Stochastic Lazy Knowledge Compilation for Inference in Discrete Probabilistic Programs

MADDY BOWERS*, Massachusetts Institute of Technology, USA
ALEXANDER K. LEW*, Massachusetts Institute of Technology, USA and Yale University, USA
JOSHUA B. TENENBAUM, Massachusetts Institute of Technology, USA
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology, USA
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology, USA

We present new techniques for exact and approximate inference in discrete probabilistic programs, based on two new ways of exploiting lazy evaluation. First, we show how knowledge compilation, a state-of-the art technique for exact inference in discrete probabilistic programs, can be made lazy, enabling asymptotic speed-ups. Second, we show how a probabilistic program's lazy semantics naturally give rise to a division of its random choices into subproblems, which can be solved in sequence by sequential Monte Carlo with locally-optimal proposals automatically computed via lazy knowledge compilation. We implement our approach in a new tool, Pluck, and evaluate its performance against state-of-the-art approaches to inference in discrete probabilistic languages. We find that on a suite of inference benchmarks, lazy knowledge compilation can be faster than state-of-the-art approaches, sometimes by orders of magnitude.

CCS Concepts: • **Mathematics of computing** → *Probabilistic reasoning algorithms*; • **Computing methodologies** → *Knowledge representation and reasoning*.

Additional Key Words and Phrases: probabilistic program, exact inference, sequential Monte Carlo, laziness

1 INTRODUCTION

Given a probabilistic program and a possible outcome, *inference* refers to the related tasks of (1) quantifying the total probability with which the program generates the given outcome, and (2) inferring *how* the program could have generated the given outcome (i.e., sampling from or otherwise characterizing the *posterior distribution* over executions, conditioned on the outcome). Intuitively, inference requires reasoning about a program's execution in a *query-directed* way: starting from the observed outcome, we must look backward through the program to find the choices and computations that could have produced the observed result. Early research on inference in probabilistic programs identified *laziness* as a useful tool in this endeavor [30, 31, 43]. Lazy evaluation is naturally query-directed, and truncates executions as soon as the query's result is known, effectively marginalizing any random choices not relevant to the query at hand. Researchers observed that inference algorithms such as path enumeration, importance sampling, and variable elimination could be much more efficient when implemented lazily [30].

However, with the development of more sophisticated approaches to inference in probabilistic programs, based on state-of-the-art static analysis techniques [2, 7, 24, 26, 36, 42, 49, 59] and Monte Carlo or variational inference algorithms [3, 4, 11, 35, 60], laziness has largely been abandoned. Part of the issue is that researchers have only recently developed a rigorous foundation for reasoning about lazy probabilistic programs [12].[1] Another problem is that, for many modern inference

*Both authors contributed equally to this research.

[1]At least one PPL that moved away from laziness has explicitly cited the difficulty of reasoning compositionally about the meanings of lazy programs as a key obstacle [44, §5.5].

Authors' addresses: Maddy Bowers, mlbowers@csail.mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Alexander K. Lew, alexlew@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA and Yale University, New Haven, CT, USA; Joshua B. Tenenbaum, jbt@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Armando Solar-Lezama, asolar@csail.mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Vikash K. Mansinghka, Massachusetts Institute of Technology, Cambridge, MA, USA, vkm@mit.edu.

techniques, it is unclear how laziness can be exploited. Simple techniques such as importance sampling directly sample possible program paths, and it is relatively straightforward to sample lazy paths rather than eager ones. Many modern approaches, by contrast, look more like compilers or model checkers than interpreters, and it is less clear how to "lazify" a compiler. A third factor in the move away from laziness is the rise of programmable inference architectures, which reify executions in user-facing traces that record assignments for all random choices in a program [3, 4, 11, 33, 37, 38, 46, 51]. Relative to a trace, there are no irrelevant choices for laziness to marginalize.

In this work, we revisit lazy inference in light of modern advances in both the theory of lazy probabilistic programming [12] and the practice of inference [6, 19, 26]. We restrict our attention to *discrete* probabilistic programs, but otherwise the language we consider is quite expressive, with recursion, higher-order functions, and iso-recursive types (§3). In this context, we first develop and establish the correctness of a lazy variant of *knowledge compilation*, a state-of-the-art technique for exact inference in discrete probabilistic programs (§4). By avoiding some forms of unnecessary work during the compilation process, lazy knowledge compilation can provide asymptotic speed-ups over standard knowledge compilation for many programs. Next, we develop a new family of approximate inference algorithms that exploit laziness to automatically identify natural *inference subproblems*, each of which contains only the random choices necessary to lazily resolve one component of the program's output (§5). These subproblems form the basis of a sequential Monte Carlo algorithm that uses lazy knowledge compilation within each subproblem to automatically compute locally optimal proposals. Our approach also supports a new form of *programmability* for approximate inference, letting the user influence the size and order of the lazily constructed subproblems. Our empirical results (§6) suggest that incorporating laziness into modern approaches to inference can yield significant performance benefits relative to non-lazy versions.

Contributions. This paper contributes:

(1) *Lazy knowledge compilation*, a new algorithm for efficiently compiling discrete probabilistic programs to binary decision diagrams, facilitating fast exact inference (§4).
(2) *Lazy programmable subproblem Monte Carlo* (LPSMC), a new family of programmable approximate inference algorithms that exploits laziness to identify tractable subproblems in intractable inference queries, and uses lazy knowledge compilation to exactly solve the subproblems (§5).
(3) *Theoretical validation.* We establish the soundness of lazy knowledge compilation with respect to a denotational semantics of our language in quasi-Borel predomains [12, 54] (Thm. 4.7).
(4) *Empirical evaluation.* We implement our proposed approach in a new compiler called Pluck, and perform an extensive empirical evaluation (§6). On a suite of benchmark problems from the literature, we find that our proposed algorithms can significantly outperform standard knowledge compilation as implemented in state-of-the-art discrete PPLs, though on programs where laziness is less useful our approach can also introduce overhead.

2 OVERVIEW

We now present an overview of our approach in the context of two running examples, illustrated in Figs. 1 and 2. First, consider the probabilistic program in Fig. 1, which models the process by which a typist noisily renders a latent string s, stochastically inserting, substituting, and deleting characters. Given an input string s, *perturb*(s) has support over an infinite set of output strings: every output s' is reachable by some sequence of typos, and generally by many sequences of typos.

Below the main program, we illustrate several queries to Pluck's inference engine. The first is a **marginal** query, which computes the exact distribution of a given expression's possible outcomes. Our query quantifies the probability with which particular input string (`"lazy"`) is transformed by *perturb* into a particular observed output (`"lucky"`). Computing such queries efficiently is crucial for

```
;; model of typos

maybeInsert s =
  if flip 0.99 then s else
    cons(rchar(), maybeInsert(s))

maybeCons (c, cs) =
  if flip 0.99 then cons(c, cs) else cs

maybeSub c =
  if flip 0.99 then c else rchar()

perturb s = match s with
  nil() ⇒ maybeInsert(nil())
  cons(c, cs) ⇒
    maybeInsert(
      maybeCons(maybeSub(c),
        perturb(cs)))

;; querying a marginal distribution

marginal(eq?(perturb("lazy"), "lucky"))
  ⇒ true: 1.57e-10 | false: 0.999…

;; querying a posterior distribution

posterior(if flip 0.5 then "goat" else "bat",
          λw.eq?(perturb(w), "gat"))
  ⇒ goat: 0.962 | bat: 0.038

posterior(perturb("pluck"),
          λs.length(s) < 3))
  ⇒ pk: 0.098 | lk : 0.098 | · · · | xz: 1.52e-7
```

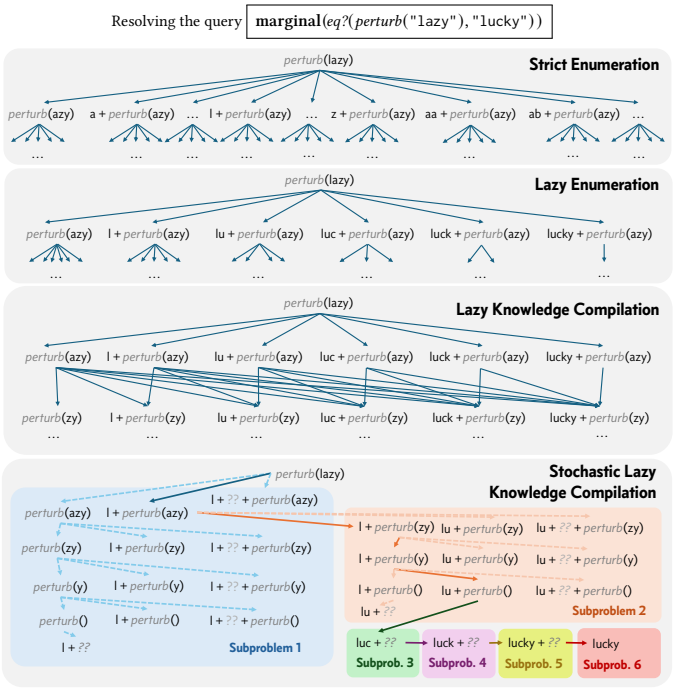


Fig. 1. **A recursive probabilistic program modeling random typos of a string.** Here, *rchar*() generates a random character at uniform from the alphabet. Each iteration of *perturb* may insert characters, delete a character, or substitute a character before recursing on the remainder of the string.

the performance of larger approximate inference algorithms for data cleaning [32]. Pluck returns in milliseconds with the answer: the Boolean test evaluates to *true* with probability 1.57×10^{-10}.

In addition to marginal distribution queries, Pluck can also solve *posterior distribution* queries, which compute the conditional distribution of a query expression's possible outcomes, *given* that a user-specified predicate holds of the result. For example, our second query conditions on the observation that a typist produced the string `"gat"`, and asks for the posterior distribution over the intended word (assuming it was either `"bat"` or `"goat"` with equal prior probability). Pluck determines that under our model, `"goat"` is roughly 26 times more likely.[2]

Eager and Lazy Enumeration. To illustrate the relevance of laziness for efficient inference, we first consider its impact on the simplest approach to inference in discrete programs: *path enumeration*. Given a query (for example, **marginal**(*eq?*(*perturb*(`"lazy"`), `"lucky"`))), enumeration exhaustively explores every possible way the query expression may execute, and computes a sum of path probabilities across all paths that result in each outcome. Under the usual, eager evaluation strategy for functional probabilistic programs, our example program can execute in infinitely many ways; we illustrate this in the top right panel of Fig. 1, with the space of executions depicted as an infinitely wide tree. Because it must fully explore this tree, (eager) path enumeration never terminates.

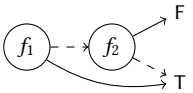
Intriguingly, however, a simple change to the algorithm lets us apply it to this query, and more generally, can make inference faster for a wide range of problems. The key insight is that under a *lazy* evaluation order, our query does *not* have infinitely many executions: with laziness, *eq?* checks

[2]From `"bat"`, *perturb* would need to have explicitly generated a g (from 26 options); from `"goat"` it can simply delete the o.

equality incrementally, immediately returning false if *perturb* generates a prefix that disagrees with `"lucky"`. This behavior *prunes* the vast space of possible executions, marginalizing infinitely many random choices that do not affect the query. Early probabilistic programming languages (e.g., the Stochastic Lisp of Koller et al. [31]; IBAL [43]; HANSEI [30]) supported lazy enumeration, i.e. exhaustive search over this reduced space of lazy executions. In this example, note that the search tree is still exponentially large in the lengths of the strings (Fig. 1, "**Lazy Enumeration**").

Knowledge Compilation. Knowledge compilation is a state-of-the-art approach to inference in discrete probabilistic programs [6, 19, 26, 29], based on a reduction from inference to *weighted model counting*: computing a weighted count of the satisfying assignments for a particular Boolean formula. Given a program, knowledge compilation compositionally constructs *formulae* encoding the conditions under which the program returns each possible outcome. The variables in the formula correspond to coin flips in the source program. For example, for the program

$$\textbf{let } x = \textbf{flip}(0.4), y = x \vee \textbf{flip}(0.7) \textbf{ in if } x \textbf{ then } y \textbf{ else } not(y)$$



knowledge compilation generates the formula $\phi = \mathsf{f}_1 \vee \neg \mathsf{f}_2$ to encode when the program returns true (in this case, exactly when its first coin flip is heads or its second coin flip is tails).

The key to knowledge compilation's efficiency is the data structure it uses to represent the formulae, the *binary decision diagram* (BDD) [5]. BDDs are directed acyclic graphs, where each node is either a literal (T or F), or a *decision node*, labeled with the name of a variable (e.g. f_1), and featuring two outgoing edges—the edge to take if the variable is true (drawn as a solid line), and if it is false (drawn as a dotted line). As BDDs are constructed, they are simplified so that they satisfy a set of invariants, which ensure that Boolean functions have *canonical* representations as BDDs. This makes it possible to quickly recognize *shared structure* and to exploit it by merging isomorphic subgraphs. In Boolean functions encoding the logic of probabilistic programs, such shared structure occurs often, thanks in part to independence relationships between sets of random variables. Given a formula represented as a BDD, weighted model counting is linear in the size of the BDD.

Eagerness of Knowledge Compilation. Because only finitely many of the choices *perturb* makes influence the result of the *eq?* query, there exists a finite BDD over the coin flips in *perturb* that encodes the condition *eq?*(*perturb*(`"lazy"`), `"lucky"`). Even better, the BDD has size polynomial in the lengths of the queried strings (`"lazy"` and `"lucky"` in our example). The BDD exploits the fact that, conditioned on an already-generated prefix (e.g. `"lu"`) and a remainder-of-the-input-string to process (e.g. `"zy"`), the particular choices we made to generate the prefix are independent of the choices that will be used to generate the suffix. Unfortunately, despite the existence of this finite BDD, standard approaches to knowledge compilation do not find it, and instead loop infinitely on the program from Fig. 1. The problem is that the knowledge compilation algorithm processes the program in an eager manner. For example, when compiling the expression **let** $x = e_1$ **in** e_2, today's knowledge compilation-based PPLs compile e_1 whether or not x appears free in e_2.

Lazy Knowledge Compilation. Our key contribution in this work is a way to make knowledge compilation lazy. In our running example, lazy knowledge compilation successfully builds a polynomial-size BDD and solves the *perturb* query in milliseconds. To illustrate in more detail how lazy knowledge compilation works, we turn temporarily to our second example (Fig. 2, left)—a program with fewer moving parts where the algorithm's execution can be more easily visualized.

The program in Fig. 2 generates a random *sorted* list of natural numbers. At each iteration, it flips a coin to decide whether to add another element to the list. If so, it generates the next element by adding a geometrically distributed random number to the previous element of the list. The query is

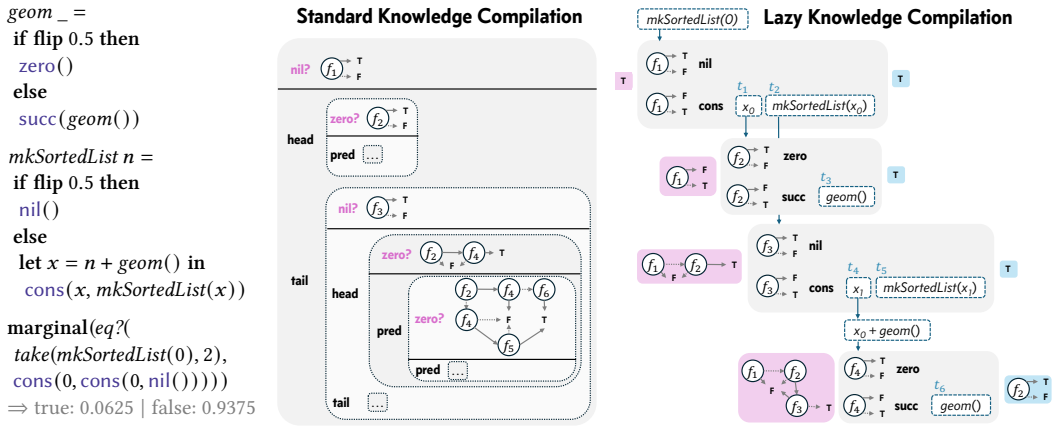


Fig. 2. **Eager versus lazy knowledge compilation on a sorted list generator.**

to compute the probability that the first two elements of the generated list exist and are equal to $[0, 0]$. Note that like *perturb*, *mkSortedList* has infinitely many possible execution paths.

Standard knowledge compilation is *eager* in that it immediately attempts to construct a symbolic representation of the distribution of *mkSortedList(0)*, without regard for how that expression is probed by the query at hand. To represent inductive types like lists and natural numbers, knowledge compilation-based PPLs (e.g., Dice.jl [19]) use encodings like the one illustrated in the middle panel of Fig. 2.[3] These encodings have two parts, separated in our illustration by a horizontal black line: (1) a BDD encoding when each constructor is active (nil vs. cons, zero vs. succ), and (2) encodings in the same format for constructor arguments (the head and tail of a list, or the predecessor of a natural number). In the *mkSortedList* program, there is no bound on the sizes of the generated natural numbers or on the length of the generated list, and so such an encoding would be infinite. We can make the program finite by adding a recursion bound on both *geom* and *mkSortedList*, but the representation generated by standard knowledge compilation will still be enormous, and must carefully track the many dependencies between different values in the list—work that is in principle unnecessary to answer the query.

We propose *lazy* knowledge compilation (Fig. 2, right), which compiles the user's program in stages as demanded by the query:

(1) *eq?* begins by pattern matching its argument, which forces us to evaluate *mkSortedList(0)*—but only up to *weak-head normal form*. Our compiler determines that there are two possible results: nil() and cons(t_1, t_2), where t_1 and t_2 are *thunks* storing yet-to-be-compiled expressions. Each possible result is *guarded* by a predicate in the form of a BDD. Our compiler also tracks a *path condition* (highlighted in pink) encoding the conditions under which the input expression is evaluated at all. Because *eq? always* forces its arguments, initially the path condition is T.
(2) We consider both cases (nil and cons). If *mkSortedList(0)* returns nil(), *eq?* immediately returns false. Otherwise, *eq?* evaluates the head of the list t_1. We invoke the compiler on t_1, with an updated path condition indicating that this expression is not always evaluated. The compiler determines that the first element ($x_0 = 0 + geom()$) can be zero() or succ(t_3), each with a guard.

[3]Dice.jl also supports fixed-width binary integers [6]; here, we focus on the inductive type of (unbounded) natural numbers.

(3) If the head evaluates to succ(t_3), *eq?* immediately returns false; otherwise, *eq?* forces the tail of the list t_2. When we compile the tail, we again expand the path condition, to reflect that the head was zero(). We again see two options: nil() or cons(t_4, t_5).
(4) Finally, if the tail is non-empty, *eq?* forces t_4, $x_1 = x_0 + geom()$. Because the path condition encodes that $x_0 = 0$, the compiler can reduce this to $geom()$. **This is a key benefit of laziness:** by the time we are compiling the second element of the list, we have already conditioned on the first element's value. The symbolic representation the compiler returns for x_1 does not capture its full, marginal distribution; it only captures its *conditional distribution*, given that we are evaluating it at all (i.e., given that the first element x_0 matched its observed value). This can save a significant amount of work. Unfortunately, it also complicates *caching*: if the thunk t_4 is forced elsewhere in the program, where the path condition does *not* imply $x_0 = 0$, it would be unsound to reuse these results. To facilitate aggressive but sound caching, our compiler also computes a *weakest validity condition*, highlighted in blue, encoding minimal requirements that a path condition must satisfy in order for previously compiled results to be soundly reused.

As we show in §6, by exploiting both laziness and independence, our approach can deliver significant performance gains over both standard knowledge compilation and lazy enumeration.

Extension to Approximate Inference. Lazy knowledge compilation compiles representations of the entire distribution over a program's possible (lazy) executions. But often, a very good estimate of the distribution can be obtained more cheaply by considering a representative sample of the program's possible executions [34]. In §5, we build on lazy knowledge compilation to obtain a powerful algorithm for lazy *approximate* inference. Our algorithm is based on sequential Monte Carlo [8], a family of sampling techniques that work by decomposing an inference problem into a sequence of subproblems. Our key observation is that, for models over compound data (e.g., tuples, lists, or trees), lazy evaluation provides a natural way to decompose inference into a sequence of useful subproblems: each subproblem contains exactly those random choices necessary to resolve the next component of the program output. Even when exact inference is too expensive for a given problem, it may still be tractable to solve each such subproblem exactly.

Returning to Fig. 1, the bottom right panel illustrates our approach on the *perturb* query. Instead of compiling the entire query *eq?*(*perturb*("lazy"), "lucky"), we first compile only the check that the first *character* of each string is equal. From the many lazy execution paths consistent with this first condition, we *sample* a representative path (a linear-time operation in the size of the compiled BDD). This representative path is then used as the path condition in a second stage of compilation, where we check that the both of the first *two* characters of each string are equal. This process of compiling-then-sampling repeats until the entire equality predicate has been run. By incorporating *importance weights* based on the weighted model counts of the BDDs generated at each step, we obtain unbiased estimates of the marginal distribution of the query (i.e., the probabilities that *eq?* returns true and that it returns false).

3 A CORE CALCULUS FOR LAZY, RECURSIVE, PROBABILISTIC PROGRAMS

Listing 3.1 gives the grammar of λ_{PLUCK}, our core calculus for lazy, recursive, discrete probabilistic programs. λ_{PLUCK} is a simply typed λ-calculus with iso-recursive types,[4] extended with the construct **flip** r for Bernoulli coin flips with probability r of heads.[5]

[4]The syntax $\mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n$ is used for both recursive and non-recursive sum types; if non-recursive, α will not occur in the τ_i. We implicitly unroll **match** scrutinees, and roll ℓ_i e into a value of the corresponding recursive type.
[5]λ_{PLUCK} has no inline **observe** construct because posterior conditioning is implemented as a top-level query (see Fig. 1).

Listing 3.1 Grammar of λ_{PLUCK}.

Syntactic category		**Productions**
Types τ	::=	$1 \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\ \ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n$
Expressions e	::=	$() \mid x \mid \textbf{flip}\ r \mid \ell\ e \mid \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n x_n \Rightarrow e_n\} \mid$ $(e_1, e_2) \mid \textbf{match}\ e_1\ \textbf{with}\ (x_1, x_2) \Rightarrow e_2 \mid \lambda x.\ e \mid e_1\ e_2 \mid Y(\lambda x_1.\lambda x_2.e)$
Real constants r	$\in$	$[0, 1]$
Sugar		**Desugaring**
$\mathbb{B}$	=	$\mu\_.\ \text{true}\ 1 + \text{false}\ 1$
$\mathbb{N}$	=	$\mu\alpha.\ \text{zero}\ 1 + \text{succ}\ \alpha$
$\mathbb{L}\ \tau$	=	$\mu\alpha.\ \text{nil}\ 1 + \text{cons}\ (\tau \times \alpha)$
$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	=	$(\lambda x.e_2)\ e_1$
$\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	=	$\textbf{match}\ e\ \textbf{with}\ \text{true}\ \_ \Rightarrow e_1 \mid \text{false}\ \_ \Rightarrow e_2$

Listing 3.2 Selected rules for semantics of λ_{PLUCK} expressions $\Gamma \vdash e : \tau$. Recall that for $\omega \in \Omega$, we write ω_i for $\omega[\varepsilon \cdot i] = \lambda\sigma.\omega(\varepsilon \cdot i \cdot \sigma)$, the i^{th} child of the rose tree ω. See Listing A.4 for all rules.

