subseteq E_X$ is closed under suprema, then $Y = (E_Y, M_Y, \leq_Y)$ is again an ω-qbs, where:

- $M_Y = \{\alpha \mid \alpha \in M_X \text{ and } \alpha(r) \in E_Y \text{ for all } r \in \mathbb{R}\}$
- $y_1 \leq_Y y_2$ if and only if $y_1 \leq_X y_2$

If E_Y contains a least element $\bot_Y$ then Y is pointed.

Example 32 (Space of ω-quasi-Borel measures)**.** The ω-quasi-Borel measures on X form a pointed ω-qbs, Meas X, a subspace of $(X \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \overline{\mathbb{R}}_{\geq 0}$ with bottom element the **zero measure** $f \mapsto 0$.

Example 33 (Space of ω-quasi-Borel probability measures)**.** The ω-quasi-Borel probability measures on X form an ω-qbs, Prob X, a subspace of Meas X. If X is pointed, then Prob X is pointed, with bottom element $\bot_{\text{Prob } X} = \delta_{\bot_X} = (f \mapsto f(\bot_X))$.

2.3.2. Domain-theoretic semantics of recursive programs

We now redefine our semantics once again: each type is now interpreted as a pointed quasi-Borel predomain, and each expression is interpreted as a Scott-Borel map.

Semantics of types. We interpret numeric types using the lifted quasi-Borel predomains of Example 27, e.g. $\mathbb{N}_\bot$ and $\mathbb{R}_\bot$.[3] The unit type 1 is *not* augmented with

[3]Note that the associated orders have $\bot \leq x$ for all x, but otherwise distinct elements are incomparable. That is, we do *not* use the linear order on $\mathbb{R}$ or $\mathbb{N}$ to interpret these numeric types, but rather a flat order augmented with a bottom element $\bot$.

Listing 2.9 Semantics of the probabilistic λ-calculus with recursion.

ω-Qbs Semantics of Types

Type		*ω-qbs of values*
$[\![1]\!]$	=	1
$[\![\mathbb{N}]\!]$	=	$\mathbb{N}_\perp$
$[\![\mathbb{R}]\!]$	=	$\mathbb{R}_\perp$
$[\![\mathbb{R}_{[0,1]}]\!]$	=	$[0,1]_\perp \subseteq \mathbb{R}_\perp$
$[\![\mathbb{R}_{>0}]\!]$	=	$(0,\infty)_\perp \subseteq \mathbb{R}_\perp$
$[\![\tau_1 \times \tau_2]\!]$	=	$[\![\tau_1]\!] \times [\![\tau_2]\!]$
$[\![\sum_{i=1}^n \ell_i \tau_i]\!]$	=	$(\sum_{i=1}^n \ell_i [\![\tau_i]\!])_\perp$
$[\![\tau_1 \to \tau_2]\!]$	=	$[\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$
$[\![P\ \tau]\!]$	=	Meas $[\![\tau]\!]$
$[\![P_{lazy}\ \tau]\!]$	=	Prob $[\![\tau]\!]$

ω-Qbs Semantics of Contexts

Context		*ω-qbs of environments*
$[\![\cdot]\!]$	=	$(E = \{[]\},$
		$M = \{r \mapsto []\},$
		$\leq\ = \{([],[])\})$
$[\![\Gamma, x:\tau]\!]$	=	$(E = \{\gamma[x \mapsto v] \mid$
		$\gamma \in E_{[\![\Gamma]\!]}, v \in E_{[\![\tau]\!]}\},$
		$M = \{r \mapsto \alpha_1(r)[x \mapsto \alpha_2(r)] \mid$
		$\alpha_1 \in M_{[\![\Gamma]\!]}, \alpha_2 \in M_{[\![\tau]\!]}\},$
		$\leq\ = \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid$
		$v_1 \leq_{[\![\tau]\!]} v_2, \gamma_1 \leq_{[\![\Gamma]\!]} \gamma_2\})$

ω-Qbs Semantics of Expressions

Expression		*Value*
$[\![\Gamma \vdash \mathbf{return}_{lazy}\ e : P_{lazy}\ \tau]\!](\gamma)$	=	$\delta_{[\![e]\!](\gamma)}$
$[\![\Gamma \vdash \mathbf{do}_{lazy}\{e\} : P_{lazy}\ \tau]\!](\gamma)$	=	$[\![e]\!](\gamma)$
$[\![\Gamma \vdash \mathbf{do}_{lazy}\{x \leftarrow e; m\} : P_{lazy}\ \tau]\!](\gamma)$	=	$\int [\![\mathbf{do}_{lazy}\{m\}]\!](\gamma[x \leftarrow v])[\![e]\!](\gamma, dv)$
$[\![\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \{\ell_i x_i \mapsto e_i\}_{i=1}^n : \tau]\!](\gamma)$	=	$\begin{cases} \perp_{[\![\tau]\!]} & \text{if } [\![e]\!](\gamma) = \perp \\ [\![e_i]\!](\gamma[x_i \mapsto v]) & \text{if } [\![e]\!](\gamma) = \ell_i(v) \end{cases}$
$[\![\Gamma \vdash \mu x.\ e : \tau]\!](\gamma)$	=	$\sup_{i \in \mathbb{N}} v_i$, where
		$v_0 = \perp_{[\![\tau]\!]}$
		$v_{i+1} = [\![e]\!](\gamma[x \mapsto v_i])$

$\perp$: a lazy evaluation strategy never forces a value of unit type, so no program can distinguish between a diverging or a terminating computation at unit type. Note that 1 is still a pointed ω-qbs, with $\perp_1 = ()$.

To interpret product and function types, we use the corresponding constructions on pointed quasi-Borel predomains (Examples 28 and 30). Sum types $\ell_1\tau_1 + \cdots + \ell_n\tau_n$ are interpreted as *lifted* sums $(\ell_1[\![\tau_1]\!] + \cdots + \ell_n[\![\tau_n]\!])_\perp$.

The probabilistic types $P\ \tau$ and $P_{lazy}\ \tau$ are interpreted as spaces of ω-quasi-Borel measures (Examples 32 and 33): we choose $[\![P\ \tau]\!] = \text{Meas}\ [\![\tau]\!]$ and $[\![P_{lazy}\ \tau]\!] = \text{Prob}\ [\![\tau]\!]$. Recall that we intend these two types to model divergence differently. P models divergence strictly: divergence of e implies divergence of $\mathbf{do}\ \{x \leftarrow e; m\}$. P_{lazy} models divergence lazily: divergence of e does not cause $\mathbf{do}\ \{x \leftarrow e; m\}$ to diverge unless m needs the value of x. In Meas $[\![\tau]\!]$, divergence is represented by the zero measure $\mathbf{0}$. In Prob $[\![\tau]\!]$, by contrast, the bottom element is $\delta_{\perp_{[\![\tau]\!]}}$, the Dirac distribution at $\perp_{[\![\tau]\!]}$.

Plugging in each of these representations for e in the semantics of sequencing yields:

$$\begin{aligned}
[\![\mathbf{do}\ \{x \leftarrow e; m\}]\!](\gamma) &= \iint [\![\mathbf{do}\ \{m\}]\!](\gamma[x \mapsto v])\ \mathbf{0}(dv) = \mathbf{0} \\
[\![\mathbf{do}_{lazy}\ \{x \leftarrow e; m\}]\!](\gamma) &= \iint [\![\mathbf{do}_{lazy}\ \{m\}]\!](\gamma[x \mapsto v])\ \delta_\perp(dv) \\
&= [\![\mathbf{do}_{lazy}\ \{m\}]\!](\gamma[x \mapsto \perp])
\end{aligned}$$

In the first case, no matter whether m uses x or not, we get the $\mathbf{0}$ measure, representing divergence. In the second case, we do not necessarily get divergence: it depends on whether m forces x. Thus, the two spaces yield different interpretations of divergence, that capture precisely the behaviors we want to model.

Semantics of contexts and environments. The denotation of a context Γ is again a space of environments. Environments satisfying the assumptions of a given context Γ can be ordered variable-wise: $\gamma_1 \leq_{[\![\Gamma]\!]} \gamma_2$ if and only if $\gamma_1[x] \leq_{[\![\tau]\!]} \gamma_2[x]$ for all assumptions $(x : \tau) \in \Gamma$.

Semantics of expressions. Our denotational semantics of expressions has to change in two key ways. First, we need to encode the way that divergence propagates through a program: if a sub-computation diverges, does the whole computation diverge? Second, we need to give a semantics to recursive definitions.

For deterministic computations, we model a lazy operational semantics similar to Haskell's, so we avoid propagating divergence unless it is necessary. Our semantics of π_1, π_2, and of function application are the same as before; they may return a value even if a subexpression evaluates to $\perp$. But the **match** construct does force the evaluation of its scrutinee, leading to a new rule:

$$[\![\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \{\ell_i x_i \mapsto e_i\}_{i=1}^n : \tau]\!](\gamma) = \begin{cases} \perp_{[\![\tau]\!]} & \text{if } [\![e]\!](\gamma) = \perp \\ [\![e_i]\!](\gamma[x_i \mapsto v]) & \text{if } [\![e]\!](\gamma) = \ell_i(v) \end{cases}.$$

It is also the case that most of our primitive operations force the values of their arguments. For example, the denotation of + is now revised to handle the case where one or both of its arguments are $\perp$:

$$[\![+]\!] = (x, y) \mapsto \begin{cases} \perp & \text{if } x = \perp \text{ or } y = \perp \\ x + y & \text{otherwise} \end{cases}.$$

Semantics of recursion. To assign a denotation to the construct $\mu x.\ e$, we compute the least upper bound of an infinite chain of denotations:

$$[\![\mu x.\ e]\!](\gamma) = \sup_{i \in \mathbb{N}} v_i,$$

where $v_0 = \bot_{[\![\tau]\!]}$ and $v_{i+1} = [\![e]\!](x \mapsto v_i)$ for all $i \in \mathbb{N}$. This is well-defined because the v_i form an increasing chain in $[\![\tau]\!]$. To see why, note that $[\![e]\!]$ is Scott-Borel and thus is monotone. The start of the chain v_0 is the bottom element and so is necessarily less than or equal to v_1. By monotonicity, this implies that $v_1 = [\![e]\!](x \mapsto v_0) \leq_{[\![\tau]\!]} [\![e]\!](x \mapsto v_1) = v_2$. This argument can be repeated to show that $v_i \leq_{[\![\tau]\!]} v_{i+1}$ for all $i \in \mathbb{N}$.[4]

Scott continuity implies that $[\![e]\!](x \mapsto v) = [\![e]\!](\sup_{i\in\mathbb{N}} v_i) = \sup_{i\in\mathbb{N}} [\![e]\!](x \mapsto v_i) = v$, so v is a fixed point of $v \mapsto [\![e]\!](x \mapsto v)$. Indeed, it is the **least fixed point** of this map: the smallest element $v \in E_{[\![\tau]\!]}$ that satisfies the fixed-point equation $v = [\![e]\!](x \mapsto v)$.

Remark 12 (Intuition for semantics of recursion)**.** One intuition is that each value v_i in the chain $\bot, [\![e]\!](x \mapsto \bot), [\![e]\!](x \mapsto [\![e]\!](x \mapsto \bot)), \ldots$ represents the denotation of the program $\mu x.\, e$ under a fixed *recursion limit* i, after which the program is assumed to diverge. For example, consider the factorial function from Example 24. Its denotation is the supremum of the chain

$$\begin{aligned} v_0 &= \bot_{\mathbb{N}_\bot \to \mathbb{N}_\bot} = \lambda n.\, \bot \\ v_1 &= \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times v_0(n-1) \\ v_2 &= \lambda n.\, \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times v_1(n-1) \\ &\vdots \end{aligned}$$

Each v_i gives the correct factorial for arguments $n < i$, and diverges if the recursion limit i is exceeded. As $i \to \infty$, v_i converges to the actual factorial function, defined to equal $n!$ on all inputs n.

Remark 13 (Recursive definitions of probabilistic programs)**.** When we define a recursive *function*, each v_i allows the function to terminate on more inputs than v_{i-1} did. When we define a recursive probabilistic program, each v_i instead *accumulates more mass* on each *output* than v_{i-1} did. For example, consider the geometric distribution from Example 25. Its denotation is the supremum of the chain

$$\begin{aligned} v_0 &= \bot_{\text{Meas } \mathbb{N}_\bot} = \mathbf{0} = (f \mapsto 0) \\ v_1 &= 0.5 \cdot \delta_0 + 0.5 \cdot (x \mapsto x+1)_* v_0 = 0.5 \cdot \delta_0 = (f \mapsto 0.5 \cdot f(0)) \\ v_2 &= 0.5 \cdot \delta_0 + 0.5 \cdot (x \mapsto x+1)_* v_1 = 0.5 \cdot \delta_0 + 0.25 \cdot \delta_1 = (f \mapsto 0.5 \cdot f(0) + 0.25 \cdot f(1)) \\ &\vdots \end{aligned}$$

Each v_i represents a *subprobability measure* on $\mathbb{N}_\bot$, assigning mass 0.5^k to the outcome $k-1$, from $k = 1$ to i. Divergence is represented semantically as "missing mass" from the distribution. As $i \to \infty$, the probability of divergence 0.5^i approaches 0, and v_i converges to the complete geometric distribution.

This program denotes the geometric distribution in both the strict and the lazy

[4] Note that monotonicity *does not imply* that $v \leq_{[\![\tau]\!]} [\![e]\!](x \mapsto v)$ for all v. The argument relies on the fact that we start the chain with v_0, which is less than v_1 because it is less than everything.

semantics (P and P_{lazy}), but the particular chain of denotations whose fixed point we take is different in the two cases. With P_{lazy}, the ambient space is Prob $\mathbb{N}_\perp$, which does not contain subdistributions, only full probability distributions. The chain of denotations becomes:

$$\begin{aligned}
v_0 &= \bot_{\text{Prob } \mathbb{N}_\perp} = \delta_\perp = (f \mapsto f(\perp)) \\
v_1 &= 0.5 \cdot \delta_0 + 0.5 \cdot (x \mapsto x+1)_* v_0 = (f \mapsto 0.5 \cdot f(0) + 0.5 \cdot f(\perp)) \\
v_2 &= 0.5 \cdot \delta_0 + 0.5 \cdot (x \mapsto x+1)_* v_1 = (f \mapsto 0.5 \cdot f(0) + 0.25 \cdot f(1) + 0.25 \cdot f(\perp)) \\
&\vdots
\end{aligned}$$

Note that in Prob, divergence is represented as mass assigned to the bottom element $\perp$, rather than "missing mass" as in Meas.

Remark 14 (Mutual recursion)**.** Mutually recursive definitions can be made using tuples. For example, the functions *isEven* and *isOdd* can be defined as the two components of the tuple

$$\mu fs.\ (\lambda n.\ n = 0 \vee \pi_2(fs)(n-1), \lambda n. \neg(n = 0 \vee \pi_1(fs)(n))).$$

2.4 Probabilistic programming with recursive types

Our final extension to our calculus is the addition of **recursive types**. We extend our grammar of types to include **type variables** α, and a binder μ for defining types that refer to themselves:

$$\tau ::= 1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \tau_1 \times \tau_2 \mid \textstyle\sum_{i=1}^n \ell_i \tau_i \mid \tau_1 \to \tau_2 \mid P\ \tau \mid P_{lazy}\ \tau \mid \alpha \mid \mu\alpha.\tau$$

The expression $\mu\alpha.\ \tau$ binds the type variable α for use in the type expression τ, where it can be used to refer recursively to the type being defined. We add two new expressions to our language for building and consuming values of recursive types:

$$\textsc{Unroll}\ \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \textbf{unroll}\ e : \tau[\alpha \mapsto \mu\alpha.\tau]} \qquad \textsc{Roll}\ \frac{\Gamma \vdash e : \tau[\alpha \mapsto \mu\alpha.\tau]}{\Gamma \vdash \textbf{roll}\ e : \mu\alpha.\tau}$$

In these rules, the notation $\tau[\alpha \mapsto \mu\alpha.\tau]$ refers to the result of substituting $\mu\alpha.\tau$ for all free occurrences of the type variable α in the type expression τ.

Example 34 (Lazy streams)**.** The type $\mu\alpha.\ \mathbb{R} \times \alpha$ is one way to encode a type of infinite streams of real numbers. Informally, a value of this type is always a pair, whose first element is a real number (the head of the stream) and whose second element is another stream (the tail of the stream). Actually building such an infinite stream requires a recursive definition. For example, the infinite stream of all natural

Listing 2.10 Well-formedness rules for types with type variables.

$$\textsc{T-Ground}\ \frac{\tau \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}\}}{\Delta \vdash \tau \text{ type}} \qquad \textsc{T-Arrow}\ \frac{\Delta \vdash \tau_1 \text{ type} \qquad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \to \tau_2 \text{ type}}$$

$$\textsc{T-Prod}\ \frac{\Delta \vdash \tau_1 \text{ type} \qquad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \times \tau_2 \text{ type}} \qquad \textsc{T-Sum}\ \frac{\Delta \vdash \tau_1 \text{ type} \qquad \dots \qquad \Delta \vdash \tau_n \text{ type}}{\Delta \vdash \ell_1 \tau_1 + \dots + \ell_n \tau_n \text{ type}}$$

$$\textsc{T-Prob}\ \frac{\Delta \vdash \tau \text{ type}}{\Delta \vdash P\ \tau \text{ type}} \qquad \textsc{T-Prob-Lazy}\ \frac{\Delta \vdash \tau \text{ type}}{\Delta \vdash P_{lazy}\ \tau \text{ type}} \qquad \textsc{T-Var}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha \text{ type}} \qquad \textsc{T-Rec}\ \frac{\Delta, \alpha \vdash \tau \text{ type}}{\Delta \vdash \mu\alpha.\tau \text{ type}}$$

numbers can be defined as follows:

$$\begin{aligned} \mathit{nats} &:= \mu s.\ \textbf{roll}\ (0, \mathit{mapStream}\ (\lambda r.\ r+1)\ s) \\ \mathit{mapStream} &:= \mu g.\lambda f.\lambda s.\ \textbf{roll}\ (f(\pi_1(\textbf{unroll}\ s)), g\ f\ (\pi_2(\textbf{unroll}\ s))) \end{aligned}$$

Example 35 (Inductive type of natural numbers)**.** The type $\mu\alpha.\mathit{zero}\ 1 + \mathit{succ}\ \alpha$ is an inductively defined type of natural numbers: values can either be $\mathit{zero}()$ or $\mathit{succ}(n)$ for some n of the same type.

2.4.1. Type contexts and types as functors

A **type context** Δ is a list of type variables α. We say a type expression τ is **well-formed** with respect to a type context Δ if Δ contains all the type variables that appear free in τ. We write $\Delta \vdash \tau$ type for the judgment that τ is well-formed with respect to Δ; the rules for this judgment are given in 2.10.

A **type assignment** $\vec{X}$ for a type context Δ is a list of pointed ω-qbses, with elements X_α for each $\alpha \in \Delta$. If a type τ is well-formed in in a type context Δ, then for each type assignment $\vec{X}$ for Δ, we obtain a different denotation $[\![\tau]\!](\vec{X})$ for the type expression. For example, for $\tau = (\alpha_1 \times \mathbb{N}) \to \alpha_2$, which is well-formed in $\Delta = (\alpha_1, \alpha_2)$, we have $[\![\tau]\!](X_1, X_2) = (X_1 \times \mathbb{N}_\perp) \Rightarrow X_2$. Listing 2.11 gives these denotations for the type expressions in our language. Note that when Δ is empty (so τ has no type variables), these semantics exactly coincide with those from Listing 2.9.

A value in $[\![\tau]\!](\vec{X})$ can be systematically transformed into a value in $[\![\tau]\!](\vec{Y})$ for some alternative type assignment $\vec{Y}$, if we have maps $\delta^+_\alpha : X_\alpha \to Y_\alpha$ and $\delta^-_\alpha : Y_\alpha \to X_\alpha$ for each $\alpha \in \Delta$. We need maps in both directions because type variables α can appear both *positively* and *negatively* in τ. For example, consider the type expression $\tau = \alpha_1 \to \alpha_2$. If we have a value $f \in [\![\tau]\!](X_1, X_2) = X_1 \Rightarrow X_2$ and wish to transform

Listing 2.11 Functorial semantics for types with type variables.

Type		**ω-Qbs of values**	**Functorial action on maps**
$\Delta \vdash \tau$		$[\![\tau]\!]((X_\alpha)_{\alpha\in\Delta})$	$[\![\tau]\!](\vec{X}) \xrightarrow{[\![\tau]\!]((\delta_\alpha^-:Y_\alpha\to X_\alpha)_{\alpha\in\Delta},(\delta_\alpha^+:X_\alpha\to Y_\alpha)_{\alpha\in\Delta})} [\![\tau]\!](\vec{Y})$
$[\![1]\!](\vec{X})$	$=$	1	id_1
$[\![\mathbb{N}]\!](\vec{X})$	$=$	$\mathbb{N}_\perp$	$\mathrm{id}_{\mathbb{N}_\perp}$
$[\![\mathbb{R}]\!](\vec{X})$	$=$	$\mathbb{R}_\perp$	$\mathrm{id}_{\mathbb{R}_\perp}$
$[\![\mathbb{R}_{[0,1]}]\!](\vec{X})$	$=$	$[0,1]_\perp \subseteq \mathbb{R}_\perp$	$\mathrm{id}_{[0,1]_\perp}$
$[\![\mathbb{R}_{>0}]\!](\vec{X})$	$=$	$(0,\infty)_\perp \subseteq \mathbb{R}_\perp$	$\mathrm{id}_{(0,\infty)_\perp}$
$[\![\alpha]\!](\vec{X})$	$=$	X_α	δ_α^+
$[\![\tau_1 \times \tau_2]\!](\vec{X})$	$=$	$[\![\tau_1]\!](\vec{X}) \times [\![\tau_2]\!](\vec{X})$	$\langle [\![\tau_1]\!](\vec{\delta^-},\vec{\delta^+}), [\![\tau_2]\!](\vec{\delta^-},\vec{\delta^+})\rangle$
$[\![\sum_{i=1}^n \ell_i\tau_i]\!](\vec{X})$	$=$	$(\sum_{i=1}^n \ell_i [\![\tau_i]\!](\vec{X}))_\perp$	$x \mapsto \begin{cases} \perp & x = \perp \\ \ell_i\,([\![\tau_i]\!](\vec{\delta^-},\vec{\delta^+})(v)) & x = \ell_i\, v \end{cases}$
$[\![\tau_1 \to \tau_2]\!](\vec{X})$	$=$	$[\![\tau_1]\!](\vec{X}) \Rightarrow [\![\tau_2]\!](\vec{X})$	$f \mapsto [\![\tau_2]\!](\vec{\delta^-},\vec{\delta^+}) \circ f \circ [\![\tau_1]\!](\vec{\delta^+},\vec{\delta^-})$
$[\![P\ \tau]\!](\vec{X})$	$=$	$\mathrm{Meas}\ [\![\tau]\!](\vec{X})$	$p \mapsto [\![\tau]\!](\vec{\delta^-},\vec{\delta^+})_* p$
$[\![P_{lazy}\ \tau]\!](\vec{X})$	$=$	$\mathrm{Prob}\ [\![\tau]\!](\vec{X})$	$p \mapsto [\![\tau]\!](\vec{\delta^-},\vec{\delta^+})_* p$
$[\![\mu\alpha.\ \tau]\!](\vec{X})$	$=$	$\mu X_\alpha.[\![\tau]\!]((\vec{X},X_\alpha))$	$\pi_2\left(\mu\delta_\alpha^{\mp}.\begin{pmatrix}\mathbf{map}([\![\tau]\!]((\vec{\delta^+},\pi_2\delta_\alpha^{\mp}),(\vec{\delta^-},\pi_1\delta_\alpha^{\mp}))),\\ \mathbf{map}([\![\tau]\!]((\vec{\delta^-},\pi_1\delta_\alpha^{\mp}),(\vec{\delta^+},\pi_2\delta_\alpha^{\mp})))\end{pmatrix}\right)$ where $\mathbf{map}(f) \coloneqq \mathbf{roll} \circ f \circ \mathbf{unroll}$

it into $g \in [\![\tau]\!](Y_1,Y_2) = Y_1 \Rightarrow Y_2$, we need a way to transform the inputs $y \in Y_1$ into inputs in X (for which we use $\delta_{\alpha_1}^- : Y_1 \to X_1$), and a way to transform the outputs $f(\delta_{\alpha_1}^-(y))$ back into outputs in Y (for which we use $\delta_{\alpha_2}^+ : X_2 \to Y_2$). That is, $g = \delta_{\alpha_2}^+ \circ f \circ \delta_{\alpha_1}^-$. The map transforming f into g is written $[\![\alpha_1 \to \alpha_2]\!](\vec{\delta^-},\vec{\delta^+})$; more generally, we write $[\![\tau]\!](\vec{\delta^-},\vec{\delta^+}) : [\![\tau]\!](\vec{X}) \to [\![\tau]\!](\vec{Y})$ for the function that systematically transforms values of type $[\![\tau]\!](\vec{X})$ into values of type $[\![\tau]\!](\vec{Y})$ using $\vec{\delta^+}$ and $\vec{\delta^-}$. The rules defining this map for general τ are given in the right column of Listing 2.11.

Remark 15. Category theoretically, each well-formed type expression actually corresponds to a **mixed-variance functor** $F_\tau : (\omega\mathbf{Qbs}^{\mathrm{op}})^{|\Delta|} \times \omega\mathbf{Qbs}^{|\Delta|} \to \omega\mathbf{Qbs}$, and we write $[\![\tau]\!](\vec{X})$ for $F_\tau(\vec{X},\vec{X})$, and $[\![\tau]\!](\vec{\delta^-},\vec{\delta^+})$ for $F_\tau(\vec{\delta^-},\vec{\delta^+})$.

2.4.2. Recursive types

Now consider a well-formed type expression τ in the context Δ,α. Given a type assignment $\vec{X}$ for Δ (not including α), we can define the *sequence* of ω-Qbses $Y_0, Y_1, \ldots,$

where

$$Y_0 = \{\bot\}$$
$$Y_{i+1} = [\![\tau]\!]((\vec{X}, Y_i))$$

Between each Y_i and Y_{i+1} we can define an **embedding-projection pair** $e : Y_i \to Y_{i+1}$, $p : Y_{i+1} \to Y_i$, satisfying $p(e(y_i)) = y_i$ for all $y_i \in E_{Y_i}$ and $e(p(y_{i+1})) \leq_{Y_{i+1}} y_{i+1}$ for all $y_{i+1} \in E_{Y_{i+1}}$. We do so inductively:

$$\begin{array}{rcl} e_0 & = & y_0 \mapsto \bot_{Y_1} \\ e_{i+1} & = & [\![\tau]\!]((\vec{\mathrm{id}}, p_i), (\vec{\mathrm{id}}, e_i)) \end{array} \qquad \begin{array}{rcl} p_0 & = & y_1 \mapsto \bot_{Y_0} \\ p_{i+1} & = & [\![\tau]\!]((\vec{\mathrm{id}}, e_i), (\vec{\mathrm{id}}, p_i)) \end{array}$$

where $\vec{\mathrm{id}} = (\mathrm{id}_{X_\alpha})_{\alpha \in \Delta}$. Using these, we can take the **bilimit** of our sequence of spaces,[5] $Y := \mu\alpha.\, [\![\tau]\!](\vec{X}, \alpha) = (E_Y, M_Y, \leq_Y)$, where:

- $E_Y := \{(y_0, y_1, \dots) \mid \forall i \in \mathbb{N}, y_i \in Y_i, y_i = p(y_{i+1})\}$
- $M_Y := \{\alpha \mid \forall i \in \mathbb{N}, \pi_i \circ \alpha \in M_{Y_i}\}$
- $y \leq_Y y'$ if and only if $y_i \leq_{Y_i} y'_i$ for all $i \in \mathbb{N}$.

Note that this is a pointed space: the bottom element is $\bot_Y = (\bot_{Y_0}, \bot_{Y_1}, \dots)$. It is also closed under suprema: we can take the suprema elementwise, and the consistency condition $y_i = p(y_{i+1})$ will still hold, because p is Scott continuous.

What makes this a good interpretation for our recursive types $\mu\alpha.\, \tau$ is that there is an isomorphism $\mathbf{roll} : [\![\tau]\!]((\vec{X}, Y)) \to Y$, i.e. the space Y is isomorphic to the interpretation of $\tau[\alpha \mapsto Y]$, and thus Y is a fixed point of the up-to-isomorphism domain equation $Y \cong [\![\tau]\!]((\vec{X}, Y))$.

To define the isomorphism concretely, first note that between each Y_i and Y we also have an embedding-projection pair, $e_i^Y : Y_i \to Y$ and $p_i^Y : Y \to Y_i$. The projections $p_i^Y := \pi_i$ just project out the appropriate element of the infinite tuple. The embeddings are of the form

$$e_i^Y(y_i) = (p_{i \to 0}(y_i), p_{i \to 1}(y_i), \dots, y_i, e_{i \to i+1}(y_i), \dots),$$

where $p_{i \to j} = p_j \circ \cdots \circ p_{i-1}$ for $j < i$ and $e_{i \to j} = e_{j-1} \circ \cdots \circ e_i$ for $i < j$. Then the isomorphism **roll**, and its inverse **unroll**, can be defined as follows:

$$\mathbf{roll}(v) := (\bot, [\![\tau]\!]((\vec{\mathrm{id}}, e_0^Y), (\vec{\mathrm{id}}, p_0^Y))(v), [\![\tau]\!]((\vec{\mathrm{id}}, e_1^Y), (\vec{\mathrm{id}}, p_1^Y))(v), \dots)$$
$$\mathbf{unroll}(y_0, y_1, \dots) := \sup_{i \in \mathbb{N}} [\![\tau]\!]((\vec{\mathrm{id}}, p_i^Y), (\vec{\mathrm{id}}, e_i^Y))(y_{i+1})$$

[5] The bilimit is simultaneously the limit of the diagram $Y_0 \xleftarrow{p_0} Y_1 \xleftarrow{p_1} Y_2 \xleftarrow{p_2} \cdots$ and the colimit of the diagram $Y_0 \xrightarrow{e_0} Y_1 \xrightarrow{e_1} Y_2 \xrightarrow{e_2} \cdots$.

Part II

Transforming Probabilistic Programs

3

INTEGRALS AND UNBIASED ESTIMATES

Together, this chapter and Chapter 5 constitute a significant reformulation and extension of the research originally published in Lew et al. [2023b], which was done jointly with Mathieu Huot, Sam Staton, and Vikash Mansinghka.

3.1 Motivation and overview

Probabilistic programs offer a formal language for describing complex generative processes and probability distributions. A fundamental task when analyzing such models is to characterize their behavior via *integration*. Given a probabilistic program defining a measure μ on a space X, we can learn a lot about μ by computing the integrals of functions $g : X \to \mathbb{R}$ with respect to μ. When μ is a probability measure, these integrals represent **expected values** $\mathbb{E}_\mu[g] = \int g(x)\mu(dx)$.

Most directly, integrals characterize the *average behavior* of a probability distribution, as observed by some function. (More formally, $\mathbb{E}_\mu[g]$ is the constant that the sample averages $\frac{1}{n}\sum_{i=1}^n g(x_i)$ converge to as $n \to \infty$, for $x_i \sim \mu$.) But integration with respect to a measure can reveal much more about the distribution than its mean. For instance, the probability of an event $A \subseteq X$ is simply the expected value of its indicator function, $\mathbb{P}(A) = \mathbb{E}_\mu[\mathbf{1}_A]$. Computing such probabilities is crucial in areas like reliability analysis or *rare event estimation*, where the goal is to characterize the tail behavior of a distribution by computing probabilities of unlikely events. Furthermore, many higher-order statistical properties can be computed as *functions of* integrals; the variance of a function, $\mathbb{V}_\mu[g] = \mathbb{E}_\mu[g^2] - (\mathbb{E}_\mu[g])^2$, is a prime example. Thus, a general mechanism for computing integrals with respect to probabilistic programs would provide a powerful and general tool for probing their behavior.

Intractability of integration. Unfortunately, computing these integrals exactly is generally intractable. The measures defined by probabilistic programs can involve complex dependencies, continuous state spaces, and infinite support, leading to

integrals or sums that lack closed-form solutions or are computationally prohibitive to evaluate directly. The standard alternative, naïve Monte Carlo estimation (sampling $x_i \sim \mu$ and averaging $g(x_i)$), suffers from significant drawbacks:

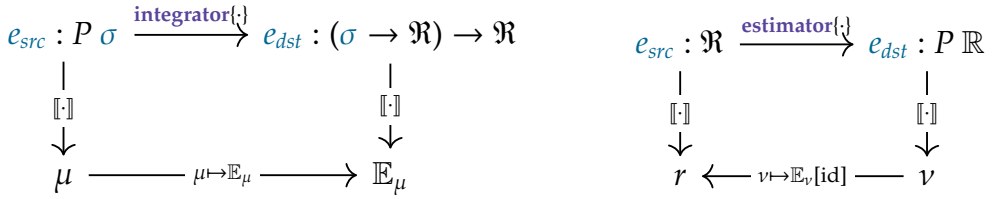
- **Variance.** The estimator's *variance* can be high, requiring an impractically large number of samples for acceptable accuracy. This is especially true when the magnitude of g can be large even for inputs that are rare under μ.
- **Bias under composition.** This approach can also introduce significant bias when estimating composite quantities. For example, estimating $\mathbb{V}_\mu[g]$ by the squaring the sample mean of $g(x_i)$, or estimating $e^{\mathbb{E}_\mu[g]}$ by the sample mean of $e^{g(x_i)}$, yields biased results due to non-linearities applied outside the expectation.

Approach. This chapter presents a framework for automatically deriving *provably unbiased estimators* for integrals (and functions of integrals) with respect to measures defined as probabilistic programs, addressing both intractability and the challenges of composition. Our core contribution is a pair of program transformations, drawing on several techniques from the programming language literature:

1. **Automatic integration (integrator{·}):** We first employ a continuation-passing style transformation to compile a probabilistic program e (denoting μ) into a deterministic program integrator{e}. This target program represents the mathematical *expectation operator* $\mathbb{E}_\mu : (X \to \Re) \to \Re$. Because the exact integral value may be infinite or intractable, its result type $\Re$ (denoting the space of extended reals) is treated as *opaque*—representing the true value symbolically without necessarily computing it.
2. **Automatic unbiased estimation (estimator{·}):** We then use *higher-order operator overloading* to transform a deterministic program operating on opaque $\Re$ values (like those produced by integrator{·}) into a new *probabilistic program*. This resulting program, when run, generates samples from a distribution whose mean is precisely the original opaque value. That is, our transformation synthesizes an unbiased estimator.

Critically, there is often more than one way to construct an unbiased estimator for a given integral, leading to different variance and computational cost characteristics. For example, the expectation $\mathbb{E}_{x\sim\mu}[g(x) + h(x)]$ can be estimated by sampling $x \sim \mu$ and returning $g(x) + h(x)$, or by independently estimating $\mathbb{E}[g]$ and $\mathbb{E}[h]$ and summing the results, or even by randomly choosing to estimate either $\mathbb{E}[g]$ or $\mathbb{E}[h]$ and appropriately re-weighting. Our estimator{·} transformation handles this by defining overloaded implementations for primitive operations (like $+_\Re$, $\times_\Re$, or primitive expectation operators like $\mathbf{E}_{normal}$). Each primitive can have multiple estimation strategies associated with it (e.g., `SUM` vs. `SAMPLE` for $+_\Re$). Users can guide the selection of these strategies via lightweight annotations, allowing them to rapidly explore the combinatorial space of possible unbiased estimators and tailor the variance-cost trade-off for their specific application.

The framework developed in this chapter serves as a crucial foundation for the rest



(a) The **integrator**{·} transformation (b) The **estimator**{·} transformation

Figure 3-1: Commutative diagrams illustrating the two program transformations developed in this chapter. (a) The **integrator**{·} transformation compiles a source program e_{src} into a target program e_{dst}. The denotation $[\![e_{dst}]\!]$ is the true expectation operator $\mathbb{E}_\mu$ corresponding to the source measure $\mu = [\![e_{src}]\!]$. (b) The **estimator**{·} transformation compiles a program e_{src} denoting an intractable value r into an program e_{dst} that unbiasedly estimates r (i.e., for $\nu = [\![e_{dst}]\!]$, $\mathbb{E}_\nu[\mathrm{id}] = r$).

of this thesis. Chapter 4 will show how to automatically transform probabilistic programs into integral expressions representing their Radon-Nikodym derivatives (i.e., ratios of probability densities). Chapter 5 will similarly show how gradients of expected values and densities—essential in reinforcement learning, variational inference, and sensitivity analysis—can also be formulated as integrals in our framework. A significant advantage of this approach is that because densities and gradients are ultimately expressed as (potentially complex, nested) integrals, the **estimator**{·} transformation developed here can be directly applied to obtain unbiased estimators for them. Such unbiased estimators can be plugged soundly into algorithms like stochastic gradient descent, Markov chain Monte Carlo, importance sampling, and sequential Monte Carlo, enabling the application of probabilistic programming to a wide range of inference and optimization tasks. Collectively, these techniques provide a powerful, compositional, and provably sound framework for reasoning about and estimating the intractable integrals, densities, and gradients that arise ubiquitously in probabilistic modeling.

Correctness and technical challenges. To reason about the correctness of our transformations, we develop **logical relations** that specify their expected correctness properties in a type-directed way, and inductive proofs that our transformations satisfy these properties for all programs in our language. Although these proofs are simple for unsigned integrands and non-recursive programs, they require non-trivial extensions to cover our full language:

- **Recursion:** For our correctness proofs to apply to recursive probabilistic programs and recursive integral equations, our logical relations need to be **admissible**, and naïvely, ours are not. One intuition for what goes wrong is as follows. A standard proof technique for reasoning about recursion involves showing that, for every *recursion depth limit*, the desired correctness property holds for a depth-truncated source program and the corresponding depth-truncated target program. But when we transform a recursive integral expression into a recursive unbiased estimator, this may not be the case: if we expand the source integral expression up

to some depth limit, we may find that we need to expand the target estimator to a larger depth limit for it to unbiasedly estimate the value of the truncated source expression. In Section 3.4, we develop a more sophisticated approach that allows the two programs to evolve out of sync as the recursion depth limit is increased.

- **Signed integrands:** As recursive probabilistic programs are permitted to execute for longer, they accumulate more mass onto their possible outputs, and integrals of non-negative functions increase monotonically. Our proofs rely on this property. In the presence of signed integrands, this monotonicity no longer holds. In Section 3.5, we extend our approach to handle arbitrary real-valued integrands and their potentially undefined integrals, by separately tracking the positive and negative values that our estimators may return.

- **First-class (nestable) integration and estimation:** In Section 3.6, we show how integration and unbiased estimation can be added as first-class language constructs, rather than as external program transformations. This allows users to nest estimation and integration, for example, to estimate the variance of an automatically derived estimator. However, this change requires understanding the type $\Re$ as an infinite recursive type, whose inhabitants not only represent a value, but also a way to unbiasedly estimate it, to take integrals with respect to that unbiased estimator, to estimate those integrals, to take integrals with respect to *those* estimators, and so on. Reasoning about correctness then requires the use of recursively defined logical relations on recursively defined spaces, for which we adapt a technique introduced by Pitts [1996].

Together, these innovations allow for first-class integration and unbiased estimation with correctness guarantees in a very expressive language.

3.2 Automatic integration with continuations

Our goal in this section is to automatically compile a probabilistic program e, denoting a measure μ, into a new program **integrator**$\{e\}$ that implements the **expectation operator** $\mathbb{E}_\mu$. To start, we will consider the probabilistic λ-calculus *without* recursion, and integrals of only *non-negative* functions. However, we will lift both these restrictions in the remainder of the chapter.

As discussed in the previous section, an immediate challenge is that our language can express many measures μ and functions f for which $\mathbb{E}_\mu[f]$ is intractable to compute exactly. This fact motivates two key design decisions:

- First, in this section, we add a new type $\Re_{\geq 0}$ to our language to represent the results of (possibly intractable) integrals. Because the results of integrals may be infinite, we take $[\![\Re_{\geq 0}]\!] = \overline{\mathbb{R}}_{\geq 0}$, the space $[0, \infty]$ of extended non-negative reals. And because the results of integrals may be intractable to compute, in our implementation this type is **opaque**: evaluating a program of type $\Re_{\geq 0}$ will print "<`value of type` $\Re_{\geq 0}$>", but give no information as to what the particular value is. Semantically,

Listing 3.1 Grammar of $\lambda_{\mathfrak{R}}$, extending the probabilistic λ-calculus from Listing 2.4. New productions are highlighted.

Syntactic category		Productions
Types τ	::=	$1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \mathfrak{R}^n \mid \tau_1 \times \tau_2 \mid \ell_1 \tau_1 + \cdots + \ell_n \tau_n \mid \tau_1 \to \tau_2 \mid P\,\tau$
Expressions e	::=	$x \mid c \mid f \mid (e_1, e_2) \mid \pi_1\, e \mid \pi_2\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid \ell_i\, e \mid$ **match** e **with** $\{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} \mid$ **return** $e \mid$ **do** $\{m\}$
Blocks m	::=	$e \mid x \leftarrow e; m$
Constants c	::=	$() \mid n \mid r$
Built-ins f_{det}	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \ldots$
f_{prb}	::=	$flip \mid normal \mid uniform \mid \ldots$
f_{ext}	::=	$cast_{\mathbb{R}\to\mathfrak{R}} \mid +_{\mathfrak{R}} \mid -_{\mathfrak{R}} \mid \times_{\mathfrak{R}} \mid exp_{\mathfrak{R}} \mid sum_{\infty} \mid \mathbb{E}_{f_{prb}} \mid \ldots$
Variable names x	$\in$	$\Sigma^* \setminus \{match, with, exp, \ldots\}$
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$

however, we assign every program of type $\mathfrak{R}_{\geq 0}$ a well-defined denotation: there is an *intended result* of the computation, even if we cannot compute it exactly. The goal of the **integrator** transformation is to compositionally build programs that, if we could tractably execute them, would return correct integrals.

- Then, in the next section, we will define a new program transormation **estimator**$\{\cdot\}$ that transforms *deterministic* programs returning an opaque $\mathfrak{R}_{\geq 0}$ value (like the one produced by **integrator**$\{\cdot\}$) into a *probabilistic* program. This new program, when executed, produces random samples whose expected value is precisely the original, intractable $\mathfrak{R}_{\geq 0}$ value. That is, it computes an *unbiased estimate*.

By composing these two transformations, users can automatically derive probabilistic programs that compute unbiased estimates of integrals—or even *functions of* one or more integrals—with respect to arbitrary user-specified measures.

3.2.1. Syntactic extensions to the core language

We extend the probabilistic λ-calculus (Listing 2.4) with new types and primitives:

- For each natural number $n \in \mathbb{N}$, we add a type $\mathfrak{R}^n_{\geq 0}$ to our language, to represent the results of integrals of non-negative vector-valued functions. Semantically, $[\![\mathfrak{R}^n_{\geq 0}]\!] = \overline{\mathbb{R}}^n_{\geq 0}$, the space of n-dimensional vectors of non-negative extended reals. We need to use extended reals here because integrals can be infinite. When $n = 1$, we omit the superscript.
- We add new primitives for computing with values of type $\mathfrak{R}^n_{\geq 0}$. This includes primitives for casting ordinary reals as values of type $\mathfrak{R}^n_{\geq 0}$ ($cast_{\mathbb{R}\to\mathfrak{R}}$), and for arithmetic operations ($+_{\mathfrak{R}}, \times_{\mathfrak{R}}, \ldots$) on these values. Because $\mathfrak{R}^n_{\geq 0}$ is meant to

Listing 3.2 New primitive operations in $\lambda_{\Re}$ (Listing 3.1). See Section 3.5 for extension to signed extension reals.

Primitive f_{ext}	$\mathsf{Arg}(f_{ext}) \to \mathsf{Ret}(f_{ext})$	$\mathsf{Operation}(f_{ext})$
$cast_{\mathbb{R}^n_{>0} \to \Re^n_{\geq 0}}$	$\mathbb{R}_{>0} \times \cdots \times \mathbb{R}_{>0} \to \Re^n_{\geq 0}$	$\lambda x.x$
$+_{\Re}$	$\Re_{\geq 0} \times \Re_{\geq 0} \to \Re_{\geq 0}$	$\lambda(x,y).\begin{cases}\infty & \text{if } \infty \in \{x,y\} \\ x+y & \text{otherwise}\end{cases}$
$\times_{\Re_{\geq 0}}$	$\Re_{\geq 0} \times \Re_{\geq 0} \to \Re_{\geq 0}$	$\lambda(x,y).\begin{cases}\infty & \text{if } \infty \in \{x,y\} \\ x \times y & \text{otherwise}\end{cases}$
$exp_{\Re}$	$\Re_{\geq 0} \to \Re_{\geq 0}$	$\lambda x.\begin{cases}\infty & \text{if } x = \infty \\ exp(x) & \text{otherwise}\end{cases}$
$\mathbf{E}^n_{f_{prb}}$	$\tau \to (\sigma \to \Re^n_{\geq 0}) \to \Re^n_{\geq 0}$ (for $f_{prb} : \tau \to P\ \sigma$)	$\lambda\theta.\ \lambda g.\ \mathbb{E}_{f_{prb}(\theta)}[g]$

Listing 3.3 Semantics of new types in $\lambda_{\Re}$ (Listing 3.1).

Type		*Qbs of values*
$[\![\Re^n_{\geq 0}]\!]$	$=$	$\overline{\mathbb{R}}_{\geq 0} \times \cdots \times \overline{\mathbb{R}}_{\geq 0}$

represent possibly infinite and intractable-to-compute values, we can also add new primitives that would not work for $\mathbb{R}$. For example, we add a primitive

$$\mathrm{sum}_{\infty} : (\mathbb{N} \to \Re_{\geq 0}) \to \Re_{\geq 0}$$

denoting the (possibly infinite) sum of an infinite sequence of non-negative reals.

- Finally, we add new **higher-order primitives**

$$\mathbf{E}^n_{f_{prb}} : \mathsf{ArgType}(f_{prb}) \to (\mathsf{RetType}(f_{prb}) \to \Re^n_{\geq 0}) \to \Re^n_{\geq 0}$$

for each probabilistic primitive f_{prb} and each $n \in \mathbb{N}$, denoting expectation operators for the corresponding probability distributions. In addition to the usual arguments for f_{prb} (e.g., the bias of the coin for *flip*, or the mean and standard deviation for *normal*), these higher-order primitives also take as input a (possibly vector-valued) **integrand** $g : \mathsf{RetType}(f_{prb}) \to \Re^n_{\geq 0}$. They then return the (possibly vector-valued) expected value, or integral, of this integrand, under the measure denoted by f_{prb}.

For example, we assume a primitive $\mathbf{E}_{flip}$ denoting the expectation operator $\mathbb{E}_{flip} = p \mapsto (g \mapsto g(true) \cdot p + g(false) \cdot (1-p))$. As another example, $\mathbf{E}^3_{normal}$ denotes

Probabilistic program

$e : P\ \mathbb{R}$
$e =$ **do** {
  $b \leftarrow\ flip(0.3)$;
  if b **then do** {
    $x \leftarrow\ normal(0, 2)$;
    return $(3 \times x)$}
  else do {
    $x \leftarrow\ normal(0.7, 1)$;
    return $(x \div 2)$}}

Expectation operator

$\mathbf{E}^n_e : (\mathbb{R} \to \mathfrak{R}^n_{\geq 0}) \to \mathfrak{R}^n_{\geq 0}$
$\mathbf{E}^n_e = \lambda g.$
  $\mathbf{E}^n_{flip(0.3)}\ (\lambda b.$
    if b **then**
      $\mathbf{E}^n_{normal(0,2)}\ (\lambda x.$
        $g\ (3 \times x))$
    else
      $\mathbf{E}^n_{normal(0.7,1)}\ (\lambda x.$
        $g\ (x \div 2)))$

Figure 3-2: The expectation operator associated with a probabilistic program can be written by nesting multiple simpler expectation operators, each with respect to some primitive probability distribution that appeared in the original program.

the Gaussian expectation operator for 3-dimensional vector-valued integrands:

$$\mathbb{E}^3_{normal} = (\mu, \sigma) \mapsto \left(g \mapsto \left[\int \pi_i(g(x)) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \right]_{i=1}^{3} \right).$$

Because $\mathfrak{R}^n_{\geq 0}$ is an opaque type in our implementation, we do not need to actually implement exact versions of these primitives; at this stage, we just implement stubs and define their semantics on pen and paper.

These changes are summarized in Listing 3.1.

3.2.2. Expectation operators of composite probabilistic programs

Our language now has primitives representing the expectation operators associated with *primitive* probability distributions, but what about expected values with respect to user-defined probabilistic programs?

It turns out that if a program $e : P\ \tau$ is written as a composition of primitive probability distributions, then its associated expectation operator can be written as a composition of primitive expectation operators. For example, the program $e : P\ \mathbb{R}$ in Figure 3-2 defines a probability measure $[\![e]\!]$ on $\mathbb{R}$: it is the output distribution of the process that samples a boolean $b \sim$ Bernoulli(0.3), then samples x from a Gaussian whose parameters depend on b, and finally returns $3x$ if b is true, or $\frac{x}{2}$ otherwise. The corresponding expectation operator, $\mathbf{E}^n_e$, is shown in the right panel of Figure 3-2. It is implemented by nesting the expectation operators associated with the primitives that appeared in the original program e (i.e., $\mathbf{E}^n_{flip}$ and $\mathbf{E}^n_{normal}$).

The integrator program transformation. We can automate the translation of a probabilistic program into its expectation operator, using a variant of the popular

Listing 3.4 Specification for the **integrator** program transformation. A well-typed source-language expression $\Gamma \vdash e : \tau$ is transformed into a new well-typed expression $\textbf{integrator}\{\Gamma\} \vdash \textbf{integrator}\{e\} : \textbf{integrator}\{\tau\}$, such that

$$(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma} \implies (\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2)) \in R^{\int}_{\tau}.$$

Source type τ	**Transformed type $\textbf{integrator}\{\tau\}$ and spec $R^{\int}_{\tau}$**
$P\ \tau$	$(\textbf{integrator}\{\tau\} \to \Re^n_{\geq 0}) \to \Re^n_{\geq 0}$ $R^{\int}_{P\ \tau} = \left\{(\mu, \mathbb{E}^n_\mu) \mid \left(g \mapsto \int g\, d\mu, \mathbb{E}^n_\mu\right) \in R^{\int}_{(\tau \to \Re^n_{\geq 0}) \to \Re^n_{\geq 0}}\right\}$
$\sigma \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}, \Re^n_{\geq 0}\}$	σ $R^{\int}_{\sigma} = \{(x, y) \mid x = y\}$
$\tau_1 \times \tau_2$	$\textbf{integrator}\{\tau_1\} \times \textbf{integrator}\{\tau_2\}$ $R^{\int}_{\tau_1 \times \tau_2} = \left\{(x, y) \mid \forall i \in \{1, 2\}, (\pi_i\, x, \pi_i\, y) \in R^{\int}_{\tau_i}\right\}$
$\ell_1 \tau_n + \cdots + \ell_n \tau_n$	$\ell_1 \textbf{integrator}\{\tau_1\} + \cdots + \ell_n \textbf{integrator}\{\tau_n\}$ $R^{\int}_{\sum_{i=1}^n \ell_i \tau_i} = \left\{(\ell_i\, x, \ell_i\, y) \mid (x, y) \in R^{\int}_{\tau_i}, i \in \{1, \ldots, n\}\right\}$
$\tau_1 \to \tau_2$	$\textbf{integrator}\{\tau_1\} \to \textbf{integrator}\{\tau_2\}$ $R^{\int}_{\tau_1 \to \tau_2} = \left\{(f, g) \mid \forall (x, y) \in R^{\int}_{\tau_1}, (f(x), g(y)) \in R^{\int}_{\tau_2}\right\}$
Source context Γ	**Transformed context $\textbf{integrator}\{\Gamma\}$ and spec $R^{\int}_{\Gamma}$**
$\cdot$ *(empty context)*	$\cdot$ $R^{\int}_{\cdot} = \{([], [])\}$
$\Gamma, x : \tau$	$\textbf{integrator}\{\Gamma\}, x : \textbf{integrator}\{\tau\}$ $R^{\int}_{\Gamma, x:\tau} = \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}, (v_1, v_2) \in R^{\int}_{\tau}\}$

continuation-passing-style transform from functional programming. The transformation is given, as a macro, in Listings 3.4 and 3.5.

The ultimate goal of the **integrator** macro is to transform an expression e of type $P\ \sigma$ (for some *ground type* σ) into a new expression $\textbf{integrator}\{e\}$ of type $(\sigma \to \Re^n) \to \Re^n$

Listing 3.5 Implementation of the **integrator** program transformation.

Source expression e	**Transformed expression** $\textbf{integrator}\{e\}$
$()$	$()$
r	r
n	n
x	x
f_{det}	f_{det}
f_{ext}	f_{ext}
$\textbf{match}\ e\ \textbf{with}\ \{\ell_i x_i \mapsto e_i\}_{i=1}^n$	$\textbf{match}\ \textbf{integrator}\{e\}\ \textbf{with}\ \{\ell_i x_i \mapsto \textbf{integrator}\{e_i\}\}_{i=1}^n$
(e_1, e_2)	$(\textbf{integrator}\{e_1\}, \textbf{integrator}\{e_2\})$
$\pi_1\ e$	$\pi_1\ \textbf{integrator}\{e\}$
$\pi_2\ e$	$\pi_2\ \textbf{integrator}\{e\}$
$\lambda x.\, e$	$\lambda x.\, \textbf{integrator}\{e\}$
$e_1\ e_2$	$\textbf{integrator}\{e_1\}\ \textbf{integrator}\{e_2\}$
$\textbf{return}\ e$	$\lambda g.\, g(\textbf{integrator}\{e\})$
$\textbf{do}\{e\}$	$\textbf{integrator}\{e\}$
$\textbf{do}\{x \leftarrow e; m\}$	$\lambda g.\, \textbf{integrator}\{e\}\ (\lambda x.\, \textbf{integrator}\{\textbf{do}\{m\}\}\ g)$
f_{prb}	$\mathbb{E}_{f_{prb}}$

implementing the expectation operator. The macro does this compositionally, by recursively translating sub-expressions and combining them into a translation of e.

A sub-expression of type τ in context Γ is translated into a sub-expression of type $\textbf{integrator}\{\tau\}$ in context $\textbf{integrator}\{\Gamma\}$, where $\textbf{integrator}\{\Gamma\}$ is the context obtained by replacing each assumption $x_i : \tau_i \in \Gamma$ by the assumption $x_i : \textbf{integrator}\{\tau_i\}$ in $\textbf{integrator}\{\Gamma\}$. The mapping from types τ to transformed types $\textbf{integrator}\{\tau\}$ is defined in Listing 3.4. Intuitively, expressions that are completely deterministic (e.g., expressions of ground type σ) are left unchanged, whereas expressions that implement or manipulate probability distributions are transformed into expressions that implement or manipulate expectation operators. More concretely:

- If a sub-expression is of ground type $\sigma \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}, \mathfrak{R}^n_{\geq 0}\}$, then it is not probabilistic, and its transformed type is just $\textbf{integrator}\{\sigma\} = \sigma$.
- If a sub-expression is of type $P\ \sigma$ (for one of the ground types σ listed above), then the transformed sub-expression will be the corresponding expectation operator, of type $\textbf{integrator}\{P\ \sigma\} = (\sigma \to \mathfrak{R}^n_{\geq 0}) \to \mathfrak{R}^n_{\geq 0}$.
- If a sub-expression is of type $\tau_1 \to \tau_2$, then the transformed sub-expression is also a function, but it maps *transformed* values into *transformed* values:

$$\textbf{integrator}\{\tau_1 \to \tau_2\} = \textbf{integrator}\{\tau_1\} \to \textbf{integrator}\{\tau_2\}.$$

  For instance, consider the expression $\lambda p.\textbf{do}\{x_1 \leftarrow p; x_2 \leftarrow p; \textbf{return}\ (x_1 + x_2)\}$ of

type $P\,\mathbb{R} \to P\,\mathbb{R}$. The original expression tells us how to take a sampler for a distribution p and turn it into a sampler for the distribution $+_*(p \otimes p)$. When we transform the expression, we want instead a recipe for converting the *expectation operator* associated with p into the *expectation operator* associated with $+_*(p \otimes p)$. That is, the transformed program should have type

$$\textbf{integrator}\{P\,\mathbb{R} \to P\,\mathbb{R}\} = ((\mathbb{R} \to \mathfrak{R}^n_{\geq 0}) \to \mathfrak{R}^n_{\geq 0}) \to (\mathbb{R} \to \mathfrak{R}^n_{\geq 0}) \to \mathfrak{R}^n_{\geq 0}.$$

- If a sub-expression is of type $\tau_1 \times \tau_2$, then—because the surrounding expression may access either element of the tuple—the transformed sub-expression should compute translations of both elements. Thus, the transformed type is $\textbf{integrator}\{\tau_1 \times \tau_2\} = \textbf{integrator}\{\tau_1\} \times \textbf{integrator}\{\tau_2\}$. Similarly, if a sub-expression is of type $\ell_1\tau_1 + \cdots + \ell_n\tau_n$, because the surrounding expression may access the value stored insid the sum, we need to compute a translation of that value, yielding a transformed type $\ell_1\textbf{integrator}\{\tau_1\} + \cdots + \ell_n\textbf{integrator}\{\tau_n\}$.

- If a program is of type $P\,\tau$ for some non-ground type τ, then the transformed program should be an expectation operator for distributions over $\textbf{integrator}\{\tau\}$.

The implementation of **integrator** is given in Listing 3.5. The program transformation is implemented compositionally, by recursively translating sub-expressions of the input program and combining them into a translation of the entire program. Probabilistic primitives are translated directly into their primitive expectation operators. Most language constructs are translated straightforwardly: pairs are translated into pairs, functions are translated into functions, and so on.

The only non-trivial cases are those for the **return** and **do** constructs:

- The translation of **return** e is $\lambda g.\ g(\textbf{integrator}\{e\})$. Recall that the meaning of **return** e is a Dirac delta distribution located at the value of e. The corresponding expectation operator simply evaluates the input function g at the value.

- The translation of $\textbf{do}\{x \leftarrow e; m\}$ is $\lambda g.\ \textbf{integrator}\{e\}\ (\lambda x.\ \textbf{integrator}\{\textbf{do}\{m\}\}\ g)$. This translation is based on the law of total expectation:

  $$\mathbb{E}_{x\sim[\![e]\!],y\sim[\![\textbf{do}\{m\}]\!](x\mapsto x)}[g(y)] = \mathbb{E}_{x\sim[\![e]\!]}[\mathbb{E}_{y\sim[\![\textbf{do}\{m\}]\!](x\mapsto x)}[g(y)]].$$

  This allows us to rewrite an integral with respect to a compound probabilistic program as a *nested* integral with respect to, first, the first statement in the probabilistic program, and second, the remainder of the probabilistic program.

Returning to the example in Fig. 3-2, $\textbf{integrator}\{e\}$ computes the expectation operator shown in the right panel, by systematically replacing sampling primitives with expectation operator primitives, and **do** expressions with nested integration.

3.2.3. Correctness via logical relations

For each type τ, Listing 3.4 also defines a *logical relation* $R^{\int}_{\tau} \subseteq [\![\tau]\!] \times [\![\textbf{integrator}\{\tau\}]\!]$. This serves as a per-type *specification* or *correctness criterion* for the transformation: an expression e_{dst} is a *correct translation* of the expression e_{src} if $([\![e_{src}]\!], [\![e_{dst}]\!]) \in R^{\int}_{\tau}$. Concretely:

- The simplest logical relation is $R^{\int}_{\sigma}$, for $\sigma \in \{1, \mathbb{B}, \mathbb{N}, \mathbb{R}, \Re^{n}_{\geq 0}\}$, which is the identity relation on $[\![\sigma]\!]$ ($\{(x, y) \mid x = y\}$). This specifies that if e is of type σ, then $\textbf{integrator}\{e\}$ should have the same meaning as e.

- The logical relation $R^{\int}_{\tau_1 \times \tau_2}$ specifies that expressions of pair type should be translated into new expressions of pair type computing valid translations of each component of the original pair.

- The logical relation $R^{\int}_{\tau_1 \to \tau_2}$ specifies that functions should be translated into new functions that, when called on a correct translation of an input term, yield a correct translation of the original function's output on that input.

- The logical relation $R^{\int}_{\sum_{i=1}^{n} \ell_n \tau_i}$ specifies that experssions of sum type should be translated into new expressions of sum type that evaluate to the same label ℓ_i, but with the value held inside (of type τ_i) translated correctly (into something of type $\textbf{integrator}\{\tau_i\}$).

- The logical relation $R^{\int}_{P\ \tau}$ specifies that probabilistic programs should be translated into new expressions that compute valid translations of the expectation operator of the original program.

We can now state and prove correctness of the **integrator** transformation. We start with the fundamental lemma.

Lemma 1 (Fundamental lemma for **integrator**)**.** *Suppose* $\Gamma \vdash e : \tau$ *is a well-typed open term. Then if* $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$*, then* $([\![e]\!](\gamma_1), [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R^{\int}_{\tau}$.

Proof. We proceed by induction on the derivation of $\Gamma \vdash e : \tau$.

- **Rule:** $\dfrac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$ (Var) (if $x \notin \Gamma'$)

  We have that $\gamma_1, \gamma_2 \in R^{\int}_{\Gamma, x:\tau, \Gamma'}$, and so in particular,

  $$([\![x]\!](\gamma_1), [\![\textbf{integrator}\{x\}]\!](\gamma_2)) = ([\![x]\!](\gamma_1), [\![x]\!](\gamma_2)) = (\gamma_1[x], \gamma_2[x]) \in R_{\tau},$$

  as required.

- **Rules:**

$$\textsc{ConstR}\ \frac{}{\Gamma \vdash r : \mathbb{R}} \qquad \textsc{ConstN}\ \frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1}$$

In all these cases, $\textbf{integrator}\{e\} = e$ and $[\![e]\!](\gamma)$ does not depend on γ, so for all (γ_1, γ_2), $[\![e]\!](\gamma_1) = [\![\textbf{integrator}\{e\}]\!](\gamma_2)$. This satisfies the specification, because for each of these rules, the result type τ is a ground type, and so $R_\tau^{\int} = \{(x, y) \mid x = y\}$, the identity relation.

- **Rule:**

$$\textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Let $(\gamma_1, \gamma_2) \in R_\Gamma^{\int}$. By the inductive hypotheses for the rule's premises, we have that $([\![e_i]\!](\gamma_1), [\![\textbf{integrator}\{e_i\}]\!](\gamma_2)) \in R_{\tau_i}$. Since $\pi_i\ [\![(e_1, e_2)]\!](\gamma) = [\![e_i]\!](\gamma)$, this implies $([\![(e_1, e_2)]\!](\gamma_1), [\![(\textbf{integrator}\{e_1\}, \textbf{integrator}\{e_2\})]\!](\gamma_2)) \in R_{\tau_1 \times \tau_2}^{\int}$, as required.

- **Rule:**

$$\textsc{Proj}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i}$$

Let $(\gamma_1, \gamma_2) \in R_\Gamma^{\int}$. By the inductive hypothesis applied to the premise,

$$([\![e]\!](\gamma_1), [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R_{\tau_1 \times \tau_2}^{\int}.$$

By the definition of $R_{\tau_1 \times \tau_2}^{\int}$, for $i \in \{1, 2\}$, we have

$$(\pi_i\ [\![e]\!](\gamma_1), \pi_i\ [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R_{\tau_i}^{\int}.$$

Therefore,

$$([\![\pi_i\ e]\!](\gamma_1), [\![\pi_i\ \textbf{integrator}\{e\}]\!](\gamma_2)) = (\pi_i\ [\![e]\!](\gamma_1), \pi_i\ [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R_{\tau_i}^{\int},$$

as required.

- **Rule:**

$$\textsc{Inj}\ \frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \ell_i\ e : \ell_1 \tau_1 + \cdots + \ell_n \tau_n}$$

Let $(\gamma_1, \gamma_2) \in R_\Gamma^{\int}$. By the inductive hypothesis applied to the premise,

$$([\![e]\!](\gamma_1), [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R_{\tau_i}^{\int}.$$

By the definition of $R^{\int}_{\sum_{j=1}^{n} \ell_j \tau_j}$, we have $(\ell_i([\![e]\!](\gamma_1)), \ell_i([\![\textbf{integrator}\{e\}]\!](\gamma_2))) \in R^{\int}_{\sum_{j=1}^{n} \ell_j \tau_j}$.

- **Rule:**

$$\textsc{Match}\quad \frac{\Gamma \vdash e : \ell_1\tau_1 + \cdots + \ell_n\tau_n \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \ldots \qquad \Gamma, x_n : \tau_n \vdash e_n : \tau}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} : \tau}$$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$. By the inductive hypothesis applied to the first premise, $([\![e]\!](\gamma_1), [\![\textbf{integrator}\{e\}]\!](\gamma_2)) \in R^{\int}_{\sum_{i=1}^{n} \ell_i \tau_i}$.

Suppose $[\![e]\!](\gamma_1) = \ell_i(v_1)$ for some $i \in \{1, \ldots, n\}$ and $v_1 \in [\![\tau_i]\!]$. From the definition of $R^{\int}_{\sum_{i=1}^{n} \ell_i \tau_i}$, we know that $[\![\textbf{integrator}\{e\}]\!](\gamma_2) = \ell_i(v_2)$ where $(v_1, v_2) \in R^{\int}_{\tau_i}$.

For each $j \in \{1, \ldots, n\}$, we have by the inductive hypothesis applied to the corresponding premise that if $(\gamma_1[x_j \mapsto v_1], \gamma_2[x_j \mapsto v_2]) \in R^{\int}_{\Gamma, x_j : \tau_j}$, then

$$([\![e_j]\!](\gamma_1[x_j \mapsto v_1]), [\![\textbf{integrator}\{e_j\}]\!](\gamma_2[x_j \mapsto v_2])) \in R^{\int}_{\tau}.$$

Since $(v_1, v_2) \in R^{\int}_{\tau_i}$ and $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$, we have $(\gamma_1[x_i \mapsto v_1], \gamma_2[x_i \mapsto v_2]) \in R^{\int}_{\Gamma, x_i : \tau_i}$.

Therefore, $([\![e_i]\!](\gamma_1[x_i \mapsto v_1]), [\![\textbf{integrator}\{e_i\}]\!](\gamma_2[x_i \mapsto v_2])) \in R^{\int}_{\tau}$, as required.

- **Rule:**

$$\textsc{Abs}\quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$. By the definition of $R^{\int}_{\tau_1 \to \tau_2}$, we need to show that for all $(v_1, v_2) \in R^{\int}_{\tau_1}$, $([\![\lambda x.\, e]\!](\gamma_1)(v_1), [\![\lambda x.\, \textbf{integrator}\{e\}]\!](\gamma_2)(v_2)) \in R^{\int}_{\tau_2}$.

For any $(v_1, v_2) \in R^{\int}_{\tau_1}$, we have:

$$\begin{aligned}&([\![\lambda x.\, e]\!](\gamma_1)(v_1), [\![\lambda x.\, \textbf{integrator}\{e\}]\!](\gamma_2)(v_2))\\ &\quad = ([\![e]\!](\gamma_1[x \mapsto v_1]), [\![\textbf{integrator}\{e\}]\!](\gamma_2[x \mapsto v_2]))\end{aligned}$$

Since $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$ and $(v_1, v_2) \in R^{\int}_{\tau_1}$, we have $(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in R^{\int}_{\Gamma, x : \tau_1}$. Therefore, by the inductive hypothesis applied to the premise,

$$([\![e]\!](\gamma_1[x \mapsto v_1]), [\![\textbf{integrator}\{e\}]\!](\gamma_2[x \mapsto v_2])) \in R^{\int}_{\tau_2},$$

as required.

- **Rule:**

$$\textsc{App}\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$. By the inductive hypotheses applied to the premises:

$$(\llbracket e_1 \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e_1\} \rrbracket(\gamma_2)) \in R^{\int}_{\tau_1 \to \tau_2}$$
$$(\llbracket e_2 \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e_2\} \rrbracket(\gamma_2)) \in R^{\int}_{\tau_1}$$

Then by the definition of $R^{\int}_{\tau_1 \to \tau_2}$, we have

$$(\llbracket e_1 \rrbracket(\gamma_1)(\llbracket e_2 \rrbracket(\gamma_1)), \llbracket \textbf{integrator}\{e_1\} \rrbracket(\gamma_2)(\llbracket \textbf{integrator}\{e_2\} \rrbracket(\gamma_2))) \in R^{\int}_{\tau_2}.$$

- **Rule:**

$$\frac{}{\Gamma \vdash f : \mathsf{ArgType}(f) \to \mathsf{RetType}(f)} \textsc{BuiltIn}$$

For deterministic built-ins, the transformation leaves them unchanged, so we have $\textbf{integrator}\{f_{det}\} = f_{det}$. All our deterministic built-ins have argument types and return types that are *ground types*, with identity logical relations. Thus, the corresponding logical relations at their function types are also the identity relations. Thus, by leaving f_{det} unchanged, we trivially satisfy the specification.

For probabilistic built-ins f_{prb}, we transform them into their corresponding built-in expectation operators. The argument types of probabilistic built-ins are ground types, and the return types are $P\ \sigma$ for some ground type σ. Thus, the specification for the transform is that for any argument $x \in \llbracket \mathsf{ArgType}(f_{prb}) \rrbracket$, we have $(\llbracket f_{prb} \rrbracket(x), \llbracket \mathbf{E}_{f_{prb}(x)} \rrbracket) \in R_{P\ \sigma}$. But this is simply the requirement that the built-in expectation operator is really the expectation operator for the built-in probabilistic primitive, which holds by construction.

- **Rule:**

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{return}\ e : P\ \tau} \textsc{Return}$$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$, and suppose $(g_1, g_2) \in R^{\int}_{\tau \to \mathfrak{R}^n_{\geq 0}}$. Then we want to show that

$$\left(\int g_1 d\llbracket \textbf{return}\ e \rrbracket(\gamma_1), g_2(\llbracket \textbf{integrator}\{e\} \rrbracket) \right) \in R^{\int}_{\mathfrak{R}^n_{\geq 0}}.$$

From the semantics of **return**, we know that $\llbracket \textbf{return}\ e \rrbracket(\gamma_1) = \delta_{\llbracket e \rrbracket(\gamma_1)}$, the Dirac measure at $\llbracket e \rrbracket(\gamma_1)$. Therefore, $\int g_1 d\llbracket \textbf{return}\ e \rrbracket(\gamma_1) = g_1(\llbracket e \rrbracket(\gamma_1))$. By the fact that $(g_1, g_2) \in R^{\int}_{\tau \to \mathfrak{R}^n_{\geq 0}}$, we have

$$(\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2)) \in R^{\int}_{\tau} \implies (g_1(\llbracket e \rrbracket(\gamma_1)), g_2(\llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2))) \in R^{\int}_{\mathfrak{R}^n_{\geq 0}}$$

The hypothesis on the left holds by induction, and so we obtain the conclusion.

- **Rule:** $\boxed{\text{Do}\ \dfrac{\Gamma \vdash e : P\ \tau}{\Gamma \vdash \mathbf{do}\ \{e\} : P\ \tau}}$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$. By the inductive hypothesis applied to the premise,

$$([\![e]\!](\gamma_1), [\![\mathbf{integrator}\{e\}]\!](\gamma_2)) \in R^{\int}_{P\ \tau}.$$

We have $[\![\mathbf{do}\ \{e\}]\!](\gamma_1) = [\![e]\!](\gamma_1)$ and $[\![\mathbf{integrator}\{\mathbf{do}\ \{e\}\}]\!](\gamma_2) = [\![\mathbf{integrator}\{e\}]\!](\gamma_2)$, so $([\![\mathbf{do}\ \{e\}]\!](\gamma_1), [\![\mathbf{integrator}\{\mathbf{do}\ \{e\}\}]\!](\gamma_2)) \in R^{\int}_{P\ \tau}$, as required.

