



Scaling Exact Inference for Discrete Probabilistic Programs

STEVEN HOLTZEN, University of California, Los Angeles, United States of America

GUY VAN DEN BROECK, University of California, Los Angeles, United States of America

TODD MILLSTEIN, University of California, Los Angeles, United States of America

Probabilistic programming languages (PPLs) are an expressive means of representing and reasoning about probabilistic models. The computational challenge of *probabilistic inference* remains the primary roadblock for applying PPLs in practice. Inference is fundamentally hard, so there is no one-size-fits all solution. In this work, we target scalable inference for an important class of probabilistic programs: those whose probability distributions are *discrete*. Discrete distributions are common in many fields, including text analysis, network verification, artificial intelligence, and graph analysis, but they prove to be challenging for existing PPLs.

We develop a domain-specific probabilistic programming language called Dice that features a new approach to exact discrete probabilistic program inference. Dice exploits program structure in order to *factorize* inference, enabling us to perform exact inference on probabilistic programs with hundreds of thousands of random variables. Our key technical contribution is a new reduction from discrete probabilistic programs to *weighted model counting* (WMC). This reduction separates the structure of the distribution from its parameters, enabling logical reasoning tools to exploit that structure for probabilistic inference. We (1) show how to compositionally reduce Dice inference to WMC, (2) prove this compilation correct with respect to a denotational semantics, (3) empirically demonstrate the performance benefits over prior approaches, and (4) analyze the types of structure that allow Dice to scale to large probabilistic programs.

CCS Concepts: • **Mathematics of computing** → **Probabilistic representations; Probabilistic inference problems.**

Additional Key Words and Phrases: Probabilistic programming

ACM Reference Format:

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (November 2020), 31 pages. <https://doi.org/10.1145/3428208>

1 INTRODUCTION

The primary analysis task in probabilistic programming languages is *probabilistic inference*, computing the probability that an event occurs according to the distribution defined by the program. Probabilistic inference generalizes many well-known program analysis tasks, such as reachability, and hence inference for a sufficiently expressive language is an extremely hard program analysis task. The key to scaling inference is to strategically make assumptions about the structure of programs and place restrictions on which programs can be written, while retaining a useful and expressive language.

In this paper, we focus on scaling inference for an important class of probabilistic programs: those whose probability distributions are *discrete*. Most PPLs today focus on handling continuous random

Authors' addresses: Steven Holtzen, University of California, Los Angeles, United States of America, sholtzen@cs.ucla.edu; Guy Van den Broeck, University of California, Los Angeles, United States of America, guyvdb@cs.ucla.edu; Todd Millstein, University of California, Los Angeles, United States of America, todd@cs.ucla.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/11-ART140

<https://doi.org/10.1145/3428208>

variables. In the continuous setting one usually desires approximate inference techniques, such as forms of sampling [Bingham et al. 2019; Carpenter et al. 2016; Chaganty et al. 2013; Dillon et al. 2017; Jordan et al. 1999; Kucukelbir et al. 2015; Nori et al. 2014; Wingate and Weber 2013]. However, handling continuous variables typically requires making strong assumptions about the structure of the program: many of these inference techniques have strict differentiability requirements that preclude their application to programs with discrete random variables. For instance, momentum-based sampling algorithms like HMC and NUTS [Hoffman and Gelman 2014] and many variational approximations [Kucukelbir et al. 2017] are restricted to continuous latent random variables and almost-everywhere differentiability of the posterior distribution. Yet many application domains are naturally discrete: for example mixture models, networks and graphs, ranking and voting, and text. This key deficiency in some of the most popular PPLs has led to a recent rise in interest in handling discreteness in probabilistic programs [Gorinova et al. 2020; Obermeyer et al. 2019; Zhou et al. 2020].

In this work we focus entirely on the challenge of designing a fast and efficient discrete probabilistic program inference engine. We describe Dice, a domain-specific language for representing discrete probabilistic programs, along with a new algorithm for exact inference for such programs. Dice extends a simple first-order, non-recursive functional language with support for making discrete probabilistic choices. It also provides first-class *observations*, which enables Dice to support Bayesian reasoning in the presence of evidence.