Expression		*Outcome on random seed* ω *in environment* γ
$[\![\Gamma \vdash () : 1]\!](\gamma, \omega)$	=	$\star$
$[\![\Gamma \vdash \ell_i\ e : \mu\alpha.\ \ell_1\tau_1 + \cdots + \ell_n\tau_n]\!](\gamma, \omega)$	=	$\text{roll}\ (\text{in}_i([\![e]\!](\gamma, \omega)))$
$[\![\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n x_n \Rightarrow e_n\} : \tau]\!](\gamma, \omega)$	=	$\begin{cases} \bot_\tau & \text{unroll}([\![e]\!](\gamma, \omega_0)) = \bot \\ [\![e_1]\!](\gamma[x_1 \mapsto v], \omega_1) & \text{unroll}([\![e]\!](\gamma, \omega_0)) = \text{in}_1\ v \\ \cdots & \cdots \\ [\![e_n]\!](\gamma[x_n \mapsto v], \omega_n) & \text{unroll}([\![e]\!](\gamma, \omega_0)) = \text{in}_n\ v \end{cases}$
$[\![\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!](\gamma, \omega)$	=	$([\![e_1]\!](\gamma, \omega_1), [\![e_2]\!](\gamma, \omega_2))$
$[\![\Gamma \vdash e_1\ e_2 : \tau_2]\!](\gamma, \omega)$	=	$[\![e_1]\!](\gamma, \omega_0)([\![e_2]\!](\gamma, \omega_1), \omega_2)$
$[\![\Gamma \vdash Y(\lambda x_1.\lambda x_2.e) : \tau_1 \to \tau_2]\!](\gamma, \omega)$	=	$\text{fix}(\lambda f.\lambda(y, \omega').[\![e]\!](\gamma[x_1 \mapsto f, x_2 \mapsto y], \omega'))$
$[\![\Gamma \vdash \textbf{flip}\ r : \mathbb{B}]\!](\gamma, \omega)$	=	$\begin{cases} \text{roll}(\text{in}_1 \star) & \omega(\varepsilon) \le r \\ \text{roll}(\text{in}_2 \star) & \textit{otherwise} \end{cases}$

3.1 Random-Seed Semantics

Space of Random Seeds. Inspired by the approach of Dash et al. [12], we model lazy probabilistic programs as mapping infinite *random seeds* ω to possible outcomes. Formally, our space of random seeds is $\Omega = [0, 1]^{\mathbb{N}^*}$: that is, a random seed ω is an infinite collection of real numbers (in $[0, 1]$), labeled by (possibly empty) sequences of natural numbers σ. Equivalently, we can think of ω as an infinitely wide, infinitely deep tree (a *rose tree*) whose nodes are labeled with real numbers. The value at the root of the tree is $\omega(\varepsilon)$, the value at the third child is $\omega(\varepsilon \cdot 3)$, the value at that node's second child is $\omega(\varepsilon \cdot 3 \cdot 2)$, and so on. We write $\omega[\sigma]$ for the subtree of ω rooted at path σ (i.e., $\omega[\sigma_1](\sigma_2) = \omega(\sigma_1 +\!\!+ \sigma_2)$), and $\omega_i = \omega[\varepsilon \cdot i]$ for the i^{th} child of ω.

Semantics. In our semantics, types τ denote *quasi-Borel predomains with least element* $[\![\tau]\!]$, sets of values equipped with additional structure for reasoning about measurability and recursion [54]. Expressions e of type τ denote maps $[\![e]\!]$ from *environments* γ and *random seeds* ω to *outcomes*, which are values in $[\![\tau]\!]$. The definition of $[\![\cdot]\!]$ is given in Listing 3.2. In the remainder of this section, we give a high-level tour of our semantics, deferring the full treatment to Appendix A.

Deterministic expressions. Deterministic expressions e have denotations $[\![e]\!](\gamma,\omega)$ that do not depend at all on the random seed ω. For example, $[\![x]\!](\gamma,\omega) = \gamma[x]$, where $\gamma[x]$ looks up the value of variable x in environment γ, and $[\![()]\!](\gamma,\omega) = \star$, where $\star \in [\![1]\!]$ is the unit value.

Values of sum type. For a (possibly recursive) sum type $\tau := \mu\alpha.\, \ell_1\tau_1 + \cdots + \ell_n\tau_n$, if $v \in [\![\tau_i[\alpha \mapsto \tau]]\!]$, then $\mathsf{roll}(\mathsf{in}_i\ v) \in [\![\tau]\!]$.[6] For example, $[\![\mathsf{true}()]\!](\gamma,\omega) = \mathsf{roll}(\mathsf{in}_1\ \star)$, because true is the first constructor of $\mathbb{B} = \mu\_.\mathsf{true}\ 1 + \mathsf{false}\ 1$. As another example, the natural number 1 is $[\![\mathsf{succ}(\mathsf{zero}())]\!](\gamma,\omega) = \mathsf{roll}(\mathsf{in}_2(\mathsf{roll}(\mathsf{in}_1\ \star)))$, because succ and zero are the second and first constructors of $\mathbb{N}$, respectively.

Coin flips. We have that $[\![\mathbf{flip}\ r]\!](\gamma,\omega) = \begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega(\varepsilon) \le r \\ \mathsf{roll}(\mathsf{in}_2\ \star) & \omega(\varepsilon) > r\end{cases}$. That is, **flip** yields $[\![\mathsf{true}()]\!](\gamma,\omega)$ when the uniform random number at the *root* of the rose tree, $\omega(\varepsilon)$, is less than r.

Rose tree paths. In compound expressions, each subexpression is evaluated on a *subtree* of the overall random seed ω. For example, we have the rule $[\![(e_1,e_2)]\!](\gamma,\omega) = ([\![e_1]\!](\gamma,\omega_1),[\![e_2]\!](\gamma,\omega_2))$. Thus, coin flips that occur in different parts of the program depend on random numbers from different parts of the rose tree. Concretely, for example, we have that

$$[\![(\mathbf{flip}\ r_1, \mathbf{flip}\ r_2)]\!](\gamma,\omega) = \begin{cases}(\mathsf{roll}\ (\mathsf{in}_1\ \star), \mathsf{roll}\ (\mathsf{in}_1\ \star)) & \omega(\varepsilon\cdot 1) \le r_1 \wedge \omega(\varepsilon\cdot 2) \le r_2 \\ (\mathsf{roll}\ (\mathsf{in}_1\ \star), \mathsf{roll}\ (\mathsf{in}_2\ \star)) & \omega(\varepsilon\cdot 1) \le r_1 \wedge \omega(\varepsilon\cdot 2) > r_2 \\ (\mathsf{roll}\ (\mathsf{in}_2\ \star), \mathsf{roll}\ (\mathsf{in}_1\ \star)) & \omega(\varepsilon\cdot 1) > r_1 \wedge \omega(\varepsilon\cdot 2) \le r_2 \\ (\mathsf{roll}\ (\mathsf{in}_2\ \star), \mathsf{roll}\ (\mathsf{in}_2\ \star)) & \omega(\varepsilon\cdot 1) > r_1 \wedge \omega(\varepsilon\cdot 2) > r_2\end{cases}.$$

For each coin flip in a program, a unique *path* σ identifies the random value $\omega(\sigma)$ it depends on.

Function abstraction and application. The function type $\tau_1 \to \tau_2$ is inhabited by *stochastic maps*, i.e. functions from $[\![\tau_1]\!] \times \Omega$ to $[\![\tau_2]\!]$. For example:

$$[\![\lambda x_1.((\mathbf{flip}\ r, x_1), x_2)]\!](\gamma,\omega) = \lambda(x_1,\omega').\begin{cases}((\mathsf{roll}(\mathsf{in}_1\ \star), x_1), \gamma[x_2]) & \omega'(\varepsilon\cdot 1\cdot 1) \le r \\ ((\mathsf{roll}(\mathsf{in}_2\ \star), x_1), \gamma[x_2]) & \omega'(\varepsilon\cdot 1\cdot 1) > r\end{cases}.$$

Note that the free variable x_2 is looked up in γ, and that the outcome does not depend on ω, reflecting the fact that the λ-term itself deterministically produces a stochastic function; only *applying* the function will yield random results. This is achieved by the rule for application, $[\![e_1\ e_2]\!](\gamma,\omega) = [\![e_1]\!](\gamma,\omega_0)([\![e_2]\!](\gamma,\omega_1),\omega_2)$, which splits off a fresh seed ω_2 for the call.

Recursion and non-termination. Our types τ come equipped with partial orders $\le_\tau$ and distinguished *least elements* $\bot_\tau$, which serve as the denotations of non-terminating computations. Recursive definitions are handled in the standard way, via least fixed points of increasing chains of denotations. For example, $[\![Y(\lambda x_1.\lambda x_2.\mathbf{if}\ \mathbf{flip}\ 0.5\ \mathbf{then}\ \mathsf{zero}()\ \mathbf{else}\ \mathsf{succ}(x_1()))]\!](\gamma,\omega)$ is the limit of the sequence

$$\begin{aligned}
f_0 &= \bot_{1\to\mathbb{N}} = \lambda(_,\omega').\bot_{\mathbb{N}} \\
f_1 &= \lambda(_,\omega').\begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega'(\varepsilon\cdot 0) \le 0.5 \\ \mathsf{roll}(\mathsf{in}_2(f_0(\star,\omega'[\varepsilon\cdot 2\cdot 2]))) & \omega'(\varepsilon\cdot 0) > 0.5\end{cases} = \lambda(_,\omega').\begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega'(\varepsilon\cdot 0) \le 0.5 \\ \mathsf{roll}(\mathsf{in}_2(\bot_{\mathbb{N}})) & \omega'(\varepsilon\cdot 0) > 0.5\end{cases} \\
f_2 &= \lambda(_,\omega').\begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega'(\varepsilon\cdot 0) \le 0.5 \\ \mathsf{roll}(\mathsf{in}_2(f_1(\star,\omega'[\varepsilon\cdot 2\cdot 2]))) & \omega'(\varepsilon\cdot 0) > 0.5\end{cases} \\
&= \lambda(_,\omega').\begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega'(\varepsilon\cdot 0) \le 0.5 \\ \mathsf{roll}(\mathsf{in}_2(\mathsf{roll}(\mathsf{in}_1\ \star))) & \omega'(\varepsilon\cdot 0) > 0.5 \wedge \omega'(\varepsilon\cdot 2\cdot 2\cdot 0) \le 0.5 \\ \mathsf{roll}(\mathsf{in}_2(\mathsf{roll}(\mathsf{in}_2(\bot_{\mathbb{N}})))) & \omega'(\varepsilon\cdot 0) > 0.5 \wedge \omega'(\varepsilon\cdot 2\cdot 2\cdot 0) > 0.5\end{cases} \\
f_3 &= \ldots
\end{aligned}$$

[6] Here, in_i injects a value in $[\![\tau_i[\alpha \mapsto \tau]]\!]$ into the disjoint union $\sqcup_i [\![\tau_i[\alpha \mapsto \tau]]\!]$, and roll is a bijection (with inverse unroll) from this disjoint union to the recursive type $[\![\tau]\!]$.

Listing 4.1 Syntax and selected typing rules for guarded weak-head normal forms; see Listing B.1.

$$
\begin{array}{rcl}
\Omega\text{-predicates } \phi & \in & \Omega \Rightarrow 2 \\
\text{Rose tree paths } \sigma & ::= & \varepsilon \mid \sigma \cdot n \text{ (for } n \in \mathbb{N}) \\
\text{Environments } \gamma & ::= & \varepsilon \mid \gamma[x \mapsto t] \\
\text{Finite guarded sets } S(\text{pattern}) & ::= & \{y_{\phi_y}\}_{y \in Y} \text{ (for } Y \text{ a finite set of terms with syntax } \textit{pattern}) \\
\text{Thunks } t & ::= & S(\langle e, \gamma, \sigma \rangle) \\
\text{Weak-head normal forms } v & ::= & \star_\phi \mid (t_1, t_2) \mid S(\ell\ t) \mid S(\langle x.e, \gamma \rangle)
\end{array}
$$

$$
\textsc{Guarded-Thunk}\quad \frac{\forall\, \langle e_y, \gamma_y, \sigma_y \rangle \in Y.\ \exists\, \Gamma_y.\ \left(\Gamma_y \vdash e_y : \tau \wedge \gamma_y :_E \Gamma_y\right) \qquad (\phi_y)_{y \in Y} \text{ mutually exclusive}}{\{y_{\phi_y}\}_{y \in Y} :_T \tau}
$$

$$
\textsc{Env-}\Gamma\quad \frac{\gamma :_E \Gamma \qquad t :_T \tau}{\gamma[x \mapsto t] :_E (\Gamma, x : \tau)} \qquad \textsc{VUnit}\quad \frac{}{\star_\phi :_V 1}
$$

$$
\textsc{VSum}\quad \frac{\forall\, (\ell_{i_y} t_y) \in Y.\ t_y :_T \tau_{i_y}[\alpha \mapsto \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n] \qquad (i_y)_{y \in Y} \text{ distinct} \qquad (\phi_y)_{y \in Y} \text{ mutually exclusive}}{\{y_{\phi_y}\}_{y \in Y} :_V \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n}
$$

Note that because our language is lazy, non-termination of a sub-expression does not imply non-termination for the whole expression. For example, if $[\![e_1]\!](\gamma, \omega_1) = \bot_\tau$ and $[\![e_2]\!](\gamma, \omega_2) = \star$, then $[\![(e_1, e_2)]\!](\gamma, \omega) = (\bot_\tau, \star)$ and $[\![\textbf{match}\ (e_1, e_2)\ \textbf{with}\ (x_1, x_2) \Rightarrow x_2]\!](\gamma, \omega) = \star$, reflecting that a call-by-need interpreter never attempts to evaluate e_1 and thus avoids its looping behavior.

3.2 Distributional Semantics

We now fix a probability distribution μ_Ω on our space of random seeds Ω: the uniform distribution on rose trees [12]. For $\omega \sim \mu_\Omega$, each $\omega(\sigma)$ is distributed according to $\mathrm{Uniform}(0, 1)$, independent of all other nodes in ω. Given a program $\Gamma \vdash e : \tau$, we define $[\![e]\!]_P(\gamma) := [\![e]\!](\gamma, -)_* \mu_\Omega$, so that $[\![e]\!]_P$ is a (quasi-Borel) probability kernel from $[\![\Gamma]\!]$ to $[\![\tau]\!]$. Unlike in some other approaches [54, e.g.], we do not use *sub*-probability kernels to model possibly non-terminating programs. Our programs all denote probability kernels, which may or may not place mass on $\bot$, the outcome representing non-termination. Note that programs may place positive mass on partial *values* (e.g., $(3, \bot)$) even if they place no mass on $\bot$. Operationally, such programs can be understood as halting almost surely under a lazy evaluation strategy, but computing values that may be partial.

4 LAZY KNOWLEDGE COMPILATION

Lazy knowledge compilation is an algorithm for efficiently compiling a probabilistic program to a concrete, symbolic representation of the distribution it denotes. We first tour our symbolic representation (§4.1), then the lazy knowledge compilation algorithm itself (§4.2).

4.1 Lazy Symbolic Representations for Distributions

When a lazy interpreter evaluates an expression $e : \tau$, it computes only up to *weak-head normal form*: the top-level constructor is resolved, but the remainder of the value is left unevaluated. A key idea in our approach is that our symbolic representations can similarly encode *distributions* only up to weak-head normal form, tracking the distributions of top-level constructors, but leaving distributions of arguments unevaluated.

Listing 4.1 gives the syntax of our symbolic representations v, and Listing 4.2 gives their semantics. As in §3, we think of distributions on a space $[\![\tau]\!]$ as maps $\Omega \to [\![\tau]\!]$. Generally speaking, our

Listing 4.2 Selected rules from the semantics of guarded weak-head normal forms; see Listing B.2.

$$[\![\gamma[x \mapsto t]]\!]_E(\omega) = [\![\gamma]\!]_E(\omega)[x \mapsto [\![t]\!]_T(\omega)] \qquad [\![\star_\phi]\!]_V(\omega) = \begin{cases} \star & \text{if } \phi(\omega) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\{\langle e_y, \gamma_y, \sigma_y\rangle_{\phi_y}\}_{y\in Y}]\!]_T(\omega) = \begin{cases} [\![e_y]\!]([\![\gamma_y]\!]_E(\omega), \omega[\sigma_y]) & \text{if } \phi_y(\omega) \text{ for some } y \in Y \\ \bot_\tau & \text{otherwise} \end{cases}$$

$$[\![\{(\ell_{i_y}\ t_y)_{\phi_y}\}_{y\in Y}]\!]_V(\omega) = \begin{cases} \mathsf{roll}(\mathsf{in}_{i_y}\ [\![t_y]\!]_T(\omega)) & \text{if } \phi_y(\omega) \text{ for some } y \in Y \\ \mathsf{roll}(\bot) & \text{otherwise} \end{cases}$$

symbolic representations work by partitioning Ω into disjoint regions carved out by symbolic predicates $\phi : \Omega \to 2$, and in each region, specifying a concrete value up to weak-head normal form. As a simple example, consider a distribution over $[\![1]\!]$, which assigns some mass to $\star$ and some to $\bot$. The symbolic representation $\star_\phi$ precisely captures which random seeds ω lead to the outcome $\star$ (as opposed to $\bot$) in the form of the symbolic predicate ϕ. We write $\star_\phi :_V 1$ to indicate that $\star_\phi$ is a well-formed representation of a distribution over 1, and $[\![\star_\phi]\!]_V : \Omega \to [\![1]\!]$ for the denotation it represents.

Thunks, Environments, and Closures. A *singleton thunk* $\langle e, \gamma, \sigma\rangle$ represents an ω-dependent value whose symbolic representation we have not yet computed. It combines an expression $\Gamma \vdash e : \tau$ (the program code we will need to process if we ever force this value) with (1) an *environment* γ over the free variables in Γ (assigning each variable name to its *own* thunk), and (2) a rose-tree path σ indicating what part of the randomness in Ω this value depends on.

Example 4.1. Consider the expression $e :=$ **if** (**flip** 0.5) **then** $\ell_1\ e_1$ **else** $\ell_2\ e_2$. Its denotation $[\![e]\!](\gamma, \omega)$ is $\mathsf{roll}(\mathsf{in}_1([\![e_1]\!](\gamma, \omega_1)))$ when $\omega(\epsilon \cdot 0) \le 0.5$ and $\mathsf{roll}(\mathsf{in}_2([\![e_2]\!](\gamma, \omega_2)))$ otherwise. Our symbolic representation will explicitly represent the distribution of the top-level constructor, but will leave e_1 and e_2 unevaluated, inside thunks. Concretely, when we compile this expression in a symbolic environment γ, we compute the representation

$$\{(\ell_1\langle e_1, \gamma, \varepsilon \cdot 1\rangle)_{\omega(\epsilon\cdot 0)\le 0.5}, (\ell_2\langle e_2, \gamma, \varepsilon \cdot 2\rangle)_{\neg(\omega(\epsilon\cdot 0)\le 0.5)}\}.$$

This representation is a *set* of sub-representations, each guarded by a predicate on ω. Each sub-representation is a label ℓ applied to a thunk, which remembers the environment γ as well as the *path* σ that determines what sub-seed $\omega[\sigma]$ the expression should be evaluated with respect to.

If both branches were to use the same constructor ℓ_1, we would merge the two thunks into a *full thunk*: a finite set of singleton thunks $y = \langle e_y, \gamma_y, \sigma_y\rangle$ all of the same type, each guarded by a predicate ϕ_y. We write $\{\langle e_y, \gamma_y, \sigma_y\rangle_{\phi_y}\}_{y\in Y}$ for the full thunk with possibilities $y \in Y$.

Example 4.2. Now suppose our expression $e :=$ **if** (**flip** 0.5) **then** $\ell_1\ e_1$ **else** $\ell_1\ e_2$. Its denotation $[\![e]\!](\gamma, \omega)$ is $\mathsf{roll}(\mathsf{in}_1([\![e_1]\!](\gamma, \omega_1)))$ when $\omega(\epsilon \cdot 0) \le 0.5$ and $\mathsf{roll}(\mathsf{in}_1([\![e_2]\!](\gamma, \omega_2)))$ otherwise. Our symbolic representation is now

$$\{(\ell_1\ \{\langle e_1, \gamma, \varepsilon \cdot 1\rangle_{\omega(\epsilon\cdot 0)\le 0.5}, \langle e_2, \gamma, \varepsilon \cdot 2\rangle_{\neg(\omega(\epsilon\cdot 0)\le 0.5)}\})_{\mathsf{T}}\}.$$

Note that the guard on the overall form $\ell_1\ t$ is T, because the value has constructor ℓ_1 no matter the seed ω. The uncertainty has been pushed inward, yielding a set of possible singleton thunks.

The semantics of thunks and environments are given by a mutual recursion. For environments, $[\![\gamma]\!]_E(\omega)$ maps each variable x to $[\![\gamma(x)]\!]_T(\omega)$—the meaning of the thunk stored for variable x. For full thunks, $[\![\{\langle e_y, \gamma_y, \sigma_y\rangle_{\phi_y}\}_{y\in Y}]\!]_T(\omega)$ branches on which predicate ϕ_y holds of ω, then evaluates $[\![e_y]\!]$ in environment $[\![\gamma_y]\!]_E$ on random seed $\omega[\sigma_y]$. The base case of this mutual recursion is the empty environment. To represent ω-dependent values of function type $\tau_1 \to \tau_2$, we use finite sets Y of *singleton closures* $y = \langle x_y.e_y, \gamma_y\rangle$, guarded by predicates ϕ_y. A singleton closure pairs a function body e_y with an environment $\gamma_y :_E \Gamma_y$ such that $\Gamma_y, x_y : \tau_1 \vdash e_y : \tau_2$. Unlike thunks, closures do not store a rose-tree path σ, because functions use new parts of ω every time they are called.

Products and Sums. A distribution over a product type is represented as a pair of thunks (t_1, t_2). The representation of a distribution on (recursive) sum types is a set of possibilities $y = \ell_{i_y}\ t_y$, each guarded by a predicate ϕ_y. We require that the predicates are mutually exclusive, and that the i_y are all distinct. The predicates tell us when each constructor ℓ_{i_y} is active, and the thunks t_y are the (delayed) constructor arguments, of type $\tau_{i_y}[\alpha \mapsto \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n]$. When no predicate ϕ_y holds of a random seed ω, the outcome for ω is implicitly $\mathsf{roll}(\bot)$.

Encoding Predicates as Binary Decision Diagrams. We work with a restricted class of predicates $\phi : \Omega \to 2$ that can be represented concretely as *binary decision diagrams*. In particular, our predicates arise as finite Boolean formulae over *atomic predicates* $\omega(\sigma) \le r$, where every σ always appears with the same r. For each atomic predicate $\omega(\sigma) \le r$, we instantiate a fresh Boolean variable $b_{\sigma,r}$. Then a composite predicate ϕ is represented as a directed acyclic graph, where each node is either a *decision node* labeled with the name of a variable, or a *terminal node* labeled with T or F. Decision nodes have two outgoing edges, an edge to follow when the variable is true and an edge to follow when it is false. We work with *reduced*, *ordered* BDDs, which means that: (1) isomorphic subgraphs are merged; (2) decision nodes whose high and low edges point to the same child node are removed; and (3) along every path from the root, variables are encountered in the same, fixed order. This ensures that every predicate has a *canonical* representation as a BDD. We use the Rust library `rsdd` for constructing and manipulating BDDs. In the worst case, the BDDs representing a program's semantics may have size exponential in the number of variables, but they can often be much smaller by exploiting program structure, leading to efficient inference [6, 19, 26, 29].

4.2 Lazy Knowledge Compilation

Given a program $e : \tau$, the goal of knowledge compilation is to automatically compute a representation $v :_V \tau$ such that $[\![e]\!] = [\![v]\!]_V$.[7] Our algorithm, given in Listing 4.3, takes four inputs that evolve as the algorithm recursively invokes itself on (possibly open) subterms:

- The expression $\Gamma \vdash e : \tau$ to compile.
- An environment $\gamma :_E \Gamma$ containing thunks for any free variables in e.
- A *path condition* $\phi : \Omega \to 2$. When invoking lazy knowledge compilation at the top level on a closed term, the path condition is T, but in recursive calls to compile subterms, the path condition evolves to encode the conditions under which the subterm e must be evaluated.
- A *rose-tree path* $\sigma \in \mathbb{N}^*$. At the top level, the rose-tree path is ε. In recursive calls, it indicates which portion of the rose tree ω the sub-expression e uses.

If the algorithm terminates, it returns:

- A representation $v :_V \tau$ of the distribution of e in environment $[\![\gamma]\!]_E$ on random seed $\omega[\sigma]$.

[7] Exact inference is in general uncomputable. An intuitive condition for when our algorithm succeeds is that a *lazy* sampler for e must generate a bounded number of fresh coin flips, whereas standard knowledge compilation succeeds only if an *eager* sampler's traces are bounded. The Appendix presents a small modification to our algorithm that enables it to compute bounds or unbiased estimates of distributions whenever the source program almost surely terminates.