- **Rule:** $\boxed{\textsc{Bind}\ \dfrac{\Gamma \vdash e : P\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash \mathbf{do}\ \{m\} : P\ \tau_2}{\Gamma \vdash \mathbf{do}\ \{x \leftarrow e; m\} : P\ \tau_2}}$

Let $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$. The semantics of $\mathbf{do}\ \{x \leftarrow e; m\}$ is:

$$[\![\mathbf{do}\ \{x \leftarrow e; m\}]\!](\gamma_1) = \int\!\!\int [\![\mathbf{do}\ \{m\}]\!](\gamma_1[x \mapsto v])\ [\![e]\!](\gamma_1, dv)$$

The expectation operator corresponding to this measure is

$$g \mapsto [\![e]\!](\gamma_1)(v \mapsto [\![\mathbf{do}\ \{m\}]\!](\gamma_1[x \mapsto v])(g))$$

The semantics of the transformed expression is:

$$g \mapsto [\![\mathbf{integrator}\{e\}]\!](\gamma_2)(v \mapsto [\![\mathbf{integrator}\{\mathbf{do}\ \{m\}\}]\!](\gamma_2[x \mapsto v])(g))$$

Now suppose $(g_1, g_2) \in R^{\int}_{\tau_2 \to \mathfrak{R}^n_{\geq 0}}$, and consider the result of applying the expectation operator to g_1 and the transformed expression's semantics to g_2.

By the inductive hypothesis for the second premise, for all $(v_1, v_2) \in R^{\int}_{\tau_1}$, we have $(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in R^{\int}_{\Gamma, x:\tau_1}$, which implies:

$$([\![\mathbf{do}\ \{m\}]\!](\gamma_1[x \mapsto v_1])(g_1), [\![\mathbf{integrator}\{\mathbf{do}\ \{m\}\}]\!](\gamma_2[x \mapsto v_2])(g_2)) \in R^{\int}_{\mathfrak{R}^n_{\geq 0}}.$$

Thus, the maps

$$G_1 := v \mapsto [\![\mathbf{do}\ \{m\}]\!](\gamma_1[x \mapsto v])(g_1)$$
$$G_2 := v \mapsto [\![\mathbf{integrator}\{\mathbf{do}\ \{m\}\}]\!](\gamma_2[x \mapsto v])(g_2)$$

are related by $R^{\int}_{\tau_1 \to \mathfrak{R}^n_{\geq 0}}$. By the inductive hypotheses applied to the first premise,

$$(\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2)) \in R^{\int}_{P\ \tau_1}$$

and thus $(\llbracket e \rrbracket(\gamma_1)(G_1), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2)(G_2)) \in R^{\int}_{\mathfrak{R}^n_{\geq 0}}$, as required.

This completes the case analysis for all typing rules, establishing that the **integrator** transformation preserves the logical relation for all well-typed terms. □

Theorem 1 (Correctness of **integrator**). *Let e be a program of type $P\ \sigma$. Then $\llbracket \textbf{integrator}\{e\} \rrbracket$ is the corresponding expectation operator $\mathbb{E}^n_{\llbracket e \rrbracket}$.*

Proof. By Lemma 1, $(\llbracket e \rrbracket, \llbracket \textbf{integrator}\{e\} \rrbracket) \in R_{P\ \sigma}$, which immediately gives the desired result. □

3.3 Automatic unbiased estimation via higher-order operator overloading

With automatic integration, users can define programs that encode integrals (and functions of integrals) that they wish to compute. We now aim to transform these programs—which would generally be intractable to evaluate exactly—into *probabilistic* programs that unbiasedly estimate their values.

3.3.1. Higher-order operator overloading

To transform a program of type $\mathfrak{R}^n_{\geq 0}$ into an unbiased estimator, the **estimator** transformation (Listings 3.6 and 3.7) recurses through the program, looking for primitive operations f_{ext} that either *produce* or *consume* values of type $\mathfrak{R}^n_{\geq 0}$. When such a primitive is encountered, **estimator** replaces it with code that instead produces and consumes unbiased estimators, of type $P\ (\mathbb{R} \times \cdots \times \mathbb{R})$. Thus, **estimator** implements a form of **operator overloading**: each primitive that operates on extended reals is *overloaded* to also operate on unbiased estimators.

This recursive find-and-replace approach is formalized in Listings 3.6 and 3.7. At the type level, **estimator** replaces all occurrences of $\mathfrak{R}^n_{\geq 0}$ with $P\ (\mathbb{R} \times \cdots \times \mathbb{R})$, the type of probabilistic programs returning n-dimensional vectors of standard real numbers. At the expression level, **estimator** leaves most program structure unchanged (e.g., functions are transformed into functions, pairs into pairs) but replaces primitives f_{ext} operating on $\mathfrak{R}^n_{\geq 0}$ values with their overloaded versions.

Overloading first-order primitives. The core requirement for overloading a primitive f_{ext} is that its translated implementation, **estimator**$\{f_{ext}\}$, must bridge the gap between the original semantics and the world of estimators. If f_{ext} originally maps arguments $a_1, \ldots, a_k$ to a result r, then **estimator**$\{f_{ext}\}$ must map inputs $\widehat{a_1}, \ldots, \widehat{a_k}$ to an output $\widehat{r}$, where each $\widehat{a_i}$ is an unbiased estimator for a_i and $\widehat{r}$ is an unbiased estimator for r. For example, the primitive $+_{\mathfrak{R}} : \mathfrak{R}_{\geq 0} \times \mathfrak{R}_{\geq 0} \to \mathfrak{R}_{\geq 0}$ operates on two (potentially

Listing 3.6 Specification for the **estimator** program transformation. A well-typed source-language expression $\Gamma \vdash e : \tau$ is transformed into a new well-typed expression $\textbf{estimator}\{\Gamma\} \vdash \textbf{estimator}\{e\} : \textbf{estimator}\{\tau\}$, such that

$$(\gamma_1, \gamma_2) \in R^{\sim}_{\Gamma} \implies (\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{estimator}\{e\} \rrbracket(\gamma_2)) \in R^{\sim}_{\tau}.$$

Source type τ	**Transformed type** $\textbf{estimator}\{\tau\}$ **and spec** $R^{\sim}_{\tau}$
$\Re^n_{\geq 0}$	$P\,(\mathbb{R} \times \cdots \times \mathbb{R})$ $R^{\sim}_{\Re^n_{\geq 0}} = \{(r, \widehat{r}) \mid \forall i \in \{1, \ldots, n\}, \pi_i r = \mathbb{E}_{\widehat{r}}[\pi_i]\}$
$\sigma \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}\}$	σ $R^{\sim}_{\sigma} = \{(x, y) \mid x = y\}$
$\tau_1 \times \tau_2$	$\textbf{estimator}\{\tau_1\} \times \textbf{estimator}\{\tau_2\}$ $R^{\sim}_{\tau_1 \times \tau_2} = \left\{(x, y) \mid \forall i \in \{1, 2\}, (\pi_i\, x, \pi_i\, y) \in R^{\sim}_{\tau_i}\right\}$
$\ell_1 \tau_n + \cdots + \ell_n \tau_n$	$\ell_1 \textbf{estimator}\{\tau_1\} + \cdots + \ell_n \textbf{estimator}\{\tau_n\}$ $R^{\sim}_{\sum_{i=1}^n \ell_i \tau_i} = \left\{(\ell_i\, x, \ell_i\, y) \mid (x, y) \in R^{\sim}_{\tau_i}, i \in \{1, \ldots, n\}\right\}$
$\tau_1 \to \tau_2$	$\textbf{estimator}\{\tau_1\} \to \textbf{estimator}\{\tau_2\}$ $R^{\sim}_{\tau_1 \to \tau_2} = \left\{(f, g) \mid \forall (x, y) \in R^{\sim}_{\tau_1}, (f(x), g(y)) \in R^{\sim}_{\tau_2}\right\}$
$P\,\tau$	$P\,\textbf{estimator}\{\tau\}$ $R^{\sim}_{P\,\tau} = \{(\mu, \nu) \mid \exists Q \in \text{Meas}\,\llbracket \tau \times \textbf{estimator}\{\tau\} \rrbracket.$ $\pi_{1*}Q = \mu, \pi_{2*}Q = \nu, Q(1_{R^{\sim}_{\tau}}) = 1\}$
Source context Γ	**Transformed context** $\textbf{estimator}\{\Gamma\}$ **and spec** $R^{\sim}_{\Gamma}$
$\cdot$ *(empty context)*	$\cdot$ $R^{\sim}_{\cdot} = \{([], [])\}$
$\Gamma, x : \tau$	$\textbf{estimator}\{\Gamma\}, x : \textbf{estimator}\{\tau\}$ $R^{\sim}_{\Gamma, x:\tau} = \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid$ $(\gamma_1, \gamma_2) \in R^{\sim}_{\Gamma}, (v_1, v_2) \in R^{\sim}_{\tau}\}$

intractable) non-negative extended reals. Its overloaded implementations must take two probabilistic programs $\widehat{x} : P\,\mathbb{R}$ and $\widehat{y} : P\,\mathbb{R}$—which are assumed to be unbiased estimators for x and y respectively—and return a new probabilistic program

Listing 3.7 Implementation of the **estimator** program transformation.

Expression e	**Transformed expression** **estimator**$\{e\}$
$()$	$()$
r	r
n	n
x	x
f_{det}	f_{det}
f_{prb}	f_{prb}
if e_1 **then** e_2 **else** e_3	**if** **estimator**$\{e_1\}$ **then** **estimator**$\{e_2\}$ **else** **estimator**$\{e_3\}$
(e_1, e_2)	(**estimator**$\{e_1\}$, **estimator**$\{e_2\}$)
$\pi_1\ e$	π_1 **estimator**$\{e\}$
$\pi_2\ e$	π_2 **estimator**$\{e\}$
$\lambda x.\, e$	$\lambda x.$ **estimator**$\{e\}$
$e_1\ e_2$	**estimator**$\{e_1\}$ **estimator**$\{e_2\}$
return e	**return** **estimator**$\{e\}$
do$\{e\}$	**estimator**$\{e\}$
do$\{x \leftarrow e; m\}$	**do**$\{x \leftarrow$ **estimator**$\{e\}$; **estimator**$\{$**do**$\{m\}\}\}$
f_{ext}	*see Listing 3.8 for translations of primitives* f_{ext}

estimator$\{+_\Re\}(\widehat{x}, \widehat{y}) : P\ \mathbb{R}$ which unbiasedly estimates the true sum $x +_\Re y$.

A key distinction from the **integrator** transformation is that the translation of primitives under **estimator** is often *not unique*. As shown in Listing 3.8, many primitives f_{ext} admit multiple valid translations into unbiased estimators. For example, the primitive $+_\Re : \Re_{\geq 0} \times \Re_{\geq 0} \to \Re_{\geq 0}$ can be translated using a `SUM` strategy, where we independently estimate both operands and sum the results:

$$\textbf{estimator}\{+_\Re\}^{\texttt{SUM}} = \lambda(\widehat{x}, \widehat{y}).\ \textbf{do}\ \{\tilde{x} \leftarrow \widehat{x}; \tilde{y} \leftarrow \widehat{y}; \textbf{return}\ (\tilde{x} + \tilde{y})\}$$

Alternatively, it can be translated using a `SAMPLE` strategy, where we randomly choose one operand to estimate and re-weight the result:

$$\textbf{estimator}\{+_\Re\}^{\texttt{SAMPLE}} = \lambda(\widehat{x}, \widehat{y}).\ \textbf{do}\ \{b \leftarrow \mathit{flip}(0.5); \tilde{s} \leftarrow \textbf{if}\ b\ \textbf{then}\ \widehat{x}\ \textbf{else}\ \widehat{y}; \textbf{return}\ (2 \times \tilde{s})\}$$

Both translations yield unbiased estimators for the sum, but the latter is cheaper and generally higher-variance.

Sometimes, the estimator can be substantially more complex than the original primitive. For example, consider $exp_\Re : \Re_{\geq 0} \to \Re_{\geq 0}$. Its overloaded implementation **estimator**$\{exp_\Re\}$ takes an estimator $\widehat{x} : P\ \mathbb{R}$ for $x \in \Re_{\geq 0}$ and must return an estimator for $exp_\Re(x)$. The `POISSON` strategy shown in Listing 3.8 achieves this using a randomized version of the Taylor series for e^x, sampling an index n from a Poisson distribution and using the input estimator $\widehat{x}$ to estimate the n-th term.

Listing 3.8 Several possible implementations of the **estimator** program transformation for $\lambda_{\Re}$ primitives.

Primitive f	**Estimation strategy**	**Transformed primitive** **estimator**$\{f\}$
$cast_{\mathbb{R}\to\Re}$	`EXACT`	$\lambda x.$ **return** x
$+_{\Re}$	`SUM`	$\lambda(x, y).$ **do** $\{\widehat{x} \leftarrow x; \widehat{y} \leftarrow y;$ **return** $(\widehat{x} + \widehat{y})\}$
	`SAMPLE`	$\lambda(x, y).$ **do** $\{$ $\quad b \leftarrow flip(0.5);$ $\quad \widehat{s} \leftarrow$ **if** b **then** x **else** $y;$ $\quad$ **return** $(2 \times \widehat{s})$ $\}$
$\times_{\Re}$	`PRODUCT`	$\lambda(x, y).$ **do** $\{\widehat{x} \leftarrow x; \widehat{y} \leftarrow y;$ **return** $(\widehat{x} \times \widehat{y})\}$
$exp_{\Re}$	`POISSON`$(\lambda : \mathbb{R}_{>0})$	$\lambda x.$ **do** $\{$ $\quad n \leftarrow poisson(\lambda);$ $\quad \widehat{xs} \leftarrow repeat_P\ x\ n;$ $\quad$ **return** $(exp(\lambda) \times (prod\ (map(\lambda\widehat{x}.\ \widehat{x} \div \lambda)\ \widehat{xs})))$ $\}$
sum_{∞}	`SUM`	$\lambda g.$ **do** $\{$ $\quad n \leftarrow geometric(0.5);$ $\quad \widehat{xs} \leftarrow map_P$ $\quad\quad (\lambda i.$**do**$\{\widehat{x_i} \leftarrow g(i);$ **return** $(\widehat{x_i} \times 2^i)\})$ $\quad\quad (range\ n);$ $\quad$ **return** $(sum\ \widehat{xs})$ $\}$
	`SAMPLE`	$\lambda g.$ **do** $\{$ $\quad n \leftarrow geometric(0.5);$ $\quad \widehat{x_n} \leftarrow g(n);$ $\quad$ **return**$(2^{n+1} \times \widehat{x_n})$ $\}$

Overloading expectation operators. The expectation operator primitives

$$\mathbf{E}^n_{f_{prb}} : \mathsf{ArgType}(f_{prb}) \to (\sigma \to \Re^n_{\geq 0}) \to \Re^n_{\geq 0}$$

(where $\mathsf{RetType}(f_{prb}) = P\ \sigma$) are higher-order, and thus have a more complex specification. The arguments $\theta = \mathsf{ArgType}(f_{prb})$ are typically ground types (like $\mathbb{R}$ for the mean of a normal distribution) and are left unchanged by **estimator**. The crucial argument is the integrand $g : \sigma \to \Re^n_{\geq 0}$. The overloaded primitive **estimator**$\{\mathbf{E}^n_{f_{prb}}\}$

Listing 3.9 Several possible implementations of the **estimator** program transformation for expectation operator primitives.

Primitive f	Estimation strategy	Transformed primitive **estimator**$\{f\}$
$\mathbf{E}_{flip}$	`SAMPLE`	$\lambda p.\ \lambda g.$ **do** $\{\widehat{x} \leftarrow flip(p); g(\widehat{x})\}$

	`FAIR`	$\lambda p.\ \lambda g.$ **do** {
		$\widehat{x} \leftarrow flip(0.5)$;
		$\widehat{y} \leftarrow g(\widehat{x})$;
		if $\widehat{x}$ **then**
		return $(2 \times p \times \widehat{y})$
		else
		return $(2 \times (1-p) \times \widehat{y})$
		}

	`ENUMERATE`	$\lambda p.\ \lambda g.$ **do** {
		$\widehat{y_T} \leftarrow g(\textbf{true})$;
		$\widehat{y_F} \leftarrow g(\textbf{false})$;
		return $(p \times \widehat{y_T} + (1-p) \times \widehat{y_F})$
		}

$\mathbf{E}_{normal}$	`SAMPLE`	$\lambda(\mu, \sigma).\ \lambda g.$ **do** $\{\widehat{x} \leftarrow normal(\mu, \sigma); g(\widehat{x})\}$

	`ANTITHETIC`	$\lambda(\mu, \sigma).\ \lambda g.$ **do** {
		$\widehat{x} \leftarrow normal(\mu, \sigma)$;
		$\widehat{y_1} \leftarrow g(\widehat{x})$;
		$\widehat{y_2} \leftarrow g(2 \times \mu - \widehat{x})$;
		return $((\widehat{y_1} + \widehat{y_2}) \div 2)$
		}

$\mathbf{E}_{geometric}$	`SAMPLE`	$\lambda p.\ \lambda g.$ **do** $\{\widehat{x} \leftarrow geometric(p); g(\widehat{x})\}$

	`SUM`	$\lambda p.\ \lambda g.$ **do** {
		$n \leftarrow geometric(p)$;
		$\widehat{ys} \leftarrow map_P\ g\ (range\ n)$;
		return $((1-p) \times (sum\ \widehat{ys}))$
		}

takes θ and a function $\widehat{g} : \sigma \to P\ (\mathbb{R} \times \cdots \times \mathbb{R})$, which represents an *estimator for the integrand*. The specification for **estimator**$\{\mathbf{E}^n_{f_{prb}}\}$ is as follows: if, for every possible sample $x \in [\![\sigma]\!]$, the program $\widehat{g}(x)$ unbiasedly estimates the true vector $g(x)$, then the program returned by **estimator**$\{\mathbf{E}^n_{f_{prb}}\}(\theta)(\widehat{g})$ must be an unbiased estimator for the true integral $\mathbb{E}_{f_{prb}(\theta)}[g]$. Several examples in Listing 3.9, such as the different strategies for **estimator**$\{\mathbf{E}_{flip}\}$ and **estimator**$\{\mathbf{E}_{normal}\}$, illustrate how this higher-order specification can be satisfied. For instance, the `SAMPLE` strategy for $\mathbf{E}_{f_{prb}}$ works by drawing a sample $\widehat{x} \sim f_{prb}(\theta)$ and then running the integrand estimator $\widehat{g}(\widehat{x})$ to get

an estimate of $g(\widehat{x})$. By the law of total expectation, if $\widehat{g(\widehat{x})}$ unbiasedly estimates $g(\widehat{x})$, then the overall process unbiasedly estimates $\mathbb{E}_{f_{prb}(\theta)}[g]$.

Estimation strategy annotations. The multiplicity of estimation strategies offers significant flexibility. For each of our primitives, we select a default estimation strategy, but allow users to annotate the primitive with a different one.[1] By selecting different strategies for different primitives, users can explore a combinatorial space of unbiased estimators for their target integral expression, allowing them to tailor the resulting estimator's variance-cost profile to the specific needs of their application. The compositional nature of the **estimator** transformation ensures that any combination of valid primitive translations results in a globally unbiased estimator for the original $\mathfrak{R}^n_{\geq 0}$-valued program, as the next section establishes.

3.3.2. Correctness via logical relations

Just as with the **integrator** transformation, we establish the correctness of **estimator** using logical relations. For each type τ, Listing 3.6 defines a transformed type **estimator**$\{\tau\}$ and a logical relation $R^{\sim}_{\tau} \subseteq [\![\tau]\!] \times [\![\textbf{estimator}\{\tau\}]\!]$. This relation formalizes the expected relationship between the denotation of an original expression and its transformed version under **estimator**.

The core of the specification lies in the relation for the extended real type, $R^{\sim}_{\mathfrak{R}^n_{\geq 0}}$. It states that a pair $(r, \widehat{r})$ is related if $r \in [\![\mathfrak{R}^n_{\geq 0}]\!]$ (an n-dimensional vector of non-negative extended reals) and $\widehat{r} \in [\![P\ (\mathbb{R}^n)]\!]$ (a probabilistic program returning n-dimensional real vectors) such that, for each dimension i, the expected value of the i-th component returned by $\widehat{r}$ is equal to the i-th component of r. That is, $\widehat{r}$ must be an unbiased estimator for r. For other types, the relations are standard: ground types relate identical values, pairs relate component-wise related pairs, functions relate functions preserving the relation, and probabilistic program types relate measures that can be coupled to produce related samples.

The fundamental lemma asserts that the **estimator** transformation preserves the logical relation for all well-typed terms:

Lemma 2 (Fundamental lemma for **estimator**)**.** *Suppose* $\Gamma \vdash e : \tau$ *is a well-typed open term. Then if* $(\gamma_1, \gamma_2) \in R^{\sim}_{\Gamma}$*, then* $([\![e]\!](\gamma_1), [\![\textbf{estimator}\{e\}]\!](\gamma_2)) \in R^{\sim}_{\tau}$.

Proof sketch. The proof proceeds by induction on the typing derivation of e. Most cases involving standard language constructs (variables, constants, pairs, projections, sums, matches, functions, application, standard deterministic primitives) follow exactly the same logic as in Lemma 1. What is new is the need to verify the

[1] Formally, the estimation strategy can be viewed as an additional argument of sum type, e.g. `SAMPLE` 1 + `SUM` 1, which does not change the semantics of the primitive but does determine which estimation rule is chosen by **estimator**$\{\cdot\}$. This encoding also allows for strategies to be parameterized by dynamically computed values. For example, $exp_{\mathfrak{R}}$ accepts a strategy argument of unary sum type `POISSON` $\mathbb{R}_{>0}$, where the argument is the Poisson rate λ and can be dynamically chosen.

lemma for the primitives f_{ext} operating on $\mathfrak{R}^n_{\geq 0}$ values, where **estimator**$\{\cdot\}$ performs overloading.

We give a few representative examples:

- **Primitive:** $cast_{\mathbb{R}\to\mathfrak{R}} : \mathbb{R} \to \mathfrak{R}_{\geq 0}$. The `EXACT` estimation strategy is given by

$$\textbf{estimator}\{cast_{\mathbb{R}^n_{>0}\to\mathfrak{R}^n_{\geq 0}}\} = \lambda x.\textbf{return}\ x.$$

  We need to show that for any $(x, \widehat{x}) \in R^{\sim}_{\mathbb{R}_{>0}\times\cdots\times\mathbb{R}_{>0}}$ (which implies $x = \widehat{x}$ are equal vectors of positive reals), we have $([\![cast_{\mathbb{R}^n_{>0}\to\mathfrak{R}^n_{\geq 0}}]\!](x), [\![\textbf{estimator}\{cast_{\mathbb{R}^n_{>0}\to\mathfrak{R}^n_{\geq 0}}\}]\!](\widehat{x})) \in R^{\sim}_{\mathfrak{R}^n_{\geq 0}}$. Unfolding the semantics of **return**, this pair is $(x, \delta_{\widehat{x}}) = (x, \delta_x)$. This pair is in $R^{\sim}_{\mathfrak{R}_{\geq 0}}$ because $\mathbb{E}_{\delta_x}[\pi_i] = \pi_i x$, as required.

- **Primitive:** $+_{\mathfrak{R}} : \mathfrak{R}_{\geq 0} \times \mathfrak{R}_{\geq 0} \to \mathfrak{R}_{\geq 0}$. Consider the `SUM` strategy: $\textbf{estimator}\{+_{\mathfrak{R}}\}^{\texttt{SUM}} = \lambda(\widehat{x}, \widehat{y}).\textbf{do}\ \{\tilde{x} \leftarrow \widehat{x}; \tilde{y} \leftarrow \widehat{y}; \textbf{return}\ (\tilde{x} + \tilde{y})\}$. Assume we have inputs $(x, \widehat{x})$ and $(y, \widehat{y})$ such that $(x, \widehat{x}) \in R^{\sim}_{\mathfrak{R}_{\geq 0}}$ and $(y, \widehat{y}) \in R^{\sim}_{\mathfrak{R}_{\geq 0}}$. This means $\mathbb{E}_{\widehat{x}}[\mathrm{id}] = x$ and $\mathbb{E}_{\widehat{y}}[\mathrm{id}] = y$. We need to show that $(x + y, [\![\textbf{estimator}\{+_{\mathfrak{R}}\}^{\texttt{SUM}}]\!](\widehat{x}, \widehat{y})) \in R^{\sim}_{\mathfrak{R}_{\geq 0}}$. The expected value of the resulting estimator is $\mathbb{E}_{\tilde{x}\sim\widehat{x},\tilde{y}\sim\widehat{y}}[\tilde{x} + \tilde{y}] = \mathbb{E}_{\tilde{x}\sim\widehat{x}}[\tilde{x}] + \mathbb{E}_{\tilde{y}\sim\widehat{y}}[\tilde{y}] = x + y$, as desired. Thus, the relation holds.

- **Primitive:** $\mathbf{E}_{flip} : \mathbb{R}_{[0,1]} \to (\mathbb{B} \to \mathfrak{R}_{\geq 0}) \to \mathfrak{R}_{\geq 0}$. Consider the `SAMPLE` strategy: $\textbf{estimator}\{\mathbf{E}_{flip}\}^{\texttt{SAMPLE}} = \lambda p.\lambda\widehat{g}.\textbf{do}\ \{\widehat{b} \leftarrow flip(p); \widehat{g(b)}\}$. Assume $(p, \widehat{p}) \in R^{\sim}_{\mathbb{R}_{[0,1]}}$ (so $p = \widehat{p}$) and $(g, \widehat{g}) \in R^{\sim}_{\mathbb{B}\to\mathfrak{R}_{\geq 0}}$. The latter means that for any $b \in \{true, false\}$, $(g(b), \widehat{g}(b)) \in R^{\sim}_{\mathfrak{R}_{\geq 0}}$, i.e., $\mathbb{E}_{\widehat{g}(b)}[\mathrm{id}] = g(b)$. We need to show

$$([\![\mathbf{E}_{flip}]\!](p)(g), [\![\textbf{estimator}\{\mathbf{E}_{flip}\}^{\texttt{SAMPLE}}]\!](\widehat{p})(\widehat{g})) \in R^{\sim}_{\mathfrak{R}_{\geq 0}}.$$

  The left side denotes the true expectation $\mathbb{E}_{b\sim flip(p)}[g(b)] = p\cdot g(true) + (1-p)\cdot g(false)$. The right side is the denotation of the probabilistic program $\textbf{do}\ \{\widehat{b} \leftarrow flip(p); \widehat{g(b)}\}$. Its expected value is:

$$\begin{aligned}\mathbb{E}_{\widehat{b}\sim flip(p)}[\mathbb{E}_{\widehat{g(b)}}[\mathrm{id}]] &= \mathbb{E}_{\widehat{b}\sim flip(p)}[\widehat{g(b)}] \\ &= p\cdot g(true) + (1-p)\cdot g(false)\end{aligned}$$

  This matches the true expectation, so the relation holds. The proofs for other strategies follow similar reasoning, applying properties of expectation and the inductive hypotheses.

By verifying all primitives and standard rules, the fundamental lemma holds. □

The fundamental lemma directly implies the main correctness theorem for the **estimator** transformation.

Theorem 2 (Correctness of **estimator**)**.** *Let e be a closed program of type $\mathfrak{R}^n_{\geq 0}$. Then $[\![\textbf{estimator}\{e\}]\!]$ is an unbiased estimator for $[\![e]\!]$. That is, $\mathbb{E}_{[\![\textbf{estimator}\{e\}]\!]}[\pi_i] = \pi_i([\![e]\!])$ for all*

$i \in \{1, \dots, n\}$.

Proof. Apply Lemma 2 to the closed term e in the empty context to obtain that $(\llbracket e \rrbracket, \llbracket \mathbf{estimator}\{e\} \rrbracket) \in R^{\sim}_{\mathfrak{R}^n_{\geq 0}}$. The definition of $R^{\sim}_{\mathfrak{R}^n_{\geq 0}}$ gives the desired result. □

3.4 Compiling recursive probabilistic programs and integral expressions

The **integrator** and **estimator** program transformations can be extended to handle recursion syntactically in a straightforward manner, by simply recursing into the body of a μ-expression:

$$\mathbf{integrator}\{\mu x.\, e\} \coloneqq \mu x.\, \mathbf{integrator}\{e\}$$
$$\mathbf{estimator}\{\mu x.\, e\} \coloneqq \mu x.\, \mathbf{estimator}\{e\}$$

Establishing the correctness of these rules, however, requires new proof techniques.

3.4.1. Semantics of recursion for the extended language.

Consider the following two programs, the first of type $\mathbb{R}$ and the second of type $\mathfrak{R}_{\geq 0}$:

$$e_{\mathbb{R}} \coloneqq \mu r.\, r \times 0.5 + 1$$
$$e_{\mathfrak{R}} \coloneqq \mu r.\, r \times_{\mathfrak{R}} \mathit{cast}_{\mathbb{R}_{>0} \to \mathfrak{R}_{\geq 0}}(0.5) +_{\mathfrak{R}} \mathit{cast}_{\mathbb{R}_{>0} \to \mathfrak{R}_{\geq 0}}(1)$$

Under our ω-qbs semantics, in which $\llbracket \mathbb{R} \rrbracket = \mathbb{R}_{\perp}$, the denotation of the first program, $\llbracket e_{\mathbb{R}} \rrbracket$, is $\perp$: when evaluating r, we unconditionally attempt to recursively evaluate r, leading to an infinite loop. We have not yet assigned an ω-qbs denotation to $\mathfrak{R}_{\geq 0}$, however, so the second program's denotation is not yet possible to compute.

Operationally, values of type $\mathfrak{R}$ are opaque, so nothing in the implementation requires that we enter an infinite loop when evaluating $e_{\mathfrak{R}}$. Furthermore, if we transform $e_{\mathfrak{R}}$ using **estimator** (with the `SAMPLE` strategy for $+_{\mathfrak{R}}$ and the `EXACT` strategy for *cast*), we get a program that can actually be run:

$$\mathbf{estimator}\{e_{\mathfrak{R}}\} = \mu \widehat{r}.\, \mathbf{do}\ \{b \leftarrow \mathit{flip}(0.5); \mathbf{if}\ b\ \mathbf{then\ return}\ 2\ \mathbf{else\ do}\ \{x \leftarrow \widehat{r}; \mathbf{return}\ x\}\}$$

This probabilistic program has expected value 2, which is a fixed point of the equation $r = r \times 0.5 + 1$. This suggests that whatever correctness result we prove for **estimator** in the presence of recursion should relate the derived estimator to the (real-valued) fixed point of the recursive definition of type $\mathfrak{R}_{\geq 0}$, if it exists.

These observations motivate our choice of ω-qbs interpretation for $\mathfrak{R}_{\geq 0}$: we equip the qbs denotation $\overline{\mathbb{R}}^n_{\geq 0}$ with the partial order

$$(r_1, \dots, r_n) \leq_{\mathfrak{R}_{\geq 0}} (r'_1, \dots, r'_n) \iff \forall i \in \{1, \dots, n\}, r_i \leq r'_i,$$

where $\leq$ is the standard linear order on $\overline{\mathbb{R}}_{\geq 0} = [0, \infty]$. Note that this order admits

a bottom element $(0, \ldots, 0)$, and that our space is closed under least upper bounds, which are taken componentwise. (In each component, a non-decreasing chain $r_0 \leq r_1 \leq \ldots$ either has a limit in $\mathbb{R}$ or has supremum ∞.) Note that under this interpretation, the definition $e_{\Re}$ denotes the supremum of the chain $0, 1, 1\frac{1}{2}, 1\frac{3}{4}, 1\frac{7}{8}, \ldots$, converging to 2 instead of immediately diverging to $\bot$.

3.4.2. Challenges for correctness

Recall the typing rule for recursion:

$$\textsc{Mu}\ \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mu x.\, e : \tau}$$

Since our correctness proof is by induction on the typing derivation of a term, the addition of this rule means one new case to handle in our proof: we must show that if $(\llbracket e \rrbracket(x \mapsto v), \llbracket \textbf{estimator}\{e\} \rrbracket(x \mapsto \widehat{v})) \in R^{\sim}_{\tau}$ whenever $(v, \widehat{v}) \in R^{\sim}_{\tau}$, then $(\llbracket \mu x.\, e \rrbracket, \llbracket \mu x.\textbf{estimator}\{e\} \rrbracket) \in R^{\sim}_{\tau}$. The standard logical relations approach to this case is to show that (1) $(\bot_{\tau}, \bot_{\textbf{estimator}\{\tau\}}) \in R^{\sim}_{\tau}$, (2) therefore $(v_i, \widehat{v_i}) \in R^{\sim}_{\tau}$ for each i (where $v_0 = \bot, v_1 = \llbracket e \rrbracket(x \mapsto v_0), \ldots$, and similarly for $\widehat{v_i}$), and (3) this implies $(\sup v_i, \sup \widehat{v_i}) \in R^{\sim}_{\tau}$. A relation $R^{\sim}_{\tau}$ for which (1) and (3) hold—i.e., a relation that contains $(\bot_{\tau}, \bot_{\textbf{estimator}\{\tau\}})$ and that is closed under suprema—is called **admissible**.

Unfortunately, it is not obvious that we can reason about unbiased estimation using this strategy. To see why, consider our recursive program from earlier, $e_{\Re}$, but with the following estimation strategy for $+_{\Re}$:

$$\begin{aligned}
&\textbf{estimator}\{+_{\Re}^{\texttt{SAMPLE}_2}\} := \lambda(\widehat{x}, \widehat{y}).\ \textbf{do}\ \{\\
&\quad b \leftarrow \mathit{flip}(0.5);\\
&\quad \tilde{s}_1 \leftarrow \textbf{if}\ b\ \textbf{then}\ \widehat{x}\ \textbf{else}\ \widehat{y};\\
&\quad \tilde{s}_2 \leftarrow \textbf{if}\ b\ \textbf{then}\ \widehat{x}\ \textbf{else}\ \widehat{y};\\
&\quad \textbf{return}\ (\tilde{s}_1 + \tilde{s}_2)\\
&\}
\end{aligned}$$

The chain of denotations for $e_{\Re} = \mu r.\ r \times_{\Re} \mathit{cast}_{\mathbb{R}_{>0} \to \Re_{\geq 0}}(0.5) +_{\Re} \mathit{cast}_{\mathbb{R}_{>0} \to \Re_{\geq 0}}(1)$ and its estimators are then:

$$\begin{aligned}
v_0 &= 0 & \widehat{v_0} &= \mathbf{0}\\
v_1 &= 1 & \widehat{v_1} &= 0.5 \cdot \delta_2\\
v_2 &= 1.5 & \widehat{v_2} &= 0.625 \cdot \delta_2\\
v_3 &= 1.75 & \widehat{v_3} &= 0.6953125 \cdot \delta_2\\
v_4 &= 1.875 & \widehat{v_4} &= 0.7417297363 \cdot \delta_2\\
&\vdots & &\vdots
\end{aligned}$$

The chain on the left converges to 2 and on the right converges to δ_2, so in the limit we do get an unbiased (indeed, exact) estimator. But the chains converge at different rates, and at each step, it is not clear what "unbiasedness" relationship (if any) holds between v_i and $\widehat{v_i}$.

3.4.3. Correctness via auxiliary logical relations

Intuition for our approach. Our observation is that although the subprobability measures $\widehat{v_i}$ are not directly unbiased estimators for v_i, they are *upper bounded* by a sequence of probability measures that *are* unbiased estimators for the values v_i:

$$
\begin{array}{lll}
v_0 = 0 & \widehat{v_0} = \mathbf{0} & \leq \delta_0 \\
v_1 = 1 & \widehat{v_1} = 0.5 \cdot \delta_2 & \leq 0.5 \cdot \delta_2 + 0.5 \cdot \delta_0 \\
v_2 = 1.5 & \widehat{v_2} = 0.625 \cdot \delta_2 & \leq 0.625 \cdot \delta_2 + 0.25 \cdot \delta_1 + 0.125 \cdot \delta_0 \\
v_3 = 1.75 & \widehat{v_3} = 0.6953125 \cdot \delta_2 & \leq 0.6953125 \cdot \delta_2 + 0.15625 \cdot \delta_{1.5} + 0.109375 \cdot \delta_1 + 0.03125 \cdot \delta_{0.5} + 0.0078125 \cdot \delta_0 \\
\vdots & \vdots &
\end{array}
$$

These are upper bounds according to the order on Meas $\mathbb{R}_\perp$, the denotation of $P\,\mathbb{R}$. The key insight is that if the estimator program is *almost-surely terminating*, then its denotation will be a probability measure assigning no mass to $\perp$, and so the only probability measure greater than or equal to it is itself. It must then equal its own unbiased upper bound.

How do we get this chain of unbiased upper bounds? They can be computed mechanically in $[\![P_{lazy}\ \mathfrak{R}_{\geq 0}]\!]$. We start with δ_0, the bottom element of this space, which is trivially an upper bound on the zero measure $\mathbf{0}$. We then iterate the recursive definition for our estimator, but using a modified estimation rule that builds programs of type $P_{lazy}\ \mathfrak{R}_{\geq 0}$ instead of $P\,\mathbb{R}$:

$$
\begin{array}{l}
\textbf{estimator}\{+_{\mathfrak{R}}\}_L^{\dagger,\texttt{SAMPLE}_2} := \lambda(\widehat{x}, \widehat{y}).\ \mathbf{do}_{lazy}\ \{ \\
\quad b \leftarrow \mathit{flip}(0.5); \\
\quad \tilde{s}_1 \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \widehat{x}\ \mathbf{else}\ \widehat{y}; \\
\quad \tilde{s}_2 \leftarrow \mathbf{if}\ b\ \mathbf{then}\ \widehat{x}\ \mathbf{else}\ \widehat{y}; \\
\quad \mathbf{return}_{lazy}\ (\tilde{s}_1 +_{\mathfrak{R}} \tilde{s}_2)\}
\end{array}
$$

The key difference between the non-lazy version and the lazy version is that the non-lazy version requires *both* $\tilde{s}_1$ and $\tilde{s}_2$ to yield non-bottom values in order for it to return a non-bottom value, whereas in the lazy version, if $\tilde{s}_1$ or $\tilde{s}_2$ happen to be 0, their sum may still be non-zero.

Formalizing our approach. Formally, we introduce an auxiliary transformation $\textbf{estimator}\{\cdot\}^\dagger$, used solely for the correctness proof. This transformation maps expressions of type $\mathfrak{R}_{\geq 0}^n$ not to the usual strict estimator of type $P\,(\mathbb{R} \times \cdots \times \mathbb{R})$, but to a lazy estimator of type $P_{lazy}\,(\mathfrak{R}_{\geq 0}^n \times (\mathbb{R} \times \cdots \times \mathbb{R}))$.

The auxiliary transformation $\textbf{estimator}\{\cdot\}^\dagger$ is defined compositionally. On primitives

Listing 3.10 Modified specification for the **estimator**† program transformation, to support recursion. A well-typed source-language expression $\Gamma \vdash e : \tau$ is transformed into a new well-typed expression $\textbf{estimator}\{\Gamma\}^\dagger \vdash \textbf{estimator}\{e\}^\dagger : \textbf{estimator}\{\tau\}^\dagger$, such that

$$(\gamma_1, \gamma_2) \in R^{\sim\dagger}_\Gamma \implies (\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{estimator}\{e\}^\dagger \rrbracket(\gamma_2)) \in R^{\sim\dagger}_\tau.$$

Source type τ	**Transformed type** $\textbf{estimator}\{\tau\}^\dagger$ **and spec** $R^{\sim\dagger}_\tau$
$\Re^n_{\geq 0}$	$P_{lazy}(\Re^n_{\geq 0} \times (\mathbb{R} \times \cdots \times \mathbb{R}))$ $R^{\sim\dagger}_{\Re^n_{\geq 0}} = \{(r, \widehat{r}) \mid \forall i \in \{1, \ldots, n\},$ $\mathbb{E}_{\widehat{r}}[\pi_i \circ \pi_1] = \pi_i r \wedge \mathbb{P}_{(x_L, x_S) \sim \widehat{r}}[\pi_i x_S \in \{\bot, \pi_i x_L\}] = 1\}$

f_{ext} operating on $\Re^n_{\geq 0}$, it provides overloaded implementations that produce *both* a lazy and a strict result. For instance, for $+_\Re$:

$$\textbf{estimator}\{+_\Re\}^{\dagger,\texttt{SUM}} = \lambda(\widehat{x}, \widehat{y}).\ \textbf{do}_{lazy}\{(\tilde{x}_L, \tilde{x}_S) \leftarrow \widehat{x}; (\tilde{y}_L, \tilde{y}_S) \leftarrow \widehat{y};\ \textbf{return}_{lazy}\ (\tilde{x}_L +_\Re \tilde{y}_L, \tilde{x}_S + \tilde{y}_S)\}$$

If either $\widehat{x}$ or $\widehat{y}$ diverges, then at least one of $\tilde{x}_S$ or $\tilde{y}_S$ will be $\bot$, causing the strict result (their sum) to also be $\bot$. By contrast, if either $\tilde{x}_L$ or $\tilde{y}_L$ is 0 (the bottom element in $\llbracket \Re_{\geq 0} \rrbracket$), their sum under $+_\Re$ may still be non-zero.

We define a new logical relation $R^{\sim\dagger}_{\Re^n_{\geq 0}}$ for the augmented transformation. A tuple $(r, \widehat{r}) \in \overline{\mathbb{R}}^n_{\geq 0} \times \text{Prob}\,(\overline{\mathbb{R}}^n_{\geq 0} \times (\mathbb{R}_\bot)^n)$ is in $R^{\sim\dagger}_{\Re^n_{\geq 0}}$ if:

1. **Lazy estimator is unbiased:** The idealized lazy estimator $\pi_{1*}\widehat{R} \in \text{Prob}\ \overline{\mathbb{R}}^n_{\geq 0}$ correctly computes the expected value: for all $i \in \{1, \ldots, n\}$, $\mathbb{E}_{\widehat{R}}[\pi_i \circ \pi_1] = \pi_i r$.
2. **Lazy estimator is coupled to strict estimator:** for $(x_L, x_S) \sim \widehat{r}$, for each $i \in \{1, \ldots, n\}$, we have $\pi_i x_S \in \{\bot, \pi_i x_L\}$ with probability 1.

The second criterion forces the strict and lazy results to agree unless the strict result is $\bot$. It implies that each lazy marginal is an upper bound on each strict marginal: the lazy marginal must be equal to the restriction of the strict marginal to $\mathbb{R}$, plus a "remainder" measure whose total mass is exactly the mass that the strict marginal assigns to $\bot$.

This logical relation is admissible, and so we can prove the fundamental lemma for $\textbf{estimator}\{\cdot\}^\dagger$ in a setting with recursion.

Lemma 3 (Fundamental lemma for **integrator** and **estimator**† with recursion)**.** *Suppose $\Gamma \vdash e : \tau$ is a well-typed open term. Then:*

- *If $(\gamma_1, \gamma_2) \in R^{\int}_\Gamma$, then $(\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2)) \in R^{\int}_\tau$.*
- *If $(\gamma_1, \gamma_2) \in R^{\sim\dagger}_\Gamma$, then $(\llbracket e \rrbracket(\gamma_1), \llbracket \textbf{estimator}\{e\}^\dagger \rrbracket(\gamma_2)) \in R^{\sim\dagger}_\tau$.*

Proof sketch. The proof proceeds by induction on the typing derivation. Most of the proof matches our proofs of Lemmas 1 and 2, with two key exceptions. First, we need to extend the proofs to handle the new typing rule Mu. Second, for **estimator**†, the logical relation at the type $\mathfrak{R}^n_{\geq 0}$ has changed, so we need to verify that primitives f_{ext} have correct translations relative to the new specification.

First, we focus on recursion with μ.

Let $\Gamma \vdash \mu x.\, e : \tau$ be derived using the Mu rule, with premise $\Gamma, x : \tau \vdash e : \tau$. Assume the lemma holds for the premise, i.e., for all $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$ and $(v, \widehat{v}) \in R^{\int}_{\tau}$, we have $(\llbracket e \rrbracket(\gamma_1[x \mapsto v]), \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2[x \mapsto \widehat{v}])) \in R^{\int}_{\tau}$. We need to show that for any $(\gamma_1, \gamma_2) \in R^{\int}_{\Gamma}$, we have $(\llbracket \mu x.\, e \rrbracket(\gamma_1), \llbracket \mu x.\, \textbf{integrator}\{e\} \rrbracket(\gamma_2)) \in R^{\int}_{\tau}$.

The standard technique requires showing that the logical relation $R^{\int}_{\tau}$ is **admissible**, meaning it contains the bottom pair $(\bot_{\llbracket \tau \rrbracket}, \bot_{\llbracket \textbf{integrator}\{\tau\} \rrbracket})$ and is closed under suprema of ω-chains. At non-probabilistic types, this holds trivially, so we focus on $\tau = P\ \tau'$.

- **Bottom:** We have $\bot_{\llbracket P\ \tau' \rrbracket} = \mathbf{0}$ (zero measure) and

$$\bot_{\llbracket \textbf{integrator}\{P\ \tau'\} \rrbracket} = \bot_{(\llbracket \textbf{integrator}\{\tau'\} \rrbracket \Rightarrow \overline{\mathbb{R}}_{\geq 0}) \Rightarrow \overline{\mathbb{R}}_{\geq 0}} = (g \mapsto 0).$$

  The pair $(\mathbf{0}, g \mapsto 0)$ is in $R^{\int}_{P\ \sigma}$, because $\int g_1 d\mathbf{0} = 0$, and we need $(0, 0) \in R^{\int}_{\mathfrak{R}_{\geq 0}}$, which holds.

- **Suprema:** Let $(\mu_k, \mathbb{E}_k)_{k \in \mathbb{N}}$ be a non-decreasing chain in $R^{\int}_{P\ \tau'}$. Let $\mu = \sup_k \mu_k$ and $\mathbb{E} = \sup_k \mathbb{E}_k$. For any (g_1, g_2) in the function relation, we need $(\int g_1 d\mu, \mathbb{E}(g_2)) \in R^{\int}_{\mathfrak{R}_{\geq 0}}$. Since integration and function application are Scott-continuous, $\int g_1 d\mu = \sup_k \int g_1 d\mu_k$ and $\mathbb{E}(g_2) = \sup_k \mathbb{E}_k(g_2)$. Since $(\int g_1 d\mu_k, \mathbb{E}_k(g_2)) \in R^{\int}_{\mathfrak{R}_{\geq 0}}$ for all k, and $R^{\int}_{\mathfrak{R}_{\geq 0}}$ (identity on $\overline{\mathbb{R}}^n_{\geq 0}$) is closed under suprema, the result holds.

Since $R^{\int}_{\tau}$ is admissible, the standard proof for recursion applies. Let $\mu_k = (\mu \mapsto \llbracket e \rrbracket(\gamma_1[x \mapsto \mu]))^k(\mathbf{0})$ and $\mathbb{E}_k = (\mathbb{E} \mapsto \llbracket \textbf{integrator}\{e\} \rrbracket(\gamma_2[x \mapsto \mathbb{E}]))^k(g \mapsto 0)$. By induction on k (and applying our inductive hypothesis for e), $(\mu_k, \mathbb{E}_k) \in R^{\int}_{P\ \tau'}$ for all k. Then the limits of the chains, $\mu = \sup \mu_k$ and $\mathbb{E} = \sup \mathbb{E}_k$, are in the relation by its closure under suprema.

The proof for **estimator**$\{\cdot\}^\dagger$ follows the same structure, using the admissibility of $R^{\sim\dagger}$.

- **Bottom:** For $\tau = \mathfrak{R}^n_{\geq 0}$, the bottom pair is $((0, \ldots, 0), \delta_{((0,\ldots,0),(\bot,\ldots,\bot))})$. We verify the conditions: (1) $\mathbb{E}_{\delta_{((0,\ldots,0),(\bot,\ldots,\bot))}}[\pi_i \circ \pi_1] = 0 = \pi_i((0, \ldots, 0))$, holds. (2) With probability 1 for $(x_L, x_S) \sim \delta_{((0,\ldots,0),(\bot,\ldots,\bot))}$, $\pi_i x_S = \bot \in \{\bot, \pi_i x_L\}$, as required.

- **Suprema:** Let $(r_k, \widehat{r_k})$ be an ω-chain in $R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$. Let $r = \sup r_k$ and $\widehat{r} = \sup \widehat{r_k}$. Condition (1) $\mathbb{E}_{\widehat{r}}[\pi_i \circ \pi_1] = \pi_i r$ holds by Scott-continuity of expectation and of π_i. Condition

(2) holds for a more subtle reason. Define

$$f_1^i(x_L, x_S) := \begin{cases} (0,0) & \pi_i x_S = \bot \\ (\pi_i x_L, \pi_i x_L) & \text{otherwise} \end{cases} \qquad f_2^i(x_L, x_S) := \begin{cases} (0,0) & \pi_i x_S = \bot \\ (\pi_i x_L, \pi_i x_S) & \text{otherwise} \end{cases}$$

and consider the pushforwards $f_{1*}^i \widehat{r_k}$ and $f_{2*}^i \widehat{r_k}$. That the relation holds for each $\widehat{r_k}$ tells us that for each k and i, these two pushforwards are equal. Since f_1 and f_2 are Scott-continuous, $f_{1*}^i \widehat{r} = \sup_{k\in\mathbb{N}} f_{1*}^i \widehat{r_k} = \sup_{k\in\mathbb{N}} f_{2*}^i \widehat{r_k} = f_{2*}^i \widehat{r}$. Now, the pushforward of any measure by f_1^i has the property that both outputs are equal with probability with 1, and since the pushforward by f_1 is equal to the pushforward by f_2, this property must hold for $f_{2*}^i \widehat{r}$. Inspecting f_2, this can only be the case if with probability 1 under $\widehat{r}$, either $\pi_i x_S = \bot$ or $\pi_i x_S = \pi_i x_L$, as desired.

Thus $R^{\sim\dagger}$ is admissible, and the recursion step holds for $\textbf{estimator}\{\cdot\}^\dagger$.

Second, we must verify the primitive translations for $\textbf{estimator}\{\cdot\}^\dagger$ satisfy the new specification $R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$. Consider $+_\mathfrak{R}$ with the `SUM` strategy. Assume inputs $(r_x, \widehat{x})$ and $(r_y, \widehat{y})$ satisfy $R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$. The transformation yields $\widehat{z} = \mathbf{do}_{lazy}\{(\tilde{x}_L, \tilde{x}_S) \leftarrow \widehat{x}; (\tilde{y}_L, \tilde{y}_S) \leftarrow \widehat{y}; \mathbf{return}_{lazy}\ (\tilde{x}_L +_\mathfrak{R} \tilde{y}_L, \tilde{x}_S + \tilde{y}_S)\}$. Let $r_z = r_x + r_y$. We must show $(r_z, \widehat{z}) \in R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$.

1. **Unbiasedness:** $\mathbb{E}_{\widehat{z}}[\pi_i \circ \pi_1] = \mathbb{E}_{(\dots)\sim\widehat{x},(\dots)\sim\widehat{y}}[\pi_i(\tilde{x}_L +_\mathfrak{R} \tilde{y}_L)]$. By the inductive hypothesis, $\mathbb{E}[\pi_i \tilde{x}_L] = \pi_i r_x$ and $\mathbb{E}[\pi_i \tilde{y}_L] = \pi_i r_y$. By linearity of expectation over $\overline{\mathbb{R}}_{\geq 0}$, the result is $\pi_i r_x + \pi_i r_y = \pi_i r_z$.
2. **Coupling:** Let $(z_L, z_S) \sim \widehat{z}$. Then $z_L = \tilde{x}_L +_\mathfrak{R} \tilde{y}_L$ and $z_S = \tilde{x}_S + \tilde{y}_S$, where $(\tilde{x}_L, \tilde{x}_S) \sim \widehat{x}$ and $(\tilde{y}_L, \tilde{y}_S) \sim \widehat{y}$. By the inductive hypothesis, $\pi_i \tilde{x}_S \in \{\bot, \pi_i \tilde{x}_L\}$ and $\pi_i \tilde{y}_S \in \{\bot, \pi_i \tilde{y}_L\}$. If $\pi_i \tilde{x}_S = \bot$ or $\pi_i \tilde{y}_S = \bot$, then $\pi_i z_S = \pi_i \tilde{x}_S + \pi_i \tilde{y}_S = \bot$, satisfying the condition. If both $\pi_i \tilde{x}_S \neq \bot$ and $\pi_i \tilde{y}_S \neq \bot$, then $\pi_i \tilde{x}_S = \pi_i \tilde{x}_L$ and $\pi_i \tilde{y}_S = \pi_i \tilde{y}_L$. In this case, $\pi_i z_S = \pi_i \tilde{x}_L + \pi_i \tilde{y}_L = \pi_i z_L$. Thus, $\pi_i z_S \in \{\bot, \pi_i z_L\}$ holds with probability 1.

Other primitives are verified similarly. □

Lemma 4 (Relationship between $\textbf{estimator}^\dagger$ and **estimator**). *Let $e : \mathfrak{R}^n_{\geq 0}$ be a closed expression. Then $\pi_{2*}[\![\textbf{estimator}\{e\}^\dagger]\!] = \mathbf{lift}_{M\to P}([\![\textbf{estimator}\{e\}]\!])$, where $\mathbf{lift}(\mu) = \mu + (1 - |\mu|) \cdot \delta_\bot$ lifts a subprobability measure to a probability measure by adding mass at $\bot$.*

Proof. We introduce an auxiliary logical relation T_τ between $[\![\textbf{estimator}\{\tau\}]\!]$ and $[\![\textbf{estimator}\{\tau\}^\dagger]\!]$ for each type τ. At $\tau = \mathfrak{R}^n_{\geq 0}$, we define

$$T_{\mathfrak{R}^n_{\geq 0}} = \{(\mu, \nu) \mid \mu \in \text{Meas}\ (\mathbb{R}_\bot)^n, \nu \in \text{Prob}\ (\overline{\mathbb{R}}^n_{\geq 0} \times (\mathbb{R}_\bot)^n), \pi_{2*}\nu = \mathbf{lift}(\mu)\},$$

and at other types, we use the standard definitions (following the same pattern as in the definition of $R^\sim_\tau$).

Fundamental lemma for T. We prove by induction on the typing derivation of $\Gamma \vdash e : \tau$ that if $(\gamma, \gamma') \in T_\Gamma$, then $([\![\textbf{estimator}\{e\}]\!](\gamma), [\![\textbf{estimator}\{e\}^\dagger]\!](\gamma')) \in T_\tau$.

- **Base cases (variables, constants, non-$\mathfrak{R}$ Primitives):** Here $\textbf{estimator}\{\cdot\}$ and $\textbf{estimator}\{\cdot\}^\dagger$ act as identity, and the lemma holds trivially.
- **Structural cases (pairs, projections, lambdas, applications, sums, matches, return, do):** These follow standard logical relation arguments, as both transformations distribute over the structure, and the definition of T mirrors this.
- **Recursive definitions:** This follows from the admissibility of $T_{\mathfrak{R}^n_{\geq 0}}$ (which follows from the Scott-continuity and strictness of **lift**).
- **Extended real primitives (f_{ext} operating on $\mathfrak{R}^n_{\geq 0}$):** This is the main case. We must show that $([\![\textbf{estimator}\{f_{ext}\}]\!], [\![\textbf{estimator}\{f_{ext}\}^\dagger]\!]) \in T_{\mathsf{ArgType}(f_{ext})\to\mathsf{RetType}(f_{ext})}$.

  We illustrate the logic by working out the case for $+_{\mathfrak{R}} : \mathfrak{R}_{\geq 0} \times \mathfrak{R}_{\geq 0} \to \mathfrak{R}_{\geq 0}$ with the `SUM` strategy. Let $f = [\![\textbf{estimator}\{+_{\mathfrak{R}}\}^{\texttt{SUM}}]\!]$ and $g = [\![\textbf{estimator}\{+_{\mathfrak{R}}\}^{\dagger,\texttt{SUM}}]\!]$. We need to show $(f, g) \in T_{\mathfrak{R}_{\geq 0}\times\mathfrak{R}_{\geq 0}\to\mathfrak{R}_{\geq 0}}$. Let $(\mu_x, \nu_x) \in T_{\mathfrak{R}_{\geq 0}}$ and $(\mu_y, \nu_y) \in T_{\mathfrak{R}_{\geq 0}}$. We must show $(f(\mu_x, \mu_y), g(\nu_x, \nu_y)) \in T_{\mathfrak{R}_{\geq 0}}$. That is, letting $\mu_z = f(\mu_x, \mu_y)$ and $\nu_z = g(\nu_x, \nu_y)$, we require $\pi_{2*}\nu_z = \textbf{lift}(\mu_z)$. Recall the definitions:

$$f(\mu_x, \mu_y) = \textbf{do}\{x \leftarrow \mu_x; y \leftarrow \mu_y; \textbf{return } (x + y)\}$$
$$g(\nu_x, \nu_y) = \textbf{do}_{lazy}\{(x_L, x_S) \leftarrow \nu_x; (y_L, y_S) \leftarrow \nu_y; \textbf{return}_{lazy } (x_L +_{\mathfrak{R}} y_L, x_S + y_S)\}$$

  So we have:

$$\begin{aligned}
\pi_{2*}\nu_z &= \iint \delta_{x_S+y_S}\ \pi_{2*}\nu_x(dx_S)\pi_{2*}\nu_y(dy_S) \\
&= \iint \delta_{x_S+y_S}\ \textbf{lift}(\mu_x)(dx_S)\textbf{lift}(\mu_y)(dy_S) \\
&= (1 - |\mu_y|)\delta_\perp + \iint \delta_{x_S+y}\ \textbf{lift}(\mu_x)(dx_S)\mu_y(dy) \\
&= (1 - |\mu_y|)\delta_\perp + \iint (1 - |\mu_x|)\delta_\perp + \iint \delta_{x+y}\mu_x(dx)\mu_y(dy) \\
&= (1 - |\mu_y| + |\mu_y|(1 - |\mu_x|))\delta_\perp + \iint \delta_{x+y}\mu_x(dx)\mu_y(dy) \\
&= (1 - |\mu_y||\mu_x|)\delta_\perp + \iint \delta_{x+y}\mu_x(dx)\mu_y(dy) \\
&= (1 - |\mu_z|)\delta_\perp + \iiint \delta_{x+y}\mu_z(dx, dy) \\
&= \textbf{lift}(\mu_z)
\end{aligned}$$

By carrying out this argument for each primitive in the language, we complete the proof of the fundamental lemma for T.

Proof of Lemma 4. Apply the fundamental lemma for T to the closed expression $e : \mathfrak{R}^n_{\geq 0}$ in the empty context $\cdot$. The empty environments $([], [])$ are trivially related by $T_\cdot$. The lemma implies $([\![\textbf{estimator}\{e\}]\!], [\![\textbf{estimator}\{e\}^\dagger]\!]) \in T_{\mathfrak{R}^n_{\geq 0}}$. By the definition

of $T_{\mathfrak{R}^n_{\geq 0}}$, this establishes that $\pi_{2*}[\![\textbf{estimator}\{e\}^\dagger]\!] = \textbf{lift}([\![\textbf{estimator}\{e\}]\!])$. □

Using these two lemmas, we can prove our overall correctness result for the language with recursion. Note that automatically derived estimators must now be almost surely terminating for the correctness guarantees to kick in.

Theorem 3 (Correctness of **integrator** and **estimator**, with recursion)**.** *Both program transformations are correct for the probabilistic language with recursion:*

- *Let e be a closed program of type $P\,\tau$. Then* **integrator**$\{e\}$ *denotes the integration operator corresponding to the measure $[\![e]\!]$.*
- *Let e be a closed program of type $\mathfrak{R}^n_{\geq 0}$. Then if* **estimator**$\{e\}$ *is an almost-surely terminating probabilistic program (i.e.,* $[\![\textbf{estimator}\{e\}]\!]$ *is a probability measure assigning zero mass to any vectors containing $\bot$), then it implements an unbiased estimator of $[\![e]\!]$. That is, $\mathbb{E}_{[\![\textbf{estimator}\{e\}]\!]}[\pi_i] = \pi_i([\![e]\!])$ for all $i \in \{1, \ldots, n\}$.*

Proof of Theorem 3. We apply Lemma 3 to the closed term e in the empty context $\cdot$. Since $R^{\int}_{\cdot} = R^{\sim\dagger}_{\cdot} = \{([], [])\}$, the lemma yields:

- If $e : P\,\tau'$, then $([\![e]\!], [\![\textbf{integrator}\{e\}]\!]) \in R^{\int}_{P\,\tau'}$. By definition of $R^{\int}_{P\,\tau'}$, this means $[\![\textbf{integrator}\{e\}]\!]$ is the integration operator for $[\![e]\!]$.
- If $e : \mathfrak{R}^n_{\geq 0}$, then $(r, \widehat{r}) \in R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$, where $r = [\![e]\!]$ and $\widehat{r} = [\![\textbf{estimator}\{e\}^\dagger]\!]$. Let $\widehat{r_L} = \pi_{1*}\widehat{r}$ and $\widehat{r_S} = \pi_{2*}\widehat{r} = \textbf{lift}([\![\textbf{estimator}\{e\}]\!])$. The definition of $R^{\sim\dagger}_{\mathfrak{R}^n_{\geq 0}}$ gives:

  1. $\mathbb{E}_{\widehat{r}}[\pi_i \circ \pi_1] = \pi_i r$ for all i.
  2. $\mathbb{P}_{(x_L, x_S)\sim\widehat{r}}[\pi_i x_S \in \{\bot, \pi_i x_L\}] = 1$ for all i.

  We are given that $[\![\textbf{estimator}\{e\}]\!]$ terminates almost surely, so: (a) we have that $\widehat{r_S} = \textbf{lift}([\![\textbf{estimator}\{e\}]\!]) = [\![\textbf{estimator}\{e\}]\!]$, and (b) $\mathbb{P}_{(x_L, x_S)\sim\widehat{r}}[\pi_i x_S = \bot] = 0$ for all i. From condition (2), if $x_S \neq \bot$, then $\pi_i x_S = \pi_i x_L$ for all i. Since $\pi_i x_S \neq \bot$ almost surely for each i, this implies $x_S = x_L$ almost surely (as vectors in $\mathbb{R}^n$). Since x_L is unbiased (condition (1) above), so is x_S. Thus, $\mathbb{E}_{[\![\textbf{estimator}\{e\}]\!]}[\pi_i] = \pi_i([\![e]\!])$ as required.

□

3.5 Signed integrands

So far, we have worked with only non-negative extended reals, using the type $\mathfrak{R}_{\geq 0}$. In this section, we define the type $\mathfrak{R}$ of general integral results.

The key challenge is to make this extension play nice with recursion. Our semantics of $\mathfrak{R}_{\geq 0}$ equipped it with the linear order, with bottom element 0. Maps such as $+_{\mathfrak{R}}$ and $\times_{\mathfrak{R}}$ are Scott-continuous under this order. But it is unclear what order to choose on the larger space of positive and negative extended reals. We would ideally like

Listing 3.11 Primitive operations in $\lambda_{\mathfrak{R}}$ (Listing 3.1) with formal-difference semantics.

Primitive f_{ext}	$\mathsf{Arg}(f_{ext}) \to \mathsf{Ret}(f_{ext})$	$\mathsf{Operation}(f_{ext})$
$cast_{\mathbb{R}^n \to \mathfrak{R}^n}$	$\mathbb{R} \times \cdots \times \mathbb{R} \to \mathfrak{R}^n$	$(x_1, \ldots, x_n) \mapsto \left(\begin{cases} (x_i, 0) & \text{if } x_i \geq 0 \\ (0, -x_i) & \text{if } x_i < 0 \\ (0,0) & \text{if } x_i = \bot \end{cases} \right)_{i=1}^{n}$
$+_{\mathfrak{R}}$	$\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$	$(x,y) \mapsto (x^+ + y^+, x^- + y^-)$
$-_{\mathfrak{R}}$	$\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$	$(x,y) \mapsto (x^+ + y^-, x^- + y^+)$
$\times_{\mathfrak{R}}$	$\mathfrak{R} \times \mathfrak{R} \to \mathfrak{R}$	$(x,y) \mapsto (x^+y^+ + x^-y^-, x^+y^- + x^-y^+)$
$exp_{\mathfrak{R}}$	$\mathfrak{R} \to \mathfrak{R}$	$x \mapsto (e^{x^+}\cosh(x^-), e^{x^+}\sinh(x^-))$
$\mathbf{E}^n_{f_{prb}}$	$\tau \to (\sigma \to \mathfrak{R}^n) \to \mathfrak{R}^n$ (for $f_{prb} : \tau \to P\,\sigma$)	$\theta \mapsto g \mapsto \left(\mathbb{E}_{f_{prb}(\theta)}[g^+], \mathbb{E}_{f_{prb}(\theta)}[g^-]\right)$

addition, multiplication, subtraction, and so on to all be Scott-continuous, but they are not even necessarily monotone for obvious choices of order on $\mathfrak{R}$.

3.5.1. The domain of formal differences

Our approach is to take $\mathfrak{R}$ to denote a space of **formal differences** $x^+ - x^-$. Formally, $[\![\mathfrak{R}^n]\!] = (\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n$, where each component of the vector is a pair (r_i^+, r_i^-) of a **positive part** r_i^+ and a **negative part** r_i^-. The value (∞, ∞) is used to represent the result of an integral that is undefined (e.g. $\int_{\mathbb{R}^2} x - y \, d(x,y)$).

We equip this space $[\![\mathfrak{R}^n]\!]$ with the standard *product order*:

$$(r_1^+, r_1^-), \cdots \leq ((r_1')^+, (r_1')^-), \cdots \iff \forall i. r_i^+ \leq (r_i')^+ \wedge r_i^- \leq (r_i')^-$$

where $\leq$ on the right is the standard order on $\overline{\mathbb{R}}_{\geq 0}$. This makes $[\![\mathfrak{R}^n]\!]$ an ω-qbs with bottom element $((0,0), \ldots, (0,0))$. Listing 3.11 gives the semantics of our primitives in terms of these formal differences. If f_{ext} is intended to represent some function $f : \overline{\mathbb{R}} \to \overline{\mathbb{R}}$, then we aim to choose a formal-differences denotation $[\![f_{ext}]\!]$ such that if $(y^+, y^-) = [\![f_{ext}]\!](x^+, x^-)$, then $f(x^+ - x^-) = y^+ - y^-$. When such a choice exists, we say that the function f is **representable** in $\mathfrak{R}$. Many functions which are not monotone as functions on $\overline{\mathbb{R}}$ are nevertheless representable (e.g., subtraction).

Remark 16 (Representability in $\mathfrak{R}$ and user-facing limitations)**.** Note that representability is not a restriction on the *integrals* that users can express, but rather on the *functions of integrals* that can be expressed. Any function $f : \mathbb{R} \to \mathbb{R}$ can be composed with $cast_{\mathbb{R} \to \mathfrak{R}}$ to yield an *integrand* $g : \mathbb{R} \to \mathfrak{R}$ that can be integrated with respect to a measure on $\mathbb{R}$. But once the integration operator has been applied, a value of type $\mathfrak{R}$ is obtained, and only representable functions can be applied to it. So, for example, we can express nested integrals, sums, differences, and products of integrals, as well as exponentiation of integrals or the sine of an integral. However,

Listing 3.12 Modified specification for the **estimator** program transformation, to support signed integrands.

Source type τ	**Transformed type** $\textbf{estimator}\{\tau\}^\dagger$ **and spec** $R_\tau^{\sim\dagger}$
$\mathfrak{R}^n$	$P_{lazy}(\mathfrak{R}^n \times ((\mathbb{R}\times\mathbb{R}) \times \cdots \times (\mathbb{R}\times\mathbb{R})))$
	$R_{\mathfrak{R}^n}^{\sim\dagger} = \{(r, \widehat{r}) \mid \forall i \in \{1,\dots,n\}, \mathbb{E}_{\widehat{r}}[\pi_i \circ \pi_1] = \pi_i r \wedge$ $\mathbb{P}_{(x_L,x_S)\sim\widehat{r}}[\pi_i x_S \in \{(\bot,\bot),((\pi_i x_L)^+,(\pi_i x_L)^-)\}] = 1\}$

we cannot *divide* two integrals: division is not representable in $\mathfrak{R}$. Interestingly, if there exists a valid translation $\textbf{estimator}\{f\}$ for a function f, which transforms measures on $\mathbb{R}$ that unbiasedly estimate a value x into measures that unbiasedly estimate the value $f(x)$, then f is representable in $\mathfrak{R}$. Thus, unrepresentability is not just an incidental property of our semantics, but rather a fundamental property of the task of automatically and compositionally estimating functions of integrals.

3.5.2. Automatic signed integration

The $\textbf{integrator}\{\cdot\}$ transformation can easily be extended to handle the new type $\mathfrak{R}^n$.

- The target type becomes $\textbf{integrator}\{P\ \tau\} = (\textbf{integrator}\{\tau\} \to \mathfrak{R}^n) \to \mathfrak{R}^n$.
- Primitives $\mathbf{E}^n_{f_{prb}}$ now take integrands $g : \sigma \to \mathfrak{R}^n$ (semantically, $g : [\![\sigma]\!] \to (\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n$) and compute the expectation component-wise:

$$[\![\mathbf{E}^n_{f_{prb}}]\!](\theta)(g) = \left(\left(\int \pi_1(\pi_i(g(x)))\ \mu(dx), \int \pi_2(\pi_i(g(x)))\ \mu(dx)\right)\right)_{i=1}^n$$

  where $\mu = [\![f_{prb}]\!](\theta)$.

- The logical relation $R^{\int}_{P\ \tau}$ is now defined in terms of integrators of $\mathfrak{R}^n$-valued integrands: $R^{\int}_{P\ \tau} = \{(\mu, \mathbb{E}_\mu) \mid (g \mapsto \mu(g), \mathbb{E}_\mu) \in R^{\int}_{(\textbf{integrator}\{\tau\}\to\mathfrak{R}^n)\to\mathfrak{R}^n}\}$.

The correctness proof (Lemma 3 for $\textbf{integrator}\{\cdot\}$) goes through unchanged.

3.5.3. Automatic signed estimation

The $\textbf{estimator}\{\cdot\}$ transformation is extended to treat $\mathfrak{R}^n$ the same way it treats $\mathfrak{R}^n_{\geq 0}$:

$$\textbf{estimator}\{\mathfrak{R}^n\} = P\ (\mathbb{R} \times \cdots \times \mathbb{R}).$$

Intuitively, the estimator should be unbiased for the vector $(r_1^+ - r_1^-, \dots, r_n^+ - r_n^-)$.