Discrete programs are not a new challenge, and there are existing PPLs that support exact inference for discrete probabilistic programs [Albarghouthi et al. 2017; Bingham et al. 2019; Claret et al. 2013; Gehr et al. 2016; Geldenhuys et al. 2012; Goodman and Stuhlmüller 2014; Narayanan et al. 2016; Pfeffer 2007b; Sankaranarayanan et al. 2013; Wang et al. 2018]. However, we identify several compelling example programs from text analysis, network verification, and discrete graphical models on which existing methods fail. The reason that they fail is that the existing methods do not find and automatically exploit the necessary factorizations and structure. Dice’s inference algorithm is inspired by techniques for exact inference on discrete graphical models, which leverage the graphical structure to factorize the inference computation. For example, a common property is *conditional independence*: if a variable z is conditionally independent of x given y , then y acts as a kind of *interface* between x and z that allows inference to be split into two separate analyses. This kind of structure abounds in typical probabilistic programs. For example, a function call is conditionally independent of the calling context given the actual argument value. Dice’s inference algorithm automatically identifies and exploits these independences in order to factorize inference. This enables Dice to scale to extremely large discrete probabilistic programs: our experiments in Section 5 show Dice performing exact inference on a real-world probabilistic program that is 1.9MB large.

At its core, Dice builds on the *knowledge compilation* approach to probabilistic inference [Chavira and Darwiche 2005, 2008; Chavira et al. 2006; Darwiche 2009; Fierens et al. 2015]. We show how to compile Dice programs to *weighted Boolean formulas* (WBF) and then perform exact inference via *weighted model counting* (WMC) on those formulas. We use binary decision diagrams (BDDs) to represent these formulas. We show that during compilation, these BDDs naturally identify and exploit conditional independence and other forms of structure, thereby avoiding exponential explosion for many classes of interesting programs. Further, BDDs support efficient WMC, linear in the size of the BDD.

Employing knowledge compilation for probabilistic inference in Dice requires us to generalize the prior approaches in several ways. First, in order to support logical compilation of traditional programming constructs such as conditionals, local variables, and arbitrarily nested tuples, we develop novel compilation rules that compositionally associate Dice programs with weighted

Boolean formulas. A key challenge here is supporting arbitrary observations. To do this, a Dice program, as well as each Dice function, is compiled to *two* BDDs. Intuitively, one BDD represents all possible executions of the program, ignoring observations, and the other BDD represents all executions that satisfy the program’s observations. We show how to use WMC on these formulas to perform exact Bayesian inference, with arbitrary observations throughout the program. Second, Dice compiles functions *modularly*: each function is compiled to a BDD once, and we exploit efficient BDD composition operations to reuse this BDD at each call site. This technique produces the same final BDD that would otherwise be produced, but it allows us to amortize the costly BDD construction phase across all callers, which we demonstrate can provide orders-of-magnitude speedups.

In sum, this paper presents the following technical contributions:

- We describe the Dice language and illustrate its utility through three motivating examples (Section 2).
- We formalize Dice’s semantics (Section 3) and its compilation to weighted Boolean formulas (Section 4). We prove that the compilation rules are correct with respect to the denotational semantics: the probability distribution represented by a compiled Dice program is equivalent to that of the original program.
- We empirically compare Dice’s performance to that of prior PPLs with exact inference (Section 5). We describe new and challenging benchmark probabilistic programs from cryptography, network analysis, and discrete Bayesian networks, and show that Dice scales to orders-of-magnitude larger programs than existing probabilistic programming languages, and is competitive with specialized Bayesian network inference engines on certain tasks.
- We analyze some of the benefits of Dice’s compilation strategy in Section 6. First we note that Dice inference is PSPACE-hard. Then we characterize cases where Dice scales efficiently, and which types of structure it exploits in the distribution. We illustrate where to find that structure in the program code as well as the compiled BDD form. We use these results to provide a technical comparison with prior exact inference algorithms.

Dice is available at <https://github.com/SHoltzen/dice>. The full version of this paper is available on arXiv as Holtzen et al. [2020], which contains full proofs.

2 AN OVERVIEW OF DICE

This section overviews the Dice language and its inference algorithm. First we use a simple example program to show how Dice exploits program structure to perform inference in a *factorized* manner. Then we use an example from network verification to show how Dice exploits the modular structure of functions. Finally we use a cryptanalysis example to illustrate how inference in Dice is augmented to support Bayesian inference in the presence of *evidence*.

2.1 Factorizing Inference

Probabilistic programming languages (PPLs) endow traditional programming languages with probabilistic operations that enable the construction of probability distributions [Albarghouthi et al. 2017; Borgström et al. 2011; Claret et al. 2013; Huang and Morrisett 2016; Kozen 1979], and Dice is no exception. Specifically, Dice extends a first-order functional language that supports non-recursive functions and a form of bounded iteration. Despite its simplicity, this language can express a wide variety of statistical models, and exact probabilistic inference in Dice is fundamentally hard. In addition to its standalone usage, we anticipate Dice being used as a core language for discrete inference inside other probabilistic programming systems.