Listing 4.3 Selected rules from the big-step operational semantics of lazy knowledge compilation. If $\gamma :_E \Gamma$ and $\Gamma \vdash e : \tau$, then the judgment $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ means that lazy knowledge compilation, run on expression e, environment γ, path condition $\phi : \Omega \to 2$, and rose-tree path $\sigma \in \mathbb{N}^*$, yields value $v :_V \tau$, and that the result v is valid for all ω in $\phi_u \supseteq \phi$. See Listing B.3 for all rules.

$$\textsc{False}\ \frac{\phi \implies \mathsf{F}}{\gamma, \phi, \sigma \vdash e \rightsquigarrow \varepsilon \dashv \mathsf{F}} \qquad \textsc{Flip}\ \frac{}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow \{(\mathsf{true}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\omega(\sigma) \le r}, (\mathsf{false}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\neg(\omega(\sigma) \le r)}\} \dashv \mathsf{T}}$$

$$\textsc{CacheHit}\ \frac{\gamma, \phi^*, \sigma \vdash e \rightsquigarrow v \dashv \phi_u \qquad \phi \implies \phi_u}{\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u} \qquad \textsc{Constructor}\ \frac{}{\gamma, \phi, \sigma \vdash \ell_i\ e \rightsquigarrow (\ell_i\ \langle e, \gamma, \sigma \rangle_{\mathsf{T}})_{\mathsf{T}} \dashv \mathsf{T}} \qquad \textsc{Abs}\ \frac{}{\gamma, \phi, \sigma \vdash \lambda x.\ e \rightsquigarrow \langle x.e, \gamma \rangle_{\mathsf{T}} \dashv \mathsf{T}}$$

$$\textsc{Var}\ \frac{\gamma(x) = \{\langle e_i, \gamma_i, \sigma_i \rangle_{\phi_i}\}_{i \in I} \qquad \gamma_i, \phi \wedge \phi_i, \sigma_i \vdash e_i \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \mathsf{T}) = (v, \phi_u)}{\gamma, \phi, \sigma \vdash x \rightsquigarrow v \dashv \phi_u}$$

$$\textsc{App}\ \frac{\begin{array}{c}\gamma, \phi, \sigma \cdot 0 \vdash e_1 \rightsquigarrow \{\langle x_i.e_i^f, \gamma_i \rangle_{\phi_i}\}_{i \in I} \dashv \phi_u \\ \gamma_i[x_i \mapsto \langle e_2, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}], \phi \wedge \phi_i, \sigma \cdot 2 \vdash e_i^f \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \phi_u) = (v, \phi_u^*)\end{array}}{\gamma, \phi, \sigma \vdash e_1\ e_2 \rightsquigarrow v \dashv \phi_u^*}$$

$$\textsc{MatchSum}\ \frac{\begin{array}{c}\gamma, \phi, \sigma \cdot 0 \vdash e \rightsquigarrow \{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y \in Y} \dashv \phi_u \\ \gamma[x_y \mapsto v_y], \phi \wedge \phi_y, \sigma \cdot i_y \vdash e_{i_y} \rightsquigarrow v_y^* \dashv \phi_y^u \qquad \mathsf{join}(\{(v_y^*, \phi_y, \phi_y^u)\}_{y \in Y}, \phi_u) = (v, \phi_u^*)\end{array}}{\gamma, \phi, \sigma \vdash \mathbf{match}\ e\ \mathbf{with}\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n \rightsquigarrow v \dashv \phi_u^*}$$

- A *validity condition* ϕ_u encoding the conditions under which v is a correct representation of e's semantics. The input path condition ϕ always implies the returned validity condition ϕ_u, but the validity condition may be significantly more general (i.e., impose fewer constraints on ω).

The judgment $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ means that, when invoked on expression e, environment γ, path condition ϕ, and rose-tree path σ, lazy knowledge compilation succeeds and returns the representation v and the validity condition ϕ_u. The rules can be read bottom-up-and-down-again: given the inputs to the algorithm (left side of the conclusion), it makes various recursive calls (premises), and uses their results to compute its results (right side of the conclusion). As the algorithm runs, all predicates are represented jointly as a *multi-rooted* BDD—that is, isomorphic subgraphs are shared in memory across all predicates constructed by the algorithm, with shared Boolean variables representing atomic predicates $\omega(\sigma) \le r$. We give brief intuition for the rules:

- False: If the path condition is false (F), then this expression is unreachable, and laziness dictates that we not compile it.[8] We return the empty representation ε of the appropriate type: $\star_{\mathsf{F}}$ for $e : 1$, $(\emptyset, \emptyset)$ when $e : \tau_1 \times \tau_2$, and $\emptyset$ for sum and function types.
- Flip: A coin flip may evaluate to true() or false(). The predicates guarding each option are the atomic predicate $\omega(\sigma) \le r$ and its negation.
- Constructor, Abs: These rules apply to expressions that, under lazy evaluation, immediately produce a value. These values are deterministic, so all guards are T, indicating no dependence on ω. Pairs and recursive functions are handled analogously; see Listing B.3 for details.

[8] Even if other rules also match the input expression, our algorithm always applies rule False when it can, under the principle that any unnecessary compilation should be avoided.

- Var, App, MatchSum: Because thunks, closures, and sums are all represented as sets $\{y_{\phi_y}\}_{y \in Y}$ of possible values, guarded by mutually exclusive predicates ϕ_y, the elimination forms for these types all involve a similar pattern, where we invoke a recursive call for each possibility $y \in Y$, and then join the resulting values v_y. The full implementation of join is given in the Appendix (Listing B.4), but its behavior can be understood by examining the specification it satisfies:

  Lemma 4.3. *Let $f : \Omega \to [\![\tau]\!]$ and suppose that for some mutually exclusive collection of predicates ϕ_i on Ω, there exist $f_i : \Omega \to [\![\tau]\!]$ such that for all ω satisfying an additional predicate ϕ_u, we have*

  $$f(\omega) = \begin{cases} f_i(\omega) & \text{if } \phi_i(\omega) \\ \bot_\tau & \text{if } \forall i, \neg\phi_i(\omega) \end{cases}.$$

  Further suppose that $(v_i, \phi_i^u)_{i \in I}$ are such that whenever $\phi_i^u(\omega)$ holds, $f_i(\omega) = [\![v_i]\!]_V(\omega)$. Then $\mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \phi_u)$ *yields (v, ϕ_u^*) such that:*
  (1) $(\phi_u \wedge \phi_u^i \wedge \phi_i)(\omega) \implies \phi_u^*(\omega)$ for all $i \in I$;
  (2) $(\phi_u \wedge \forall i \in I, \neg\phi_i)(\omega) \implies \phi_u^*(\omega)$; and
  (3) for all ω such that $\phi_u^*(\omega)$ holds, $[\![v]\!]_V(\omega) = f(\omega)$.

  Put simply, this lemma says that join correctly merges representations of each piece of a piecewise function into a representation of the overall function. For example, consider the Var rule. The function f in this case is the denotation $\lambda\omega.[\![x]\!]([\![\gamma]\!]_E, \omega) = [\![\gamma(x)]\!]_T(\omega)$. The premise of the Var rule establishes that $\gamma(x)$ is $\{(t_y)_{\phi_y}\}_{y \in Y}$, which means that f is given piecewise on pieces carved out by the predicates ϕ_y. The Var rule recursively compiles representations v_y for each piece, then uses join to combine them into a representation of f.

Guarded Caching. Without *caching*, lazy evaluation strategies can easily duplicate computations, blowing up the complexity of a program. The same need for caching arises in our setting: we do not want to recompile the expression in a thunk anew every time the corresponding variable is used. In standard lazy evaluation, the typical solution is to equip thunks with local storage, and after they are evaluated once, to cache their computed values for reuse if the thunk is evaluated again. But a wrinkle in our setting prevents us from directly employing this solution. Lazy knowledge compilation uses the path condition ϕ to avoid compiling a *complete* representation of the distribution of an expression. For example, if the path condition includes that a randomly generated number n is less than 5, the compilation of an expression that uses n needs only consider execution paths reachable when $n < 5$. However, if the same expression is later evaluated under a different path condition, the earlier-compiled result may no longer be valid. Therefore, our thunks store *guarded* cached results: when we first compile the expression inside a thunk, we cache the result alongside the *validity condition* ϕ_u computed during compilation. Future evaluations of the thunk check whether the path condition ϕ implies the cached validity condition; if so, the previously compiled value can be reused, and if not, the compiler is invoked anew.

This caching is reflected in the rule CacheHit, which allows us to establish $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ simply by checking that ϕ *implies* the validity condition ϕ_u returned by a previous invocation of our algorithm on e, regardless of the specific path condition ϕ^* active during the original computation.

4.3 Correctness

Our first correctness result states that the representations computed by lazy knowledge compilation correctly reflect the distribution of the input expression.

Lemma 4.4. *Suppose $\Gamma \vdash e : \tau$ and $\gamma :_E \Gamma$. Then for any predicate $\phi : \Omega \to 2$ and rose tree path $\sigma \in \mathbb{N}^*$, if $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$, then:*

(1) $\phi(\omega)$ *implies* $\phi_u(\omega)$*, and*
(2) for all $\omega \in \Omega$ *such that* $\phi_u(\omega)$ *holds, we have* $[\![v]\!]_V(\omega) = [\![e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma])$.

Specializing this lemma to the case when e has sum type (e.g. $\mathbb{B}$), we see that running lazy knowledge compilation produces a *guard formula* ϕ characterizing the seeds ω for which the program returns each constructor of the sum. We next show that a *weighted model count* of this formula—a linear-time operation in the size of the binary decision diagram—calculates the exact probability with which $[\![e]\!]$ generates a value with the corresponding top-level constructor.

Definition 4.5 (Weighted model count). Let $\mathcal{X}$ be a set of Boolean variables, let $\phi : \{0,1\}^{\mathcal{X}} \to \{0,1\}$ be a Boolean formula over the variables, and let $wt : \mathcal{X} \to [0,1]$ be a function assigning to each Boolean variable a continuous weight. Then the *weighted model count* $\mathsf{WMC}(\phi, wt)$ is defined as

$$\mathsf{WMC}(\phi, wt) := \sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}} \mid \phi(\mathbf{x})\}} \prod_{b \in \mathcal{X}} wt(b)^{\mathbf{x}(b)} \cdot (1 - wt(b))^{1-\mathbf{x}(b)}.$$

Recall that the predicates $\phi : \Omega \to \{0,1\}$ generated by lazy knowledge compilation are defined over *atomic predicates* of the form $\omega(\sigma) \leq r$. Thus, they can be equivalently viewed either Boolean formulae over Boolean variables $b_{(\sigma,r)}$ representing these atomic predicates. Formally, we write $\mathcal{X}(\phi) := \{b_{(\sigma,r)} \mid \text{"}\omega(\sigma) \leq r\text{"} \text{ appears as an atomic predicate in } \phi\}$ for the set of these Boolean variables relevant to a given predicate ϕ. We then write $\lfloor\phi\rfloor : \{0,1\}^{\mathcal{X}(\phi)} \to \{0,1\}$ for the *formula* that ϕ encodes over these Boolean variables. Note that for all ω, $\phi(\omega) = \lfloor\phi\rfloor(((b_{(\sigma,r)} \mapsto \mathbf{1}_{[0,r]}(\omega(\sigma)))_{b_{(\sigma,r)} \in \mathcal{X}(\phi)})$. Our next lemma is that the weighted model count of the formula $\lfloor\phi\rfloor$ with respect to a particular weight function yields the probability under μ_Ω that the predicate ϕ holds.

Lemma 4.6. *Let* ϕ *be a predicate on* Ω *defined in terms of atomic predicates* $\omega(\sigma) \leq r$*. Then* $\mathbb{P}_{\omega \sim \mu_\Omega}[\phi(\omega)] = \mathsf{WMC}(\lfloor\phi\rfloor, (b_{(\sigma,r)} \mapsto r)_{b_{(\sigma,r)} \in \mathcal{X}(\phi)})$.

Combining these lemmas yields our overall correctness result for marginal probability evaluation:[9]

Theorem 4.7. *Let* e *be a closed term of sum type. If* $\varepsilon, \mathsf{T}, \varepsilon \vdash e \rightsquigarrow \{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y \in Y} \dashv \mathsf{T}$*, then*

$$[\![e]\!]_P(\mathsf{roll}\ \mathsf{in}_{i_y}([\![\tau_{i_y}]\!])) = \mathsf{WMC}(\lfloor\phi_y\rfloor, (b_{(\sigma,r)} \mapsto r)_{b_{(\sigma,r)} \in \mathcal{X}(\phi_y)}).$$

5 LAZY PROGRAMMABLE SUBPROBLEM MONTE CARLO (LPSMC)

We now present an algorithm for *approximate* inference based on lazy knowledge compilation (Alg. 1). Our algorithm is a variant of sequential Monte Carlo [8], a family of sampling techniques that work by decomposing an inference problem into a sequence of subproblems. A typical Pluck query runs a program and performs a Boolean test on its output, e.g. for equality with an observed outcome. To support decomposition into subqueries, we define a new type of *suspendible Booleans*:

$$\textbf{SuspendibleBool} := \mu\alpha.\mathsf{FinallyTrue}\ 1 + \mathsf{FinallyFalse}\ 1 + \mathsf{Suspend}\ \alpha.$$

A suspendible Boolean can either represent a fully computed result, or a partially computed (suspended) result. We can use it to define predicates that evaluate in stages, rather than all at once. For example, consider the following equality predicate, which checks the equality of two lists one

[9]To resolve posterior probability queries, we can reduce them to marginal probability queries. Concretely, to resolve the query **posterior**(e_1, e_2), where $e_1 : \tau_1$ and $e_2 : \tau_1 \to \mathbb{B}$, we apply lazy knowledge compilation to the expression **let** $x = e_1$ **in match** $e_2\ x$ **with** true() $\Rightarrow x$. This denotes $\perp$ when the conditioning predicate e_2 does not hold and yields the value of e_1 when the predicate does hold; performing weighted model counts for each constructor of τ_1 and renormalizing the resulting set of probabilities yields the desired posterior.

Algorithm 1 Lazy Subproblem Monte Carlo

Input: Closed query expression $\vdash e$: **SuspendibleBool**
Input: Maximum number of iterations N
Output: p_T and p_F, unbiased estimates of the probabilities, under $[\![e]\!]_P$, of truthy and falsy outcomes up to N layers of nesting (i.e., for p_T, $\{[\![\underbrace{\mathsf{Suspend}(\ldots\mathsf{Suspend}}_{i\text{ times}}(\mathsf{FinallyTrue}())\ldots)]\!] \mid 0 \le i \le N\}$).

1: $queue \leftarrow [(e, \mathsf{T}, [], \varepsilon, 1, 0)]$
2: $p_\mathsf{T} \leftarrow 0$
3: $p_\mathsf{F} \leftarrow 0$
4: **while** *queue* is not empty **do**
5: $(e, \phi, \gamma, \sigma, w, i) \leftarrow queue$.pop!() ▷ expr., path condition, env., rose-tree path, weight, depth
6: $(\mathsf{FinallyTrue}_{\phi_T}, \mathsf{FinallyFalse}_{\phi_F}, \mathsf{Suspend}(t)_{\phi_S}) \leftarrow$ **LazyKnowledgeCompilation**$(e, \gamma, \phi, \sigma)$
7: $p_\mathsf{T} \leftarrow p_\mathsf{T} + w \cdot \mathrm{WMC}(\phi \wedge \phi_T)$
8: $p_\mathsf{F} \leftarrow p_\mathsf{F} + w \cdot \mathrm{WMC}(\phi \wedge \phi_F)$
9: **if** $i < N \wedge \phi_S \neq \mathsf{F}$ **then**
10: $\phi \leftarrow \phi \wedge \phi_S$
11: $\phi_s, q \leftarrow$ **sample**(ϕ) ▷ Sample one concrete path from Suspend BDD
12: $w \leftarrow w \cdot \frac{1}{q}$ ▷ Update importance weight
13: **for** $\langle e_j, \gamma_j, \sigma_j \rangle_{\phi_j} \in t$ **do** ▷ Consider possible resumptions of suspended path
14: $queue$.push!$((e_j, \phi \wedge \phi_s \wedge \phi_j, \gamma_j, \sigma_j, w, i+1))$
15: **end for**
16: **end if**
17: **end while**
18: **return** $(p_\mathsf{T}, p_\mathsf{F})$

element at a time, suspending computation before each recursive call:

$$
\begin{aligned}
&eq?(s_1, s_2) = \textbf{match } s_1 \textbf{ with}\\
&\quad \mathsf{nil}() \Rightarrow \textbf{match } s_2 \textbf{ with } \{\mathsf{nil}() \Rightarrow \mathsf{FinallyTrue}() \mid \mathsf{cons}(\_,\_) \Rightarrow \mathsf{FinallyFalse}()\}\\
&\quad \mathsf{cons}(x, xs) \Rightarrow \textbf{match } s_2 \textbf{ with } \mathsf{nil}() \Rightarrow \mathsf{FinallyFalse}()\\
&\qquad\qquad \mathsf{cons}(y, ys) \Rightarrow \textbf{if } x = y \textbf{ then } \mathsf{Suspend}(eq?(xs, ys)) \textbf{ else } \mathsf{FinallyFalse}()
\end{aligned}
$$

We think of such a predicate as implicitly defining a subproblem decomposition strategy; in this case, for programs that generate lists, *eq?* defines each subproblem to resolve one additional element of the list. However, we could easily write other predicates that divide the task up differently, for example: (1) resolving k elements at a time; (2) resolving the length of the list before resolving its elements; (3) checking that all elements are non-zero before resolving their specific values; (4) for lists of lists (e.g. representing a grid), resolving one column or row at a time, or resolving small rectangular patches; and so on.

Given a suspendible query—i.e., a query returning a **SuspendibleBool**—we can run lazy knowledge compilation to obtain a symbolic representation of the distribution over the weak-head normal forms $(\mathsf{FinallyTrue}_{\phi_T}, \mathsf{FinallyFalse}_{\phi_F}, \mathsf{Suspend}(t)_{\phi_S})$. The guards ϕ_T and ϕ_F summarize paths through the program that immediately yield a result; we can perform weighted model counts on both guards to obtain *lower bounds* on the probabilities of true and false, respectively. We can then *estimate* the remaining mass of both true and false outcomes by *sampling* a random path through the binary decision diagram representing ϕ_S, and compiling the expressions in the suspended thunk t conditional on the sampled path. This is a standard instance of Monte Carlo estimation,

Table 1. **Exact Inference Benchmarks.** For recursive programs, *Eager* and *Dice.jl* are run with manually set optimal fuel arguments. See Table 4 (in Appendix) for additional benchmarks from Holtzen et al. [26].

Benchmark	Enumeration		Knowledge Compilation		BDD Stats (Ours)	
	Eager	*Lazy*	*Dice.jl*	*Ours*	vars	nodes
Bayesian Networks						
Cancer (5 nodes)	0.34 ms	**0.16 ms**	0.58 ms	0.27 ms	11	26
Survey (6 nodes)	1.09 ms	1.54 ms	0.81 ms	**0.48 ms**	21	71
Alarm (37 nodes)	timeout	timeout	**131.29 ms**	693.53 ms	296	94,947
Insurance (27 nodes)	timeout	timeout	**211.45 ms**	1,657.41 ms	540	101,044
Hepar2 (70 nodes)	timeout	timeout	**9.79 ms**	53.34 ms	297	3,934
Hailfinder (56 nodes)	timeout	timeout	**487.63 ms**	16,945.98 ms	1,757	65,384
Pigs (441 nodes)	timeout	timeout	11.54 ms	**7.93 ms**	64	2,851
Water (32 nodes)	timeout	698.07 ms	21.83 ms	**4.47 ms**	99	1,331
Munin (1041 nodes)	timeout	timeout	86.88 ms	**71.82 ms**	254	11,066
Network Reachability						
Diamond Network	timeout	timeout	**3.98 ms**	10.32 ms	200	400
Ladder Network	timeout	timeout	**12.99 ms**	47.68 ms	200	796
Sequence Models						
HMM (50 steps)	timeout	timeout	**2.64 ms**	12.52 ms	199	398
PCFG (23 terminals)	timeout	**7.80 ms**	9,183.68 ms	8.25 ms	67	50
String Editing (4⟶5 chars)	timeout	91.03 ms	58,343.28 ms	**8.01 ms**	154	2,760
Sorted List Gen. (8 elements)	timeout	4.51 ms	20,926.96 ms	**3.98 ms**	37	74

replacing a sum over all accepting paths through the BDD ϕ_S with a single-sample weighted estimate: for any probability distribution q with full support over the paths in ϕ_S, and any real-valued function f, we have $\sum_{path \in \phi_S} f(path) = \mathbb{E}_{path \sim q}\left[\frac{1}{q(path)} f(path)\right]$. In our case, we choose q to be proportional to the weights of each path through the BDD. Iterating this process N times gives an unbiased estimate of the total probability of the outcomes FinallyTrue(), Suspend(FinallyTrue()), Suspend(Suspend(FinallyTrue())), and so on, up to N nestings of Suspend constructors. This process is illustrated in the bottom right panel of Fig. 1. At each step, lazy knowledge compilation generates a BDD representing all paths that could reach the next suspension point (shown as dotted lines), then samples one to condition on (shown via solid lines). See Appx. D for a multi-path variant of the algorithm that uses the top K paths through ϕ_S plus one sampled path for reduced variance.

6 EVALUATION

We implement our approach in Julia, and compare to several baselines also implemented in Julia: Dice.jl, a port of the state-of-the-art Dice probabilistic programming language (which uses eager knowledge compilation); lazy enumeration (as used e.g. in HANSEI [30]); and strict enumeration (as used e.g. in WebPPL [22]).

6.1 Performance, Scalability, and Robustness of Lazy Knowledge Compilation

First, we compare our performance on several benchmark tasks (Table 1). The *Bayesian Networks* and *Network Reachability* benchmarks are adapted from the Dice repository. For each Bayesian network, the task is to compute a single marginal distribution of a node that is topologically at the bottom of the Bayes net. In network reachability, the task is to compute the probability that a packet reaches its destination through a network with a certain topology. (For the Diamond and Ladder networks, we consider networks with 100 repetitions of the given motif.) *HMM* is a 50-step

hidden Markov model with Boolean latent states and observations. *PCFG* queries the probability of a string under a PCFG with productions $S \to XY \mid YX; X \to a \mid bXX; Y \to c \mid cX \mid cS$. *String Editing* queries the probability that a source string is perturbed to a target string under the program from Fig. 1. *Sorted List Generation* queries the probability that the program from Fig. 2 generates a particular list. We use the standard definitions of inductive list and natural number types.

Results on Dice's benchmarks. The benchmarks from Dice's repository are not defined over compound data and do not exercise many of lazy knowledge compilation's strengths. In general, on such tasks we expect lazy knowledge compilation to perform slightly worse than eager knowledge compilation, as our algorithm adds several sources of overhead (e.g., computing path conditions, validity conditions, and cache checks). This overhead is most pronounced on the *Insurance* and *Hailfinder* examples, which are large Bayesian networks encoded as deeply nested if-statements. This is a worst-case pattern for our caching: the same variables appear again and again, in different deeply nested branches, where some aspect of the large path condition is relevant to how they are compiled, thus leading to cache misses. We find that disabling cache checks, validity conditions, and path conditions bring *Insurance* and *Hailfinder* to 618 ms and 1,578 ms respectively. We obtained similar runtimes when we implemented an eager variant of our knowledge compiler. Therefore, we suspect that these smaller differences in runtime stem from differences in our encoding choices (Dice.jl uses binary-encoded integers for the random variables of Bayesian networks, along with a fast bitwise if-then-else operation, while we use enums), the order of BDD operations performed by the compilers, the choice of BDD backend, and other implementation details. On some Bayesian networks, lazy knowledge compilation appears to outperform Dice.jl, but by a modest amount – likely due to these smaller differences playing out in our favor.

Results on sequence models. The sequence model benchmarks *are* defined over compound data (lists or strings), and therefore can better exploit laziness. In particular, whereas Dice.jl has to represent the full distribution of each element of a generated list, lazy knowledge compilation only has to represent the conditional distribution of each element given the observed prefix. This does not help in the Boolean-state Hidden Markov Model, where the marginal distributions of each element in the sequence are easy to represent concisely with small BDDs. In cases where the full distribution is more unweildy, however, Dice.jl is orders-of-magnitude slower than Pluck even on relatively small problem instances.

Scalability analysis. We also investigate the *scalability* of inference to larger queries. The first five plots in Fig. 3 are log-log scaling plots for benchmarks from Table 1 that have a natural "size" parameter that can be varied. We find that on examples where laziness cannot be effectively exploited, lazy knowledge compilation achieves the same polynomial-time asymptotic behavior as Dice.jl, indicated by both curves becoming eventually linear on the log-log scale, with the same slope. However, we generally incur constant-factor overhead that differs on a problem-by-problem basis. By contrast, on examples such as sorted list generation and PCFG, Dice.jl scales exponentially while lazy approaches are able to scale polynomially. In these regimes, lazy knowledge compilation sometimes has preferable asymptotic scaling to lazy enumeration (see e.g. the higher slope for lazy enumeration in the Sorted List Generation plot). For Dice.jl, query size is not the only variable of interest: we also must provide a *fuel* or recursion bound in order for Dice.jl to terminate. If the fuel is set too low, Dice.jl's answer is incorrect, whereas if it is set too high, the sixth plot in Fig. 3 illustrates that performance can degrade rapidly. (In Tab. 1, we report results for the minimum fuel value for which Dice.jl produced the right answer.) Lazy approaches automatically determine the appropriate bound to provide exact answers to a given query.