However, as in the previous section, we do not prove unbiasedness of **estimator** directly. Rather, we use the auxiliary transformation $\textbf{estimator}\{\cdot\}^\dagger$, which we modify as follows to handle $\mathfrak{R}^n$:

- Target type: The $\mathbf{estimator}^\dagger$ transformation now maps $\mathfrak{R}^n$ to a type of programs that explicitly represent the positive and negative parts of the value being estimated:

$$\mathbf{estimator}\{\tau\}^\dagger = P_{lazy}(\mathfrak{R}^n \times ((\mathbb{R}_\perp \times \mathbb{R}_\perp) \times \cdots \times (\mathbb{R}_\perp \times \mathbb{R}_\perp)))$$

- Logical relation $R^{\sim\dagger}_{\mathfrak{R}^n}$: As defined in Listing 3.12, $(r, \widehat{r}) \in R^{\sim\dagger}_{\mathfrak{R}^n}$ where $r \in [\![\mathfrak{R}^n]\!]$ and $\widehat{r} \in [\![\mathbf{estimator}\{\mathfrak{R}^n\}^\dagger]\!]$ if for all $i \in \{1, \ldots, n\}$:

  1. **Unbiasedness:** $\mathbb{E}_{\widehat{r}}[\pi_i \circ \pi_1] = \pi_i r$. The expectation $\mathbb{E}_{\widehat{r}}[\cdot]$ is taken componentwise on the pair structure of $[\![\mathfrak{R}^n]\!]$, i.e., $(\mathbb{E}[\pi_1 \circ \pi_i \circ \pi_1], \mathbb{E}[\pi_2 \circ \pi_i \circ \pi_1]) = (\pi_1(\pi_i r), \pi_2(\pi_i r))$.
  2. **Coupling:** For $(x_L, x_S) \sim \widehat{r}$, the i-th component of the strict estimate $\pi_i x_S$ must match the corresponding component of the lazy estimate $\pi_i x_L$, unless the strict component is $(\perp, \perp)$. That is, $\mathbb{P}[\pi_i x_S \in \{(\perp, \perp), (\pi_1(\pi_i x_L), \pi_2(\pi_i x_L))\}] = 1$.

  This relation is admissible by the same arguments as used in the proof of Lemma 3.

- Overloaded primitives: Primitives are overloaded for $\mathbf{estimator}\{\cdot\}^\dagger$ to explicitly handle positive and negative parts. For example, for subtraction:

$$\begin{aligned}\mathbf{estimator}\{-_{\mathfrak{R}}\}^{\dagger,\mathtt{SUM}} = \lambda(\widehat{x}, \widehat{y}).\ \mathbf{do}_{lazy}\{&(x_L, (x_S^+, x_S^-)) \leftarrow \widehat{x};\\ &(y_L, (y_S^+, y_S^-)) \leftarrow \widehat{y};\\ &\mathbf{return}_{lazy}\ (x_L -_{\mathfrak{R}} y_L, (x_S^+ + y_S^-, x_S^- + y_S^+))\}\end{aligned}$$

  This implementation satisfies the conditions for $R^{\sim\dagger}_{\mathfrak{R}}$. Unbiasedness follows from linearity of expectation. Coupling holds because if either (x_S^+, x_S^-) or (y_S^+, y_S^-) is $(\perp, \perp)$, the resulting strict sum/difference is $(\perp, \perp)$; otherwise, the strict result $(x_S^+ + y_S^-, x_S^- + y_S^+)$ matches the lazy result $x_L -_{\mathfrak{R}} y_L = (x_L^+ + y_L^-, x_L^- + y_L^+)$ because $x_S^\pm = x_L^\pm$ and $y_S^\pm = y_L^\pm$.

Correctness theorem. The fundamental lemma (Lemma 3) holds for $\mathbf{integrator}\{\cdot\}$ and $\mathbf{estimator}\{\cdot\}^\dagger$ by an easy extension to the signed case. But we still need to relate the explicit formal differences produced by $\mathbf{estimator}^\dagger$ to the real-valued estimators produced by $\mathbf{estimator}$. We do this using an analogue of Lemma 4:

Lemma 5 (Relationship between $\mathbf{estimator}^\dagger$ and $\mathbf{estimator}$ (Signed))**.** *Let $e : \mathfrak{R}^n$ be a closed expression. Define $\phi : ((\mathbb{R}_\perp \times \mathbb{R}_\perp) \times \cdots \times (\mathbb{R}_\perp \times \mathbb{R}_\perp)) \to (\mathbb{R}_\perp \times \cdots \times \mathbb{R}_\perp)$ by $\phi((y_1^+, y_1^-), \ldots, (y_n^+, y_n^-)) = (y_1^+ - y_1^-, \ldots, y_n^+ - y_n^-)$, where $y^+ - y^- = \perp$ if $y^+ = \perp$ or $y^- = \perp$. Then $(\phi \circ \pi_2)_*[\![\mathbf{estimator}\{e\}^\dagger]\!] = \mathbf{lift}([\![\mathbf{estimator}\{e\}]\!])$.*

Proof sketch. Define a logical relation T_τ between $[\![\mathbf{estimator}\{\tau\}]\!]$ and $[\![\mathbf{estimator}\{\tau\}^\dagger]\!]$. For $\tau = \mathfrak{R}^n$, let $T_{\mathfrak{R}^n} = \{(\mu, \nu) \mid (\phi \circ \pi_2)_*\nu = \mathbf{lift}(\mu)\}$. Extend T to other types in the standard way. Prove the fundamental lemma for T by induction, showing primitives like $\mathbf{estimator}\{-_{\mathfrak{R}}\}$ preserve the relation. The main lemma follows by applying the fundamental lemma to e. □

We can then prove the extended correctness theorem for signed integration and estimation.

Theorem 4 (Correctness of **integrator** and **estimator** with signed integrands)**.** *Both transformations are correct for the probabilistic language with recursion and signed integrands:*

- *Let e be a closed program of type $P\ \tau$. Then* **integrator**$\{e\}$ *denotes the integration operator corresponding to the measure $[\![e]\!]$, mapping functions* **integrator**$\{\tau\} \to \mathfrak{R}^n$ *to $\mathfrak{R}^n$.*
- *Let e be a closed program of type $\mathfrak{R}^n$. Let $r = [\![e]\!] \in [\![\mathfrak{R}^n]\!]$. If* **estimator**$\{e\}$ *is an almost-surely terminating probabilistic program (i.e.,* $[\![$**estimator**$\{e\}]\!]$ *assigns zero mass to vectors containing $\bot$), then it implements an unbiased estimator of the real value represented by r. That is, for $\mu = [\![$**estimator**$\{e\}]\!]$, we have $\mathbb{E}_\mu[\pi_i] = \pi_1(\pi_i r) - \pi_2(\pi_i r)$ for all $i \in \{1, \dots, n\}$.*

Proof sketch. The proof for **integrator**$\{\cdot\}$ is unchanged.

For **estimator**$\{\cdot\}$, first, note that Lemma 3 gives $(r, \widehat{r}) \in R^{\sim\dagger}_{\mathfrak{R}^n}$, where $r = [\![e]\!]$ and $\widehat{r} = [\![$**estimator**$\{e\}^\dagger]\!]$. Lemma 5 relates $\widehat{r}$ to $\mu = [\![$**estimator**$\{e\}]\!]$: $(\phi \circ \pi_2)_* \widehat{r} =$ **lift**(μ). Since μ terminates almost surely, **lift**$(\mu) = \mu$ and $\mathbb{P}_{x\sim\mu}[\pi_i x = \bot] = 0$ for all $i \in \{1, \dots, n\}$.

Now consider $(x_L, x_S) \sim \widehat{r}$. Since $\phi(x_S)$ is distributed according to μ, which is almost never $\bot$ in any component, we also have $\pi_i x_S \neq (\bot, \bot)$ almost surely, for all $i \in \{1, \dots, n\}$. This implies that $\phi(\pi_i x_S) = \pi_1(\pi_i x_L) - \pi_2(\pi_i x_L)$ almost surely. Taking expectations, we get $\mathbb{E}_\mu[\pi_i] = \mathbb{E}_{\widehat{r}}[\pi_1(\pi_i x_L) - \pi_2(\pi_i x_L)]$. By linearity of expectation and $R^{\sim\dagger}_{\mathfrak{R}^n}$ condition (1), we have $\mathbb{E}_{\widehat{r}}[\pi_1(\pi_i x_L)] - \mathbb{E}_{\widehat{r}}[\pi_2(\pi_i x_L)] = \pi_1(\pi_i r) - \pi_2(\pi_i r)$. Composing these equalities, we get $\mathbb{E}_\mu[\pi_i] = \pi_1(\pi_i r) - \pi_2(\pi_i r)$, as required. □

This extension allows us to correctly handle recursive definitions involving signed quantities and derive unbiased estimators for them, if the resulting estimator terminates.

3.6 Integration and estimation as first-class constructs via recursive types

We have formulated **integrator** and **estimator** as *program transformations*: tools that take source code as input and produce new source code as output. An alternative is to include them as operations *within* the language:

- *estimate* : $\mathfrak{R}^n \to P\ (\mathbb{R} \times \cdots \times \mathbb{R})$ yields an unbiased estimator for the value represented by its argument
- $integrate^n_\tau : P\ \tau \to (\tau \to \mathfrak{R}^n) \to \mathfrak{R}^n$ yields an integrator for the measure denoted by its argument.

This approach is ultimately more flexible from the user's point of view, but can be more challenging to formalize and implement, requiring a fundamental shift in the semantics of our types, particularly $\mathfrak{R}^n$ and $P\ \tau$.

3.6.1. Mutually recursive types for nested integration and estimation

If *estimate* is part of the language, then $[\![\mathit{estimate}]\!]$ must be a (mathematical) function sending values in $[\![\Re^n]\!]$ to values in $[\![P\ \mathbb{R}^n]\!]$. This implies that values in $[\![\Re^n]\!]$ must somehow *already contain* the information needed to uniquely specify a particular unbiased estimator for *estimate* to return. Similarly, because *integrate* returns values of type $\Re^n$, which can then be estimated, a value in $[\![P\ \tau]\!]$ must uniquely specify a recipe for estimating any integral with respect to the measure it represents. These estimators are themselves of type $P\ \mathbb{R}$, and when *estimate* and *integrate* are first-class language constructs, the user can request integrals with respect to *them*, and then estimates of those integrals, and so on, ad infinitum. This motivates our choice to recast $\Re^n$ and $P\ \tau$ as **mutually recursive types**.

The remainder of this section will develop these new semantics and an updated correctness proof. But our use of mutually recursive types is not just a theoretical solution: it also provides a blueprint for practical implementations. For example, in Haskell, we can define:

```
data ExtendedReal = ExpectedValueOf {estimate :: P Double}
data P a = P {sample :: Prob a, integrate :: (a → ExtendedReal) → ExtendedReal}
```

where Prob *a* is an existing monad supporting random effects (e.g., Haskell's `IO`). We can then define operations on these types, recursively, following the semantic definitions we describe in this section. For example, we can make *P* a monad by implementing *return* and »=, then implement operations such as *cast* and *flip*:

```
instance Monad P where
  return x = P {
    sample = Prob.return x,
    integrate = \g → g x
  }
  (»=) p f = P {
    sample = Prob.do {
      x ← sample p;
      sample (f x)
    }
    integrate = \g →
      integrate p
        (\x → integrate (f x) g)
  }
cast x = ExpectedValueOf (return x)
```

```
data FlipStrat = ENUMERATE | SAMPLE
flip :: (Double, FlipStrat) → P Bool
flip (p, strat) = P {
  sample = Prob.bernoulli p,
  integrate = \g → case strat of
    SAMPLE → ExpectedValueOf P.do {
      b ← flip(p, SAMPLE);
      estimate g(b)
    }
    ENUMERATE → ExpectedValueOf P.do {
      x_T ← estimate g(True);
      x_F ← estimate g(False);
      return (p × x_T + (1 − p) × x_F)
    }
}
```

Notice how *flip* is defined recursively: when we estimate the expected value of a

function g under $\mathit{flip}(p)$ using the `SAMPLE` strategy, the estimator flips a coin using the *flip* operation currently being defined. Notice also that the implementation does not have to explicitly represent the true values of *ExtendedReal*s, as they are intended to be opaque: the only thing the user can do with an *ExtendedReal* is estimate it. All of the complexity of the semantics, including the use of formal differences for signed quantities, for instance, is not present in the implementation: it exists solely to help us reason about the correctness of the implementation.

Formalization. We formalize this setup using recursively defined semantic domains (see Section 2.4). To reflect that this is a new semantic setting, we write $[\![\cdot]\!]_{\int}$ rather than $[\![\cdot]\!]$. Listing 3.13 gives domain equations for the denotation of $\mathfrak{R}^n$ and $P\ \tau$. Solving them via the bilimit construction of Section 2.4 yields pointed ω-qbses $[\![\mathfrak{R}^n]\!]_{\int}$ and $[\![P\ \tau]\!]_{\int}$, together with isomorphisms **roll** and **unroll** for each type.

Given a value $\bar{r}$ of type $\mathfrak{R}^n$, **unroll**$(\bar{r})$ is a triple $(r, \widehat{R}, \widehat{r})$, where:

- $r \in (\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n$ is the actual vector of values, each a formal difference (r_i^+, r_i^-).
- $\widehat{r} \in [\![P\ (\mathbb{R}^n)]\!]_{\int}$ is the user-visible unbiased estimator (i.e., the one returned by *estimate*), which itself lives in the recursive domain for probabilistic programs.
- $\widehat{R} \in \mathrm{Prob}\ ((\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n \times (\mathbb{R}_\perp \times \mathbb{R}_\perp)^n)$ is an auxiliary coupling used by our proofs, playing the role that **estimator**$\{\cdot\}^\dagger$ played in Lemma 3.

Similarly, given a value $\bar{\mu}$ of type $P\ \tau$, **unroll**(μ) is a pair (μ, ν), where:

- $\mu \in \mathrm{Meas}\ [\![\tau]\!]_{\int}$ is the underlying measure on the semantic domain of τ.
- $\nu = (\nu_n)_{n \in \mathbb{N}}$ is a family of integration operators, where $\nu_n : ([\![\tau]\!]_{\int} \Rightarrow [\![\mathfrak{R}^n]\!]_{\int}) \Rightarrow [\![\mathfrak{R}^n]\!]_{\int}$ takes an integrand (mapping semantic values of τ to semantic values of $\mathfrak{R}^n$) and produces the resulting integral (as a semantic value of $\mathfrak{R}^n$).

3.6.2. Correctness via unary logical relations

Since we no longer have separate source programs and translated programs, our logical relations will now be *unary predicates* $R_\tau^{\int\sim} \subseteq [\![\tau]\!]_{\int}$. These predicates carve out the "well-behaved" values of our recursive types: values where all the bundled components (values, estimators, measures, integrators) are mutually consistent. Listing 3.13 defines these predicates recursively. In each case, the relations encode the same logic as our previous logical relations, together with the requirement that any nested values are also well-behaved according to their specifications.

It is not immediately obvious that these recursive definitions are well-formed, i.e., that each predicate's description really picks out a subset of $[\![\tau]\!]_{\int}$. We adopt techniques from Pitts [1996] to compute their fixed points. First, recall that an **admissible relation** is one that contains $\perp_\tau$ and is closed under suprema of chains of elements. In particular, for each type τ, $[\![\tau]\!]_{\int}$ (the trivially true predicate) and $\{\perp_\tau\}$

Listing 3.13 Semantics for first-class integration and estimation.

Type τ	**Semantics** $[\![\tau]\!]_\int$ **and spec** $R_\tau^{\int\sim}$
$\mathfrak{R}^n$	$(\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n \times \text{Prob}\,((\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0})^n \times (\mathbb{R}_\perp \times \mathbb{R}_\perp)^n) \times [\![P\,(\mathbb{R}^n)]\!]_\int$ $(\textbf{roll}(r, \widehat{R}, \widehat{r} = \textbf{roll}(\widehat{r_\mu}, \widehat{r_\int}))) \in R_{\mathfrak{R}^n}^{\int\sim}$ if and only if: 1. $\widehat{r} \in R_{P\,(\mathbb{R}\times\cdots\times\mathbb{R})}^{\int\sim}$ 2. $(\phi \circ \pi_2)_* \widehat{R} = \textbf{lift}(\widehat{r_\mu})$ 3. for each $i \in \{1, \ldots, n\}$: (a) $\pi_i r = \mathbb{E}_{\widehat{R}}[\pi_i \circ \pi_1]$ (b) for $(x_L, x_S) \sim \widehat{R}$, $\pi_i x_S \in \{(\perp, \perp), \pi_i x_L\}$ w.p. 1.
$P\,\tau$	$\text{Meas}\,[\![\tau]\!]_\int \times \prod_{n\in\mathbb{N}} \left(\left([\![\tau]\!]_\int \Rightarrow [\![\mathfrak{R}^n]\!]_\int\right) \Rightarrow [\![\mathfrak{R}^n]\!]_\int\right)$ $\textbf{roll}(\mu, \nu) \in R_{P\,\tau}^{\int\sim}$ if and only if: 1. $\mu(1_{R_\tau^{\int\sim}}) = \lvert\mu\rvert$ 2. $\forall n \in \mathbb{N}, f \in R_{\tau\to\mathfrak{R}^n}^{\int\sim}$: (a) $\pi_n \nu(f) \in R_{\mathfrak{R}^n}^{\int\sim}$ (b) $\mu(\pi_1 \circ \textbf{unroll} \circ f) = \pi_1(\textbf{unroll}((\pi_n \nu)(f)))$
$\sigma \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}\}$	$[\![\sigma]\!]$ $R_\sigma^{\int\sim} = [\![\sigma]\!]$
$\tau_1 \times \tau_2$	$[\![\tau_1]\!]_\int \times [\![\tau_2]\!]_\int$ $R_{\tau_1\times\tau_2}^{\int\sim} = \left\{(x_1, x_2) \mid \forall i \in \{1,2\}, x_i \in R_{\tau_i}^{\int\sim}\right\}$
$\ell_1\tau_n + \cdots + \ell_n\tau_n$	$\ell_1[\![\tau_1]\!]_\int + \cdots + \ell_n[\![\tau_n]\!]_\int$ $R_{\sum_{i=1}^n \ell_i\tau_i}^{\int\sim} = \left\{\ell_i\, x \mid x \in R_{\tau_i}^{\int\sim}, i \in \{1, \ldots, n\}\right\}$
$\tau_1 \to \tau_2$	$[\![\tau_1]\!]_\int \Rightarrow [\![\tau_2]\!]_\int$ $R_{\tau_1\to\tau_2}^{\int\sim} = \left\{f \mid \forall x \in R_{\tau_1}^{\int\sim}, f(x) \in R_{\tau_2}^{\int\sim}\right\}$

(the predicate that *only* holds of $\perp_\tau$) are both admissible, and the set of admissible relations on a type are closed under intersection.

Now consider the complete lattice whose elements are tuples of admissible relations $R = (R_{\mathfrak{R}^n}^+, R_{\mathfrak{R}^n}^-, R_{P\,\mathbb{R}^n}^+, R_{P\,\mathbb{R}^n}^-)$, where $R_\tau^\pm \subseteq [\![\tau]\!]_\int$. A tuple R is less than or equal to a tuple

Listing 3.14 Semantics of first-class estimation and integration constructs.

Construct	Semantics $[\![\cdot]\!]_\int$
$estimate_n$	$x \mapsto \pi_3(\textbf{unroll}(x))$
$integrate_n$	$p \mapsto g \mapsto \pi_n(\pi_2(\textbf{unroll}(p)))(g)$

S if $R^+_\tau \subseteq S^+_\tau$ and $S^-_\tau \subseteq R^+_\tau$ for each type $\tau \in \{P\ \mathbb{R}^n, \mathfrak{R}^n\}$. This lattice has top element $([\![\mathfrak{R}^n]\!]_\int, \{\bot_{\mathfrak{R}^n}\}, [\![P\ \mathbb{R}^n]\!]_\int, \{\bot_{P\ \mathbb{R}^n}\})$ and bottom element $(\{\bot_{\mathfrak{R}^n}\}, [\![\mathfrak{R}^n]\!]_\int, \{\bot_{P\ \mathbb{R}^n}\}, [\![P\ \mathbb{R}^n]\!]_\int)$. Meets in the lattice are given by intersections of the positive components and joins of the negative components, where the join of two relations is the smallest admissible relation of which both are subsets.

Our recursive definitions of $R^{\int\sim}_\tau$ induce a monotone map on this lattice. In particular, we map each R to $S = (S^+_{\mathfrak{R}^n}, S^-_{\mathfrak{R}^n}, S^+_{P\ \mathbb{R}^n}, S^-_{P\ \mathbb{R}^n})$ where $S^\pm_\tau$ is computed by reading the definition of $R^{\int\sim}_\tau$ with negative recursive references to $R^{\int\sim}_\tau$ replaced by $R^\mp_\tau$ and positive recursive references replaced by $R^\pm_\tau$. This monotone map has a least fixed point R^* in our lattice, and Pitts [1996] shows that $R^{*+}_\tau = R^{*-}_\tau$ for each type τ.

This yields interpretations for the logical relations $R^{\int\sim}_{\mathfrak{R}^n}$ and $R^{\int\sim}_{P\ \mathbb{R}^n}$.[2] In our correctness arguments, we will need to show that the denotations of certain programs belong to these relations. If program is recursive, i.e. defined as a fixed point of some Scott-continuous map f, then we can simply verify that whenever x is in the relation, so is $f(x)$. Since the relations are admissible, they contain the least fixed points of all relation-preserving Scott-continuous maps f.

Semantics of expressions. The semantics of our language constructs must now be modified to produce values within these recursive domains. The semantics of our new constructs $estimate_n$ and $integrate_n$ (Listing 3.14) are straightforward: they simply project out the corresponding component already stored within the semantic value of their argument. The real work lies in defining the semantics of the core language constructs (primitives, **return**, **do**) such that they actually build values satisfying $R^{\int\sim}$. Listing 3.15 gives several examples. Note that for primitives, we can generally define their semantics by computing the semantics of their translations under our previously defined program transformations **estimator** and **integrator**. These semantics are sometimes computed using $[\![\cdot]\!]$ (i.e., using our previously defined denotations), and sometimes using $[\![\cdot]\!]_\int$. In the latter case, this can lead our definitions to be recursive. For example, $[\![flip]\!]_\int$ is defined in terms of $[\![\textbf{integrator}\{flip\}]\!]_\int$, which is equal to $[\![\mathbf{E}_{flip}]\!]_\int$, which is defined in terms of $[\![\textbf{estimator}\{\mathbf{E}_{flip}\}]\!]_\int$, which is (for some estimation strategies, such as `SAMPLE`) defined in terms of $[\![flip]\!]_\int$. This is the same recursion we observed in the Haskell definition of *flip* in the previous section.

[2] For $P\ \tau$ at other types τ, note that our definition is not (mutually) recursive and thus is trivially well-defined.

Listing 3.15 Semantics of expressions in the first-class setting.

Expression e	**Denotation** $[\![e]\!]_{\int}(\gamma)$
return e	$\mathbf{roll}(\mu, \nu)$ where: $\mu = \delta_{[\![e]\!]_{\int}(\gamma)}$ $\nu_n(g) = g([\![e]\!]_{\int}(\gamma))$
do $\{e\}$	$[\![e]\!]_{\int}(\gamma)$
do $\{x \leftarrow e; m\}$	$\mathbf{roll}(\mu, \nu)$ where: $\mu = \oint \mu_{2,v}\ \mu_1(dv)$ $\nu_n(g) = \pi_n(\nu_1)(v \mapsto \pi_n(\nu_{2,v})(g))$ $(\mu_1, \nu_1) = \mathbf{unroll}([\![e]\!]_{\int}(\gamma))$ $(\mu_{2,v}, \nu_{2,v}) = \mathbf{unroll}([\![\mathbf{do}\{m\}]\!]_{\int}(\gamma[x \mapsto v]))$
$cast_{\mathbb{R}^n \to \mathfrak{R}^n}$	$x \mapsto \mathbf{roll}(r, \widehat{R}, \widehat{r})$ where: $r = [\![cast_{\mathbb{R}^n \to \mathfrak{R}^n}]\!](x)$ (using formal difference semantics) $\widehat{R} = [\![\mathbf{estimator}\{cast_{\mathbb{R}^n \to \mathfrak{R}^n}\}^{\dagger}]\!](x)$ $\widehat{r} = [\![\mathbf{estimator}\{cast_{\mathbb{R}^n \to \mathfrak{R}^n}\}]\!]_{\int}(x)$
$+_{\mathfrak{R}}$	$(x, y) \mapsto \mathbf{roll}(r, \widehat{R}, \widehat{r})$ where: $r = [\![+_{\mathfrak{R}}]\!](\pi_1(\mathbf{unroll}(x)), \pi_1(\mathbf{unroll}(y)))$ $\widehat{R} = [\![\mathbf{estimator}\{+_{\mathfrak{R}}\}^{\dagger}]\!](\pi_2(\mathbf{unroll}(x)), \pi_2(\mathbf{unroll}(y)))$ $\widehat{r} = [\![\mathbf{estimator}\{+_{\mathfrak{R}}\}]\!]_{\int}(\pi_3(\mathbf{unroll}(x)), \pi_3(\mathbf{unroll}(y)))$
$flip$	$p \mapsto \mathbf{roll}(\mu, \nu)$ where: $\mu = [\![flip]\!](p)$ $\nu_n(g) = [\![\mathbf{integrator}\{flip\}^n]\!]_{\int}(p)(g)$
$\mathbf{E}^n_{flip}$	$p \mapsto g \mapsto \mathbf{roll}(r, \widehat{R}, \widehat{r})$, where: $r = [\![\mathbf{E}^n_{flip}]\!](p, \pi_1 \circ \mathbf{unroll} \circ g)$ $\widehat{R} = [\![\mathbf{estimator}\{\mathbf{E}^n_{flip}\}^{\dagger}]\!](p, \pi_2 \circ \mathbf{unroll} \circ g)$ $\widehat{r} = [\![\mathbf{estimator}\{\mathbf{E}^n_{flip}\}]\!]_{\int}(p, \pi_3 \circ \mathbf{unroll} \circ g)$

Correctness. The main correctness theorem in this setting states that every well-typed expression denotes a coherent value.

Theorem 5 (Correctness with first-class integration and estimation)**.** *If* $\Gamma \vdash e : \tau$ *is well-typed, then for any environment* $\gamma \in R_{\Gamma}^{\int\sim}$*, we have* $[\![e]\!]_{\int}(\gamma) \in R_{\tau}^{\int\sim}$*.*

Proof sketch. The proof proceeds by induction on the typing derivation. Because our semantics are defined in terms of **integrator** and **estimator**, and our logical

relations are based on those in previous sections, the arguments from the proofs of our previous correctness theorems can be adapted with minor changes to handle the corresponding cases in this proof. □

This theorem ensures that although the semantics are complex and recursively defined, they remain consistent: the estimators produced by $estimate_n$ are indeed unbiased for the value component, and the integrals computed by $integrate_n$ match the underlying measure semantics.

4

RADON-NIKODYM DERIVATIVES

The work in this chapter was originally published in Lew et al. [2023a]. It is joint work with Matin Ghavami, Martin Rinard, and Vikash Mansinghka.

4.1 Motivation and overview

In the previous chapter, we developed a framework for automatically constructing unbiased estimators for integrals with respect to measures defined by probabilistic programs. This capability is powerful, as many quantities of interest in probabilistic modeling, such as expected values, probabilities of events, and even variances, can be expressed as integrals. However, many advanced Monte Carlo algorithms and statistical techniques rely not just on the ability to compute or estimate an integral, but on the ability to compute or estimate *densities* or *density ratios*. Specifically, the **Radon-Nikodym derivative**, which represents the density of one measure with respect to another, is a fundamental quantity that arises again and again in practical algorithms for Monte Carlo and variational inference.

Consider, for example, importance sampling. To reweight samples from a proposal distribution Q to target a distribution P, we need to compute the importance weights $w(x) = p(x)/q(x)$, where p and q are densities of P and Q respectively (with respect to some common base measure). Similarly, the Metropolis-Hastings algorithm computes an acceptance ratio that involves a ratio of target densities and proposal densities. Even beyond these core Monte Carlo methods, many techniques in variational inference, Bayesian model comparison, and causal inference involve computing or reasoning about ratios of probability densities.

While our framework from Chapter 3 allows us to obtain unbiased estimators for individual (intractable) densities (by viewing them as integrals), it does not directly provide a way to obtain unbiased estimators for *ratios* of these densities. The ratio of two unbiased estimators is, in general, a biased estimator of the ratio. This presents

a significant challenge: how can we provide automation to support the wide range of algorithms that depend on Radon-Nikodym derivatives?

This chapter introduces the **stochastic probability interface (SPI)**, a more general computational interface designed to address this challenge. The SPI goes beyond simply providing an unbiased estimator for a single density. Instead, for a given measure μ and a reference measure ν, it provides a pair of operations:

1. A **density estimator** ξ: a kernel that, for any point x, yields an unbiased estimator for the Radon-Nikodym derivative $\frac{d\mu}{d\nu}(x)$.
2. A **weighted sampler** χ: a measure on pairs (x, w) such that x is distributed according to μ, and w is a weight. Critically, this sampler has the property that $1/w$ can be used as an unbiased estimator for the *reciprocal* of the density, $\frac{d\nu}{d\mu}(x)$, when x is sampled via χ.

As we will show, this pair of operations is sufficient to soundly implement a variety of inference algorithms that require density ratios, by combining estimates from ξ (for numerators) and χ (for denominators).

The chapter proceeds as follows. We first formally define the SPI and explore its key properties (Section 4.2). We then demonstrate how standard inference algorithms like importance sampling, sequential Monte Carlo, and Metropolis-Hastings can be implemented soundly using the SPI (Theorem 6). Following this, we introduce programming language constructs for *absolutely continuous probabilistic programming* (Section 4.3), which allow users to define probabilistic programs whose densities are guaranteed to exist with respect to predefined *stock measures*. Finally, we present a program transformation that automatically compiles these absolutely continuous probabilistic programs into implementations of the SPI (Section 4.4), enabling the automated construction of the necessary density estimators and weighted samplers. This compilation process also involves generating automated *unit tests* to help ensure the validity of user-provided components, such as proposal distributions for importance-sampling-based marginalization strategies. Together, these contributions provide a robust and expressive framework for probabilistic programming with automated support for Radon-Nikodym derivatives.

4.2 The stochastic probability interface

In this chapter, we will be concerned with automatically constructing *density functions* for the measures denoted by probabilistic programs. For complex programs, these densities are typically intractable, requiring a large sum or integral over all possible program paths consistent with a given observation. For this reason, we could represent densities using our type $\mathfrak{R}_{\geq 0}$ of possibly intractable integral results from Chapter 3. The machinery developed in that chapter could then be used to automatically build *unbiased estimators* for these densities, using the first-class construct $\mathit{estimate} : \mathfrak{R}_{\geq 0} \to P\, \mathbb{R}_{\geq 0}$.

Just as unbiased estimates of gradients can be used within many optimization algorithms (e.g., stochastic gradient descent), many Monte Carlo algorithms can be soundly accelerated by using unbiased estimates of densities when exact densities are too slow [Andrieu and Roberts, 2009, Fearnhead et al., 2010, Lew et al., 2022b]. However, this is only half of the story. Most Monte Carlo algorithms require not just a density, but a *density ratio*, or **Radon-Nikodym derivative**. For example, the **importance weights** we must compute in importance sampling are ratios of densities, with one distribution in the numerator and another in the denominator. As another example, the **Metropolis-Hastings acceptance ratio** is also a ratio of densities, with a forward transition distribution in the denominator and a backward transition distribution in the numerator.

Our language supports estimating some functions of $\Re_{\geq 0}$ values, but not division: there is no general way to transform unbiased estimators of r_1 and r_2 into an unbiased estimator of r_1/r_2. Therefore, to support the full range of use cases for density functions in Monte Carlo and variational inference, we need to go beyond automating estimates of densities. The **stochastic probability interface** (SPI) is our proposal for a more general computational interface to distributions that admit densities.

4.2.1. Definitions

Let μ and ν be measures on a space X.

Definition 15 (density / Radon-Nikodym derivative)**.** We say $\frac{d\mu}{d\nu} : X \to \overline{\mathbb{R}}_{\geq 0}$ is a **density** (or **Radon-Nikodym derivative**) of μ with respect to ν if for all $g : X \to \overline{\mathbb{R}}_{\geq 0}$,

$$\int g(x) \cdot \frac{d\mu}{d\nu}(x)\, \nu(dx) = \int g(x)\, \mu(dx).$$

Definition 16 (scaling of a measure by a density)**.** For $f : X \to \overline{\mathbb{R}}_{\geq 0}$, we define the **scaled measure** $f \odot \nu$ by

$$\int g\, d(f \odot \nu) \coloneqq \int f(x) g(x) d\nu(x).$$

That is, $f \odot \nu$ is the unique measure for which f is a density with respect to ν.

Definition 17 (absolute continuity)**.** We say μ is **absolutely continuous** with respect to ν, written $\mu \ll \nu$, if there exists a density $\frac{d\mu}{d\nu}$ of μ with respect to ν.

Definition 18 (proper weighting)**.** Let χ be a measure on $X \times \mathbb{R}_{\geq 0}$, and suppose $(x, w) \sim \chi$. We say the pair (x, w) is **properly weighted for** ν if, for all $f : X \to \overline{\mathbb{R}}_{\geq 0}$,

$$\int w \cdot f(x)\, \chi(dx, dw) = \int f(x)\, \nu(dx).$$

The measure χ is said to be a **properly weighted sampler** for ν.

Definition 19 (stochastic probability interface)**.** Let $\xi : X \to \text{Meas}\, \mathbb{R}_{\geq 0}$ be a kernel, and let $\chi : \text{Meas}\, (X \times \mathbb{R}_{\geq 0})$ be a measure. We say (ξ, χ) implement the **stochastic probability interface** (SPI) for μ with respect to ν if:

1. (**Unbiasedness of** ξ) The map $x \mapsto \int w\, \xi(x, dw)$ is a density of μ with respect to ν.
2. (**Correspondence of** χ **to** ξ) The measure χ is equal to $((x, w) \mapsto w) \odot (\nu \otimes \xi)$.

We call ξ the **density estimator** and χ the **weighted sampler**.

Proposition 1. *Suppose* (ξ, χ) *implement the SPI for* μ *with respect to* ν*. Then:*

1. ***Sampler admits*** μ ***as its marginal:*** $\pi_{1*}\chi = \mu$
2. $(x, \frac{1}{w})$ ***is properly weighted for*** $\widehat{\nu}$***:*** *For all* $f : X \to \overline{\mathbb{R}}_{\geq 0}$*,*

$$\int \mathbf{1}[w > 0] \cdot \frac{1}{w} \cdot f(x)\, \chi(dx, dw) = \int f(x) \widehat{\nu}(dx),$$

where $\widehat{\nu} = \left(x \mapsto \int\, \mathbf{1}[w > 0]\, \xi(x, dw)\right) \odot \nu$.

Proof. Let $f : X \to \overline{\mathbb{R}}_{\geq 0}$.

1. We have:

$$\begin{aligned}\int f(x)\, \pi_{1*}\chi(dx) &= \int f(x) \left(\int w\, \xi(x, dw) \right) \nu(dx) \\ &= \int f(x) \frac{d\mu}{d\nu}(x) \nu(dx) = \int f(x) \mu(dx).\end{aligned}$$

   Since this holds for all f, we have $\pi_{1*}\chi = \mu$.

2. We have:

$$\begin{aligned}\int \mathbf{1}[w > 0] \cdot \frac{1}{w} \cdot f(x)\, \chi(dx, dw) &= \int \mathbf{1}[w > 0] \cdot \frac{1}{w} \cdot f(x) \cdot w\, (\nu \otimes \xi)(dx, dw) \\ &= \int \mathbf{1}[w > 0] \cdot f(x)\, (\nu \otimes \xi)(dx, dw) \\ &= \int f(x) \left(\int \mathbf{1}[w > 0]\, \xi(x, dw) \right) \nu(dx) \\ &= \int f(x) \widehat{\nu}(dx).\end{aligned}$$

□

Estimating general Radon-Nikodym derivatives. When ξ is a probability kernel, it implements an unbiased density estimator for μ with respect to ν. When ξ is an

almost surely positive probability kernel for ν-almost-all inputs x, and $(x, w) \sim \chi$, $(x, \frac{1}{w})$ is properly weighted for ν. This latter condition implies that for $(x, w) \sim \chi$,

$$\mathbb{E}\left[\frac{1}{w} \,\middle|\, x\right] = \frac{d\nu}{d\mu}(x),$$

i.e., $1/w$ is an unbiased estimate for the *reciprocal* of the density function. Now suppose we want to estimate $\frac{d\mu_1}{d\mu_2}$ for two measures μ_1 and μ_2, and we have SPI implementations (ξ_1, χ_1) and (ξ_2, χ_2) for each measure, with respect to the same reference measure ν. Assuming both ξ_1 and ξ_2 satisfy the conditions just given,

$$\mathbb{E}_{(x,w_2)\sim\chi_2, w_1\sim\xi_1(x)}\left[\frac{w_1}{w_2} \,\middle|\, x\right] = \frac{d\nu}{d\mu_2}(x) \cdot \frac{d\mu_1}{d\nu}(x) = \frac{d\mu_1}{d\mu_2}(x).$$

Thus, we can use the SPI to unbiasedly estimate ratios of densities. However, note that the formula above is not a recipe for estimating $\frac{d\mu_1}{d\mu_2}(x)$ at arbitrary points x, but rather for x *sampled from* μ_2 (because $\pi_{1*}\chi_2 = \mu_2$). Luckily, this turns out to cover most cases of interest in Monte Carlo inference, as we will see in the next section.

Examples of SPI implementations. For concreteness, we give a couple examples of SPI implementations.

Example 36 (fair coin)**.** Let $\mu = \mathbf{bern}(0.5)$ model a fair coin with outcomes $\{\mathtt{t}, \mathtt{f}\}$. Then $\xi(x, dw) = \frac{1}{2} \cdot \delta_{0.25} + \frac{1}{2} \cdot \delta_{0.75}$ is an unbiased density estimator. Its corresponding weighted sampler is $\chi(dx, dw) = \frac{1}{8} \cdot \delta_{(\mathtt{t},0.25)} + \frac{3}{8} \cdot \delta_{(\mathtt{t},0.75)} + \frac{1}{8} \cdot \delta_{(\mathtt{f},0.25)} + \frac{3}{8} \cdot \delta_{(\mathtt{f},0.75)}$.

Example 37 (unbiased densities from importance sampling)**.** Let X and Z be spaces, and suppose μ is the first marginal of a joint distribution μ_{joint} over $X \times Z$. We can then estimate the density of μ (with respect to a reference measure ν on X) using importance sampling. Let $Q(x, dz)$ be a proposal distribution for importance sampling, with $\mu_{\text{joint}} \ll \mu \otimes Q$. Then the importance sampling estimator

$$\xi(x) = \int_Z \delta_{\frac{d\mu_{\text{joint}}}{d(\nu\otimes Q)}(x,z)} Q(x, dz)$$

is an unbiased density estimator for μ with respect to ν. Its corresponding weighted sampler is $\chi(dx, dw) = \int_{X\times Z} \delta_{\left(x, \frac{d\mu_{\text{joint}}}{d(\nu\otimes Q)}(x,z)\right)} \mu_{\text{joint}}(dx, dz)$.

4.2.2. Inference against the stochastic probability interface

Using the stochastic probability interface, it is possible to soundly implement many standard algorithms for probabilistic inference. Where these algorithms would usually compute densities, we now use ξ and χ to compute appropriate estimates.

Assumptions. In what follows, *target measures* P are assumed to be equipped with implementations (ξ_P, χ_P) of the SPI, with ξ_P a probability kernel. *Proposal*

Algorithm 1 Importance Sampling

Require: Target $P(dx)$
Require: Proposal $Q(dx)$ such that $P \ll Q$
Ensure: $x \sim Q(dx)$
Ensure: (x, w) properly weighted for P

1: $(x, w_Q) \sim \chi_Q(dx, dw_Q)$
2: $w_P \sim \xi_P(x, dw_P)$
3: $w \leftarrow \frac{w_P}{w_Q}$
4: **return** (x, w)

Algorithm 2 Metropolis-Hastings

Require: Target $P(dx)$
Require: Transition proposal $Q(x, dx')$ such that $swap_*(P \otimes Q) \ll P \otimes Q$
Require: Current state $(x, \widehat{p})$
Ensure: If $(x, \widehat{p}) \sim \chi_P$ then $(x_{new}, \widehat{p}_{new}) \sim \chi_P$

1: $(x', w_{forward}) \sim \chi_Q(x, dx', dw_{forward})$
2: $w_{back} \sim \xi_Q(x', x, dw_{back})$
3: $\widehat{p'} \sim \xi_P(x', d\widehat{p'})$
4: $\alpha \leftarrow \frac{\widehat{p'} \cdot w_{back}}{\widehat{p} \cdot w_{forward}}$
5: $u \sim$ **uniform**$(0, 1)$
6: **if** $u \leq \alpha$ **then**
7: $\quad (x_{new}, \widehat{p}_{new}) \leftarrow (x', \widehat{p'})$
8: **else**
9: $\quad (x_{new}, \widehat{p}_{new}) \leftarrow (x, \widehat{p})$
10: **return** $(x_{new}, \widehat{p}_{new})$

Algorithm 3 Simple SMC Step

Require: $(x_{1:t-1}, w_{t-1})$ properly weighted for $P_1(dx_1) \prod_{i=2}^{t-1} P_i(x_{i-1}, dx_i)$
Require: Conditional target $P_t(x_{1:t-1}, dx_t)$
Require: Proposal $K(x_{1:t-1}, dx_t)$ such that $P_t(x_{1:t-1}) \ll K(x_{1:t-1})$
Ensure: $x_t \sim K(x_{t-1}, dx_t)$
Ensure: $(x_{1:t}, w_t)$ properly weighted for $P_1(dx_1) \prod_{i=2}^{t} P_i(x_{i-1}, dx_i)$

1: $(x_t, w_K) \sim \chi_K(x_{1:t-1}, dx_t, dw_K)$
2: $w_P \sim \xi_{P_t}(x_t, dw_P)$
3: $w_t \leftarrow w_{t-1} \cdot \frac{w_P}{w_K}$
4: **return** $(x_{1:t}, w_t)$

Algorithm 4 General SMC Step

Require: Targets $P_{t-1}(dx_{t-1})$ and $P_t(dx_t)$
Require: Proposals $K(x_{t-1}, dx_t)$, $L(x_t, dx_{t-1})$, such that $swap_*(P_t \otimes L) \ll P_{t-1} \otimes K$
Require: $((x_{t-1}, v_{t-1}), w_{t-1})$ properly weighted for $\chi_{P_{t-1}}$
Ensure: $x_t \sim K(x_{t-1}, dx_t)$
Ensure: $((x_t, v_t), w_t)$ properly weighted for χ_{P_t}

1: $(x_t, w_K) \sim \chi_K(x_{t-1}, dx_t, dw_K)$
2: $v_t \sim \xi_{P_t}(x_t, dv_t)$
3: $w_L \sim \xi_L(x_t, x_{t-1}, dw_L)$
4: $w_t \leftarrow w_{t-1} \cdot \frac{v_t \cdot w_L}{v_{t-1} \cdot w_K}$
5: **return** $((x_t, v_t), w_t)$

Figure 4-1: Algorithms for inference against the stochastic probability interface

distributions Q (and also K and L) are also assumed to have implementations (ξ_Q, χ_Q) of the SPI, with ξ_Q a probability kernel. We also assume that for Q-almost all x, $\xi_Q(x)$ is almost surely positive. All SPI implementations for measures on a given space X are assumed to be with respect to a common reference measure ν on X.

Algorithms. Figure 4-1 gives implementations of importance sampling, sequential Monte Carlo, and Metropolis-Hastings that use the SPI to approximate the necessary Radon-Nikodym derivatives. Despite their use of stochastic approximations, these algorithms enjoy the same soundness guarantees as their exact-density counterparts. This is made precise by the following theorem:

Theorem 6. *Under the assumptions stated at the start of this section, the inference algorithms in Figure 4-1 are sound, in the following senses:*

1. *Importance Sampling: If $P \ll Q$, then Alg. 1 returns (x, w) properly weighted for P.*

2. *Metropolis-Hastings: If $((x, \widehat{p}), w)$ is properly weighted for χ_P, then $((x_{new}, \widehat{p}_{new}), w)$ is*

properly weighted for χ_P, for $(x_{new}, \widehat{p}_{new})$ returned by Alg. 2.

3. *Sequential Monte Carlo (simple): If $P_1(dx_1)\prod_{i=2}^{t} P_i(x_{i-1}, dx_i)$ is absolutely continuous with respect to $P_1(dx_1)(\prod_{i=2}^{t-1} P_i(x_{i-1}, dx_i))K(x_{1:t-1}, dx_t)$, then Alg. 3 returns $(x_{1:t}, w_t)$ properly weighted for $P_1(dx_1)\prod_{i=2}^{T} P_i(x_{i-1}, dx_i)$.*

4. *Sequential Monte Carlo (general): If $P_t(dx_t)L(x_t, dx_{t-1})$ is absolutely continuous with respect to $P_{t-1}(dx_{t-1})K(x_{t-1}, dx_t)$, and $((x_{t-1}, v_{t-1}), w)$ is properly weighted for $\chi_{P_{t-1}}$, then Alg. 4 returns $((x_t, v_t), w_t)$ properly weighted for χ_{P_t}.*

Proof. We prove each item in turn.

1. **Importance Sampling (Alg. 1):** We need to show (x, w) is properly weighted for P, i.e., $\mathbb{E}[wf(x)] = \int f(x)P(dx)$. We have:

$$\begin{aligned}
\mathbb{E}[wf(x)] &= \mathbb{E}_{(x,w_Q)\sim\chi_Q}\left[\mathbb{E}_{w_P\sim\xi_P(x)}\left[\frac{w_P}{w_Q}f(x) \;\middle|\; x, w_Q\right]\right] \\
&= \mathbb{E}_{(x,w_Q)\sim\chi_Q}\left[\frac{f(x)}{w_Q}\mathbb{E}_{w_P\sim\xi_P(x)}[w_P]\right] \\
&= \mathbb{E}_{(x,w_Q)\sim\chi_Q}\left[\frac{f(x)}{w_Q}\frac{dP}{d\nu}(x)\right] && \text{Def. 19 (1)} \\
&= \int f(x)\frac{dP}{d\nu}(x)\nu(dx) && \text{Prop. 1 (2), a.s. positivity of } \xi_Q \\
&= \int f(x)P(dx)
\end{aligned}$$

2. **Metropolis-Hastings (Alg. 2):** Let the state be $s = (x, v_P)$, where v_P is the sampled density estimate. We want to show that the transition kernel $K(s, ds')$ defined by Alg. 2 leaves the target measure $\chi_P(dx, dv_P)$ invariant. We use the extended state space method. Consider the extended state $Z = (x, v_P, x', v'_P, w_f, w_b)$ and the joint measure defined by the sampling process:

$$\Pi(dZ) = \chi_P(dx, dv_P)\chi_Q(x, dx', dw_f)\xi_Q(x', x, dw_b)\xi_P(x', dv'_P)$$

The algorithm can be seen as first resampling (x', v'_P, w_f, w_b) from their exact conditional distributions given (x, v_P), and then proposing a deterministic swap move $T : Z \mapsto Z_{swap} = (x', v'_P, x, v_P, w_b, w_f)$ and accepting with probability $\min(1, R)$, where $R = \frac{d\Pi}{dT_*\Pi}(Z_{swap})$. This leaves Π invariant. We compute the ratio R:

$$\frac{d\Pi}{dT_*\Pi}(Z_{swap}) = \frac{d\chi_P(dx', dv'_P)\chi_Q(x', dx, dw_b)\xi_Q(x, x', dw_f)\xi_P(x, dv_P)}{d\chi_P(dx, dv_P)\chi_Q(x, dx', dw_f)\xi_Q(x', x, dw_b)\xi_P(x', dv'_P)}(Z)$$

Substitute the definitions $\chi_P(dx, dv_P) = v_P\nu(dx)\xi_P(x, dv_P)$ and $\chi_Q(x, dx', dw_f) =$

$w_f \nu(dx')\xi_Q(x, x', dw_f)$:

$$
\begin{aligned}
R &= \frac{d[v'_P \nu(dx')\xi_P(x', dv'_P)][w_b \nu(dx)\xi_Q(x', x, dw_b)]\xi_Q(x, x', dw_f)\xi_P(x, dv_P)}{d[v_P \nu(dx)\xi_P(x, dv_P)][w_f \nu(dx')\xi_Q(x, x', dw_f)]\xi_Q(x', x, dw_b)\xi_P(x', dv'_P)}(Z) \\
&= \frac{d(v'_P \cancel{\nu(dx')}\cancel{\xi_P(x', dv'_P)} w_b \cancel{\nu(dx)}\cancel{\xi_Q(x', x, dw_b)}\cancel{\xi_Q(x, x', dw_f)}\cancel{\xi_P(x, dv_P)})}{d(v_P \cancel{\nu(dx)}\cancel{\xi_P(x, dv_P)} w_f \cancel{\nu(dx')}\cancel{\xi_Q(x, x', dw_f)}\cancel{\xi_Q(x', x, dw_b)}\cancel{\xi_P(x', dv'_P)})}(Z) \\
&= \frac{v'_P w_b}{v_P w_f}
\end{aligned}
$$

This ratio is exactly the α computed in Alg. 2 (with v_P, v'_P corresponding to $\widehat{p}, \widehat{p'}$). The algorithm returns just $(x_{new}, \widehat{p}_{new})$. This corresponds to the marginal transition kernel of the MH algorithm on the extended space Π with the swap proposal T. Since it leaves Π invariant, its marginal kernel leaves the marginal χ_P invariant.

3. **Sequential Monte Carlo (Simple, Alg. 3):** Assume $(x_{1:t-1}, w_{t-1})$ is properly weighted for $P_{1:t-1}$. We need $(x_{1:t}, w_t)$ to be properly weighted for $P_{1:t} = P_{1:t-1} \otimes P_t$.

$$
\begin{aligned}
\mathbb{E}[w_t f(x_{1:t})] &= \mathbb{E}_{(x_{1:t-1}, w_{t-1})}\left[w_{t-1}\mathbb{E}_{(x_t, w_K)\sim\chi_K(x_{1:t-1})}\left[\frac{1}{w_K}\mathbb{E}_{w_P\sim\xi_{P_t}(x_t)}[w_P]f(x_{1:t})\right]\right] \\
&= \mathbb{E}_{(x_{1:t-1}, w_{t-1})}\left[w_{t-1}\mathbb{E}_{(x_t, w_K)\sim\chi_K(x_{1:t-1})}\left[\frac{1}{w_K}\frac{dP_t}{dv_t}f(x_{1:t})\right]\right] \\
&= \mathbb{E}_{(x_{1:t-1}, w_{t-1})}\left[w_{t-1}\int \frac{dP_t}{dv_t}f(x_{1:t})\nu_t(dx_t)\right] \\
&= \mathbb{E}_{(x_{1:t-1}, w_{t-1})}\left[w_{t-1}\int f(x_{1:t})P_t(x_{t-1}, dx_t)\right] \\
&= \int\left(\int f(x_{1:t})P_t(x_{t-1}, dx_t)\right)P_{1:t-1}(dx_{1:t-1}) = \int f(x_{1:t})P_{1:t}(dx_{1:t}).
\end{aligned}
$$

4. **Sequential Monte Carlo (General, Alg. 4):** Assume $((x_{t-1}, v_{t-1}), w_{t-1})$ is properly

weighted for $\chi_{P_{t-1}}$. We need to show that $((x_t, v_t), w_t)$ is properly weighted for χ_{P_t}.

$$
\begin{aligned}
\mathbb{E}[w_t f(x_t, v_t)] &= \mathbb{E}_{((x_{t-1}, v_{t-1}), w_{t-1})}\left[w_{t-1}\mathbb{E}_{(x_t, w_K)\sim\chi_K(x_{t-1}), v_t\sim\xi_{P_t}(x_t), w_L\sim\xi_L(x_t, x_{t-1})}\left[\frac{v_t w_L}{v_{t-1} w_K} f(x_t, v_t)\right]\right] \\
&= \mathbb{E}_{(x_{t-1}, v_{t-1})\sim\chi_{P_t}}\left[\mathbb{E}_{(x_t, w_K)\sim\chi_K(x_{t-1}), v_t\sim\xi_{P_t}(x_t), w_L\sim\xi_L(x_t, x_{t-1})}\left[\frac{v_t w_L}{v_{t-1} w_K} f(x_t, v_t)\right]\right] \\
&= \int\int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t), w_L\sim\xi_L(x_t, x_{t-1})}\left[v_t w_L f(x_t, v_t)\right] \nu_t(dx_t)\nu_{t-1}(dx_{t-1}) \\
&= \int\int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t)}\left[v_t f(x_t, v_t)\right] \mathbb{E}_{w_L\sim\xi_L(x_t, x_{t-1})}[w_L]\nu_t(dx_t)\nu_{t-1}(dx_{t-1}) \\
&= \int\int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t)}\left[v_t f(x_t, v_t)\right] \frac{dL(x_t)}{d\nu_{t-1}}(x_{t-1})\nu_t(dx_t)\nu_{t-1}(dx_{t-1}) \\
&= \int\int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t)}\left[v_t f(x_t, v_t)\right] \frac{dL(x_t)}{d\nu_{t-1}}(x_{t-1})\nu_{t-1}(dx_{t-1})\nu_t(dx_t) \\
&= \int\int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t)}\left[v_t f(x_t, v_t)\right] L(x_t, dx_{t-1})\nu_t(dx_t) \\
&= \int \mathbb{E}_{v_t\sim\xi_{P_t}(x_t)}\left[v_t f(x_t, v_t)\right] \nu_t(dx_t) \\
&= \mathbb{E}_{(x_t, v_t)\sim\chi_{P_t}}\left[f(x_t, v_t)\right].
\end{aligned}
$$

□

4.3 Absolutely continuous probabilistic programming

Before we can compile densities (and more generally, SPI implementations) for probabilistic programs, we must decide on a *reference measure* ν with respect to which the densities will be computed. The approach we take in this chapter is to choose a **stock measure** ν_σ for each first-order type σ in our language. Our choices of stock measures are given in Listing 4.1: we choose the Lebesgue measure for real-valued types, the counting measure for discrete types, and then compute sum and product measures for compound types. These choices (and the use of stock measures in general) are standard in the probabilistic programming literature [Cusumano-Towner et al., 2019b, Ge et al., 2018, Bingham et al., 2019b, Lew et al., 2020a].[1]

Using the general machinery for probabilistic programming from the previous sections, it is very easy to write probabilistic programs that fail to be absolutely continuous with respect to our stock measures. In this section, we introduce new programming constructs for *absolutely continuous probabilistic programming*: building distributions that are absolutely continuous *by construction* with respect to the stock measure. Many of our new constructs (Listing 4.2) are analogous to constructs in existing PPLs that need to compute densities, such as Gen, Pyro, and

[1] However, see Narayanan and Shan [2020] for a discussion of how to automatically infer program-specific reference measures, rather than choosing one-size-fits-all stock measures.

Listing 4.1 Stock measures for first-order types.

Type σ	**Stock Measure** ν_σ
1	Dirac measure $\delta_()$
$\mathbb{N}$	Counting measure $\mathbf{counting}_{\mathbb{N}} = \sum_{n\in\mathbb{N}} \delta_n$
$\mathbb{R}$	Lebesgue measure $\mathbf{lebesgue}_{\mathbb{R}} = \Lambda$
$\mathbb{R}_{[0,1]}$	Lebesgue measure $\mathbf{lebesgue}_{[0,1]}$ restricted to $[0,1]$
$\mathbb{R}_{>0}$	Lebesgue measure $\mathbf{lebesgue}_{>0}$ restricted to $(0,\infty)$
$\mathbb{R}_{\geq 0}$	Lebesgue measure $\mathbf{lebesgue}_{\geq 0}$ restricted to $[0,\infty)$
$\sigma_1 \times \sigma_2$	Product measure $\nu_{\sigma_1} \otimes \nu_{\sigma_2}$
$\ell_1\sigma_1 + \cdots + \ell_n\sigma_n$	Sum of measures $\sum_{i=1}^{n}(\ell_{i*}\nu_{\sigma_i})$
Trace	$\mathrm{lfp}\left(\nu \mapsto f \mapsto \sum_{s\in\mathbb{S}}\left(\otimes_{(k,\sigma)\in s}\nu_\sigma[\nu]\right)(f \circ \mathbf{roll}_{\mathbb{T}} \circ \iota_s)\right)$, where $\nu_\sigma[\nu] = \nu_\sigma$ if $\sigma \neq$ Trace and ν if $\sigma =$ Trace

Turing [Cusumano-Towner et al., 2019b, Ge et al., 2018, Bingham et al., 2019b].

4.3.1. Absolutely continuous distributions

Our first extension to the core calculus is a new type $D\ \sigma$ of **absolutely continuous distributions** $\mu \ll \nu_\sigma$. Denotationally, $[\![D\ \sigma]\!]$ is the subspace $\mathrm{Meas}_{\ll\nu_\sigma} \subseteq \mathrm{Meas}\ [\![\sigma]\!]$ consisting of those measures that are absolutely continuous with respect to ν_σ.

Since all of our probabilistic primitives f_{prb} yield measures that satisfy the absolute continuity property, we can add versions $f_{prb_\div} : \mathsf{ArgType}(f_{prb}) \to D\ \sigma$ that yield values of type $D\ \sigma$ instead of $\mathsf{RetType}(f_{prb}) = P\ \sigma$. We can then compose larger distributions using the following constructs:

- **Product distributions:** Given $e_1 : D\ \sigma_1$ and $e_2 : \sigma_1 \to D\ \sigma_2$, we can form the product distribution $e_1 \otimes e_2 : D\ (\sigma_1 \times \sigma_2)$. Note that e_2 denotes a kernel that may depend on the first argument, so this is really a dependent product measure (Definition 8). The product $[\![e_1 \otimes e_2]\!]$ is absolutely continuous with respect to the product of the stock measures $\nu_{\sigma_1} \otimes \nu_{\sigma_2}$.
- **Marginal distributions:** Given $e : D\ (\sigma_1 \times \sigma_2)$, we can form the *marginal distribution* $marginal(e) : D\ \sigma_2$, which is the pushforward of $[\![e]\!]$ by the projection π_2 onto the second component.[2] This is again guaranteed to be absolutely continuous.
- **Pushforwards by bijections:** For any PAP bijection $f : [\![\sigma_1]\!] \to [\![\sigma_2]\!]$, we can add a primitive $f_* : D\ \sigma_1 \to D\ \sigma_2$ that forms the pushforward of its argument by f. This includes, for example, $swap_* : D\ (\sigma_1 \times \sigma_2) \to D\ (\sigma_2 \times \sigma_1)$ (which allows us to use *marginal* to take the first marginal, and not just the second, of a product distribution).
- **More general pushforwards:** By composing pushforwards by bijections with marginalization, we can also push distributions forward through more general

[2] Technically, we have a different primitive $marginal_{\sigma_1,\sigma_2}$ for each pair of first-order types, but we omit the subscripts when they are clear from context.

Listing 4.2 Grammar of $\lambda_{\ll}$, an extension of $\lambda_{\Re}$ from Listing 3.1 to support absolutely continuous probabilistic programming. New productions are highlighted.

Syntactic category		Productions
First-order types σ	::=	$1 \mid \mathbb{N} \mid \mathbb{R} \mid \mathbb{R}_{[0,1]} \mid \mathbb{R}_{>0} \mid \mathbb{R}_{\geq 0} \mid \sigma_1 \times \sigma_2 \mid \ell_1\sigma_1 + \cdots + \ell_n\sigma_n \mid$ Str $\mid$ Trace
Types τ	::=	$\sigma \mid \tau_1 \times \tau_2 \mid \ell_1\tau_1 + \cdots + \ell_n\tau_n \mid \tau_1 \to \tau_2 \mid P\ \tau \mid \Re^n_{\geq 0} \mid D\ \sigma \mid G\ \tau$
Expressions e	::=	$x \mid c \mid f \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e \mid \lambda x.e \mid e_1\ e_2 \mid \ell_i\ e \mid$ **match** e **with** $\{\ell_i x_i \mapsto e_i\}_{i=1}^n \mid e_1 \Cat e_2 \mid \{e_{key} \mapsto e_{val}\} \mid$ **return** $\mid$ **do** $\{m\} \mid$ **return**$_G\ e \mid$ **do**$_G\ \{m\} \mid$ **sample**$_G\ e_{dist}\ e_{name}$
Blocks m	::=	$e \mid x \leftarrow e; m$
Constants c	::=	$() \mid n \mid r \mid$ "s" $\mid \{\}$
Built-ins f_{det}	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \ldots$
f_{prb}	::=	$uniform \mid flip \mid normal \mid \ldots$
f_{ext}	::=	$cast_{\mathbb{R}_{\geq 0} \to \Re_{\geq 0}} \mid +_{\Re} \mid \times_{\Re} \mid \ldots$
f_{abs}	::=	$f_{prb_\div} \mid \otimes \mid marginal_{\sigma_1,\sigma_2} \mid f_{bij*} \mid dist_T \mid ret^{\tau}_T$
f_{bij}	::=	${f_{bij}}^{-1} \mid swap \mid negate \mid \ldots$
Variable names x	$\in$	$\Sigma^* \setminus \{match, with, exp, \ldots\}$
Strings s	$\in$	Σ^*_{Str}
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$

functions $f : [\![\sigma_1]\!] \to [\![\sigma_2]\!]$. For example, it is often possible to find a bijection $g : [\![\sigma_1]\!] \to [\![\sigma_3]\!] \times [\![\sigma_2]\!]$ such that $f = \pi_2 \circ g$, so that $f_* = marginal \circ g_*$.

4.3.2. Traced probabilistic programming

Although the above constructs are somewhat expressive, they are also quite a bit more cumbersome to use than the very general probabilistic programming constructs developed in Chapter 2. **Tracing** is a method of bridging the gap, allowing users to write arbitrary samplers while building an absolutely continuous distribution behind the scenes.

The key idea is that as a probabilistic program runs, it can track all of the random choices it makes, and store them in a **trace**: a dictionary mapping (user-specified) names of random choices to the values they took on in the execution. The program can be understood as defining a complex distribution over outputs in two steps: (1) it defines a distribution over traces, called the **trace distribution**, which is guaranteed to be absolutely continuous with respect to a universal stock measure ν_{Trace}, and (2) it defines a deterministic map, called the **return value function**, from traces to outputs. Under certain conditions, the pushforward by the return value function

will be absolutely continuous with respect to the return type's stock measure, and the trace can be marginalized away.

Preliminary definitions. We begin by defining a formal space of traces, and the universal stock measure on this space.

Definition 20 (space $\mathbb{T}$ of traces)**.** A **trace** $\{s_1 \mapsto v_1, \ldots, s_n \mapsto v_n\}$ is a finite dictionary mapping string-valued *names* s_i to values v_i of first-order types σ_i. We include a type of traces, Trace, as a first-order type in our language. Thus, traces may contain other traces nested within them.

Formally, we define a qbs of traces by solving the domain equation $\mathbb{T} \cong F(\mathbb{T})$, where

$$F(X) = \bigsqcup_{s \in \mathbb{S}} \prod_{(k,\sigma) \in s} [\![\sigma]\!][X]$$

Here, $\mathbb{S}$ is the set of **trace shapes**, lexicographically sorted lists of (k, σ) pairs, with unique string-valued names k. The notation $[\![\sigma]\!][X]$ resolves to $[\![\sigma]\!]$ if $\sigma \neq \text{Trace}$ and X if $\sigma = \text{Trace}$. The solution of the domain equation can be derived as the *initial algebra* of the polynomial functor F. We have $\mathbf{roll}_{\mathbb{T}} : F(\mathbb{T}) \to \mathbb{T}$ and its inverse $\mathbf{unroll}_{\mathbb{T}}$.

Definition 21 (Operations on traces)**.** We write $\{\}$ for the empty trace $\mathbf{roll}_{\mathbb{T}}([], ())$, and $\{k \mapsto v\}$ for the singleton trace $\mathbf{roll}_{\mathbb{T}}([(k, \sigma)], (v))$. We write $t_1 +\!\!+ t_2$ for the concatenation of two traces (returning $\{\}$ if names overlap in the two input traces). We also assume a map $\mathit{lookup}_\sigma : \text{Str} \to \mathbb{T} \to [\![\sigma]\!]$ for each σ, which returns a default value in $[\![\sigma]\!]$ if the given key is not found or if it maps to a different type. When σ is clear from context, we write $t[k]$ for $\mathit{lookup}_\sigma(k, t)$. Listing 4.1 gives the stock measure ν_{Trace} for traces, which is a sum over all possible trace shapes $s \in \mathbb{S}$ of products of the stock measures ν_σ for each σ in the shape.

Definition 22 (universal stock measure)**.** We define the universal stock measure ν_{Trace} on $\mathbb{T}$ as the sum of all possible trace shapes $s \in \mathbb{S}$ of products of the stock measures ν_σ for each σ in the shape. Formally, it is given as a least fixed point of the following map from measures to measures:

$$\nu \mapsto (g \mapsto \sum_{s \in \mathbb{S}} \left(\bigotimes_{(k,\sigma) \in s} \nu_\sigma[\nu] \right) (g \circ \mathbf{roll}_{\mathbb{T}} \circ \iota_s))$$

where ι_s is the canonical injection for the trace shape s into the disjoint union $F(\mathbb{T})$ and $\nu_\sigma[\nu]$ is either ν_σ (if σ is not Trace) or ν (if σ is Trace).

Lastly, we define a useful property of some distributions on $\mathbb{T}$—a property enjoyed by all trace distributions induced by traced probabilistic programs.

Definition 23 (discrete structure)**.** A measure μ on $\mathbb{T}$ has **discrete structure** if, for $(\mu \otimes \mu)$-almost-all pairs of traces (t_1, t_2), if $t_1 \neq t_2$ then there is some (k, σ) appearing in both traces' shapes such that $t_1[k] \neq t_2[k]$.

Programming with traces. We define a new family of types $G\ \tau$ of **traced probabilistic programs**. Semantically, $[\![G\ \tau]\!] \coloneqq \mathrm{Meas}^{DS}_{\ll \nu_{\mathrm{Trace}}} \times (\mathbb{T} \to [\![\tau]\!])$: a traced probabilistic program consists of a discrete-structured measure on traces that is absolutely continuous with respect to ν_{Trace}, together with a deterministic map from traces to values of type τ. These programs can be built compositionally using the following constructs:

- **Sampling from an absolutely continuous distribution:** Given a distribution $e_{dist} : D\ \sigma$ and a name $e_{name} : \mathrm{Str}$, we can form the probabilistic program $\mathbf{sample}_G\ e_{dist}\ e_{name} : G\ \sigma$. The program samples from the given distribution, records the result under the given name in the trace, and returns the result.

- **Deterministic computation:** Given an expression e of type τ, we can form the deterministic program $\mathbf{return}_G\ e : G\ \tau$. The program generates an empty trace and returns the deterministically computed value.

- **Sequential composition:** Given $e : G\ \tau_1$ and $x : \tau_1 \vdash \mathbf{do}\ \{m\} : G\ \tau_2$, we can form the sequential composition $\mathbf{do}_G\ \{x \leftarrow e; m\} : G\ \tau_2$, which runs the first program to generate a trace t_1 and a result v, then runs the second with x assigned to v to yield a trace t_2 and a result v'. The two traces are concatenated to form the output trace $t_1 +\!\!+ t_2$, and the result of the second program is returned.

Having constructed a traced probabilistic program, the built-in function

$$dist^{\tau}_{T} : G\ \tau \to D\ \mathrm{Trace}$$

extracts the absolutely continuous distribution on traces, and

$$ret^{\tau}_{T} : G\ \tau \to \mathrm{Trace} \to \tau$$

extracts the function mapping traces to return values.

Example 38. The program from Fig. 3-2 can be rewritten as a traced program:

$$\begin{aligned}
&e : G\ \mathbb{R}\\
&e = \mathbf{do}_G\ \{\\
&\quad b \leftarrow \mathbf{sample}_G\ flip_{\div}(0.3)\ \texttt{"b"};\\
&\quad \mathbf{if}\ b\ \mathbf{then}\ \mathbf{do}_G\ \{\\
&\qquad x \leftarrow \mathbf{sample}_G\ normal_{\div}(0,2)\ \texttt{"x"};\\
&\qquad \mathbf{return}_G\ (3 \times x)\}\\
&\quad \mathbf{else}\ \mathbf{do}_G\ \{\\
&\qquad x \leftarrow \mathbf{sample}_G\ normal_{\div}(0.7,1)\ \texttt{"y"};\\
&\qquad \mathbf{return}_G\ (x \div 2)\}\}
\end{aligned}$$

It denotes a distribution $\mu = [\![dist^{\mathbb{R}}_{T}\ e]\!]$ over traces of the form $\{\texttt{"b"} \mapsto true, \texttt{"x"} \mapsto r\}$

Listing 4.3 Semantics of absolutely continuous and traced probabilistic programs

Expression e	**Denotation** $[\![e]\!](\gamma)$
f_{prb_D}	$[\![f_{prb}]\!](\gamma)$
$e_1 \otimes e_2$	$[\![e_1]\!](\gamma) \otimes [\![e_2]\!](\gamma)$
$marginal(e)$	$\pi_{2*}([\![e]\!](\gamma))$
$f_{bij*}(e)$	$([\![f_{bij}]\!])_*([\![e]\!](\gamma))$
sample$_G$ e_{dist} e_{name}	(μ_T, f_R) where: $\mu_T = (\text{add}_k)_*[\![e_{dist}]\!](\gamma)$ $f_R(t) = t[k]$ $k = [\![e_{name}]\!](\gamma)$, $\text{add}_k(v) = \{k \mapsto v\}$
return$_G$ e	(μ_T, f_R) where: $\mu_T = \delta_{\{\}}$ $f_R(t) = [\![e]\!](\gamma)$
do$_G$ $\{x \leftarrow e; m\}$	(μ_T, f_R) where: $\mu_T = (+\!+)_*(disj? \odot (\mu_1 \otimes (\lambda t_1.\mu_{2,f_1(t_1)})))$ $f_R(t) = f_{2,f_1(t)}(t)$ $(\mu_1, f_1) = [\![e]\!](\gamma)$ $(\mu_{2,v}, f_{2,v}) = [\![\textbf{do}_G\ \{m\}]\!](\gamma[x \mapsto v])$ $disj?(t_1, t_2) = \mathbf{1}[keys(t_1) \cap keys(t_2) = \emptyset]$
$dist_T^\tau$	$(\mu_T, f_R) \mapsto \mu_T$
ret_T^τ	$(\mu_T, f_R) \mapsto f_R$

or $\{\texttt{"b"} \mapsto true, \texttt{"y"} \mapsto r\}$, for $r \in \mathbb{R}$, together with the *return value function*

$$f(t) = [\![ret_T^{\mathbb{R}}\ e]\!](t) = \begin{cases} t[\texttt{"x"}] \times 3 & \text{if } t[\texttt{"b"}] \\ t[\texttt{"y"}] \div 2 & \text{if } \neg t[\texttt{"b"}] \end{cases}$$

The trace distribution μ is absolutely continuous with respect to ν_{Trace}, and in this case, we also have that $f_*\mu \ll \nu_{\mathbb{R}}$. This latter property does not hold in general, however. For example, if the **else** branch returned a constant value c, then the pushforward $f_*\mu$ would not be absolutely continuous with respect to $\nu_{\mathbb{R}}$.

Marginalizing traces. If the user has written a program $e : G\ (D\ \sigma)$, denoting a measure μ over traces and a return value function f, we can construct the marginal

distribution $\mathit{marginal}(\mathit{dist}_T\ e \otimes \mathit{ret}_T\ e)$ of type $D\ \sigma$. We have

$$[\![\mathit{marginal}(\mathit{dist}_T\ e \otimes \mathit{ret}_T\ e)]\!] = \int\!\!\!\!\!\!\square f(t)\mu(dt)$$

and, as guaranteed by the type, it is absolutely continuous with respect to ν_σ.

Remark 17 (Connection to Pyro, Gen, Turing, and others)**.** The language for building traced probabilistic programs is very similar to that found in high-profile probabilistic programming languages such as Pyro [Bingham et al., 2019b], Turing [Ge et al., 2018], and Gen [Cusumano-Towner et al., 2019b]. Our type $D\ \sigma$ is analogous to the `Distribution` type in these systems. However, none of these systems support a construct akin to *marginal*, allowing a compound probabilistic program to be marginalized back down to a `Distribution` over its final outputs. This is because, even though the result of this operation is an absolutely continuous measure, its density function is typically not tractable to compute exactly, and existing systems rely on the exact computation of densities of `Distributions`. This chapter's proposal is to systematically replace the exact density requirement with something more lenient—the stochastic probability interface. Then, because we are able to tractably *estimate* the densities that arise, we are able to support modeling features (like *marginal*) that other systems cannot.