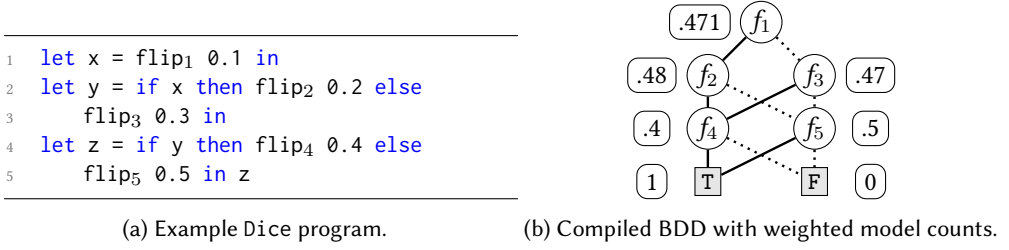


Fig. 1. Illustration of compiling a Dice program that exploits factorization.

We begin with a simple motivating example that highlights the challenge of performing inference efficiently and how Dice meets this challenge. Consider the example Dice program in Figure 1a. The syntax is standard except for the introduction of a probabilistic expression `flip θ` , which flips a coin that returns true with probability θ and false with probability $1 - \theta$. The subscript on each flip is not part of the syntax but rather used to refer to them uniquely in our discussion.

The goal of probabilistic inference is to produce a program's output probability distribution, so in Figure 1a we desire the probability that z is true and the probability that z is false. Consider computing the probability that z is true, which we denote $\Pr(z = T)$. The most straightforward way to compute this quantity is via *path enumeration*: we can consider all possible assignments to all flips and sum the probability of all assignments under which $z = T$. A number of existing PPLs directly implement path enumeration to perform inference [Albarghouthi et al. 2017; Filieri et al. 2013; Geldenhuys et al. 2012; Sankaranarayanan et al. 2013]. Concretely this would involve computing the following sum of products:

$$\underbrace{0.1}_{x=T} \cdot \underbrace{0.2}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.1}_{x=T} \cdot \underbrace{0.8}_{y=F} \cdot \underbrace{0.5}_{z=T} + \underbrace{0.9}_{x=F} \cdot \underbrace{0.3}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.9}_{x=F} \cdot \underbrace{0.7}_{y=F} \cdot \underbrace{0.5}_{z=T} \quad (1)$$

In this work we focus on the problem of scaling inference, so we ask: how does exhaustive enumeration scale as this program grows in size? In this case we grow the program by adding one additional layer to the chain of flips that depends on the previous. With this growing pattern, the number of terms that a path enumeration must explore grows exponentially in the number of layers, so clearly exhaustive enumeration does not scale on this simple example. Despite its apparent simplicity, many existing inference algorithms cannot scale to large instances of this example; see Figure 10d in Section 5.

However, the sum in Equation 1 has redundant computation, and thus can be factorized as

$$\underbrace{0.1}_{x=T} \cdot \left(\underbrace{0.2}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.8}_{y=F} \cdot \underbrace{0.5}_{z=T} \right) + \underbrace{0.9}_{x=F} \cdot \left(\underbrace{0.3}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.7}_{y=F} \cdot \underbrace{0.5}_{z=T} \right). \quad (2)$$

Such factorizations are abundant in this example, and in many others. Dice exploits these factorizations to scale, and in Section 5 we show that Dice scales to orders of magnitude larger programs than existing methods in part by exploiting these forms of factorization. Such factorizations are extremely common in probabilistic models, and finding and exploiting them is an essential strategy for scaling exact inference algorithms, for example for graphical models [Boutilier et al. 1996; Chavira and Darwiche 2008; Darwiche 2009; Koller and Friedman 2009; Pearl 1988].

Factorized inference in Dice. Inference in Dice is designed to find and exploit factorizations like the one shown above. The key insight is to separate the logical representation of the state space of the program from the probabilities, which allows Dice to identify factorizations implied by

the structure of the program that are otherwise difficult to detect. This separation is achieved by compiling each program to a *weighted Boolean formula*:

Definition 2.1. Let φ be a Boolean formula over variables X , let L be the set of all *literals* (assignments to variables) over X , and $w : L \rightarrow \mathbb{R}$ be a *weight function* that associates a real-valued weight with each literal L . The pair (φ, w) is a *weighted Boolean formula* (WBF).

To compile the program in Figure 1a into a WBF, we introduce one Boolean variable f_i for each expression $\text{flip}_i \theta$ in the program. Our goal is for the resulting boolean formula over these variables to represent all possible flip valuations that cause z to be true, so one choice of WBF is $\varphi_{ex} = f_1 f_2 f_4 \vee f_1 \bar{f}_2 f_5 \vee \bar{f}_1 f_3 f_4 \vee \bar{f}_1 \bar{f}_3 f_5$. Separately, the weight function represents the specific probabilities for each expression $\text{flip}_i \theta$ from the program: the weight of f_i is θ if f_i is true and $1 - \theta$ otherwise.