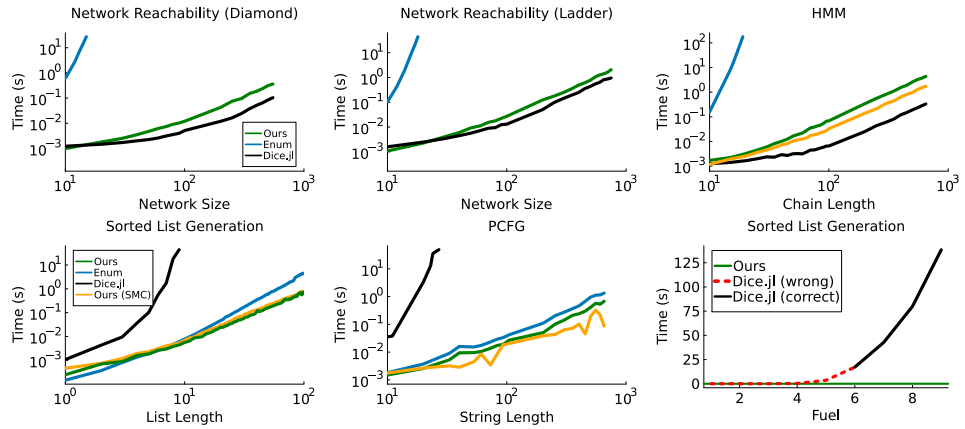


Fig. 3. **Scaling of inference.** *First five plots*: Log-log scaling plots showing performance of inference as query size grows. *Bottom right plot*: Scaling for a fixed query (size 8) as recursion bound (fuel) increases.

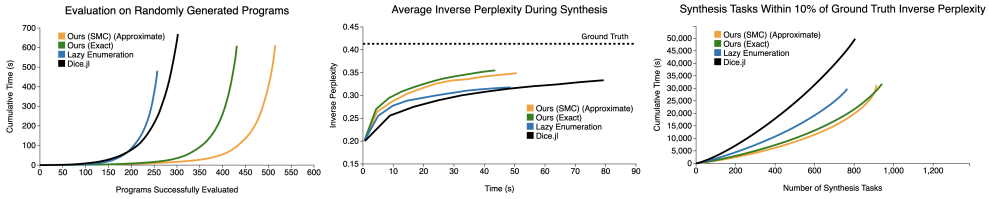


Fig. 4. **(Left)** Cactus plot for randomly generated program corpora, plotting cumulative number of programs solved by each method against cumulative time taken. Programs on which inference timed out do not count toward programs solved or cumulative time. **(Middle)** Average inverse perplexity of the outputs under the current MCMC hypothesis over time. **(Right)** Cactus plot in which success is measured as MCMC obtaining a program that yields an inverse perplexity that is only 10% (or less) below the ground truth.

Robustness analysis. To evaluate the *robustness* of our likelihood evaluator, we run it on programs randomly sampled from a grammar, so that we do not have precise control over how the programs are written. We generate 600 (*program*, *inputs*, *outputs*) triples (see Appx. F.1); the task is then to efficiently evaluate $\prod_{(input,output)} \mathbb{P}_{x\sim program(input)}[x = output]$.

For each program in our dataset, we run each method with a timeout of 24 seconds; see cactus plot in Fig. 4 (Left). Strict enumeration is excluded, as it only terminated on one program. We find that among the exact approaches, ours terminates on the most programs (72%, compared to 51% and 43% for Dice.jl and lazy enumeration, respectively). Our approximate inference algorithm (LPSMC) terminates with an unbiased estimate of the likelihood for a larger set of programs (86%). Its estimates are relatively accurate: averaged across programs, the relative root mean square error is 1.32. When the likelihood estimate was positive, the log likelihood was off by an average of 0.77.

6.2 Further Evaluation of Approximate Inference

Our scalability and robustness analyses from the previous section showed that Pluck's approximate inference algorithm, LPSMC, generally terminates more quickly on more programs than its exact inference algorithm, and that the relative error of its likelihood estimates was reasonably small. We now study the performance of approximate inference more closely via particular case studies.

Comparison to automated approximate inference in existing PPLs. Existing PPLs (e.g., Turing [20], Pyro [4], WebPPL [22], Anglican [53], and Monad-Bayes [50]) feature a handful of automated sampling algorithms for approximate inference, including importance sampling, sequential Monte Carlo, and single-site Metropolis-Hastings. However, these algorithms are poorly suited to answer our benchmark queries *as written*. For example, consider the first query from Fig. 1, which asks for the probability that *perturb*("lazy") yields "lucky". When encoding such a task directly in a language like WebPPL, one would run inference on the program **let** x = *perturb*("lazy") **in condition**(x == "lucky") and evaluate the system's *normalizing constant estimate*. While this produces an unbiased estimate of the marginal likelihood (like in LPSMC), the PPLs' Monte Carlo machinery effectively reduces to rejection sampling on these programs, because conditioning occurs at the program's end through a complex predicate, rather than through direct **observe** statements constraining primitive random samples. For this query in particular, because the conditioned event only occurs with probability 1.57×10^{-10}, the generic importance sampling and sequential Monte Carlo algorithms in WebPPL and similar languages would require tens of billions of particles to produce an accurate marginal likelihood estimate.

To achieve better performance, we can manually rewrite our programs into a form that is more amenable to automated approximate inference in existing PPLs. For example, sequential Monte Carlo implementations in systems like WebPPL, Anglican, and Turing expect programs to *interleave* random sampling with conditioning, rather than applying conditions only at the end. For programs that generate lists and then check equality against an observed list (as in both §2 examples), we could rewrite them to check elements *as they are generated*. One way to achieve this would be to rewrite the generative process explicitly as a stateful lazy generator $gen : \text{state} \to \textbf{Maybe}\ (\text{elem}, \text{state})$, then wrap it in a recursive "runner", as illustrated in Fig. 5. Note that in rewriting the program in this way, the user has *manually identified and exploited structure that LPSMC identifies and exploits automatically* using laziness. This rewriting enables WebPPL to decompose inference into a sequence of more manageable subproblems, leading to significant efficiency gains over rejection sampling. However, as the results in Table 2 attest, even with this rewriting, LPSMC still has a significant edge. For each method, the table reports runtime in milliseconds and an estimate of the *relative root mean squared error* (RMSE), $\sqrt{\mathbb{E}[((\hat{p}/p) - 1)^2]}$, where $\hat{p}$ is the algorithm's estimated likelihood and p is the true value. Even with hundreds of times more particles, WebPPL's SMC produces much higher-error estimates than LPSMC. Furthermore, small changes to constants in the program (e.g., changing the 0.99s in *perturb* to 0.995) lead to decreased performance in WebPPL, whereas LPSMC is largely unaffected. This is because, even with a reasonable division into subproblems, WebPPL still solves each subproblem with rejection sampling from the prior, whereas LPSMC uses lazy knowledge compilation to solve subproblems exactly. This difference is even more pronounced in the *mkSortedList* program from Fig. 2 (run with an eight-element observed list, as in Table 1). Whereas WebPPL requires thousands of particles to compute reasonable estimates, LPSMC is exact with just 1 particle on this program, due to the Markov structure (the solutions to each subproblem are conditionally independent, given the observed list).

Use within a probabilistic program synthesis pipeline. To evaluate whether LPSMC strikes an effective balance between efficiency and accuracy, we consider its use *as a component* within a downstream task: probabilistic program synthesis from input-output examples [41, 48, 58]. Given a set of input-output examples, synthesis searches for a probabilistic program $e : \tau_1 \to \tau_2$ that assigns *high likelihood* to the outputs, given the inputs, i.e. a program that makes the likelihood $[\![e(x)]\!]_P([x \mapsto x_i], y_i)$ high for each input-output example (x_i, y_i). A key component of the synthesis algorithm is the *likelihood evaluator*, which evaluates this likelihood for a candidate probabilistic program e. Both *efficiency* and *accuracy* are important: a faster likelihood evaluator

Lazy generator for *perturb*

```
def perturbGen(state: str):
  # Leave be or insert?
  if flip(0.99):
    if state == '':
      return None

    next_c = state.first
    state = state.rest

    # Leave be or delete?
    if flip(0.99):
      # Leave be or substitute?
      if flip(0.99):
        return (next_c, state)
      else:
        c = rchar()
        return (c, state)
    else:
      return perturbGen(state)
  else:
    c = rchar()
    return (c, state)
```

Recursive runner

```
def runner(gen, state, obs):
  # If observation is empty,
  # observe that generator is done
  if isempty(obs):
    condition(gen(state) is None)

  # Otherwise, observe that
  # generator is not done…
  else:
    res = gen(state)
    condition(res is not None)

    # …that next elem is equal…
    (elem, state) = res
    condition(elem == obs.first)

    # …and that rest of
    # generation is equal
    return runner(gen, obs.rest, state)
```

Query

```
runner(perturbGen, "lazy", "lucky")
```

Fig. 5. Python pseudocode for a rewritten version of *perturb* (from Fig. 1) that generates the perturbed string as an explicit lazy stream.

LPSMC (8 particles), *perturb* ($p = 0.99$)	
Runtime: 11.2 ms	
RMSE: 0.017	
WebPPL (2×10^3 particles), *perturb* ($p = 0.99$)	
Runtime: 527 ms	
RMSE: 2.25	
LPSMC (8 particles), *perturb* ($p = 0.995$)	
Runtime: 11.2 ms	
RMSE: 0.007	
WebPPL (2×10^3 particles), *perturb* ($p = 0.995$)	
Runtime: 512 ms	
RMSE: 3.85	
LPSMC (1 particle), *mkSortedList* ($p = 0.5$)	
Runtime: 5.94 ms	
RMSE: 0.0	
WebPPL (5×10^3 particles), *mkSortedList* ($p = 0.5$)	
Runtime: 736 ms	
RMSE: 1.06	

Table 2. Comparison of LPSMC to WebPPL SMC on rewritten programs.

lets synthesis test more candidate programs in a given amount of time, whereas an accurate likelihood evaluator is essential for correctly guiding the search toward high-scoring programs.

Fig. 4 (middle and right panels) shows the results of probabilistic program synthesis using different inference algorithms as likelihood evaluators. Each synthesis task was created by running randomly generated ground truth programs on several randomly generated inputs, and recording the sampled outputs (see Appendix G for details). In the middle panel of Fig. 4, we show the average relationship between synthesis time[10] and *inverse perplexity*,[11] a measure of how well the synthesized program fits the input-output examples. In the right-hand plot, we show a cactus plot that tracks cumulative number of synthesis tasks solved over time, where a task is considered *solved* if synthesis generates a program that achieves inverse perplexity within 10% of the ground-truth program's. Both plots show that LPSMC significantly improves the performance of synthesis relative to lazy enumeration and Dice.jl baselines, enabling us to solve more synthesis tasks more quickly. This suggests that LPSMC (here, with 1 particle) is accurate enough to guide synthesis to high-inverse-perplexity programs. However, we when compared to exact lazy knowledge compilation, LPSMC does not yield significant improvements, and even leads to a slight loss in the performance of synthesis. This suggests that on these datasets, the modest speed-up of approximate lazy inference over exact lazy inference is not worth the drop in accuracy.

Navigating runtime/accuracy trade-offs. Finally, to illustrate the levers that LPSMC users have to trade off accuracy and computational expense, we measure in Table 3 how both variables change as a function of (1) subproblem size (controlled by the user's *eq?* predicate) and (2) particle count (the parameter k in the multi-particle variant of our algorithm, Appx. D). As in standard sequential Monte Carlo, we see that error tends to be smaller with higher particle counts, although the relationship is not theoretically guaranteed to be monotonic.

[10] MCMC-based synthesis is an anytime algorithm; at each time step, the algorithm updates a current candidate program. This plot shows inverse perplexity of the program discovered after T seconds, as a function of T.

[11] If p_x is the distribution of the generated program when run on input x, then the inverse perplexity on a set of input-output examples (x_i, y_i) is $\left(\prod_{(x_i,y_i)} p_{x_i}(y_i)\right)^{\frac{1}{\sum_i |y_i|}}$ is the inverse perplexity. Here we assume the outputs are of list type, and $|y_i|$ is the length of the output list y_i.

Table 3. RMSE (smaller is better) and runtime as a function of subproblem size and particle count in LPSMC, on a four-letter *perturb* query.

	Number of particles			
Subproblem size	1 particle	2 particles	4 particles	8 particles
1-letter subproblems	6.08×10^{-1} (1.46 ms)	6.51×10^{-2} (1.59 ms)	6.46×10^{-3} (1.70 ms)	2.33×10^{-3} (2.06 ms)
2-letter subproblems	1.13×10^{-2} (1.72 ms)	8.56×10^{-4} (1.84 ms)	9.47×10^{-5} (1.87 ms)	2.39×10^{-5} (2.07 ms)

7 RELATED WORK

Laziness in Probabilistic Programming. Laziness has long been recognized as potentially valuable for inference; for example, researchers have developed lazy versions of enumeration, importance sampling, and variable elimination [30, 31, 43]. Like our approach, lazy variable elimination can exploit some forms of independence in probabilistic programs. However, to our knowledge, existing approaches to lazy variable elimination compromise on laziness and therefore do not reap its full benefits. For example, HANSEI [30]'s approach to variable elimination requires programmers to manually *reify* and *reflect* certain computations in a model, which causes HANSEI's core lazy enumeration algorithm to eagerly evaluate the entire computation and cache its result. As a different example, Figaro's lazy factored inference [45] does not track path conditions and so, whenever a variable is encountered, its full distribution is compiled into a factor graph, rather than its conditional distribution given the control flow path. As such, it can instantiate many choices in its factor graph that would never be sampled in a lazy interpretation of the program. On the theoretical side, Dash et al. [12] recently presented a new approach to reasoning about lazy probabilistic programs, upon which we base our formalization.

Lazy Predicates in Property-Based Testing. Using lazy evaluation to efficiently prune the space of values that satisfy a predicate is a well-established practice in property-based testing (PBT) [9, 16, 47]. These techniques are closely related to the use of laziness for posterior sampling in probabilistic programs, but instead of conditioning arbitrary prior distributions given as probabilistic programs, the prior is often taken to be a uniform distribution over values of some type and size. It would be interesting to see whether algorithms developed for PBT, e.g. the adaptive lazy rejection sampler of Claessen et al. [9], could be generalized to apply to arbitrary probabilistic programs; and conversely whether lazy knowledge compilation has applications to PBT.

Exact Inference in Discrete Probabilistic Programs. Our work builds on knowledge compilation-based approaches to discrete inference in probabilistic programs [19, 26, 29]. Systems like Hakaru [40], BernoulliProb [10], PSI [21], and SPPL [49] (as well as some probabilistic model-checkers, such as Storm [15]) instead compute symbolic, algebraic representations of distributions, using computer algebra expressions (Hakaru), probabilistic circuits (BernoulliProb, Storm, and SPPL), or custom symbolic representations (PSI). Many of these techniques can handle both continuous and discrete variables. These approaches are eager, and either do not accept or do not terminate on programs like *perturb* from Fig. 1. It would be interesting to develop lazy versions of these approaches, and in fact, our initial attempt at lazy inference propagated sum-product representations similar to SPPL's. We eventually focused on lazy knowledge compilation, because knowledge compilation is known to outperform several symbolic approaches [26], and because the use of BDDs lent itself to an elegant solution to the caching problem (see §4, *Guarded Caching*).

To handle certain types of general recursion, several PPLs compile programs to systems of equations, which can then be solved using iterative methods [7, 52]. Unlike our approach, which produces exact answers only if a program has finitely many lazy executions, the equation-solving

approach can compute probabilities that arise as non-trivial infinite sums. However, to make compilation efficient, researchers have had to introduce other language restrictions that would rule out many of the programs we study in §6. For example, Chiang et al. [7] require that variables of certain types are used affinely (i.e., at most once); the higher-order function *map*, which is used in four of our six synthesis DSLs, violates this constraint.

Another line of work uses information flow typing to automatically detect conditional independencies in a model, enabling efficient variable elimination [24, 36]. Li et al. [36] recently extended this technique to a language similar to ours in expressiveness,[12] and showed that their system, MaPPL, can recover polynomial-time dynamic programming algorithms for tasks such as parsing in probabilistic grammars. Interestingly, achieving acceptable performance in MaPPL appears to require manually "pushing" observations early in the generative process, which lazy evaluation accomplishes automatically. For example, the MaPPL program encoding a probabilistic grammar does not generate a string and then check that it is equal to an observed string, but rather manually interleaves observation of the string with generative random choices. It would be interesting to see whether our approach could be adapted to support a lazy version of the MaPPL compiler.

Knowledge compilation has also been applied in the setting of *probabilistic logic programs* [13, 14, 17, 18, 28, 55]. Some probabilistic logic programming languages use *backward reasoning* when grounding a program to avoid query-irrelevant deductions, a technique that bears some resemblance to lazy knowledge compilation in its query-directedness. But beyond the difference in setting (our approach applies to functional programs, not logic programs), our lazy semantics also leads to a different notion of *query-irrelevance*, which enables optimizations not captured by backward reasoning alone. For example, consider the functional (left) and logic (right) programs

$$g(x) = \textbf{flip}(0.5) \qquad\qquad 0.5 :: g(X, \textbf{true});\ 0.5 :: g(X, \textbf{false}).$$
$$y = \textbf{let}\ x\ = f()\ \textbf{in}\ g(x) \qquad\qquad y(Y) \text{ :- } f(X), g(X, Y).$$

Under our lazy semantics, the functional program's meaning does not depend on the definition of f. By contrast, we can only derive $y(\textbf{true})$ in the logic program if we can derive $f(x)$ for some x. Thus, f's definition *is* relevant to the logic programming query, and backward reasoning alone will not automatically marginalize x (as laziness does in the functional program). In practice, backward reasoning is insufficient for many of our benchmarks (e.g., Fig. 2 loops infinitely in ProbLog).

Programmable Inference. Many probabilistic programming languages feature *programmable inference* constructs for compositionally tailoring approximate inference algorithms to specific models or applications [1, 3, 4, 11, 25, 33, 37, 38, 56]. Our approximate inference algorithm is programmable in that users can control the sizes and orders of subproblems by defining *suspendible* predicates which examine different chunks of a probabilistic program's output in stages. This is in contrast to other strategies for defining subproblems, based on naming sets of *latent* variables [11, 20, 25, 37], manually marking subproblem boundaries [32], or expressing a model via recursion or iteration patterns (scans and folds, e.g.) that lead naturally to a notion of "step" [4, 11, 39].

8 DISCUSSION

Our results suggest that laziness—known from probabilistic programming's earliest days to be useful for inference—can be fruitfully combined with modern advances. However, our experience also suggests that finding the right way to "lazify" a particular technique for inference can be highly non-trivial. It is instructive to consider the key tensions that we had to resolve to obtain an efficient algorithm. On the one hand, a key benefit of laziness is that, by waiting until a variable is *used* to compile a representation of its distribution, we can avoid compiling its *whole* distribution, and

[12] Although MaPPL's formalization includes support for sum types such as lists, the MaPPL implementation does not.

instead compile its *conditional* distribution, given the control flow path along which the variable was forced. On the other hand, if a variable is used *multiple* times, along different paths, we want to avoid compiling it many times. Indeed, a distinguishing feature of practical lazy languages is their use of caching to avoid repeated evaluations of thunked expressions. To resolve this tension, our algorithm propagates *weakest validity conditions* under which a previous compilation result can be soundly reused. This solution made sense for a lazy version of knowledge compilation, which already constructs and manipulates predicates over program states; it is an open question how to adapt it for very different inference techniques, such as information-flow-typing approaches [24, 36].

Although lazy knowledge compilation has currently been implemented for only discrete probabilistic programs, we believe it should be possible to extend to hybrid programs, using ideas recently proposed by Garg et al. [19]. That work uses fixed-width binary representations for real numbers; it would be interesting to explore whether laziness would allow us to automatically discover the precision with which real numbers need to be represented in order to answer a particular query. More generally, we have mostly considered the implications of laziness for inference in this work, but as Dash et al. [12] explain, laziness also enables modeling with various kinds of infinite data; our algorithm should immediately apply to such models.

More work is needed to further reduce the overhead that we observed laziness to introduce in settings where eager knowledge compilation is already sufficient. However, our experiments suggest that it can sometimes be an acceptable tradeoff for asymptotic efficiency gains in many other programs.

ACKNOWLEDGMENTS

We would like to thank Steven Holtzen, Nadia Polikarpova, Mauricio Barba da Costa, Sam Staton, and Emanuele Sansone for helpful discussions and feedback.‘ We would also like to thank the anonymous reviewers for valuable comments and suggestions. M.B. is supported by the National Science Foundation (NSF) Graduate Research Fellowship under Grant No. 2141064. A.K.L. and V.K.M. are supported by an anonymous philanthropic gift as well as gifts from the Siegel Family Foundation that support the MIT Siegel Family Quest for Intelligence. J.B.T. is supported by AFOSR, the MIT Quest for Intelligence, the MIT-IBM Watson AI Lab, ONR Science of AI, and Siegel Family Endowment. A.S. is supported by the NSF under Grant No. 1918839. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of sponsors.

REFERENCES

[1] Eric Atkinson, Cambridge Yang, and Michael Carbin. 2018. Verifying handcoded probabilistic inference procedures. *arXiv preprint arXiv:1805.01863* (2018).

[2] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. 2022. Semi-symbolic inference for efficient streaming probabilistic programming. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1668–1696.

[3] McCoy R Becker, Alexander K Lew, Xiaoyan Wang, Matin Ghavami, Mathieu Huot, Martin C Rinard, and Vikash K Mansinghka. 2024. Probabilistic programming with programmable variational inference. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 2123–2147.

[4] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of machine learning research* 20, 28 (2019), 1–6.

[5] Randal E Bryant. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)* 24, 3 (1992), 293–318.

[6] William X Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling integer arithmetic in probabilistic programs. In *Uncertainty in Artificial Intelligence*. PMLR, 260–270.

[7] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact recursive probabilistic programming. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 665–695.

[8] Nicolas Chopin, Omiros Papaspiliopoulos, et al. 2020. *An introduction to sequential Monte Carlo*. Vol. 4. Springer.

[9] Koen Claessen, Jonas Duregard, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *Journal of Functional Programming* 25 (2015), e8. https://doi.org/10.1017/S0956796815000143

[10] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. 92–102.

[11] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*. 221–236.

[12] Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2023. Affine monads and lazy structures for Bayesian programming. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1338–1368.

[13] Luc De Raedt and Angelika Kimmig. 2015. Probabilistic (logic) programming concepts. *Machine Learning* 100 (2015), 5–47.

[14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*. IJCAI-INT JOINT CONF ARTIF INTELL, 2462–2467.

[15] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is coming: A modern probabilistic model checker. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*. Springer, 592–600.

[16] Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24*. Springer, 383–405.

[17] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. 2012. Inference in probabilistic logic programs using weighted CNF's. *arXiv preprint arXiv:1202.3719* (2012).

[18] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401.

[19] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2024. Bit Blasting Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 865–888.

[20] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*. PMLR, 1682–1690.

[21] Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: exact inference for higher-order probabilistic programs. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*. 883–897.

[22] Noah D Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages.

[23] Noah D. Goodman, Joshua B. Tenenbaum, Jacob Feldman, and Thomas L. Griffiths. 2008. A Rational Analysis of Rule-Based Concept Learning. *Cogn. Sci.* 32, 1 (2008), 108–154. https://doi.org/10.1080/03640210701802071

[24] Maria I Gorinova, Andrew D Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional independence by typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 1 (2021), 1–54.

[25] Shivam Handa. 2019. *Composable inference metaprogramming using subproblems*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[26] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[27] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101, 1-2 (1998), 99–134.

[28] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming* 11, 2-3 (2011), 235–262.

[29] Angelika Kimmig, Bernd Gutmann, Theofrastos Mantadelis, Guy Van den Broeck, Vitor Santos Costa, Gerda Janssens, and Luc De Raedt. 2011. ProbLog. *ALP Newsletter* (2011).

[30] Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 360–384.

[31] Daphne Koller, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*. 740–747.

[32] Alexander Lew, Monica Agrawal, David Sontag, and Vikash Mansinghka. 2021. PClean: Bayesian data cleaning at scale with domain-specific probabilistic programming. In *International conference on artificial intelligence and statistics*. PMLR, 1927–1935.

[33] Alexander K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2019. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

[34] Alexander K Lew, Matin Ghavamizadeh, Martin C Rinard, and Vikash K Mansinghka. 2023. Probabilistic programming with stochastic probabilities. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1708–1732.

[35] Jianlin Li, Leni Ven, Pengyuan Shi, and Yizhou Zhang. 2023. Type-preserving, dependence-aware guide generation for sound, effective amortized probabilistic inference. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1454–1482.

[36] Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. Compiling probabilistic programs for variable elimination with information flow. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1755–1780.

[37] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099* (2014).

[38] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 603–616.

[39] Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

[40] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13*. Springer, 62–79.

[41] Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. 2015. Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices* 50, 6 (2015), 208–217.

[42] Jingwen Pan and Amir Shaikhha. 2023. Compiling Discrete Probabilistic Programs for Vectorized Exact Inference. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 13–24.

[43] Avi Pfeffer. 2001. IBAL: A probabilistic rational programming language. In *IJCAI*. Citeseer, 733–740.