Semantics for traced probabilistic programs. The formal semantics of these constructs is given in Listing 4.3. Note that to compute the return value of a **do** expression given a trace, we do not attempt to split the trace into a first and second part; we give the entire trace to both computations. This may seem odd, but is based on a feature of the return value functions in our semantics: when $t \sim (\pi_1 \circ [\![e]\!])(\gamma)$, with probability 1, $(\pi_2 \circ [\![e]\!])(\gamma)(t +\!\!+ t') = (\pi_2 \circ [\![e]\!])(\gamma)(t)$ for all t' with names disjoint from those in t. That is, if t is a valid trace of e, then we can concatenate extra values to t without changing the value of e's value function on u.

Our semantics validates the following theorem:

Theorem 7 (Absolute continuity)**.** *Our semantics for expressions is well-defined with respect to our semantics of types, i.e., $[\![e]\!]$ does denote a value in $[\![\tau]\!]$ whenever $e : \tau$ holds. In particular:*

- *If $e : D\ \sigma$, then $[\![e]\!]$ is absolutely continuous with respect to ν_σ.*
- *If $e : G\ \tau$, then $\pi_1[\![e]\!]$ has discrete structure and is absolutely continuous w.r.t. ν_{Trace}.*

4.4 Compiling densities for absolutely continuous probabilistic programs

We now turn to the problem of automatically compiling absolutely continuous probabilistic programs to correct implementations of the stochastic probability interface (Def. 19). That is, given a program e denoting a measure μ (either of type $D\ \sigma$ or the trace measure component of $G\ \tau$), we want to automatically generate

Listing 4.4 Specification for the **spi** program transformation. A well-typed source-language expression $\Gamma \vdash e : \tau$ is transformed into a new well-typed expression $\mathbf{spi}\{\Gamma\} \vdash \mathbf{spi}\{e\} : \mathbf{spi}\{\tau\}$, such that

$$(\gamma_1, \gamma_2) \in R_\Gamma^{\doteq} \implies (\llbracket e \rrbracket(\gamma_1), \llbracket \mathbf{spi}\{e\} \rrbracket(\gamma_2)) \in R_\tau^{\doteq}.$$

Cases not shown are standard (see, e.g., Listing 3.6).

Type τ	**Transformed type** $\mathbf{spi}\{\tau\}$ **and spec** $R_\tau^{\doteq}$
$D\ \sigma$	$(\sigma \to P\ \mathbb{R}_{\geq 0}) \times P\ (\sigma \times \mathbb{R}_{\geq 0}) \times P\ \mathbb{B}$ $(\mu, (\xi, \chi, \epsilon)) \in R_{D\ \sigma}^{\doteq}$ if and only if: 1. χ **matches** ξ**:** $((x, w) \mapsto \mathbf{1}[w > 0]) \odot \chi = ((x, w) \mapsto w) \odot (\nu_\sigma \otimes \xi)$ 2. ξ **is unbiased:** If $\epsilon = \delta_{true}$, then $x \mapsto \int w\ \xi(x, dw)$ is a density of μ with respect to ν_σ.
$G\ \tau$	$(\text{Trace} \to P\ (\mathbb{R}_{\geq 0} \times \text{Trace})) \times P\ (\text{Trace} \times \mathbb{R}_{\geq 0}) \times P\ \mathbb{B} \times (\text{Trace} \to \mathbf{spi}\{\tau\})$ $((\mu, f), (\xi, \chi, \epsilon, g)) \in R_{G\ \tau}^{\doteq}$ if and only if: 1. **return value is correctly translated:** for all $t \in \mathbb{T}$, $(f(u), g(u)) \in R_\tau^{\doteq}$ 2. **SPI implementation is valid:** $(\mu, (\pi_{1*}\xi|_{\mathbb{T}_\mu}, \chi, \epsilon)) \in R_{D\ \text{Trace}}^{\doteq}$ 3. **trace remainders are correctly computed:** for μ-almost-all $t \in \mathbb{T}$, and all t' with addresses disjoint from those in t, $\pi_{2*}(\xi(t +\!\!+ t')) = \delta_{t'}$ 4. **density ignores irrelevant addresses:** $\forall t \notin \mathbb{T}_\mu, \pi_{1*}(\xi(t)) = \pi_{1*}(\xi(\mathbb{T}_\mu(t)))$ where $\mathbb{T}_\mu = \{t \in \mathbb{T} \mid \exists t' \in \mathbb{T}\ \text{s.t.}\ \frac{d\mu}{d\nu_{\text{Trace}}}(t +\!\!+ t') > 0\}$, and for $t \notin \mathbb{T}_\mu$, $\mathbb{T}_\mu(t)$ is the unique trace $t' \in \mathbb{T}_\mu$ s.t. t' is a subtrace of t and all other subtraces of t in $\mathbb{T}_\mu$ are subtraces of t'.

the code for its density estimator ξ and weighted sampler χ. As with **estimator** from Chapter 3, there are in principle many valid implementations of the SPI for a given measure μ, which strike different trade-offs between variance and computational cost. We adopt the same approach as we did for **estimator**, allowing user annotations to choose between the possible estimation strategies.

The code generated by our SPI compiler can be used directly within implementations of the algorithms from Fig. 4-1, or, since ξ is an unbiased density estimator, the user can apply the **integrator** transformation from Chapter 3 to obtain a (deterministic) density of type $\sigma \to \Re_{\geq 0}$. This density can then be further transformed (to generate unbiased estimates of *functions* of the density), or differentiated using the techniques we develop in Chapter 5.

Our compiler is formalized as a program transformation, denoted $\mathbf{spi}\{\cdot\}$, which translates a source expression $e : \tau$ into a target expression $\mathbf{spi}\{e\} : \mathbf{spi}\{\tau\}$. The transformation's high-level behavior (and its mapping on types) is given in Listing 4.4,

Listing 4.5 Several possible implementations of **spi** for $\lambda_{\ll}$ primitives.

Primitive f	**Strategy**	**Transformed primitive** $\mathbf{spi}\{f\}$
$flip_{\div}$	`EXACT`	$\lambda p.(\xi_p, \chi_p, \delta_{true})$, where:
	density	$\xi_p = \lambda b.$ **return** (**if** b **then** p **else** $1-p$)
	sampler	$\chi_p =$ **do** $\{b \leftarrow flip(p); w \leftarrow \xi_p(b);$ **return** $(b, w)\}$
$normal_{\div}$	`EXACT`	$\lambda(\mu,\sigma).(\xi_{\mu,\sigma}, \chi_{\mu,\sigma}, \delta_{true})$, where:
	density	$\xi_{\mu,\sigma} = \lambda x.$ **return** $\mathcal{N}(x;\mu,\sigma)$
	sampler	$\chi_{\mu,\sigma} =$ **do** $\{x \leftarrow normal(\mu,\sigma);\ w \leftarrow \xi_{\mu,\sigma}(x);$ **return** $(x,w)\}$
$\otimes$	`PRODUCT`	$\lambda((\xi_1,\chi_1,\epsilon_1),k).\ (\xi_\otimes(\pi_1\circ k), \chi_\otimes(\pi_2\circ k), \epsilon_\otimes(\pi_3\circ k))$, where:
	density	$\xi_\otimes = \lambda k_\xi.\lambda(x_1,x_2).$**do** $\{$ $w_1 \leftarrow \xi_1(x_1); w_2 \leftarrow k_\xi(x_1)(x_2);$ **return** $(w_1 \times w_2)\}$
	sampler	$\chi_\otimes = \lambda k_\chi.$ **do** $\{(x_1,w_1) \leftarrow \chi_1; (x_2,w_2) \leftarrow k_\chi(x_1);$ **return** $((x_1,x_2), w_1\times w_2)\}$
	unit test	$\epsilon_\otimes = \lambda k_\epsilon.$ **do** $\{b_1 \leftarrow \epsilon_1;$ $(x_1,w_1) \leftarrow \chi_1; b_2 \leftarrow k_\epsilon(x_1);$ **return** $(b_1 \wedge (w_1 = 0 \vee b_2))\}$
f_{bij*}	`CHANGE-OF-VAR`	$\lambda(\xi,\chi,\epsilon).(\xi_f,\chi_f,\epsilon)$, where:
	density	$\xi_f = \lambda y.$**do** $\{w \leftarrow \xi(f_{bij}{}^{-1}(y));$ **return** $\left(w \times \frac{df_{bij*}\nu_{\sigma_1}}{d\nu_{\sigma_2}}(y)\right)\}$
	sampler	$\chi_f =$ **do** $\{(x,w) \leftarrow \chi;$ **return** $(f_{bij}(x), \left(w \times \frac{df_{bij*}\nu_{\sigma_1}}{d\nu_{\sigma_2}}(f_{bij}(x))\right))\}$
$marginal_{\sigma_1,\sigma_2}$	`IS`$(q : \sigma_2 \to D\ \sigma_1)$	$\lambda(\xi_j,\chi_j,\epsilon_j).\ (\xi(\pi_2\circ q), \chi(\pi_1\circ q), \epsilon(\pi_1\circ q))$, where:
	density	$\xi = \lambda\chi_q.\ \lambda y.$ **do** $\{$ $(x,w_q) \leftarrow \chi_q(y);\ w_j \leftarrow \xi_j(x,y);$ **return** (**if** $w_q = 0$ **then** 0 **else** $w_j \div w_q)\}$
	sampler	$\chi = \lambda\xi_q.$ **do** $\{$ $((x,y),w_j) \leftarrow \chi_j; w_q \leftarrow \xi_q(y)(x);$ **return** $((x,y),$ **if** $w_q = 0$ **then** 0 **else** $w_j \div w_q)\}$
	unit test	$\epsilon = \lambda\xi_q.$ **do** $\{b \leftarrow \epsilon_j; ((x,y),w_j) \leftarrow \chi_j;$ $w_q \leftarrow \xi_q(y)(x);$ **return** $(b \wedge (w_j = 0 \vee w_q > 0))\}$

Listing 4.6 Implementation of the **spi** program transformation.

Expression e	**Transformed expression** $\mathbf{spi}\{e\}$
$()$	$()$
r	r
n	n
x	x
f (except f_{abs})	f
match e **with** $\{\ell_i x_i \mapsto e_i\}_{i=1}^n$	**match** $\mathbf{spi}\{e\}$ **with** $\{\ell_i x_i \mapsto \mathbf{spi}\{e_i\}\}_{i=1}^n$
(e_1, e_2)	$(\mathbf{spi}\{e_1\}, \mathbf{spi}\{e_2\})$
$\pi_i\ e$	$\pi_i\ \mathbf{spi}\{e\}$
$\lambda x.\, e$	$\lambda x.\, \mathbf{spi}\{e\}$
$e_1\ e_2$	$\mathbf{spi}\{e_1\}\ \mathbf{spi}\{e_2\}$
$\mathbf{return}_G\ e$	(ξ, χ, ϵ, f), where:
trace density	$\xi = \lambda t.\ \mathbf{return}\ (1, t)$
trace sampler	$\chi = \mathbf{return}\ (\{\}, 1)$
unit test	$\epsilon = \mathbf{return}\ \mathit{true}$
return function	$f = \lambda\_.\ \mathbf{spi}\{e\}$
$\mathbf{sample}_G\ e_{dist}\ e_{name}$	(ξ, χ, ϵ, f), where:
trace density	$\xi = \lambda t.\mathbf{do}\{w \leftarrow \xi_d(t[k]); \mathbf{return}\ (w \times has_\sigma(k), t \setminus \{k\})\}$
trace sampler	$\chi = \mathbf{do}\ \{(v, w) \leftarrow \chi_d; \mathbf{return}\ (\{k \mapsto v\}, w)\}$
unit test	$\epsilon = \epsilon_d$
return function	$f = \lambda t.t[k]$
	$(\xi_d, \chi_d, \epsilon_d) = (\pi_1\mathbf{spi}\{e_{dist}\}, \pi_2\mathbf{spi}\{e_{dist}\}, \pi_3\mathbf{spi}\{e_{dist}\})$
	$k = \mathbf{spi}\{e_{name}\}$
$\mathbf{do}_G\ \{x \leftarrow e; m\}$	(ξ, χ, ϵ, f), where:
trace density	$\xi = \lambda t.\ \mathbf{do}\ \{(w_1, t_1') \leftarrow \xi_1(t); x = f_1(t);$ $(w_2, t_2') \leftarrow \xi_2(t_1'); \mathbf{return}\ (w_1 \times w_2, t_2')\}$
trace sampler	$\chi = \mathbf{do}\ \{(t_1, w_1) \leftarrow \chi_1; x = f_1(t_1); (t_2, w_2) \leftarrow \chi_2;$ $\mathbf{return}\ (t_1 +\!\!+ t_2, w_1 \times w_2 \times \mathbf{1}[\mathit{disj?}(t_1, t_2)])\}$
unit test	$\epsilon = \mathbf{do}\ \{b_1 \leftarrow \epsilon_1; (t, w) \leftarrow \chi_1; x = f_1(t); b_2 \leftarrow \epsilon_2;$ $\mathbf{return}\ (b_1 \wedge (w = 0 \vee b_2))\}$
return function	$f = \lambda t.\mathbf{let}\ x = f_1(t)\ \mathbf{in}\ f_2(t)$
	$(\xi_1, \chi_1, \epsilon_1, f_1) = (\pi_i\mathbf{spi}\{e\})_{i=1}^4$
	$(\xi_2, \chi_2, \epsilon_2, f_2) = (\pi_i\mathbf{spi}\{\mathbf{do}\ \{m\}\})_{i=1}^4$
f_{abs}	*see Listing 4.5 for translations of primitives* f_{abs}

and the implementation is detailed in Listings 4.5 and 4.6. Below, we explain the intuition behind these translations for the key constructs.

4.4.1. Primitives, products, and pushforwards

Primitives ($f_{prb_\div}$): For primitive distributions like $normal_\div$ or $flip_\div$, we assume that hand-coded implementations of the SPI are provided; see Listing 4.5 for two examples. These typically involve evaluating the exact density function for the primitive (so $\xi(x)$ is a Dirac delta on the density of x, and χ returns an exact sample from the primitive along with its density). However, the framework allows for system designers to build in estimated densities of primitive distributions, which could be useful if a primitive's exact density is expensive to evaluate.

Products ($\otimes$): The density of a joint distribution factorizes as a product of densities, of the first marginal and the conditional:

$$\frac{d(\mu \otimes k)}{d(\nu_1 \otimes \nu_2)}(x, y) = \frac{d\mu}{d\nu_1}(x) \times \frac{dk(x)}{d\nu_2}(y)$$

whenever $k(x) \ll \nu_2$ for μ-almost all x. Thus, if each factor is estimated unbiasedly (and independently), then the product of the estimates is an unbiased density estimator for the joint. This is the logic employed by the rule for $\otimes$ in Listing 4.5.

Pushforwards by bijections (f_{bij_*}): Let μ be a distribution on $[\![\sigma_1]\!]$ and $f : [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ be a PAP bijection.[3] We want the SPI for the pushforward measure $f_*\mu$, whose density is given by

$$\begin{aligned}\frac{df_*\mu}{d\nu_{\sigma_2}}(y) &= \frac{df_*\mu}{df_*\nu_{\sigma_1}}(y) \cdot \frac{df_*\nu_{\sigma_1}}{d\nu_{\sigma_2}}(y) \\ &= \frac{d\mu}{d\nu_{\sigma_1}}(f^{-1}(y)) \cdot \frac{df_*\nu_{\sigma_1}}{d\nu_{\sigma_2}}(y)\end{aligned}$$

We can estimate the first term using the density estimator for μ, and we can generally compute the second term exactly. For example, if f is implemented a (deterministic) host language supporting (standard) automatic differentiation, Huot et al. [2023] shows that one can obtain the *intensional Jacobian* of f, whose absolute determinant is precisely the necessary Radon-Nikodym derivative. For a more detailed argument and an explanation of what Jacobian to use when σ_1 and σ_2 are mixed discrete/continuous types or traces, see Lew et al. [2023c], Appendix D. For the purposes of the present chapter, we will assume that the system implementor has worked out the appropriate Radon-Nikodym derivative and built its implementation into the translation for the f_{bij_*} primitive.

[3] See Lee et al. [2020b] and Huot et al. [2023] for details on the class of PAP functions.

4.4.2. Marginal distributions

Measures built using only primitives, products, and pushforwards by bijections have tractable exact density functions, so we have not yet needed the full flexibility of the SPI. Marginalization changes this picture. Given a joint measure μ_j on $\sigma_1 \times \sigma_2$, the marginal $\pi_{2*}\mu_j$ has density

$$\frac{d\pi_{2*}\mu_j}{d\nu_2}(y) = \int \frac{d\mu_j}{d(\nu_{\sigma_1} \otimes \nu_{\sigma_2})}(x, y)\, \nu_{\sigma_1}(dx),$$

i.e., we need to compute (or estimate) an integral with respect to the stock measure ν_{σ_1}. Because there are many ways to estimate this integral, striking different trade-offs between variance and computational cost, we ideally want to give the user control over the choice of estimation strategy—but without requiring them to implement the estimators themselves, a generally error-prone task.

As in Chapter 3, our approach is to allow the user to annotate applications of each primitive (in this case, *marginal*) with a *strategy*. Interestingly, our automation of densities opens up new avenues for customizable estimation, most prominently via **importance sampling**. When specifying the `IS` estimation strategy for *marginal* (Listing 4.5), the user can choose an arbitrary **proposal kernel** $q : \sigma_2 \to D\ \sigma_1$, expressed within our language. The role of the proposal kernel is to specify which distribution $q(y, dx)$ to sample x values from when estimating the marginal density of y. Given a proposal kernel, we can then automatically generate the importance sampling estimator for the marginal density; critically, we *use the compiled SPI for the proposal kernel* to compute the necessary importance weights.[4]

Because the proposal kernel q may itself have been constructed using *marginal*, the class of estimation strategies expressible (and automatable) via `IS`(q) is quite general, and includes for example the *recursive auxiliary-variable inference* (RAVI) estimators presented by Lew et al. [2022b]. This recursive nesting is made possible by the fact that the SPI includes not just a density estimator ξ, but also a sampler χ that can be seen as estimating the reciprocals of densities—exactly what we need for the proposal kernel in importance sampling.

Proposal design. The choice of proposal kernel q determines the variance of the resulting importance sampling estimator. The optimal choice for q is the kernel mapping each y to the **posterior distribution** on x given y, i.e., the kernel q^* such that $\mu_j = swap_*(\pi_{2*}\mu_j \otimes q^*)$. If q^* is used as a proposal kernel, and its own SPI implementation is exact, then the marginal distribution's SPI implementaiton will also be exact. As q^* deviates from the optimal proposal, the variance of the

[4] In Listing 4.5, when q shows up in the target source code, it has already been compiled to the SPI. To understand why, recall from Chapter 3 that the estimation strategy for each primitive can be viewed as an additional argument to that primitive. Thus, the translation of the primitive should accept as *its* strategy argument the translated version of the source-program strategy. Note that because `IS`(q) is, technically, just an argument, it can in principle be chosen dynamically based on values only available at runtime, all without compromising our correctness results.

importance sampling estimator increases (in a sense that can be made precise in terms of the χ^2 divergence; see Lew et al. [2022b], Theorem 3).

Proposal validity and unit tests. It is also, unfortunately, possible to choose q that leads to an *invalid* SPI implementation for the marginal. In particular, if $q(y)$ is not *absolutely continuous* with respect to the posterior $q^*(y)$ for a non-zero measure set of y values, then the marginal's SPI implementation will no longer unbiasedly estimate the density of $\pi_{2*}\mu_j$. Intuitively, this requirement says that a proposal should cover the *support* of the posterior. Lew et al. [2020a] develops a type system to ensure statically that this property holds, but it comes at the cost of various restrictions on the programming language. In this work, we take a more permissive approach. We allow the user to freely specify q, but we automatically construct stochastic **unit tests** $\epsilon : P\ \mathbb{B}$ that return *false* if they detect that a choice of proposal is invalid. These unit tests are compositionally constructed as a program is built, and our correctness theorem in Section 4.4.4 establishes that *if* the unit tests constructed for a program pass with probability 1, then that program's SPI implementation is guaranteed to be correct.

4.4.3. Traced probabilistic programs

Our rules for compiling traced probabilistic programs ($G\ \tau$) are given in Listing 4.6. Our goal is ultimately to build SPI implementations for the trace distributions of such programs. However, in order to do so compositionally, we must also track some additional information.

First, unlike with $D\ \sigma$, where the distribution is always over first-order values, the type τ in $G\ \tau$ is arbitrary, and may itself be a type of distributions or traced programs. Therefore, we need to ensure that we are correctly compiling a translation of a traced program's *return value function* in addition to an SPI implementation for its trace distribution.

Second, unlike the distribution product constructor $\otimes$ (which produces a distribution on tuples), the sequencing operation for traced programs $\mathbf{do}_G\ \{x \leftarrow e; m\}$ yields a distribution on concatenations of traces. The concatenation operation erases the information of which addresses came from e and which came from m. Thus, given a trace whose density we wish to estimate, there is no way *a priori* to tell what subtrace to hand to the compiled SPI for e and which to save for the SPI for m. To get around this, the density estimators ξ that we compile for traced programs take in a trace t that they themselves must split into two parts, one to estimate the density of, and one to return as the **trace remainder**. Thus, the type of ξ is $\text{Trace} \to P\ (\mathbb{R}_{\geq 0} \times \text{Trace})$. Its desired behavior is formalized via the logical relation $R^{\div}_{G\ \tau}$ given in Listing 4.4.

Besides these wrinkles, the translation rules in Listing 4.6 are relatively straightforward: $\mathbf{sample}_G$ bears some resemblance to the rule for pushforwards (as it pushes a distribution forward by the bijection $v \mapsto \{k \mapsto v\}$ for some string k), and the rules for sequencing with $\mathbf{do}_G$ are similar to those for products, except for the need to split traces as described above.

4.4.4. Conditional correctness modulo automated unit tests

Automated unit tests for conditional correctness. As in Chapter 3, we now aim to reason about the correctness of our compilation procedure via logical relations. But to do so, we need to overcome a key challenge. Logical relations are used to prove by induction that some property holds of *all* programs in a language. But as discussed in Section 4.4.2, the correctness of our compiler (and particularly of its behavior on *marginal*) requires certain technical conditions on the user's program that we have not statically enforced. Intuitively, we want a way to state and prove a property of the form, "*If the user never violates rule X, then property Y holds.*" The challenge is making the premise ("*the user never violates rule X*") sufficiently formal, and integrating it into our proof.

Our approach to this problem is to extend our translation **spi** to *automatically generate unit tests* that check that our rules have not been violated. This allows us to specify formally and compositionally the exact assumptions necessary for correctness to hold. This approach integrates well with logical relations as a proof technique: we can use logical relations to establish the *unconditional* correctness of the unit tests (i.e., that they *always* encode sufficient conditions for the correctness of **spi** as a whole). The approach also has practical benefits: because the unit tests are runnable programs in our language, they also give users a way to detect when they have violated one of these assumptions.[5]

Listings 4.5 and 4.6 give the concrete construction for the unit tests $\epsilon : P\ \mathbb{B}$. We briefly summarize the logic:

- *Primitives*: Primitive distributions come equipped with unconditionally valid SPI implementations, so the unit tests deterministically **return** *true*.
- *Products*: A product $\mu \otimes k$ is translated correctly if μ is correct, and if for μ-almost-all x, $k(x)$ is translated correctly. A single run of the product measure's unit test (1) runs the unit test for μ, (2) generates $x \sim \mu$ using the weighted sampler for μ,[6] and (3) runs the unit test for $k(x)$. Thus, if there is some set of positive μ-measure on which k is faulty, this test will detect the problem with positive probability.
- *Pushforwards by bijections*: Pushforwards by bijections have valid SPI implementations if the original measures (of which they are pushforwards) do. Therefore, the unit test for a pushforward just runs the unit test for the argument measure.

[5] The unit tests are stochastic, and the guarantee is bugs are detected with *non-zero* probability (if they exist), but not necessarily with *high* probability. But because the unit tests are generally simple probabilistic programs, it may be possible for probabilistic model checkers to automatically verify that a given unit test can or cannot return *false*. Alternatively, even without additional tooling, the generated unit tests can be read as documentation, helping users to understand exactly what assumptions the system is making about the program they've written.

[6] Note that μ may be a subprobability measure, but our generated weighted samplers χ always are probability measures. To capture this, χ may sometimes return a weight of 0, in which case the accompanying x is not necessarily in the support of μ. Therefore, if we generate a weight of 0, the test passes without checking $k(x)$.

- *Marginals*: This is the key case. We require that the joint distribution μ_j being marginalized is valid (so we run its unit test ϵ_j), and that for μ_j-almost all (x, y) pairs, $q(y)$'s density estimator is almost surely non-zero for input x. This ensures absolute continuity of μ_j with respect to $swap_*(\pi_{2*}\mu_j \otimes q)$ (the usual requirement for importance sampling), and also validates (via Proposition 1) that q's weighted sampler is properly weighted for the restriction of ν_{σ_1} to the support of the posterior. Note that we do *not* need to run the unit test associated with the proposal q in order to ensure correctness for *marginal*. This is because our compiler *always* (i.e., unconditionally) generates valid SPI implementations for *some* measure; the unit test simply ensures that it is for the *right* measure. Because q is being used simply as a proposal for importance sampling, we do not actually care whether its SPI implementation is correct for the measure it is *intended* to denote, only that it is correct for *some* measure that is a valid proposal distribution for the task at hand.

The logic for traced probabilistic programs is similar, with sequential composition mirroring the product and $\mathbf{sample}_G$ mirroring the pushforward.

Logical relations for conditional correctness. We begin with the fundamental lemma for our logical relations $R^{\div}_{\tau}$ (Listing 4.4):

Lemma 6 (Fundamental lemma for **spi**)**.** *Let $\Gamma \vdash e : \tau$ be a well-typed expression in context Γ. Then* **spi**$\{e\}$ *has type* **spi**$\{\tau\}$ *in context* **spi**$\{\Gamma\}$*, and for all pairs of environments $(\gamma_1, \gamma_2) \in R^{\div}_{\Gamma}$, $([\![e]\!](\gamma_1), [\![\mathbf{spi}\{e\}]\!](\gamma_2)) \in R^{\div}_{\tau}$.*

Because our logical relation is entirely standard at all types except $D\ \sigma$ and $G\ \tau$, establishing correctness boils down to checking that our translation of each construct in Listings 4.6 (for $\mathbf{return}_G, \mathbf{sample}_G, \mathbf{do}_G$) and 4.5 (for primitives f_{abs}) properly preserves the necessary relations.

We can then prove our overall correctness theorem, which says that we compute valid implementations of the SPI for the denotation of the user's source program (Def. 19), *so long as our automatically generated unit test passes*.

Theorem 8 (Correctness of **spi**)**.** *Let $e : D\ \sigma$ be a closed program, denoting a measure $\mu = [\![e]\!]$ on $[\![\sigma]\!]$. Let $(\xi, \chi, \epsilon) = [\![\mathbf{spi}\{e\}]\!]$. Then if ϵ returns true with probability 1, (ξ, χ) implement the SPI for μ with respect to ν_σ.*

We also have the following corollary, by composing Theorem 1 with Theorem 8:

Corollary 1. *Let $e : D\ \sigma$ be a closed program denoting a measure $\mu = [\![e]\!]$ as above. Further suppose that the automated unit test $\epsilon = \pi_3([\![\mathbf{spi}\{e\}]\!])$ passes with probability 1 ($\epsilon = \delta_{true}$). Then*

$$[\![\lambda x.\ \mathbf{integrator}\{\pi_1 \mathbf{spi}\{e\}\ x\}\ (cast_{\mathbb{R}_{\geq 0} \to \mathfrak{R}_{\geq 0}})]\!] : [\![\sigma]\!] \to \overline{\mathbb{R}}_{\geq 0}$$

is a density of μ with respect to ν_σ.

5

DERIVATIVES

Together with the work in Chapter 3, this chapter constitutes a significant reformulation and extension of the research originally published in Lew et al. [2023b], which was done jointly with Mathieu Huot, Sam Staton, and Vikash Mansinghka.

5.1 Motivation and overview

Specifying and solving optimization problems has never been easier, thanks in large part to the maturation of programming languages and libraries that support *automatic differentiation* (AD). With AD, users can specify objective functions as programs, then automate the construction of programs for computing their derivatives. These derivatives can be fed into optimization algorithms, such as gradient descent or ADAM, to find local minima or maxima of the original objective function.

Unfortunately, there is an important class of functions that today's AD systems *cannot* differentiate correctly: those defined as *expected values* of probabilistic processes. Consider, for example, the reinforcement learning problem of optimizing the parameters of a robot's policy, based on simulations of its behavior in random environments. The practitioner hopes maximize the *expected* (i.e., average) reward across all possible runs of the simulator. But obtaining gradients of this objective is not straight-forward; naively applying AD to the stochastic reward simulator will in general give incorrect results. Instead, practitioners often resort to hand-derived gradient estimators that they must manually prove correct. And this dilemma is hardly unique to robotics: the optimization of expected values is a ubiquitous problem, arising also in machine learning, computational finance, and operations research.

In this chapter, we develop an automatic differentiation algorithm for our language. Crucially, our algorithm can differentiate functions that return the *opaque* real number type $\mathfrak{R}$ introduced in Chapter 3. This opens up a new workflow for practitioners

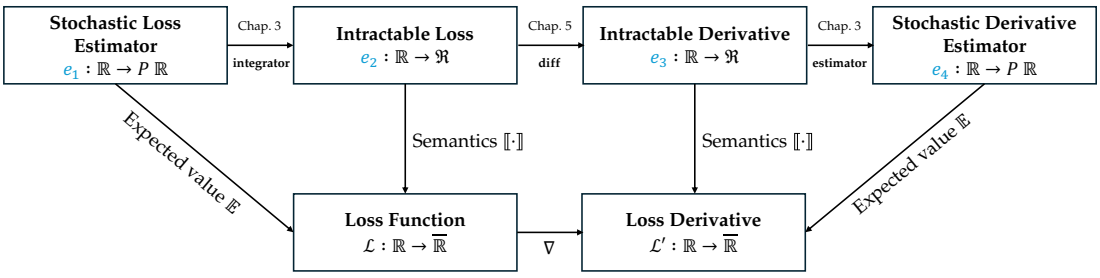


Figure 5-1: Our approach to differentiating loss functions defined as expected values. We start with a probabilistic program e_1, which, given a parameter θ of type $\mathbb{R}$, stochastically generates an *estimate* of the loss $\mathcal{L}(\theta)$ for parameter θ. By applying the **integrator** transformation of Chapter 3, we obtain e_2, a program denoting the deterministic loss function $\mathcal{L}$. This chapter's **diff** transformation can then differentiate the loss to obtain a program e_3 representing $\mathcal{L}'(\theta)$. Finally, the **estimator** transformation from Chapter 3 can be applied to obtain a stochastic estimator e_4 for the derivative. Running e_4 yields provably unbiased estimates of the loss's derivative, which can be used to guide optimization.

who need to optimize the expected values of stochastic processes:

1. The user provides a program e encoding a probabilistic process dependent on a real-valued parameter θ.
2. The user's goal is to find $\theta^* = \operatorname{argmin}_\theta \mathcal{L}(\theta)$, where the **loss function** $\mathcal{L}$ maps a parameter value θ to the *expected return value* of e, run on input θ.
3. The user applies the **integrator** transformation to e to obtain a new program representing $\mathcal{L}$. Since $\mathcal{L}$ is intractable, the new program returns the *opaque* real number type $\Re$.
4. Applying our AD algorithm to the new program yields another program of the same type, representing the derivative $\mathcal{L}'(\theta)$.
5. The user can then use the **estimator** transformation to obtain a provably unbiased estimator of $\mathcal{L}'(\theta)$—a runnable program of type $P\ \mathbb{R}$—which can be used during stochastic optimization.

Example. Figure 5-2 illustrates our method on a toy example. The loss function $\mathcal{L}$ is defined as the expectation of a program that flips a biased coin, with probability-of-heads θ. Depending on the outcome, we receive either 0 loss (the 'heads' case), or a *negative* loss of $-\frac{\theta}{2}$ (indicating a positive reward). The problem is to find the θ that minimizes expected loss. Intuitively, the optimal strategy must trade off the *benefits* of increasing θ (higher payoff in the 'tails' case) with its *drawbacks* (lower probability of entering the 'tails' case in the first place). The expected loss $\mathcal{L}(\theta) = \frac{\theta^2-\theta}{2}$ is minimized at $\theta = 0.5$.

Applying AD to only the deterministic parts of the program fails to account for

Stochastic Loss as a Probabilistic Program	AD on Deterministic Parts Only (incorrect)	Our Approach (correct derivative)
$e_{loss} = \lambda\theta.\,\textbf{do}\,\{$ $\quad b \leftarrow flip\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \textbf{return}\ -(\theta \div 2)$ $\}$	$e_{naive} = \lambda\theta.\,\textbf{do}\,\{$ $\quad b \leftarrow flip\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \textbf{return}\ -1 \div 2$ $\}$	$e_{correct} = \lambda\theta.\textbf{do}\,\{$ $\quad b \leftarrow flip\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \textbf{let}\ \delta p = 1 \div (\theta - 1)$ $\quad\quad \textbf{let}\ \delta l = -1 \div 2$ $\quad\quad \textbf{let}\ l = -\theta \div 2$ $\quad\quad \textbf{return}\ \delta l + l \times \delta p\}$
$\mathcal{L}(\theta) = \frac{\theta^2-\theta}{2}$	$\mathcal{L}'_{naive}(\theta) = \frac{\theta-1}{2}$	$\mathcal{L}'_{correct}(\theta) = \theta - \frac{1}{2}$

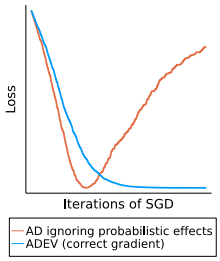


Figure 5-2: Differentiating a stochastic loss function. *Left:* The user has written a probabilistic program e_{loss} that produces stochastic estimates of a loss function. The objective is to minimize the true loss, $\mathcal{L}(\theta) = \mathbb{E}_{x\sim[\![e_{loss}]\!]}[x]$. *Middle:* Naively running standard automatic differentiation on the probabilistic program e_{loss} yields e_{naive}, where only the deterministic parts of the program have been differentiated. The resulting program yields *biased* estimates of the true derivative of the loss. That is, $\mathcal{L}'_{naive}(\theta) = \mathbb{E}_{x\sim[\![e_{naive}]\!]}[x] = \frac{\theta-1}{2} \neq \mathcal{L}'(\theta)$. Intuitively, standard AD fails to account for the dependence of the *distribution* of random choices (in this case, the coin flip) on the input parameter θ. *Right:* Our approach—in which the user explicitly integrates e_{loss} using the **integrator** transformation from Chapter 3, differentiates the resulting deterministic loss using **diff** from this chapter, and then estimates the resulting derivative using **estimator** (also from Chapter 3)—yields an *unbiased* estimator of the derivative, $e_{correct}$. *Plot:* Using an unbiased estimator within stochastic gradient descent leads to successful optimization, whereas using a biased estimator causes optimization to converge to the wrong value.

the effect of increasing θ on the *probability* of entering the high-reward branch. The resulting (incorrect) gradient is negative for all $\theta \in (0, 1)$; optimizing with it significantly overshoots the optimal value of 0.5. By contrast, our approach automatically introduces additional terms to account for the dependence of b on θ, leading to a gradient that can be soundly used to optimize the loss.

5.2 Review: Forward-mode automatic differentiation

In this section, we first review a standard algorithm for **forward-mode automatic differentiation** (AD) [Wengert, 1964] of deterministic programs without integrals. In the next section, we extend the algorithm to handle the type $\Re$ of intractable integral results (Chapter 3), enabling the composition of AD with the transformations developed earlier in this thesis.

Listing 5.1 Grammar of λ_∇, extending the language $\lambda_\Re$ from Listing 3.1. New productions are highlighted.

Syntactic category		Productions
Numeric types κ	::=	$\mathbb{R} \mid \mathbb{R}_{>0} \mid \mathbb{R}_{\geq 0} \mid \mathbb{R}_{[0,1]}$
Types τ	::=	$1 \mid \mathbb{N} \mid \kappa \mid \underline{\kappa} \mid \Re^n \mid \tau_1 \times \tau_2 \mid \ell_1 \tau_1 + \cdots + \ell_n \tau_n \mid \tau_1 \to \tau_2 \mid P\,\tau$
Expressions e	::=	$x \mid c \mid f \mid (e_1, e_2) \mid \pi_1\, e \mid \pi_2\, e \mid \lambda x.\, e \mid e_1\, e_2 \mid$ $\ell_i\, e \mid$ **match** e **with** $\{\ell_1 x_1 \mapsto e_1 \mid \cdots \mid \ell_n x_n \mapsto e_n\} \mid$ **return** $e \mid$ **do** $\{m\}$
Blocks m	::=	$e \mid x \leftarrow e; m$
Constants c	::=	$() \mid n \mid r$
Built-ins f_{det}	::=	$+ \mid - \mid \times \mid \div \mid exp \mid \iota_{\kappa \to \underline{\kappa}} \mid \underline{+} \mid \underline{\times} \mid \underline{-} \mid \underline{\div} \mid \underline{exp} \mid \ldots$
f_{prb}	::=	$flip \mid normal \mid \underline{normal} \mid geometric \mid \ldots$
f_{ext}	::=	$cast_{\mathbb{R} \to \Re} \mid cast_{\underline{\mathbb{R}} \to \Re} \mid +_\Re \mid -_\Re \mid \times_\Re \mid exp_\Re \mid sum_\infty \mid \mathbf{E}_{f_{prb}} \mid \ldots$
f_{dif}	::=	$f^1_{ext_D} \mid f^2_{ext_D}$
Variable names x	$\in$	$\Sigma^* \setminus \{match, with, exp, \ldots\}$
Naturals n	$\in$	$\mathbb{N}$
Real numbers r	$\in$	$\mathbb{R}$

5.2.1. Tracking differentiability with types

Listing 5.1 presents a very minor extension to the language $\lambda_\Re$ of Chapter 3. In our new grammar, we use κ to refer to any of our continuous, numeric ground types ($\mathbb{R}$, $\mathbb{R}_{>0}$, $\mathbb{R}_{[0,1]}$, $\mathbb{R}_{\geq 0}$). Then, we extend our type system to include not just the *raw* versions of each of these types, but also their *smooth* versions $\underline{\kappa} \in \{\underline{\mathbb{R}}, \underline{\mathbb{R}_{>0}}, \underline{\mathbb{R}_{\geq 0}}, \underline{\mathbb{R}_{[0,1]}}\}$. Semantically, $[\![\underline{\kappa}]\!] = [\![\kappa]\!]$: both the raw and smooth versions of a type denote the same underlying set of numbers. But we will use $\underline{\kappa}$ as the type for *parameters* with respect to which we want to differentiate, as well as any *functions of those parameters*. The type system will enforce that parameters of type $\underline{\kappa}$ are used *smoothly*, by assigning appropriate types to primitives. For example, we have:

$$\underline{+} : \underline{\mathbb{R}} \times \underline{\mathbb{R}} \to \underline{\mathbb{R}} \qquad \text{but} \qquad < : \mathbb{R} \times \mathbb{R} \to \mathbb{B}$$

indicating that addition is smooth, but comparison (via the less-than operator) is not. By default, constants r are not functions of any parameter, so they have type $\mathbb{R}$. But any value of type κ can be *cast* to a value of type $\underline{\kappa}$ using the primitive $\iota_{\kappa \to \underline{\kappa}}$. As syntactic sugar, we write $\underline{r}$ for $\iota_{\mathbb{R} \to \underline{\mathbb{R}}}(r)$.

A program that returns a value of type $\underline{\kappa}$ is guaranteed to be smooth with respect to any arguments of type $\underline{\kappa}$. For example, the following program has type $\underline{\mathbb{R}} \times \mathbb{R}_{>0} \to \underline{\mathbb{R}}$:

$$\lambda(\theta, x).\textbf{if } x < 2 \textbf{ then } \underline{exp}(\theta \div \iota_{\mathbb{R}_{>0} \to \underline{\mathbb{R}_{\geq 0}}}(x)) \textbf{ else } \underline{exp}(\theta \times \iota_{\mathbb{R}_{>0} \to \underline{\mathbb{R}_{\geq 0}}}(x))$$

The type system ensures that the function has a derivative with respect to θ, but

not necessarily with respect to x (and in this case, we indeed do have a point of non-differentiability at $x = 2$ for $\theta \neq 0$). In general, when clear from context, we will omit the $\iota_{\kappa \to \underline{\kappa}}$ casts, as well as the underlines on operations ($\underline{+}, \underline{\times}$, etc.).

5.2.2. Differentiating programs without integrals

Given a closed program e of type $\underline{\kappa} \to \underline{\mathbb{R}}$, we would like to automate the computation of its *derivative* $[\![e]\!]'(\theta)$ with respect to its argument θ. Forward-mode automatic differentiation [Wengert, 1964] does this in two steps:

1. First, we transform the program e into a new program $\mathbf{diff}\{e\} : \underline{\kappa} \times \underline{\mathbb{R}} \to \underline{\mathbb{R}} \times \underline{\mathbb{R}}$. This new program operates on **dual numbers**, pairs of numbers representing the *value* and *derivative* of some parameter-dependent function. For any differentiable h, applying $[\![\mathbf{diff}\{e\}]\!]$ to the dual number $(h(\theta), h'(\theta))$ should yield the dual number $(([\![e]\!] \circ h)(\theta), ([\![e]\!] \circ h)'(\theta))$.

2. Second, we construct the program $\lambda\theta.\pi_2(\mathbf{diff}\{e\}(\theta, 1))$. Since $(\theta, 1) = (id(\theta), id'(\theta))$, $[\![\mathbf{diff}\{e\}]\!](\theta, 1)$ should yield the dual number $([\![e]\!](\theta), [\![e]\!]'(\theta))$, and extracting the second component should give the derivative of the original program.

The **diff** transformation is given in Listings 5.2, 5.3, and 5.4.[1] The transformation replaces primitives f_{det} with *dual-number* versions, and otherwise, leaves the structure of the program entirely unchanged. The dual-number version of a primitive accepts dual numbers as arguments and returns dual numbers as output, propagating derivative information using the chain rule. For example, the dual-number version of multiplication is given by

$$\mathbf{diff}\{\times\} = \lambda((x, \frac{dx}{d\theta}), (y, \frac{dy}{d\theta})).\, (x \times y, x \times \frac{dy}{d\theta} + y \times \frac{dx}{d\theta}).$$

Instead of accepting two numbers as inputs, it accepts two *dual* numbers, $(x, \frac{dx}{d\theta})$ and $(y, \frac{dy}{d\theta})$. Its output is again a dual number, pairing the usual output of $\times$ (the product of x and y) with the *derivative* of this value with respect to θ, which can be computed using the product rule.

Expressions of raw numeric type (e.g., constants r of type $\mathbb{R}$) are guaranteed not to depend on parameters, and so when translating them, we leave them be: we do not need to track their derivatives with dual numbers. When casting a raw number into a parameter-dependent number, using the primitive $\iota_{\kappa \to \underline{\kappa}}$, we attach the dual component 0, to indicate that the value is a constant function of the parameter. This is reflected in the rule $\mathbf{diff}\{\iota_{\mathbb{R} \to \underline{\mathbb{R}}}\} = \lambda x.\, (\iota_{\mathbb{R} \to \underline{\mathbb{R}}} x, \underline{0})$.

[1] These listings also specify the behavior of the **diff** transformation on the type $\mathfrak{R}$, which we discuss in the next section. In this section we focus on just the standard AD algorithm, i.e., the restriction of **diff** to the simply-typed λ-calculus without $\mathfrak{R}$.

Listing 5.2 Specification for the **diff** program transformation. A well-typed source-language expression $\Gamma \vdash e : \tau$ is transformed into a new well-typed expression $\mathbf{diff}\{\Gamma\} \vdash \mathbf{diff}\{e\} : \mathbf{diff}\{\tau\}$, such that

$$(\gamma_1, \gamma_2) \in R^{\nabla}_{\Gamma} \implies (\llbracket e \rrbracket \circ \gamma_1, \llbracket \mathbf{diff}\{e\} \rrbracket \circ \gamma_2) \in R^{\nabla}_{\tau}.$$

Source type τ	**Transformed type $\mathbf{diff}\{\tau\}$ and spec R^{∇}_{τ}**
$\underline{\kappa}$	$\underline{\kappa} \times \underline{\mathbb{R}}$ $(f, f_D) \in R^{\nabla}_{\underline{\kappa}}$ if and only if: 1. f is a smooth function from $\mathbb{R}$ to $\llbracket \kappa \rrbracket$ 2. for all $\theta \in \mathbb{R}$, $f_D(\theta) = (f(\theta), \frac{df}{d\theta}(\theta))$
$\mathfrak{R}$	$\mathfrak{R}^2 \times (\mathbb{R} \to \underline{\mathbb{R}})$ $(f, \langle f_D, h \rangle) \in R^{\nabla}_{\mathfrak{R}}$ if and only: 1. for all $s \in \mathbb{R}$, $\theta \mapsto h(\theta)(s)$ is smooth 2. for all $\theta \in \mathbb{R}$, $f(\theta) = \int h(\theta)(s) ds$ 3. for all $\theta \in \mathbb{R}$, $f_D(\theta) = \left(f(\theta), \int \frac{\partial}{\partial \theta} h(\theta)(s) ds\right)$
$\sigma \in \{1, \mathbb{N}, \mathbb{R}, \mathbb{R}_{[0,1]}, \mathbb{R}_{>0}\}$	σ $\quad R^{\nabla}_{\sigma} = \{(f, f_D) \mid f = f_D\}$
$\tau_1 \times \tau_2$	$\mathbf{diff}\{\tau_1\} \times \mathbf{diff}\{\tau_2\}$ $R^{\nabla}_{\tau_1 \times \tau_2} = \left\{(f, f_D) \mid \forall i \in \{1,2\}, (\pi_i \circ f, \pi_i \circ f_D) \in R^{\nabla}_{\tau_i}\right\}$
$\ell_1 \tau_n + \cdots + \ell_n \tau_n$	$\ell_1 \mathbf{diff}\{\tau_1\} + \cdots + \ell_n \mathbf{diff}\{\tau_n\}$ $R^{\nabla}_{\sum_{i=1}^{n} \ell_i \tau_i} = \left\{(\ell_i \circ f, \ell_i \circ f_D) \mid (f, f_D) \in R^{\nabla}_{\tau_i}, i \in \{1, \ldots, n\}\right\}$
$\tau_1 \to \tau_2$	$\mathbf{diff}\{\tau_1\} \to \mathbf{diff}\{\tau_2\}$ $R^{\nabla}_{\tau_1 \to \tau_2} = \{(f, f_D) \mid \forall (h, h_D) \in R^{\nabla}_{\tau_1}, (\theta \mapsto f(\theta)(h(\theta)),$ $\theta \mapsto f_D(\theta)(h_D(\theta))) \in R^{\nabla}_{\tau_2}\}$

Source context Γ	**Transformed context $\mathbf{diff}\{\Gamma\}$ and spec R^{∇}_{Γ}**
$\cdot$ *(empty context)*	$\cdot$ $\quad R^{\nabla}_{\cdot} = \{(\theta \mapsto [], \theta \mapsto [])\}$
$\Gamma, x : \tau$	$\mathbf{diff}\{\Gamma\}, x : \mathbf{diff}\{\tau\}$ $R^{\nabla}_{\Gamma, x:\tau} = \{(\theta \mapsto \gamma(\theta)[x \mapsto v(\theta)], \theta \mapsto \gamma_D(\theta)[x \mapsto v_D(\theta)]) \mid$ $(\gamma, \gamma_D) \in R^{\nabla}_{\Gamma}, (v, v_D) \in R^{\nabla}_{\tau}\}$

Listing 5.3 Implementation of the **diff** program transformation.

Expression e	**Transformed expression** $\mathbf{diff}\{e\}$
$()$	$()$
r	r
n	n
x	x
$\mathbf{match}\ e\ \mathbf{with}\ \{\ell_i x_i \mapsto e_i\}_{i=1}^n$	$\mathbf{match}\ \mathbf{diff}\{e\}\ \mathbf{with}\ \{\ell_i x_i \mapsto \mathbf{diff}\{e_i\}\}_{i=1}^n$
(e_1, e_2)	$(\mathbf{diff}\{e_1\}, \mathbf{diff}\{e_2\})$
$\pi_1\ e$	$\pi_1\ \mathbf{diff}\{e\}$
$\pi_2\ e$	$\pi_2\ \mathbf{diff}\{e\}$
$\lambda x.\, e$	$\lambda x.\, \mathbf{diff}\{e\}$
$e_1\ e_2$	$\mathbf{diff}\{e_1\}\ \mathbf{diff}\{e_2\}$
$\underline{r}$	$(\underline{r}, \underline{0})$
f_{det}	*see Listing 5.4 for translations of primitives* f_{det}
f_{ext}	*see Listing 5.5 for translations of primitives* f_{ext}

Listing 5.4 Implementation of the **diff** transformation for standard primitives.

Primitive f	**Transformed primitive** $\mathbf{diff}\{f\}$
$+$	$\lambda((x, dx), (y, dy)).\ (x + y, dx + dy)$
$\times$	$\lambda((x, dx), (y, dy)).\ (x \times y, x \times dy + y \times dx)$
exp	$\lambda(x, dx).\ (exp(x), exp(x) \times dx)$

5.2.3. Correctness via logical relations on curves

If all of the primitives have been translated to correctly propagate dual numbers, then the entire translated program will also correctly propagate dual numbers. To prove this, we turn again to the technique of logical relations, as we did in both Chapter 3 and Chapter 4.

However, our logical relations will now have to be more sophisticated. In previous chapters, we have defined logical relations directly on the denotations of closed programs: we have defined relations R such that $(\llbracket e \rrbracket, \llbracket \mathbf{translate}\{e\} \rrbracket) \in R$ if $\mathbf{translate}\{e\}$ is a correct translation of e. But the same recipe cannot quite be applied here: at the type $\underline{\mathbb{R}}$, for example, there is no notion of when a particular dual number (r, dr) is a correct translation under **diff** of a number r. The number could have any dual component, depending on the program in which it appears.

The way around this problem comes from Huot et al. [2020], who show that correctness can be established by defining relations over pairs of *curves* rather

than over pairs of values. In Listing 5.2, for each type τ, we define a relation $R^{\nabla}_{\tau} \subseteq (\mathbb{R} \to [\![\tau]\!]) \times (\mathbb{R} \to [\![\mathbf{diff}\{\tau\}]\!])$, that relates a function $f : \mathbb{R} \to [\![\tau]\!]$ with its dual-number version $f_D : \mathbb{R} \to [\![\mathbf{diff}\{\tau\}]\!]$. At smooth numeric types $\underline{\kappa}$, for example, we have:

$$R^{\nabla}_{\underline{\kappa}} = \{(f, f_D) \mid f \text{ is smooth and } \forall \theta \in \mathbb{R}.\, f_D(\theta) = (f(\theta), f'(\theta))\}.$$

As before, this relation captures a notion of correctness for the translation of terms of type τ into terms of type $\mathbf{diff}\{\tau\}$. But now, instead of characterizing the correct translations of closed programs, we characterize the correct translations of programs that depend on some external parameter θ. Intuitively, if e is a term of type τ with a free variable $x : \underline{\mathbb{R}}$, then $\mathbf{diff}\{e\}$ is a correct translation of e if for all smooth h,

$$(\theta \mapsto [\![e]\!](x \mapsto h(\theta)), \theta \mapsto [\![\mathbf{diff}\{e\}]\!](x \mapsto (h(\theta), h'(\theta))) \in R^{\nabla}_{\tau}.$$

Apart from $\underline{\kappa}$, the logical relations at other types (products, sums, functions) are standard. We also define relations R^{∇}_{Γ} on pairs of environment-valued functions, relating environments that depend on an external parameter θ to dual-number environments that track derivatives with respect to θ. For ground types that are *not* $\underline{\kappa}$, we have $R^{\nabla}_{\tau} = \{(f, f_D) \mid f = f_D\}$: because values of these types cannot depend on the parameter θ, we do not require their translated versions to track derivatives.

Lemma 7 (Fundamental lemma for $\mathbf{diff}$, $\underline{\kappa}$-valued programs)**.** *Let $\Gamma \vdash e : \tau$ be a term of λ_{∇} without the type $\mathfrak{R}$ or any of the built-in primitives that operate on values of type $\mathfrak{R}$. Let $(\gamma, \gamma_D) \in R^{\nabla}_{\Gamma}$ be a related pair of environment-valued functions. Then*

$$([\![e]\!] \circ \gamma, [\![\mathbf{diff}\{e\}]\!] \circ \gamma_D) \in R^{\nabla}_{\tau}.$$

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : \tau$, and exactly mirrors that found in Huot et al. [2020].

Base cases. The base cases of the induction are the translations of the primitives f_{det} in Listing 5.4: we must verify that they are correct dual-number versions of the corresponding primitive functions. We give two examples:

- The translation of the multiplication primitive is given by

  $$\mathbf{diff}\{\underline{\times}\} = \lambda((x, dx), (y, dy)).\, (\underline{\times}(x, y), \underline{\times}(dx, y) + \underline{\times}(x, dy)).$$

  We must show that for all smooth functions $\mathbf{x}$ and $\mathbf{y}$,

  $$[\![\mathbf{diff}\{\underline{\times}\}]\!]((\mathbf{x}(\theta), \mathbf{x}'(\theta)), (\mathbf{y}(\theta), \mathbf{y}'(\theta))) = (\mathbf{x}(\theta) \times \mathbf{y}(\theta), \frac{d}{d\theta}(\mathbf{x}(\theta) \times \mathbf{y}(\theta))).$$

  This follows immediately from the product rule.

- The translation of the $\iota_{\mathbb{R}\to\underline{\mathbb{R}}}$ primitive is given by

$$\mathbf{diff}\{\iota_{\mathbb{R}\to\underline{\mathbb{R}}}\} = \lambda x.\,(\iota_{\mathbb{R}\to\underline{\mathbb{R}}} x, \underline{0}).$$

  We must show that for all constant real-valued functions $\mathbf{x}(\theta) = r$,

$$[\![\mathbf{diff}\{\iota_{\mathbb{R}\to\underline{\mathbb{R}}}\}]\!](\mathbf{x}(\theta)) = (\mathbf{x}(\theta), \frac{d}{d\theta}(\mathbf{x}(\theta))).$$

  This holds because the derivative of a constant function is 0.

Inductive cases. The inductive cases are automatic, because the logical relations for products, sums, and functions have been defined in the standard way, and the **diff** translation is defined to preserve introduction and elimination rules for products, sums, and functions. We again give two representative examples to illustrate the logic:

- Consider the expression $\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2$, which is translated into the expression $\mathbf{diff}\{\Gamma\} \vdash \lambda x.\, \mathbf{diff}\{e\} : \mathbf{diff}\{\tau_1\} \to \mathbf{diff}\{\tau_2\}$. Let $(\gamma, \gamma_D) \in R^{\nabla}_{\Gamma}$. By the inductive hypothesis for the premise $\Gamma, x : \tau_1 \vdash e : \tau_2$, we have that, for all $(h, h_D) \in R^{\nabla}_{\tau_1}$,

$$(\theta \mapsto [\![e]\!](\gamma(\theta)[x \mapsto h(\theta)]), \theta \mapsto [\![\mathbf{diff}\{e\}]\!](\gamma_D(\theta)[x \mapsto h_D(\theta)])) \in R^{\nabla}_{\tau_2}.$$

  But note that, by the definition of $[\![\lambda x.\, e]\!]$, this immediately implies that

$$(\theta \mapsto ([\![\lambda x.\, e]\!] \circ \gamma)(\theta)(h(\theta)), \theta \mapsto ([\![\lambda x.\, \mathbf{diff}\{e\}]\!] \circ \gamma_D)(\theta)(h_D(\theta))) \in R^{\nabla}_{\tau_2}.$$

  By the definition of $R^{\nabla}_{\tau_1\to\tau_2}$, this in turn implies that

$$([\![\lambda x.\, e]\!] \circ \gamma, [\![\lambda x.\, \mathbf{diff}\{e\}]\!] \circ \gamma_D) \in R^{\nabla}_{\tau_1\to\tau_2},$$

  as desired.

- Consider the expression $\Gamma \vdash \pi_1\ e : \tau_1$, which is translated into the expression $\mathbf{diff}\{\Gamma\} \vdash \pi_1\ \mathbf{diff}\{e\} : \mathbf{diff}\{\tau_1\}$. Let $(\gamma, \gamma_D) \in R^{\nabla}_{\Gamma}$. By the inductive hypothesis for the premise $\Gamma \vdash e : \tau_1 \times \tau_2$, we have that $(\theta \mapsto [\![e]\!](\gamma(\theta)), \theta \mapsto [\![\mathbf{diff}\{e\}]\!](\gamma_D(\theta))) \in R^{\nabla}_{\tau_1\times\tau_2}$. By the definition of $R^{\nabla}_{\tau_1\times\tau_2}$, this in turn implies that $(\theta \mapsto \pi_1([\![e]\!](\gamma(\theta))), \pi_1([\![\mathbf{diff}\{e\}]\!](\gamma_D(\theta)))) \in R^{\nabla}_{\tau_1}$. By the definition of $[\![\pi_1\ e]\!]$, this implies $([\![\pi_1\ e]\!] \circ \gamma, [\![\pi_1\ \mathbf{diff}\{e\}]\!] \circ \gamma_D) \in R^{\nabla}_{\tau_1}$, as desired.

We continue in this way for all base cases and inductive cases. □

With this lemma, we can establish the correctness of derivatives of user-defined functions:

Theorem 9 (Correctness of standard AD for $\underline{\kappa}$-valued programs)**.** *Let e be a closed program of type $\underline{\kappa_1} \to \underline{\kappa_2}$. Then $[\![\lambda x.\pi_2(\mathbf{diff}\{e\}(x, \underline{1}))]\!]$ is the derivative of $[\![e]\!]$.*

Proof. By Lemma 7, we have that $(\theta \mapsto [\![e]\!], \theta \mapsto [\![\mathbf{diff}\{e\}]\!]) \in R^{\nabla}_{\underline{\kappa_1}\to\underline{\kappa_2}}$. This implies

that, for all $(h, h_D) \in R^{\nabla}_{\underline{\kappa_1}}$, $([\![e]\!] \circ h, [\![\mathbf{diff}\{e\}]\!] \circ h_D) \in R^{\nabla}_{\underline{\kappa_2}}$. Let $r \in [\![\kappa_1]\!]$ and consider (h, h_D) defined as follows, depending on κ_1:

- If $\kappa_1 = \mathbb{R}$, then let $h(\theta) = \theta + r$ and $h_D(\theta) = (h(\theta), h'(\theta))$.
- If $\kappa_1 = \mathbb{R}_{[0,1]}$, then let $h(\theta) = \frac{r}{r+(1-r)e^{-\theta/(r-r^2)}}$, and $h_D(\theta) = (h(\theta), h'(\theta))$.
- If $\kappa_1 = \mathbb{R}_{>0}$ or $\mathbb{R}_{\geq 0}$, then let $h(\theta) = re^{\theta/r}$ and $h_D(\theta) = (h(\theta), h'(\theta))$.

In all cases, $(h, h_D) \in R^{\nabla}_{\underline{\kappa_1}}$. Furthermore, in all cases, $h(0) = r$ and $h'(0) = 1$. Now consider the value $[\![\lambda x.\pi_2(\mathbf{diff}\{e\}(x, \underline{1}))]\!](r)$, which is equal to $\pi_2(([\![\mathbf{diff}\{e\}]\!] \circ h_D)(0))$. But $([\![\mathbf{diff}\{e\}]\!] \circ h_D)(0) = (([\![e]\!] \circ h)(0), ([\![e]\!] \circ h)'(0))$, with second component

$$([\![e]\!] \circ h)'(0) = [\![e]\!]'(h(0)) \cdot h'(0)$$

by the chain rule. But since $h(0) = r$ and $h'(0) = 1$, this is just $[\![e]\!]'(r)$. Since the choice of r was arbitrary, this establishes that $[\![\lambda x.\pi_2(\mathbf{diff}\{e\}(x, \underline{1}))]\!] = [\![e]\!]'$, as desired. □

5.3 Differentiating integrals

We now extend the algorithm of the previous section to differentiate not just $\underline{\kappa}$-valued programs, but also $\mathfrak{R}$-valued programs. This enables us to compose AD with the transformations developed earlier in this thesis, to differentiate intractable integrals and density functions.

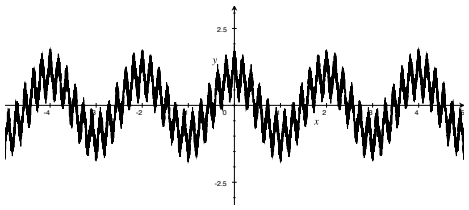
5.3.1. Dual extended-real numbers

The basic idea behind the extension is straight-forward: just as we replaced numbers with dual numbers in the previous section, we now replace intractable extended-real numbers ($\mathfrak{R}$) with *dual extended-real numbers* ($\mathfrak{R}^2$).

Challenges for correctness. Intuitively, just as an ordinary dual number tracks a parameter-dependent real number and its derivative with respect to the parameter, a dual extended-real number should track a parameter-dependent value of type $\mathfrak{R}$ and *its* derivative with respect to the parameter. But there is a challenge in making this precise. Recall from Section 3.5.1 that our extended-real numbers are, semantically, formal differences $a - b$, where either a or b (or both) may be infinite. It is not immediately clear how to think about derivatives of $\mathfrak{R}$-valued functions: what is the derivative of a function whose output may be infinite or undefined ($\infty - \infty$)?

Even when an $\mathfrak{R}$-valued function is finite, it may fail to be differentiable, because the higher-order primitives in our language do not necessarily preserve differentiability. For example, consider the primitive expectation operator for the geometric distribution, $\mathbf{E}_{geometric}$, with denotation

$$\mathbb{E}_{geometric} = p \mapsto f \mapsto \sum_{k=0}^{\infty} p(1-p)^k f(k).$$



$$g : \underline{\mathbb{R}} \to \mathfrak{R}$$
$$g = \lambda\theta.\ \mathbf{E}_{geometric}(0.5)\ (\lambda k.$$
$$cast_{\underline{\mathbb{R}} \to \mathfrak{R}}(\underline{2} \underline{\times} \underline{cos}(\underline{pow}(\underline{11}, k) \underline{\times} \underline{\pi} \underline{\times} \theta)))$$

Figure 5-3: In our language, it is possible to write a program g that denotes an everywhere-finite but nowhere-differentiable function $[\![g]\!] : \mathbb{R} \to \overline{\mathbb{R}}$. This poses a challenge for reasoning about the differentiation of programs with return type $\mathfrak{R}$.

Fig. 5-3 shows how we can use this primitive to define the function

$$g(\theta) = \mathbb{E}_{geometric}(0.5)(k \mapsto 2\cos(11^k \cdot \pi \cdot \theta)),$$

which is nowhere-differentiable with respect to θ.[2] This is the case even though cosine and multiplication are clearly differentiable. That is, even when the argument to the higher-order primitive $\mathbf{E}_{geometric}$ produces outputs that are smooth functions of θ, the expected value can fail to be smooth with respect to θ.

Specification for dual extended-real numbers. These difficulties lead us to adopt a different approach to dual numbers for the $\mathfrak{R}$ type. Recall that ordinary dual numbers track parameter-dependent values and their derivatives with respect to the parameter, $(f(\theta), f'(\theta))$. Dual extended-real numbers will similarly track a pair of θ-dependent values, $(f(\theta), g(\theta))$, where f and g both take values in $[\![\mathfrak{R}]\!]$, the space of formal differences. But the relationship between f and g is not as simple as it was for dual numbers: now, we require that there exists a function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, smooth with respect to its second argument, such that:

- $f(\theta) = \int_{\mathbb{R}} h(s, \theta)\ ds$ (that is, the positive and negative parts of $f(\theta)$ are given by $f^+(\theta) = \int_{\mathbb{R}} \max(h(s, \theta), 0)\ ds$ and $f^-(\theta) = -\int_{\mathbb{R}} \min(h(s, \theta), 0)\ ds$).
- $g(\theta) = \int_{\mathbb{R}} \frac{\partial}{\partial\theta} h(s, \theta)\ ds$ (that is, the positive and negative parts of $g(\theta)$ are given by $g^+(\theta) = \int_{\mathbb{R}} \max(\frac{\partial}{\partial\theta} h(s, \theta), 0)\ ds$ and $g^-(\theta) = -\int_{\mathbb{R}} \min(\frac{\partial}{\partial\theta} h(s, \theta), 0)\ ds$).

Although this is a more complex specification, many of the same intuitions carry over from the simple dual number case. For example, if $f : \mathbb{R} \to \mathbb{R}$ is a differentiable real-valued function, then we can consider its $[\![\mathfrak{R}]\!]$-valued analogue

$$([\![cast_{\mathbb{R} \to \mathfrak{R}}]\!] \circ f)(\theta) = \max(f(\theta), 0) - \max(-f(\theta), 0),$$

for which $([\![cast_{\mathbb{R} \to \mathfrak{R}}]\!](f(\theta)), [\![cast_{\mathbb{R} \to \mathfrak{R}}]\!](f'(\theta)))$ is a valid dual extended-real number representation. To see why it is valid, take $h(s, \theta) = 1_{[0,1]}(s) \cdot f(\theta)$, with derivative $\frac{\partial}{\partial\theta} h(s, \theta) = 1_{[0,1]}(s) \cdot f'(\theta)$. Taking integrals with respect to s yields the positive and negative parts of $f(\theta)$ and $f'(\theta)$.

[2] It is the Weierstrass function with parameters $a = 0.5$ and $b = 11$.

Explicit construction for correctness witnesses. In Listing 5.2, we give the **diff** translation of the $\mathfrak{R}$ type, along with the corresponding logical relation. When translating a (possibly parameter-dependent) expression of type $\mathfrak{R}$, we produce not just a dual number, but also the function $h : \mathbb{R} \to \underline{\mathbb{R}}$ that *witnesses* the correctness of the dual number we produce.[3] The logical relation $R^{\nabla}_{\mathfrak{R}}$, given in the listing, makes this idea precise. As before, the logical relation can be read as specifying when an *open* expression $e : \mathfrak{R}$, with a free variable $x : \underline{\mathbb{R}}$, is translated correctly: we require that $(\theta \mapsto [\![e]\!](x \mapsto \theta), \theta \mapsto [\![\textbf{diff}\{e\}]\!](x \mapsto (\theta, 1)))$ are related. Unpacking this, we see that $\textbf{diff}\{e\}$ is translated into two parts: a dual-number component $g(\theta) \coloneqq \pi_1([\![\textbf{diff}\{e\}]\!](x \mapsto (\theta, 1)))$, and a witness component $h(\theta) \coloneqq s \mapsto \pi_2([\![\textbf{diff}\{e\}]\!](x \mapsto (\theta, 1)))(s)$. The requirement for correctness is that $g(\theta) = (\int h(\theta)(s)\, ds, \int \frac{\partial}{\partial \theta}(h(\theta)(s))\, ds)$.

Transforming $\mathfrak{R}$-valued primitives. Listing 5.5 gives several examples of how **diff** transforms primitives with either inputs or outputs of type $\mathfrak{R}$. For each primitive f_{ext}, the translation is a short λ-expression that calls two helper primitives, $f^1_{ext_D}$ and $f^2_{ext_D}$, with which we assume the (target) language has been extended. The first primitive, $f^1_{ext_D}$, is responsible for computing the dual extended-real number output, based on dual number inputs. The second primitive, $f^2_{ext_D}$, is responsible for computing the correctness witness h, based on the correctness witnesses for any dual number inputs. Note that in both cases, implementors of our approach do not have to actually write code implementing the semantics of these new helper primitives: the first returns the opaque type $\mathfrak{R}$ (and so implementations can just return an opaque value), and the second is used solely for reasoning about correctness. That said, as we will see in the next section, implementors *do* need to pay attention to the semantics, because they will be responsible for extending the transformations from earlier in this thesis (and in particular, **estimator** from Chapter 3) to handle the new $\mathfrak{R}$-valued primitives $f^1_{ext_D}$.

We now walk through two example translations and why they are correct, relative to our specification:

- $cast_{\underline{\mathbb{R}} \to \mathfrak{R}}$: The input is a parameter-dependent real number, so in the transformed program, the input is a (standard) dual number, (x, dx). The output has two components: the output dual number, $cast^1_D(x, dx)$, and the witness of correctness, $cast^2_D(x, dx)$. The symbols $cast^1_D$ and $cast^2_D$ refer to new primitives whose semantics are given immediately after they are used in Listing 5.5. The first, $cast^1_D$, casts the real numbers x and dx to their $[\![\mathfrak{R}]\!]$-valued counterparts. The second, $cast^2_D$, maps a real number s to $1_{[0,1]}(s) \cdot x$.

  The correctness criterion we need to validate is as follows. Suppose the incoming dual number (x, dx) is equal to $(f(\theta), f'(\theta))$ for some $f : \mathbb{R} \to \mathbb{R}$. Then define $h(s, \theta) = [\![cast^2_D]\!](f(\theta), f'(\theta))(s)$. We must check that $[\![cast^1_D]\!](f(\theta), f'(\theta)) = (\int h(s, \theta)\, ds, \int \frac{\partial}{\partial \theta} h(s, \theta)\, ds)$. Unfolding the definition of $[\![cast^2_D]\!]$, we have $h(s, \theta) = 1_{[0,1]}(s) \cdot f(\theta)$, with derivative $\frac{\partial}{\partial \theta} h(s, \theta) = 1_{[0,1]}(s) f'(\theta)$. Integrating the positive

[3] The single argument to h is the number s; the parameter θ is implicit, just as it is in the original expression.

and negative parts of these functions with respect to s, we obtain the formal differences $[\![cast_{\underline{\mathbb{R}}\to\mathfrak{R}}]\!](f(\theta))$ and $[\![cast_{\underline{\mathbb{R}}\to\mathfrak{R}}]\!](f'(\theta))$ respectively, which is precisely $[\![cast^1_D]\!](f(\theta), f'(\theta))$, as required.

- $+_{\mathfrak{R}}$: The inputs are a pair of extended-real numbers, and so in the transformed program, the inputs are two *translated* extended-real numbers, δx and δy. Because **diff** translates extended reals into (a) dual extended-reals, and (b) correctness witnesses, each of δx and δy is in fact a pair of these two components. That is, $\delta x = ((x, dx), h_x)$ and $\delta y = ((y, dy), h_y)$. We assume that the inputs are valid, meaning that there are some functions $f_x, g_x, f_y, g_y : \mathbb{R} \to [\![\mathfrak{R}]\!]$ and $\mathbf{h}_x, \mathbf{h}_y : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ such that for each $v \in \{x, y\}$, we have $((v, dv), h_v) = ((f_v(\theta), g_v(\theta)), \mathbf{h}_v(\theta))$, $f_v(\theta) = \int \mathbf{h}_v(\theta)(s)\, ds$, and $g(\theta) = \int \frac{\partial}{\partial\theta}(\mathbf{h}_v(\theta)(s))\, ds$. Given all these inputs, the transformation of $+_{\mathfrak{R}}$ outputs both a new dual extended-real number, $[\![+^1_{\mathfrak{R}_D}]\!]((x, dx), (y, dy))$, and a correctness witness, $[\![+^2_{\mathfrak{R}_D}]\!](h_x, h_y)$.

  The dual extended-real number produced by $+^1_{\mathfrak{R}_D}$ takes extended-real sums in both the primal and dual components: $[\![+^1_{\mathfrak{R}_D}]\!]((x, dx), (y, dy)) = ([\![+_{\mathfrak{R}}]\!](x, y), [\![+_{\mathfrak{R}}]\!](dx, dy))$. The correctness witness produced by $+^2_{\mathfrak{R}_D}$ maps s to $1_{[0,1]}(s_L) \cdot h_x(s_R) + 1_{[1,2]}(s_L) \cdot h_y(s_R)$, where (s_L, s_R) is the result of *splitting* the number s into two real numbers. The function $split : \mathbb{R} \to \mathbb{R}^2$ can be taken to be any measurable isomorphism between $\mathbb{R}$ and $\mathbb{R}^2$ with the property that $split_* \Lambda = \Lambda^2$, where Λ is the Lebesgue measure.[4] In particular, this means that for all $f : \mathbb{R}^2 \to \mathbb{R}$, $\int_{\mathbb{R}} f(split(s))\, ds = \int_{\mathbb{R}} \int_{\mathbb{R}} f(s_L, s_R)\, ds_L\, ds_R$; that is, *split* lets us emulate a two-dimensional integral with a one-dimensional integral.

  We need to validate the following correctness property. First, define the function $h(s, \theta) = [\![+^2_{\mathfrak{R}_D}]\!](\mathbf{h}_x(\theta), \mathbf{h}_y(\theta))(s)$. We must check that

$$[\![+^1_{\mathfrak{R}_D}]\!]((f_x(\theta), g_x(\theta)), (f_y(\theta), g_y(\theta))) = \left(\int h(s, \theta)\, ds, \int \frac{\partial}{\partial\theta} h(s, \theta)\, ds\right).$$

  By the definition of $[\![+^2_{\mathfrak{R}_D}]\!]$, we have $h(s, \theta) = 1_{[0,1]}(s_L)\mathbf{h}_x(\theta)(s_R) + 1_{[1,2]}(s_L)\mathbf{h}_y(\theta)(s_R)$, where $(s_L, s_R) = split(s)$. The derivative of h is then given by $\frac{\partial}{\partial\theta} h(s, \theta) = 1_{[0,1]}(s_L)\frac{\partial}{\partial\theta}\mathbf{h}_x(\theta)(s_R) + 1_{[1,2]}(s_L)\frac{\partial}{\partial\theta}\mathbf{h}_y(\theta)(s_R)$. Integrating these functions with respect

[4] One concrete construction goes as follows. For any real number t, consider its unique representation as a sum $k + r$ with $k = \lfloor t \rfloor$ and $r \in [0, 1)$. Similarly, for any point $(t, s) \in \mathbb{R}^2$, write it as $(m + u, n + v)$, with each component expressed in the integer-plus-remainder representation. Let ϕ be a bijection $\mathbb{Z} \to \mathbb{Z}^2$. Now, given $t = k + r \in \mathbb{R}$, construct the pair of real numbers $split(t) = (m + u, n + v)$, where $(m, n) = \phi(k)$, and (u, v) are obtained by taking the even and odd bits, respectively, of the infinite binary expansion of r (the one ending in infinitely many zeros if r is dyadic).

to s, we obtain, for the primal value,

$$\begin{aligned}\int h(s,\theta)\,ds &= \int\int 1_{[0,1]}(s_L)\mathbf{h}_x(\theta)(s_R) + 1_{[1,2]}(s_L)\mathbf{h}_y(\theta)(s_R)\,ds_L\,ds_R \\ &= \int \mathbf{h}_x(\theta)(s_R)\,ds_R + \int \mathbf{h}_y(\theta)(s_R)\,ds_R \\ &= [\![+_{\Re}]\!](f_x(\theta), f_y(\theta)) = \pi_1([\![+^1_{\Re_D}]\!]((x,dx),(y,dy)))\end{aligned}$$

and for the dual component,

$$\begin{aligned}\int \frac{\partial}{\partial\theta}h(s,\theta)\,ds &= \int\int 1_{[0,1]}(s_L)\frac{\partial}{\partial\theta}\mathbf{h}_x(\theta)(s_R) + 1_{[1,2]}(s_L)\frac{\partial}{\partial\theta}\mathbf{h}_y(\theta)(s_R)\,ds_L\,ds_R \\ &= \int \frac{\partial}{\partial\theta}\mathbf{h}_x(\theta)(s_R)\,ds_R + \int \frac{\partial}{\partial\theta}\mathbf{h}_y(\theta)(s_R)\,ds_R \\ &= [\![+_{\Re}]\!](g_x(\theta), g_y(\theta)) = \pi_2([\![+^1_{\Re_D}]\!]((x,dx),(y,dy))).\end{aligned}$$

Translating expectation operator primitives. In addition to the first-order primitives of type $\Re$, we also have expectation operators $\mathbf{E}_{f_{prob}}$ in our language, which accept $\Re$-valued functions as inputs and produce $\Re$ values as outputs. Listings 5.6 and 5.7 give translations under **diff** of these primitives, following similar logic to the cases above.

5.3.2. Estimation rules for dual extended-real primitives

Our translations of primitives f_{ext} and $\mathbf{E}_{f_{prb}}$ under **diff** required us to introduce new primitives $f^1_{ext_D}$ and $f^2_{ext_D}$ into the language. In order for the **diff** program transformation to compose with the **estimator** program transformation from Chapter 3, we need to define the behavior of **estimator** on these new primitives (and in particular, on the primitives $f^1_{ext_D}$, which generate dual extended-real numbers that users may wish to esitmate). These estimation rules are highlighted in yellow in Listing 5.5. Note that dual numbers are estimated *jointly*: that is, some of the randomness used to estimate x may be shared with the randomness used to estimate dx.

5.3.3. Estimation rules for dual number expectation operators

Similarly, we must also attach estimation rules to the dual-number expectation operator primitives $\mathbf{E}^1_{f_{prbD}}$. Examples are given in Listings 5.6 and 5.7. Interstingly, many well-known gradient estimation strategies, such as the REINFORCE estimator [Williams, 1992] and the reparameterization trick [Kingma et al., 2014] can be naturally encoded as estimation rules for dual-number expectation operator primitives, in just a few lines of code each.

Some of these gradient estimation strategies come with restrictions on when they can be soundly applied. For example, the reparameterization trick can be applied to estimate derivatives of the form $\frac{d}{d\theta}\mathbb{E}_{x\sim\mathcal{N}(\mu(\theta),\sigma(\theta))}[g(\theta,x)]$, but only for

Listing 5.5 Example implementations of the **diff** program transformation for primitives f_{ext}.

Primitive f	**Transformed primitive** **diff**$\{f\}$
$cast_{\underline{\mathbb{R}}\to\mathfrak{R}}$	$\lambda(x, dx).(cast^1_D(x, dx), cast^2_D(x, dx))$
$[\![cast^1_D]\!]$	$(x, dx) \mapsto ([\![cast_{\underline{\mathbb{R}}\to\mathfrak{R}}]\!](x), [\![cast_{\underline{\mathbb{R}}\to\mathfrak{R}}]\!](dx))$
$[\![cast^2_D]\!]$	$(x, dx) \mapsto s \mapsto \mathbf{1}[s \in (0,1)] \cdot x$
estimator$\{cast^1_D\}$	$\lambda(x, dx)$. **return** (x, dx)
$+_{\mathfrak{R}}$	$\lambda(\delta x, \delta y).(+^1_{\mathfrak{R}_D}(\pi_1\delta x, \pi_1\delta y), +^2_{\mathfrak{R}_D}(\pi_2\delta x, \pi_2\delta y))$
$[\![+^1_{\mathfrak{R}_D}]\!]$	$((x, dx), (y, dy)) \mapsto ([\![+_{\mathfrak{R}}]\!](x, y), [\![+_{\mathfrak{R}}]\!](dx, dy))$
$[\![+^2_{\mathfrak{R}_D}]\!]$	$(h_x, h_y) \mapsto s \mapsto \mathbf{1}[s_L \in (0,1)] \cdot h_x(s_R) + \mathbf{1}[s_L \in (1,2)] \cdot h_y(s_R)$ where $(s_L, s_R) = split(s)$
estimator$\{+^1_{\mathfrak{R}_D}\}$ SUM strategy	$\lambda(\widehat{\delta x}, \widehat{\delta y})$. **do** { $(\tilde{x}, \widetilde{dx}) \leftarrow \widehat{\delta x}$; $(\tilde{y}, \widetilde{dy}) \leftarrow \widehat{\delta y}$; **return** $(\tilde{x} + \tilde{y}, \widetilde{dx} + \widetilde{dy})$ }
estimator$\{+^1_{\mathfrak{R}_D}\}$ SAMPLE strategy	$\lambda(\widehat{\delta x}, \widehat{\delta y})$. **do** { $b \leftarrow flip(0.5)$; $(\tilde{s}, \widetilde{ds}) \leftarrow$ **if** b **then** $\widehat{\delta x}$ **else** $\widehat{\delta y}$; **return** $(2 \times \tilde{s}, 2 \times \widetilde{ds})$ }
sum_∞	$\lambda\delta g.\ (sum^1_{\infty_D}(\pi_1 \circ \delta g), sum^2_{\infty_D}(\pi_2 \circ \delta g))$
$[\![sum^1_{\infty_D}]\!]$	$dg \mapsto ([\![sum_\infty]\!](\pi_1 \circ dg), [\![sum_\infty]\!](\pi_2 \circ dg))$
$[\![sum^2_{\infty_D}]\!]$	$h_g \mapsto s \mapsto \mathbf{1}[s_L \geq 0] \cdot h_g(floor(s_L))(s_R)$, where $(s_L, s_R) = split(s)$
estimator$\{sum^1_{\infty_D}\}$ SAMPLE strategy	$\lambda\widehat{\delta g}$. **do** { $n \leftarrow geometric(0.5)$; $(\tilde{x}, \widetilde{dx}) \leftarrow \widehat{\delta g}(n)$; **return** $(2^{n+1} \times \tilde{x}, 2^{n+1} \times \widetilde{dx})$ }

functions g that are differentiable with respect to both θ and x. It turns out such requirements can be enforced ergonomically using our type system. For continuous

Listing 5.6 Two strategies for estimating derivatives of $\mathbf{E}_{flip}$.

Primitive f	**Transformed primitive** $\mathbf{diff}\{f\}$
$\mathbf{E}_{flip_D}$	$\lambda(p, dp).\lambda(\delta g).(\mathbf{E}^1_{flip_D}(p, dp)(\pi_1 \circ \delta g), \mathbf{E}^2_{flip_D}(p)(\pi_2 \circ \delta g))$
$[\![\mathbf{E}^1_{flip_D}]\!]$	$(p, dp) \mapsto \delta g \mapsto ([\![\mathbf{E}_{flip}]\!](p)(g),$ $[\![+_{\Re}]\!]([\![\mathbf{E}_{flip}]\!](p)(dg), dp \cdot [\![-_{\Re}]\!](g(true)), g(false)))$ where $g = \pi_1 \circ \delta g, dg = \pi_2 \circ \delta g$
$[\![\mathbf{E}^2_{flip_D}]\!]$	$p \mapsto h_g \mapsto s \mapsto [s_L \in (0,1)] \cdot p \cdot h_g(true)(s_R) +$ $[s_L \in (1,2)] \cdot (1-p) \cdot h_g(false)(s_R)$ where $(s_L, s_R) = split(s)$
$\mathbf{estimator}\{\mathbf{E}^1_{flip_D}\}$	$\lambda(p, dp).\lambda\widehat{\delta g}.$ **do** {
`REINFORCE` strategy	$b \leftarrow flip(p);$ $(\tilde{l}, \widetilde{dl}) \leftarrow \widehat{\delta g}(b);$ **return** $(\tilde{l}, \widetilde{dl} +$ (**if** b **then** $dp \div p$ **else** $-dp \div (1-p)) \times \tilde{l})$
	}
`ENUMERATE` strategy	$\lambda\delta p.\lambda\widehat{\delta g}.$ **do** { $\widetilde{\delta l_1} \leftarrow \widehat{\delta g}(true);$ $\widetilde{\delta l_2} \leftarrow \widehat{\delta g}(false);$ **return** $(\mathbf{diff}\{\delta p \times \widetilde{\delta l_1} + (1-\delta p) \times \widetilde{\delta l_2}\})$
	}

probabilistic primitives such as *normal*, we introduce two variants, with different types: $normal : \underline{\mathbb{R}} \times \underline{\mathbb{R}_{>0}} \to P\,\mathbb{R}$ and $\underline{normal} : \underline{\mathbb{R}} \times \underline{\mathbb{R}_{>0}} \to P\,\underline{\mathbb{R}}$. With the first type, even if the parameters μ and σ depend on some continuous parameter θ, the sampled output x need not be used smoothly; it is not considered to depend smoothly on the parameter. By contrast, with the second type, the sampled output x has type $\underline{\mathbb{R}}$, obligating downstream computations to use it only in smooth ways. As a consequence of these types, the corresponding expectation operators accept different types of integrand: $\mathbf{E}_{normal}(\mu, \sigma)$ has type $(\mathbb{R} \to \Re) \to \Re$, whereas $\mathbf{E}_{\underline{normal}}(\mu, \sigma)$ has type $(\underline{\mathbb{R}} \to \Re) \to \Re$. This in turn that means that when estimating derivatives of $\mathbf{E}_{\underline{normal}}$, we can assume that integrand is a smooth function of the normal sample x, and thus it is safe to apply the reparameterization trick.

5.3.4. Correctness modulo dominated convergence

We can now extend Lemma 7 to cover the whole λ_∇ language, including the type $\Re$:

Lemma 8 (Fundamental lemma for **diff**, full version)**.** *Let $\Gamma \vdash e : \tau$ be a term of λ_∇. Let $(\gamma, \gamma_D) \in R^\nabla_\Gamma$ be a related pair of environment-valued functions. Then*

$$([\![e]\!] \circ \gamma, [\![\mathbf{diff}\{e\}]\!] \circ \gamma_D) \in R^\nabla_\tau.$$

Listing 5.7 Two strategies for estimating derivatives of Gaussian expectations, for $\mathbf{E}_{normal}$ and $\mathbf{E}_{\underline{normal}}$.

Primitive f	**Transformed primitive** $\mathbf{diff}\{f\}$
$\mathbf{E}_{normal}$	$\lambda(\delta\mu,\delta\sigma).\lambda(\delta g).(\mathbf{E}^1_{normal_D}(\delta\mu,\delta\sigma)(\pi_1\circ\delta g),$ $\mathbf{E}^2_{normal_D}(\pi_1(\delta\mu),\pi_1(\delta\sigma))(\pi_2\circ\delta g))$
$[\![\mathbf{E}^1_{normal_D}]\!]$	$(\delta\mu,\delta\sigma)\mapsto\delta g\mapsto([\![\mathbf{E}_{normal}]\!](\pi_1(\delta\mu),\pi_1(\delta\sigma))(\pi_1\circ\delta g),$ $[\![\mathbf{E}_{normal}]\!](0,1)(x\mapsto\frac{1}{\mathcal{N}(x;0,1)}\cdot$ $\pi_2([\![\mathbf{diff}\{cast_{\mathbb{R}\to\mathfrak{R}}(\mathcal{N}(x_x;x_\mu,x_\sigma))\times_{\mathfrak{R}}x_g(x_x)\}]\!](x_x\mapsto x,x_\mu\mapsto\delta\mu,x_\sigma\mapsto\delta\sigma,x_g\mapsto\delta g])))$
$[\![\mathbf{E}^2_{normal_D}]\!]$	$(\mu,\sigma)\mapsto h_g\mapsto s\mapsto\mathcal{N}(s_L;\mu,\sigma)\times h_g(s_L)(s_R)$ where $(s_L,s_R)=split(s)$
$\mathbf{estimator}\{\mathbf{E}^1_{normal_D}\}$ `REINFORCE` strategy	$\lambda(\delta\mu,\delta\sigma).\lambda\widehat{\delta g}.$ **do** { $x\leftarrow normal(\pi_1\delta\mu,\pi_1\delta\sigma);$ $(\tilde{l},\widetilde{dl})\leftarrow\widehat{\delta g}(x);$ $\delta x=(x,0)$ $\delta logp\leftarrow\mathbf{diff}\{\log(\mathcal{N}(\delta x;\delta\mu,\delta\sigma))\}$ **return** $(\tilde{l},\widetilde{dl}+\tilde{l}\times\pi_2(\delta logp))$ }
$\mathbf{E}_{\underline{normal}}$	$\lambda(\delta\mu,\delta\sigma).\lambda(\delta g).(\mathbf{E}^1_{\underline{normal}_D}(\delta\mu,\delta\sigma)(\pi_1\circ\delta g),$ $\mathbf{E}^2_{\underline{normal}_D}(\pi_1(\delta\mu),\pi_1(\delta\sigma))(\pi_2\circ\delta g))$
$[\![\mathbf{E}^1_{\underline{normal}_D}]\!]$	$((\mu,d\mu),(\sigma,d\sigma))\mapsto\delta g\mapsto$ $[\![\mathbf{E}_{normal}]\!](0,1)(\delta g\circ(x\mapsto(x\times\sigma+\mu,\sigma+x\times d\sigma+d\mu)))$
$[\![\mathbf{E}^2_{\underline{normal}_D}]\!]$	$(\mu,\sigma)\mapsto h_g\mapsto s\mapsto\mathcal{N}(s_L;0,1)\times h_g(\iota_{\mathbb{R}\to\underline{\mathbb{R}}}(s_L)\times\sigma+\mu)(s_R)$ where $(s_L,s_R)=split(s)$
$\mathbf{estimator}\{\mathbf{E}^1_{\underline{normal}_D}\}$ `REPARAM` strategy	$\lambda(\delta\mu,\delta\sigma).\lambda\widehat{\delta g}.$ **do** { $\epsilon\leftarrow normal(0,1);$ $\widehat{\delta g}((\iota_{\mathbb{R}\to\underline{\mathbb{R}}}(\epsilon),1)\ \mathbf{diff}\{\times\}\ \delta\sigma\ \mathbf{diff}\{+\}\ \delta\mu)$ }

Proof. The proof extends that of Lemma 7, adding new base cases for the primitives that handle values of type $\mathfrak{R}$. These cases are proven using the logic described in Section 5.3.1. □

The fundamental lemma allows us to prove a *weak* correctness lemma for the behavior of **diff** when applied to closed terms e of type $\underline{\mathbb{R}}\to\mathfrak{R}$:

Lemma 9 (Weak correctness of **diff**). *Let e be a closed term of type $\underline{\mathbb{R}}\to\mathfrak{R}$, let*

$g = [\![\lambda x.\pi_2(\pi_1(\mathbf{diff}\{e\}(x,\underline{1})))]\!]$, and let $h = [\![\lambda x.\pi_2(\mathbf{diff}\{e\}(x,\underline{1}))]\!]$. Then: (1) the map $(s,\theta) \mapsto h(\theta)(s)$ is differentiable with respect to θ for all s; (2) $[\![e]\!] = \theta \mapsto \int h(\theta)(s)\,ds$; and (3) $g = \theta \mapsto \int \frac{\partial}{\partial\theta}(h(\theta)(s))\,ds$.

Proof. Claim (1) follows from the fact that $(s,\theta) \mapsto h(\theta)(s)$ is the denotation of a term whose type is $\mathbb{R} \times \underline{\mathbb{R}} \to \underline{\mathbb{R}}$, namely $\lambda(s,\theta).\,\pi_2(\mathbf{diff}\{e\}(\theta,\underline{1}))(s)$.

By Lemma 8, we have that $(\theta \mapsto [\![e]\!], \theta \mapsto [\![\mathbf{diff}\{e\}]\!]) \in R^{\nabla}_{\underline{\kappa}\to\mathfrak{R}}$. This implies that for all $(k,k_D) \in R^{\nabla}_{\underline{\kappa}}$, $([\![e]\!] \circ k, [\![\mathbf{diff}\{e\}]\!] \circ k_D) \in R^{\nabla}_{\mathfrak{R}}$. In particular, let $k = \mathrm{id}$ and $k_D(\theta) = (\theta, 1)$. Then consider the values $[\![e]\!](\theta)$, $g(\theta)$, and $h(\theta)$, which are equal to $([\![e]\!] \circ k)(\theta)$, $\pi_2(\pi_1(([\![\mathbf{diff}\{e\}]\!] \circ k_D)(\theta)))$, and $\pi_2(([\![\mathbf{diff}\{e\}]\!] \circ k_D)(\theta))$ respectively. Because $([\![e]\!] \circ k, [\![\mathbf{diff}\{e\}]\!] \circ k_D) \in R^{\nabla}_{\mathfrak{R}}$, we have that $([\![e]\!] \circ k)(\theta) = \int_{\mathbb{R}} (\pi_2(([\![\mathbf{diff}\{e\}]\!] \circ k_D)(\theta))(s))\,ds$ for all θ, so $[\![e]\!](\theta) = \int_{\mathbb{R}} h(\theta)(s)\,ds$, establishing claim (2). Finally, the fact that $([\![e]\!] \circ k, [\![\mathbf{diff}\{e\}]\!] \circ k_D) \in R^{\nabla}_{\mathfrak{R}}$ also gives us that for all θ, $g(\theta) = \int_{\mathbb{R}} \frac{\partial}{\partial\theta}(\pi_2(([\![\mathbf{diff}\{e\}]\!] \circ k_D)(\theta))(s))\,ds$. Letting $\mathbf{h}(a,b,s) = \pi_2(([\![\mathbf{diff}\{e\}]\!])(a,b))(s)$, we can apply the chain rule to see that

$$\frac{\partial}{\partial\theta}\mathbf{h}(k(\theta),k'(\theta),s) = \frac{\partial}{\partial a}\mathbf{h}(a,1,s)|_{a=k(\theta)} \cdot \frac{d}{d\theta}k(\theta) + \frac{\partial}{\partial b}\mathbf{h}(\theta,b,s)|_{b=k'(\theta)} \cdot \frac{d}{d\theta}k'(\theta).$$

But since $k'(\theta) = 1$ is constant, the second term is zero, and we are left just with the first term, which is equal to $\frac{\partial}{\partial\theta}\mathbf{h}(\theta,1,s) = \frac{\partial}{\partial\theta}h(\theta)(s)$. Thus claim (3) also holds. □

This weak correctness result can be made stronger, to show that automatic differentiation really does compute the derivative of the user's program, if we can verify a condition on the derivative of the correctness witness h.

Definition 24 (Local domination)**.** A function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is **locally dominated** if, for every $\theta \in \mathbb{R}$, there is a neighborhood $U(\theta) \subseteq \mathbb{R}$ of θ and an integrable function $m_{U(\theta)} : \mathbb{R} \to \mathbb{R}_{\geq 0}$ such that $\forall\theta' \in U(\theta), \forall s \in \mathbb{R}, |h(s,\theta')| \leq m_{U(\theta)}(s)$.

Establishing that the derivative of h with respect to θ is locally dominated ensures that any interchanges of integration with differentiation assumed by **diff** were sound, enabling us to prove the full correctness result for differentiation of $\mathfrak{R}$-valued programs:

Theorem 10 (Correctness of AD for $\mathfrak{R}$-valued functions, modulo local domination)**.** *Let e be a closed term of type $\underline{\mathbb{R}} \to \mathfrak{R}$, and suppose that $[\![e]\!](\theta)$ yields a finite formal difference $f^+(\theta) - f^-(\theta)$ for all $\theta \in \mathbb{R}$. Let $h = [\![\lambda x.\pi_2(\mathbf{diff}\{e\}(x,\underline{1}))]\!]$. Then the map $(s,\theta) \mapsto h(\theta)(s)$ is differentiable with respect to θ, and if the map $(s,\theta) \mapsto \frac{\partial}{\partial\theta}(h(\theta)(s))$ is locally dominated, then:*

- *For all $\theta \in \mathbb{R}$, $[\![\lambda x.\pi_2(\pi_1(\mathbf{diff}\{e\}(x,\underline{1})))]\!](\theta)$ is a formal difference $g^+(\theta) - g^-(\theta)$ with both components finite.*
- *The map $\theta \mapsto g^+(\theta) - g^-(\theta)$ is the derivative of the map $\theta \mapsto f^+(\theta) - f^-(\theta)$.*

Proof. Let $\mathbf{h}(s, \theta) = h(\theta)(s)$. By Lemma 9, we have that $f^+(\theta) - f^-(\theta) = \int_{\mathbb{R}} \mathbf{h}(s, \theta)\, ds$, so the condition that f^+ and f^- are finite implies $\mathbf{h}(\cdot, \theta)$ is Lebesgue-integrable for all θ. Lemma 9 also gives us that $\mathbf{h}$ is differentiable with respect to θ for all s. These two facts, together with the local domination of $\frac{\partial}{\partial\theta}\mathbf{h}(s, \theta)$, allow us to apply the measure-theoretic version of the Leibniz integration rule, which states that $\frac{d}{d\theta}\int_{\mathbb{R}} \mathbf{h}(s, \theta)\, ds = \int_{\mathbb{R}} \frac{\partial}{\partial\theta}\mathbf{h}(s, \theta)\, ds$. But by Lemma 9, the left-hand side is $\frac{d}{d\theta}(f^+(\theta) - f^-(\theta))$ and the right-hand side is $g^+(\theta) - g^-(\theta)$, as desired. □

One way of understanding this correctness guarantee is that, as a program is compositionally differentiated by the **diff** transformation, at various points certain assumptions are made about local domination in order to pass differentiation under the integral sign. But instead of presenting the user with a laundry list of assumptions to check, made at different points during differentiation, **diff** produces a *single* function—the correctness witness h—whose local domination must be checked to prove that the derivative computed by **diff** is correct. This can be viewed as a lightweight static analysis that summarizes the conditions under which the derivatives are correct; an interesting avenue for future work would be to develop further static analyses capable of discharging the local domination assumption.[5]

5.4 Related work

Gradient Estimation in Machine Learning. Our expectation operator primitives (and their translations under **estimator**) compositionally package many gradient estimation strategies developed in the machine learning community [Mohamed et al., 2020, Kingma and Welling, 2014a, Ranganath et al., 2014]. The work also extends a growing literature on *stochastic computation graphs* (SCGs) [Schulman et al., 2015, Weber et al., 2019, Foerster et al., 2018], the goal of which is to help practitioners derive unbiased gradient estimators for expectations of probabilistic processes represented as graphs. Recently, Krieken et al. [2021] presented Storchastic, a practical system for AD of stochastic computation graphs. Storchastic provides reverse-mode AD (often more efficient than the forward-mode AD in our paper); and is implemented for PyTorch [Paszke et al., 2019], a widely used, practical deep learning framework. Our work is complementary. We precisely formalize the general problem of automatic differentiation of expected values of probabilistic processes, in a way that applies to broad classes of probabilistic programs (including higher-order) that cannot easily be represented as computation graphs. Furthermore, our logical relations allow us to precisely formulate general conditions that new primitives' gradient estimators must satisfy to be compositionally added to the language.

Concurrently with our work, Arya et al. [2022] developed an intriguing new

[5] We note that existing frameworks for differentiating stochastic losses (e.g. Schulman et al. [2015], Weber et al. [2019], Krieken et al. [2021]) do not even state the local domination conditions in their correctness theorems—only the differentiability conditions that in our framework are enforced automatically with types.

approach to AD of probabilistic programs, which arises by extending forward-mode AD, but which unlike our approach, is not based on composing existing, well-understood estimation strategies. It is unclear what source-language features are covered by their algorithm (the authors caution, e.g., that general `if` statements are unsupported), but the low variance their estimators appear to achieve may open the door to stable optimization of objectives that have been out of reach using existing estimators. We have worked with the authors to incorporate their approach into our system as an additional estimator; our preliminary implementation adds little new code, inherits our support for `if` statements and other control flow, and interoperates with all of our other estimators.

To our knowledge, among frameworks for deriving unbiased gradient estimators (based on SCGs or Arya et al. [2022]'s stochastic triples), ours is the only one that handles objectives defined as *functions of* one or more expected values.

Correctness and Semantics for Probabilistic and Differentiable Programming. Partly enabled by new semantic foundations for probabilistic [Zhang and Amin, 2022a, Heunen et al., 2017, Ehrhard et al., 2018] and differentiable [Huot et al., 2020, Vákár, 2020, Sherman et al., 2021] programming, researchers have recently established a variety of correctness results for both automatic differentiation [Krawiec et al., 2022, Mazza and Pagani, 2021, Lee et al., 2020b, Abadi and Plotkin, 2020] and probabilistic program transformations [Ścibior et al., 2018, Lew et al., 2020b, Lee et al., 2020a] for increasingly expressive languages. We build most closely on *logical relations* approaches [Katsumata, 2013, Ahmed, 2006, Appel et al., 2007, Pientka et al., 2019] for proving properties of AD algorithms [Huot et al., 2020, Barthe et al., 2020, Brunel et al., 2020, Mazza and Pagani, 2021], and on works that use quasi-Borel spaces as a model of synthetic measure theory [Kock, 2011, Ścibior et al., 2018, Vákár et al., 2019]. Recent work has begun to formally investigate interactions of differentiability and probabilistic programming [Mak et al., 2021, Lee et al., 2020a, Lew et al., 2021b, Sherman et al., 2021], but not yet the properties of *AD* in the general probabilistic programming setting.

AD of Languages with Integration. Researchers have recently proposed languages with support both for integration and AD, including Teg [Bangaru et al., 2021], a differentiable first-order expression language with compact-domain integrals and arithmetic, its successor Potto [Michel et al., 2024], and λ_S [Sherman et al., 2021], a higher-order language with computable integration on $[0, 1]$ as a primitive. Using compact-domain integration, it is possible to express some probabilistic program expectations, but not all (e.g., λ_S cannot express probabilistic programs that use Gaussian distributions). Furthermore, unlike in Teg, Potto, and λ_S, the output of our algorithm can be fed to **estimator**, to produce a new probabilistic program that can be directly run to produce gradient estimates for optimization. A unique aspect of Teg and Potto is their support for *parametric discontinuities*, which can sometimes be mimicked in our language using discrete random choices like `flip`, but are in general prohibited by our type system.

AD in PPLs. Many practical probabilistic programming languages [Cusumano-Towner et al., 2019c, Bingham et al., 2019b, Narayanaswamy et al., 2017] support the automated estimation of gradients of a *particular* expectation with respect to probabilistic programs q: the gradient of the ELBO, $\nabla_\theta \mathbb{E}_{x \sim q_\theta}[\log p_\theta(x) - \log q_\theta(x)]$. We formalize and prove correct a more general algorithm for arbitrary expected values, giving theory that could help to understand when these algorithms are correct (as studied in a first-order language for independent Gaussians by Lee et al. [2020a]), and how they can be modularly extended to support new gradient estimation strategies, or the estimation of other expectations. Many PPLs also rely on AD for reasons *other than* differentiating expectations. Typically, these languages differentiate *deterministic* programs that are derived from or related to probabilistic ones. For example, the widely-used and highly-optimized Stan [Carpenter et al., 2017a] probabilistic programming system differentiates log densities of probabilistic programs, for use within Hamiltonian Monte Carlo. Venture [Mansinghka et al., 2014a, 2018b] and Gen [Cusumano-Towner et al., 2019c] also differentiate log densities, for HMC, gradient-based MAP optimization, and Metropolis-Adjusted Langevin Ascent. Gen also computes derivatives of user-defined *involutions* to automatically compute Jacobian corrections in reversible-jump MCMC [Cusumano-Towner et al., 2020].

Part III

Scaling Probabilistic Inference

6

PROGRAMMABLE BAYESIAN INFERENCE

This chapter combines results originally published in Lew et al. [2023a] and Becker et al. [2024]. It is joint work with McCoy Becker, Mathieu Huot, Matin Ghavami, Xiaoyan Wang, Martin Rinard, and Vikash Mansinghka.

A central application of the operations automated by this thesis is to *Bayesian inference*. Both Monte Carlo and variational approaches—which together account for the vast majority of Bayesian inference algorithms used in practice—require the evaluation of Radon-Nikodym derivatives, integrals of Radon-Nikodym derivatives, and/or gradients of these quantities. In this chapter, we use the program transformations developed by this thesis to implement modular libraries for *programmable inference* [Mansinghka et al., 2014b, 2018a], enabling users to specify models, proposal distributions, and variational families in a very expressive probabilistic language and automate the math necessary for inference. We evaluate our systems by developing case studies that apply custom Monte Carlo and variational inference algorithms to several challenging probabilistic models.

6.1 Programmable Monte Carlo inference with automated estimation of Radon-Nikodym derivatives

As discussed in Section 4.2, many Monte Carlo inference algorithms require the evaluation of *density ratios* or *Radon-Nikodym derivatives*. For example, in importance sampling with model P and proposal Q, we generate weighted samples (x_i, w_i) according to

$$x_i \sim Q \quad \text{and} \quad w_i = \frac{dP}{dQ}(x_i),$$

requiring the evaluation or estimation of the density of P with respect to Q. Similarly, in Metropolis-Hastings targeting a measure P, we propose a new state x' from a

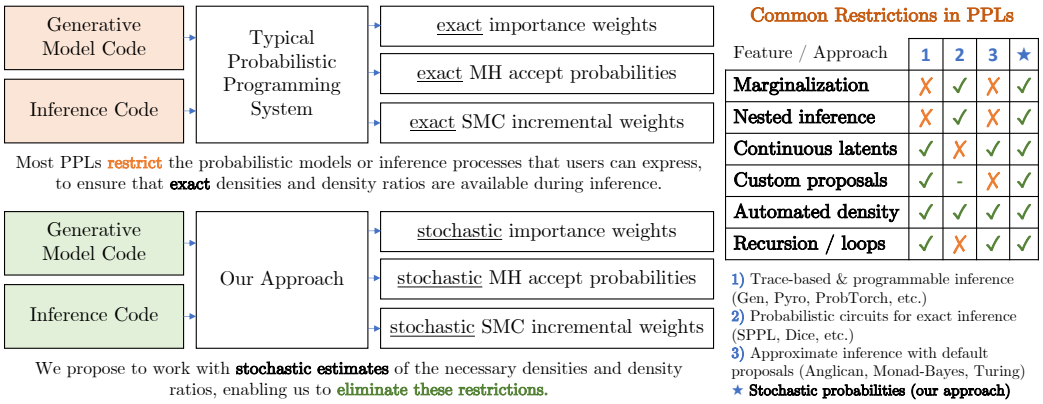


Figure 6-1: Many PPLs place restrictions on the models or inference algorithms users can encode, so exact densities can be efficiently automated. We eliminate these restrictions by using only stochastic estimates.

proposal K and accept it with probability

$$\alpha = \min\left(1, \frac{d(swap_*(P \otimes K))}{d(P \otimes K)}(x, x')\right).$$

Existing probabilistic programming languages provide some machinery for automating these density ratios for complex probabilistic models, which has helped researchers to invent and apply sophisticated modeling and inference techniques, achieving state-of-the-art results in a range of fields, including 3D scene understanding [Gothoskar et al., 2021], time series prediction [Saad et al., 2019b], Bayesian phylogeny [Ronquist et al., 2021], and large-scale scientific simulation [Baydin et al., 2019]. But existing systems typically place restricitons on the models or inference processes that users can express, in order to ensure that the necessary density ratios can be computed *exactly*.

In this section, we show how the machinery developed in this thesis—and particularly the ability to *estimate* intractable density ratios using the *stochastic probability interface* (SPI) introduced in Chapter 4—can be used to automate a broad range of Monte Carlo inference algorithms for a broad range of probabilistic models, beyond the reach of existing systems.

6.1.1. Motivation: Monte Carlo inference with intractable densities

Many existing probabilsitic programming languages omit certain features, to ensure that the density ratios necessary for Monte Carlo inference can be computed exactly. For example:

- **Marginalization.** In Gen [Cusumano-Towner et al., 2019c], Pyro [Bingham et al., 2019b], and ProbTorch [Stites et al., 2021b], users' models and proposals must be *joint* probability distributions over many primitive random choices; there is

no construct for *marginalizing* variables, because in general, marginal densities would require infinite sums or integrals to compute exactly.

- **Normalization (i.e., nested inference).** With few exceptions, PPLs do not support models and proposals that are *themselves* defined using the posteriors of other probabilistic programs [Zhang and Amin, 2022b, Rainforth, 2018]. Evaluating their densities would require computing *normalizing constants* for the inner models, which are generally not available exactly.
- **Infinite-support latents.** Languages with exact inference, such as Dice [Holtzen et al., 2020, Cheng et al., 2021] and SPPL [Saad et al., 2021], support marginalization and normalization, but forbid or highly restrict the use of *continuous* and other *infinite-support* probability distributions. These restrictions ensure that programs can be compiled to probabilistic circuits with densities that can be computed exactly, as finite sums.[1]
- **Custom proposals.** In Monad-Bayes [Ścibior et al., 2018], Anglican [Wood et al., 2014], and Turing [Ge et al., 2018], the *proposal distributions* used during inference are restricted to match the prior. When combined with another restriction—that *observed variables* must be modeled using primitive distributions as likelihoods—this ensures that the density ratios between models and proposals can be computed exactly, even if their individual densities cannot.
- **Automation for densities.** Some languages, such as Stan [Carpenter et al., 2017a], require users to directly encode their models *as* effective procedures for evaluating exact log densities, over a fixed number of continuous random variables.[2]

We show that these restrictions can be lifted, if we use Chapter 4.2's *stochastic probability estimates* instead of exact densities. As established by Theorem 6, using such estimates does not compromise the soundness of the overall Monte Carlo inference algorithms.

We have implemented the **spi** program transformation from Chapter 4 in an open-source tool called GenSP, which extends Gen [Cusumano-Towner et al., 2019c]. We use GenSP to evaluate the impact of using our estimated density ratios on six challenging inference problems, measuring the runtime overhead of our automation (compared to hand-coded versions of the same algorithms), as well as the impact on convergence of using stochastic probability estimates.

We find that stochastic probability density estimators automated by GenSP are competitive with hand-coded implementations of the same estimators, and when

[1] Some languages with exact inference (e.g., λPSI [Gehr et al., 2020] and Hakaru [Shan and Ramsey, 2017]) rely on sound but incomplete computer algebra to simplify some integrals and infinite sums; their modeling languages are not *explicitly* restricted, but *implicit* restrictions arise because on many programs, these symbolic techniques fail to produce densities that can be evaluated. These languages also lack general recursion.

[2] Stan has syntactic sugar that more closely resembles the syntax of other PPLs, and desugaring could be viewed as density automation. But this modeling sub-language has many of the restrictions already mentioned, e.g. *no marginalization*.

Table 6.1: Microbenchmarks for GenSP density estimators. Runtime of each density implementation for seven distributions from our case studies. For each distribution, runtime is reported for several typical density queries, with inputs of varying size. The mark ✗ indicates that no exact density evaluator is available.

benchmark	runtime			speedup vs. exact
	Exact[1]	Handcoded[2]	GenSP.jl[3]	
3DP3 [Gothoskar et al., 2021]				
5 objects, 2.1k points, 1.7k observed	984 ± 9 ms	57 ± 7 ms	40 ± 6 ms	**17x**
2D cloud, 18.2k points, 34.9k observed	47 ± 2 s	0.87 ± 0.05 s	1.04 ± 0.02 s	**45x**
Context-Correct [Mays et al., 1991]				
6 letters, edit distance 2	✗	14 ± 3 µs	47 ± 204 µs	∞
6 letters, edit distance 5	✗	90 ± 477 µs	220 ± 484 µs	∞
10 letters, edit distance 2	✗	19 ± 6 µs	65 ± 500 µs	∞
Context-Correct (trunc.) [Mays et al., 1991]				
6 letters, edit distance 2, max typos 2	1.06 ± 0.003 s	13 ± 1 µs	45 ± 153 µs	**23,555x**
6 letters, edit distance 5, max typos 2	1.10 ± 0.01 s	52 ± 379 µs	130 ± 803 µs	**2,068x**
10 letters, edit distance 2, max typos 2	3.89 ± 0.13 s	19 ± 6 µs	59 ± 212 µs	**204,736x**
4 letters, edit distance 3, max typos 3	145 ± 1 s	9 ± 102 µs	40 ± 335 µs	**3,625,000x**
RAVI-DPMM [Lew et al., 2022b]				
10 datapoints, 7 merges (good clustering)	2.51 ± 0.01 s	2.7 ± 1.6 ms	4.9 ± 2.3 ms	**512x**
40 datapoints, 37 merges (good clustering)	> 10 min	168 ± 4 ms	243 ± 6 ms	**>2,500x**
40 datapoints, 39 merges (bad clustering)	> 10 min	235 ± 7 ms	297 ± 7 ms	**>2,020x**
RSA [Goodman and Frank, 2016]				
depth 1	13 ± 4 ms	131 ± 422 µs	171 ± 480 µs	**76x**
depth 2	11 ± 0.1 s	1.3 ± 1.4 ms	1.7 ± 1.5 ms	**6,470x**
RANSAC-Regress [Fischler and Bolles, 1981]				
10 points, subset of 3	0.94 ± 1.3 ms	6 ± 24 µs	29 ± 182 µs	**32x**
10 points, subset of 5	1.7 ± 1.3 ms	9 ± 52 µs	28 ± 137 µs	**60x**
100 points, subset of 3	805 ± 233 ms	8 ± 43 µs	28 ± 153 µs	**28,750x**
100 points, subset of 5	> 10 min	7 ± 29 µs	31 ± 207 µs	**>19,000,000x**
Goal-Infer [Cusumano-Towner et al., 2017]				
300 RRT iters, 3500 refinements, likely dest.	✗	959 ± 252 µs	5.2 ± 1.5 ms	∞
300 RRT iters, 3500 refinements, unlikely dest.	✗	334 ± 237 µs	4.0 ± 1.5 ms	∞

[1]: exact density evaluators hand-coded in Julia, against the Gen.jl distribution interface, to expose as Gen primitives
[2]: density estimators hand-coded in Julia, against the GenSP.jl distribution interface, to expose as GenSP primitives
[3]: density estimators implemented within GenSP's unbiased-by-construction estimator DSL, exploiting automation

used within higher-level inference algorithms, the impact of estimator variance on convergence is negligible. These results suggest that automated estimators of Radon-Nikodym derivatives are a promising approach to scaling Bayesian inference to complex models and proposals.

6.1.2. Case studies

We evaluate the performance of our approach using GenSP, a version of the Gen.jl probabilistic programming library for Julia [Cusumano-Towner, 2020], extended to support our novel constructs.

Benchmarks. Our evaluation is based on a selection of inference problems from the literature; for each, we implement a model and inference algorithm in GenSP.

- **3DP3**: We implement Gothoskar et al. [2021]'s model of multi-object 3D scenes, based on a prior over *scene graphs* [Johnson et al., 2015, 2018], whose nodes are 3D objects from the YCB database [Calli et al., 2015], and whose edges encode contact relationships and relative poses. The inference task is to infer the scene graph

from a point cloud captured by a depth camera. We use the same custom MCMC proposals as in Gothoskar et al. [2021], but we use a GenSP density estimator for the likelihood, which is implemented as the **marginal** of a process that repeatedly generates noisy points from a latent cloud. As a result, the algorithm becomes a *pseudomarginal* MCMC algorithm [Beaumont, 2003, Andrieu and Roberts, 2009, Doucet et al., 2015].

- **Context-Correct**: A context-sensitive spelling correction task [Mays et al., 1991], applied to retyped sentences from telenovela screenplays. In our model of typos, each word is corrupted by a sequence of zero or more insertions, deletions, and substitutions; **marginal** is used to marginalize out the sequence of edits. We implement both MCMC inference and SMC inference that fixes one word at a time. Due to the estimated likelihood, these are instances of pseudomarginal MCMC [Andrieu and Roberts, 2009] and *random-weight* particle filtering [Fearnhead et al., 2010].

- **RSA**: We implement a model of pragmatic language understanding, based on the Rational Speech Acts framework [Goodman and Frank, 2016]. Unlike in typical formulations of RSA, in our GenSP model, agents reason about one another using *approximate* inference, encoded via **normalize**. As such it can be seen as a boundedly rational version of the model [Zhi-Xuan et al., 2020]. GenSP automatically estimates the densities of the approximate inference routines used by the agents. The resulting inference algorithm is an instance of IS^2 [Tran et al., 2013], because it nests an importance sampling estimator of the likelihood within importance sampling.

- **RANSAC-Regress**: A standard Bayesian regression model, but with inference based on the RANSAC algorithm [Fischler and Bolles, 1981, Cusumano-Towner and Mansinghka, 2018]. RANSAC chooses a small subset of the data, fits model parameters to it, and proposes nearby parameters from a Gaussian; we use **marginal** to marginalize out the choice of data subset. As a result, the proposal density is estimated by GenSP, and the overall algorithm is an instance of random-weight [Chopin et al., 2020, Chapter 8] or RAVI [Lew et al., 2022b] importance sampling.

- **Goal-Infer**: A model of an agent planning a path in a room, where the goal of inference is to infer the agent's destination from observations of their movements [Cusumano-Towner, 2020, Chapter 6]. The model is a GenSP program that uses **marginal** to marginalize out the agent's path-planning choices, which the agent makes using a randomized path planner [Zucker et al., 2007]. The resulting algorithm is an instance of IS^2 [Tran et al., 2013].

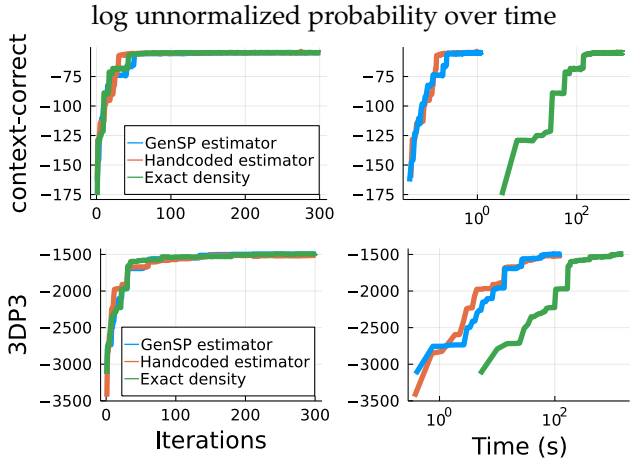
- **RAVI-DPMM**: A Bayesian agglomerative clustering algorithm from Lew et al. [2022b], applied to astronomy data. The model is a standard Dirichlet process mixture model [Neal, 2000], but importance sampling inference is performed using an involved proposal that runs a randomized agglomerative clustering algorithm on the data to propose a partition. As in Lew et al. [2022b], we use

a GenSP density estimator for the *proposal*, whose exact marginal density is an intractable sum over all possible sequences of merges that the agglomerative clustering algorithm makes. Our GenSP estimator uses **smc** within **marginal** to estimate the proposal's marginal density.

Experiments. Our evaluation is designed to answer two questions:

- *How do our sound-by-construction inference algorithms, with automated density estimators, perform, compared to hand-coded versions of the same algorithms?* Without GenSP, if practitioners wish to use fast density estimators within their inference algorithm implementation, they must hand-code the density estimators and ensure that they are unbiased. This can be time-consuming and error-prone, especially since there is no easy way to unit-test the unbiasedness of hand-coded estimators. GenSP provides a language for exploring the space of unbiased density estimation strategies, and convenient, correct automation. However, it also introduces overhead. **Our first experiment measures the overhead of GenSP's density estimators compared to hand-coded versions.** We used Julia to implement hand-coded versions of each density estimator automated by GenSP. We manually derived expressions for the density estimates, and wrote code to compute them, performing several manual optimizations (e.g., avoiding the computation of a density factor if it appeared both in the numerator and denominator of a density ratio, and using local variables to store random samples instead of heap-allocated traces). We measure both the overhead of each density query, and the overhead of the entire inference algorithm.
- *How does the noise introduced by stochastic probability estimators impact the accuracy of inference, compared to an idealized algorithm using exact inference?* GenSP's fast unbiased density estimation makes it possible to run inference using models and proposals for which it would be impossible, or impossibly slow, to evaluate exact densities. Thm. 6 proves inference is still sound, i.e., accurate in the limit of infinite MCMC iterations or SMC particles. But a natural question is whether inference accuracy with finite computation suffers as a result of using these stochastic estimators. **Our second experiment measures the convergence rates of MCMC and SMC using exact vs. stochastic densities, in terms of *number of iterations or particles*.** When exact densities are available but slow, we use them; when they are unavailable, we approximate them using estimates with very high particle counts, which we chose to ensure negligible variance. We also report wall-clock times for the GenSP and exact variants of each density and inference algorithm; these help illustrate that even when GenSP's convergence may be slightly slower as a function of iteration or particle count, it is significantly faster in wall-clock time.

Results: Overhead. Table 6.1 shows microbenchmarks from our first experiment. The overhead of our automation is generally 1-5x for individual density queries, but *end-to-end* overhead is lower: in Figs. 6-2 and 6-3, the righthand plots show that in wall-clock time, the GenSP inference algorithms are only 1-2x slower than the hand-coded versions. This is because our fast unbiased estimators account for



	Exact	GenSP
Context-Correct	41.1%	37.6%
3DP3	25.0%	18.5%

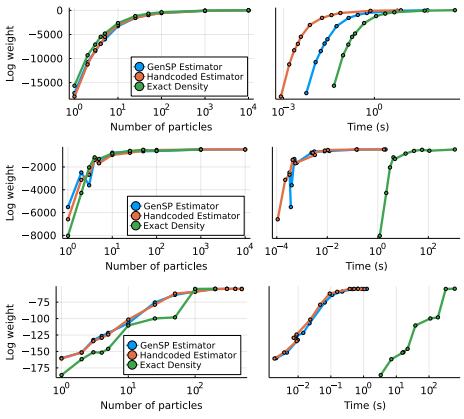
MH acceptance rates. Each iteration of Context-Correct performs 6 Metropolis-Hastings moves, and each iteration of 3DP3 performs 45. We report the percentage of iterations on which at least one proposal was accepted, across 10 runs of 300 iterations each. On average, using estimated densities led to a 10-25% reduction in acceptance rate, due to noise in the computation of the acceptance probability. Margin of error is about ±3% in all four measurements.

Figure 6-2: Impact of GenSP density estimators on convergence of MCMC inference. *Left:* We plot exact log probability vs. # of iterations, for MCMC algorithms for context-sensitive spelling correction and 3D scene understanding. *Right:* For each algorithm, we report the average acceptance probability across its run.

Goal-Infer (expensive, low-variance estimator used in lieu of intractable exact density)

RANSAC-LinReg

Context-Correct (truncated exact density)



	Exact	GenSP
Context-Correct	16.8	22.1
RAVI-DPMM	6.3	6.4
Goal-Infer	49.8	64.9
RANSAC-LinReg	1521	2402

Relative variance of particle weights. The number of particles required for accurate inference depends on the variance of the importance weights. Two factors contribute to variance [Lew et al., 2022b]: the quality of the proposal distribution (equal across settings), and the noise in the computation of densities (higher for GenSP).

Figure 6-3: Impact of GenSP density estimators on convergence of SMC and IS inference. *Left:* We plot average log marginal likelihood estimate vs. # of particles, for SMC algorithms for context-sensitive spelling correction, goal inference, and robust regression. *Right:* We report the estimated variance of particle weights when exact densities vs. GenSP density estimators are used. *Both:* When exact densities were unavailable or timed out, we instead used estimated densities with enough replicates to eliminate non-negligible variance.

relatively little of the inference runtime in these benchmarks. For example, in the CONTEXT-CORRECT benchmark, one iteration of MCMC takes about 3ms, of which 500μs (16%) is density evaluation. Using a hand-coded density cuts this to about 150μs, but an iteration still takes about 2.65ms; thus, inference with GENSP is only 1.13x slower than hand-coded. Profiling suggests future engineering could mitigate the sources of overhead, which include the construction of trace data structures, and the failure to automatically 'cancel' (and thus avoid computing) terms that match in the numerators and denominators of density ratios. For example, one optimization might fuse the bodies of proposal and model programs, eliminating common sub-expressions, and removing the need to construct traces to communicate between the two programs; local variables could be used directly to remember the proposed variables and score them under the model.

Results: Speed and convergence. The use of GENSP's unbiased density estimates does reduce the acceptance probability (in Metropolis-Hastings) and increases the variance of particle weights (in SMC), but these effects appear small and do not significantly affect the number of iterations or particles necessary for convergence, as shown in the plots in Figures 6-2 and 6-3. In the righthand plots, it can be seen that in wall-clock time the GENSP algorithms converge significantly faster than their exact-density counterparts. This is also reflected in Table 6.1, which shows orders-of-magnitude speedups over exact densities when they are available. We also see that our unbiased estimators scale favorably with problem size, whereas exact densities scale poorly.

6.2 Programmable variational inference with automated estimation of gradients of variational objectives

Variational inference is a popular approach to two fundamental probabilistic modeling tasks:

- **Fitting probabilistic models to data.** Given a family of joint probability distributions $\mathcal{P} = \{P_\theta(x, y) \mid \theta \in \mathbb{R}^n\}$ defined over *latent variables* x and *observed variables* y, find the one that best explains an observed dataset $\mathbf{y}$. For example, writing p_θ for the probability density function of P_θ, we may be interested in finding $\theta \in \mathbb{R}^n$ that maximizes the *marginal likelihood*

$$p_\theta(\mathbf{y}) = \int_X p_\theta(x, \mathbf{y}) dx. \tag{6.1}$$

- **Approximating intractable posterior distributions.** For a particular probabilistic model $P_\theta(x, y)$, find the best approximation to the (usually intractable) posterior distribution $P_\theta(x \mid \mathbf{y})$, from a class $\mathcal{Q} = \{Q_\phi(x) \mid \phi \in \mathbb{R}^m\}$ of tractable approximations (the *variational family*). For example, again using lower-case letters for probability density functions, we may be interested in finding ϕ that minimizes the *reverse*

KL divergence

$$D_{KL}(Q_\phi(x) \,||\, P_\theta(x \mid \mathbf{y})) = -\mathbb{E}_{x\sim Q_\phi}\left[\log \frac{p_\theta(x \mid \mathbf{y})}{q_\phi(x)}\right]. \tag{6.2}$$

Practitioners often aim to solve both these tasks at once, simultaneously fitting a probabilistic model and approximating its posterior distribution. To do so, one defines a *variational objective* $\mathcal{F} : \mathcal{P} \times \mathcal{Q} \to \mathbb{R}$, mapping particular distributions P_θ and Q_ϕ to a scalar *loss* (or *reward*). For example, one common choice is the *evidence lower bound*, or ELBO:

$$\textbf{ELBO}(P, Q) := \mathbb{E}_{x\sim Q}[\log p(x, \mathbf{y}) - \log q(x)] = \log p(\mathbf{y}) - D_{KL}(Q(x)||P(x \mid \mathbf{y})) \tag{6.3}$$

As the decomposition on the right-hand side suggests, maximizing the ELBO simultaneously maximizes the (log) marginal likelihood of the data $\mathbf{y}$ and minimizes the KL divergence of the posterior approximation Q to the posterior. Besides the ELBO, researchers have also proposed many alternative objectives [Dempster et al., 1977, Hinton et al., 1995, Agakov and Barber, 2004, Bornschein and Bengio, 2015, Burda et al., 2016, Ranganath et al., 2016, Rainforth et al., 2018, Sobolev and Vetrov, 2019, Malkin et al., 2022], which formalize the two goals of *fitting models* and *approximating posteriors* differently (e.g., by using divergences other than the KL). Once a variational objective has been defined, practitioners aim to find parameters (θ, ϕ) that maximize (or minimize) $\mathcal{F}(P_\theta, Q_\phi)$. There are many possible approaches to performing this optimization. The most popular methods rely on *gradients* $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$ of the objective—or more often, unbiased stochastic estimates of these gradients. Designing and implementing algorithms for estimating these gradients, with sufficiently low variance and computational expense, is the key roadblock on the path from *defining* a variational inference problem to solving it.

Indeed, although variational inference algorithms have found widespread adoption in Bayesian statistics [Fox and Roberts, 2012, Kucukelbir et al., 2017, **?**, Hoffman et al., 2013, Blei and Jordan, 2006] and in probabilistic deep learning [Kingma and Welling, 2014b, Pu et al., 2016, Kingma et al., 2021, Vahdat and Kautz, 2020, Malkin et al., 2022], implementing variational inference algorithms by hand remains a tedious and error-prone endeavor. The key mathematical ingredients specifying a variational inference problem—$\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{F}$—are typically not represented directly in code; rather, the practitioner must:

1. use algebra, probability theory, and calculus to derive a *gradient estimator*: a way to rewrite the gradient $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$ as an expectation $\mathbb{E}_{y\sim M_{(\theta,\phi)}}[f_{(\theta,\phi)}(y)]$, for some family of distributions M and some family of functions f; and then
2. write code to sample $y \sim M_{(\theta,\phi)}$ and evaluate $f_{(\theta,\phi)}(y)$—an unbiased estimate of $\nabla_{(\theta,\phi)}\mathcal{F}(P_\theta, Q_\phi)$.

It is often non-trivial to ensure that M and f are faithfully implemented, and that the math used to derive them in the first place is error-free. Small changes to $\mathcal{F}$, $\mathcal{P}$,

or Q, or to the gradient estimation strategy employed in step (1), can require large, non-local changes to M and f, and in implementing these changes, it is easy to introduce hard-to-detect bugs. When optimization fails, it is often unclear whether the problem is with the math, the code, or just the hyperparameters.

In this section, we show how the program transformations developed in this thesis can help to automate this process, producing unbiased estimates of the necessary gradients more modularly and with higher coverage than supported by existing approaches to VI in PPLs.

6.2.1. Motivation: Programmable variational inference with automated gradient estimators

Reflecting the importance of variational inference, many probabilistic programming languages (PPLs) (especially "deep" PPLs, such as Pyro [Bingham et al., 2018] and ProbTorch [Stites et al., 2021a]), feature varying degrees of automation for VI workflows. In these languages, users can express both models P and variational families Q as probabilistic programs; the system then automates the estimation of gradients for a pre-defined set of supported variational objectives $\mathcal{F}$. This design significantly lowers the cost of implementing and iterating on variational inference algorithms, but several pain points remain:

- **Incomplete coverage.** Existing PPLs offer limited or no support for many variational objectives, including forward KL objectives [Naesseth et al., 2020]; hierarchical, nested, or recursive variational objectives [Ranganath et al., 2016, Zimmermann et al., 2021, Lew et al., 2022a]; symmetric divergences [Domke, 2021]; trajectory-balance objectives [Malkin et al., 2022]; SMC-based objectives [Maddison et al., 2017, Gu et al., 2015, Li et al., 2023b, Naesseth et al., 2018]; and others. Today's PPLs also do not automate many powerful gradient estimation strategies, for example those based on measure-valued differentiation [Mohamed et al., 2020].

- **Duplicative engineering effort.** For PPL maintainers, supporting new gradient estimation strategies or language features requires separately introducing the same logic into the implementations of multiple variational objectives. Because this engineering effort is non-trivial, many capabilities are not uniformly supported. For example, as of this writing, Pyro's `ReweightedWakeSleep` objective [Le et al., 2019] does not support minibatching, even though other objectives do. As another example, variance reduction strategies such as data-dependent baselines and enumeration of discrete latents are implemented for the ELBO in Pyro, but not, e.g., for the importance-weighted ELBO.

- **Difficulty of reasoning.** The monolithic implementations of each variational objective's gradient estimation logic intertwine various concerns, including log density accumulation, automatic differentiation, gradient propagation through stochastic choices, and variance reduction logic. This can make it difficult to reason about correctness. Indeed, while the community has made tremendous progress

in understanding the compositional correctness arguments of an increasingly broad class of Monte Carlo inference methods for probabilistic programs [Ścibior et al., 2017, 2018, Lew et al., 2023a, Lundén et al., 2021, Borgström et al., 2016], pioneering work on correctness for variational inference [Lee et al., 2019, 2023, Li et al., 2023a] has generally focused on specific properties (e.g., smoothness and absolute continuity) in somewhat restricted languages, and not to end-to-end correctness of gradient estimation for variational inference.

The contributions of this thesis enable a highly modular, programmable alternative for supporting variational inference in PPLs. In this approach, all three ingredients of the variational inference problem—$\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{F}$—can be encoded as programs in our expressive probabilistic language, which supports compositional annotations for specifying the desired mix of gradient estimation strategies. We can then apply a sequence of modular program transformations—**spi**, **integrator**, **diff**, then **estimator**—to construct unbiased gradient estimators for the user's variational objective.

We have implemented this approach in `genjax.vi`, as well as in Haskell and Julia, and evaluated our system on several benchmark tasks, including the training of the challenging Attend-Infer-Repeat model [Eslami et al., 2016]. We find that `genjax.vi` makes it possible to encode new gradient estimators that converge faster and to better solutions than standard estimators, including those automated by Pyro [Bingham et al., 2019b]. Our results show that our transformations can be scalably and performantly implemented, to deliver competitive performance on realistic probabilistic deep learning workloads.

6.2.2. Example: Fitting a variational approximation

The typical workflow for a user of our system is as follows:

- **Model and variational programs.** The user begins by writing two probabilistic programs in the language $\lambda_{\ll}$ of Chapter 4: a *model program* and a *variational program*. Recall that $\lambda_{\ll}$ is a trace-based PPL that resembles Pyro [Bingham et al., 2018], ProbTorch [Stites et al., 2021a], and Gen [Cusumano-Towner et al., 2019b]. The model program encodes a family of joint probability distributions $P_\theta(x, y)$, and the variational program encodes a family of distributions $Q_\phi(x)$, possible approximations to the posterior $P_\theta(x \mid \mathbf{y})$ for data $\mathbf{y}$.

- **Objective function.** The user now seeks to find values of (θ, ϕ) that simultaneously (1) fit P_θ to the data $\mathbf{y}$, and (2) make Q_ϕ close to the posterior $P_\theta(x \mid \mathbf{y})$. To make these informal desiderata precise, the user defines an *objective function*, which is often defined by taking integrals and Radon-Nikodym derivatives with respect to the measures denoted by the model and variational programs. This is done using the program transformations from Chapters 3 and 4. The availability of program transformations for integration and for Radon-Nikodym derivatives allows users to concisely express objectives like the ELBO (Eqn. 6.3).

- **Gradient estimator.** The final step is to optimize the objective function via

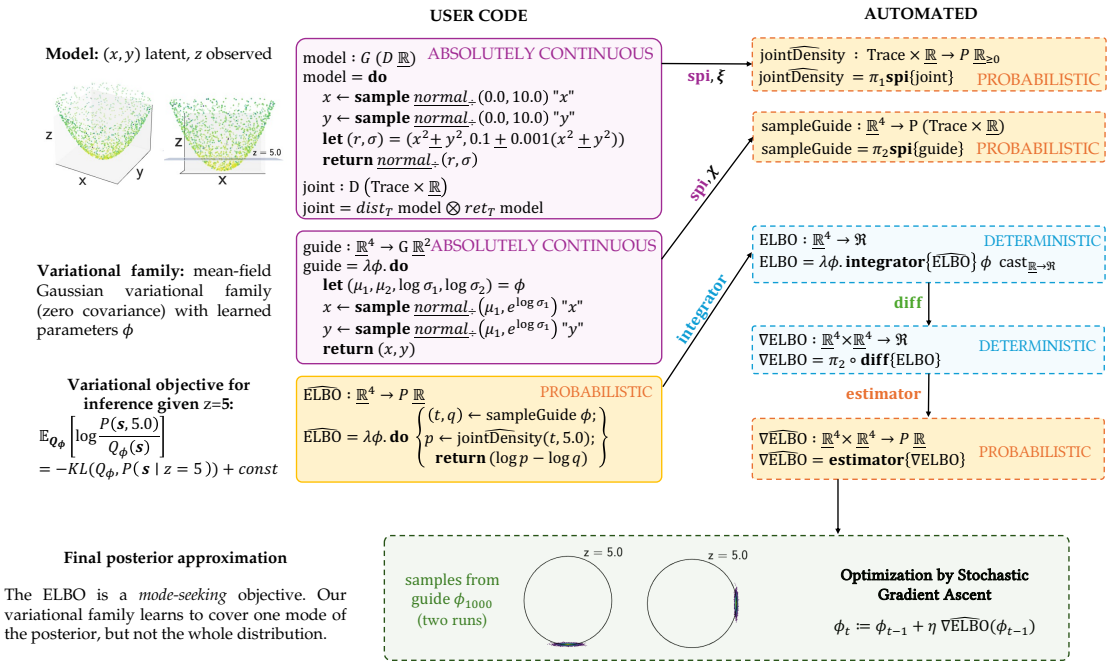


Figure 6-4: We compose multiple program transformations to automate the construction of unbiased gradient estimators for variational inference. The user begins by writing programs in the language of Chapter 4 (purple) to encode a model and variational family. These programs are compiled into procedures for density evaluation and simulation in the core probabilistic language of Chapter 2 (yellow). These automated procedures can be used with Chapter 3's integration transformation to concisely define a variational objective, which may be intractable. We can then apply the automatic differentiation algorithm of Chapter 5 to differentiate the objective, yielding an intractable gradient. Finally, we can apply Chapter 3's estimation transformation to obtain a *gradient estimator*, which unbiasedly estimates gradients of the variational objective. Solid outlines indicate user-written programs, whereas dashed outlines indicate automatically constructed programs.

stochastic gradient ascent. We construct unbiased gradient estimators for the user's objective function by composing the **diff** transformation from Chapter 5 with the **estimator** transformation from Chapter 3. Users can rapidly explore a combinatorial space of estimation strategies with compositional annotations on their generative programs, to navigate tradeoffs between the variance and the computational cost of the automated gradient estimator.

Example. To make this concrete, consider the toy problem illustrated in Fig. 6-4, which we seek to solve by training a variational approximation to the posterior. We go through the following steps:

- **Define a model**. Our model encodes a generative process for points (x, y, z) around a 3D cone. We use $\textbf{sample}_G$ to sample latents x and y with string-valued *names*. Our goal is to infer (x, y) consistent with this observation.

- **Define a variational family**. In the second panel of Fig. 6-4, we construct a *variational family*, a parametric family of possible approximations to the posterior distribution. Our variational inference task will be to learn parameters that maximize the quality of the approximation. Our $\texttt{q}_{\texttt{NAIVE}}$ is a *mean-field* variational approximation, i.e., it generates x and y independently. Note that primitive distributions (here, `normal`) are annotated with gradient estimation strategies (here, `REPARAM`) for propagating derivative information through the corresponding primitive.[3]

- **Define a variational objective**. We now use the differentiable language to define the objective function we wish to optimize. Three constructs are especially useful: (1) **density**, which computes or estimates densities of probabilistic programs; (2) **sim**, which generates a pair (x, w) of a sample and its density from a probabilistic program; and (3) $\mathbb{E}$, which takes the expected value of a stochastic procedure. We use them together to implement the ELBO objective from Eqn. 6.3.

- **Perform stochastic optimization.** Our objective is compiled into an *unbiased estimator* of its *gradient* with respect to the input parameters ϕ. We can then apply stochastic optimization algorithms, such as stochastic gradient ascent and ADAM. The bottom of Fig. 6-4 illustrates samples from the guide after training. Because the ELBO minimizes the *reverse* (or *mode-seeking*) KL divergence, our variational approximation learns to hug one edge of the circle-shaped posterior.

6.2.3. Case studies

We evaluate our approach using `genjax.vi`, a prototype of our proposed architecture implemented as an extension to a JAX-hosted version of Gen [Cusumano-Towner et al., 2019b]. All experiments were run on a single device with an AMD Ryzen 7 7800X3D @ 5.050 GHz CPU and an Nvidia RTX 4090 GPU. We consider several case studies designed to answer the following questions:

- **Overhead.** *How much overhead is incurred by using our automated gradient estimators, over hand-coded versions?* We compare the same gradient estimator for a variational autoencoder [Kingma and Welling, 2014b] constructed (a) via a hand-coded implementation and (b) via our automation.

- **Overall performance.** *How well can we solve a challenging inference problem using our system compared to other PPLs that support variational inference?* We consider the *Attend-Infer-Repeat* (AIR) model [Eslami et al., 2016] and compare the capabilities of our system to Pyro [Bingham et al., 2018].

- **Expressivity and compositional correctness.** *For the objectives and estimator*

[3] For a primitive distribution μ_θ over values of type X, parameterized by arguments $\theta \in \mathbb{R}^n$, a gradient estimation strategy for μ_θ is an approach to unbiasedly estimating $\nabla_\theta \mathbb{E}_{x\sim\mu_\theta}[f(\theta, x)]$ for functions $f : \mathbb{R}^n \times X \to \mathbb{R}$. ADEV composes these primitive estimation strategies into composite strategies for estimating gradients of variational objectives. Supported strategies vary by primitive; for the Normal distribution, for instance, they include `REPARAM`, `MEASURE-VALUED`, and `REINFORCE`, corresponding to different approaches to gradient estimation from the literature.

Table 6.2: Timing (ms) our estimators versus hand coded estimators for the VAE.

Batch size	Ours	Hand coded
64	0.11 ± 0.02	0.09 ± 0.04
128	0.22 ± 0.2	0.16 ± 0.08
256	0.31 ± 0.18	0.29 ± 0.17
512	0.56 ± 0.35	0.54 ± 0.34
1024	1.58 ± 1.13	1.07 ± 0.70

Table 6.3: Time (in seconds) to train the AIR model Eslami et al. [2016] for one epoch (batch size 64) with different objectives and estimators. All discrete variables use the same estimation strategy. IWELBO runs have $n = 2$ particles.

System	REINFORCE	ENUM	MVD	IWAE + REINFORCE	IWAE + MVD
`genjax.vi`	1.52 ± 0.05	6.22 ± 0.29	1.74 ± 0.04	2.28 ± 0.12	3.74 ± 0.05
`pyro`	12.28 ± 0.55	122.93 ± 1.74	X	22.17 ± 1.2	X

strategies expressible in our system, is it possible to combine all objectives and estimator strategies while maintaining correctness? We evaluate the expressiveness of our system vs. Pyro on the AIR model, and on a hierarchical variational inference problem [Agakov and Barber, 2004].

Overhead. Table 6.2 presents a runtime comparison between an automated gradient estimator in `genjax.vi` and a hand-coded implementation of the same gradient estimator in JAX, for a variational autoencoder. We measure the wall time required to compute a gradient estimate for different batch sizes n. We find that our automation introduces a small amount of runtime overhead (around 3-10%) compared to our hand coded implementation.

Overall Performance. We consider the *Attend, Infer, Repeat* [Eslami et al., 2016] (AIR) model. We plot accuracy and loss curves over time in Fig. 6-5, for several estimators expressed in our system and in Pyro. We also compare timing results, shown in Table 6.3. Our implementation's performance is competitive, and we support a broader class of estimators and objectives than Pyro. We find that some estimators we support (in particular those based on measure-valued derivatives) lead to faster convergence than the estimators automated by Pyro.

Expressivity and Compositional Correctness. In Table 6.4, we enumerate several possible combinations of gradient estimation strategies and objectives for the AIR model. In Table 6.5, we consider a simple instance of hierarchical variational inference [Ranganath et al., 2016]. We implement a simple, non-hierarchical variational family in all three systems (`genjax.vi`, Pyro and NumPyro), and we exploit our composability with Chapter 4's density estimation to implement a hierarchical (i.e., auxiliary-variable) variational family in `genjax.vi`. We show statistics on final mean objective values for different variational objectives.

Table 6.4: Combinatorial space of gradient estimators and objective functions for the AIR model, which our programmable approach helps to explore.

Gradient Estimation Strategies				**Objective**	**Batch**	**System**	
REINFORCE	ENUM	MVD	BASELINE			Pyro	Ours
✓				ELBO	≥ 1	✓	✓
				IWAE	≥ 1	✓	✓
	✓			ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
		✓		ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
			✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓			ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
✓		✓		ELBO	≥ 1	✓	✓
				IWAE	≥ 1	×	✓
✓			✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓	✓		ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓		✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
		✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓	✓		ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓	✓		✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
✓		✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
	✓	✓	✓	ELBO	≥ 1	×	✓
				IWAE	≥ 1	×	✓
Not Exploited				RWS	1	✓	✓
					> 1	×	✓

Table 6.5: Mean objective value (in nats) on repeated runs for several variational objectives, including ones which utilize **marginal**. n and m denote particle sizes for SIR algorithms.