[44] Avi Pfeffer. 2006. IBAL: An Expressive, Functional Probabilistic Modeling Language. (2006).

[45] Avi Pfeffer, Brian Ruttenberg, Amy Sliva, Michael Howard, and Glenn Takata. 2015. Lazy factored inference for functional probabilistic programming. *arXiv preprint arXiv:1509.03564* (2015).

[46] Long Pham, Di Wang, Feras A Saad, and Jan Hoffmann. 2024. Programmable MCMC with Soundly Composed Guide Programs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1051–1080.

[47] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. *ACM SIGPLAN Notices* 44, 2 (2008), 37–48.

[48] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.

[49] Feras A Saad, Martin C Rinard, and Vikash K Mansinghka. 2021. SPPL: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*. 804–819.

[50] Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

[51] Sam Stites, Heiko Zimmermann, Hao Wu, Eli Sennesh, and Jan-Willem van de Meent. 2021. Learning proposals for probabilistic programs with inference combinators. In *Uncertainty in Artificial Intelligence*. PMLR, 1056–1066.

[52] Andreas Stuhlmüller and Noah D Goodman. 2012. A dynamic programming algorithm for inference in recursive probabilistic programs. *arXiv preprint arXiv:1206.3555* (2012).

[53] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages*. 1–12.

[54] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[55] Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. 2011. Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*. AAAI Press/International Joint Conferences on Artificial Intelligence; Menlo …, 2178–2185.

[56] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound probabilistic inference via guide types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 788–803.

[57] David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 770–778.

[58] Yuan Yang and Steven T Piantadosi. 2022. One model for the learning of language. *Proceedings of the National Academy of Sciences* 119, 5 (2022), e2021865119.

[59] Fabian Zaiser, Andrzej Murawski, and Chih-Hao Luke Ong. 2024. Exact Bayesian inference on discrete models via probability generating functions: a probabilistic programming approach. *Advances in Neural Information Processing Systems* 36 (2024).

[60] Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. 2020. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. In *International Conference on Machine Learning*. PMLR, 11534–11545.

Listing A.1 Grammar of λ_{PLUCK}.

Syntactic category		Productions
Types τ	::=	$1 \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\ \ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n$
Contexts Γ	::=	$\varepsilon \mid \Gamma, x : \tau$
Expressions e	::=	$() \mid x \mid \textbf{flip}\ r \mid \ell\ e \mid \textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n x_n \Rightarrow e_n\} \mid (e_1, e_2) \mid \textbf{match}\ e_1\ \textbf{with}\ (x_1, x_2) \Rightarrow e_2 \mid \lambda x.\ e \mid e_1\ e_2 \mid Y(\lambda x_1.\lambda x_2.e)$
Real constants r	$\in$	$[0, 1]$

Sugar		Desugaring
$\mathbb{B}$	=	$\mu\_.\ \text{true}\ 1 + \text{false}\ 1$
$\mathbb{N}$	=	$\mu\alpha.\ \text{zero}\ 1 + \text{succ}\ \alpha$
$\mathbb{L}\ \tau$	=	$\mu\alpha.\ \text{nil}\ 1 + \text{cons}\ (\tau \times \alpha)$
$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	=	$(\lambda x.e_2)\ e_1$
$\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	=	$\textbf{match}\ e\ \textbf{with}\ \text{true}\ \_ \Rightarrow e_1 \mid \text{false}\ \_ \Rightarrow e_2$

Listing A.2 Typing rules for λ_{PLUCK}.

$$\textsc{TUnit}\ \frac{}{\Delta \vdash 1\ \text{type}} \qquad \textsc{TVar}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \text{type}} \qquad \textsc{TProd}\ \frac{\Delta \vdash \tau_1\ \text{type} \qquad \Delta \vdash \tau_2\ \text{type}}{\Delta \vdash \tau_1 \times \tau_2\ \text{type}} \qquad \textsc{TArrow}\ \frac{\vdash \tau_1\ \text{type} \qquad \Delta \vdash \tau_2\ \text{type}}{\Delta \vdash \tau_1 \to \tau_2\ \text{type}}$$

$$\textsc{TRec}\ \frac{\Delta, \alpha \vdash \tau_i\ \text{type}}{\Delta \vdash \mu\alpha.\ \ell_1\tau_1 + \cdots + \ell_n\tau_n\ \text{type}} \qquad \textsc{Var}\ \frac{x \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{Flip}\ \frac{}{\Gamma \vdash \textbf{flip}\ r : \mathbb{B}}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash e : \tau_i[\alpha \mapsto \mu\alpha.\ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n]}{\Gamma \vdash \ell_i\ e : \mu\alpha.\ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n} \qquad \textsc{Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \textsc{MatchProd}\ \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{match}\ e_1\ \textbf{with}\ (x_1, x_2) \Rightarrow e_2 : \tau}$$

$$\textsc{MatchSum}\ \frac{\Gamma \vdash e : \mu\alpha.\ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n \qquad \Gamma, x_i : \tau_i[\alpha \mapsto \mu\alpha.\ell_1\ \tau_1 + \cdots + \ell_n\ \tau_n] \vdash e_i : \tau}{\textbf{match}\ e\ \textbf{with}\ \{\ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n\} : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad Y\ \frac{\Gamma, x_1 : \tau_1 \to \tau_2, x_2 : \tau_1 \vdash e : \tau_2}{\Gamma \vdash Y(\lambda x_1.\lambda x_2.e) : \tau_1 \to \tau_2}$$

A A CORE CALCULUS FOR LAZY, RECURSIVE, PROBABILISTIC PROGRAMS

We now introduce λ_{PLUCK}, a core calculus for lazy, recursive, discrete probabilistic programs. λ_{PLUCK} is a simply typed λ-calculus with iso-recursive types,[13] extended with the construct **flip** r for Bernoulli coin flips with probability r of heads. Its grammar is given in Listing A.1 and its typing rules in Listing A.2. Our syntax and typing rules are largely standard.

λ_{PLUCK} features general recursion, so we can express, for example, a geometric distribution, as

$$geom = Y(\lambda rec.\ \lambda\_.\ \textbf{if}\ (\textbf{flip}\ 0.5)\ \textbf{then}\ \text{zero}()\ \textbf{else}\ \text{succ}(rec())) : 1 \to \mathbb{N}.$$

[13] The case $\mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n$ in our grammar is used to construct both recursive and non-recursive sum types; if non-recursive, α will not occur in the argument types τ_i. We implicitly unroll scrutinees of pattern matches into bare sums, and roll constructed values $\ell_i\ e$ into values of their corresponding recursive types.

Listing A.3 Semantics of λ_{PLUCK} types as functors of quasi-Borel predomains with least element.

$$
\begin{aligned}
[\![1]\!](\Delta) &= 1_\perp \\
[\![\alpha]\!](\Delta) &= \Delta[\alpha] \\
[\![\tau_1 \times \tau_2]\!](\Delta) &= [\![\tau_1]\!](\Delta) \times [\![\tau_2]\!](\Delta) \\
[\![\tau_1 \to \tau_2]\!](\Delta) &= [\![\tau_1]\!](\cdot) \times \Omega \Rightarrow [\![\tau_2]\!](\Delta) \qquad (\tau_1 \text{ must be closed}) \\
[\![\mu\alpha.\ \ell_1\tau_1 + \cdots + \ell_n\tau_n]\!](\Delta) &= \operatorname{colim}\left(\{\perp\} \overset{e_0}{\hookrightarrow} F(\{\perp\}) \overset{e_1}{\hookrightarrow} F^2(\{\perp\}) \overset{e_2}{\hookrightarrow} \cdots\right), \\
&\quad \text{where } F(X_\alpha) = (\sqcup_{i=1}^n [\![\tau_i]\!](\Delta[\alpha \mapsto X_\alpha]))_\perp, \\
&\quad \text{and } e_n(\perp) = \perp, e_n(\mathsf{in}_i\ x_n^i) = \mathsf{in}_i\ ([\![\tau_i]\!](\Delta[\alpha \mapsto e_{n-1}])(x_n^i))
\end{aligned}
$$

Knowledge compilation-based approaches to inference in probabilistic programs typically cannot handle general recursion: during compilation, they need to discover every unique **flip** in a program, and programs like *geom* may flip an unbounded number of coins. This limitation is less severe when we take a lazy approach. This is because, in many programs that *call* recursive functions (e.g., **if** $geom() < 4$ **then** e_1 **else** e_2), a lazy evaluation strategy will only ever force a bounded number of flips. Our lazy knowledge compilation algorithm (§4) can exploit this to successfully compile such programs, despite the presence (from a strict perspective) of unbounded recursion.[14] λ_{PLUCK} does not feature an **observe** construct for conditioning a probabilistic program. As Dash et al. [12] noted, laziness and observation are in tension: in an expression like **let** _ = **observe** e_1 **in** e_2, laziness dictates that the **observe** not be evaluated. (More generally, an **observe** could be nested deeply inside a larger otherwise-unused expression.) Even without **observe**, however, λ_{PLUCK} can still be used to compute posterior probabilities. We define types Maybe $\tau := \mu\_.\text{nothing}\ 1 + \text{just}\ \tau$, and write programs that return nothing() when a condition has been violated, e.g.

let $x = e$ **in if** $predicate(x)$ **then** just x **else** nothing().

Lazy knowledge compilation computes the output distribution of the program; the posterior probability of an outcome x can then be obtained by dividing the probability of just x by 1 minus the probability of nothing().

A.1 Random-Seed Semantics

Space of Random Seeds. Inspired by the approach of Dash et al. [12], we model lazy probabilistic programs as mapping infinite *random seeds* ω to possible outcomes. Formally, our space of random seeds is $\Omega = [0,1]^{\mathbb{N}^*}$: that is, a random seed ω is an infinite collection of real numbers (in $[0,1]$), labeled by (possibly empty) sequences of natural numbers σ. Equivalently, we can think of ω as an infinitely wide, infinitely deep tree (a *rose tree*) whose nodes are labeled with real numbers. The value at the root of the tree is $\omega(\epsilon)$, the value at the third child is $\omega(\epsilon \cdot 3)$, the value at that node's second child is $\omega(\epsilon \cdot 3 \cdot 2)$, and so on. We write $\omega[\sigma]$ for the subtree of ω rooted at path σ (i.e., $\omega[\sigma_1](\sigma_2) = \omega(\sigma_1 +\!\!+ \sigma_2)$), and $\omega_i = \omega[\epsilon \cdot i]$ for the i^{th} child of ω.

Semantics of Types. In Listing A.3, we assign to each λ_{PLUCK} type expression τ a *functor* mapping a typing environment Δ to a space $[\![\tau]\!](\Delta)$ of values. The spaces we consider are *quasi-Borel predomains* [54], an alternative to measurable spaces suitable for modeling higher-order recursive

[14] Of course, there are still programs in λ_{PLUCK} that even under a lazy evaluation order can flip an unbounded number of coins (e.g., *geom() < geom()*). It would be interesting to explore whether there are sensible syntactic restrictions that make such programs inexpressible. A challenge is that even problematic programs can be subexpressions of safe programs—e.g., **let** $x = geom()$ **in** $(x < 4)$ && $(x < geom())$. In this paper, we instead work with an unrestricted core language and prove correctness of exact inference for a restricted class of programs. We also show that for programs outside this class, a variant of our algorithm can still produce unbiased estimates and lower and upper bounds on a program's output distribution.

Listing A.4 Semantics of well-typed λ_{PLUCK} expressions $\Gamma \vdash e : \tau$ as ωQbs maps. Recall that for $\omega \in \Omega$, we write ω_i for $\omega[\varepsilon \cdot i] = \lambda\sigma.\omega(\varepsilon \cdot i \cdot \sigma)$, the i^{th} child of the rose tree ω.

Expression		*Outcome on random seed ω in environment γ*
$[\![\Gamma \vdash x : \tau]\!](\gamma, \omega)$	$=$	$\gamma[x]$
$[\![\Gamma \vdash () : 1]\!](\gamma, \omega)$	$=$	$\star$
$[\![\Gamma \vdash \ell_i\ e : \mu\alpha.\ \ell_1\tau_1 + \cdots + \ell_n\tau_n]\!](\gamma, \omega)$	$=$	$\mathsf{roll}\ (\mathsf{in}_i([\![e]\!](\gamma, \omega)))$
$\left[\!\!\left[\Gamma \vdash \begin{array}{l}\textbf{match}\ e\ \textbf{with}\\ \{\ell_1 x_1 \Rightarrow e_1\\ \mid \ldots\\ \mid \ell_n x_n \Rightarrow e_n\}\end{array} : \tau\right]\!\!\right](\gamma, \omega)$	$=$	$\begin{cases}\bot_\tau & \mathsf{unroll}([\![e]\!](\gamma, \omega_0)) = \bot\\ [\![e_1]\!](\gamma[x_1 \mapsto v], \omega_1) & \mathsf{unroll}([\![e]\!](\gamma, \omega_0)) = \mathsf{in}_1\ v\\ \ldots & \ldots\\ [\![e_n]\!](\gamma[x_n \mapsto v], \omega_n) & \mathsf{unroll}([\![e]\!](\gamma, \omega_0)) = \mathsf{in}_n\ v\end{cases}$
$[\![\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!](\gamma, \omega)$	$=$	$([\![e_1]\!](\gamma, \omega_1), [\![e_2]\!](\gamma, \omega_2))$
$\left[\!\!\left[\Gamma \vdash \begin{array}{l}\textbf{match}\ e_1\ \textbf{with}\\ (x_1, x_2) \Rightarrow e_2\end{array} : \tau\right]\!\!\right](\gamma, \omega)$	$=$	$[\![e_2]\!]\left(\gamma\left[\begin{array}{l}x_1 \mapsto \pi_1([\![e_1]\!](\gamma, \omega_1)),\\ x_2 \mapsto \pi_2([\![e_1]\!](\gamma, \omega_1))\end{array}\right], \omega_2\right)$
$[\![\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2]\!](\gamma, \omega)$	$=$	$\lambda(v, \omega').\ [\![e]\!](\gamma[x \mapsto v], \omega')$
$[\![\Gamma \vdash e_1\ e_2 : \tau_2]\!](\gamma, \omega)$	$=$	$[\![e_1]\!](\gamma, \omega_0)([\![e_2]\!](\gamma, \omega_1), \omega_2)$
$[\![\Gamma \vdash Y(\lambda x_1.\lambda x_2.e) : \tau_1 \to \tau_2]\!](\gamma, \omega)$	$=$	$\mathsf{fix}(\lambda f.\lambda(y, \omega').[\![e]\!](\gamma[x_1 \mapsto f, x_2 \mapsto y], \omega'))$
$[\![\Gamma \vdash \textbf{flip}\ r : \mathbb{B}]\!](\gamma, \omega)$	$=$	$\begin{cases}\mathsf{roll}(\mathsf{in}_1\ \star) & \omega(\varepsilon) \le r\\ \mathsf{roll}(\mathsf{in}_2\ \star) & \textit{otherwise}\end{cases}$

probabilistic programming languages.[15] A quasi-Borel predomain X is a tuple $(|X|, M_X, \leq_X)$, where $|X|$ is the underlying set of values, M_X is a set of *random elements* (playing a role analogous to sigma-algebras in measure theory), and $\leq_X$ is a partial order on $|X|$ used to model recursion via least fixed points. Maps between quasi-Borel predomains must satisfy certain properties, encoding measurability with respect to M_X and Scott-continuity with respect to $\leq_X$. In our treatment, every closed type is interpreted as a *pointed* quasi-Borel predomain, which means that there is a distinguished *least element* $\bot_\tau \in |[\![\tau]\!]|$ such that $\bot_\tau \leq_{[\![\tau]\!]} x$ for all $x \in |[\![\tau]\!]|$.

In our semantics, function types $\tau_1 \to \tau_2$ represent *possibly stochastic* functions, and as such, they denote maps on *pairs* containing both an input value and a random seed $\omega \in \Omega$. The semantics of recursive types are more intricate, involving an inverse limit construction similar to that of Vákár et al. [54]. We give further details in Appendix C, but remark here on a couple salient differences from that work. First, to simplify our semantic development, we do not allow free type variables α on the left-hand side of the function arrow. As a result, the functors whose fixed points we compute are always covariant, as opposed to the mixed-variance functors in Vákár et al. [54]. Second, to model laziness, our treatment more closely matches the semantics of Haskell algebraic types, whereas Vákár et al. [54] stay closer to ML's. As a concrete example, consider interpreting the type $\mathbb{L}\ 1$ of lists of unit. In our semantics, we construct the limit of a sequence of spaces $F^i(\{\bot\})$, where $F(X) = (1 + 1_\bot \times X)_\bot$. By contrast, Vákár et al. [54] consider the sequence $F^i(\emptyset)$, with the functor $F(X) = 1 + 1 \times X$. As a result, their interpretation of $\mathbb{L}\ 1$ contains only finite lists, whereas ours contains finite lists, partial lists (whose tail after some point is $\bot$), and infinite lists.

Semantics of Terms. Listing A.4 gives the semantics of λ_{PLUCK} terms. We interpret terms-in-context $\Gamma \vdash e : \tau$ as maps from an environment $\gamma \in [\![\Gamma]\!] := \prod_{x:\tau \in \Gamma} [\![\tau]\!]$ and a random seed $\omega \in \Omega$ to

[15] One may reasonably wonder why we need such high-powered machinery to model a discrete PPL—can't we just use sets? But laziness introduces an interesting wrinkle. The program $unif = Y(\lambda rec.\ \lambda\_.\ \mathsf{cons}(\textbf{flip}(0.5), rec())) : 1 \to \mathbb{L}\ \mathbb{B}$ does not diverge in our semantics, and instead represents the uniform distribution on the Cantor space $2^{\mathbb{N}}$—a distribution with uncountable support that requires a measure-theoretic treatment to handle formally.

outcomes $[\![\tau]\!]$. There are two key points of interest: the way the random seed is consumed by a program, and the way laziness is reflected.

On randomness: given a random seed ω, the program **flip** r looks directly at the value $\omega(\varepsilon)$ at the root of the rose tree to decide whether to return true (i.e., $\mathsf{roll}(\mathsf{in}_1\ \star)$) or false (i.e., $\mathsf{roll}(\mathsf{in}_2\ \star)$). Compound expressions, such as (e_1, e_2), use different children of ω as the random seeds when evaluating different subexpressions. In function application $e_1\ e_2$, three distinct children of ω are used: ω_0 to resolve e_1 (which may randomly choose which function to apply); ω_1 to resolve e_2 (which may randomly sample a function argument), and ω_2 for the randomness needed by the function call itself. In this way, each coin flip required when evaluating the program is assigned a unique path σ through the rose tree ω. This path σ is somewhat analogous to a callstack, growing as our "interpreter" makes recursive calls. This analogy between unique names of random choices in PPLs and stack addresses has been previously noted by Wingate et al. [57].

On laziness: our semantics is lazy in that non-termination (modeled with $\bot_\tau$) of a sub-expression does not imply non-termination of the overall program. Only constructs which force sub-expressions to a value propagate non-termination: **match** on sum types forces the scrutinee, and function application $e_1\ e_2$ forces the function e_1.

A.2 Distributional Semantics

We now fix a probability distribution μ_Ω on our space of random seeds Ω: the uniform distribution on rose trees [12]. For $\omega \sim \mu_\Omega$, each $\omega(\sigma)$ is distributed according to $\mathsf{Uniform}(0,1)$, independent of all other nodes in ω. Given a program $\Gamma \vdash e : \tau$, we define $[\![e]\!]_P(\gamma) := [\![e]\!](\gamma, -)_*\mu_\Omega$, so that $[\![e]\!]_P$ is a (quasi-Borel) probability kernel from $[\![\Gamma]\!]$ to $[\![\tau]\!]$. Unlike in some other approaches [54, e.g.], we do not use *sub*-probability kernels to model possibly non-terminating programs. Our programs all denote probability kernels, which may or may not place mass on $\bot$, the outcome representing non-termination. Note that programs may place positive mass on partial *values* (e.g., $(3, \bot)$) even if they place no mass on $\bot$. Operationally, such programs can be understood as halting almost surely under a lazy evaluation strategy, but computing a value only up to *weak-head normal form*.

B LAZY KNOWLEDGE COMPILATION

Lazy knowledge compilation is an algorithm for efficiently compiling a probabilistic program to a concrete, symbolic representation of the distribution it denotes. We first describe the symbolic representation (§B.1), before presenting the lazy knowledge compilation algorithm (§B.2).

B.1 Lazy Symbolic Representations for Distributions

When a lazy interpreter evaluates an expression $e : \tau$, it does not fully resolve the value in $[\![\tau]\!]$ that e denotes. Rather, it computes only up to *weak-head normal form*: the top-level constructor is resolved, but the remainder of the value is left unevaluated. A key idea in our approach is that the symbolic representations we manipulate can similarly encode *distributions* only up to weak-head normal form. That is, we can track the distribution of the top-level constructor, but leave the distributions of the arguments unevaluated.

Listing B.1 gives the syntax of our symbolic representations v, and Listing B.2 gives their semantics. As in §A, we think of distributions on a space $[\![\tau]\!]$ as maps $\Omega \to [\![\tau]\!]$. Generally speaking, our symbolic representations work by partitioning Ω into disjoint regions carved out by symbolic predicates $\phi : \Omega \to 2$, and in each region, specifying a concrete value up to weak-head normal form. As a simple example, consider a distribution over $[\![1]\!] = 1_\bot$. The symbolic representation $\star_\phi$ precisely captures which random seeds ω lead to the outcome $\star$ (as opposed to $\bot$) in the form of the symbolic predicate ϕ. We write $\star_\phi :_V 1$ to indicate that $\star_\phi$ is a well-formed representation of a distribution over 1, and $[\![\star_\phi]\!]_V$ for the distribution it represents.

Listing B.1 Syntax of guarded weak-head normal forms.

$$
\begin{array}{rcl}
\Omega\text{-predicates } \phi & \in & \Omega \Rightarrow 2 \\
\text{Rose tree paths } \sigma & ::= & \varepsilon \mid \sigma \cdot n \text{ (for } n \in \mathbb{N}) \\
\text{Environments } \gamma & ::= & \varepsilon \mid \gamma[x \mapsto t] \\
\text{Finite guarded sets } S(\text{pattern}) & ::= & \{y_{\phi_y}\}_{y \in Y} \text{ (for } Y \text{ a finite set of terms with syntax } \textit{pattern}) \\
\text{Thunks } t & ::= & S(\langle e, \gamma, \sigma \rangle) \\
\text{Weak-head normal forms } v & ::= & \star_\phi \mid (t_1, t_2) \mid S(\ell\ t) \mid S(\langle x.e, \gamma \rangle)
\end{array}
$$

$$
\textsc{Guarded-Thunk}\quad \frac{\forall\, \langle e_y, \gamma_y, \sigma_y \rangle \in Y.\ \exists\, \Gamma_y.\ \left(\Gamma_y \vdash e_y : \tau \wedge \gamma_y :_E \Gamma_y\right) \qquad (\phi_y)_{y \in Y} \text{ mutually exclusive}}{\{y_{\phi_y}\}_{y \in Y} :_T \tau}
$$

$$
\textsc{VClosure}\quad \frac{\forall\, \langle x_y.e_y, \gamma_y \rangle \in Y.\ \exists\, \Gamma_y.\ \left(\Gamma_y, x_y : \tau_1 \vdash e_y : \tau_2 \wedge \gamma_y :_E \Gamma_y\right) \qquad (\phi_y)_{y \in Y} \text{ mutually exclusive}}{\{y_{\phi_y}\}_{y \in Y} :_V \tau_1 \to \tau_2}
$$

$$
\textsc{Env-}\varepsilon\quad \frac{}{\varepsilon :_E \varepsilon} \qquad \textsc{Env-}\Gamma\quad \frac{\gamma :_E \Gamma \qquad t :_T \tau}{\gamma[x \mapsto t] :_E (\Gamma, x : \tau)} \qquad \textsc{VUnit}\quad \frac{}{\star_\phi :_V 1}
$$

$$
\textsc{VSum}\quad \frac{\forall\, (\ell_{i_y} t_y) \in Y.\ t_y :_T \tau_{i_y}[\alpha \mapsto \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n] \qquad (i_y)_{y \in Y} \text{ distinct} \qquad (\phi_y)_{y \in Y} \text{ mutually exclusive}}{\{y_{\phi_y}\}_{y \in Y} :_V \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n}
$$

$$
\textsc{VProd}\quad \frac{t_1 :_T \tau_1 \qquad t_2 :_T \tau_2}{(t_1, t_2) :_V \tau_1 \times \tau_2}
$$

Thunks and Environments. Distributions over sum and product types, for which the weak-head normal form is not the full value, require some way of representing unevaluated parts of a distribution. A *singleton thunk* $\langle e, \gamma, \sigma \rangle$ represents an ω-dependent value whose symbolic representation we have not yet computed. It combines an expression $\Gamma \vdash e : \tau$ (the program code we will need to process if we ever force this value) with (1) an *environment* γ over the free variables in Γ (assigning each variable name to its *own* thunk), and (2) a rose-tree path σ indicating what part of the randomness in Ω this value depends on. Singleton thunks allow us to delay the evaluation of the expression e when computing a distribution's weak-head normal form representation. For example, consider the expression **if** (**flip** 0.5) **then** $\ell_1\ e_1$ **else** $\ell_2\ e_2$. Without evaluating e_1 or e_2, we can say that this (closed) program yields $\ell_1\ \langle e_1, \varepsilon, \varepsilon \cdot 1 \rangle$ when $\omega(\varepsilon \cdot 0) \leq 0.5$, and $\ell_2\ \langle e_2, \varepsilon, \varepsilon \cdot 2 \rangle$ otherwise. But singleton thunks are not sufficient to represent all distributions over weak-head normal forms. For example, consider the very similar expression **if** (**flip** 0.5) **then** $\ell_1 e_1$ **else** $\ell_1 e_2$. With probability 1, its constructor is ℓ_1, but for some ω the value inside is given by e_1, whereas for others it is given by e_2. We merge these possibilities into *full thunks*, finite sets of singleton thunks $y = \langle e_y, \gamma_y, \sigma_y \rangle$ all of the same type, each guarded by a predicate ϕ_y. We write $\{\langle e_y, \gamma_y, \sigma_y \rangle_{\phi_y}\}_{y \in Y}$ for the full thunk with possibilities $y \in Y$. In this example, the program *always* yields $\ell_1\ t$, where $t = \{\langle e_1, \varepsilon, \varepsilon \cdot 1 \rangle_{\omega(\varepsilon \cdot 0) \leq 0.5}, \langle e_2, \varepsilon, \varepsilon \cdot 2 \rangle_{\omega(\varepsilon \cdot 0) > 0.5}\}$.

The semantics of thunks and environments are given by a mutual recursion. For environments, $[\![\gamma]\!]_E(\omega)$ maps each variable x to $[\![\gamma(x)]\!]_T(\omega)$—the meaning of the (full) thunk they store for variable x. For full thunks, $[\![\{\langle e_y, \gamma_y, \sigma_y \rangle_{\phi_y}\}_{y \in Y}]\!]_T(\omega)$ branches on which predicate ϕ_y holds of ω,

Listing B.2 Semantics of guarded weak-head normal forms.

$$[\![\varepsilon]\!]_E(\omega) = \varepsilon \qquad [\![\gamma[x \mapsto t]]\!]_E(\omega) = [\![\gamma]\!]_E(\omega)[x \mapsto [\![t]\!]_T(\omega)]$$

$$[\![\{\langle e_y, \gamma_y, \sigma_y\rangle_{\phi_y}\}_{y\in Y}]\!]_T(\omega) = \begin{cases} [\![e_y]\!]([\![\gamma_y]\!]_E(\omega), \omega[\sigma_y]) & \text{if } \phi_y(\omega) \text{ for some } y \in Y \\ \bot_\tau & \text{otherwise} \end{cases}$$

$$[\![\star_\phi]\!]_V(\omega) = \begin{cases} \star & \text{if } \phi(\omega) \\ \bot & \text{otherwise} \end{cases} \qquad [\![(t_1, t_2)]\!]_V(\omega) = ([\![t_1]\!]_T(\omega), [\![t_2]\!]_T(\omega))$$

$$[\![\{(\ell_{i_y}\ t_y)_{\phi_y}\}_{y\in Y}]\!]_V(\omega) = \begin{cases} \mathsf{roll}(\mathsf{in}_{i_y}\ [\![t_y]\!]_T(\omega)) & \text{if } \phi_y(\omega) \text{ for some } y \in Y \\ \mathsf{roll}(\bot) & \text{otherwise} \end{cases}$$

$$[\![\{\langle x_y.e_y, \gamma_y\rangle_{\phi_y}\}_{y\in Y}]\!]_V(\omega) = \lambda(v, \omega').\begin{cases} [\![e_y]\!]([\![\gamma_y]\!]_E(\omega)[x_y \mapsto v], \omega') & \text{if } \phi_y(\omega) \text{ for some } y \in Y \\ \bot_{\tau_2} & \text{otherwise} \end{cases}$$

then evaluates $[\![e_y]\!]$ in the environment $[\![\gamma_y]\!]_E$ on the sub-seed of ω corresponding to path σ_y. The base case of this mutual recursion is the empty environment.

Products and Sums. The weak-head normal form of a value of product type resolves that it *is* a pair, but not the values of its elements. As such, we represent a distribution over a product type as a pair of thunks (t_1, t_2). The representation of a distribution on (recursive) sum types is a set of possibilities $y = \ell_{i_y}\ t_y$, each guarded by a predicate ϕ_y. We require that the predicates are mutually exclusive, and that the i_y are all distinct. The predicates tell us when each constructor ℓ_{i_y} is active, and the thunks t_y are the (delayed) constructor arguments, of type $\tau_{i_y}[\alpha \mapsto \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n]$. When no predicate ϕ_y holds of a random seed ω, the outcome for ω is implicitly $\mathsf{roll}(\bot)$.

Closures. To represent ω-dependent values of function type $\tau_1 \to \tau_2$, we use finite sets Y of *singleton closures* $y = \langle x_y.e_y, \gamma_y\rangle$, each guarded by a predicate ϕ_y. A singleton closure pairs a function body e_y with an environment $\gamma_y :_E \Gamma_y$ such that $\Gamma_y, x_y : \tau_1 \vdash e_y : \tau_2$. Unlike a singleton thunk, a singleton closure does not need to store a rose tree path σ. The reason for this difference is that whereas a thunk represents a computation that uses a fixed portion of the randomness in ω, a closure represents a stochastic function that will use a new part of ω every time it is called.

Encoding Predicates as Binary Decision Diagrams. We work with a restricted class of predicates $\phi : \Omega \to 2$ that can be represented concretely as *binary decision diagrams*. In particular, our predicates arise as finite Boolean formulae over *atomic predicates* $\omega(\sigma) \le r$, where every σ always appears with the same r. To represent such a predicate as a binary decision diagram, we instantiate a fresh Boolean variable $x_{\sigma,r}$ for each atomic predicate $\omega(\sigma) \le r$. Then the composite predicate ϕ is represented as a directed acyclic graph, where each node is either a *decision node* labeled with the name of a variable, or a *terminal node* labeled with T or F. Decision nodes have two outgoing edges, a *high* edge and a *low* edge. To test whether ϕ holds of a particular ω, we can walk the graph from the root node to a terminal node and check if we landed at T; at each decision node $x_{\sigma,r}$, we follow the high edge if $\omega(\sigma) \le r$ and the low edge if $\omega(\sigma) > r$.

We work with *reduced*, *ordered* binary decision diagrams, which means that: (1) isomorphic subgraphs are merged; (2) decision nodes whose high and low edges point to the same child node are removed; and (3) along every path from the root, variables are encountered in the same, pre-fixed

order.[16] This ensures that a given predicate ϕ has a unique *canonical* representation as a binary decision diagram. We use an off-the-shelf library for constructing and manipulating binary decision diagrams, which supports implementations of conjunctions, disjunctions, implications, and other Boolean operations.

In the worst case, the binary decision diagrams needed to represent the semantics of a probabilistic program may have size exponential in the number of variables. However, binary decision diagrams can exploit many forms of independence and local structure, making them the representation of choice for multiple probabilistic programming systems [6, 19, 26, 29].

B.2 Lazy Knowledge Compilation

We now turn to the algorithm for *computing* symbolic representations of the distribution denoted by a program. Given a closed program $\vdash e : \tau$, our goal is to automatically compute a representation $v :_V \tau$ such that $[\![e]\!] = [\![v]\!]_V$.[17]

Compiling Programs. The lazy knowledge compilation algorithm is given in Listing B.3. It is designed to operate on both closed and open terms, and takes four inputs that evolve as the algorithm recursively invokes itself on subterms of the input program. They are:

- The expression $\Gamma \vdash e : \tau$ to compile.
- An environment $\gamma :_E \Gamma$ containing thunks for any free variables in e.
- A *path condition* $\phi : \Omega \to 2$. When invoking lazy knowledge compilation at the top level on a closed term, the path condition is T, but in recursive calls to compile subterms, the path condition evolves to encode the conditions under which the subterm e must be evaluated.
- A *rose-tree path* $\sigma \in \mathbb{N}^*$. When invoking lazy knowledge compilation at the top level, the rose-tree path is ε. In recursive calls, the rose-tree path indicates which portion of the random seed ω the sub-expression e uses.

If the algorithm terminates, it returns:

- A representation $v :_V \tau$ of the distribution of e in environment $[\![\gamma]\!]_E$ on random seed $\omega[\sigma]$.
- A *validity condition* ϕ_u encoding the conditions under which v is a correct representation of e's semantics. The input path condition ϕ always implies the returned validity condition ϕ_u, but the validity condition may be significantly more general (i.e., impose fewer constraints on ω). This can be useful in case the need arises to compile the same expression e for a different path condition ϕ_{new} (e.g., if a thunk of expression e is forced in multiple different control-flow paths): as long as ϕ_{new} implies ϕ_u, we can reuse the same result v.

In Listing B.3, the judgment $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ means that, when invoked on expression e, environment γ, path condition ϕ, and rose-tree path σ, lazy knowledge compilation succeeds and returns the representation v and the validity condition ϕ_u. The rules can be read bottom-up-and-down-again: given the inputs to the algorithm (left side of the conclusion), it makes various recursive calls (premises), and uses their results to compute its results (right side of the conclusion). As the algorithm runs, all predicates are represented jointly as a *multi-rooted* binary decision

[16] This does not mean every variable appears along every path, merely that if two variables do appear on the same path, they always appear in the same order. See next section for further discussion on the variable order we use.

[17] We note that for general programs e this task is uncomputable, as it requires generating a finite Boolean formula describing the random seeds on which e halts. An intuitive condition for when our algorithm succeeds is that a lazy sampler for e must generate a bounded number of fresh coin flips. Note that many recursive programs that would *not* satisfy this criterion under a strict evaluation order nevertheless *do* satisfy it under a lazy one. For example, even though a probabilistic context-free grammar may have infinitely many execution paths with no bound on the total length, the program that generates from a PCFG and then checks whether the returned string is equal to a finite observed string *does* examine only a bounded number of coin flips, under a lazy evaluation order.

Listing B.3 Big-step operational semantics of lazy knowledge compilation. If $\gamma :_E \Gamma$ and $\Gamma \vdash e : \tau$, then the judgment $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ means that lazy knowledge compilation, run on expression e, environment γ, path condition $\phi : \Omega \to 2$, and rose-tree path $\sigma \in \mathbb{N}^*$, yields value $v :_V \tau$, and that the result v is valid for all ω in $\phi_u \supseteq \phi$.

$$\textsc{False}\ \frac{\phi \implies \mathsf{F}}{\gamma, \phi, \sigma \vdash e \rightsquigarrow \varepsilon \dashv \mathsf{F}} \qquad \textsc{Flip}\ \frac{}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow \{(\mathsf{true}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\omega(\sigma) \le r}, (\mathsf{false}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\neg(\omega(\sigma) \le r)}\} \dashv \mathsf{T}}$$

$$\textsc{Unit}\ \frac{}{\gamma, \phi, \sigma \vdash () \rightsquigarrow \star_{\mathsf{T}} \dashv \mathsf{T}} \qquad \textsc{Constructor}\ \frac{}{\gamma, \phi, \sigma \vdash \ell_i\ e \rightsquigarrow (\ell_i\ \langle e, \gamma, \sigma \rangle_{\mathsf{T}})_{\mathsf{T}} \dashv \mathsf{T}}$$

$$\textsc{Pair}\ \frac{}{\gamma, \phi, \sigma \vdash (e_1, e_2) \rightsquigarrow (\langle e_1, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}, \langle e_2, \gamma, \sigma \cdot 2 \rangle_{\mathsf{T}}) \dashv \mathsf{T}} \qquad \textsc{Abs}\ \frac{}{\gamma, \phi, \sigma \vdash \lambda x.\ e \rightsquigarrow \langle x.e, \gamma \rangle_{\mathsf{T}} \dashv \mathsf{T}}$$

$$\mathrm{Y}\ \frac{}{\gamma, \phi, \sigma \vdash Y(\lambda x_1.\lambda x_2.e) \rightsquigarrow \langle x_2.e, \gamma[x_1 \mapsto \langle Y(\lambda x_1.\lambda x_2.e), \gamma, \sigma \rangle_{\mathsf{T}}] \rangle_{\mathsf{T}} \dashv \mathsf{T}} \qquad \textsc{CacheHit}\ \frac{\gamma, \phi^*, \sigma \vdash e \rightsquigarrow v \dashv \phi_u \qquad \phi \implies \phi_u}{\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u}$$

$$\textsc{Var}\ \frac{\gamma(x) = \{\langle e_i, \gamma_i, \sigma_i \rangle_{\phi_i}\}_{i \in I} \qquad \gamma_i, \phi \wedge \phi_i, \sigma_i \vdash e_i \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \mathsf{T}) = (v, \phi_u)}{\gamma, \phi, \sigma \vdash x \rightsquigarrow v \dashv \phi_u}$$

$$\textsc{App}\ \frac{\begin{array}{c}\gamma, \phi, \sigma \cdot 0 \vdash e_1 \rightsquigarrow \{\langle x_i.e_i^f, \gamma_i \rangle_{\phi_i}\}_{i \in I} \dashv \phi_u \\ \gamma_i[x_i \mapsto \langle e_2, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}], \phi \wedge \phi_i, \sigma \cdot 2 \vdash e_i^f \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \phi_u) = (v, \phi_u^*)\end{array}}{\gamma, \phi, \sigma \vdash e_1\ e_2 \rightsquigarrow v \dashv \phi_u^*}$$

$$\textsc{MatchProd}\ \frac{\gamma, \phi, \sigma \cdot 1 \vdash e_1 \rightsquigarrow (v_1, v_2) \dashv \phi_u \qquad \gamma[x_1 \mapsto v_1, x_2 \mapsto v_2], \phi, \sigma \cdot 2 \vdash e_2 \rightsquigarrow v \dashv \phi_u^*}{\gamma, \phi, \sigma \vdash \mathbf{match}\ e_1\ \mathbf{with}\ (x_1, x_2) \Rightarrow e_2 \rightsquigarrow v \dashv \phi_u \wedge \phi_u^*}$$

$$\textsc{MatchSum}\ \frac{\begin{array}{c}\gamma, \phi, \sigma \cdot 0 \vdash e \rightsquigarrow \{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y \in Y} \dashv \phi_u \\ \gamma[x_y \mapsto v_y], \phi \wedge \phi_y, \sigma \cdot i_y \vdash e_{i_y} \rightsquigarrow v_y^* \dashv \phi_y^u \qquad \mathsf{join}(\{(v_y^*, \phi_y, \phi_y^u)\}_{y \in Y}, \phi_u) = (v, \phi_u^*)\end{array}}{\gamma, \phi, \sigma \vdash \mathbf{match}\ e\ \mathbf{with}\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n \rightsquigarrow v \dashv \phi_u^*}$$

diagram—that is, isomorphic subgraphs are shared in memory across all predicates constructed by the algorithm, and they all use the same Boolean variables to represent atomic predicates $\omega[\sigma] \le r$. We comment briefly on the logic of each case.

- False: If the path condition is false (F), then this expression is unreachable, and laziness dictates that we not compile it.[18] We return the empty representation ε of the appropriate type: $\star_{\mathsf{F}}$ for $e : 1$, $(\emptyset, \emptyset)$ when $e : \tau_1 \times \tau_2$, and $\emptyset$ for sum and function types.
- Flip: A coin flip may evaluate to true() or false(). The predicates guarding each option are the atomic predicate $\omega[\sigma] \le r$ and its negation.
- Unit, Constructor, Pair, Abs, Y: These rules all apply to expressions that, under lazy evaluation, immediately produce a value. Their weak-head normal forms are deterministic (i.e., not a function

[18] Even if other rules also match the input expression, our algorithm always applies rule False when it can, under the principle that any unnecessary compilation should be avoided.

Listing B.4 Definition of $\mathsf{join}(\{(v_i, \phi_i, \phi_{i,u})\}_{i\in I}, \phi_u)$ helper.

$$\mathsf{join}(\{(v_i, \phi_i, \phi_{i,u})\}_{i\in I}, \phi_u) = \left(\bigoplus_{i\in I}(v_i, \phi_i), \phi_u \wedge \left(\bigwedge_{i\in I}(\phi_i \implies \phi_{i,u})\right)\right)$$

$$(\star_{\phi^1_\star}, \phi_1) \oplus (\star_{\phi^2_\star}, \phi_2) = \star_{(\phi_1\wedge\phi^1_\star)\vee(\phi_2\wedge\phi^2_\star)} \qquad (t_1, t_2) \oplus (t_3, t_4) = (t_1 \oplus t_3, t_2 \oplus t_4)$$

$$(\{y_{\phi^1_y}\}_{y\in Y_1}, \phi_1) \oplus (\{y_{\phi^2_y}\}_{y\in Y_2}, \phi_2) = \emptyset \uplus (y_{1,1}, \phi_1 \wedge \phi^1_{y_{1,1}}) \uplus \cdots \uplus (y_{2,|Y_2|}, \phi_2 \wedge \phi^2_{y_{2,|Y_2|}})$$

$$\{y_{\phi_y}\}_{y\in Y} \uplus (y', \phi) = \begin{cases} \{y_{\phi_y}\}_{y\in Y} \cup \{y'_\phi\} & \text{if } \forall y \in Y, y' \not\approx y \\ \{y_{\phi_y}\}_{y'\not\approx y\in Y} \cup \{\mathsf{merge}(y, y')_{\phi_y\vee\phi}\} & \text{if } y \approx y' \text{ for some } y \in Y \end{cases}$$

$$y_1 \approx y_2 \text{ iff } y_1 = y_2 \vee \begin{pmatrix} y_1 = \ell_i\ t_1 \\ \wedge \\ y_2 = \ell_i\ t_2 \end{pmatrix} \qquad \mathsf{merge}(\ell_i\ t_1, \ell_i\ t_2) = \ell_i\ (t_1 \oplus t_2) \qquad \mathsf{merge}(y, y) = y$$

of ω), so all guard predicates are just T. Note that when creating thunks for the two components in rule Pair, we extend their rose-tree paths, to reflect that our semantics splits the random seed ω into ω_1 and ω_2 for use by each pair component.

- MatchProd: Matching on an expression e_1 of product type induces a recursive call, yielding a representation (t_1, t_2) of the pair's value. We place these thunks into the environment and then compile the body e_2. Rose-tree paths are extended in both recursive calls to reflect the splitting of the random seed ω. Our final returned value is valid only if the validity conditions computed by both recursive calls hold, so our validity condition is their intersection.
- Var, App, MatchSum: Because thunks, closures, and sums are all represented as sets $\{y_{\phi_y}\}_{y\in Y}$ of possible values, guarded by mutually exclusive predicates ϕ_y, the elimination forms for these types all involve a similar pattern, where we invoke a recursive call for each possibility $y \in Y$, and then *join* the resulting values v_y, using the helper function join defined in Listing B.4. The specification satisfied by join is captured by the following lemma.

Lemma B.1 (Restatement of Lemma 4.3). *Let $f : \Omega \to [\![\tau]\!]$ and suppose that for some mutually exclusive collection of predicates ϕ_i on Ω, there exist $f_i : \Omega \to [\![\tau]\!]$ such that for all ω satisfying an additional predicate ϕ_u, we have*

$$f(\omega) = \begin{cases} f_i(\omega) & \text{if } \phi_i(\omega) \\ \bot_\tau & \text{if } \forall i, \neg\phi_i(\omega) \end{cases}.$$

Further suppose that $(v_i, \phi^u_i)_{i\in I}$ are such that whenever $\phi^u_i(\omega)$ holds, $f_i(\omega) = [\![v_i]\!]_V(\omega)$. Then $\mathsf{join}(\{(v_i, \phi_i, \phi^u_i)\}_{i\in I}, \phi_u)$ *yields (v, ϕ^*_u) such that:*

(1) $(\phi_u \wedge \phi^i_u \wedge \phi_i)(\omega) \implies \phi^*_u(\omega)$ for all $i \in I$;
(2) $(\phi_u \wedge \forall i \in I, \neg\phi_i)(\omega) \implies \phi^*_u(\omega)$; and
(3) for all ω such that $\phi^*_u(\omega)$ holds, $[\![v]\!]_V(\omega) = f(\omega)$.

Proof. For the first two conclusions, first note that $\phi^*_u = \phi_u \wedge (\bigwedge_{i\in I}(\phi_i \implies \phi^u_i))$. For (1), if we assume $\phi_u \wedge \phi^u_i \wedge \phi_i$, then clearly ϕ_u holds, and for all $j \neq i$, the implication $\phi_j \implies \phi^u_j$ holds trivially because the premise is false (the ϕ_i are mutually exclusive). The implication $\phi_i \implies \phi^u_i$ holds because we have assumed ϕ^u_i. For (2), if we assume $\phi_u \wedge \forall i \in I, \neg\phi_i$, then ϕ_u holds by assumption and all the implications $\phi_i \implies \phi^u_i$ hold trivially since their premises are false.

The third conclusion (3) follows from the fact that

$$[\![(v_1, \phi_1) \oplus (v_2, \phi_2)]\!]_V(\omega) = \begin{cases} [\![v_1]\!]_V(\omega) & \text{if } \phi_1(\omega) \\ [\![v_2]\!]_V(\omega) & \text{if } \phi_2(\omega) \\ \bot & \text{otherwise} \end{cases},$$

which can be verified by unfolding the definitions of $[\![\cdot]\!]_V$ (in Listing 4.2) and of $\oplus$ (in Listing B.4). □

Guarded Caching. Without *caching*, lazy evaluation strategies can easily duplicate computations, blowing up the complexity of a program. The same need for caching arises in our setting: we do not want to recompile the expression in a thunk anew every time the corresponding variable is used. In standard lazy evaluation, the typical solution is to equip thunks with local storage, and after they are evaluated once, to cache their computed values for reuse if the thunk is evaluated again. But a wrinkle in our setting prevents us from directly employing this solution. Lazy knowledge compilation uses the path condition ϕ to avoid compiling a *complete* representation of the distribution of an expression. For example, if the path condition includes that a randomly generated number n is less than 5, the compilation of an expression that uses n needs only consider execution paths reachable when $n < 5$. However, if the same expression is later evaluated under a different path condition, the earlier-compiled result may no longer be valid. Therefore, our thunks store *guarded* cached results: when we first compile the expression inside a thunk, we cache the result alongside the *validity condition* ϕ_u computed during compilation. Future evaluations of the thunk check whether the path condition ϕ implies the cached validity condition; if so, the previously compiled value can be reused, and if not, the compiler is invoked anew.

This caching is reflected in the rule CacheHit, which allows us to establish $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$ simply by checking that ϕ *implies* the validity condition ϕ_u returned by a previous invocation of our algorithm on e, regardless of the specific path condition ϕ^* active during the original computation.

Variable Ordering. It is well known that the efficiency of BDDs as a representation depends on the ordering used for the variables. Existing PPLs with knowledge compilation use the simple heuristic that variables in BDDs appear in the same order they are encountered by the compiler [26]. Our compiler processes **flip**s in a different order to that of standard knowledge compilation, and so this heuristic leads to a different variable order in our setting. However, we can recover the standard heuristic by ordering variables $\omega(\sigma) \leq r$ lexicographically on the rose-tree path σ.

Handling Nontermination. One benefit of laziness is that many recursive programs with infinitely many executions under a strict evaluation order have only finitely many under a lazy evaluation order, making exact inference possible. However, there are of course still some recursive programs for which it is not possible to compile a finite BDD representation, lazily or otherwise. To ensure termination for all almost-surely terminating source programs, we can take either or both of the following two approaches, which cause our algorithm to approximate a program's semantics either with *bounds* on its distribution, or *unbiased estimates* of it:

- **Bounding**: If we add the rule Bound from Listing B.5, our algorithm can decide at any point to return the empty representation of the appropriate type. In practice, our algorithm has configurable time and depth limits after which further recursive calls to the compiler return the empty representation. Note that this corresponds to including the rule Bound deep in the derivation tree for a top-level judgment; the algorithm may still (and often does) return non-trivial results for the top-level inference query. The guarantee for these results v is that $[\![v]\!]_V \leq [\![e]\!]_P$. Here, $\leq$ is an order on quasi-Borel probability measures over an $\omega Qbs\ X$. Formally, $\mu \leq \nu$ if for all ωQbs maps $f : X \to [0, \infty]$ (where $[0, \infty]$ is equipped with the linear order), $\mu(f) \leq \nu(f)$.