System	ELBO	IWELBO ($n = 5$)	HVI	IWHVI ($m = 5$)	DIWHVI ($n = 5, m = 5$)
`genjax.vi`	-8.08	-7.79	-9.75	-8.18	-7.33
`numpyro`	-8.08	-7.77	✓/X	X	X
`pyro`	-8.08	-7.75	✓/X	X	X

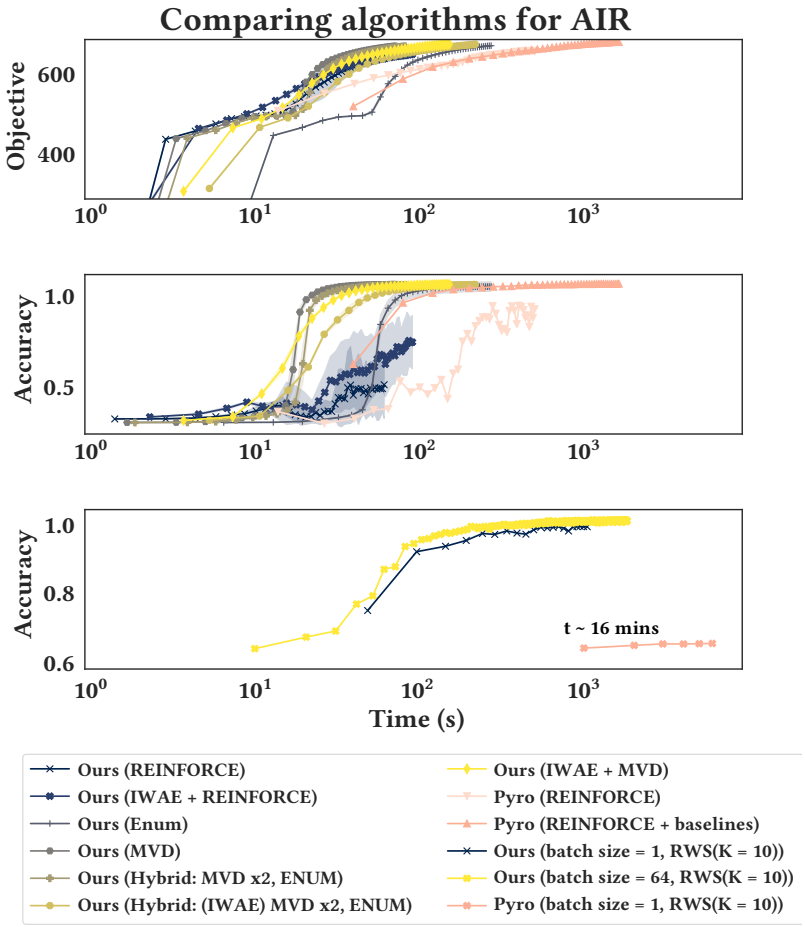


Figure 6-5: We evaluate a variety of custom estimators and objectives (ELBO, IWAE, RWS) using our system. Our on average best estimator (IWAE + MVD, not expressible in Pyro) converges an order of magnitude faster than Pyro's recommended estimator.

7

SCALABLE AUTOMATED REASONING ABOUT RELATIONAL DATA

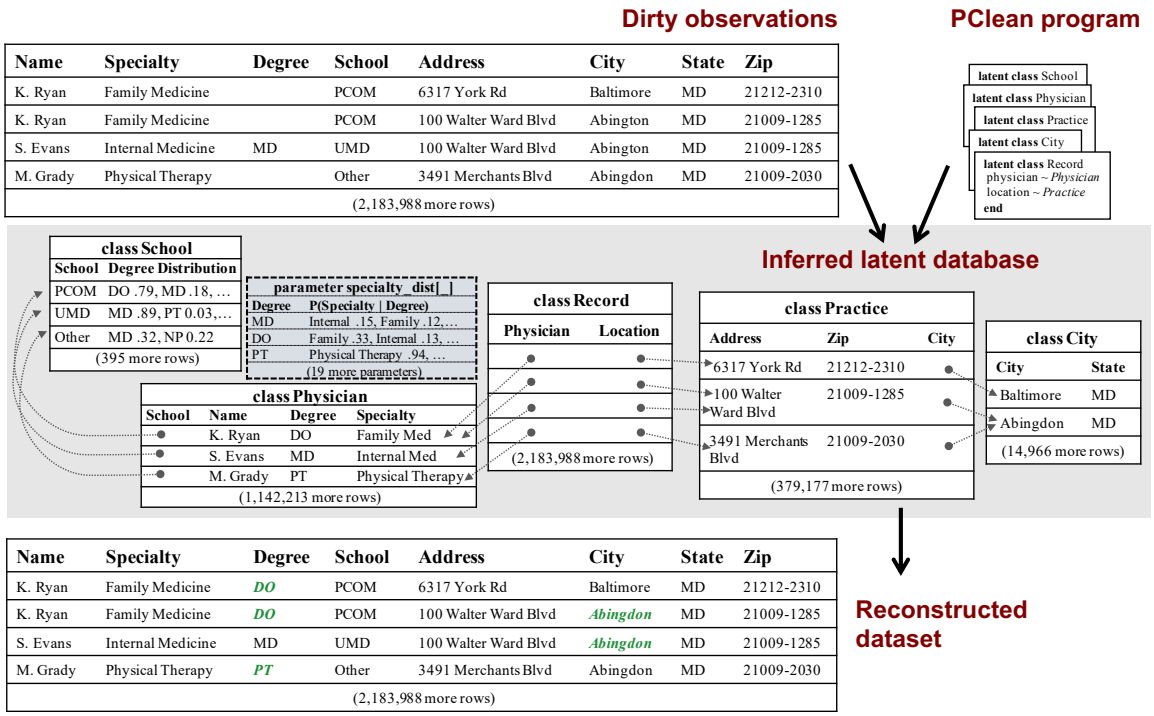
The work in this chapter was originally published in Lew et al. [2021a]. It is joint work with Monica Agrawal, David Sontag, and Vikash Mansinghka.

In this chapter, we describe the design and implementation of PClean, a *domain-specific* probabilistic programming system, specialized for modeling tabular data about interrelated entities in the world. Given a (possibly dirty) dataset and a program in the language, PClean infers a clean latent database of interrelated objects that might underlie the observed table. This is accomplished via an automated implementation of a sequential Monte Carlo (SMC) algorithm with Markov chain Monte Carlo (MCMC) rejuvenation, with a specially constructed sequence of target and proposal distributions.

Running this algorithm depends crucially on the ability to compute *incremental weights* for sequential Monte Carlo and *acceptance probabilities* for MCMC. These quantities are **Radon-Nikodym derivatives**, and Chapter 4 gives general recipes for computing them automatically; Theorem 6 establishes the soundness of using them within SMC and MCMC. Our implementation of PClean specializes these techniques to the restricted class of model and proposal distributions that arise when working in the PClean DSL.

7.1 Motivation: Bayesian reasoning over real-world relational data

Real-world data is often noisy and incomplete, littered with NULLs, typos, duplicates, and inconsistencies. Cleaning dirty data is important for many workflows, but can be difficult to automate, requiring judgment calls about objects in the world (e.g., to decide whether two records refer to the same hospital, or which of several cities called "Jefferson" someone lives in). Generative models provide



Name	Specialty	Degree	School	Address	City	State	Zip
K. Ryan	Family Medicine		PCOM	6317 York Rd	Baltimore	MD	21212-2310
K. Ryan	Family Medicine		PCOM	100 Walter Ward Blvd	Abington	MD	21009-1285
S. Evans	Internal Medicine	MD	UMD	100 Walter Ward Blvd	Abington	MD	21009-1285
M. Grady	Physical Therapy		Other	3491 Merchants Blvd	Abingdon	MD	21009-2030
(2,183,988 more rows)							

Name	Specialty	Degree	School	Address	City	State	Zip
K. Ryan	Family Medicine	*DO*	PCOM	6317 York Rd	Baltimore	MD	21212-2310
K. Ryan	Family Medicine	*DO*	PCOM	100 Walter Ward Blvd	*Abingdon*	MD	21009-1285
S. Evans	Internal Medicine	MD	UMD	100 Walter Ward Blvd	*Abingdon*	MD	21009-1285
M. Grady	Physical Therapy	*PT*	Other	3491 Merchants Blvd	Abingdon	MD	21009-2030
(2,183,988 more rows)							

Figure 7-1: PClean applied to Medicare's 2.2-million-row Physician Compare National database. Based on a user-specified relational model, PClean infers a latent database of entities, which it uses to correct systematic errors (e.g. the misspelled *Abington, MD* appears 152 times in the dataset) and impute missing values.

a conceptually appealing approach to automating this sort of reasoning, but the diversity of real-world errors [Abedjan et al., 2016] and the difficulty of inference pose significant challenges.

This chapter presents PClean, a domain-specific generative probabilistic programming language (PPL) for Bayesian data cleaning. As in some existing PPLs (e.g. BLOG [Milch and Russell, 2006]), PClean programs encode prior knowledge about relational domains, and quantify uncertainty about the latent networks of objects that underlie observed data. However, PClean's approach is inspired by *domain-specific* PPLs, such as Stan [Carpenter et al., 2017b] and Picture [Kulkarni et al., 2015]: it aims not to serve all conceivable relational modeling needs, but rather to deliver fast inference, concise model specification, and accurate cleaning on large-scale problems. It does this via three modeling and inference contributions:

1. PClean introduces a domain-general non-parametric prior on the number of latent objects and their link structure. PClean programs customize the prior via a relational schema and via generative models for objects' attributes.

2. PClean inference is based on a novel sequential Monte Carlo (SMC) algorithm, to initialize a latent object database with plausible guesses, and novel rejuvenation

updates to fix mistakes.

3. PClean provides a compiler that generates near-optimal SMC proposals and Gibbs rejuvenation kernels given the user's data, PClean program, and inference hints. These proposals improve over generic top-down PPL inference by incorporating local Bayesian reasoning within user-specified subproblems, and heuristics from traditional cleaning systems.

Together, these innovations improve over generic PPL inference, enabling fast and accurate cleaning of challenging real-world datasets with millions of rows.

7.2 Modeling with domain-specific probabilistic programs

In this section, we present the PClean modeling language, which is designed for the concise encoding of domain-specific knowledge about data and likely errors into generative models. PClean programs specify (i) a prior $p(\mathbf{R})$ over a latent ground-truth relational database of entities, and (ii) an observation model $p(\mathbf{D} \mid \mathbf{R})$ describing how the attributes of entities from $\mathbf{R}$ are reflected in an observed flat data table $\mathbf{D}$. Unlike general-purpose PPLs, PClean does not afford complete freedom in specifying $p(\mathbf{R})$. Instead, we impose a novel domain-general structure prior $p(\mathbf{S})$ on the *skeleton* of the database $\mathbf{R}$: $\mathbf{S}$ determines how many entities are in each latent database table, and which entities are related. The user's program encodes only $p(\mathbf{R} \mid \mathbf{S})$, a probabilistic relational model over the attributes of the objects whose existence and relationships are given by $\mathbf{S}$. This decomposition limits the PClean model class, but enables the development of an efficient SMC inference algorithm.

7.2.1. PClean modeling language

A PClean program defines a set of *classes* $C = (C_1, \ldots, C_k)$, one for each type of object underlying the user's data, and a *query* $\mathbf{Q}$ that describes how latent objects inform the observed flat dataset $\mathbf{D}$.

Class declarations. The declaration of a PClean class C includes three kinds of statement: *reference statements* ($Y \sim C'$), which define a reference slot $C.Y$ connecting objects of class C to objects of a target class $T(C.Y) = C'$; *attribute statements* ($X \sim \phi_{C.X}(\ldots)$), which define a new *attribute* $C.X$ that objects of the class possess, and declare the prior distribution $\phi_{C.X}$ that it follows; and *parameter statements* (**parameter** $\theta_C \sim p_{\theta_C}(\ldots)$), which introduce mutually independent hyperparameters shared by all objects of the class C, to be learned from the noisy data. The prior $\phi_{C.X}$ for an attribute may depend on the values of a *parent set* $Pa(C.X)$ of attributes, potentially accessed via reference slots. For example, in Figure 7-2, the *Physician* class has a *school* reference slot with target class **School**, and a *degree* attribute whose value depends on *school.degree_dist*. Together, the attribute statements specify a *probabilistic relational model* Π for the user's schema (possibly parameterized by hyperparameters $\{\theta_C\}_{C \in \mathcal{C}}$) [Friedman et al., 1999].

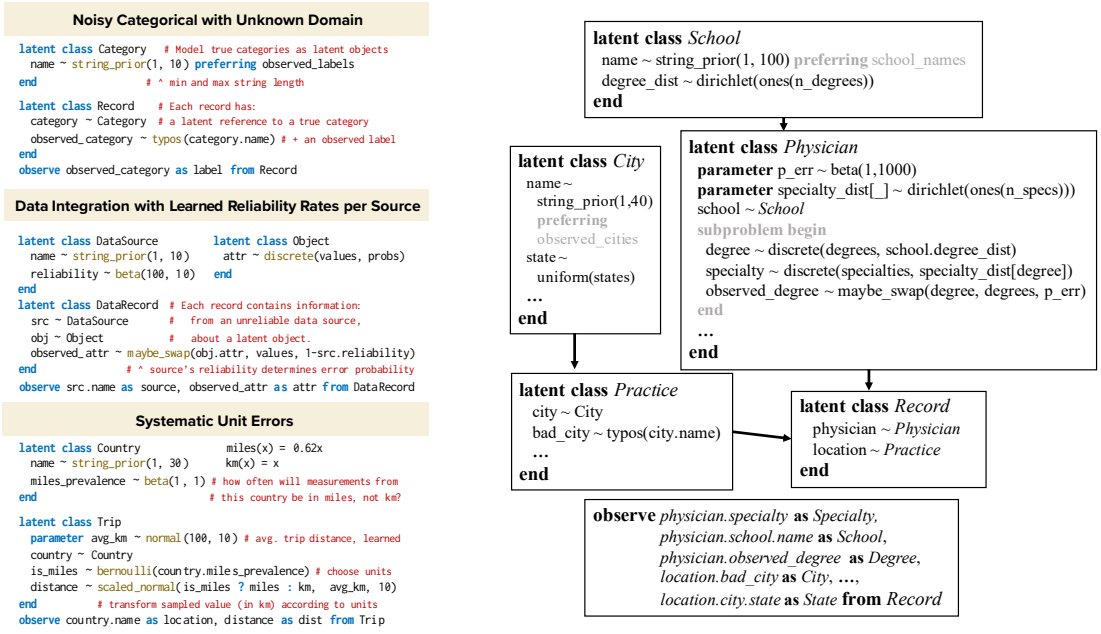


Figure 7-2: **Left:** PClean programs can model a variety of data cleaning scenarios. All of these patterns can be used individually or combined in a single script, depending on the user's dataset. **Right:** An example PClean program, for the dataset depicted in Fig. 7-1. PClean programs define: (i) an *acyclic* relational schema, comprising a set of classes C, and for each class C, sets $\mathcal{A}(C)$ of attributes and $\mathcal{R}(C)$ of reference slots; (ii) a probabilistic relational model Π encoding priors for object attributes; and (iii) a query **Q** (last line), specifying how attributes are observed in the flat data table **D**. *Inference hints* (in gray) do not change the model.

Query. A PClean program ends with a *query*, connecting the schema of the latent relational database to the fields of the observed dataset. The query has the form **observe** $(U_1$ **as** $x_1), \cdots, (U_k$ **as** $x_k)$ **from** C_{obs}, where C_{obs} is a class that models the records of the observed dataset (*Record*, in Figure 7-2), x_i are the names of the columns in the observed dataset, and U_i are dot-expressions (e.g., *physician.school.name*), picking out attributes accessible via zero or more reference slots from C_{obs}. The records of the dataset **D** are modeled as directly recording the values of the attributes named by U_i, each for a distinct object in C_{obs}. As such, errors must be modeled as *part* of the latent database **R**, not as a separate stage of the generative process. For example, the program on the right in Figure 7-2 models systematic typos in the *City* field, by associating each *Practice* with a possibly misspelled version *bad_city* of the practice's city.

We emphasize that the relational schema and query are modeling choices: much of PClean's expressive power comes from the freedom to posit latent relational structure that is not directly reflected in the dataset. The left panel of Figure 7-2 shows how this freedom can be used to capture several common data-cleaning motifs.

Algorithm 5 Non-parametric structure prior $p(\mathbf{S})$ over the relational skeleton $\mathbf{S}$ for a schema C.

GenerateSkeleton($C, |\mathbf{D}|$):
  ▷ Create one C_{obs} object per observed record
  $\mathbf{S}_{C_{obs}} := \{1, \ldots, |\mathbf{D}|\}$
  ▷ Generate a class *after* all referring classes:
  for class $C \in$ TopoSort($C \setminus \{C_{obs}\}$) **do**
    ▷ Collect references to class C
    $\mathbf{Ref}_{\mathbf{S}}^{C} := \{(r, Y) \mid r \in \mathbf{S}_{C'}, T(C'.Y) = C\}$
    ▷ Generate targets of those references
    $\mathbf{S}_C \sim$ GenerateObjectSet($C, \mathbf{Ref}_{\mathbf{S}}^{C}$)
    ▷ Assign reference slots pointing to C
    for object $r' \in \mathbf{S}_C$ **do**
      for referring object $(r, Y) \in r'$ **do**
        $r.Y := r'$
  ▷ Return the skeleton
  return $\{\mathbf{S}_C\}_{C \in C}, (r, Y) \mapsto r.Y$

GenerateObjectSet($C, \mathbf{Ref}_{\mathbf{S}}^{C}$):
  $s_C \sim Gamma(1, 1)$; $d_C \sim Beta(1, 1)$
  ▷ Partition references into co-referring subsets
  $\mathbf{S}_C \sim CRP(\mathbf{Ref}_{\mathbf{S}}^{C}, s_C, d_C)$
  return $\mathbf{S}_C$

7.2.2. Non-parametric structure prior $p(\mathbf{S})$

A PClean program's class declarations specify a probabilistic relational model that can be used to generate the attributes of objects in the latent database, but does not encode a prior over how many objects exist in each class or over their relationships. (The one exception is C_{obs}, the designated observation class, whose objects are assumed to be in one-to-one correspondence with the rows of the observed dataset $\mathbf{D}$.) In this section, we introduce a domain-general structure prior $p(\mathbf{S}; |\mathbf{D}|, C)$ that encodes a non-parametric generative process over the *object sets* $\mathbf{S}_C$ associated with each class C, and over the values of each object's reference slots. The parameter $|\mathbf{D}|$ is the number of observed data records; $p(\mathbf{S}; |\mathbf{D}|, C)$ places mass only on relational skeletons in which there are exactly $|\mathbf{D}|$ objects in C_{obs} and every other object is connected via some chain of reference slots to one of them.

PClean's generative process for relational skeletons is shown in Figure 5. First, with probability 1, we set $\mathbf{S}_{C_{obs}} = \{1, \ldots, |\mathbf{D}|\}$ (a set of $|\mathbf{D}|$ distinct object IDs). PClean requires that the directed graph with edges $(C, T(C.Y))$ for each reference slot $C.Y$ be acyclic, which allows us to generate the remaining object sets class-by-class, processing a class only *after* any classes with reference slots targeting it. To generate an object set for class C, we first consider the reference set $\mathbf{Ref}_{\mathbf{S}}^{C}$ of all objects with

reference slots targeting C:

$$\mathbf{Ref}_{\mathbf{S}}^{C} = \{(r, Y) \mid Y \in \mathcal{R}(C') \wedge T(C'.Y) = C \wedge r \in \mathbf{S}_{C'}\}$$

The elements of $\mathbf{Ref}_{\mathbf{S}}^{C}$ are pairs (r, Y) of an object and a reference slot; if a single object has two reference slots targeting class C, then the object will appear twice in the reference set. The point is to capture all of the places in $\mathbf{S}$ that will refer to objects of class C.

We then generate a *co-reference partition* of $\mathbf{Ref}_{\mathbf{S}}^{C}$, i.e., we partition the references to class C into disjoint subsets, within each of which we take all references to target the same object. To do this, we use the two-parameter Chinese restaurant process $CRP(X, s, d)$, which defines a non-parametric distribution over partitions of its set-valued parameter X. The strength s and discount d control the sizes of the clusters. The CRP generates a partition of all references to class C, and *we treat the resulting partition as the object set* $\mathbf{S}_C$, i.e., each component defines one object of class C:

$$\mathbf{S}_C \mid \mathbf{Ref}_{\mathbf{S}}^{C} \sim CRP(\mathbf{Ref}_{\mathbf{S}}^{C}, s_C, d_C)$$

To set the reference slots $r.Y$ with target class $T(\mathbf{Class}(r).Y) = C$, we simply look up which partition component (r, Y) (viewed as an element of $\mathbf{Ref}_{\mathbf{S}}^{C}$) was assigned to. Since we have equated these partition components with objects of class C, we can directly set $r.Y$ to point to the component (object) that contains (r, Y) as an element:

$$r.Y := \text{the unique } r' \in \mathbf{S}_{T(\mathbf{Class}(r).Y)} \text{ s.t. } (r, Y) \in r'$$

This procedure can be applied iteratively to generate object sets for every class and fill all reference slots.

7.3 Inference with sequential Monte Carlo via automated Radon-Nikodym derivatives

PClean's non-parametric structure prior ensures that PClean models admit a sequential representation, which can be used as the basis of a resample-move sequential Monte Carlo inference scheme (Section 7.3.1). However, if the SMC and rejuvenation proposals are made from the model prior, as is typical in PPLs, inference will still require prohibitively many particles to deliver accurate results. To address this issue, PClean uses a *proposal compiler* that exploits conditional independence in the model to generate fast enumeration-based proposal kernels for both SMC and MCMC rejuvenation (Section 7.3.2). Finally, to help users scale these proposals to large data, we introduce *inference hints*, lightweight annotations in the PClean program that can divide variables into subproblems to be separately handled by the proposal, or direct the enumerator to focus its efforts on a dynamically computed subset of a large discrete domain (Section 7.3.3).

7.3.1. Per-observation sequential Monte Carlo with per-object rejuvenation

One version of the PClean model's generative process was given in Section 7.2: a skeleton can be generated from $p(\mathbf{S})$, then attributes can be filled in using the user's probabilistic relational model $p_\Pi(\mathbf{R} \mid \mathbf{S})$. Finally an observed dataset $\mathbf{D}$ can be generated according to the query $\mathbf{Q}$. But importantly, the model also admits a sequential representation, in which the latent database $\mathbf{R}$ is built in stages: at each stage, a single record is added to the observation class C_{obs}, along with any new objects in other classes that it refers to. Using this representation, we can run SMC on the model, building a particle approximation to the posterior that incorporates one observation at a time.

Database increments. Let $\mathbf{R}$ be a database with designated observation class C_{obs}. Assume $\mathbf{R}_{C_{obs}}$, the object set for the class C_{obs}, is $\{1, \ldots, |\mathbf{D}|\}$. Then the database's i^{th} *increment* $\Delta^i_{\mathbf{R}}$ is the object set

$$\{r \in \mathbf{R} \mid \exists K,\, i.K = r \wedge \forall K', \forall j < i,\, j.K' \neq r\},$$

along with their attribute values and targets of their reference slots. Objects in $\Delta^i_{\mathbf{R}}$ may refer to other objects within the increment, or in earlier increments. Intuitively, the i^{th} increment of a database is the set of objects referenced by the i^{th} observation object, but *not* by any previous observation object $j < i$.

Sequential generative process. Figure 6 shows a generative process equivalent to the one in Section 7.2, but which generates the attributes and reference slots of each increment sequentially. Intuitively, the database is generated via a Chinese-restaurant 'social network': Consider a collection of restaurants, one for each class C, where each table serves a dish r representing an object of class C. Upon entering a restaurant, customers either sit at an existing table or start a new one, as in the usual generalized CRP construction. But these restaurants require that to start a new table, customers must first send $|\mathcal{R}(C)|$ friends to *other* restaurants (one to the target of each reference slot). Once the friends are seated at these *parent* restaurants, the original customer samples attributes $r.X$ of the new table's object, possibly informed by *their friends'* dishes (the objects $r.Y$ of class $T(C.Y)$). The process starts with $|\mathbf{D}|$ customers at the restaurant for C_{Obs}, who sit at separate tables; each customer who sits down triggers the sampling of one increment.

SMC inference with per-object rejuvenation. The sequential representation yields a sequence of intermediate unnormalized target densities $\tilde{\pi}_i$ for SMC:

$$\tilde{\pi}_i(\mathbf{R}) = \prod_{j=1}^{i} p(\Delta^{\mathbf{R}}_j \mid \Delta^{\mathbf{R}}_1, \ldots, \Delta^{\mathbf{R}}_{j-1}) p(d_j \mid \Delta^{\mathbf{R}}_1, \ldots, \Delta^{\mathbf{R}}_j).$$

Particles are initialized to hold an empty database, to which proposed increments $\Delta^{\mathbf{R}}_i$ are added each iteration. As is typical in SMC, at each step, the particles are reweighted according to how well they explain the new observed data, and

Algorithm 6 Sequential model representation.

GenerateDataset(Π, $\mathbf{Q}$, $|\mathbf{D}|$):
  $\mathbf{R}^{(0)} \leftarrow \emptyset$ ▷ Initialize empty database
  for observation $i \in \{1, \ldots, |\mathbf{D}|\}$ **do**
    $\Delta_i^{\mathbf{R}} \leftarrow$ GenerateDbIncr($\mathbf{R}^{(i-1)}, C_{obs}$)
    $\mathbf{R}^{(i)} \leftarrow \mathbf{R}^{(i-1)} \cup \Delta_i^{\mathbf{R}}$
    $r \leftarrow$ the unique object of class C_{obs} in $\Delta_i^{\mathbf{R}}$
    $d_i \leftarrow \{X \mapsto r.\mathbf{Q}(X),\ \forall X \in \mathcal{A}(\mathbf{D})\}$
  return $\mathbf{R} = \mathbf{R}^{(|\mathbf{D}|)}, \mathbf{D} = (d_1, \ldots, d_{|\mathbf{D}|})$

GenerateDbIncr($\mathbf{R}^{(i-1)}$, root class C):
  $\Delta \leftarrow \emptyset$; $r_* \leftarrow$ a new object of class C
  for each reference slot $Y \in \mathcal{R}(C)$ **do**
    $C' \leftarrow T(C.Y)$
    for each object $r \in \mathbf{R}_{C'}^{(i-1)} \cup \Delta_{\mathbf{R}_{C'}}$ **do**
      $n_r \leftarrow |\{r' \mid r' \in \mathbf{R}^{(i-1)} \cup \Delta \wedge \exists \tau, r'.\tau = r\}|$
    $r_*.Y \leftarrow r$ w.p. $\propto n_r - d_{C'}$, or
      $\star$ w.p. $\propto s_{C'} + d_{C'} |\mathbf{R}_{C'}^{(i-1)} \cup \Delta_{\mathbf{R}_{C'}}|$
    if $r_*.Y = \star$ **then**
      $\Delta' \leftarrow$ GenerateDbIncr($\mathbf{R}^{(i-1)} \cup \Delta, C'$)
      $\Delta \leftarrow \Delta \cup \Delta'$
      $r_*.Y \leftarrow$ the unique r' of class C' in Δ'
  for each $X \in \mathcal{A}(C)$, in topological order **do**
    $r_*.X \sim \phi_{C.X}(\cdot \mid \{r_*.U\}_{U \in Pa(C.X)})$
  return $\Delta \cup \{r_*\}$

resampled to cull low-weight particles while cloning and propagating promising ones. Formally, our sequential Monte Carlo algorithm is an instance of Algorithm 3 from Section 4.2, and we compute the weight updates using automated Radon-Nikodym derivatives (Chapter 4). This process allows the algorithm to hypothesize new latent objects as needed to explain each new observation, but not to revise earlier inferences about latent objects (or delete previously hypothesized objects) in light of new observations; we address this problem with MCMC rejuvenation moves. These moves select an object r, and update all r's attributes and reference slots in light of all relevant data incorporated so far. In doing so, these moves may also lead to the "garbage collection" of objects that are no longer connected to the observed dataset, or to the insertion of new objects as targets of r's reference slots.

7.3.2. Compiling data-driven SMC proposals

Proposal quality is the determining factor for the quality of SMC inference: at each step of the algorithm, a proposal $Q_i(\Delta_i^{\mathbf{R}}; \mathbf{R}^{(i-1)}, d_i)$ generates proposed additions $\Delta_i^{\mathbf{R}}$ to the existing latent database $\mathbf{R}^{(i-1)}$ to explain the i^{th} observed data point, d_i. A key

Algorithm 7 Compiling SMC proposal to Bayesian network

procedure GENERATEINCREMENTBAYESNET(partial instance $\mathbf{R}^{(i-1)}$, data d_i)

▷ Set the vertices to all attributes and reference slots accessible from C_{obs}

$U \leftarrow \mathcal{A}(C_{obs}) \cup \{K \mid C_{obs}.K \text{ is a valid slot chain}\} \cup \{K.X \mid X \in \mathcal{A}(T(C_{obs}.K))\}$

▷ Determine parent sets and CPDs for each variable

for each variable $u \in U$ **do**

if $u \in \mathcal{A}(C_{obs})$ **then**

Set $Pa(u) = Pa^{\Pi}(C.u)$

Set $\phi_u(v_u \mid \{v_{u'}\}_{u' \in Pa(u)}) = \phi^{\Pi}_{C.u}(v_u \mid \{v_{u'}\}_{u' \in Pa(u)})$

else if $u = K.X$ for $X \in \mathcal{A}(T(C_{obs}.K))$ **then**

Set $Pa(u) = Pa^{\Pi}(T(C_{obs}.K).X) \cup \{K\} \cup \{u'.X \mid u' \text{ already processed} \wedge T(C_{obs}.u') = T(C_{obs}.K)\}$

Set

$$\phi_u(v_u \mid \{v_{u'}\}_{u' \in Pa(u)}) = \begin{cases} \mathbf{1}[v_u = v_K.X] & v_K \in \mathbf{R}^{(i-1)} \\ \phi^{\Pi}_{T(C_{obs}.K).X}(v_u \mid \{v_{u'}\}_{u' \in Pa^{\Pi}(T(C_{obs}.K).X)}) & v_K = \mathbf{new}_K \\ \mathbf{1}[v_u = v_{u'.X}] & v_K = \mathbf{new}_{u'}, u' \neq K \end{cases}$$

else

Set $Pa(u)$ to already-processed slot chains u' s.t. $T(C.u') = T(C.u)$, and K if $u = K.Y$

Set domain $V(u) = \mathbf{R}^{(i-1)}_{T(C.u)} \cup \{\mathbf{new}_{u'} \mid u' \in Pa(u) \cup \{u\}\}$

Set $\phi_u(v_u \mid \{v_{u'}\}_{u' \in Pa(u)})$ according to CRP, or to $\mathbf{1}[v_u = v_K.Y]$ if $u = K.Y$ and $v_K \in \mathbf{R}^{(i-1)}$

for attribute $X \in \mathcal{A}(\mathbf{D})$ **do**

Change node $\mathbf{Q}(u)$ to be observed with value $d_i.x$, **unless** $d_i.x$ is missing

limitation of the sequential Monte Carlo implementations in most general-purpose PPLs today is that the proposals Q_i are not *data-driven*, but rather based only on the prior: they make blind guesses as to the latent variable values and thus tend to make proposals that explain the data poorly. By contrast, PClean compiles proposals that use exact enumerative inference to propose discrete variables in a data-driven way. This approach extends ideas from Arora et al. [2012] to the block Gibbs rejuvenation and block SMC setting, with user-specified blocking hints. These proposals are *locally optimal* for models that contain only discrete finite-domain variables, meaning that of all possible proposals Q_i they minimize the divergence

$$KL(\pi_{i-1}(\mathbf{R}^{(i-1)})Q_i(\Delta_i^{\mathbf{R}}; \mathbf{R}^{(i-1)}, d_i) \| \pi_i(\mathbf{R}^{(i-1)} \cup \Delta_i^{\mathbf{R}})).$$

The distribution on the left represents a perfect sample $\mathbf{R}^{(i-1)}$ from the target given the first $i-1$ observations, extended with the proposal Q_i. The distribution on the right is the target given the first i data points. In our setting the locally optimal

proposal is given by

$$Q_i(\Delta_i^{\mathbf{R}}; \mathbf{R}^{(i-1)}, d_i) \propto \\ p(\Delta_i^{\mathbf{R}} \mid \Delta_1^{\mathbf{R}}, \ldots, \Delta_{i-1}^{\mathbf{R}}) p(d_i \mid \Delta_1^{\mathbf{R}}, \ldots, \Delta_i^{\mathbf{R}}).$$

Algorithm 7 shows how to compile this distribution to a Bayesian network; when the latent attributes have finite domains, the normalizing constant can be computed and the locally optimal proposal can be simulated (and evaluated) exactly. This is possible because there are only a finite number of instantiations of the random increment $\Delta_i^{\mathbf{R}}$ to consider. The compiler generates efficient enumeration code separately for each pattern of missing values it encounters in the dataset, exploiting conditional independence relationships in each Bayes net to yield potentially exponential savings over naive enumeration. A similar strategy can be used to compile data-driven object-wise rejuvenation proposals, and to handle some continuous variables with conjugate priors. Once these proposals have been compiled as probabilistic programs, we can compute Radon-Nikodym derivatives

$$w = \frac{d\tilde{\pi}_i}{d(\tilde{\pi}_{i-1} \otimes Q_i)}(\mathbf{R}^{(i-1)} \cup \Delta_i^{\mathbf{R}})$$

of the target measure with respect to them, which are used as importance weights in SMC. A slightly different weight is similarly used to inform accept/reject decisions in MCMC rejuvenation.

7.3.3. Scaling to large models and data with inference hints

Scaling to models with large-domain variables and to datasets with many rows is a key challenge. In PClean, users can specify lightweight *inference hints* to the proposal compiler, shown in gray in Figure 7-2, to speed up inference without changing model's meaning.

Programmable subproblems. First, users may group attribute and reference statements into blocks by wrapping them in the syntax **subproblem begin** ... **end**. This partitions the attributes and reference slots of a class into an ordered list of *subproblems*, which SMC uses as intermediate target distributions. This makes enumerative proposals faster to compute, at the cost of considering less information at each step; rejuvenation moves can often compensate for short-sighted proposals.

Adaptive mixture proposals with dynamic preferred values. A random variable within a model may be intractable to enumerate. For example, `string_prior(1, 100)` is a distribution over all strings between 1 and 100 letters long. To handle these, PClean programs may declare *preferred values hints*. Instead of $X \sim d(E, \ldots, E)$, the user can write $X \sim d(E, \ldots, E)$ **preferring** E, where the final expression gives a list of values ξ_X on which the posterior mass is expected to concentrate. When enumerating, PClean replaces the CPD ϕ_X with a surrogate $\widehat{\phi_X}$, which is equal to ϕ_X for preferred value inputs in ξ_X, but 0 for all other values. The mass not captured by

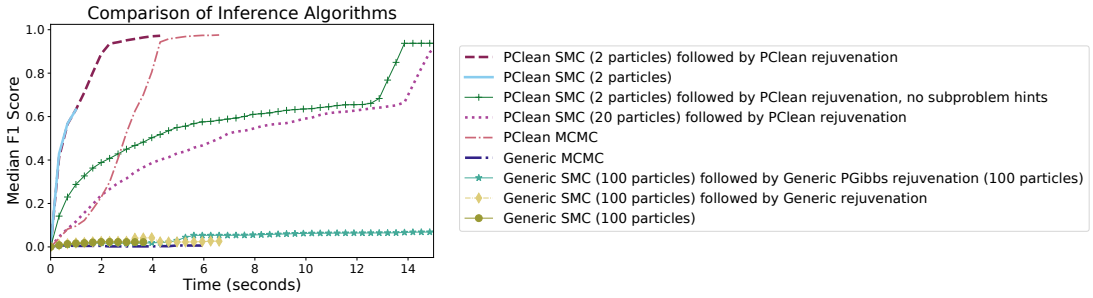


Figure 7-3: Median accuracy vs. runtime for five runs of alternative inference algorithms on the *Hospital* dataset [Chu et al., 2013], with an additional 20% of cells artificially deleted so as to test both repair and imputation.

the preferred values, $1 - \sum_{x \in \xi_X} \phi_X(x)$, is assigned to a special **other** token. Enumeration yields a partial proposal $\widehat{Q}$ over a modified domain; the full proposal Q first draws from $\widehat{Q}$ then replaces **other** tokens with samples from the appropriate CPDs $\phi_X(\cdot \mid Pa(X))$. This yields a mixture proposal between the enumerative posterior on preferred values and the prior: when none of the preferred values explain the data well, **other** will dominate, causing the attribute to be sampled from its prior. But if any of the preferred values are promising, they will almost certainly be proposed.

7.4 Empirical evlauation

In this section, we demonstrate empirically that (1) PClean's inference works when standard PPL inference strategies fail, (2) short PClean programs suffice to compete with existing data cleaning systems in both runtime and accuracy, (3) PClean can scale to large real-world datasets, and (4) PClean's inference can deliver calibrated and useful estimates of uncertainty. In Experiments (1)-(3), we evaluate PClean's accuracy using a single posterior sample (the last iterate of PClean's final MCMC rejuvenation sweep); in Experiment (4), we consider an uncertainty-aware, multi-sample estimator of the clean dataset, which exploits our Bayesian framework for higher-precision repairs. Experiments were run on a laptop with a 2.6 GHz CPU and 32 GB of RAM.

(1) Comparison to generic PPL inference. We evaluate PClean's inference against standard PPL inference algorithms reimplemented to work on PClean models, on a popular benchmark from the data cleaning literature (Figure 7-3). We do not compare directly to other PPLs' implementations, because many (e.g. BLOG) cannot represent PClean's non-parametric prior. Some languages (e.g. Turing) have explicit support for non-parametric distributions, but could not express PClean's recursive use of CRPs. Others could in principle express PClean's model, but would complicate an algorithm comparison in other ways: Venture's dynamic dependency tracking is thousands of times slower than SOTA; Pyro's focus is on variational inference, hard to apply in PClean models; and Gen supports non-parametrics only via the use of mutation in its slower dynamic modeling language (making

Task	Metric	PClean	HoloClean (Unpublished)	HoloClean	NADEEF	NADEEF + Manual Java Heuristics
Flights	F_1	**0.90**	0.64	0.41	0.07	**0.90**
	Time	**3.1s**	45.4s	32.6s	9.1s	14.5s
Hospital	F_1	**0.91**	0.90	0.83	0.84	0.84
	Time	**4.5s**	1m 10s	1m 32s	27.6s	22.8s
Rents	F_1	**0.69**	0.48	0.48	0	0.51
	Time	1m 20s	20m 16s	13m 43s	13s	**7.2s**

Table 7.1: Results of PClean and various baseline systems on three diverse cleaning tasks.

SMC $O(N^2)$) or via low-level extensions that would amount to reimplementing PClean using Gen's abstractions. Nonetheless, the algorithms in Figure 7-3 are inspired by the generic automated inference provided in many PPLs, which use top-down proposals from the prior for SMC, MH [Goodman and Stuhlmüller, 2014, Ritchie et al., 2016], and PGibbs [Wood et al., 2014, Murray, 2015, Mansinghka et al., 2014b]. Our results show that PClean suffices for fast, accurate inference where generic techniques fail, and also demonstrate why inference hints are necessary for scalability: without subproblem hints, PClean takes much longer to converge, even though it eventually arrives at a similar F_1 value.

(2) Applicability to data cleaning. To check PClean's modeling and inference capabilities are good for data cleaning *in absolute terms* (rather than relative to generic PPL inference), we contextualize PClean's accuracy and runtime against two SOTA data-cleaning systems on three benchmarks with known ground truth (Table 7.1). The datasets are *Hospital*, a standard benchmark with artificial typos in 5% of cells that can be corrected by leveraging duplication of entities across rows; *Flights*, a standard benchmark integrating flight information (e.g. arrival, departure times) from conflicting real-world data sources; and *Rents*, a synthetic dataset based on census statistics, featuring continuous and discrete values, misspelled county names, missing apartment sizes, and unit errors. The baseline systems are *HoloClean* [Rekatsinas et al., 2017], based on probabilistic machine learning, and *NADEEF*, which uses a MAX-SAT solver to minimize violations of user-defined cleaning rules [Dallachiesat et al., 2013]. For HoloClean, we consider both the original code and the authors' latest (unpublished) version on GitHub; for NADEEF, we include results both using NADEEF's streamlined rule-definition interface and with custom, handwritten Java rules (more expressive but also more cumbersome).

Table 7.1 reports F_1 scores and cleaning speed. We do not aim to anoint a single 'best cleaning system,' since optimality depends on the available domain knowledge and the user's desired level of customization. Further, while we followed system authors' per-dataset recommendations where possible, a pure system comparison is difficult, since each system relies on its own rule configuration. Rather, we note that short (<50-line) PClean programs can encode knowledge useful in practice

for cleaning diverse data, and inference is good enough to achieve F_1 scores as good or better than SOTA data-cleaning systems on all three datasets, often in less wall-clock time. As an illustration of the value and convenience of encoding relevant knowledge, on *Flights*, a baseline, 16-line PClean program earns an F_1 score of 0.60, but the F_1 can be boosted to 0.69 by encoding that sources have varying reliability (+1 line), and to 0.90 by encoding that for a given flight, an airline's own website is most likely to be reliable (+1 line). By contrast, adding a similar reliability heuristic to NADEEF required 50 lines of Java.

(3) Scalability to large, real-world data. We ran PClean on the Medicare Physician Compare National dataset, shown earlier in Figure 7-1. It contains 2.2 million records, each listing a clinician and a practice location; the same clinician may work at multiple practices, and many clinicians may work at the same practice. NULLs and systematic errors are common (e.g. consistently misspelled city names at a practice).

PClean took 7h36m, performing 8,245 repairs and 1,535,415 imputations. Out of 100 randomly chosen imputations, 90% agreed with manually obtained ground truth. We also verified that 7,954 repairs (96.5%) were correct (some were correct normalization, e.g. choosing a single spelling for cities whose names could be spelled multiple ways). By contrast, NADEEF changed 88 cells across the whole dataset, and HoloClean did not initialize in 24 hours, using the configuration provided by HoloClean's authors.

Figure 7-1 shows PClean's real behavior on four rows. Consider the misspelling *Abington, MD*, which appears in 152 entries. The correct spelling *Abingdon, MD* occurs in only 42. PClean still recognizes *Abington* as an error, because all 152 instances share a single practice address, and errors are modeled as systematic at the practice level. Next, consider PClean's correct inference that Ryan's degree is *DO*: more *Family Medicine* doctors are MDs than DOs, but the school *PCOM* awards many more DOs than MDs. All parameters enabling this reasoning are learned from the dirty data.

(4) MAP estimation and Bayesian uncertainty quantification. Our previous experiments used a single posterior sample to estimate the clean dataset. This experiment investigates strategies for exploiting richer information about the posterior, namely: (1) using the most common predictions across multiple independent posterior samples (approximating the MAP clean dataset), and (2) setting a confidence threshold for repairs, to trade recall for higher precision. In particular, we ran PClean's inference with 10 parallel chains on the *Rents* dataset, and collected 1 posterior sample from each (the last iterate of MCMC rejuvenation). Figure 7-4's left panel shows, in blue, the precision and recall achieved by considering each sample individually, and in red, the various precision/recall tradeoffs achievable by using *most common* prediction (across all 10 samples) for each cell, or leaving a cell unmodified if the confidence (i.e., the proportion of samples in which the modal value was predicted) does not surpass a threshold. The optimal F_1 of 0.73 ($R = 0.70, P = 0.77$), a 4-point

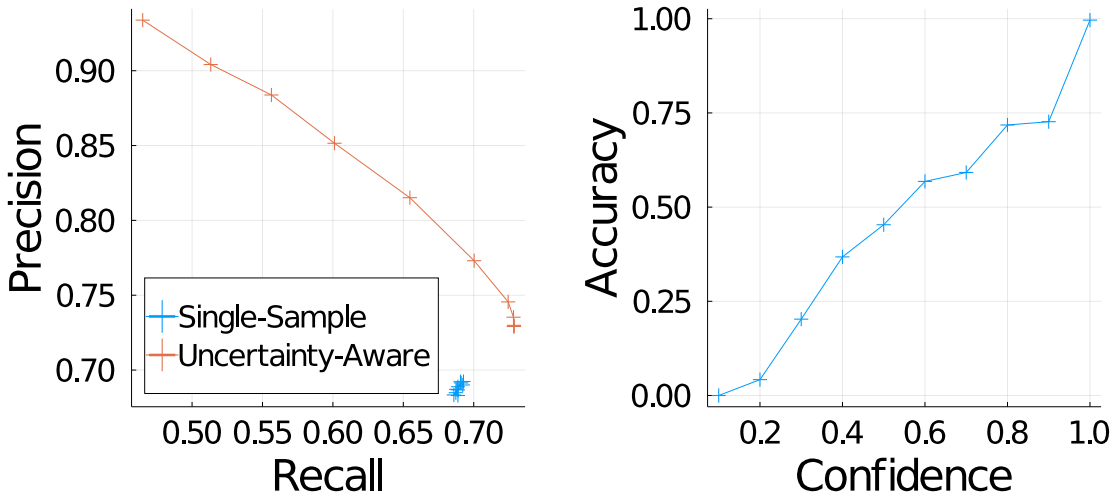


Figure 7-4: Exploiting Bayesian uncertainty with PClean on *Rents*. Left: Ten independent posterior samples were generated. Blue marks show precision and recall achieved by considering each sample separately, whereas red curve shows precision vs. recall tradeoff when using modal predictions across samples, making repairs only when predictions surpass a confidence threshold. Right: PClean's uncertainty quantification appears to be well-calibrated on *Rents*. 'Confidence' is the proportion of independent posterior samples that agree on the modal predicted value for a cell; 'accuracy' is the proportion of cells at a certain confidence level for which the modal prediction is correct.

improvement over the results from Table 7.1, is achieved by thresholding at 0.5. The right panel of Figure 7-4 shows a calibration plot, obtained by binning the cells of the *Rents* dataset into confidence levels and measuring the proportion of cells in each bin for which the most common prediction was correct. A caveat of our approach to uncertainty-aware cleaning is that performing independent repairs *per cell* will not necessarily approximate the overall MAP clean dataset, unless the posterior is actually independent. Moreover, *Rents* is a challenging but synthetic dataset; the degree to which these calibration results replicate in real-world problems will depend on the fidelity of the user's PClean program. However, our results suggest that PClean's inference engine is capable of delivering not only accurate cleaning results but also useful estimates of uncertainty.

7.5 Related work

Many researchers have proposed generative models for cleaning specific datasets or fixing particular error patterns [Pasula et al., 2003, Kubica and Moore, 2003, Mayfield et al., 2009, Matsakis, 2010, Xiong et al., 2011, Hu et al., 2012, Zhao et al., 2012, Abedjan et al., 2016, De et al., 2016, Steorts et al., 2016, Winn et al., 2017,

De Sa et al., 2019, Eduardo et al., 2020, Marchant et al., 2021]. Such formulations specify priors over ground truth data, and likelihoods that model errors. In contrast, PClean's PPL makes it easy to write short (<50 line) programs to specify custom models of new datasets, and automates an inference algorithm that delivers fast, accurate cleaning results.

PClean draws on a rich literature of Bayesian approaches to modeling relational data [Friedman et al., 1999], including open-universe models with identity and existence uncertainty [Milch and Russell, 2006]. Many PPLs could express PClean-like data cleaning models [Milch et al., 2005, Goodman et al., 2008a, Goodman and Stuhlmüller, 2014, Mansinghka et al., 2014b, Gordon et al., 2014, Tolpin et al., 2016, Ścibior et al., 2018, Bingham et al., 2019a, Cusumano-Towner et al., 2019a], but in practice, generic PPL inference is often too slow. This chapter introduces new algorithms that scale better, and demonstrates external validity of the results by calibrating PClean's runtime and accuracy against SOTA data-cleaning baselines [Dallachiesat et al., 2013, Rekatsinas et al., 2017] that use machine learning and weighted logic (typical of discriminative approaches [Mccallum and Wellner, 2003, Wellner et al., 2004, Wick et al., 2013]).

Some of PClean's inference innovations have close analogues in traditional cleaning systems; for example, PClean's preferred values from Section 7.3.3 are related to HoloClean's notion of domain restriction. In fact, PClean can be viewed as a scalable, Bayesian, domain-specific PPL implementation of the PUD framework from De Sa et al. [2019] (which abstractly characterizes the HoloClean implementation from Rekatsinas et al. [2017], but does not itself include PClean's modeling or inference innovations). Some of PClean's inference contributions also have precursors in LibBi and Birch [Murray, 2015, Murray and Schön, 2018], which, like PClean, employ sequential Monte Carlo algorithms with data-driven proposals. However, Birch's delayed sampling technique [Murray et al., 2018] would *not* yield intelligent, data-driven proposals in PClean's non-parametric model class, and in Section 7.4, we show that PClean's novel inference contributions (including its static generation of model-specific proposal code, and its per-object rejuvenation schedule) are necessary for efficient, accurate cleaning.

7.6 Discussion

PClean, like other domain-specific PPLs, aims to be more automated and scalable than general purpose PPLs, by leveraging structure in its restricted model class to deliver fast inference. At the same time, it aims to be expressive enough to concisely solve a broad class of real-world data cleaning problems.

Future development of PClean could build a more extensive standard library of primitives for modeling diverse data types (perhaps including neural models for text or image data), and a more robust proposal compiler for free-text and continuous latent variables (perhaps based on learning neural proposals for selected attributes). Integrating PClean with the abstractions of a mature probabilistic

programming system, such as Gen, could facilitate implementation. PClean's scalability could also be improved, by exploring distributed variants of PClean based on Bayesian formulations of blocking [Marchant et al., 2021]. A more speculative research direction is to (partially) automate PClean program authoring, by applying techniques such as automated error modeling [Heidari et al., 2019] or probabilistic program synthesis [Saad et al., 2019a, Choi et al., 2020]. It could also be fruitful to develop hierarchical variants of PClean that enable learned parameters and latent objects to transfer across datasets.

PClean could be described as a *data-driven* probabilistic expert system [Horvitz et al., 1988, Pearl, 1988, Heckerman et al., 1992, Shafer, 1996], incorporating ideas from probabilistic programming to scale to messy, real-world domain knowledge and data. Crucially, since PClean can infer the objects and parameters of a domain from data, users need only encode higher-level domain knowledge, not brittle details. It remains to be seen whether systems like PClean can be made to give meaningful explanations of individual judgments (like those offered by human experts).

Our results show that probabilistic programs can clean dirty, denormalized data with state-of-the-art accuracy and performance. More broadly, PClean joins existing domain-specific PPLs in demonstrating that it is feasible and useful to integrate sophisticated styles of modeling and inference, developed over years of research, into simple languages and specialized inference engines. We hope PClean proves useful to practitioners, and that it encourages researchers to develop new domain-specific PPLs for other important problems.

8

CONTROLLABLE GENERATION FROM LANGUAGE MODELS

The work in this chapter combines results from three papers: Lew et al. [2023d], Loula et al. [2025], and Lipkin et al. [2025]. It is joint work with João Loula, Tan Zhi-Xuan, Gabe Grand, Ben Lipkin, Ben LeBrun, Jacob Hoover Vigly, Clemente Pasti, Tianyu Liu, Yahya Emara, Marjorie Freedman, David MacIver, Li Du, Jason Eisner, Joshua Tenenbaum, Ryan Cotterell, Jacob Andreas, Vikash Mansinghka, Tim Vieira, and Tim O'Donnell.

This chapter presents new techniques for controllable generation from language models, based on sequential Monte Carlo. The key idea is that users can *programmatically* specify a *target distribution* from which they wish to generate samples, as a product of the language model's base distribution over strings with additional *potential functions* that encode domain-specific constraints. We explore several strategies for efficiently generating samples approximately distributed according to this distribution, based on different *proposal distributions*.

Crucially, at each step of inference, we require an (unbiased estimate of) the **Radon-Nikodym derivative** between the user's target distribution and our choice of proposal distribution. Our proposal distributions are presented in this chapter as *algorithms*, but can equivalently be seen as probabilistic programs. For example, our adaptive rejection sampling proposal can be implemented as a recursive traced probabilistic program in the language $\lambda_{\ll}$ of Chapter 4, which continually generates categorical samples from the language model's next-token distribution until a sample is accepted. The trace distribution of such a program can then be marginalized to yield a distribution on only the accepted sample, which is precisely the proposal distribution for which we require a Radon-Nikodym derivative. The properly weighted samplers described in Sections 8.3 and 8.4.2 can in fact be derived by considering the estimators that would be automated by our **spi** transformation for

such a program.[1]

8.1 Motivation: Controllable generation with language models

The goal of *controlled generation* from language models (LMs) is to produce text guided by a set of syntactic or semantic constraints. One prominent example is semantic parsing, or code generation, which involves producing text in a programming (or other formal) language, typically from a natural language prompt. We may wish to use diverse signals to guide code generation, for example:

- Checking (partial) code statically (type-checking, linting, partial evaluation);
- Running (partial) code on a test case and checking if it raises an error or returns the wrong answer;
- Simulating environments (e.g., in robotics or chemistry) and assigning a score to the resulting state;
- Rolling out possible completions of partial code and computing their max, min, or average score;
- Asking another language model to critique the code generated so far.

Such signals vary along several important dimensions: some are cheap to compute (linting), others are more costly (simulations); some can be evaluated incrementally with each sampled token (language model critique), others provide sparser guidance (running code); some enforce binary hard constraints (type-checking), others yield soft continuous scores (scoring).

One way to represent such signals uniformly is as *potential functions* $\phi(\boldsymbol{x})$ assigning non-negative scores to sequences of tokens $\boldsymbol{x}$. Given a set Φ of such potentials, we will write $\Phi(\boldsymbol{x}) = \prod_{\phi \in \Phi} \phi(\boldsymbol{x})$. We frame the problem of controlled generation probabilistically: We wish to sample from the *global product of experts* distribution on complete sequences $\boldsymbol{x}$:

$$g(\boldsymbol{x}) = \frac{1}{Z} p(\boldsymbol{x}) \Phi(\boldsymbol{x}), \tag{8.1}$$

where p is a distribution over complete token sequences defined by an autoregressive LM, and Z is a normalizing constant. The distribution g can be interpreted as a posterior with p as the prior and Φ as the likelihood function. Importantly, even

[1] We arrived at the estimators presented in this chapter following the logic of Chapter 4. In particular, we considered the behavior of the **spi** transformation on implementations of our proposal distributions constructed using *marginal* to marginalize auxiliary variables. As estimation strategies, we used *meta-proposals* also defined as probabilsitic programs. However, we did not directly use our implementations of **spi** in this work; instead, we developed a specialized probabilistic programming language, LLaMPPL [Lew et al., 2023d], that is tailored for defining and efficiently executing probabilistic progrmas that invoke language models.

when both sampling from p and evaluating $\Phi(\boldsymbol{x})$ are efficient, sampling exactly from g is generally intractable [Rosenfeld et al., 2001].

Two popular sampling-based approaches that avoid the intractability of g are *locally constrained decoding* and *sample reranking*. Locally constrained decoding uses per-token logit biasing or masking to incorporate signals at each step, for example to ensure that the complete sequence will fall in a specified regular or context-free language [e.g., Shin et al., 2021, Scholak et al., 2021, Poesia et al., 2022, Willard and Louf, 2023, Moskal et al., 2024, Ugare et al., 2024]. Sample-rerank-based approaches first generate complete sequences and then rerank or reweight these based on the specified set of signals. Examples of this approach include best-of-n reweighting with a reward model [Nakano et al., 2021, Krishna et al., 2022, Zhou et al., 2023, Gui et al., 2024, Mudgal et al., 2024, Ichihara et al., 2025] or filtering samples with a verifier [Olausson et al., 2023, Chen et al., 2024, Lightman et al., 2024, Xin et al., 2024]. Each approach suffers from significant weaknesses. Locally constrained decoding requires the guiding signals Φ to be cheap enough to evaluate very frequently (for instance on the full token vocabulary at every step of generation). Moreover, it often introduces greedy approximations that badly distort the distribution [relative to g, Lew et al., 2023d, Park et al., 2025]. Sample-rerank does not impose constraints Φ until full sequences have been sampled and, thus, cannot make use of information available incrementally during generation; this can significantly increase the number of samples needed to find high probability, constraint-satisfying sequences.

Sequential Monte Carlo (SMC) has been proposed as an effective approach to approximate inference for such intractable distributions in other difficult language modeling problems, such as infilling, prompt engineering, and prompt intersection as well as for more traditional tasks in natural language processing [Börschinger and Johnson, 2011, Dubbin and Blunsom, 2012, Buys and Blunsom, 2015, Lin and Eisner, 2018, Lew et al., 2023d, Zhao et al., 2024]. In this work, we use SMC to tackle a number of challenging semantic parsing problems, guiding generation with incremental static and dynamic analyses. When these signals are efficient enough to be used incrementally, our approach incorporates them into *proposal distributions*, gaining the benefits of locally constrained decoding; more costly potentials—as often used in sample-rerank approaches—are incorporated as *twist functions* that reweight partial sequences to favor promising paths [Naesseth et al., 2019a]. Our approach emphasizes *programmable* potentials and proposals that can easily be specialized for specific tasks or problems (e.g., by integrating libraries for molecular structure or robotic planning, see Section 8.5.1). We contrast this with the use of *learned* proposals or twist functions [Lawson et al., 2022, Zhao et al., 2024], which requires costly, problem-specific fine-tuning.

We make the following contributions:

- *SMC for constrained semantic parsing.* We develop an architecture specializing SMC for code generation under diverse syntactic and semantic constraints (Section 8.2). Unlike many previous frameworks for constrained decoding, our algorithm can integrate constraints that cannot be incrementally evaluated over the entire token

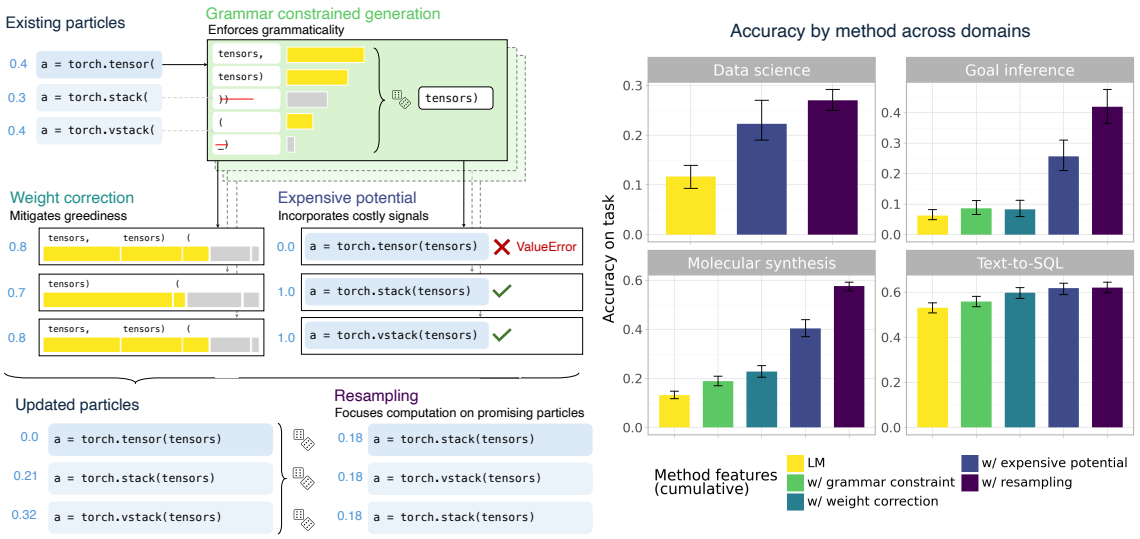


Figure 8-1: *Controlled generation from LMs via sequential Monte Carlo. Left:* We use sequential Monte Carlo to sample from high-quality approximations to posteriors over LM outputs. Partial sequences are repeatedly extended via grammar-constrained generation. We then apply weight corrections to mitigate the greediness of locally constrained decoding, as well as expensive potentials to encode rich information that cannot be included in logit masks. Finally, resampling focuses computation on promising particles. *Right:* Accuracy gains from these innovations on challenging data science, text-to-SQL, goal inference, and molecule synthesis benchmarks.

vocabulary, as well as constraints that can only be evaluated at irregular intervals during generation. The framework emphasizes *programmable inference* [Mansinghka et al., 2018b], allowing users to deploy proposals and potentials that exploit the structure of application domains. Our system also fully integrates with the language model probabilistic programming framework of Lew et al. [2023d].

- *Empirical evaluation of performance in diverse domains.* We apply our approach—and six alternatives—to four challenging problem domains: Python code generation for data science, text-to-SQL, goal inference, and molecule synthesis (Section 8.5.1). We find that, with little overhead, our approach significantly improves performance across domains, allowing small open-source language models to outperform models over 8× larger, as well as closed-source fine-tuned models. We additionally find that, with 5–10× fewer particles, SMC outperforms approaches that only incorporate constraints at the end of generation (Section 8.5.2).

- *Empirical evaluation of algorithm components.* We run ablation experiments and find that improved performance can be attributed to three algorithmic components: *weight correction*, which mitigates the greediness of locally constrained decoding; *expensive potentials*, which incorporate useful signals that several baseline methods cannot integrate; and *adaptive resampling*, which adaptively refocuses computation

on partial sequences that look more promising.

- *Empirical validation of the probabilistic perspective.* We estimate the KL divergence from each method's output distribution to the global product of experts. We find that the best-performing methods (i) have outputs that are closer in KL divergence to the global product of experts within each problem instance, and (ii) assign probabilities that are more correlated with downstream performance across problem instances (Section 8.5.3).

8.2 Monte Carlo inference for constrained generation

Notation. We use $\boldsymbol{x}$ to refer to a sequence of tokens, with x_i being the i^{th} token in the sequence. Let $\boldsymbol{x}_{<t} \stackrel{\text{def}}{=} x_1 \cdots x_{t-1}$. Let ε denote the empty sequence. We use juxtaposition (e.g., $\boldsymbol{x}\,\boldsymbol{y}$) to denote sequence concatenation. Let $\mathcal{A}$ be a vocabulary of tokens, and let $\mathcal{A}^*$ denote the set of all finite sequences of tokens in $\mathcal{A}$. We refer to the set $\mathcal{A}^*$ as the set of **partial sequences**. We use $\texttt{EOS} \notin \mathcal{A}$ to denote a special token marking the end of sequences not included in $\mathcal{A}$. We define $\mathcal{A}_{\texttt{EOS}} \stackrel{\text{def}}{=} \mathcal{A} \cup \{\texttt{EOS}\}$ and the set of **complete sequences** $\mathcal{A}^*\texttt{EOS} \stackrel{\text{def}}{=} \{\boldsymbol{x}\,\texttt{EOS} \mid \boldsymbol{x} \in \mathcal{A}^*\}$.

Language models. A **language model** p is a probability distribution over complete sequences (i.e., $\sum_{\boldsymbol{x} \in \mathcal{A}^*\texttt{EOS}} p(\boldsymbol{x}) = 1$). We assume that p provides a conditional distribution $p(x' \mid \boldsymbol{x})$ over its next token $x' \in \mathcal{A}_{\texttt{EOS}}$ given any sequence $\boldsymbol{x} \in \mathcal{A}^*$; the probability of any complete sequence factors as

$$p(\boldsymbol{x}) = \prod_{t=1}^{|\boldsymbol{x}|} p(x_t \mid \boldsymbol{x}_{<t}). \tag{8.2}$$

We find it convenient to extend the definition of $p(\boldsymbol{x})$ from Equation (8.2) to partial sequences $\boldsymbol{x} \in \mathcal{A}^*$; note, however, that $p(\boldsymbol{x})$ is a *prefix* probability,[2] so p is *not* a probability distribution over partial sequences. With this extended definition, we have $p(x' \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x}\,x')}{p(\boldsymbol{x})}$ provided $p(\boldsymbol{x}) > 0$.

Potential functions. We consider a set Φ of domain- or task-specific **potential functions** that encode relevant constraints or preferences as nonnegative scores. Each potential function $\phi \in \Phi$ has the type $\phi\colon (\mathcal{A}^* \cup \mathcal{A}^*\texttt{EOS}) \to \mathbb{R}_{\geq 0}$, meaning that it assigns a non-negative real $\phi(\boldsymbol{x})$ when evaluated on some sequence $\boldsymbol{x}$, freely using any structure in the sequence so far regardless of whether $\boldsymbol{x}$ is a partial or complete sequence. For technical reasons, we assume that all potentials satisfy $\phi(\boldsymbol{x}) = 0 \implies \phi(\boldsymbol{x}\,\boldsymbol{y}) = 0$, for all $\boldsymbol{x}$ and $\boldsymbol{y}$ such that $\boldsymbol{x}\,\boldsymbol{y} \in \mathcal{A}^*\texttt{EOS}$.

Target distribution. We formalize the goal of controlled generation as sampling sequences $\boldsymbol{x} \in \mathcal{A}^*\texttt{EOS}$ from the **target distribution** given by the **global product of**

[2] I.e., $p(\boldsymbol{x})$ is the probability that a *complete* sequence $\boldsymbol{x}' \sim p$ has the *partial* sequence $\boldsymbol{x}$ as a prefix.

experts between p and Φ:[3]

$$g(\boldsymbol{x}) = \frac{1}{Z} p(\boldsymbol{x})\Phi(\boldsymbol{x}) \quad \text{where} \quad Z = \sum_{\boldsymbol{y} \in \mathcal{A}^* \texttt{EOS}} p(\boldsymbol{y})\Phi(\boldsymbol{y}). \tag{8.3}$$

For intuition, if $\Phi(\boldsymbol{x}) \in \{0, 1\}$ for all $\boldsymbol{x}$ in $\mathcal{A}^*\texttt{EOS}$, the global product of experts can be understood as the **rejection sampling** distribution that arises by repeatedly generating $\boldsymbol{x} \sim p$, and rejecting if $\Phi(\boldsymbol{x}) = 0$. The normalizing constant Z is the rate at which samples are accepted. Thus, the expected runtime of rejection sampling is $1/Z$ per accepted sample, making it expensive if Z is small. Our work aims to accurately approximate the global product of experts with much less computation.

Locally constrained decoding. A popular approach[4] to enforcing constraints at inference-time is to apply them before sampling each token. In this approach, at each time step t, the current sequence $\boldsymbol{x}_{<t}$ is extended with a new token $x_t \sim \ell_\Phi(\cdot \mid \boldsymbol{x}_{<t})$ (until $x_t = \texttt{EOS}$) where[5]

$$\ell_\Phi(x_t \mid \boldsymbol{x}_{<t}) \stackrel{\text{def}}{=} \frac{p(x_t \mid \boldsymbol{x}_{<t}) \frac{\Phi(\boldsymbol{x}_{<t} x_t)}{\Phi(\boldsymbol{x}_{<t})}}{L_\Phi(\boldsymbol{x}_{<t})} \quad \text{where} \quad L_\Phi(\boldsymbol{x}_{<t}) \stackrel{\text{def}}{=} \sum_{x' \in \mathcal{A}_{\texttt{EOS}}} p(x' \mid \boldsymbol{x}_{<t}) \frac{\Phi(\boldsymbol{x}_{<t} x')}{\Phi(\boldsymbol{x}_{<t})} \tag{8.4}$$

We write $\ell_\Phi(\boldsymbol{x}) \stackrel{\text{def}}{=} \prod_{t=1}^{|\boldsymbol{x}|} \ell_\Phi(x_t \mid \boldsymbol{x}_{<t})$ for either the probability of $\boldsymbol{x} \in \mathcal{A}^*\texttt{EOS}$ or the prefix probability of $\boldsymbol{x} \in \mathcal{A}^*$. Note that in the former case, ℓ_Φ is a distribution over complete sequences. We call it a ***local* product of experts** model because normalization is performed *locally* (at each step of the sequence), rather than *globally* (once per complete sequence).

Despite its popularity, locally constrained decoding has two important shortcomings. First, for most practical potential functions, the local and global product of experts do not define the same distribution [Lew et al., 2023d, Park et al., 2025]. In particular, while the global product of experts is defined with respect to complete sequences, the local product typically only has access to the string generated so far and a single token of lookahead—which can lead to myopic sampling down paths that lead to globally poor solutions. In principle, this problem can be mitigated by the choice of intermediate potentials ($\Phi(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathcal{A}^*$), which implement more aggressive forms of lookahead. In particular, locally constrained decoding is an *exact* sampler when $\Phi(\boldsymbol{x}) = \Phi^*(\boldsymbol{x})$, the **expected future potential** of $\boldsymbol{x}$, $\Phi^*(\boldsymbol{x}) \stackrel{\text{def}}{=} \mathbb{E}_{\boldsymbol{x}' \sim p}\left[\Phi(\boldsymbol{x}') \,\middle|\, \boldsymbol{x} \text{ is a prefix of } \boldsymbol{x}'\right]$.[6] Unfortunately, much like Z, computing Φ^* exactly is

[3] Note that care has to be taken to ensure that the sum which defines the normalizing constant in Equation (8.3) converges. One sufficient (but not necessary) condition ensuring this is if Φ is bounded above by a constant.

[4] E.g., Shin et al. [2021], Scholak et al. [2021], Poesia et al. [2022], Willard and Louf [2023], Ugare et al. [2024].

[5] Here our assumption that $\phi(\boldsymbol{x}) = 0 \implies \phi(\boldsymbol{x}\,\boldsymbol{y}) = 0$ ensures that whenever $\Phi(\boldsymbol{x}_{<t}) = 0$, all extensions will be 0 as well, making it safe in such cases to define $\frac{\Phi(\boldsymbol{x}_{<t}\,x)}{\Phi(\boldsymbol{x}_{<t})} = 0$.

[6] Φ^* is also known in the SMC literature as the *optimal twist function* [e.g., Zhao et al., 2024].

typically intractable. Although we may seek to approximate it, for example, by learning [Zhao et al., 2024] or adaptive methods [Park et al., 2025], here we instead focus on variants of locally constrained decoding which marginalize over the immediate next token as in Equation (8.4); see Footnote 4. Our experiments (Section 8.5) compare our method to this dominant form of local decoding from the literature, using efficient tests for whether the addition of a single candidate token can satisfy the constraint.

The second, related problem with locally constrained decoding is that the local product of experts can only be sampled efficiently when it is possible to cheaply evaluate the potentials $\phi \in \Phi$ on all possible one-token continuations $\boldsymbol{x}_{<t}\, x'_t$ of the current sequence. For some constraints (e.g., checking membership or prefixhood in the language of a regular expression or context-free grammar), algorithms exist for efficient parallel evaluation across tens of thousands of possible continuations. However, for many of the constraints of interest in the present chapter (including several listed in Table 8.1, for example, error-checking with test cases) this is not feasible. In what follows, we will assume that the set of potentials Φ can be partitioned into **expensive potentials** Φ_{exp}, which are too costly to use as part of locally constrained decoding, and **efficient potentials** Φ_{eff}, which can be used in sampling from the local product of experts.