Listing B.5 Additional rules for lazy knowledge compilation, which can be used to obtain approximate results for programs with an infinite number of lazy execution paths.

$$\textsc{Bound}\quad \frac{}{\gamma, \phi, \sigma \vdash e \rightsquigarrow \varepsilon \dashv \mathsf{T}} \qquad \textsc{Flip}\quad \frac{\neg\text{sampled}(\sigma)}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow \{(\text{in}_{\mathbf{true}}\ \langle (), \varepsilon, \varepsilon\rangle_{\mathsf{T}})_{\omega(\sigma)\le r}, (\text{in}_{\mathbf{false}}\ \langle (), \varepsilon, \varepsilon\rangle_{\mathsf{T}})_{\neg(\omega(\sigma)\le r)}\} \dashv \mathsf{T}}$$

$$\textsc{SampleTrue}\quad \frac{\text{sampled}(\sigma) \qquad \omega_{\text{compile}}(\sigma) \le r}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow (\text{in}_{\mathbf{true}}\ \langle (), \varepsilon, \varepsilon\rangle_{\mathsf{T}})_{\mathsf{T}} \dashv \mathsf{T}} \qquad \textsc{SampleFalse}\quad \frac{\text{sampled}(\sigma) \qquad \omega_{\text{compile}}(\sigma) > r}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow (\text{in}_{\mathbf{false}}\ \langle (), \varepsilon, \varepsilon\rangle_{\mathsf{T}})_{\mathsf{T}} \dashv \mathsf{T}}$$

When we are dealing with probability measures over, e.g., the unit type in our language, this is equivalent to the condition that μ assigns less or equal mass to $\star$ and more or equal mass to $\bot$ than ν. In practical terms, when a time or depth limit is hit, users still receive a symbolic representation of a distribution that can be used to compute lower bounds on probabilities of particular outcomes. It can also be used to compute upper bounds, by subtracting from 1 the total probability of all other (non-$\bot$) outcomes.

- **Sampling**: The second strategy is to allow the compiler itself to be stochastic. Suppose we fix a *compile-time* random seed $\omega_{compile} \in \Omega$ as well as an infinite map $\text{sampled} : \mathbb{N}^* \to 2$ from rose-tree paths to Booleans indicating whether the flip at that path will be sampled or enumerated. Assume furthermore that at most finitely many rose-tree paths have $\text{sampled}(\sigma)$ set to false. Then we can replace the standard Flip rule with the three rules in Listing B.5, which encode that when $\text{sampled}(\sigma)$ is true, **flip**s at rose-tree path σ are sampled by the compiler rather than enumerated. In practice, the choice of whether to sample can, like the choice of whether to bound, be based on a time or depth limit. In expectation over $\omega_{compile}$, sampling produces *unbiased estimates* of the distribution of a program. The unbiased estimates obtained from sampling are interestingly related to the bounds obtained by bounding when the same decision rule (e.g. depth limit) is used: sampling randomly produces one of two estimates—either the lower or upper bound computed by the bounded version of the algorithm—in a convex combination that ensures unbiasedness.

  Note that if both rulesets are added, we in general will get unbiased estimates *of* lower bounds.

B.3 Correctness

Our first correctness result states that the representations computed by lazy knowledge compilation correctly reflect the distribution of the input expression.

Lemma B.2 (Restatement of Lemma 4.4). *Suppose* $\Gamma \vdash e : \tau$ *and* $\gamma :_E \Gamma$. *Then for any predicate* $\phi : \Omega \to 2$ *and rose tree path* $\sigma \in \mathbb{N}^*$, *if* $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$, *then:*

(1) $\phi(\omega)$ *implies* $\phi_u(\omega)$, *and*
(2) for all $\omega \in \Omega$ *such that* $\phi_u(\omega)$ *holds, we have* $[\![v]\!]_V(\omega) = [\![e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma])$.

Proof. The proof is by induction on the derivation of $\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$. We consider each rule in turn, establishing conditions (1) and (2) from the Lemma statement in each case:

- **Case:** $\textsc{False}\ \dfrac{\phi \implies \mathsf{F}}{\gamma, \phi, \sigma \vdash e \rightsquigarrow \varepsilon \dashv \mathsf{F}}$

  (1) In the conclusion of the rule, ϕ_u is F. That $\phi \implies \mathsf{F}$ is given in the premise.

(2) Because $\phi_u = \mathsf{F}$, the condition holds vacuously.

- **Case:**

$$\textsc{Flip}\ \frac{}{\gamma, \phi, \sigma \vdash \mathbf{flip}\ r \rightsquigarrow \{(\mathsf{true}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\omega(\sigma) \le r}, (\mathsf{false}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\neg(\omega(\sigma) \le r)}\} \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication holds trivially.
(2) By the random-seed semantics (Listing A.4), we have that in any environment γ,

$$[\![\mathbf{flip}\ r]\!](\gamma, \omega[\sigma]) = \begin{cases} \mathsf{roll}(\mathsf{in}_1\ \star) & \omega(\sigma) \le r \\ \mathsf{roll}(\mathsf{in}_2\ \star) & \textit{otherwise} \end{cases}.$$

By the semantics of symbolic values (Listing B.2), we have that

$$\begin{aligned} [\![v]\!]_V(\omega) &= [\![\{(\mathsf{true}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\omega(\sigma) \le r}, (\mathsf{false}\ \langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}})_{\neg(\omega(\sigma) \le r)}\}]\!]_V(\omega) \\ &= \begin{cases} \mathsf{roll}(\mathsf{in}_1\ [\![\langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}}]\!]_T(\omega)) & \omega(\sigma) \le r \\ \mathsf{roll}(\mathsf{in}_2\ [\![\langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}}]\!]_T(\omega)) & \omega(\sigma) > r \\ \bot & \textit{otherwise} \end{cases} \end{aligned}$$

To see that the two semantics coincide, note that the cases $\omega(\sigma) \le r$ and $\omega(\sigma) > r$ are exhaustive, and that for all ω, $[\![\langle (), \varepsilon, \varepsilon \rangle_{\mathsf{T}}]\!]_T(\omega) = [\![()]\!](\varepsilon, \omega) = \star$.

- **Case:**

$$\textsc{Unit}\ \frac{}{\gamma, \phi, \sigma \vdash () \rightsquigarrow \star_{\mathsf{T}} \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication trivially holds.
(2) We have $[\![()]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) = \star = [\![\star_{\mathsf{T}}]\!]_V(\omega)$.

- **Case:**

$$\textsc{Constructor}\ \frac{}{\gamma, \phi, \sigma \vdash \ell_i\ e \rightsquigarrow (\ell_i\ \langle e, \gamma, \sigma \rangle_{\mathsf{T}})_{\mathsf{T}} \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication trivially holds.
(2) We have $[\![\ell_i\ e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) = \mathsf{roll}(\mathsf{in}_i\ [\![e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma])) = [\![(\ell_i\ \langle e, \gamma, \sigma \rangle_{\mathsf{T}})_{\mathsf{T}}]\!]_V(\omega)$.

- **Case:**

$$\textsc{Pair}\ \frac{}{\gamma, \phi, \sigma \vdash (e_1, e_2) \rightsquigarrow (\langle e_1, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}, \langle e_2, \gamma, \sigma \cdot 2 \rangle_{\mathsf{T}}) \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication trivially holds.
(2) We have

$$\begin{aligned} [\![(e_1, e_2)]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) &= ([\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1]), [\![e_2]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 2])) \\ &= ([\![\langle e_1, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}]\!]_T(\omega), [\![\langle e_2, \gamma, \sigma \cdot 2 \rangle_{\mathsf{T}})]\!]_T(\omega)) \\ &= [\![(\langle e_1, \gamma, \sigma \cdot 1 \rangle_{\mathsf{T}}, \langle e_2, \gamma, \sigma \cdot 2 \rangle_{\mathsf{T}})]\!]_V(\omega). \end{aligned}$$

- **Case:**

$$\textsc{Abs}\ \frac{}{\gamma, \phi, \sigma \vdash \lambda x.\ e \rightsquigarrow \langle x.e, \gamma \rangle_{\mathsf{T}} \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication trivially holds.
(2) We have

$$\begin{aligned}[\![\lambda x.e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) &= \lambda(v, \omega').[\![e]\!]([\![\gamma]\!]_E(\omega)[x \mapsto v], \omega') \\ &= [\![\langle x.e, \gamma\rangle_{\mathsf{T}}]\!]_V(\omega).\end{aligned}$$

- **Case:**

$$\frac{\text{Y}}{\gamma, \phi, \sigma \vdash Y(\lambda x_1.\lambda x_2.e) \rightsquigarrow \langle x_2.e, \gamma[x_1 \mapsto \langle Y(\lambda x_1.\lambda x_2.e), \gamma, \sigma\rangle_{\mathsf{T}}]\rangle_{\mathsf{T}} \dashv \mathsf{T}}$$

(1) In the conclusion of the rule, ϕ_u is T, so the implication trivially holds.
(2) We have

$$\begin{aligned}&[\![Y(\lambda x_1.\lambda x_2.e)]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) \\ &\quad= \mathsf{fix}(\lambda f.\lambda(y, \omega').[\![e]\!]([\![\gamma]\!]_E(\omega)[x_1 \mapsto f, x_2 \mapsto y], \omega')) \\ &\quad= \lambda(y, \omega').[\![e]\!]([\![\gamma]\!]_E(\omega)[x_1 \mapsto [\![Y(\lambda x_1.\lambda x_2.e)]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]), x_2 \mapsto y], \omega') \\ &\quad= \lambda(y, \omega').[\![e]\!]([\![\gamma[x_1 \mapsto \langle Y(\lambda x_1.\lambda x_2.e), \gamma, \sigma\rangle_{\mathsf{T}}]]\!]_E(\omega)[x_2 \mapsto y], \omega') \\ &\quad= [\![\langle x_2.e, \gamma[x_1 \mapsto \langle Y(\lambda x_1.\lambda x_2.e), \gamma, \sigma\rangle_{\mathsf{T}}]\rangle_{\mathsf{T}}]\!]_V(\omega).\end{aligned}$$

- **Case:**

$$\frac{\textsc{CacheHit} \quad \gamma, \phi^*, \sigma \vdash e \rightsquigarrow v \dashv \phi_u \qquad \phi \implies \phi_u}{\gamma, \phi, \sigma \vdash e \rightsquigarrow v \dashv \phi_u}$$

(1) That ϕ implies ϕ_u is given as a premise.
(2) We can directly apply the inductive hypothesis for the judgment $\gamma, \phi^*, \sigma \vdash e \rightsquigarrow v \dashv \phi_u$, which establishes that the desired equation holds for all ω such that $\phi_u(\omega)$ holds.

- **Case:**

$$\frac{\textsc{Var} \quad \gamma(x) = \{\langle e_i, \gamma_i, \sigma_i\rangle_{\phi_i}\}_{i \in I} \qquad \gamma_i, \phi \wedge \phi_i, \sigma_i \vdash e_i \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i \in I}, \mathsf{T}) = (v, \phi_u)}{\gamma, \phi, \sigma \vdash x \rightsquigarrow v \dashv \phi_u}$$

(1) By the inductive hypothesis, $\phi \wedge \phi_i \implies \phi_i^u$ for each i. By Lemma 4.3, we also have that $\phi_i \wedge \phi_i^u \implies \phi_u$ for each i. It follows that $\phi \wedge \phi_i \implies \phi_u$ for each i. We also have from Lemma 4.3 that $(\forall i \in I, \neg\phi_i) \implies \phi_u$. If ϕ holds, then either some ϕ_i also holds (in which case ϕ_u holds by the first fact we derived), or no ϕ_i holds (in which case ϕ_u holds by the second fact we derived). Therefore, $\phi \implies \phi_u$, as required.
(2) Let τ be the type of the variable x. We have

$$\begin{aligned}[\![x]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) &= [\![\gamma]\!]_E(\omega)(x) \\ &= [\![\gamma(x)]\!]_V(\omega) \\ &= [\![\{\langle e_i, \gamma_i, \sigma_i\rangle_{\phi_i}\}_{i \in I}]\!]_V(\omega) \\ &= \begin{cases} [\![e_i]\!]([\![\gamma_i]\!]_E(\omega), \omega[\sigma_i]) & \text{if } \phi_i(\omega) \\ \bot_\tau & \text{if } \forall i \in I.\neg\phi_i(\omega) \end{cases}\end{aligned}$$

By the inductive hypothesis, for each $i \in I$, we have that $[\![e_i]\!]([\![\gamma_i]\!]_E(\omega), \omega[\sigma_i]) = [\![v_i]\!]_V(\omega)$ whenever $\phi_i^u(\omega)$ holds. Then, by Lemma 4.3, we have that whenever $\phi_u(\omega)$ holds, $[\![v]\!]_V(\omega)$ is equal to the expression derived above, as required.

- **Case:**

$$\textsc{App}\ \frac{\gamma, \phi, \sigma \cdot 0 \vdash e_1 \rightsquigarrow \{\langle x_i.e_i^f, \gamma_i\rangle_{\phi_i}\}_{i\in I} \dashv \phi_u \qquad \gamma_i[x_i \mapsto \langle e_2, \gamma, \sigma \cdot 1\rangle_{\mathsf{T}}], \phi \wedge \phi_i, \sigma \cdot 2 \vdash e_i^f \rightsquigarrow v_i \dashv \phi_i^u \qquad \mathsf{join}(\{(v_i, \phi_i, \phi_i^u)\}_{i\in I}, \phi_u) = (v, \phi_u^*)}{\gamma, \phi, \sigma \vdash e_1\ e_2 \rightsquigarrow v \dashv \phi_u^*}$$

(1) By the inductive hypothesis, we know that $\phi \implies \phi_u$ and that $\phi \wedge \phi_i \implies \phi_i^u$ for each $i \in I$. We also have, by Lemma 4.3, that $\phi_u \wedge \phi_i \wedge \phi_i^u \implies \phi_u^*$ for all $i \in I$, and that $(\phi_u \wedge \forall i, \neg\phi_i) \implies \phi_u^*$. Putting these facts together, we have that $\phi \wedge \phi_i \implies \phi_u^*$ for all $i \in I$, and $\phi \wedge \forall i, \neg\phi_i \implies \phi_u^*$. Combining these two facts, we obtain that $\phi \implies \phi_u^*$.

(2) By the inductive hypothesis for the first premise, we have that for all ω such that ϕ_u holds,

$$\begin{aligned}[\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 0]) &= [\![\langle x_i.e_i^f, \gamma_i\rangle_{\phi_i}\}_{i\in I}]\!]_V(\omega) \\ &= \lambda(v, \omega').\begin{cases}[\![e_i^f]\!]([\![\gamma_i]\!]_E(\omega)[x_i \mapsto v], \omega') & \text{if } \phi_i(\omega) \\ \bot_{\tau_2} & \text{otherwise}\end{cases}.\end{aligned}$$

Therefore, when $\phi_u(\omega)$ holds, we have

$$\begin{aligned}&[\![e_1\ e_2]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) \\ &\quad= [\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 0])([\![e_2]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1]), \omega[\sigma \cdot 2]) \\ &\quad= \begin{cases}[\![e_i^f]\!]([\![\gamma_i]\!]_E(\omega)[x_i \mapsto [\![e_2]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1])], \omega[\sigma \cdot 2]) & \text{if } \phi_i(\omega) \\ \bot_{\tau_2} & \text{otherwise}\end{cases} \\ &\quad= \begin{cases}[\![e_i^f]\!]([\![\gamma_i[x_i \mapsto \langle e_2, \gamma, \sigma \cdot 1\rangle_{\mathsf{T}}]]\!]_E(\omega), \omega[\sigma \cdot 2]) & \text{if } \phi_i(\omega) \\ \bot_{\tau_2} & \text{otherwise}\end{cases}\end{aligned}$$

By the inductive hypothesis for the second premise, we have that for each $i \in I$, if $\phi_i^u(\omega)$ holds, $[\![e_i^f]\!]([\![\gamma_i[x_i \mapsto \langle e_2, \gamma, \sigma \cdot 1\rangle_{\mathsf{T}}]]\!]_E(\omega), \omega[\sigma \cdot 2]) = [\![v_i]\!]_V(\omega)$. We can then apply Lemma 4.3 to obtain that whenever $\phi_u^*(\omega)$ holds, $[\![v]\!](\omega)$ is equal to the expression derived above, as required.

- **Case:**

$$\textsc{MatchProd}\ \frac{\gamma, \phi, \sigma \cdot 1 \vdash e_1 \rightsquigarrow (v_1, v_2) \dashv \phi_u \qquad \gamma[x_1 \mapsto v_1, x_2 \mapsto v_2], \phi, \sigma \cdot 2 \vdash e_2 \rightsquigarrow v \dashv \phi_u^*}{\gamma, \phi, \sigma \vdash \textbf{match}\ e_1\ \textbf{with}\ (x_1, x_2) \Rightarrow e_2 \rightsquigarrow v \dashv \phi_u \wedge \phi_u^*}$$

(1) By the inductive hypotheses for each premise, we have $\phi \implies \phi_u$ and $\phi \implies \phi_u^*$. From this we derive that $\phi \implies \phi_u \wedge \phi_u^*$.

(2) Let ω be such that $\phi_u(\omega) \wedge \phi_u^*(\omega)$ holds. By the inductive hypothesis for the first premise, $[\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1]) = [\![(v_1, v_2)]\!]_V(\omega) = ([\![v_1]\!]_V(\omega), [\![v_2]\!]_V(\omega))$. So we have

$$\begin{aligned}&[\![\textbf{match}\ e_1\ \textbf{with}\ (x_1, x_2) \Rightarrow e_2]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) \\ &\quad= [\![e_2]\!]([\![\gamma]\!]_E(\omega)[x_1 \mapsto \pi_1\ [\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1]), x_2 \mapsto \pi_2\ [\![e_1]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 1])], \omega[\sigma \cdot 2]) \\ &\quad= [\![e_2]\!]([\![\gamma]\!]_E(\omega)[x_1 \mapsto [\![v_1]\!]_V(\omega), x_2 \mapsto [\![v_2]\!]_V(\omega)], \omega[\sigma \cdot 2]) \\ &\quad= [\![e_2]\!]([\![\gamma[x_1 \mapsto v_1, x_2 \mapsto v_2]]\!]_E(\omega), \omega[\sigma \cdot 2]) \\ &\quad= [\![v]\!]_V(\omega),\end{aligned}$$

where the last step follows from the inductive hypothesis for the second premise.

- **Case:**

$$\textsc{MatchSum}\ \frac{\gamma, \phi, \sigma \cdot 0 \vdash e \rightsquigarrow \{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y \in Y} \dashv \phi_u \qquad \gamma[x_y \mapsto v_y], \phi \wedge \phi_y, \sigma \cdot i_y \vdash e_{i_y} \rightsquigarrow v_y^* \dashv \phi_y^u \quad \mathsf{join}(\{(v_y^*, \phi_y, \phi_y^u)\}_{y \in Y}, \phi_u) = (v, \phi_u^*)}{\gamma, \phi, \sigma \vdash \textbf{match}\ e\ \textbf{with}\ \ell_1\ x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n\ x_n \Rightarrow e_n \rightsquigarrow v \dashv \phi_u^*}$$

(1) By the inductive hypotheses for the first two premises, we have that $\phi \implies \phi_u$, and that for each $y \in Y$, $\phi \wedge \phi_y \implies \phi_y^u$. By Lemma 4.3, we have that $\phi_u \wedge \phi_y \wedge \phi_y^u \implies \phi_u^*$ for all $y \in Y$, and also that $(\phi_u \wedge \forall y, \neg\phi_y) \implies \phi_u^*$. Putting these facts together, we obtain that for all $y \in Y$, $\phi \wedge \phi_y \implies \phi_u^*$, and also that $(\phi \wedge \forall y, \neg\phi_y) \implies \phi_u^*$. These two results imply that $\phi \implies \phi_u^*$, as desired.

(2) By the inductive hypothesis for the first premise, we have that whenever $\phi_u(\omega)$ holds, we have

$$\begin{aligned}[\![e]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma \cdot 0]) &= [\![\{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y \in Y}]\!]_V(\omega) \\ &= \begin{cases} \mathsf{roll}(\mathsf{in}_{i_y}\ [\![v_y]\!]_V(\omega)) & \text{if } \phi_y(\omega) \\ \mathsf{roll}(\bot) & \text{otherwise} \end{cases},\end{aligned}$$

and therefore, that

$$\begin{aligned}&[\![\textbf{match}\ e\ \textbf{with}\ \{\ell_1 x_1 \Rightarrow e_1 \mid \cdots \mid \ell_n x_n \Rightarrow e_n\}]\!]([\![\gamma]\!]_E(\omega), \omega[\sigma]) \\ &= \begin{cases} [\![e_{i_y}]\!]([\![\gamma]\!]_E(\omega)[x_y \mapsto [\![v_y]\!]_V(\omega)], \omega[\sigma \cdot i_y]) & \text{if } \phi_y(\omega) \\ \bot_\tau & \text{otherwise} \end{cases} \\ &= \begin{cases} [\![e_{i_y}]\!]([\![\gamma[x_y \mapsto v_y]]\!]_E(\omega), \omega[\sigma \cdot i_y]) & \text{if } \phi_y(\omega) \\ \bot_\tau & \text{otherwise} \end{cases}.\end{aligned}$$

By the inductive hypothesis for the second premise, for each $y \in Y$, we have that whenever $\phi_y^u(\omega)$ holds,

$$[\![v_y^*]\!]_V(\omega) = [\![e_{i_y}]\!]([\![\gamma[x_y \mapsto v_y]]\!]_E(\omega), \omega[\sigma \cdot i_y]).$$

We can then apply Lemma 4.3 to obtain that whenever ϕ_u^* holds, $[\![v]\!]_V(\omega)$ is equal to the expression derived above, as required.

□

Weighted model counts and marginal probabilities. Specializing this lemma to the case when e has sum type (e.g. $\mathbb{B}$), we see that running lazy knowledge compilation produces a *guard formula* ϕ characterizing the seeds ω for which the program returns each constructor of the sum. A *weighted model count* of this formula—a linear-time operation in the size of the binary decision diagram—calculates the exact probability with which $[\![e]\!]$ generates a value with the corresponding top-level constructor.

Definition B.3 (Weighted model count). Let $\mathcal{X}$ be a set of Boolean variables, let $\phi : \{0,1\}^{\mathcal{X}} \to \{0,1\}$ be a Boolean formula over the variables, and let $wt : \mathcal{X} \to [0,1]$ be a function assigning to each Boolean variable a continuous weight. Then the *weighted model count* $\mathrm{WMC}(\phi, wt)$ is defined as

$$\mathrm{WMC}(\phi, wt) := \sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}} \mid \phi(\mathbf{x})\}} \prod_{b \in \mathcal{X}} wt(b)^{\mathbf{x}(b)} \cdot (1 - wt(b))^{1 - \mathbf{x}(b)}.$$

Recall that the predicates $\phi : \Omega \to \{0,1\}$ generated by lazy knowledge compilation are defined over *atomic predicates* of the form $\omega(\sigma) \leq r$. Thus, they can be equivalently viewed either Boolean formulae over Boolean variables $b_{(\sigma,r)}$ representing these atomic predicates. Formally, we write $\mathcal{X}(\phi) := \{b_{(\sigma,r)} \mid \text{"}\omega(\sigma) \leq r\text{" appears as an atomic predicate in } \phi\}$ for the set of these Boolean variables relevant to a given predicate ϕ. We then write $\lfloor\phi\rfloor : \{0,1\}^{\mathcal{X}(\phi)} \to \{0,1\}$ for the *formula* that ϕ encodes over these Boolean variables. Note that for all ω, $\phi(\omega) = \lfloor\phi\rfloor(((b_{(\sigma,r)} \mapsto \mathbf{1}_{[0,r]}(\omega(\sigma)))_{b_{(\sigma,r)} \in \mathcal{X}(\phi)})$. Our next lemma is that the weighted model count of the formula $\lfloor\phi\rfloor$ with respect to a particular weight function yields the probability under μ_Ω that the predicate ϕ holds.

Lemma B.4. *Let ϕ be a predicate on Ω defined in terms of atomic predicates $\omega(\sigma) \leq r$. Then* $\mathbb{P}_{\omega\sim\mu_\Omega}[\phi(\omega)] = \mathrm{WMC}(\lfloor\phi\rfloor, (b_{(\sigma,r)} \mapsto r)_{b_{(\sigma,r)} \in \mathcal{X}(\phi)})$.

Proof. If and only if $\phi(\omega)$ holds, then $(b_{(\sigma,r)} \mapsto [\omega(\sigma) \leq r])_{b_{(\sigma,r)} \in \mathcal{X}(\phi)}$ is satisfying assignment for $\lfloor\phi\rfloor$. Thus, $\phi(\omega)$ can be equivalently written as a sum over satisfying assignments of $\lfloor\phi\rfloor$, where each summand checks if ω is consistent with the values taken on by the variables $b_{(\sigma,r)}$:

$$\phi(\omega) = \sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}(\phi)} \mid \lfloor\phi\rfloor(\mathbf{x})\}} \prod_{b_{(\sigma,r)} \in \mathcal{X}(\phi)} [\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}.$$

Taking an expectation over μ_Ω on both sides of the equation, we obtain

$$\begin{aligned}
\mathbb{P}_{\omega\sim\mu_\Omega}[\phi(\omega)] &= \mathbb{E}_{\omega\sim\mu_\Omega}\left[\sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}(\phi)} \mid \lfloor\phi\rfloor(\mathbf{x})\}} \prod_{b_{(\sigma,r)} \in \mathcal{X}(\phi)} [\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}\right] \\
&= \sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}(\phi)} \mid \lfloor\phi\rfloor(\mathbf{x})\}} \mathbb{E}_{\omega\sim\mu_\Omega}\left[\prod_{b_{(\sigma,r)} \in \mathcal{X}(\phi)} [\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}\right] \\
&= \sum_{\{\mathbf{x} \in \{0,1\}^{\mathcal{X}(\phi)} \mid \lfloor\phi\rfloor(\mathbf{x})\}} \prod_{b_{(\sigma,r)} \in \mathcal{X}(\phi)} \mathbb{E}_{\omega\sim\mu_\Omega}\left[[\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}\right],
\end{aligned}$$

where the last step follows because each dimension $\omega(\sigma)$ of ω is independent from all others, and no two variables $b_{(\sigma,r)}$ in $\mathcal{X}(\phi)$ share the same σ. Now, note that when $\mathbf{x}(b_{(\sigma,r)}) = 1$,

$$\mathbb{E}_{\omega\sim\mu_\Omega}\left[[\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}\right] = r,$$

and when $\mathbf{x}(b_{(\sigma,r)}) = 0$

$$\mathbb{E}_{\omega\sim\mu_\Omega}\left[[\omega(\sigma) \leq r]^{\mathbf{x}(b_{\sigma,r})} [\omega(\sigma) > r]^{1-\mathbf{x}(b_{\sigma,r})}\right] = 1 - r.$$

Thus, the sum-of-products above is exactly the weighted model count

$$\mathrm{WMC}(\lfloor\phi\rfloor, (b_{(\sigma,r)} \mapsto r)_{b_{(\sigma,r)} \in \mathcal{X}(\phi)}).$$

□

Putting Lemmas 4.4 and B.4 together, we obtain our correctness result for top-level marginal probability queries on terms of sum type.

Theorem B.5 (Restatement of Thm. 4.7). *Let e be a closed term of sum type. If $\varepsilon, \mathsf{T}, \varepsilon \vdash e \rightsquigarrow \{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y\in Y} \dashv \mathsf{T}$, then*

$$[\![e]\!]_P(\mathrm{roll}\ \mathrm{in}_{i_y}([\![\tau_{i_y}]\!])) = \mathrm{WMC}(\lfloor\phi_y\rfloor, (b_{(\sigma,r)} \mapsto r)_{b_{(\sigma,r)} \in \mathcal{X}(\phi_y)}).$$

Proof. Suppose e is a closed term of sum type, i.e., we have the judgment $\vdash e : \mu\alpha.\ell_1\tau_1 + \cdots + \ell_n\tau_n$. Suppose also that lazy knowledge compilation on e yields the result $\{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y\in Y}$. We are interested in the probability with which e generates a value with a particular constructor ℓ_{i_y}, that is, $[\![e]\!]_P(\mathsf{roll}\ \mathsf{in}_{i_y}([\![\tau_{i_y}]\!]))$.

By the definition of $[\![\cdot]\!]_P$,

$$[\![e]\!]_P(\mathsf{roll}\ \mathsf{in}_{i_y}([\![\tau_{i_y}]\!])) = \mathbb{E}_{\omega\sim\mu_\Omega}\left[1_{\mathsf{roll}\ \mathsf{in}_{i_y}([\![\tau_{i_y}]\!])}([\![e]\!](\omega))\right].$$

By Lemma 4.4, for all $\omega \in \Omega$, we have $[\![e]\!](\omega) = [\![\{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y\in Y}]\!]_V(\omega)$. By the definition of $[\![\cdot]\!]_V$, we have $[\![\{(\ell_{i_y}\ v_y)_{\phi_y}\}_{y\in Y}]\!]_V(\omega)$ yields a value of the form $\mathsf{roll}\ \mathsf{in}_{i_y}\ x$ for some $x \in [\![\tau_{i_y}]\!]$ precisely when $\phi_y(\omega)$ holds. Therefore, $1_{\mathsf{roll}\ \mathsf{in}_{i_y}([\![\tau_{i_y}]\!])}([\![e]\!](\omega)) = \phi_y(\omega)$ for each $y \in Y$. Substituting into the expected value expression above, we have that the probability with which e generates a value with constructor ℓ_{i_y} is equal to $\mathbb{P}_{\omega\sim\mu_\Omega}[\phi_y(\omega)]$. We conclude by applying Lemma B.4 to relate this quantity to the weighted model count for the formula $\lfloor\phi_y\rfloor$. □

C NOTES ON RECURSIVE TYPES

Given a recursive type definition $\tau := \mu\alpha.\ \ell_1\tau_1 + \cdots + \ell_n\tau_n$ and a typing environment Δ (mapping free type variables to pointed quasi-Borel predomains), we define the functor $F(X_\alpha) = (\sqcup_{i=1}^n [\![\tau_i]\!](\Delta[\alpha \mapsto X_\alpha]))_\perp$. Then we consider the sequence of spaces $\{\perp\}, F(\{\perp\}), F^2(\{\perp\}), \ldots$, where $\{\perp\}$ is the one-point quasi-Borel predomain. Between each successive pair of spaces $F^n(\{\perp\})$ and $F^{n+1}(\{\perp\})$ we define an *embedding-projection pair* $F^n(\{\perp\}) \overset{e_n}{\hookrightarrow} F^{n+1}(\{\perp\}), F^n(\{\perp\}) \overset{p_n}{\hookleftarrow} F^{n+1}(\{\perp\})$. The embedding e_n is an injection, mapping $\perp$ to $\perp$, and $\mathsf{in}_i\ v$ to $\mathsf{in}_i\ v'$, where v' is obtained by applying e_{n-1} to any values of type $F^{n-1}(\{\perp\})$ stored inside v.[19] The projection p_n is a surjection going in the opposite direction: p_0 maps any value to $\perp$, and for $n > 0$, p_n maps $\perp$ to $\perp$ and $\mathsf{in}_i\ v'$ to $\mathsf{in}_i\ v$, where v is obtained by applying p_{n-1} to any values of type $F^n(\{\perp\})$ stored inside v'. Then our recursive type τ is interpreted as the space $[\![\tau]\!](\Delta)$ of infinite tuples $x = (x_0, x_1, \ldots)$, where for all i, $x_i \in F^i(\{\perp\})$ and $x_i = p_{i+1}(x_{i+1})$. We can build values of recursive type using the map $\mathsf{roll} : F([\![\tau]\!](\Delta)) \to [\![\tau]\!](\Delta)$, which maps $\perp$ to $(\perp, \perp, \ldots)$ and $\mathsf{in}_i\ x$ to $(p_0(\mathsf{in}_i\ x_0), p_1(\mathsf{in}_i\ x_1), \ldots)$. We can pattern-match against values of recursive type using the map $\mathsf{unroll} : [\![\tau]\!](\Delta) \to F([\![\tau]\!](\Delta))$, which sends x to $\perp$ if $x = (\perp, \perp, \ldots)$, and to $\mathsf{in}_i\ (\mathsf{in}_i^{-1}(x_1), \mathsf{in}_i^{-1}(x_2), \ldots)$ if $x_1 = \mathsf{in}_i\ \perp$. As concrete examples: $[\![\mathbb{B}]\!]$ is isomorphic to $2_\perp$; $[\![\mathbb{N}]\!]$ is isomorphic to $(\mathbb{N} \cup \{\infty\})_\perp$, and $[\![\mathbb{L}\ \tau]\!]$ is isomorphic to $(\sqcup_{i\in\mathbb{N}} [\![\tau]\!]^i \sqcup [\![\tau]\!]^{\mathbb{N}})_\perp$. That is, unlike in the development of Vákár et al. [54], our recursive types have infinite inhabitants, similar to those of popular lazy languages such as Haskell.

D SUM-AND-SAMPLE LAZY SUBPROBLEM MONTE CARLO

See Alg. 2 for the multi-particle variant of LPSMC.

E DISCRETE INFERENCE BENCHMARKS

See Tab. 4 for results on additional discrete inference benchmarks.

F ROBUSTNESS ANALYSIS: EVALUATION ON RANDOMLY GENERATED PROGRAMS

F.1 Description of Grammars and Dataset Generation

The same set of grammars and dataset generation process are used for both the evaluation on randomly generated programs as well as the synthesis evaluation.

We construct a set of six grammars that exhibit a range of common forms of independence in probabilistic models. All grammars share the same productions for natural number and boolean

[19] Formally, $v' = G(e_{n-1})(v)$, where G is the functor $[\![\tau_i]\!](\Delta[\alpha \mapsto -])$.

Algorithm 2 Lazy Sum-and-Sample Subproblem Monte Carlo

Require: Closed query expression $\vdash e$: **SuspendibleBool**
Require: Integer K
Require: Maximum number of iterations N
Ensure: $\mathbb{E}[p_\mathsf{T}] = [\![e]\!]_P(\{\underbrace{\mathsf{in}_{\mathsf{Suspend}}(\ldots\mathsf{in}_{\mathsf{Suspend}}}_{i \text{ times}}(\mathsf{in}_{\mathsf{FinallyTrue}}(\star))\ldots) \mid 0 \le i \le N\})$
Ensure: $\mathbb{E}[p_\mathsf{F}] = [\![e]\!]_P(\{\underbrace{\mathsf{in}_{\mathsf{Suspend}}(\ldots\mathsf{in}_{\mathsf{Suspend}}}_{i \text{ times}}(\mathsf{in}_{\mathsf{FinallyFalse}}(\star))\ldots) \mid 0 \le i \le N\})$

$queue \leftarrow [(e, \mathsf{T}, [], \varepsilon, 1, 0)]$
$p_\mathsf{T} \leftarrow 0$
$p_\mathsf{F} \leftarrow 0$
while *queue* is not empty **do**
  $(e, \gamma, \phi, \sigma, w, i) \leftarrow queue.\text{pop!}()$
  $(\mathsf{FinallyTrue}_{\phi_T}, \mathsf{FinallyFalse}_{\phi_F}, \mathsf{Suspend}(t)_{\phi_S}) \leftarrow \textbf{compile}(e, \gamma, \phi, \sigma)$
  $p_\mathsf{T} \leftarrow p_\mathsf{T} + w \cdot \mathrm{WMC}(\phi \wedge \phi_T)$
  $p_\mathsf{F} \leftarrow p_\mathsf{F} + w \cdot \mathrm{WMC}(\phi \wedge \phi_F)$
  if $i < N \wedge \phi_S \neq \mathsf{F}$ **then**
    $\phi \leftarrow \phi \wedge \phi_S$
    $\phi_k \leftarrow \textbf{topk}(\phi, K)$ ▷ Prune all but top K paths from ϕ
    $\phi_s, q \leftarrow \textbf{sample}(\phi \wedge \neg\phi_k)$ ▷ Sample one of the remaining paths
    $\phi_{sk} = ite(\omega(\sigma_{fresh}) \le \frac{1}{1+q^{-1}}, \phi_k, \phi_s)$ ▷ Add coin flip to mix top-K paths with sample
    $w \leftarrow w \cdot (1 + \frac{1}{q})$ ▷ Update importance weight
    for $\langle e_j, \gamma_j, \sigma_j \rangle_{\phi_j} \in t$ **do** ▷ Consider possible resumptions of suspended query
      $queue.\text{push!}((e_j, \phi \wedge \phi_{sk} \wedge \phi_j, \gamma_j, \sigma_j, w, i+1))$
    end for
  end if
end while
return $(p_\mathsf{T}, p_\mathsf{F})$

Table 4. **Additional Exact Inference Benchmarks.** Smaller benchmarks from [26] extending Table 1.

Benchmark	Enumeration		Knowledge Compilation		BDD Stats (Ours)	
	Eager	*Lazy*	*Dice.jl*	*Ours*	vars	nodes
Bayesian Networks						
Noisy Or (8 nodes)	4.89 ms	0.87 ms	0.98 ms	**0.17 ms**	14	66
Burglary (6 nodes)	0.26 ms	**0.12 ms**	0.49 ms	0.13 ms	7	14
Evidence1	**0.02 ms**	**0.02 ms**	0.90 ms	0.18 ms	2	3
Evidence2	**0.01 ms**	0.02 ms	0.50 ms	0.18 ms	3	5
Grass	1.04 ms	**0.17 ms**	0.43 ms	0.25 ms	7	14
Murder Mystery	**0.06 ms**	**0.06 ms**	0.50 ms	0.20 ms	4	6
Two Coins	**0.04 ms**	**0.03 ms**	0.42 ms	0.19 ms	2	3
Other Benchmarks from [26]						
Caesar Cipher (100 chars)	timeout	313.25 ms	**69.69 ms**	**69.61 ms**	303	909
Figure 1 from [26]	timeout	timeout	**2.29 ms**	10.53 ms	201	402

Listing F.1 Primitive operations shared among all grammars.

$$
\begin{array}{lll}
nat & ::= & 0 \mid \cdots \mid 9 \mid \mathit{geometric}(0.2) \mid x \mid \mathrm{succ}(nat) \mid nat_1 + nat_2 \mid nat_1 - nat_2 \mid \\
 & & \textbf{match}\ nat_1\ \textbf{with}\ (\mathrm{zero}\ \_ \Rightarrow nat_2 \mid \mathrm{succ}\ x \Rightarrow nat_3) \mid \textbf{if}\ bool\ \textbf{then}\ nat_1\ \textbf{else}\ nat_2 \mid \\
 & & \textbf{let}\ x = nat_1\ \textbf{in}\ nat_2 \\
bool & ::= & \textbf{flip}(0.1) \mid \cdots \mid \textbf{flip}(0.9) \mid \mathit{isEven}(nat) \mid nat_1 = nat_2 \mid nat_1 > nat_2
\end{array}
$$

types, shown in Listing F.1, and differ only in their top-level independence structure given by their *list* production. In each grammar, the top-level expression has the form $\textbf{let}\ x_{global} = nat_1\ \textbf{in let}\ x_{take} = \mathit{take}(nat_2, x_{input})\ \textbf{in}\ list$. That is, programs first generate a variable x_{global} that can be used anywhere else in the expression, as well as a finite prefix x_{take} of the input stream x_{input} to read. In all programs from our grammars, the output list has the same length as x_{take}. The particular grammars we use are described below.

- cIID-Gen: Programs that ignore the input stream and sample sequences of numbers that are conditionally IID with respect to x_{global}. We express this kind of independence through the production $list_{cIID} ::= \textbf{let}\ x_{units} = \mathit{map}(\lambda x.\ \mathrm{unit}(),\ x_{take})\ \textbf{in}\ \mathit{map}(\lambda x_{unit}.\ nat,\ x_{units})$, where x_{units} is introduced to ignore the input list values, and the second *map* expresses independence across output elements. Example distributions include *length 3 lists of even numbers*, *even length lists of large numbers*, and *random length lists where all entries are equal to the length of the list.*
- cIID-IO-Gen: This grammar is identical to the cIID-Gen grammar but without replacing each input with unit(), allowing for distributions like *length 3 lists where geometric noise is added to each input* and *random length lists where each input is doubled.*
- Markov-Gen: A Markov chain of natural numbers can be expressed similarly to cIID-Gen but with a *scan* instead of a *map*. This gives the following production: $list_{Markov} ::= \textbf{let}\ x_{units} = \mathit{map}(\lambda x.\ \mathrm{unit}(), x_{take})\ \textbf{in}\ \mathit{scan}(\lambda x_{acc},\ x_{unit}.\ nat,\ x_{units})$. Example distributions this can express include *random sorted lists* or *piecewise constant lists.*
- Markov-IO-Gen: This is similar to Markov-Gen but without masking the input list. This can also be viewed as an Markov Decision Process, where the input is a sequence of agent actions and program itself defines how the environment responds to actions. This can express distributions like *a noisy cumulative sum of the list.*
- HMM-Gen: Programs that generate Markov chains of latent states and independently transform each one according to arbitrary stochastic logic. This can be expressed by wrapping the Markov-Gen list production in an extra *map*: $list_{HMM} ::= \mathit{map}(\lambda x.\ nat, list_{Markov})$.
- HMM-IO-Gen: This is identical to HMM-Gen but using $list_{Markov-IO}$ in place of $list_{Markov}$, and can also be viewed as a Partially-Observable Markov Decision Process [27].

We sample 100 random programs from each grammar. To encourage more complex programs, we first sample the program size by uniformly choosing the number of non-leaf nodes between 0 and 10, then sample a program given the size. Example outputs are generated by running the sampled program on a randomly generated input.

For each program, we sample eight input lists of length 30 containing uniformly distributed digits. These lists are finite representations of the input stream, and in practice programs rarely use the entire list. The program is run on each input to sample an output list, and a likelihood query is constructed as the probability of $\mathit{equals?}(x_{output}, \mathit{program}(x_{input}))$ evaluating to true(). To encourage variety we remove duplicate expressions, expressions whose output examples only contain one unique number, and expressions with *scan*s and *map*s that don't use their accumulators and inputs (unless these inputs are unit).

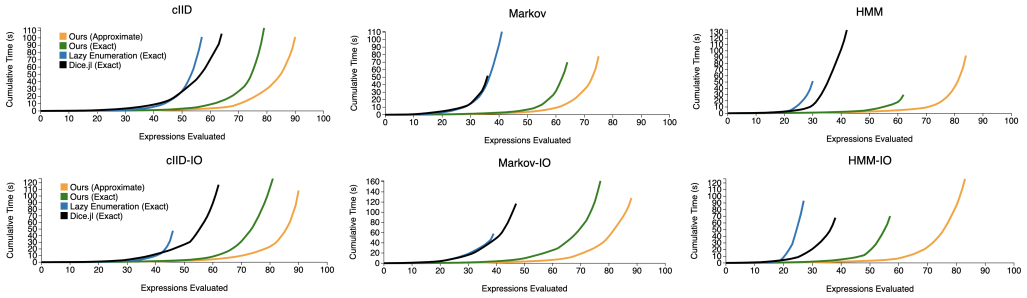


Fig. 6. **Per-domain results for evaluation on random programs**

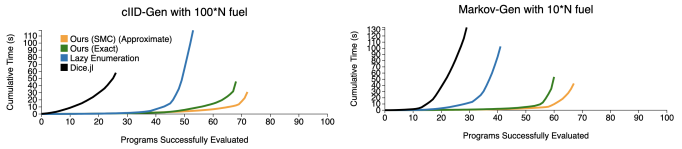


Fig. 7. **Scaling of testing of random programs with increasing fuel**. Fuel is set to (N+1)*100 where N is the maximum element in the output lists, or longest length of any given list

F.2 Recursion Bounds

Since geometrically distributed natural numbers are present in our grammar, even many of the simplest sampled programs will have an infinite set of possible executions under strict semantics, causing Dice.jl and strict enumeration to loop indefinitely. In order for Dice to terminate we replace all geometric distributions with bounded geometric distributions which return their maximum value when their recursion bound (*fuel*) is reached. Associating a fuel with a program changes the likelihoods it assigns to outputs – it both limits the size of the geometric, and puts all of cutoff probability mass onto the maximum value. These two effects can increase or decrease the likelihood of any given output.

The necessary fuel level for recovering the unbounded semantics will vary with the program and the particular input-output examples. In Section 6.1 we were able to manually determine tight bounds for the fuel in each benchmark, but this is not possible when sampling arbitrary programs from a grammar. Thus we set the fuel for each program to $2N + 1$, where N is twice the maximum number used in any of its example output lists or maximum length of any output list, whichever is larger. Preliminary experiments found that a bound of only N or slightly above N resulted in frequently distorted semantics.

While laziness obviates the need for fuel in many cases, some programs in our grammars also have infinitely many executions in lazy semantics. For example, subtracting a geometric from a geometric can produce a given constant in infinitely many ways. In such cases lazy methods will also loop indefinitely, and no bound will perfectly recover the unbounded semantics. To resolve this problem, we apply the aforementioned recursion bound to all methods.

F.3 Additional Results

G SYNTHESIS EVALUATION

For each program in the datasets generated in the robustness evaluation (details in Appx. F.1), we run 3 repetitions of stochastic search using 1000 steps of MCMC with a Metropolis-Hastings

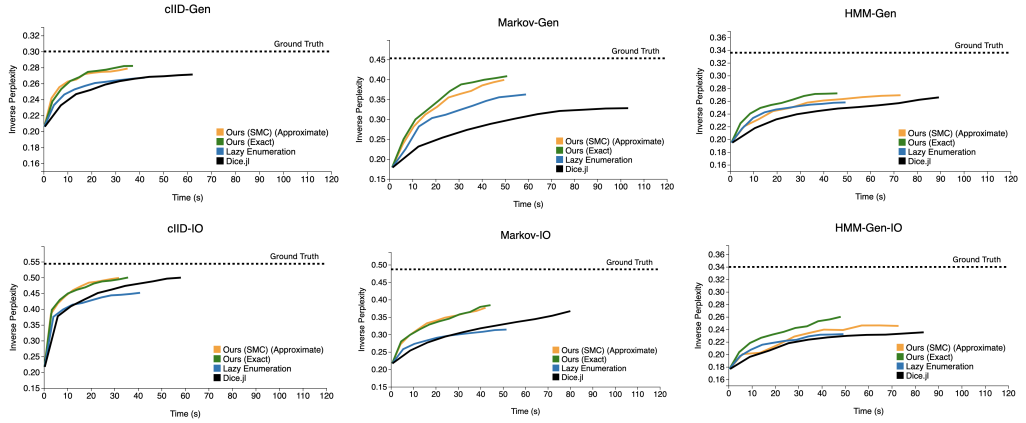


Fig. 8. **Per-domain inverse perplexity over time results**

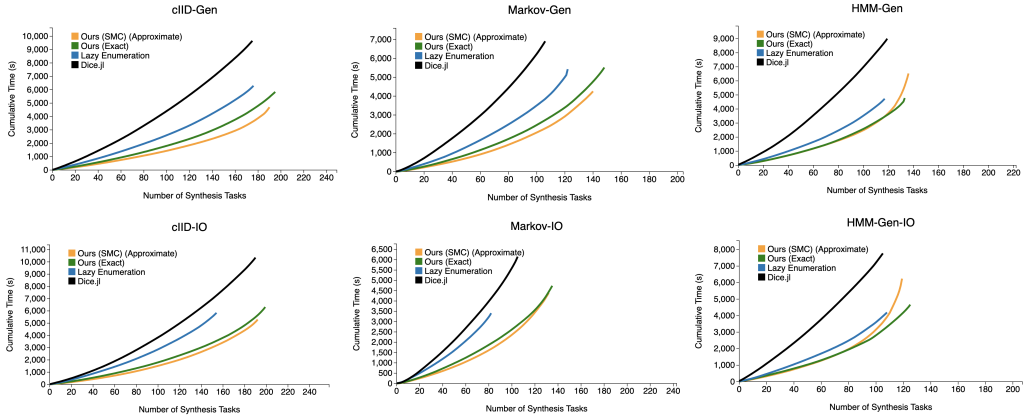


Fig. 9. **Per-domain cactus plots for inverse perplexity being within 10% of ground truth**

kernel. MCMC is run with an involutive subtree-regeneration proposal [23, 48] and the marginal likelihood of the input-output data under the proposed program is evaluated with the likelihood evaluation strategy being tested. All likelihood evaluators are run with a timeout of 200ms to evaluate 4 input-output examples, and timeouts give a likelihood of zero. The same fuel settings from Appx. F.2 are used.

To analyze the results, we track the likelihood of the current hypothesis every 100 steps. To ensure that the calculation of this likelihood isn't affected by the particular evaluator being used, we perform it with *all* exact evaluators with a 12 second timeout. To normalize for the varying output lengths, which can result in dramatically different scales of likelihoods, we calculate the perplexity of the output lists under each program. Since average perplexity can be dominated by single outliers, we instead use inverse perplexity when averaging over tasks. Average inverse perplexity plotted over the course of MCMC is shown in Figure 4 (Middle), and plots for each grammar are included in Appendix G. The dotted line shows the inverse perplexity of the ground truth program, averaged over the 77% of tasks for which exact ground truth likelihoods were obtained in the robustness

evaluation. Our exact and approximate approaches achieve higher inverse perplexities, and do so faster than lazy enumeration and strict knowledge compilation.

For MCMC runs with a ground truth inverse perplexity (77% of the 1800 runs), we construct a cactus plot in Figure 4 (Right) in which success is achieving an inverse perplexity that is smaller than the ground truth by 10% or less. Additional plots for each domain are given in Appx. G.