Importance sampling. The shortcomings of local decoding can be addressed with **importance sampling**, a Monte Carlo technique for approximating intractable distributions. We describe a particular application of the technique specialized to our setting. Here, we use the local product of experts model $\ell_{\Phi_{\text{eff}}}$ (abbreviated ℓ_{eff}) with respect to just Φ_{eff} as a **proposal distribution**, from which we sample multiple complete **particles** $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)} \overset{\text{i.i.d.}}{\sim} \ell_{\text{eff}}$. For each particle $\boldsymbol{x}^{(i)}$ we define its **importance weight** $w^{(i)}$ as

$$w^{(i)} \overset{\text{def}}{=} \frac{p(\boldsymbol{x}^{(i)}) \cdot \Phi(\boldsymbol{x}^{(i)})}{\ell_{\text{eff}}(\boldsymbol{x}^{(i)})} = \left(\prod_{t=1}^{|\boldsymbol{x}^{(i)}|} L_{\text{eff}}(\boldsymbol{x}^{(i)}_{<t}) \right) \cdot \Phi_{\text{exp}}(\boldsymbol{x}^{(i)}). \tag{8.5}$$

The numerator is an unnormalized variant of the target distribution g, which we write as $\widetilde{g}$ hereafter, while the denominator ℓ_{eff} is the proposal distribution that was used to draw the sequence. These weighted particles define the following **posterior approximation**: $\widehat{g}(\boldsymbol{x}) \overset{\text{def}}{=} \frac{\sum_{i=1}^{N} w^{(i)} \mathbf{1}_{\boldsymbol{x}=\boldsymbol{x}^{(i)}}}{\sum_{j=1}^{N} w^{(j)}}$, which under mild conditions converges to g as N grows.[7] Our importance weights simplify as shown in Equation (8.5), and we note how they correct for the two problems we identified with ℓ_{eff}. The first factor, $\prod_{t=1}^{|\boldsymbol{x}^{(i)}|} L_{\text{eff}}(\boldsymbol{x}^{(i)}_{<t})$, corrects for the greediness of ℓ_{eff}, penalizing particles where *all* possible continuations $x_t \in \mathcal{A}_{\text{EOS}}$ score poorly in context. The second factor, $\Phi_{\text{exp}}(\boldsymbol{x}^{(i)})$, incorporates the expensive potentials that could not be used in ℓ_{eff}. These importance weights can be computed efficiently: the first factor is already computed

[7] However, the number of particles required to obtain a *good* approximation of the target distribution is exponential in the KL divergence between target g and proposal ℓ_{eff} [Chatterjee and Diaconis, 2018].

as a byproduct of sampling from ℓ_{eff}, and the second factor is computed by running each of the expensive efficient potentials once on each $\boldsymbol{x}^{(i)}$.

Sequential Monte Carlo. While importance sampling addresses several shortcomings of local decoding, it too suffers from a major weakness: weight corrections and expensive potentials are not integrated until after a complete sequence has been generated from the proposal. This is despite the fact that critical information about whether a sequence can satisfy a constraint is often available much earlier and can be used to avoid large amounts of unnecessary computation. Sequential Monte Carlo [**SMC**; e.g., Chopin et al., 2020], is a natural generalization of importance sampling that constructs importance-weighted samples from a *sequence* of unnormalized target distributions $\langle \widetilde{g}_t \rangle_{t=0}^{\infty}$ to arrive at the final unnormalized target $\widetilde{g}$. In our case, we consider intermediate targets $\widetilde{g}_t$ defined on the sequence of spaces $\mathcal{A}^{(t)} \stackrel{\text{def}}{=} \{\boldsymbol{x} \in \mathcal{A}^* \mid |\boldsymbol{x}| = t\} \cup \{\boldsymbol{x} \in \mathcal{A}^* \texttt{EOS} \mid |\boldsymbol{x}| \leq t\}$, that is, partial sequences of length equal to t and complete sequences of length less than or equal to t. The targets are defined as

$$\widetilde{g}_t(\boldsymbol{x}) = p(\boldsymbol{x})\Phi(\boldsymbol{x}), \text{ for } \boldsymbol{x} \in \mathcal{A}^{(t)}. \tag{8.6}$$

Note that $\widetilde{g}$ and $\widetilde{g}_t$ are unnormalized distributions over different spaces: $\mathcal{A}^* \texttt{EOS}$ and $\mathcal{A}^{(t)}$ respectively. Whereas $\widetilde{g}$ only considers potentials on *complete* sequences, $\widetilde{g}_t$ depends also on the behavior of the potentials Φ_{exp} when applied to *partial* sequences. But as $t \to \infty$, there is less and less mass on partial sequences, and $\widetilde{g}_t$ approaches $\widetilde{g}$ no matter how the partial potentials are defined.

The particles for $\widetilde{g}_t$ are drawn as *prefixes* from ℓ_{eff} (stopping at length t if $\texttt{EOS}$ has not been reached), again requiring an importance weighting correction. The importance weight at time t can be re-expressed as the importance weight from time $t-1$ times a correction factor for step t:

$$w_t^{(i)} \stackrel{\text{def}}{=} \frac{\widetilde{g}_t(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\ell_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}\, x_t)} \tag{8.7a}$$

$$= \widetilde{g}_0(\boldsymbol{x}_{<1}^{(i)}) \prod_{t'=1}^{t} \frac{\widetilde{g}_{t'}(\boldsymbol{x}_{<t'}^{(i)}\, x_{t'}^{(i)})}{\widetilde{g}_{t'-1}(\boldsymbol{x}_{<t'}^{(i)})\, \ell_{\text{eff}}(x_{t'} \mid \boldsymbol{x}_{<t'}^{(i)})} \tag{8.7b}$$

$$= w_{t-1}^{(i)} \frac{\widetilde{g}_t(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\widetilde{g}_{t-1}(\boldsymbol{x}_{<t}^{(i)})\, \ell_{\text{eff}}(x_t \mid \boldsymbol{x}_{<t}^{(i)})} \tag{8.7c}$$

$$= w_{t-1}^{(i)} \cdot L_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}) \cdot \frac{\Phi_{\text{exp}}(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\Phi_{\text{exp}}(\boldsymbol{x}_{<t}^{(i)})}. \tag{8.7d}$$

The sequential Monte Carlo algorithm generates approximations to each intermediate target $\widetilde{g}_t$, using **resampling steps** to reallocate computation from less to more promising partial sequences. We begin with a collection of N weighted particles $(\boldsymbol{x}^{(i)}, w^{(i)}) = (\varepsilon, 1)$, where ε is the empty sequence of tokens. Then, starting at $t = 1$, we repeat the following three steps until all particles are $\texttt{EOS}$-terminated

(i.e., $\boldsymbol{x}^{(i)} \in \mathcal{A}^* \texttt{EOS}$ for all i):

1. *Extend.* For each incomplete particle $\boldsymbol{x}^{(i)}$, sample $x_t^{(i)} \sim \ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}^{(i)})$ and update $\boldsymbol{x}^{(i)} \leftarrow \boldsymbol{x}^{(i)}\, x_t^{(i)}$.
2. *Reweight.* For each extended particle $\boldsymbol{x}^{(i)}$, update the weight $w^{(i)} \leftarrow w^{(i)}\, L_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}) \frac{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)})}$.
3. *Resample.* Sample ancestor indices $a^{(1)}, \ldots, a^{(N)} \overset{\text{i.i.d.}}{\sim} \text{Categorical}\left(\frac{w^{(1)}}{W}, \ldots, \frac{w^{(N)}}{W}\right)$ where $W = \sum_{i=1}^{N} w^{(i)}$. Then, reassign all particles $(\boldsymbol{x}^{(i)}, w^{(i)}) \leftarrow (\boldsymbol{x}^{(a^{(i)})}, \frac{W}{N})$ for all i simultaneously.[8]

The extension step extends each incomplete particle with a next token proposed by the local product of experts ℓ_{eff}. The reweighting step computes the updated importance weight, incorporating a new factor for the new token. The resampling step exploits any early signal available in the updated weights at time t to abandon some less promising incomplete particles (which are unlikely to be chosen as ancestors) and focus more future computation on more promising particles (which are likely to be chosen as ancestors multiple times and then will be extended in multiple ways at time $t+1$).

Further extensions. We further extend our SMC implementation in two ways. First, potentials in Φ_{eff} may still be modestly expensive to evaluate on the entire vocabulary. In these cases, we develop cheap stochastic approximations to the local product of distributions and use these as proposals during the *Extend* step. The incremental weight computation must also be corrected to account for these approximations; we derive stochastic unbiased estimators of the incremental weights that can be soundly used within SMC (see Section 8.3). Second, the intermediate targets $\widetilde{g}_t$ do not need to advance token-by-token; in some domains, it is beneficial to consider more semantically meaningful increments. For example, in one of our experiments, the intermediate target p_t is defined over the space of all partial Python programs containing t or fewer lines of code (rather than tokens); the *Extend* step then samples a different number of tokens per particle, waiting in each partial sequence until a new full line has been generated. Such strategies can lead to better *particle alignment* [Lundén et al., 2018], making resampling more effective.

8.3 Estimating Radon-Nikodym derivatives for fast, set-based proposals

This section describes a set-based proposal speedup (Section 8.3.1). We instantiate the speedup in Section 8.3.2 for the special case of character-level potential functions, such as parsing.

[8] In practice, we only resample if the **effective sample size** $\widehat{N} \overset{\text{def}}{=} \frac{\left(\sum_{i=1}^{N} w^{(i)}\right)^2}{\sum_{i=1}^{N} (w^{(i)})^2}$ is under a threshold (e.g., $\frac{N}{3}$).

8.3.1. Framework

The *reweight* step of our sequential Monte Carlo algorithm requires us to evaluate $L_{\text{eff}}(\boldsymbol{x})$, and the *extend* step requires us to sample from the locally constrained distribution $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x})$. Both of these operations can be moderately expensive because they require the evaluation of $\Phi_{\text{eff}}(\boldsymbol{x}\, x')$ for all tokens $x' \in \mathcal{A}_{\texttt{EOS}}$. In this section, we describe a general scheme for speeding up both steps by *approximately* sampling $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x})$ and *approximately* evaluating $L_{\text{eff}}(\boldsymbol{x})$. If we are careful about how we carry out this approximation, it will not change the intermediate targets $\widetilde{g}_t(\boldsymbol{x})$, defined Equation (8.6), that control the behavior of the sequential Monte Carlo algorithm.

Sequential Monte Carlo with approximate *Extend* and *Reweight* steps. The key invariant maintained by sequential Monte Carlo is that at each step, the distribution of each particle $(\boldsymbol{x}_t^{(i)}, w_t^{(i)})$ is *properly weighted* for the intermediate target $\widetilde{g}_t$.

The *extend* and *reweight* steps of the algorithm, which update a previous particle $(\boldsymbol{x}_{<t}^{(i)}, w_{t-1}^{(i)})$ by sampling $x_t^{(i)} \sim \ell_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)})$ and returning the new particle

$$\left(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)}, w_{t-1}^{(i)} \cdot L_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}) \cdot \frac{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)})}\right)$$

are justified by the fact that if $(\boldsymbol{x}_{<t}, w_{t-1}^{(i)})$ is properly weighted for $\widetilde{g}_{t-1}$, then the new pair is properly weighted for $\widetilde{g}_t$. We are looking to improve runtime performance without compromising soundness, so we seek ways of modifying the *extend* and *reweight* steps that do not break this invariant.

In particular, suppose that instead of sampling $x_t^{(i)} \sim \ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}^{(i)})$ and computing the *exact* weight update $w_t^{(i)} \stackrel{\text{def}}{=} w_{t-1}^{(i)} \cdot L_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}) \cdot \frac{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)}\, x_t^{(i)})}{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)})}$, we instead generate (x, W) from a proposal q that is *properly weighted* for the unnormalized local product of experts $\widetilde{\ell}(x) \stackrel{\text{def}}{=} L_{\text{eff}}(\boldsymbol{x}_{<t}^{(i)}) \cdot \ell_{\text{eff}}(x \mid \boldsymbol{x}_{<t}^{(i)})$, then compute the alternative weight update

$$w_t^{(i)} \stackrel{\text{def}}{=} w_{t-1}^{(i)} \cdot W \cdot \frac{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)}\, x)}{\Phi_{\exp}(\boldsymbol{x}_{<t}^{(i)})} \tag{8.8}$$

The key observation is that, by the fact that q is properly weighted for $\widetilde{\ell}$, we know

that for every possible previous particle $(\boldsymbol{x}^{(i)}_{<t}, w^{(i)}_{t-1})$ and every function f,

$$\mathop{\mathbb{E}}_{(x,W)\sim q}[w^{(i)}_t \cdot f(\boldsymbol{x}^{(i)}_{<t}\, x)] = \mathop{\mathbb{E}}_{(x,W)\sim q}\left[w^{(i)}_{t-1} \cdot W \cdot \frac{\Phi_{\text{exp}}(\boldsymbol{x}^{(i)}_{<t}\, x)}{\Phi_{\text{exp}}(\boldsymbol{x}^{(i)}_{<t})} \cdot f(\boldsymbol{x}^{(i)}_{<t}\, x)\right] \tag{8.9}$$

$$= L_{\text{eff}}(\boldsymbol{x}^{(i)}_{<t}) \mathop{\mathbb{E}}_{x^{(i)}_t \sim \ell_{\text{eff}}(\cdot \mid \boldsymbol{x}^{(i)}_{<t})}\left[w^{(i)}_{t-1} \frac{\Phi_{\text{exp}}(\boldsymbol{x}^{(i)}_{<t}\, x^{(i)}_t)}{\Phi_{\text{exp}}(\boldsymbol{x}^{(i)}_{<t})} f(\boldsymbol{x}^{(i)}_{<t}\, x^{(i)}_t)\right] \tag{8.10}$$

$$= \mathop{\mathbb{E}}_{x^{(i)}_t \sim \ell_{\text{eff}}(\cdot \mid \boldsymbol{x}^{(i)}_{<t})}\left[w^{(i)}_t \cdot f(\boldsymbol{x}^{(i)}_{<t}\, x^{(i)}_t)\right] \tag{8.11}$$

Therefore, if the overall proper weighting invariant (with respect to the intermediate target $\widetilde{g}_t$) holds for the original update, then it will also hold for this modified *extend-and-reweight* procedure. For more on SMC with estimated weights, see Chopin et al. [2020] and Lew et al. [2022b].

A family of properly weighted updates based on the Horvitz–Thompson estimator. We now introduce a useful family of properly weighted proposals for our setting. It will allow us to generate weighted next-token proposals (x, W) while *only* evaluating the potentials Φ_{eff} on a (randomly chosen) subset $S \subseteq \mathcal{A}_{\text{EOS}}$. Our procedure is inspired by the Horvitz–Thompson estimator [Horvitz and Thompson, 1952].

First, to reduce notational clutter, we define the following shorthand: $L \stackrel{\text{def}}{=} L_{\text{eff}}(\boldsymbol{x})$, and $\ell(x') = \widetilde{\ell}(x')/L = \ell_{\text{eff}}(x' \mid \boldsymbol{x})$.

Definition 25. Given a probability distribution q over subsets of $\mathcal{A}_{\text{EOS}}$, we define the **set-based proposal speedup** by the following generation procedure:

1. Sample a subset $S \sim q$ where q is a probability distribution over subsets of $\mathcal{A}_{\text{EOS}}$.
2. Compute the *local weight* $w(x) \stackrel{\text{def}}{=} \frac{\widetilde{\ell}(x)}{\pi(x)}$ of each token $x \in S$ where $\pi_q(x)$ is the **inclusion probability** $\pi_q(x) \stackrel{\text{def}}{=} \Pr_{S'\sim q}[x \in S'] = \sum_{S'} q(S') 1_{x \in S'}$, i.e., the probability that x ends up in *any* sampled $S' \sim q$.[9]
3. Compute the set-conditioned distribution $q(x \mid S) \stackrel{\text{def}}{=} \frac{w(x)\, 1_{x\in S}}{W_S}$ where $W_S = \sum_{x\in S} w(x)$.
4. Sample $x \sim q(\cdot \mid S)$.
5. Return (x, W_S)

Then, as described above, we modify the SMC algorithm to use the sampled token x instead of $x \sim \ell_{\text{eff}}(\cdot \mid \boldsymbol{x})$ in the *extend* step, and the weight W_S instead of $L_{\text{eff}}(\boldsymbol{x})$ in the *reweight* step.

We justify this approach by showing that it produces properly weighted samples.

Proposition 2. *The set-based proposal speedup's distribution q (Definition 25) is a properly weighted proposal for $\widetilde{\ell}$.*

[9] We require q to be such that every token has a positive inclusion probability $\pi_q(x) > 0$ for all $x \in \mathcal{A}_{\text{EOS}}$.

Proof. Let f be an arbitrary real-valued function.

$$\mathop{\mathbb{E}}_{(x,W_S)\sim q}[f(x)W_S] = \sum_x f(x) \sum_S q(x \mid S) q(S) W_S \tag{8.12a}$$

$$= \sum_x f(x) \sum_S \frac{w(x)\, 1_{x \in S}}{W_S} q(S) W_S \tag{8.12b}$$

$$= \sum_x f(x) w(x) \sum_S 1_{x \in S} q(S) \tag{8.12c}$$

$$= \sum_x f(x) \frac{\widetilde{\ell}(x)}{\pi_q(x)} \pi_q(x) \tag{8.12d}$$

$$= L \sum_x f(x) \ell(x) \tag{8.12e}$$

$$= L \mathop{\mathbb{E}}_{x\sim\ell}[f(x)] \tag{8.12f}$$

□

8.3.2. Character-Based Proposal

Our character-based proposal distribution is an instance of the framework of the previous section. In particular, q samples sets of tokens S by sampling a sequence of characters. We provide the pseudocode for this algorithm in Algorithm 8, and define two key data structures used by this proposal:

Definition 26. Our **trie data structure** T is a labeled, tree-structured graph that is defined as follows:

- Let $\mathcal{A}_{\texttt{EOS}}$ be the LM's vocabulary of tokens where we represent each token as its strings of characters ending with a designated end-of-token marker `EOT`.[10] Let Σ denote the set of characters (or bytes).
- Let P be the prefix closure of the set $\mathcal{A}_{\texttt{EOS}}$: $P \stackrel{\text{def}}{=} \{\boldsymbol{p} \in \Sigma^* \,|\, \boldsymbol{p} \preceq x, x \in \mathcal{A}_{\texttt{EOS}}\}$ where $\boldsymbol{p} \preceq x$ denotes that $\boldsymbol{p}$ is a prefix of x.
- Let $T = (N, E)$ be a labeled graph with node P and labeled edges $E = \left\{\boldsymbol{p} \xrightarrow{a} \boldsymbol{p}\, a \,\middle|\, \boldsymbol{p}, (\boldsymbol{p}\, a) \in P\right\}$

Definition 27. Let `mass` be a mapping $N \to [0, 1]$, defined as follows.

$$\texttt{mass}(x') = p(x' \mid \Gamma), \qquad \text{for } x' \in \mathcal{A}_{\texttt{EOS}} \tag{8.13a}$$

$$\texttt{mass}(\boldsymbol{p}) = \sum_{\boldsymbol{p} \xrightarrow{a} \boldsymbol{p}\, a \in E} \texttt{mass}(\boldsymbol{p}\, a), \qquad \text{for } \boldsymbol{p} \in P \setminus \mathcal{A}_{\texttt{EOS}} \tag{8.13b}$$

[10] Note that `EOS` is handled specially as it is not a string of characters.

Algorithm 8 Character proposal: This procedure implements a properly weighted proposal distribution for the unnormalized version of the locally constrained distribution $\ell_{\{\phi_{\mathcal{G}}\}}(\cdot \mid \boldsymbol{x})$.

procedure `character_proposal`(Γ)
  `mass` $\leftarrow$ Apply Equation (8.13) to $p(\cdot \mid \Gamma)$
  $\boldsymbol{p} \leftarrow \varepsilon$ ▷ start at the trie's root node
  $c \leftarrow 1$ ▷ path weight under cfg
  $w \leftarrow \{\}$
  $S \leftarrow \emptyset$
  $\pi \leftarrow \{\}$
  $\pi(\varepsilon) \leftarrow 1$
  while true **do**
    $p_1 \leftarrow \left\{a : \frac{\texttt{mass}(\boldsymbol{p}\,a)}{\texttt{mass}(\boldsymbol{p})} \text{ for } \boldsymbol{p} \xrightarrow{a} \boldsymbol{p}\,a \in E\right\}$
    $p_2 \leftarrow \left\{a : \frac{\phi_{\mathcal{G}}(\Gamma\,\boldsymbol{p}\,a)}{\phi_{\mathcal{G}}(\Gamma\,\boldsymbol{p})} \text{ for } a \in \Sigma\right\}$
    if $\boldsymbol{p} \xrightarrow{\texttt{EOT}} \boldsymbol{p}\,\texttt{EOT} \in E$ **then** ▷ End-of-token available (i.e., $\boldsymbol{p} \in \mathcal{A}_{\texttt{EOS}}$)
      $w(\boldsymbol{p}) = \frac{\texttt{mass}(\boldsymbol{p}\,\texttt{EOT}) \cdot c}{\pi(\boldsymbol{p})}$
      $S \leftarrow S \cup \{\boldsymbol{p}\}$
    $\overline{q} \leftarrow \{a : p_1(a) \cdot p_2(a) \text{ for } a \in \Sigma\}$ ▷ Note: $\texttt{EOT} \notin \Sigma$.
    $Q \leftarrow \sum_{a \in \Sigma} \overline{q}(a)$
    if $Q = 0$ **then** ▷ cannot continue further
      break
    $a \sim \overline{q}/Q$ ▷ Sample next character proportional to $\overline{q}$
    $\pi(\boldsymbol{p}\,a) \leftarrow \pi(\boldsymbol{p}) \cdot \overline{q}(a)/Q$
    $\boldsymbol{p} \leftarrow \boldsymbol{p}\,a$ ▷ extend the prefix (i.e., transition to the next node)
    $c \leftarrow c \cdot p_2(a)$
  $W \leftarrow \sum_{x' \in S} w(x')$
  $x \sim w(\cdot)/W$
  return (x, W)

It is straightforward to verify that the character proposal is an instance of set-based proposal speedup (Definition 25), which is properly weighted according to Proposition 2.

8.4 Estimating Radon-Nikodym derivatives for adaptive weighted rejection sampling proposals

In this section, we develop another approach to properly weighted sampling from the local posterior $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$. Unlike the approach in the previous section, this section's method draws *exact* samples from $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$, together with unbiased estimates of the local normalizing constant $L_{\text{eff}}(\boldsymbol{x}_{<t})$, without enumerating the entire vocabulary. Indeed, it is fast enough that we can typically treat all our potentials as efficient potentials, rather than expensive potentials, when using this method. However, to

apply it, we must assume that our potentials Φ encode binary constraints, taking values in $\{0, 1\}$.

Simple rejection sampling. Simple rejection sampling is an algorithm that can provide exact samples from $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$ without looping through the entire token vocabulary. It works by drawing a token $x \sim p(\cdot \mid \boldsymbol{x}_{<t})$, then checking whether the token satisfies $\Phi(\boldsymbol{x}_{<t}x)$. If it does, the token is returned; otherwise, this process is repeated. The returned token is an exact sample from $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$.

Compared to the constant cost of token masking, this algorithm is a **Las Vegas algorithm**, i.e., an exact algorithm with a stochastic runtime. It is easy to show that the expected number of samples drawn by the algorithm before meeting the constraint is $\frac{1}{L_{\text{eff}}(\boldsymbol{x}_{<t})}$; it can furthermore be shown that $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right) = -\log L_{\text{eff}}(\boldsymbol{x}_{<t})$ [Levy, 2008, Freer et al., 2010], and thus that the expected runtime is exponential in $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right)$. When the constraint is relatively easy to satisfy (i.e., when $L_{\text{eff}}(\boldsymbol{x}_{<t}) \approx 1$ so $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right)$ is low), this can lead to runtimes that are much faster than those required by full-vocabulary token masking. However, when Z is very small ($L_{\text{eff}}(\boldsymbol{x}_{<t}) < |\mathcal{A}|^{-1}$), rejection sampling's runtime can be even worse than that of token masking.

Adaptive rejection sampling (ARS). Adaptive rejection sampling [Gilks and Wild, 1992, Mansinghka et al., 2009a] is a version of rejection sampling that is never slower than token masking and often significantly faster. In ARS, we adaptively *remove* each invalid token encountered while rejection sampling, so that we never sample the same rejected token twice. This simple modification can dramatically improve the runtime of the algorithm.

Both rejection sampling approaches evaluate $\Phi(\boldsymbol{x}_{<t}x)$ only on the samples x they draw and, thus, can be more efficient than token masking. However, neither provides a direct way to exactly compute $L_{\text{eff}}(\boldsymbol{x}_{<t})$, the weight we need to use in SMC. But as discussed in the previous section and in Section 4.2, it suffices to use independent unbiased estimates of each normalizing constant $L_{\text{eff}}(\boldsymbol{x}_{<t})$, rather than the exact quantities, as correction factors.

Estimating the necessary Radon-Nikodym derivatives. In lieu of the exact but expensive correction factors obtained as a byproduct of token masking, we wish to find cheap, unbiased estimates of $L_{\text{eff}}(\boldsymbol{x}_{<t})$ that we can compute as a byproduct of simple or adaptive rejection sampling.

Our first observation toward this goal is that the *number of rejected tokens* sampled before generating a valid next token x contains signal about the magnitude of $L_{\text{eff}}(\boldsymbol{x}_{<t})$: when many rejected tokens are generated, this suggests $L_{\text{eff}}(\boldsymbol{x}_{<t})$ is small, and can serve as a sign that the current prefix $\boldsymbol{x}_p$ has landed us in a low-probability region of the global posterior g.

Indeed, in simple rejection sampling, the number of trials $n_0 + 1$ (that is, n_0 rejections plus the final success) is an unbiased estimate of $1/L_{\text{eff}}(\boldsymbol{x}_{<t})$. Unfortunately, $1/(n_0 + 1)$

is *not* an unbiased estimate of $L_{\text{eff}}(\boldsymbol{x}_{<t})$, and we require unbiased estimates of $L_{\text{eff}}(\boldsymbol{x}_{<t})$ for sound use within SMC. The bias is even worse in *adaptive* rejection sampling, because even when $L_{\text{eff}}(\boldsymbol{x}_{<t})$ is small we may sample few rejections, limiting n_0's usefulness as a reliable source of information about $L_{\text{eff}}(\boldsymbol{x}_{<t})$.

In the next two subsections, we show how to derive unbiased estimates of $L_{\text{eff}}(\boldsymbol{x}_{<t})$. Although we present them in a stand-alone manner, these estimators were originally derived using the logic of Chapter 4's automated Radon-Nikodym derivative estimation.

8.4.1. Warm-up: Weighted rejection sampling (WRS)

In the simple rejection sampling setting, we can collect more data about $L_{\text{eff}}(\boldsymbol{x}_{<t})$ by running $L \geq 1$ *additional* rejection loops. In addition to reducing variance, this also now yields an *unbiased* estimate of $L_{\text{eff}}(\boldsymbol{x}_{<t})$. The total number of trials T (rejections and successes) required to reach $L+1$ successes follows a negative binomial distribution with parameters $L_{\text{eff}}(\boldsymbol{x}_{<t})$ and $L+1$. Letting $n = \sum_{i=0}^{L} n_i$ be the total number of rejections across the $L+1$ loops,

$$\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})} \overset{\text{def}}{=} \frac{L}{T-1} = \frac{L}{(n+(L+1))-1} = \frac{L}{n+L} \tag{8.14}$$

is a known, unbiased estimator for the $L_{\text{eff}}(\boldsymbol{x}_{<t})$ parameter of the negative binomial distribution, provided that $L \geq 1$ [Forbes et al., 2011]. This allows us to define the following algorithm for jointly generating a next token x from $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$ and an unbiased estimate $\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}$ of $L_{\text{eff}}(\boldsymbol{x}_{<t})$.

Definition 28. Given an unnormalized target $L_{\text{eff}}(\boldsymbol{x}_{<t})\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$ as above, **weighted rejection sampling** generates $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}) \sim Q_{\text{WRS}}$ as follows:

1. Run rejection sampling to obtain a valid sample x: Sample $\langle r_1, \ldots, r_{n_0}, x\rangle \sim p(\cdot \mid \boldsymbol{x}_{<t})$ through n_0 rejections r_i until obtaining an accepted token x (i.e., such that $\Phi(\boldsymbol{x}_{<t}x) = 1$).
2. For a budget of $L \geq 1$ additional loops, repeat step 1 and count the number of rejections on each loop $n_1, \ldots, n_L$.
3. Calculate the estimate $\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})} \overset{\text{def}}{=} \frac{L}{n+L}$, where $n = \sum_{i=0}^{L} n_i$.
4. Return $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})})$

Proposition 3. *For* $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}) \sim Q_{WRS}$, *$x$ is distributed according to* $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$ *and* $\mathbb{E}[\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}] = L_{\text{eff}}(\boldsymbol{x}_{<t})$.

Proposition 4. *The expected runtime of* Q_{WRS} *scales with* $O(\frac{L}{L_{\text{eff}}(\boldsymbol{x}_{<t})})$.

Using these estimates, simple rejection sampling may be soundly integrated into SMC as a proposal distribution, supporting the correction of LCD's greediness.

$L = 1$ is enough to ensure unbiasedness, and we find it to work well in practice, but L can be increased to trade higher runtime for reduced variance.

8.4.2. Adaptive weighted rejection sampling (AWRS)

In the adaptive setting, the expected number of rejections is reduced, and the negative binomial estimator can no longer be used. However, an auxiliary-variable argument based on the framework of Lew et al. [2022b] can be used to derive an alternative formula for the adaptive case based on not just the number of rejections but also the probability mass removed from $p(\cdot \mid \boldsymbol{x}_{<t})$ during adaptation.

Definition 29. Given an unnormalized target $L_{\text{eff}}(\boldsymbol{x}_{<t})\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$, AWRS generates $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}) \sim Q_{\text{AWRS}}$ as follows:

1. Sample $\langle r_1, \ldots, r_{n_0}, x \rangle$ as follows: draw n_0 *unique* rejections r_i until obtaining x s.t. $\Phi(\boldsymbol{x}_{<t}x) = 1$. Note that beyond the first step, we do not sample from $p(\cdot \mid \boldsymbol{x}_{<t})$, but a re-normalized distribution on $\mathcal{A} \setminus \mathbf{r}_{<i}$.
2. Calculate $\psi_0 = \sum_{i=1}^{n_0} p(\cdot \mid \boldsymbol{x}_{<t})(r_i)$.
3. Generate one additional trace $\langle s_1, \ldots, s_{n_1}, x^* \rangle$, by continuing to sample as above from the remaining not-yet-rejected elements, through an additional n_1 new *unique* rejections, until finding an element x^*. Note that x^* could be the same as x, since acceptances are replaced (unlike rejections).
4. Calculate the estimate $\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})} \overset{\text{def}}{=} \frac{1-\psi_0}{n+1}$, where $n = n_0 + n_1$.
5. Return $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})})$

Proposition 5. *For $(x, \widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}) \sim Q_{AWRS}$, x is distributed according to $\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t})$ and $\mathbb{E}[\widehat{L_{\text{eff}}(\boldsymbol{x}_{<t})}] = L_{\text{eff}}(\boldsymbol{x}_{<t})$.*

As above, AWRS generates next tokens and correction factors suitable for use within SMC. In addition, AWRS offers considerable runtime benefits. Trivially, since we cannot resample rejections, we must succeed after sampling at most the number of invalid tokens, no matter how small $L_{\text{eff}}(\boldsymbol{x}_{<t})$ is. AWRS also has lower *expected* runtime. Intuitively, we may think of the time it takes to sample an acceptance as follows. Each non-conforming token may be considered a *distractor* if its *individual* mass is comparable to or higher than $L_{\text{eff}}(\boldsymbol{x}_{<t})$, the sum of *all* conforming tokens. Rather than all non-conforming tokens contributing equally, expected runtime is dominated by only these—typically rare—distractor tokens.

8.5 Empirical evaluation

We compare seven approaches to constrained generation:

1. *Language model (Base LM).* This method simply samples from the base language model p (see Section 8.5.1 for details on the language models used).

2. *Language model with grammar constraint (Locally constrained decoding).* This is the approach used by much prior work (Footnote 4). In each of our domains, we formulate a context-free grammar (CFG) encoding a notion of syntactic well-formedness appropriate for the domain (see Section 8.5.1). We let Φ_{eff} encode the binary function that determines whether its input is a prefix of some valid sequence in the grammar's language. This baseline directly samples from ℓ_{eff}, i.e., it uses per-token logit masking to greedily enforce the CFG constraint.
3. *Language model with grammar constraint and weight correction (Grammar-only IS).* This method generates particles from ℓ_{eff}, then computes importance weights to correct toward the global product of p and Φ_{eff}. These weights mitigate some of the greediness of local-product-of-experts sampling, but do not yet integrate any potentials beyond Φ_{eff}.
4. *Language model with grammar constraint, weight correction, and resampling (Grammar-only SMC).* This method is a straightforward application of Lew et al. [2023d] to locally constrained decoding and is similar to Park et al. [2025], which also attempts to correct for the greediness of locally constrained decoding. As in the previous method, it targets the global product of p and Φ_{eff} but uses resampling to reallocate computation to promising particles.
5. *Language model with grammar constraint and expensive potential (Sample-Rerank).* Sample-Rerank is a common family of approaches for incorporating an external signal into an LM's generations post-hoc, for instance by choosing the best-of-n particles via a reward model [Nakano et al., 2021, Krishna et al., 2022, Zhou et al., 2023, Gui et al., 2024, Mudgal et al., 2024, Ichihara et al., 2025] or filtering via a verifier [Olausson et al., 2023, Chen et al., 2024, Lightman et al., 2024, Xin et al., 2024]. In each domain, we formulate an additional potential Φ_{exp} that encodes task-specific signals of sequence quality (see Section 8.5.1). This baseline generates grammar-constrained sequences from the local product of experts, then reweights each sequence $\boldsymbol{x}$ by $\Phi_{\text{exp}}(\boldsymbol{x})$.
6. *Language model with grammar constraint, weight correction, and expensive potential (Full IS).* This is the full importance sampling method described in Section 8.2, with $\Phi = \Phi_{\text{eff}} \cup \Phi_{\text{exp}}$. Unlike in the previous method, the importance weights here include correction terms that mitigate the greediness of local sampling, targeting the global product g. We include this method primarily as an ablation of our next method (SMC), modified not to include incremental resampling.
7. *Language model with grammar constraint, weight correction, expensive potential, and resampling (Full SMC).* This method includes all of the algorithmic contributions of our approach. It is the full sequential Monte Carlo algorithm, with $\Phi = \Phi_{\text{eff}} \cup \Phi_{\text{exp}}$. It targets the same global posterior g as the previous method but uses resampling to reallocate computation to promising particles.

We report results using $N = 10$ particles; see Section 8.5.5 and Figure 8-2 for downstream accuracy results for a varying number of particles. We ran experiments on GCP instances with 1 A100 GPU and 12 vCPUs (our CFG parser is implemented for CPU and is parallelized across particles), with the exception of the Data Science domain, for which we used 4 H100 GPUs and 64 vCPUs.

Table 8.1: Summary of tasks and potential functions. Examples are truncated for brevity. Full prompts include additional information.

Task	Potentials		Examples	
	Φ_{eff}	Φ_{exp}	Prompt	Output
Goal Inference	STRIPS parser	Plan simulation	Write the STRIPS goal condition for the planning problem described below [...]. The STRIPS initial condition is: [...]	`(:goal (and (arm-empty) (on-table b1)` [...]
Python Data Science	-	Error-checking with test cases	Here is a sample dataframe: [...] I'd like to add inverses of each existing column to the dataframe [...]	`result = df.join(df.apply(lambda x: 1/x)` [...]
Text-to-SQL	SQL parser	Alias and table-column checking	Here is a database schema: [...] For each stadium, how many concerts are there?	`SELECT T2.name, COUNT(*) FROM concert AS T1` [...]
Molecular Synthesis	SMILES parser	Incremental molecule validation	Given the following list of molecules in SMILES format, write an additional molecule [...]	`CC1=CC2(OC=N)C(=O)` [...]

8.5.1. Domains

We study the performance of our proposed sampling methods on four challenging semantic parsing domains, summarized in Table 8.1.

- **Goal inference (Planetarium).** *Task:* Formally specify an agent's goal in the STRIPS subset of the PDDL planning language, based on a natural-language description of the goal and PDDL code detailing the agent's initial conditions and plan for achieving it. *Data:* Blocksworld tasks with up to 10 objects from the Planetarium benchmark [Zuo et al., 2024]. *Metric:* Accuracy with respect to ground-truth PDDL goal. *Base LM:* Llama 3.1 8B. *Grammar:* STRIPS syntax for goals within Planetarium Blocksworld's domain definition. *Expensive potential:* Run a simulation with a ground-truth plan and check whether the resulting state conforms to the predicted (partial) goal.

- **Python for data science (DS-1000).** *Task:* Generate Python code that uses standard data science libraries (NumPy, PyTorch, Pandas, etc.) to solve a task specified in natural language and via (executable) test cases. *Data:* DS-1000 benchmark [Lai et al., 2023]. *Metric:* Accuracy of the generated program with respect to the provided test cases. *Base LM:* Llama 3 70B. *Grammar:* We use a trivial potential $\Phi_{\text{eff}}(\boldsymbol{x}) = 1$, as we find that the unconstrained LM reliably generates grammatical Python (that may nonetheless induce runtime errors). *Expensive potential:* Given a partial program $\boldsymbol{x}$, Φ_{exp} truncates $\boldsymbol{x}$ to the longest prefix of the sequence that consists of only valid Python statements (discarding any incomplete material at the end), and executes the resulting (partial) program on the provided test case, checking for runtime errors.

Table 8.2: Comparison of method performance across domains with bootstrapped 95% confidence intervals. For brevity, *grammar constraint* and *weight correction* are abbreviated as *grammar* and *correction*, respectively.

Method	Score			
	Goal inference	Molecular synthesis	Data science	Text-to-SQL
Base LM	0.063 (0.05, 0.08)	0.132 (0.12, 0.15)	0.213 (0.19, 0.24)	0.531 (0.51, 0.55)
w/ grammar constraint (Locally constrained Decoding)	0.086 (0.07, 0.11)	0.189 (0.17, 0.21)	-	0.559 (0.54, 0.58)
w/ grammar, weight correction (Grammar-only IS)	0.083 (0.06, 0.11)	0.228 (0.21, 0.25)	-	0.597 (0.57, 0.62)
w/ grammar, potential (Sample-Rerank)	0.289 (0.24, 0.34)	0.392 (0.36, 0.42)	-	0.581 (0.56, 0.60)
w/ grammar, correction, and resampling (Grammar-only SMC)	0.401 (0.34, 0.46)	0.205 (0.18, 0.23)	-	0.596 (0.57, 0.62)
w/ grammar, potential, and correction (Full IS)	0.257 (0.21, 0.31)	0.404 (0.37, 0.44)	0.346 (0.31, 0.39)	**0.618** (0.59, 0.64)
w/ grammar, potential, correction, and resampling (Full SMC)	**0.419** (0.37, 0.48)	**0.577** (0.56, 0.59)	**0.407** (0.36, 0.45)	**0.620** (0.60, 0.64)

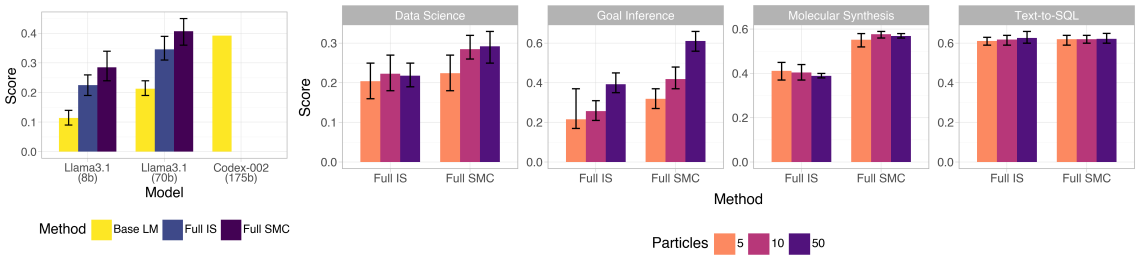


Figure 8-2: **Left:** Performance on the Data Science task (DS-1000) for different models and methods. Codex-002 performance as reported in Lai et al. [2023]. **Right:** Performance across all tasks for Full IS and Full SMC with 5, 10, and 50 particles. **Error bars:** bootstrapped 95% confidence intervals.

- **Text-to-SQL (Spider).** *Task:* Generate SQL queries from a natural language question and a database schema. *Data:* Spider development split [Yu et al., 2018]. *Metric:* Execution accuracy (whether the generated SQL query, when run against a test database, produces the same results as the ground-truth SQL query). *Base LM:* Llama 3.1 8B-Instruct. *Grammar:* SQL context-free grammars released by Roy et al. [2024], which enforce valid SQL syntax. *Expensive potential:* Check whether column names in the generated (partial) query actually belong to the queried tables, modulo aliasing. (The grammar ensures only that the column names exist in *some* table.)

- **Molecular synthesis (GDB-17).** *Task:* Generate drug-like molecules in the SMILES format [Weininger, 1988]. *Data:* Few-shot prompts constructed by repeatedly choosing 20 random examples from the GDB-17 dataset [Ruddigkeit et al., 2012]. *Metric:* Quantitative Estimate of Drug-likeness [QED; Bickerton et al., 2012], a standard molecular fitness function. *Base LM:* Llama 3.1 8B. *Grammar:* SMILES syntax for molecules. *Expensive potential:* A SMILES prefix validator implemented in the Python *partialsmiles* library [O'Boyle, 2024].

8.5.2. Evaluation of downstream performance

We begin by investigating whether our approach leads to significant performance gains. Table 8.2 reports posterior-weighted accuracy for our approach and ablations

of its components: grammar constraints, weight corrections, expensive potentials, and resampling. We first summarize the observed effects of each component in our approach:

Grammar constraints. In line with previous literature [e.g., Shin et al., 2021, Scholak et al., 2021, Poesia et al., 2022, Wang et al., 2024a], we find that the addition of a grammar constraint via Φ_{eff} improves downstream accuracy relative to the base LM across all domains in which it is used, even without the use of weight corrections.

Expensive potentials. Furthermore, we observe that integrating expensive potentials Φ_{exp} improves accuracy in models. Even without any weight corrections, the improvement in the goal inference, data science, and molecular synthesis domains is large; in the text-to-SQL domain, it is smaller but statistically significant (paired permutation test, $p < 0.01$). This suggests that making use of information that cannot be efficiently encoded in logit masks can greatly improve performance.

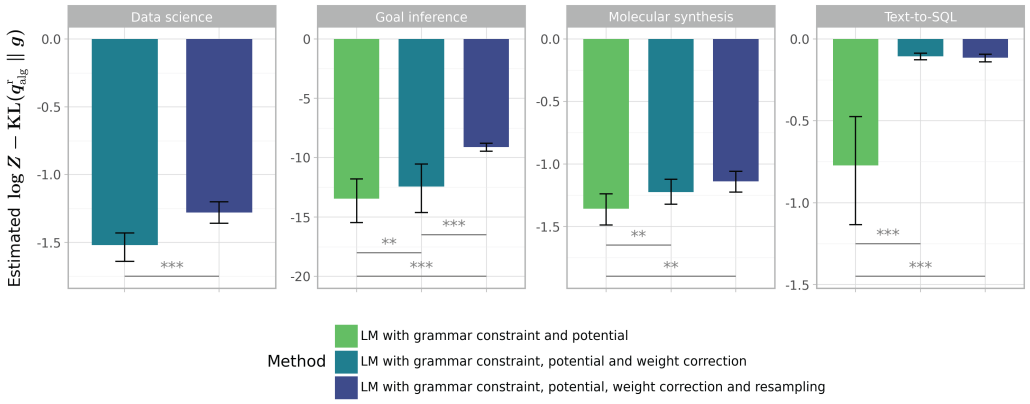
Weight corrections. Although the use of Φ_{eff} and Φ_{exp} alone leads to significant gains in downstream accuracy, these gains can be amplified with the addition of weight corrections. In cases without the expensive potential, weight corrections provide significant albeit relatively small gains in accuracy across three domains; in goal inference, it does not significantly affect performance. In the presence of the expensive potential, adding weight corrections improves accuracy for text-to-SQL and has no effect on goal inference and molecular synthesis. Overall, these results indicate that debiasing samples from a local product of experts to correctly target the global product of experts often significantly improves downstream accuracy and never harms it. That said, the accuracy gains attributable to weight corrections are modest compared to other components of the algorithm, which suggests that the bias from locally constrained decoding may be less severe in these semantic parsing domains than has been observed in other domains [e.g., constrained generation of natural language, Lew et al., 2023d].

Resampling. We observe that the addition of resampling steps improves downstream accuracy in all domains except text-to-SQL, for which they neither significantly improve nor hurt performance. These results motivate adaptively focusing computation on promising partial sequences.

Other evaluations. Next, we study the effects of varying the base language model, the number of particles used by different methods, and the computational cost of our approach: Tables 8.4, 8.5 and 8.7 report the results of these experiments. We summarize key findings:

- **Our approach allows smaller LMs to outperform larger ones:** In 3 out of 4 domains (Data Science, Molecular Synthesis, Goal Inference), Full SMC allows small language models to outperform models over 8 times larger (see Tables 8.2 and 8.4). These gains persist on larger models: Figure 8-2 shows how our method allows Llama 3.1 70b to outperform Codex-002, which has 175b parameters and

Figure 8-3: Estimated KL between the algorithm and the global product of experts for a representative problem instance in each domain. Values closer to 0 indicate that the algorithm is better at approximating g. Significant differences are indicated with ** for $p < 0.01$ and *** for $p < 0.001$ (t-test). Algorithms use $N = 10$ particles.



is fine-tuned for coding tasks.

- **Our approach makes better use of resources than approaches that apply constraints only at the end of generation:** In 3 out of 4 domains (Data Science, Molecular Synthesis, Text-to-SQL), Full SMC performs as well as or better than Full IS while using one-tenth of the particles (see Figure 8-2 and Table 8.5); in the remaining domain (Goal Inference), Full SMC outperforms IS with one fifth (10 vs 50) or one half (5 vs 10) of the particles. This is in line with the arguments drawn in Section 8.2 for the poor scaling of importance sampling and the benefits of resampling.

- **Our approach incurs minimal computational overhead:** At every token, our SMC approach incurs two computational overheads relative to a simple locally constrained decoding baseline: resampling and computing expensive potentials. Though the cost of resampling is negligible, computing expensive potentials presents a more significant cost that varies across domains: Table 8.7 shows that cost rarely rises above ~30ms per token. In general, this cost is reduced by two factors: (i) expensive potentials often need to run expensive computations only at larger, semantically meaningful units (for instance, the end of a SQL clause or a Python statement) rather than at every token—therefore significantly lessening the average cost per token, (ii) expensive potentials operations are often performed on CPU rather than GPU, and therefore cost fewer dollars per hour.

8.5.3. Validation of the probabilistic perspective

The best-performing methods from the previous section were designed to approximate the global product of experts distribution. In this section, we investigate how closely each of these methods approximates this global distribution and whether the downstream performance results from the previous section are driven by the

quality of the probabilistic inference. In particular, we find:

Within each problem instance, the best-performing methods have outputs that are closer in KL divergence to the global product of experts. We consider the distribution over sequences $q^{\mathrm{r}}_{\mathrm{alg}}(\boldsymbol{x})$ defined by each algorithm. For each $q^{\mathrm{r}}_{\mathrm{alg}}$, we estimate a tractable correlate of the KL between the algorithm and the global product of experts: $\log Z - \mathrm{KL}(q^{\mathrm{r}}_{\mathrm{alg}} \parallel g)$. We refer to this quantity as the *approximation quality*. Since the term $\log Z$ is algorithm-independent, we can directly compare the estimated approximation quality across algorithms to determine which ones have lower KL divergence relative to the global product of experts. However, because $\log Z$ is instance-specific, these comparisons can only be made at the instance level. Accordingly, for each domain, we select the instance with the median unique accuracy as a representative example. Figure 8-3 visualizes estimated approximation quality on these examples across all methods, which include Φ_{exp}. Estimates were computed across 100 runs of each algorithm.

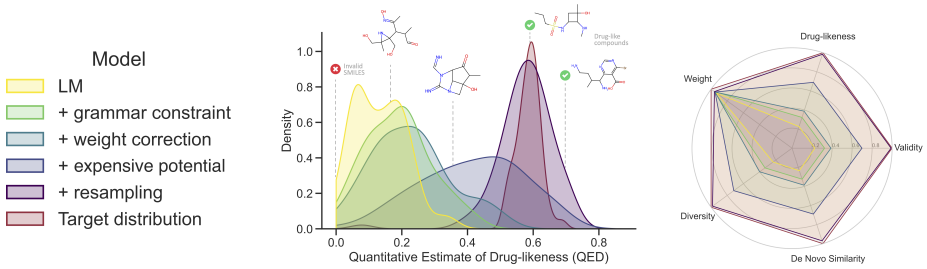
In all domains, sampling from the local product of experts without weight correction leads to significantly lower approximation quality relative to the methods that approximate the global product. The addition of resampling steps also significantly improves approximation quality in the data science and goal inference domains, but does not significantly change quality in the molecular synthesis and text-to-SQL domains. These trends in approximation quality are consistent with those observed in our evaluation of downstream accuracy: for example, we find that text-to-SQL is the domain in which weight corrections led to the most significant improvement in approximations of the global posterior, as well as the domain in which weight corrections most improve downstream performance. This suggests that the probabilistic formulation of the problem leads to practical gains in performance. Furthermore, these benefits can extend beyond our main performance metric; for instance, resampling during molecular generation yields simultaneous improvements along a number of additional dimensions of interest, including de-novo similarity and diversity (Figure 8-4).

Across problem instances, the best-performing methods assign probabilities that are more correlated with downstream performance In each of our experiments, we group output particles by *semantic equivalence*, and estimate the probability of each equivalence class under the method's approximation to the global product of experts, by summing the normalized weights of the members of each equivalence class [this is similar to the postprocessing performed in Shi et al., 2022]. We then measure the correlation between the estimated probability of a result and its score on the task-specific metric. Table 8.3 shows sequential Monte Carlo overall exhibits high correlation between (approximate) posterior probabilities and downstream performance, and that the differences in correlation between methods closely track the differences in performance in Section 8.5.2. In the goal inference, molecular synthesis, and data science domains, where expensive potentials and resampling greatly increase performance, we find that the same features also result in higher

Table 8.3: Pearson correlation between relative particle weights and accuracy scores for all weighted methods. Greater correlation indicates that relative weights are more strongly associated with downstream performance.

Method	Correlation between relative weight and score			
	Goal inference	Molecular synthesis	Data science	Text-to-SQL
LM with grammar constraints and weight correction (Grammar-Only IS)	0.138 (0.10, 0.18)	0.218 (0.16, 0.28)	0.217 (0.18, 0.26)	0.810 (0.79, 0.83)
LM with grammar constraints, potential, and weight correction (Full IS)	0.677 (0.64, 0.71)	0.570 (0.53, 0.61)	0.289 (0.25, 0.33)	0.796 (0.78, 0.81)
LM with grammar constraints, potential, weight correction, and resampling (Full SMC)	0.793 (0.76, 0.82)	0.826 (0.81, 0.84)	0.370 (0.31, 0.42)	0.810 (0.79, 0.83)

Figure 8-4: Distributional properties of compounds generated by different methods. **Middle:** Distribution of drug-likeness as measured by QED score [Bickerton et al., 2012]. **Right:** Means for other properties of interest such as diversity and de novo similarity.



correlation between result probability and performance, whereas in text-to-SQL, where the performance gains are slimmer, we find that all methods correlate and score equally well. Together, these results validate the probabilistic approach, suggesting that the global posterior captures semantically meaningful uncertainty.

8.5.4. Smaller base LMs

This section evaluates downstream accuracy across methods using smaller base language models (relative to Table 8.2). For the Text-to-SQL, Molecular Synthesis, and Goal Inference domains, which in the Section 8.5.2 experiments used Llama 3.1 (8B), we substitute Llama 3.2 (1B). In the Data Science domain, which used Llama 3 (70B) in the Section 8.5.2 experiments, we substitute Llama 3.1 (8B). All experiments were run with $N = 10$ particles, and the instruct version of Llama 3.2 (1B) was used in the text-to-SQL domain to remain consistent with the model variants used in earlier experiments.

We report posterior-weighted accuracy using the smaller LMs across all methods and domains in Table 8.4. Although accuracy is significantly lower compared to the larger LMs, we find that weight corrections, expensive potentials, and resampling steps still improve model performance. We also find that, in general, the relative gains in accuracy provided by our method are more pronounced for smaller

Table 8.4: Downstream accuracy of different methods with a **smaller** base language model (Llama 3.1 8B in Data science and Llama 3.2 1B in all other domains). Errors are bootstrapped 95% confidence intervals. Instruct model is used for Text-to-SQL.

Method	Score			
	Goal inference	Molecular synthesis	Data science	Text-to-SQL
LM	0.012 (0.01, 0.02)	0.032 (0.02, 0.04)	0.114 (0.09, 0.14)	0.224 (0.207, 0.241)
w/ grammar constraint (Locally constrained Decoding)	0.046 (0.03, 0.06)	0.031 (0.02, 0.04)	-	0.250 (0.232, 0.270)
w/ grammar, weight correction (Grammar-only IS)	0.037 (0.02, 0.06)	0.041 (0.03, 0.05)	-	0.301 (0.281, 0.323)
w/ grammar, potential (Sample-Rerank)	0.087 (0.06, 0.12)	0.119 (0.09, 0.16)	-	0.299 (0.278, 0.321)
w/ grammar, correction, and resampling (Grammar-only SMC)	0.052 (0.03, 0.08)	0.050 (0.04, 0.06)	-	0.302 (0.281, 0.324)
w/ grammar, potential, and correction (IS)	0.079 (0.05, 0.11)	0.122 (0.09, 0.16)	0.225 (0.19, 0.26)	**0.348** (0.326, 0.372)
w/ grammar, potential, correction, and resampling (SMC)	**0.125** (0.09, 0.16)	**0.517** (0.48, 0.55)	**0.285** (0.24, 0.34)	**0.348** (0.325, 0.374)

language models. With the exception of Text-to-SQL, we observe that our approach with the smaller LM outperforms the locally constrained decoding baseline (LM w/ grammar constraint) using the larger LM. In the Data Science domain, our Full SMC approach with the smaller LM outperforms the larger base LM. These results suggest that our approach can dramatically improve the performance of smaller LMs.

8.5.5. Accuracy by number of particles

This section investigates how performance improvements vary with the number of particles. Table 8.5 reports downstream accuracy for $N = 5$, $N = 10$, and $N = 50$ particles using the Llama 3.1 (8B) models. Note that we only include methods in which samples are generated from an approximation that is constructed from a set of importance-weighted particles. For the base LM and locally constrained decoding baselines, samples are generated through direct ancestral sampling. As a result, the number of particles does not influence accuracy in these cases (though additional particles can provide a better estimate of the true model accuracy), so we omit these methods from the analysis.

The main effect we observe is the more efficient use of computational resources by Full SMC compared to methods that do not incorporate incremental information, such as Full IS: the former outperforms the latter with one tenth of the particles in 3 out of 4 domains (Data Science, Molecular Sythesis, Text-to-SQL) and one fifth of the particles in the other domain (Goal Inference, see Figure 8-2 in the main text for a visualization). We note an additional patterns of results: in the Text-to-SQL and Molecular synthesis domains, increasing the number of particles has a marginal impact on downstream accuracy; however, in Goal inference and Data Science, we observe that a greater number of particles can lead to significantly better downstream accuracy (though only when increasing from 5 to 10 particles in the Data Science domain). Given that Goal Inference and Data Science are the two tasks where our expensive potentials are most informative, this pattern of results seems to be reflective of the fact that richer potentials require more computation to fully exploit.

Table 8.5: Accuracy by number of particles across methods. Errors are bootstrapped 95% confidence intervals. Llama 3.1 8B is used as the base LM for all domains. Instruct model is used for Text-to-SQL.

Method	Score			
	Goal inference	Molecular synthesis	Data science	Text-to-SQL
		5 Particles		
LM w/ grammar constraint, correction (Grammar-only IS)	0.106 (0.08, 0.14)	0.239 (0.21, 0.27)	-	0.587 (0.56, 0.61)
LM w/ grammar constraint, potential (Sample-Rerank)	0.214 (0.17, 0.26)	0.407 (0.36, 0.45)	-	0.578 (0.55, 0.60)
LM w/ grammar constraint, correction, and resampling (Grammar-only SMC)	0.310 (0.26, 0.37)	0.209 (0.18, 0.24)	-	0.599 (0.57, 0.62)
LM w/ grammar constraint, potential, and correction (Full IS)	0.216 (0.17, 0.27)	0.411 (0.37, 0.45)	0.204 (0.16, 0.25)	**0.611** (0.59, 0.63)
LM w/ grammar constraint, potential, correction, and resampling (Full SMC)	**0.319** (0.27, 0.37)	**0.552** (0.52, 0.58)	**0.224** (0.18, 0.27)	**0.620** (0.59, 0.64)
		10 Particles		
LM w/ grammar constraint, weight correction (Grammar-only IS)	0.083 (0.06, 0.11)	0.228 (0.21, 0.25)	-	0.597 (0.57, 0.62)
LM w/ grammar constraint, potential (Sample-Rerank)	0.289 (0.24, 0.34)	0.392 (0.36, 0.42)	-	0.581 (0.56, 0.60)
LM w/ grammar constraint, correction, and resampling (Grammar-only SMC)	0.401 (0.34, 0.46)	0.205 (0.18, 0.23)	-	0.596 (0.57, 0.62)
LM w/ grammar constraint, potential, and correction (Full IS)	0.257 (0.21, 0.31)	0.404 (0.37, 0.44)	0.223 (0.19, 0.27)	**0.618** (0.59, 0.64)
LM w/ grammar constraint, potential, correction, and resampling (Full SMC)	**0.419** (0.37, 0.48)	**0.577** (0.56, 0.59)	**0.285** (0.26, 0.32)	**0.620** (0.60, 0.64)
		50 Particles		
LM w/ grammar constraint, correction (Grammar-only IS)	0.069 (0.05, 0.09)	0.211 (0.20, 0.22)	-	0.603 (0.58, 0.63)
LM w/ grammar constraint, potential (Sample-Rerank)	0.416 (0.36, 0.47)	0.382 (0.37, 0.40)	-	0.585 (0.56, 0.61)
LM w/ grammar constraint, correction, and resampling (Grammar-only SMC)	0.595 (0.54, 0.65)	0.212 (0.20, 0.23)	-	0.599 (0.58, 0.62)
LM w/ grammar constraint, potential, and correction (Full IS)	0.393 (0.35, 0.45)	0.389 (0.38, 0.40)	0.218 (0.19, 0.25)	**0.626** (0.60, 0.66)
LM w/ grammar constraint, potential, correction, and resampling (Full SMC)	**0.611** (0.56, 0.66)	**0.569** (0.56, 0.58)	**0.292** (0.25, 0.33)	**0.622** (0.60, 0.65)

Table 8.6: Downstream accuracy comparison with the SMC Steering method from Lew et al. [2023d] in the text-to-SQL domain. Errors are bootstrapped 95% confidence intervals. Both methods include expensive potentials. Our method is run with 10 particles. SMC Steering is run with 5 particles and a beam size of 3. Both methods are run with Llama 3.1 8B Instruct.

Method	Score
Full SMC	**0.620** (0.60, 0.64)
SMC Steering [Lew et al., 2023d]	0.607 (0.58, 0.63)

8.5.6. Resampling without replacement

This section evaluates our approach using the without-replacement resampling method introduced in Lew et al. [2023d]. Specifically, we use our Full SMC algorithm with expensive potential (LM w/ grammar constraint, potential, correction, and resampling), and replace multinomial resampling steps with Lew et al. [2023d]'s without replacement scheme. For comparison, we ran the without replacement baseline (SMC Steering) with $N = 5$ particles and a beam size of 3, alongside our approach using multinomial resampling with $N = 10$ particles (and an ESS threshold of 0.9). These settings effectively give the SMC Steering method a particle count of $N = 15$, giving it an advantage in the comparison.

Table 8.6 reports weighted accuracy for these methods in the text-to-SQL domain (we restricted this analysis to a single domain because of limitations in computational resources). We observe that without-replacement resampling steps slightly hurt performance compared to multinomial resampling.

Table 8.7: Average per token cost (in seconds) of computing the expensive potential Φ_{exp} for each of our domains. Intervals are bootstrapped confidences estimated by selecting 10 SMC generations at random for each domain.

Method	Goal Inference	Molecular Synthesis	Data Science	Text-to-SQL
Φ_{exp} seconds per token	0.011 (0.007, 0.016)	0.0003 (0.0002, 0.0004)	0.007 (0.0009, 0.023)	0.031 (0.0204, 0.0413)

8.5.7. Computational cost

Though we have shown that practitioners can improve over locally constrained decoding by using our proposed SMC method, in practice, there is additional computational cost stemming from two sources: resampling and computing expensive potentials Φ_{exp}. The cost of resampling is negligible, consisting only of simple sum, softmax, and categorical sampling operations at every token. The cost of computing expensive potentials, on the other hand, is more significant and varies across domains. Table 8.7 shows the average per token cost of computing expensive potentials for all of our domains: we see that it rarely goes above about 30ms.

In general, the computational cost of expensive potentials is lessened by two factors: 1) expensive potentials often change not at every token, but only at larger, semantically meaningful units (for instance, the end of a SQL clause or a Python statement)—caching can therefore significantly lessen computational cost, 2) expensive potentials are often CPU rather than GPU computations (and so the cost of computation is much cheaper).

8.5.8. Value of adaptive weighted rejection sampling (AWRS)

We now measure the practical impact of AWRS on accuracy and runtime for 5 tasks in different domains.

We compare AWRS to strong baselines through consideration of two versions: AWRS-LCD, which performs simple locally constrained decoding but using *unweighted* adaptive rejection sampling to speed up the algorithm, and AWRS-SMC, which uses the weighted version within SMC.

Methods. We first compare the following *uncorrected* methods in terms of both runtime and downstream task accuracy. Methods in this section yield $N = 1$ unweighted samples:

- **Base language model (Base LM).** Sample sequences of tokens from the language model (without forcing any constraints), i.e., sample $\boldsymbol{x} \sim p$.
- **Locally constrained decoding with token masking (LCD).** The standard approach to constrained decoding. We mask the entire token vocabulary, re-normalize, and sample.[11]

[11] Due to the high cost of full token masking, we only include this baseline for one benchmark, from which we illustrate our orders-of-magnitude speed-up.

- **Locally constrained decoding with adaptive rejection sampling (AWRS-LCD).** A faster implementation of LCD: rather than masking the entire vocabulary, we draw a sample from the same LCD distribution using ARS. (We do not yet use an importance-weight correction, so we run only the first rejection loop.)

The baselines above allow us to gauge the degree to which adaptive rejection sampling improves runtime. Our next set of methods go beyond LCD, returning a weighted ensemble of N strings such that the expected weight of $\boldsymbol{x}$ in the ensemble is $p(\boldsymbol{x}) \cdot \Phi(\boldsymbol{x}) \propto g(\boldsymbol{x})$.

- **Sample-Verify.** Sample N complete strings from the LM and weight them according to Φ (which amounts to discarding strings that fail to satisfy the condition).[12]
- **Sequential Monte Carlo with constraint as twist (Twisted SMC).** Sample tokens directly from the LM, but use $\Phi(\boldsymbol{x}_{<t}x)$ as a **twist function** to filter partial sequences after they have been extended with a token x. Note that this is a programmable twist, rather than a learned twist [Naesseth et al., 2019b, Lawson et al., 2022].
- **Sequential Monte Carlo with AWRS proposal (AWRS-SMC).** Use the AWRS algorithm as a proposal distribution for SMC. Like AWRS-LCD, this method generates tokens using an adaptive rejection sampling loop, but *does* calculate the correction factor.

Metrics.

- **Accuracy.** The accuracy of a returned string is defined by the benchmark. For methods that construct a weighted ensemble, we report the expected accuracy of a system that returns a random string from this ensemble (with probability proportional to its weight).[13]
- **Runtime.** The average number of seconds it takes to generate the N complete strings.[14]

Benchmarks. We use some of the same benchmarks as in previous evaluations, and some new ones:

- **Text-to-SQL (Spider).** *Task:* Generate SQL queries from a natural language question paired with its corresponding database schema. *Data:* Development

[12]This baseline is a common approach for incorporating constraints into an LM generation pipeline [Cobbe et al., 2021, Hendrycks et al., 2021, Nakano et al., 2021, Ahn et al., 2022, Shi et al., 2022, Uesato et al., 2022, Olausson et al., 2023, Lightman et al., 2023, Ankner et al., 2024, Gandhi et al., 2024, Wang et al., 2024b, Zhang et al., 2024].

[13]The rationale is that the probability of returning $\boldsymbol{x}$ then approaches $g(\boldsymbol{x})$ as N grows, so we approximately return $\boldsymbol{x} \sim g$, just as Base LM returns $\boldsymbol{x} \sim p$. Note that we could plausibly improve accuracy further by selecting the most probable string from the ensemble, or more generally, by the **minimum Bayes risk** method of selecting or constructing a "consensus string" with low expected task loss under the weighted ensemble.

[14]Our runtimes scale sublinearly in N because we use parallel hardware (a GPU). Specifically, the calls to obtain the next-token distribution $p(\cdot \mid \boldsymbol{x}_{<t})$ from the LLM are batched over the N strings.

split of the Spider dataset [Yu et al., 2018]. *Metric:* Execution accuracy (checking if the produced SQL query, when executed on a test database, yields the same results as the ground-truth query). *Base LM:* Llama 3.1 8B-Instruct. *Constraint function:* A Python parser for the SQL context-free grammars provided by Roy et al. [2024] to enforce syntactically valid SQL.

- **JSON.** *Task:* Generate documents that conform to a specific JSON Schema. *Data:* The validation splits of the Github-trivial, -easy and -medium tasks in the JSONSchemaBench dataset [Geng et al., 2025]. *Metric:* Whether a valid JSON document conforming to the schema is generated. *Base LM:* Llama 3.1 8B-Instruct. *Constraint function:* Check the output parses as JSON, and validate using the Python jsonschema library. Parsing is done with a streaming JSON parser, which allows incremental detection of some schema violations before the full document has been generated.
- **Goal inference (Planetarium).** *Task:* Formally define an agent's goal within the STRIPS subset of the PDDL planning language, using a natural-language description of the goal alongside PDDL code that specifies the agent's starting conditions and plan to reach it. *Data:* Blocksworld tasks featuring up to 10 objects from the Planetarium benchmark [Zuo et al., 2024]. *Metric:* Equivalence to the ground-truth PDDL description. *Base LM:* Llama 3.1 8B. *Constraint function:* Check STRIPS syntax for goals as defined in the Planetarium Blocksworld domain + execute a simulation using a ground-truth plan to verify if the resulting state matches the predicted (partial) goal.
- **Molecular Synthesis** *Task:* Produce drug-like compounds using the SMILES notation [Weininger, 1988]. *Data:* Few-shot prompts created by repeatedly selecting 20 random samples from the GDB-17 database [Ruddigkeit et al., 2012]. *Metric:* Quantitative Estimate of Drug-likeness [QED; Bickerton et al., 2012], a widely used measure of molecular quality. *Base LM:* Llama 3.1 8B. *Constraint function:* A SMILES prefix validator implemented via the Python *partialsmiles* library [O'Boyle, 2024].
- **Pattern matching.** *Task:* Generate strings that conform to expressive pattern-matching specifications. Compared to formal regular expressions, these patterns contain explicit features that cannot be fully captured by deterministic finite-state automata, including unbounded center embedding and conditionals. *Data:* Over 400 automatically generated pattern-matching specifications. *Base LM:* Llama 3.1 8B-Instruct. *Metric:* Adherence to the specified pattern. *Constraint function:* An incremental pattern validator that checks whether a complete match remains possible given a prefix [Barnett, 2014].

AWRS Outperforms State-of-the-Art Controlled Generation Methods. Table 8.8 shows the accuracy and runtime of each method in each domain. We observe the following results:

- **Controlled generation outperforms uncontrolled generation.** With little overhead to runtime, AWRS-LCD improves accuracy over Base LM across all bench-

Method	Accuracy	Runtime (sec/ex)
Base LM	0.530 (0.50, 0.56)	0.79 (0.76, 0.82)
AWRS-LCD	0.569 (0.54, 0.60)	1.07 (1.01, 1.12)
Sample-Verify	0.600 (0.58, 0.62)	2.76 (2.62, 2.91)
Twisted SMC	0.596 (0.57, 0.62)	2.89 (2.73, 3.06)
AWRS-SMC	0.608 (0.58, 0.63)	5.33 (5.05, 5.61)

Text-to-SQL

Method	Accuracy	Runtime (sec/ex)
Base LM	0.683 (0.64, 0.72)	2.37 (2.16, 2.59)
AWRS-LCD	0.781 (0.74, 0.82)	3.78 (3.40, 4.15)
Sample-Verify	0.845 (0.81, 0.88)	6.24 (5.74, 6.76)
Twisted SMC	0.866 (0.84, 0.90)	6.31 (5.74, 6.90)
AWRS-SMC	0.903 (0.87, 0.93)	10.51 (9.61, 11.44)

JSON

Method	Accuracy	Runtime (sec/ex)
Base LM	0.032 (0.01, 0.06)	1.07 (0.97, 1.17)
AWRS-LCD	0.18 (0.11, 0.26)	0.77 (0.68, 0.86)
Sample-Verify	0.205 (0.13, 0.28)	4.55 (4.25, 4.84)
Twisted SMC	0.479 (0.39, 0.57)	3.20 (2.93, 3.47)
AWRS-SMC	0.528 (0.44, 0.62)	2.62 (2.42, 2.82)

Goal Inference

Method	Accuracy	Runtime (sec/ex)
Base LM	0.150 (0.10, 0.20)	0.52 (0.50, 0.54)
AWRS-LCD	0.568 (0.53, 0.60)	0.58 (0.54, 0.62)
Sample-Verify	0.539 (0.50, 0.57)	1.96 (1.93, 1.99)
Twisted SMC	0.549 (0.52, 0.57)	2.04 (1.99, 2.09)
AWRS-SMC	0.568 (0.54, 0.59)	1.52 (1.47, 1.57)

Molecular Synthesis

Method	Accuracy	Runtime (sec/ex)	Method	Accuracy	Runtime (sec/ex)
Base LM	0.570 (0.52, 0.62)	0.10 (0.09, 0.11)	Sample-Verify	0.781 (0.74, 0.82)	0.28 (0.26, 0.30)
AWRS-LCD	0.993 (0.98, 1.00)	0.13 (0.11, 0.14)	Twisted SMC	0.796 (0.76, 0.84)	0.20 (0.19, 0.22)
LCD	0.978 (0.96, 0.99)	6.91 (5.68, 8.46)	AWRS-SMC	0.990 (0.98, 1.00)	0.36 (0.33, 0.40)

Pattern Matching

Table 8.8: Comparison of method accuracy and runtime across domains with 95% bootstrapped confidence intervals. Runtime represents the average execution time (in seconds) across all instances in the dataset. Sample-Verify and Twisted SMC were run with $N = 10$ particles. AWRS-SMC was run with $N = 5$ particles.

marks.

- **Adaptive sampling is much faster than token masking, with no loss of accuracy.** On the pattern matching domain — the only one where it was computationally feasible to run LCD — AWRS-LCD matches its accuracy while being $\sim 50\times$ faster.
- **Correcting for greediness improves accuracy.** AWRS-SMC always matches or beats AWRS-LCD, significantly improving it in three domains (Goal Inference, JSON, and Text-to-SQL). The other two domains (Molecular Synthesis and Pattern Matching) suffer somewhat less under greediness because their local constraints $\Phi(\boldsymbol{x}_{<t})$ are exact, allowing a prefix only if it has a valid continuation.
- **AWRS-SMC outperforms existing approaches that correct for greediness.** With half the number of particles, AWRS-SMC attains accuracy comparable to or higher than Sample-Verify and Twisted SMC in all benchmarks.

Next, we investigate how AWRS-SMC scales with the number of particles (Figure 8-5) and LM size (Figure 8-6), compared to existing methods that sample from the global distribution.

- **AWRS-SMC is faster than existing SMC approaches.** Across all domains, AWRS-SMC achieves higher accuracy with lower runtime than Twisted SMC. That difference is most extreme in the domains whose local constraints $\Phi(\boldsymbol{x}_{<t})$ are exact (Molecular Synthesis and Pattern Matching). In that case, 1-particle

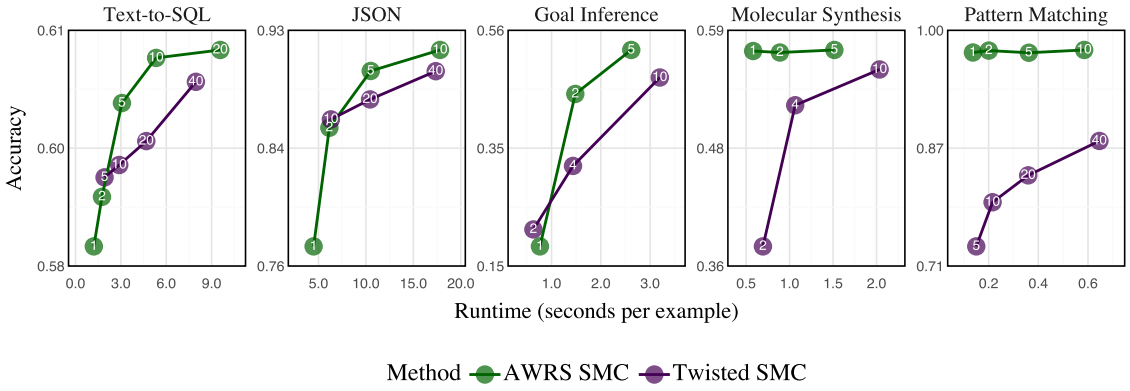


Figure 8-5: Runtime and accuracy by number of particles for AWRS-SMC and Twisted SMC.

Method	Accuracy	Runtime (sec/ex)
Base LM	0.159 (0.12, 0.20)	0.04 (0.04, 0.05)
AWRS-LCD	0.953 (0.93, 0.97)	0.07 (0.06, 0.08)
LCD	0.950 (0.93, 0.97)	8.45 (6.17, 11.39)
Sample-Verify	0.373 (0.33, 0.42)	0.20 (0.19, 0.21)
Twisted SMC	0.452 (0.41, 0.50)	0.11 (0.10, 0.13)
AWRS-SMC	0.974 (0.96, 0.99)	0.29 (0.26, 0.33)

Llama 3.2 1B

Method	Accuracy	Runtime (sec/ex)
Base LM	0.818 (0.78, 0.86)	0.22 (0.21, 0.24)
AWRS-LCD	0.993 (0.98, 1.00)	0.25 (0.23, 0.28)
LCD	0.990 (0.98, 1.00)	5.24 (4.50, 6.14)
Sample-Verify	0.858 (0.82, 0.89)	0.50 (0.46, 0.54)
Twisted SMC	0.846 (0.81, 0.88)	0.44 (0.41, 0.48)
AWRS-SMC	0.995 (0.99, 1.00)	0.50 (0.46, 0.55)

Llama 3.3 70B

Table 8.9: Comparison of method accuracy and runtime across language models of varying size on the Pattern Matching domain. Confidence intervals bootstrapped at the level of 95%. Runtime represents the average execution time (in seconds) across all instances in the dataset. Sample-Verify and Twisted SMC were run with $N = 10$ particles. AWRS-SMC was run with $N = 5$ particles. Llama 3.3 8B results are presented in Table 8.8. All models are instruct versions.

AWRS-SMC outperforms Twisted SMC with tens of particles. This suggests that the improvements in AWRS-SMC come from including the constraints in the proposal instead of having to guess and check.

- **AWRS-SMC with smaller LMs outperforms existing SMC approaches with larger LMs.** Figure 8-6 shows how AWRS-SMC using LLama 3.2 1B yields better runtime and accuracy than Twisted SMC using LLama 3.1 8B and Llama 3.3 70B. This suggests that including informative constraints in the proposal is a compute-efficient way to make small models punch above their weight.

AWRS Achieves Speedups by Allocating Computation Dynamically. AWRS scales with the difficulty of constraint conformance $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right)$, taking less time to sample a token when the constraint is expected and more time when it is not. We analyzed the LCD results of the pattern-matching benchmark, where token masking supports the exact calculation of the ground truth $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right)$ for each token sampling step. We then ran AWRS for each of these steps, illustrating

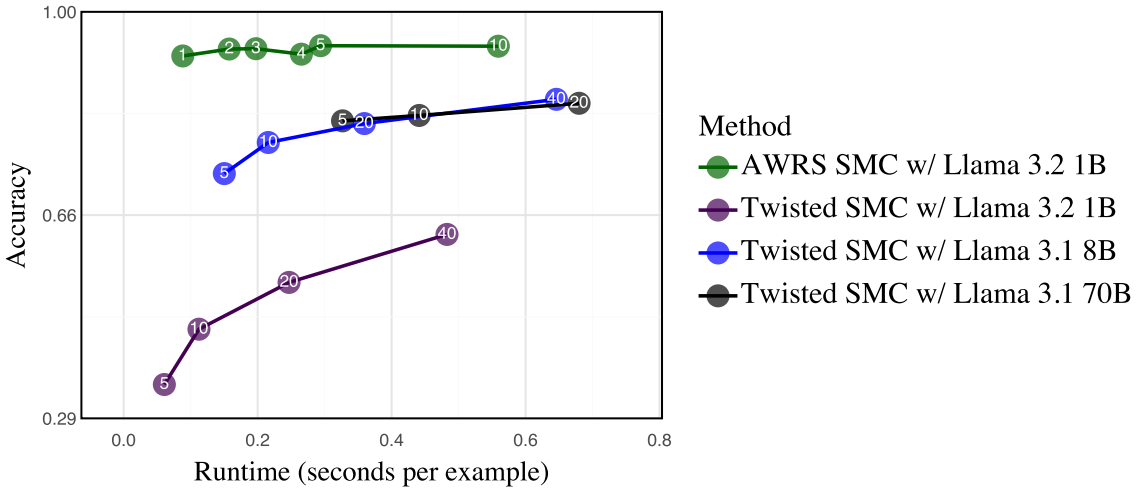


Figure 8-6: Accuracy and runtime of AWRS-SMC and Twisted SMC with varying particle counts on the Pattern Matching domain, using LMs of different sizes. AWRS-SMC with a smaller LM achieves better performance than Twisted SMC with larger LMs at the same runtime cost. All models are instruct versions.

a few key results (Figure 8-7).

1. $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \,\|\, p(\cdot \mid \boldsymbol{x}_{<t})\right)$ is small in most sampling steps; AWRS typically checks only 2 or 3 tokens.
2. As $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \,\|\, p(\cdot \mid \boldsymbol{x}_{<t})\right)$ increases for the hardest cases, the runtime of AWRS scales dynamically. An interesting consequence is that *AWRS samples faster for more accurate base models*.
3. As $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \,\|\, p(\cdot \mid \boldsymbol{x}_{<t})\right)$ grows, AWRS generally does not deteriorate. AWRS is roughly bounded by the number of non-conforming tokens whose individual probabilities are close to or exceed $L_{\text{eff}}(\boldsymbol{x}_{<t})$. This set is typically small, and it turns out empirically that more accurate models seem to reduce the size of this set. Even when a model's top choice is wrong, it often still prefers constraint-conforming tokens to arbitrary non-conforming tokens.

8.6 Related work

Our contributions are primarily situated among several bodies of work.

First, there is a large body of work leveraging LMs for semantic parsing or code generation tasks, while forcing adherence to a grammar or other constraints [Shin et al., 2021, Scholak et al., 2021, Poesia et al., 2022, Shin and Van Durme, 2022, Geng et al., 2023, Zheng et al., 2024, Moskal et al., 2024, Wang et al., 2024a, Ugare et al., 2024]. Closely related is a series of algorithmic advances that enable the efficient construction and application of grammar constraints for sequential inference

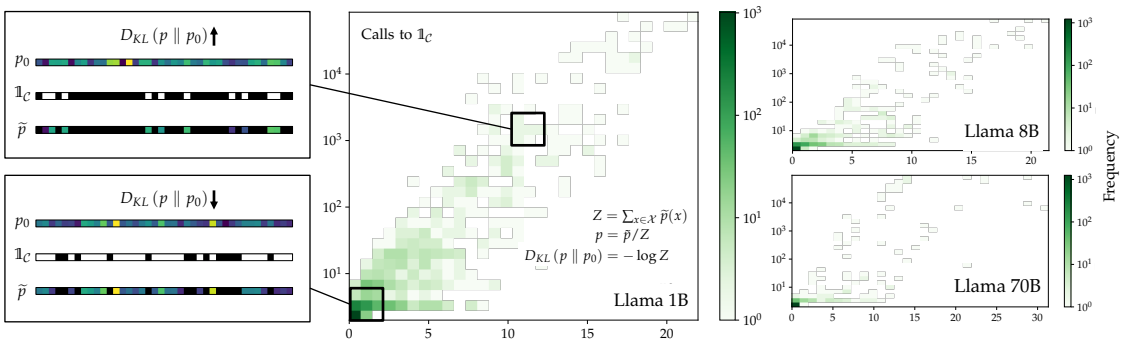


Figure 8-7: The number of AWRS calls to $\Phi(\boldsymbol{x}_{<t})$ (**y-axis**) scales with $D_{KL}\left(\ell_{\text{eff}}(\cdot \mid \boldsymbol{x}_{<t}) \parallel p(\cdot \mid \boldsymbol{x}_{<t})\right)$ (Nats; **x-axis**).

problems via compilation to automata [Deutsch et al., 2019, Willard and Louf, 2023, Kuchnik et al., 2023, Koo et al., 2024].

Second, there is a large body of work that aims to generate from LMs subject to hard or soft constraints. Many such strategies are based in reinforcement learning (RL) [Ziegler et al., 2019, Stiennon et al., 2020, Bai et al., 2022, Ouyang et al., 2022], classifier-guided control [Cheng et al., 2024], efficient probabilistic inference through tractable proxy models [Zhang et al., 2023a], and locally applied *logit biasing* or *masking* based on domain-specific potential functions [Pascual et al., 2021, Huang et al., 2024]. In addition, there is a growing set of approaches that cast constrained sequential generation as probabilistic conditioning, leveraging the toolkit of probabilistic inference to derive principled generation strategies [Miao et al., 2020, Yang and Klein, 2021, Lew et al., 2023d, Zhao et al., 2024, Park et al., 2025, Puri et al., 2025]. These works themselves are motivated by a rich history of conditioning autoregressive models in NLP, often in combination with particle-based inference methods [Börschinger and Johnson, 2011, Dubbin and Blunsom, 2012, Yang and Eisenstein, 2013, Buys and Blunsom, 2015, Lin and Eisner, 2018].

In this work, our algorithms aim to unify and improve on each of these bodies of preceding work, tackling semantic parsing and code generation tasks with a combination of grammar constraints (Φ_{eff}), expensive potentials (Φ_{exp}), and asymptotically correct inference (via SMC).

Grammar-constrained semantic parsing with LMs Shin et al. [2021] presented a system allowing LMs to be locally intersected with (boolean) CFGs to restrict generations to conform to target formal languages, and that with only a few in-context examples, such an inference-time strategy could outperform more substantial fine-tuning. Concurrently, PICARD [Scholak et al., 2021] presented an approach for intersecting LMs with an incremental parsing algorithm and showed how additional context-sensitive constraints could be imposed, such as requiring table-column matching for SQL generation via the use of programmable "guards". Synchromesh [Poesia et al., 2022] generalized these frameworks and extended the

idea of incremental guards that can impose semantic restrictions during generation—such as typing and scoping rules—by dynamically constructing constraints as regular expressions on the fly. A great deal of other work has explored variants of LM-grammar intersection including the effectiveness of pre-training models on code for these settings [Shin and Van Durme, 2022], the runtime compilation of individual task instances into highly specific, task-specialized grammars [Geng et al., 2023], and even using the LM to generate grammars directly at runtime, that then restrict their own generation to solve a task [Wang et al., 2024a]. Other work has focused more closely on the standard syntactic-constraint problem but with an emphasis on optimizing efficient data structures and algorithms for fast LM-CFG intersection [Ugare et al., 2024, Zheng et al., 2024, Moskal et al., 2024].

A parallel line of work in this space has been concerned with the efficient construction and application of constraints for sequential inference problems. Deutsch et al. [2019] first noted that regular and context-free grammar constraints could be pre-compiled to automata—these could then be used during sequential inference to impose constraints with near-zero runtime overhead. This approach was independently developed and efficiently implemented in the context of restricting LM generations to regular expressions by the Outlines [Willard and Louf, 2023] and ReLM libraries [Kuchnik et al., 2023]. Similar work was later developed by Koo et al. [2024], who extended several formal automata-theoretic characteristics of these constructions.

This work has noted the complications of efficiently intersecting grammars whose atoms are terminals and LMs whose atoms are tokens, which we refer to as the *token–terminal alignment problem*. An efficient and accurate solution to this problem space was one of several desiderata for our proposal algorithm (see Algorithm 8 in Section 8.3 for more details). These works have also discussed considerations that arise in the construction of automata, whose arcs are tokens, in the assignment of probabilities to strings. Namely, there are exponentially many latent token trajectories that correspond to a generated sequence. While the correct method for assigning string probabilities involves marginalizing over these trajectories [Cao and Rimell, 2021], in practice, simply using the *canonical tokenization* accounts for the overwhelming majority of the probability mass and can be justified [Chirkova et al., 2023, Kuchnik et al., 2023, Berglund et al., 2024, Vieira et al., 2024]. In the present work, we do not enforce this assumption and allow all token trajectories.

Conditional generation subject to constraints Language models pre-trained on a next-word objective reflect the distribution of their pre-training corpora, but often the inference-time needs of tasks necessitate that LMs modify this base distribution.

One approach to this class of problems is fine-tuning or reinforcement learning via some set of data that more closely mirrors the target task, such as via reinforcement learning from human feedback (RLHF) [Ziegler et al., 2019, Stiennon et al., 2020, Bai et al., 2022, Ouyang et al., 2022], but this method comes with challenges such as hyperparameter sensitivity and distributional collapse [Zheng et al., 2023, Zhu et al., 2023, Xiong et al., 2024]. Some of these drawbacks can be mitigated by utilizing

on-policy data [Tajwar et al., 2024] and imposing a KL penalty that penalizes shifting an LM too far from its prior distribution, casting optimization as a variational inference problem [Korbak et al., 2022, Amini et al., 2025].

Another inference-time approach to controlled generation for an LM is via direct modification to the LM's sampling distribution. This may be done via controlling intermediate layer activations with classifier guidance [Cheng et al., 2024], guiding autoregressive generation with a proxy probabilistic model for which estimation of the conditional density is tractable [Zhang et al., 2023a], or most commonly by directly intervening on the final logits before sampling to impose intersection with a potential function. Pascual et al. [2021] presented an early variant of such *logit-biasing* to encourage the presence of predefined guide words in generations.

This pattern is employed more broadly for hard constraints via *logit-masking*, setting the probability associated with particular tokens to zero, forcing the LM to sample from a subset of its distribution over sequences. This approach is used in most of the grammar-constrained semantic parsing work outlined in the previous section. Most recently, there have been attempts to restrict and re-weight generations not only via grammars but through additional expensive potentials such as grounded affordances in robotics settings [Ahn et al., 2022, Huang et al., 2024]. However, in all of these works, constraints are imposed greedily, resulting in a local product of experts construction, and care is not taken to appropriately target the implied global product of experts. It should then come as no surprise that while standard approaches to grammar-constrained generation have been successful, they have been far from a silver bullet [Tam et al., 2024].

Approximate posterior inference in large language models via sampling This leads to a third line of work that formulates constrained generation from language models as posterior inference [Zhang et al., 2023a], and employs approximate inference to sample from the desired target distribution. This is in contrast to yet another line of work that views constrained decoding as an *optimization* problem, and tackles it via search [Meister et al., 2020, Lu et al., 2021, Zhang et al., 2023b] or continuous optimization [Dathathri et al., 2019, Kumar et al., 2021].

Several approximate inference algorithms have been explored for generating constrained samples from LMs, including rejection sampling [Poesia et al., 2022], as well as MCMC [Miao et al., 2019, Hie et al., 2022, Zhang et al., 2020, Qin et al., 2022, Kumar et al., 2022, Du et al., 2024]. A weakness of MCMC-based approaches is that they do not fully exploit the autoregressive factorization of modern language models; each edit to a candidate sequence requires re-evaluating the entire sequence (or at least the entire suffix) to compute a new target density.

Lew et al. [2023d] was an earlier version of the work presented here, which proposed SMC steering of LMs via probabilistic programming specifications. Shortly thereafter, Zhao et al. [2024] independently developed a framework for expressing various LM tasks as probabilistic inference problems that can be tackled with SMC. Similar to our work, Zhao et al. [2024] guide SMC with intermediate targets—in their

case, learned twist functions via a novel contrastive method—that enable estimation of the expected future value of each candidate partial sequence. Their work also developed methods for evaluating LM inference algorithms via bi-directional bounds on the log-partition function that can be used to estimate the KL divergence between the inference and target distribution. In contrast to this prior work, our approach to SMC leverages incremental static and dynamic analyses to inform our proposal distributions and twist functions, as opposed to learning components of these algorithms via a costly contrastive fine-tuning procedure. In addition, our results directly relate the quality of our posterior approximation to improved performance on a series of standard, difficult benchmark tasks.

Concurrent with our work, Park et al. [2025] have highlighted the distinction between the prevalent locally constrained decoding approach and the more accurate targeting of the global distribution that arises from combining language models with constraints. Park et al. [2025]'s approach to approximate the global distribution is based on the concept of *expected future grammaticality*, which is the probability that the completion to be sampled from the LM will be compliant with the given grammar. The authors describe an iterative algorithm that approximates the global distribution by refining the estimates of the expected future grammatically. However, the proposed strategy shows relatively slow convergence, was specifically designed for a CFG constraint, and may not be easily adaptable to constraining with multiple potential functions.

Part IV

Conclusion

9

FUTURE DIRECTIONS

This thesis has presented a suite of program transformations for automating key mathematical operations on probabilistic programs, including integration, unbiased estimation, Radon-Nikodym differentiation, and differentiation. The empirical studies in Part III further demonstrated that these techniques can be implemented scalably and can enable new applications in Bayesian inference, data cleaning, and controllable language generation. However, there are many exciting avenues for future research to build upon this foundation.

9.1 Scaling automatic integration and differentiation

The program transformations developed in this thesis are theoretically rigorous, providing strong correctness guarantees for the automated computation of various integrals, derivatives, and density ratios. Chapter 6 further shows that these transformations can be implemented in a way that scales to practical problems, for instance, by leveraging GPU acceleration for the stochastic estimators used in variational inference. Despite these successes, more work is needed to ensure that the automatically generated code is consistently as performant as carefully hand-coded implementations, especially for widely used algorithms.

For **Radon-Nikodym derivatives**, which are central to many Monte Carlo methods, several directions for optimization are particularly promising:

- *Incremental computation:* Inference algorithms like MCMC and SMC often require evaluating Radon-Nikodym derivatives thousands or millions of times at points that are only slightly perturbed from previous evaluations. Developing techniques for incrementalizing the computation of these derivatives, reusing previous computations where possible, could yield substantial speedups.
- *Proposal fusion and common subexpression elimination:* When computing a Radon-Nikodym derivative $\frac{d\mu_1}{d\mu_2}$, there is often significant shared structure between the

measures μ_1 and μ_2. Future work could focus on automatically identifying and exploiting this shared structure to avoid redundant computations, effectively canceling out common terms before they are even computed.

For **integrals, gradients, and their unbiased estimators**, a key challenge for further scaling is to improve support for parallelization. Our current transformations, particularly for integration (Chapter 3), rely on a continuation-passing style (CPS), which can introduce explicit, sequential data-flow dependencies into code that in principle could be parallelized. Research into alternative compilation strategies or specialized parallelization techniques for CPS-transformed code could unlock further performance gains.

9.2 Static analyses for discharging preconditions for correctness

The correctness theorems presented in this thesis, particularly for Radon-Nikodym differentiation (Chapter 4) and gradient estimation (Chapter 5), come with certain preconditions. For example, the soundness of **spi** relies on an automatically generated unit test passing with probability 1, which implicitly encodes requirements like the absolute continuity of a proposal distribution with respect to a target. Similarly, some gradient estimation strategies introduce local domination conditions into the correctness theorem for **diff**.

While the automatically generated unit tests provide a dynamic check, future work should aim to develop **static analyses** that can discharge these preconditions at compile time, providing stronger upfront guarantees. Such analyses could draw on existing work in this direction, including:

- For ensuring absolute continuity in the context of density ratio computation, techniques like *trace types* [Lew et al., 2020a] or *guide types* [Wang et al., 2021] offer promising directions for statically verifying that a proposal distribution appropriately covers the support of a target measure.
- For variational inference and the estimation of gradients of expected values, Lee et al. [2020a] and Lee et al. [2023] have explored static analyses for verifying some of the conditions for correctness, including for the interchange of differentiation and integration. Khajwal et al. [2023] have also explored static approaches to ensuring soundness of gradient estimation.

Successfully developing such static analyses would not only enhance the reliability of the automated transformations but also provide users with earlier feedback on potential issues in their models or inference strategies.

9.3 Quantitative guarantees for stochastic estimators

This thesis has focused primarily on establishing the *unbiasedness* of the various stochastic estimators it automates. Unbiasedness is a crucial property, ensuring that,

on average, the estimators hit the true value. However, for practical applications, the *variance* of these estimators is equally important. An unbiased estimator with extremely high variance may require an impractically large number of samples to yield a useful result.

Future research should aim to provide **quantitative guarantees about the variance** of the automatically generated estimators. This could involve:

- Developing compositional analyses that can bound the variance of a complex estimator based on the variances of its components and the way they are combined. A key challenge is that variance does not compose as straightforwardly as unbiasedness.
- Statically or dynamically verifying the binary property of *finite variance*. For many Monte Carlo algorithms, finite variance of the underlying estimators is a prerequisite for proving convergence rates or central limit theorems.
- Providing tools or heuristics that help users understand and optimize the variance of the estimators generated for their specific programs, perhaps by guiding their choice of estimation strategy annotations (as introduced in Chapter 3).

Making progress on quantitative variance analysis would significantly enhance the practical utility and predictability of automated stochastic estimation in PPLs.

9.4 Probabilistic program synthesis

While this thesis provides tools to automate complex mathematical operations *on* probabilistic programs, the question of *who writes* these programs remains. Traditionally, the answer has been human researchers and engineers. However, an exciting and rapidly growing area of research is **probabilistic program synthesis**, where the goal is to automatically generate or infer probabilistic programs from data, high-level specifications, or natural language descriptions.

This trend is evident in several recent lines of work, including:

- Language models that generate probabilistic programs that control language models ("self-steering LMs" [Grand et al., 2025]).
- Research on using LMs to synthesize probabilistic programs for specific domains, such as cognitive modeling or data analysis [e.g., Wong et al., 2023, Li et al., 2024, Domke, 2025].
- Work on synthesizing probabilistic programs from data alone, without natural language [Saad et al., 2019b].

The techniques developed in this thesis can play an important role in this emerging field. For instance, the ability to automatically and efficiently compute likelihoods (via Radon-Nikodym derivatives) under a synthesized probabilistic program can

provide a powerful signal for guiding the synthesis process, acting as a form of automated verifier or fitness function.

9.5 Train-time language model probabilistic programming

Chapter 8 focused on *test-time* techniques for improving the behavior of pre-trained language models (LMs) by using probabilistic inference to guide generation. However, the comprehensive toolkit for differentiating probabilistic computations developed in this thesis, particularly in Chapter 5, opens up significant opportunities for applying probabilistic programming techniques during the **training** of LMs.

Potential directions include:

- *Improved gradient estimators for existing objectives:* Many current LM training objectives, such as those used in Reinforcement Learning from Human Feedback (RLHF) or other setups involving learning from verifiers or rewards, rely on estimating gradients of expected values. The techniques from Chapter 5 could lead to lower-variance or more computationally efficient unbiased estimators for these gradients, potentially accelerating training or improving final model performance.
- *New training objectives for calibrated inference:* Beyond existing objectives, our framework could enable the design of novel training paradigms that directly optimize LMs to perform more calibrated Bayesian inference or to better align their outputs with complex, programmatically specified desiderata.

These directions could ultimately lead to more robust, reliable, and controllable models.

BIBLIOGRAPHY

Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. volume 4, pages 38:1–38:28, 2020. doi: 10.1145/3371106. URL https://doi.org/10.1145/3371106.

Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? In *Proceedings of the VLDB Endowment*, 2016. doi: 10.14778/2994509.2994518.

Felix V. Agakov and David Barber. An Auxiliary Variational Method. In Nikhil Ranjan Pal, Nik Kasabov, Rajani K. Mudi, Srimanta Pal, and Swapan Kumar Parui, editors, *Neural Information Processing*, Lecture Notes in Computer Science, pages 561–566, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30499-9. doi: 10.1007/978-3-540-30499-9_86.

Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006. doi: 10.1007/11693024_6. URL https://doi.org/10.1007/11693024_6.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022. URL https://arxiv.org/abs/2204.01691.

Afra Amini, Tim Vieira, Elliott Ash, and Ryan Cotterell. Variational best-of-N alignment. In *Proceedings of International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=W9FZEQj3vv.

Christophe Andrieu and Gareth O Roberts. The pseudo-marginal approach for efficient monte carlo computations. *The Annals of Statistics*, 37(2):697–725, 2009.

Zachary Ankner, Mansheej Paul, Brandon Cui, Jonathan Daniel Chang, and Prithviraj Ammanabrolu. Critique-out-loud reward models. In *Pluralistic Alignment Workshop at NeurIPS 2024*, 2024. URL https://arxiv.org/pdf/2408.11791.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 109–122. ACM, 2007. doi: 10.1145/1190216.1190235. URL https://doi.org/10.1145/1190216.1190235.

Nimar S Arora, Rodrigo de Salvo Braz, Erik B Sudderth, and Stuart Russell. Gibbs sampling in open-universe stochastic languages. *arXiv preprint arXiv:1203.3464*, 2012.

Gaurav Arya, Moritz Schauer, Frank Schäfer, and Chris Rackauckas. Automatic differentiation of programs with discrete randomness. *CoRR*, abs/2210.08572, 2022. doi: 10.48550/arXiv.2210.08572. URL https://doi.org/10.48550/arXiv.2210.08572.

Robert J Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022. URL https://arxiv.org/abs/2212.08073.

Sai Praveen Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. Systematically differentiating parametric discontinuities. *ACM Trans. Graph.*, 40(4):107:1–107:18, 2021. doi: 10.1145/3450626.3459775. URL https://doi.org/10.1145/3450626.3459775.

Matthew Barnett. regex, 2014. URL https://github.com/mrabarnett/mrab-regex.

Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. On the versatility of open logical relations - continuity, automatic differentiation, and a containment theorem. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 56–83. Springer, 2020. doi: 10.1007/978-3-030-44914-8_3. URL https://doi.org/10.1007/978-3-030-44914-8_3.

Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, et al. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–24, 2019.

Mark A Beaumont. Estimation of population growth or decline in genetically monitored populations. *Genetics*, 164(3):1139–1160, 2003.

McCoy R Becker, Alexander K Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C Rinard, and Vikash K Mansinghka. Probabilistic programming with programmable variational inference. *Proceedings of the ACM on Programming Languages*, 8(PLDI):2123–2147, 2024.

Martin Berglund, Willeke Martens, and Brink Van der Merwe. Constructing a BPE tokenization DFA. In *International Conference on Implementation and Application of Automata*. Springer, 2024. URL https://dl.acm.org/doi/abs/10.1007/978-3-031-71112-1_5.

G. Richard Bickerton, Gaia V. Paolini, Jérémy Besnard, Sorel Muresan, and Andrew L. Hopkins. Quantifying the chemical beauty of drugs. *Nature Chemistry*, 4(2), 2012. URL https://www.nature.com/articles/nchem.1243.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming, October 2018. URL http://arxiv.org/abs/1810.09538. arXiv:1810.09538 [cs, stat].

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, F. Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 2019a. ISSN 15337928.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D

Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019b.

David M Blei and Michael I Jordan. Variational inference for dirichlet process mixtures. 2006.

Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. *ACM SIGPLAN Notices*, 51(9):33–46, 2016.

Jörg Bornschein and Yoshua Bengio. Reweighted Wake-Sleep, April 2015. URL http://arxiv.org/abs/1406.2751. arXiv:1406.2751 [cs].

Benjamin Börschinger and Mark Johnson. A particle filter algorithm for Bayesian wordsegmentation. In *Proceedings of the Australasian Language Technology Association Workshop*, December 2011. URL https://aclanthology.org/U11-1004/.

Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.*, 4 (POPL):64:1–64:27, 2020. doi: 10.1145/3371132. URL https://doi.org/10.1145/3371132.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance Weighted Autoencoders, November 2016. URL http://arxiv.org/abs/1509.00519. arXiv:1509.00519 [cs, stat].

Jan Buys and Phil Blunsom. A Bayesian model for generative transition-based dependency parsing. In *Proceedings of the International Conference on Dependency Linguistics*, 2015. URL https://aclanthology.org/W15-2108/.

Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The YCB object and model set: Towards common benchmarks for manipulation research. In *2015 International Conference on Advanced Robotics (ICAR)*, pages 510–517. IEEE, 2015.

Kris Cao and Laura Rimell. You should evaluate your language model on marginal likelihood over tokenisations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://aclanthology.org/2021.emnlp-main.161.pdf.

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

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2017b. ISSN 15487660. doi: 10.18637/jss.v076.i01.

Sourav Chatterjee and Persi Diaconis. The sample size required in importance sampling. *The Annals of Applied Probability*, 28(2), 2018. URL https://www.jstor.org/stable/pdf/26542331.pdf.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei A. Zaharia, and James Y. Zou. Are more LLM calls all you need? towards the scaling properties of compound AI systems. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/51173cf34c5faac9796a47dc2fdd3a71-Abstract-Conference.html.

Emily Cheng, Marco Baroni, and Carmen Amo Alonso. Linearly controlled language generation with performative guarantees. *NeurIPS Workshop on Foundation Model Interventions*, 2024. URL https://openreview.net/pdf?id=V2xBBD1Xtu.

Yu-Hsi Cheng, Todd Millstein, Guy Van den Broeck, and Steven Holtzen. flip-hoisting: Exploiting repeated parameters in discrete probabilistic programs. *arXiv preprint arXiv:2110.10284*, 2021.

Nadezhda Chirkova, Germán Kruszewski, Jos Rozen, and Marc Dymetman. Should you marginalize over possible tokenizations? In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2023. URL https://aclanthology.org/2023.acl-short.1.pdf.

YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. *arXiv preprint arXiv:2009.09031*, 2020.

Nicolas Chopin, Omiros Papaspiliopoulos, et al. *An introduction to sequential Monte Carlo*. Springer, 2020.

Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *Proceedings - International Conference on Data Engineering*, 2013. ISBN 9781467349086. doi: 10.1109/ICDE.2013.6544847.

Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/pdf/2110.14168.

Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. Automating involutive MCMC using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871*, 2020.

Marco F Cusumano-Towner and Vikash K Mansinghka. Using probabilistic programs as proposals. *arXiv preprint arXiv:1801.03612*, 2018.

Marco F Cusumano-Towner, Alexey Radul, David Wingate, and Vikash K Mansinghka. Probabilistic programs for inferring the goals of autonomous agents. *arXiv preprint arXiv:1704.04977*, 2017.

Marco F. Cusumano-Towner, Alexander K. Lew, Feras A. Saad, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019a. ISBN 9781450367127. doi: 10.1145/3314221.3314642.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 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, pages 221–236, New York, NY, USA, June 2019b. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314642. URL https://dl.acm.org/doi/10.1145/3314221.3314642.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236, 2019c.

Marco Francis Cusumano-Towner. Gen: A High-Level Programming Platform for Probabilistic Inference. (2011):231, 2020.

Michele Dallachiesat, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. NADEEF: A commodity data cleaning system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013. ISBN 9781450320375. doi: 10.1145/2463676.2465327.

Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. Affine monads and lazy structures for bayesian programming. *Proceedings of the ACM on Programming Languages*, 7(POPL):1338–1368, 2023.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*, 2019. URL https://arxiv.org/pdf/1912.02164.

Sushovan De, Yuheng Hu, Venkata Vamsikrishna Meduri, Yi Chen, and Subbarao Kambhampati. BayesWipe: A scalable probabilistic framework for improving data quality. *Journal of Data and Information Quality*, 8(1), oct 2016. ISSN 19361963. doi: 10.1145/2992787.

Christopher De Sa, Ihab F. Ilyas, Benny Kimelfeld, Christopher Ré, and Theodoros Rekatsinas. A formal framework for probabilistic unclean databases. In *Leibniz International Proceedings in Informatics, LIPIcs*, 2019. ISBN 9783959771016. doi: 10.4230/LIPIcs.ICDT.2019.6.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977. ISSN 0035-9246. URL https://www.jstor.org/stable/2984875. Publisher: [Royal Statistical Society, Wiley].

Daniel Deutsch, Shyam Upadhyay, and Dan Roth. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the Conference on Computational Natural Language Learning*, 2019. URL https://aclanthology.org/K19-1045/.

Justin Domke. An Easy to Interpret Diagnostic for Approximate Inference: Symmetric Divergence Over Simulations, February 2021. URL http://arxiv.org/abs/2103.01030. arXiv:2103.01030 [cs, stat].

Justin Domke. Large language bayes. *arXiv preprint arXiv:2504.14025*, 2025.

Arnaud Doucet, Michael K Pitt, George Deligiannidis, and Robert Kohn. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. *Biometrika*, 102(2):295–313, 2015.

Li Du, Afra Amini, Lucas Torroba Hennigen, Xinyan Velocity Yu, Holden Lee, Jason Eisner, and Ryan Cotterell. Principled gradient-based MCMC for conditional sampling of text. In *Proceedings of the International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/du24a.html.

Gregory Dubbin and Phil Blunsom. Unsupervised part of speech inference with particle filters. In *Proceedings of the NAACL HLT Workshop on Induction of Linguistic Structure*, Montréal, QC, 2012. URL https://aclanthology.org/W12-1907.pdf.

Simao Eduardo, Alfredo Nazábal, Christopher KI Williams, and Charles Sutton. Robust variational autoencoders for outlier detection and repair of mixed-type data. In *International Conference on Artificial Intelligence and Statistics*, pages 4056–4066. PMLR, 2020.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proc. ACM Program. Lang.*, 2(POPL):59:1–59:28, 2018. doi: 10.1145/3158147. URL https://doi.org/10.1145/3158147.

S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models, August 2016. URL http://arxiv.org/abs/1603.08575. arXiv:1603.08575 [cs].

Paul Fearnhead, Omiros Papaspiliopoulos, Gareth O Roberts, and Andrew Stuart. Random-weight particle filtering of continuous time processes. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(4):497–512, 2010.

Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

Jakob Foerster, Gregory Farquhar, Maruan Al-Shedivat, Tim Rocktäschel, Eric Xing, and Shimon Whiteson. Dice: The infinitely differentiable Monte Carlo estimator. In *International Conference on Machine Learning*, pages 1529–1538. PMLR, 2018.

Catherine Forbes, Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical distributions*. John Wiley & Sons, 2011.

Charles W Fox and Stephen J Roberts. A tutorial on variational Bayesian inference. *Artificial intelligence review*, 38:85–95, 2012.

Cameron E Freer, Vikash K Mansinghka, and Daniel M Roy. When are probabilistic programs probably computationally tractable. In *NIPS Workshop on Monte Carlo Methods for Modern Applications*, page 41, 2010. URL https://web.mit.edu/vkm/www/FreerManRoy-NIPSMC-2010.pdf.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI International Joint Conference on Artificial Intelligence*, 1999. doi: 10.1007/978-3-662-04599-2_13.

Kanishk Gandhi, Denise HJ Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah Goodman. Stream of Search (SoS): Learning to search in language. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/pdf?id=2cop2jmQVL.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: Composable inference for probabilistic programming. In Amos J. Storkey and Fernando Pérez-Cruz, editors, *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 2018. URL http://proceedings.mlr.press/v84/ge18b.html.

Timon Gehr, Samuel Steffen, and Martin T. Vechev. λpsi: exact inference for higher-order probabilistic programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 883–897. ACM, 2020. doi: 10.1145/3385412.3386006. URL https://doi.org/10.1145/3385412.3386006.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings*

of the Conference on Empirical Methods in Natural Language Processing, 2023. URL https://aclanthology.org/2023.emnlp-main.674.pdf.

Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Generating structured outputs from language models: Benchmark and studies, 2025. URL https://arxiv.org/abs/2501.10868.

Walter R Gilks and Pascal Wild. Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(2):337–348, 1992.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: A Language for Generative Models. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2008)*, pages 220–229. AUAI Press, 2008a.

Noah D Goodman and Michael C Frank. Pragmatic language interpretation as probabilistic inference. *Trends in Cognitive Sciences*, 20(11):818–829, 2016.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2020-10-15.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Kallista A. Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In David A. McAllester and Petri Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 220–229. AUAI Press, 2008b. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24.

Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgström, and John Guiver. Tabular: A schema-driven probabilistic programming language. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, (1):321–334, 2014. ISSN 07308566. doi: 10.1145/2535838.2535850.

Nishad Gothoskar, Marco F. Cusumano-Towner, Ben Zinberg, Matin Ghavamizadeh, Falk Pollok, Austin Garrett, Josh Tenenbaum, Dan Gutfreund, and Vikash K. Mansinghka. 3dp3: 3d scene perception via probabilistic programming. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 9600–9612, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/4fc66104f8ada6257fa55f29a2a567c7-Abstract.html.

Gabriel Grand, Joshua B Tenenbaum, Vikash K Mansinghka, Alexander K Lew, and Jacob Andreas. Self-steering language models. *arXiv preprint arXiv:2504.07081*, 2025.

Shixiang (Shane) Gu, Zoubin Ghahramani, and Richard E Turner. Neural Adaptive Sequential Monte Carlo. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://papers.nips.cc/paper_files/paper/2015/hash/99adff456950dd9629a5260c4de21858-Abstract.html.

Lin Gui, Cristina Garbacea, and Victor Veitch. Bonbon alignment for large language models and the sweetness of best-of-n sampling. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/056521a35eacd9d2127b66a7d3c499c5-Abstract-Conference.html.

David Earl Heckerman, Eric J Horvitz, and Bharat N Nathwani. Toward normative expert systems: Part I, the PATHFINDER project. *Methods of information in medicine*, 31(02):90–105, 1992.

Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. HoloDetect: Few-shot learning for error detection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019. ISBN 9781450356435. doi: 10.1145/3299869.3319888.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021. URL https://arxiv.org/pdf/2105.09938.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

Brian Hie, Salvatore Candido, Zeming Lin, Ori Kabeli, Roshan Rao, Nikita Smetanin, Tom Sercu, and Alexander Rives. A high-level programming language for generative protein design. *bioRxiv*, 2022. URL https://www.biorxiv.org/content/10.1101/2022.12.21.521526v1.full.pdf.

Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, and Radford M. Neal. The "Wake-Sleep" Algorithm for Unsupervised Neural Networks. *Science*, 268(5214): 1158–1161, May 1995. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.7761831. URL https://www.science.org/doi/10.1126/science.7761831.

Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 2013.

Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):140:1–140:31, 2020. doi: 10.1145/3428208. URL https://doi.org/10.1145/3428208.

Daniel G. Horvitz and Donovan J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260), 1952. URL https://www.jstor.org/stable/pdf/2280784.pdf.

Eric J Horvitz, John S Breese, and Max Henrion. Decision theory in expert systems and artificial intelligence. *International journal of approximate reasoning*, 2(3):247–302, 1988.

Yuheng Hu, Sushovan De, Yi Chen, and Subbarao Kambhampati. Bayesian Data Cleaning for Web Data. 2012. URL http://arxiv.org/abs/1204.3677.

Wenlong Huang, Fei Xia, Dhruv Shah, Danny Driess, Andy Zeng, Yao Lu, Pete Florence, Igor Mordatch, Sergey Levine, Karol Hausman, and Brian Ichter. Grounded decoding: Guiding text generation with grounded models for embodied agents. In *Advances in Neural Information Processing Systems*, volume 36, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/bb3cfcb0284642a973dd631ec9184f2f-Paper-Conference.pdf.

Mathieu Huot, Sam Staton, and Matthijs Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2020. doi: 10.1007/978-3-030-45231-5_17. URL https://doi.org/10.1007/978-3-030-45231-5_17.

Mathieu Huot, Alexander K Lew, Vikash K Mansinghka, and Sam Staton. ωpap spaces: Reasoning denotationally about higher-order, recursive probabilistic and differentiable programs. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2023.

Yuki Ichihara, Yuu Jinnai, Tetsuro Morimura, Kenshi Abe, Kaito Ariu, Mitsuki Sakamoto, and Eiji Uchibe. Evaluation of best-of-n sampling strategies for language model alignment. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=H4S4ETc8c9.

Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Li Fei-Fei. Image retrieval using scene graphs. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 3668–3678. IEEE Computer Society, 2015. doi: 10.1109/CVPR.2015.7298990. URL https://doi.org/10.1109/CVPR.2015.7298990.

Justin Johnson, Agrim Gupta, and Li Fei-Fei. Image generation from scene graphs. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 1219–1228. Computer Vision Foundation / IEEE Computer Society, 2018. doi: 10.1109/CVPR.2018.00133. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Johnson_Image_Generation_From_CVPR_2018_paper.html.

Shin-ya Katsumata. Relating computational effects by tt-lifting. *Information and Computation*, 222:228–246, 2013.

Basim Khajwal, C-H Luke Ong, and Dominik Wagner. Fast and correct gradient-based optimisation for probabilistic programming via smoothing. In *European Symposium on Programming*, pages 479–506. Springer Nature Switzerland Cham, 2023.

Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *Advances in Neural Information Processing Systems*, 34:21696–21707, 2021.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014a. URL http://arxiv.org/abs/1312.6114.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, December 2014b. URL http://arxiv.org/abs/1312.6114. arXiv:1312.6114 [cs, stat].

Diederik P. Kingma, Danilo J. Rezende, Shakir Mohamed, and Max Welling. Semi-supervised learning with deep generative models. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3581–3589, Cambridge, MA, USA, December 2014. MIT Press.

Anders Kock. Commutative monads as a theory of distributions. *arXiv preprint arXiv:1108.5952*, 2011.

Daphne Koller, David McAllester, and Avi Pfeffer. Effective bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.

Terry Koo, Frederick Liu, and Luheng He. Automata-based constraints for language model decoding. In *Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BDBdblmyzY.

Tomasz Korbak, Ethan Perez, and Christopher Buckley. RL with KL penalties is better viewed as Bayesian inference. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022. URL https://aclanthology.org/2022.findings-emnlp.77.

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew W. Fitzgibbon. Provably correct, asymptotically efficient,

higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi: 10.1145/3498710. URL https://doi.org/10.1145/3498710.

Emile Krieken, Jakub Tomczak, and Annette Ten Teije. Storchastic: A framework for general stochastic automatic differentiation. *Advances in Neural Information Processing Systems*, 34:7574–7587, 2021.

Kalpesh Krishna, Yapei Chang, John Wieting, and Mohit Iyyer. Rankgen: Improving text generation with large ranking models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*. Association for Computational Linguistics, 2022. URL https://doi.org/10.18653/v1/2022.emnlp-main.15.

Jeremy Kubica and Andrew Moore. Probabilistic noise identification and data cleaning. *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 131–138, 2003. ISSN 15504786. doi: 10.1109/icdm.2003.1250912.

Michael Kuchnik, Virginia Smith, and George Amvrosiadis. Validating large language models with RELM. *Proceedings of Machine Learning and Systems*, 5, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/93c7d9da61ccb2a60ac047e92787c3ef-Paper-mlsys2023.pdf.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic differentiation variational inference. *Journal of machine learning research*, 2017.

Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2015. ISBN 9781467369640. doi: 10.1109/CVPR.2015.7299068.

Sachin Kumar, Eric Malmi, Aliaksei Severyn, and Yulia Tsvetkov. Controlled text generation as continuous optimization with multiple constraints. *Advances in Neural Information Processing Systems*, 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/79ec2a4246feb2126ecf43c4a4418002-Paper.pdf.

Sachin Kumar, Biswajit Paria, and Yulia Tsvetkov. Gradient-based constrained sampling from language models. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2022. URL https://aclanthology.org/2022.emnlp-main.144/.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*. Proceedings of Machine Learning Research, 2023. URL https://proceedings.mlr.press/v202/lai23b/lai23b.pdf.

Dieterich Lawson, Allan Raventós, Andrew Warrington, and Scott Linderman. Sixo: Smoothing inference with twisted objectives. *Advances in Neural Information Processing Systems*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/fddc79681b2df2734c01444f9bc2a17e-Paper-Conference.pdf.

Tuan Anh Le, Adam R. Kosiorek, N. Siddharth, Yee Whye Teh, and Frank Wood. Revisiting Reweighted Wake-Sleep for Models with Stochastic Control Flow, September 2019. URL http://arxiv.org/abs/1805.10469. arXiv:1805.10469 [cs, stat].

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):16:1–16:33, December 2019. doi: 10.1145/3371084. URL https://dl.acm.org/doi/10.1145/3371084.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. Towards verified stochastic variational inference for probabilistic programs. *Proc. ACM Program. Lang.*, 4(POPL):16:1–16:33, 2020a. doi: 10.1145/3371084. URL https://doi.org/10.1145/3371084.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. On correctness of automatic differentiation for non-differentiable functions. *Advances in neural information processing systems*, 33:6719–6730, 2020b.

Wonyeol Lee, Xavier Rival, and Hongseok Yang. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *Proceedings of the ACM on Programming Languages*, 7(POPL):12:335–12:366, January 2023. doi: 10.1145/3571205. URL https://dl.acm.org/doi/10.1145/3571205.

Roger Levy. Expectation-based syntactic comprehension. *Cognition*, 106(3):1126–1177, 2008. URL https://doi.org/10.1016/j.cognition.2007.05.006.

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2020a. ISSN 24751421. doi: 10.1145/3371087.

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proc. ACM Program. Lang.*, 4(POPL):19:1–19:32, 2020b. doi: 10.1145/3371087. URL https://doi.org/10.1145/3371087.

Alexander K. Lew, Monica Agrawal, David A. Sontag, and Vikash Mansinghka. Pclean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In Arindam Banerjee and Kenji Fukumizu, editors, *The 24th International*

Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event, volume 130 of *Proceedings of Machine Learning Research*, pages 1927–1935. PMLR, 2021a. URL http://proceedings.mlr.press/v130/lew21a.html.

Alexander K. Lew, Mathieu Huot, and Vikash K. Mansinghka. Towards denotational semantics of AD for higher-order, recursive, probabilistic languages. *CoRR*, abs/2111.15456, 2021b. URL https://arxiv.org/abs/2111.15456.

Alexander K. Lew, Marco Cusumano-Towner, and Vikash K. Mansinghka. Recursive Monte Carlo and Variational Inference with Auxiliary Variables, November 2022a. URL http://arxiv.org/abs/2203.02836. arXiv:2203.02836 [cs, stat].

Alexander K. Lew, Marco F. Cusumano-Towner, and Vikash K. Mansinghka. Recursive monte carlo and variational inference with auxiliary variables. In James Cussens and Kun Zhang, editors, *Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1-5 August 2022, Eindhoven, The Netherlands*, volume 180 of *Proceedings of Machine Learning Research*, pages 1096–1106. PMLR, 2022b. URL https://proceedings.mlr.press/v180/lew22a.html.

Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. Probabilistic Programming with Stochastic Probabilities. *Proceedings of the ACM on Programming Languages*, 7(PLDI):176:1708–176:1732, June 2023a. doi: 10.1145/3591290. URL https://dl.acm.org/doi/10.1145/3591290.

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. ADEV: sound automatic differentiation of expected values of probabilistic programs. *Proc. ACM Program. Lang.*, 7(POPL):121–153, 2023b. doi: 10.1145/3571198. URL https://doi.org/10.1145/3571198.

Alexander K Lew, George Matheos, Tan Zhi-Xuan, Matin Ghavamizadeh, Nishad Gothoskar, Stuart Russell, and Vikash K Mansinghka. SMCP3: Sequential Monte Carlo with probabilistic program proposals. In *International conference on artificial intelligence and statistics*, pages 7061–7088. PMLR, 2023c.

Alexander K Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K Mansinghka. Sequential monte carlo steering of large language models using probabilistic programs. *arXiv preprint arXiv:2306.03081*, 2023d.

Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference. *Proc. ACM Program. Lang.*, 7(POPL):1454–1482, 2023a. doi: 10.1145/3571243. URL https://doi.org/10.1145/3571243.

Michael Y. Li, Dieterich Lawson, and Scott Linderman. Neural Adaptive Smoothing via Twisting. July 2023b. URL https://openreview.net/forum?id=rC6-kGN-0v.

Michael Y Li, Emily B Fox, and Noah D Goodman. Automated statistical model discovery with language models. *arXiv preprint arXiv:2402.17879*, 2024.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023. URL https://openreview.net/pdf?id=v8L0pN6EOi.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Chu-Cheng Lin and Jason Eisner. Neural particle smoothing for sampling from conditional sequence models. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018. URL https://aclanthology.org/N18-1085/.

Benjamin Lipkin, Benjamin LeBrun, Jacob Hoover Vigly, João Loula, David R MacIver, Li Du, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Timothy J O'Donnell, et al. Fast controlled generation from language models with adaptive weighted rejection sampling. *arXiv preprint arXiv:2504.05410*, 2025.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, et al. Syntactic and semantic control of large language models via sequential monte carlo. *arXiv preprint arXiv:2504.13139*, 2025.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, et al. Neurologic A* esque decoding: Constrained text generation with lookahead heuristics. *arXiv preprint arXiv:2112.08726*, 2021. URL https://aclanthology.org/2022.naacl-main.57.pdf.

Daniel Lundén, David Broman, Fredrik Ronquist, and Lawrence M Murray. Automatic alignment of sequential Monte Carlo inference in higher-order probabilistic programs. *arXiv preprint arXiv:1812.07439*, 2018. URL https://arxiv.org/pdf/1812.07439.

Daniel Lundén, Johannes Borgström, and David Broman. Correctness of sequential Monte Carlo inference for probabilistic programming languages. In *ESOP*, pages 404–431, 2021.

Chris J. Maddison, Dieterich Lawson, George Tucker, Nicolas Heess, Mohammad Norouzi, Andriy Mnih, Arnaud Doucet, and Yee Whye Teh. Filtering Variational Objectives, November 2017. URL http://arxiv.org/abs/1705.09279. arXiv:1705.09279 [cs, stat].

Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. In Nobuko Yoshida, editor, *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*, volume 12648 of *Lecture Notes in Computer Science*, pages 432–461. Springer, 2021. doi: 10.1007/978-3-030-72019-3_16. URL https://doi.org/10.1007/978-3-030-72019-3_16.

Nikolay Malkin, Salem Lahlou, Tristan Deleu, Xu Ji, Edward Hu, Katie Everett, Dinghuai Zhang, and Yoshua Bengio. Gflownets and variational inference. *arXiv preprint arXiv:2210.00580*, 2022.

Vikash Mansinghka, Daniel Roy, Eric Jonas, and Joshua Tenenbaum. Exact and approximate sampling by systematic stochastic search. In *Artificial Intelligence and Statistics*, pages 400–407. PMLR, 2009a. URL https://proceedings.mlr.press/v5/mansinghka09a.html.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014a.

Vikash Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014b. URL http://arxiv.org/abs/1404.0099.

Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic Programming with Programmable Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, volume 14, pages 603–616, New York, NY, USA, 2018a. ACM. ISBN 9781450356985. doi: 10.1145/3192366.3192409. URL https://doi.org/10.1145/3192366.3192409.

Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 603–616, 2018b.

Vikash Kumar Mansinghka et al. *Natively probabilistic computation*. PhD thesis, Citeseer, 2009b.

Neil G Marchant, Andee Kaplan, Daniel N Elazar, Benjamin IP Rubinstein, and Rebecca C Steorts. d-blink: Distributed end-to-end bayesian entity resolution. *Journal of Computational and Graphical Statistics*, pages 1–16, 2021.

Nicholas Elias Matsakis. Active Duplicate Detection with Bayesian Nonparametric Models. 2010.

Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. A statistical method for integrated data cleaning and imputation. Technical report, 2009. URL https://docs.lib.purdue.edu/cstech/1723.

Eric Mays, Fred J Damerau, and Robert L Mercer. Context based spelling correction. *Information Processing & Management*, 27(5):517–522, 1991.

Damiano Mazza and Michele Pagani. Automatic differentiation in PCF. *Proc. ACM Program. Lang.*, 5(POPL):1–27, 2021. doi: 10.1145/3434309. URL https://doi.org/10.1145/3434309.

A Mccallum and B Wellner. Object Consolodation by Graph Partitioning with a Conditionally-Trained Distance Metric. *In Proceedings of the KDD-2003 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, pages 19–24, 2003.

Clara Meister, Tim Vieira, and Ryan Cotterell. If beam search is the answer, what was the question? *arXiv preprint arXiv:2010.02650*, 2020. URL https://aclanthology.org/2020.emnlp-main.170/.

Ning Miao, Hao Zhou, Lili Mou, Rui Yan, and Lei Li. CGMH: Constrained sentence generation by metropolis-hastings sampling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, number 01, 2019. URL https://ojs.aaai.org/index.php/AAAI/article/view/4659.

Ning Miao, Yuxuan Song, Hao Zhou, and Lei Li. Do you have the right scissors? tailoring pre-trained language models via Monte-Carlo methods. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2020. URL https://aclanthology.org/2020.acl-main.314/.

Jesse Michel, Kevin Mu, Xuanda Yang, Sai Praveen Bangaru, Elias Rojas Collins, Gilbert Bernstein, Jonathan Ragan-Kelley, Michael Carbin, and Tzu-Mao Li. Distributions for compositionally differentiating parametric discontinuities. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):893–922, 2024.

Brian Milch and Stuart Russell. General-purpose MCMC inference over relational structures. *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence, UAI 2006*, pages 349–358, 2006.

Brian Milch, Bhaskara Marthi, and Stuart Russell. Blog: Relational modeling with unknown objects. In *ICML 2004 workshop on statistical relational learning and its connections to other fields*, pages 67–73, 2004.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L Ong, and Andrey Kolobov. BLOG: Probabilistic Models With Unknown Objects. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1352–1359. Morgan Kaufmann Publishers Inc., 2005.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte Carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 21 (132):1–62, 2020.

Michal Moskal, Madan Musuvathi, and Emre Kıcıman. AI Controller Interface. https://github.com/microsoft/aici/, 2024.

Sidharth Mudgal, Jong Lee, Harish Ganapathy, YaGuang Li, Tao Wang, Yanping Huang, Zhifeng Chen, Heng-Tze Cheng, Michael Collins, Trevor Strohman, Jilin Chen, Alex Beutel, and Ahmad Beirami. Controlled decoding from language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=bVIcZb7Qa0.

Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *International Conference on Artificial Intelligence and Statistics*, pages 1037–1046, 2018.

Lawrence M. Murray. Bayesian state-space modelling on high-performance hardware using LibBi. *Journal of Statistical Software*, 67(10):1–28, 2015. ISSN 15487660. doi: 10.18637/jss.v067.i10.

Lawrence M Murray and Thomas B Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

Christian Naesseth, Scott Linderman, Rajesh Ranganath, and David Blei. Variational sequential Monte Carlo. In *International conference on artificial intelligence and statistics*, pages 968–977. PMLR, 2018.

Christian A. Naesseth, Fredrik Lindsten, and Thomas B. Schön. Elements of sequential monte carlo. *Found. Trends Mach. Learn.*, 12(3), 2019a. doi: 10.1561/2200000074. URL https://doi.org/10.1561/2200000074.

Christian A Naesseth, Fredrik Lindsten, Thomas B Schön, et al. Elements of sequential Monte Carlo. *Foundations and Trends® in Machine Learning*, 12(3): 307–392, 2019b.

Christian A. Naesseth, Fredrik Lindsten, and David Blei. Markovian score climbing: variational inference with KL(p||q). In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, pages 15499–15510, Red Hook, NY, USA, December 2020. Curran Associates Inc. ISBN 978-1-71382-954-6.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021. URL https://arxiv.org/pdf/2112.09332.

Praveen Narayanan and Chung Chieh Shan. Symbolic Disintegration with a Variety of Base Measures. *ACM Transactions on Programming Languages and Systems*, 42(2), 2020. ISSN 15584593. doi: 10.1145/3374208.

Siddharth Narayanaswamy, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank D. Wood, and Philip H. S. Torr. Learning disentangled representations with semi-supervised deep generative models. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5925–5935, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/9cb9ed4f35cf7c2f295cc2bc6f732a84-Abstract.html.

Radford M Neal. Markov chain sampling methods for dirichlet process mixture models. *Journal of computational and graphical statistics*, 9(2):249–265, 2000.

Noel O'Boyle. partialsmiles: A validating SMILES parser, with support for incomplete SMILES, 2024. URL https://github.com/baoilleach/partialsmiles.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://aclanthology.org/2023.emnlp-main.313/.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/2bdc2267c3d7d01523e2e17ac0a754f3-Paper-Conference.pdf.

Damian Pascual, Beni Egressy, Clara Meister, Ryan Cotterell, and Roger Wattenhofer. A plug-and-play method for controlled text generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, November 2021. URL https://aclanthology.org/2021.findings-emnlp.334/.

Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart Russell, and Ilya Shpitser. Identity uncertainty and citation matching. *Advances in Neural Information Processing Systems*, 2003. ISSN 10495258.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 1988.

Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. A type theory for defining logics and proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi: 10.1109/LICS.2019.8785683. URL https://doi.org/10.1109/LICS.2019.8785683.

Andrew M Pitts. Relational properties of domains. *Information and computation*, 127 (2):66–90, 1996.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. Variational autoencoder for deep learning of images, labels and captions. *Advances in Neural Information Processing Systems*, 29, 2016.

Isha Puri, Shivchander Sudalairaj, Guangxuan Xu, Kai Xu, and Akash Srivastava. A probabilistic inference approach to inference-time scaling of LLMs using particle-based Monte Carlo methods. *arXiv preprint arXiv:2502.01618*, 2025. URL https://arxiv.org/pdf/2502.01618.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. Cold decoding: Energy-based constrained text generation with langevin dynamics. *Advances in Neural Information Processing Systems*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/3e25d1aff47964c8409fd5c8dc0438d7-Paper-Conference.pdf.

Tom Rainforth. Nesting probabilistic programs. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, pages 249–258. AUAI Press, 2018. URL http://auai.org/uai2018/proceedings/papers/92.pdf.

Tom Rainforth, Adam R. Kosiorek, Tuan Anh Le, Chris J. Maddison, Maximilian Igl, Frank Wood, and Yee Whye Teh. Tighter Variational Bounds are Not Necessarily Better, February 2018. URL https://arxiv.org/abs/1802.04537v3.

Rajesh Ranganath, Sean Gerrish, and David M. Blei. Black box variational inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 814–822. JMLR.org, 2014. URL http://proceedings.mlr.press/v33/ranganath14.html.

Rajesh Ranganath, Dustin Tran, and David Blei. Hierarchical Variational Models. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 324–333. PMLR, June 2016. URL https://proceedings.mlr.press/v48/ranganath16.html. ISSN: 1938-7228.

Theodoros Rekatsinas, Xu Chuy, Ihab F. Ilyasy, and Christopher Ré. HoloClean: Holistic data repairs with probabilistic inference. In *Proceedings of the VLDB Endowment*, 2017. doi: 10.14778/3137628.3137631.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*, pages 28–37, 2016.

Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B Schön, and David Broman. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *Communications biology*, 4(1):244, 2021.

Ronald Rosenfeld, Stanley Chen, and Xiaojin Zhu. Whole-sentence exponential language models: A vehicle for linguistic-statistical integration. *Computer Speech & Language*, 15, 01 2001. URL https://www.sciencedirect.com/science/article/abs/pii/S0885230800901591.

Daniel M Roy, Vikash K Mansinghka, Noah D Goodman, and Joshua B Tenenbaum. A stochastic programming perspective on nonparametric bayes. In *Nonparametric Bayesian Workshop, Int. Conf. on Machine Learning*, volume 22, page 26, 2008.

Subhro Roy, Samuel Thomson, Tongfei Chen, Richard Shin, Adam Pauls, Jason Eisner, and Benjamin Van Durme. BenchCLAMP: A benchmark for evaluating language models on syntactic and semantic parsing. In *Advances in Neural Information Processing Systems*, volume 36, 2024.

URL https://proceedings.neurips.cc/paper_files/paper/2023/file/9c1535a02f0ce079433344e14d910597-Paper-Datasets_and_Benchmarks.pdf.

Lars Ruddigkeit, Ruud Van Deursen, Lorenz C Blum, and Jean-Louis Reymond. Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17. *Journal of Chemical Information and Modeling*, 52(11), 2012. URL https://pubs.acs.org/doi/pdf/10.1021/ci300415d.

Feras A Saad, Marco Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling. *Proc. ACM Program. Lang.*, 3(POPL):37:1—-37:29, 2019a.

Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019b.

Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: Probabilistic programming with fast exact symbolic inference. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 804–819, 2021. doi: 10.1145/3453483.3454078.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://aclanthology.org/2022.emnlp-main.39/.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *Advances in Neural Information Processing Systems*, 28, 2015.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proc. ACM Program. Lang.*, 2(POPL):60:1–60:29, 2018. doi: 10.1145/3158148. URL https://doi.org/10.1145/3158148.

Glenn Shafer. *Probabilistic expert systems*. SIAM, 1996.

Chung-chieh Shan and Norman Ramsey. Exact Bayesian Inference by Symbolic Disintegration. *Principles of Programming Languages*, 2017. ISSN 07308566. doi: 10.1145/3009837.3009852.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Benjamin Sherman, Jesse Michel, and Michael Carbin. λ_s: computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.*, 5(POPL):1–31, 2021. doi: 10.1145/3434284. URL https://doi.org/10.1145/3434284.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I Wang. Natural language to code translation with execution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2022. URL https://aclanthology.org/2022.emnlp-main.231/.

Richard Shin and Benjamin Van Durme. Few-shot semantic parsing with language models trained on code. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2022. URL https://aclanthology.org/2022.naacl-main.396/.

Richard Shin, Christopher Lin, Sam Thomson, Charles Chen Jr, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. Constrained language models yield few-shot semantic parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://aclanthology.org/2021.emnlp-main.608/.

Artem Sobolev and Dmitry Vetrov. Importance Weighted Hierarchical Variational Inference, May 2019. URL http://arxiv.org/abs/1905.03290. arXiv:1905.03290 [cs, stat].

Rebecca C. Steorts, Rob Hall, and Stephen E. Fienberg. A Bayesian Approach to Graphical Record Linkage and Deduplication. *Journal of the American Statistical Association*, 111(516):1660–1672, 2016. ISSN 1537274X. doi: 10.1080/01621459.2015.1105807.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf.

Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. Learning proposals for probabilistic programs with inference combinators. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, pages 1056–1066. PMLR, December 2021a. URL https://proceedings.mlr.press/v161/stites21a.html. ISSN: 2640-3498.

Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. Learning proposals for probabilistic programs with inference combinators. In *Uncertainty in Artificial Intelligence*, pages 1056–1066. PMLR, 2021b.

Fahim Tajwar, Anikait Singh, Archit Sharma, Rafael Rafailov, Jeff Schneider, Tengyang Xie, Stefano Ermon, Chelsea Finn, and Aviral Kumar. Preference fine-tuning of LLMs should leverage suboptimal, on-policy data. In *International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/tajwar24a.html.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? A study on the impact of format restrictions on performance of large language models. *arXiv preprint arXiv:2408.02442*, 2024. URL https://arxiv.org/pdf/2408.02442.

Maria A Terres, Aiyou Chen, Rachel Zhou, and Claire M McLeod. Behavioral event detection and rate estimation for autonomous vehicle evaluation. *Applied Stochastic Models in Business and Industry*, 39(5):662–683, 2023.

David Tolpin, Jan Willem Van De Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. In *ACM International Conference Proceeding Series*, 2016. ISBN 9781450347679. doi: 10.1145/3064899.3064910.

Minh-Ngoc Tran, Marcel Scharth, Michael K Pitt, and Robert Kohn. Importance sampling squared for bayesian inference in latent variable models. *arXiv preprint arXiv:1309.3339*, 2013.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022. URL https://arxiv.org/pdf/2211.14275.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. SynCode: Improving LLM code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*, 2024. URL https://arxiv.org/pdf/2403.01632.

Arash Vahdat and Jan Kautz. Nvae: A deep hierarchical variational autoencoder. *Advances in Neural Information Processing Systems*, 33:19667–19679, 2020.

Matthijs Vákár. Denotational correctness of forward-mode automatic differentiation for iteration and recursion. *arXiv preprint arXiv:2007.05282*, 2020.

Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proceedings of the ACM on Programming Languages*, 3 (POPL):1–29, 2019.

Tim Vieira, Ben LeBrun, Mario Giulianelli, Juan Luis Gastaldi, Brian DuSell, John Terilla, Timothy J O'Donnell, and Ryan Cotterell. From language models over tokens to language models over characters. *arXiv preprint arXiv:2412.03719*, 2024. URL https://arxiv.org/pdf/2412.03719.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems*, 2024a. URL https://openreview.net/forum?id=B4tkwuzeiY&noteId=BaPOkLl42Y.

Di Wang, Jan Hoffmann, and Thomas Reps. Sound probabilistic inference via guide types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 788–803, 2021.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, 2024b. URL https://aclanthology.org/2024.acl-long.510.pdf.

Théophane Weber, Nicolas Heess, Lars Buesing, and David Silver. Credit assignment techniques in stochastic computation graphs. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2650–2660. PMLR, 2019.

David Weininger. SMILES, a chemical language and information system. *Journal of Chemical Information and Computer Sciences*, 28(1), 1988. URL https://pubs.acs.org/doi/pdf/10.1021/ci00057a005.

B Wellner, A McCallum, F Peng, and M Hay. An integrated, conditional model of information extraction and coreference with application to citation matching. *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 593–601, 2004.

Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

Michael Wick, Sameer Singh, Harshal Pandya, and Andrew McCallum. A joint model for discovering and linking entities. *AKBC 2013 - Proceedings of the 2013 Workshop on Automated Knowledge Base Construction, Co-located with CIKM 2013*, pages 67–71, 2013. doi: 10.1145/2509558.2509570.

Brandon T Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023. URL https://arxiv.org/pdf/2307.09702.

Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Language*, 8(3-4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

John Winn, John Guiver, Sam Webster, Yordan Zaykov, Martin Kukla, and Dany Fabian. Alexandria : Unsupervised High-Precision Knowledge Base Construction using a Probabilistic Program. pages 1–20, 2017. doi: 10.475/123.

Lionel Wong, Gabriel Grand, Alexander K Lew, Noah D Goodman, Vikash K Mansinghka, Jacob Andreas, and Joshua B Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought. *arXiv preprint arXiv:2306.12672*, 2023.

Frank D. Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 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*, volume 33 of *JMLR Workshop and Conference Proceedings*, pages 1024–1032. JMLR.org, 2014. URL http://proceedings.mlr.press/v33/wood14.html.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*, 2024. URL https://arxiv.org/pdf/2405.14333.

Liang Xiong, Barnabás Póczos, Jeff Schneider, Andrew Connolly, and Jake VanderPlas. Hierarchical probabilistic models for group anomaly detection. *Journal of Machine Learning Research*, 15:789–797, 2011. ISSN 15324435. doi: 10.1184/r1/6475763.

Wei Xiong, Hanze Dong, Chenlu Ye, Ziqi Wang, Han Zhong, Heng Ji, Nan Jiang, and Tong Zhang. Iterative preference learning from human feedback: Bridging theory and practice for RLHF under KL-constraint. In *International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/xiong24a.html.

Kevin Yang and Dan Klein. FUDGE: Controlled text generation with future discriminators. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021. URL https://aclanthology.org/2021.naacl-main.276/.

Yi Yang and Jacob Eisenstein. A log-linear model for unsupervised text normalization. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2013. URL https://aclanthology.org/D13-1007/.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2018. URL https://aclanthology.org/D18-1425.

Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *International Conference on*

Machine Learning. Proceedings of Machine Learning Research, 2023a. URL https://proceedings.mlr.press/v202/zhang23g/zhang23g.pdf.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024. URL https://openreview.net/pdf?id=CxHRoTLmPX.

Maosen Zhang, Nan Jiang, Lei Li, and Yexiang Xue. Language generation via combinatorial constraint satisfaction: A tree search enhanced Monte-Carlo approach. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020. URL https://aclanthology.org/2020.findings-emnlp.115/.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=Lr8cOOtYbfL.

Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022a. doi: 10.1145/3498677. URL https://doi.org/10.1145/3498677.

Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proceedings of the ACM on Programming Languages*, 6(POPL): 1–28, 2022b.

Bo Zhao, Benjamin I. P. Rubinstein, Jim Gemmell, and Jiawei Han. A Bayesian approach to discovering truth from conflicting sources for data integration. *Proceedings of the VLDB Endowment*, 5(6):550–561, feb 2012. ISSN 2150-8097. doi: 10.14778/2168651.2168656. URL http://dl.acm.org/doi/10.14778/2168651.2168656.

Stephen Zhao, Rob Brekelmans, Alireza Makhzani, and Roger Baker Grosse. Probabilistic inference in language models via twisted sequential Monte Carlo. In *Proceedings of the International Conference on Machine Learning*, 2024. URL https://proceedings.mlr.press/v235/zhao24c.html.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

Rui Zheng, Shihan Dou, Songyang Gao, Yuan Hua, Wei Shen, Binghai Wang, Yan Liu, Senjie Jin, Qin Liu, Yuhao Zhou, Limao Xiong, Lu Chen, Zhiheng Xi, Nuo

Xu, Wenbin Lai, Minghao Zhu, Cheng Chang, Zhangyue Yin, Rongxiang Weng, Wensen Cheng, Haoran Huang, Tianxiang Sun, Hang Yan, Tao Gui, Qi Zhang, Xipeng Qiu, and Xuanjing Huang. Secrets of RLHF in large language models part I: PPO. *arXiv preprint arXiv:2307.04964*, 2023. URL https://arxiv.org/pdf/2307.04964.

Tan Zhi-Xuan, Jordyn L. Mann, Tom Silver, Joshua B. Tenenbaum, and Vikash K. Mansinghka. Online Bayesian goal inference for boundedly-rational planning agents. *Advances in Neural Information Processing Systems*, 2020-December, 2020. ISSN 10495258.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

Banghua Zhu, Michael Jordan, and Jiantao Jiao. Principled reinforcement learning with human feedback from pairwise or K-wise comparisons. In *International Conference on Machine Learning*. Proceedings of Machine Learning Research, 2023. URL https://proceedings.mlr.press/v202/zhu23f/zhu23f.pdf.

Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019. URL https://arxiv.org/pdf/1909.08593.

Heiko Zimmermann, Hao Wu, Babak Esmaeili, and Jan-Willem van de Meent. Nested Variational Inference. November 2021. URL https://openreview.net/forum?id=kBrHzFtwdp.

Matt Zucker, James Kuffner, and Michael Branicky. Multipartite rrts for rapid replanning in dynamic environments. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 1603–1609. IEEE, 2007.

Max Zuo, Francisco Piedrahita Velez, Xiaochen Li, Michael L Littman, and Stephen H Bach. Planetarium: A rigorous benchmark for translating text to structured planning languages. *arXiv preprint arXiv:2407.03321*, 2024. URL https://arxiv.org/pdf/2407.03321.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order Bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60:1–60:29, December 2017. doi: 10.1145/3158148. URL https://dl.acm.org/doi/10.1145/3158148.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages*,

2(ICFP):83:1–83:29, July 2018. doi: 10.1145/3236778. URL https://dl.acm.org/doi/10.1145/3236778.