Once the program is associated with a WBF, we can perform probabilistic inference via a *weighted model count* (WMC). Formally, for a formula φ over variables X , a sentence ω is a *model* of φ if it is a conjunction of literals, contains every variable in X , and $\omega \models \varphi$. We denote the set of all models of φ as $\text{Mods}(\varphi)$. The *weight of a model*, denoted $w(\omega)$, is the product of the weights of each literal $w(\omega) \triangleq \prod_{l \in \omega} w(l)$. Then, the following defines the WMC task:

Definition 2.2. Let (φ, w) be a weighted Boolean formula. The *weighted model count* (WMC) of (φ, w) is the sum of the weights of each model, $\text{WMC}(\varphi, w) \triangleq \sum_{\omega \in \text{Mods}(\varphi)} w(\omega)$.

What has been achieved? So far, not much! The WMC task is known to be #P-hard for arbitrary Boolean formulas. Indeed, our formula φ_{ex} above is isomorphic to the structure of Equation 1, so the WMC calculation over it will be essentially equivalent. However, it has been observed in the AI literature that certain representations of Boolean formulas — such as binary decision diagrams (BDDs) — both exploit the structure of a formula to minimize its representation and support *linear time weighted model counting*, and as such are useful compilation targets [Bryant 1986; Chavira and Darwiche 2008; Darwiche and Marquis 2002]. The field of compiling Boolean formulas to representations that support tractable weighted model counting is broadly known as *knowledge compilation*, and *inference via knowledge compilation* is currently the state-of-the-art inference algorithm for certain kinds of discrete Bayesian networks [Chavira and Darwiche 2008] and probabilistic logic programs [Fierens et al. 2015].

Dice utilizes the insights of knowledge compilation to perform factorized inference. First, the generated formula φ in a compiled WBF is represented as a BDD; Figure 1b shows the compiled BDD for the program in Figure 1a. A solid edge denotes the case where the parent variable is true and a dotted edge denotes the case where the parent variable is false. This BDD is logically equivalent to φ_{ex} but the BDD's construction process exploits the program's conditional independence to efficiently produce a compact canonical representation. Specifically, there is a single subtree for f_4 , which is shared by both the path coming from f_2 and the path coming from f_3 , and similarly for f_5 . These shared sub-trees are induced by conditional independence: fixing y to the value true — and hence guaranteeing that a path to f_4 is taken in the BDD — screens off the effect of x on z , and hence reduces both the size of the final BDD and the cost of constructing it. The BDD automatically finds and exploits such factorization opportunities by caching and reusing repetitious logical sub-functions.

Dice performs inference on the original probabilistic program via WMC once the program is compiled to a BDD. Crucially, it does so without exhaustively enumerating all paths or models. By virtue of the shared sub-functions, the BDD in Figure 1b directly describes how to compute the WMC in the factorized manner. Observe that each node is annotated with the weighted model count, which is computed in linear time in a single bottom-up pass of the BDD. For instance, the WMC at

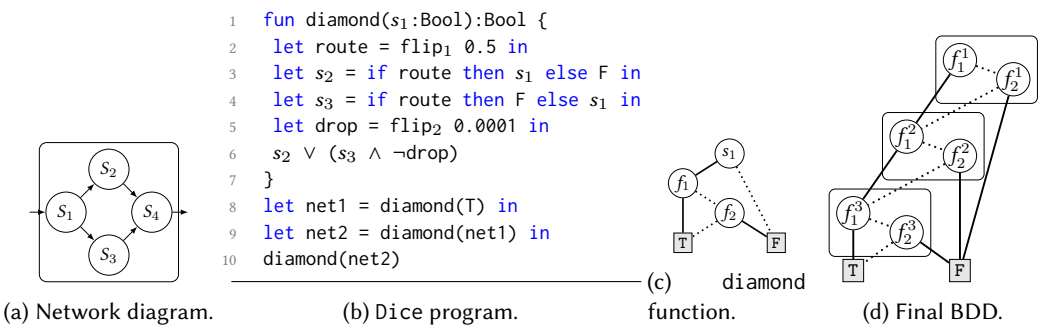


Fig. 2. A sub-network, its description as a probabilistic program, a compiled function, and the final BDD.

node f_2 is given by taking the weighted sum of the WMC of its children, $0.2 \times 0.4 + 0.8 \times 0.5$. Finally, the sum taken at the root of the BDD (the node f_1) is exactly the factorized sum in Equation 2.

2.2 Leveraging Functional Abstraction

The previous section highlights how Dice exploits factorization that comes from conditional independences in the program. One common source of such independences is functional abstraction: the behavior of a function call is independent of the calling context, given the actual argument. Dice inference as described above automatically exploits this structure as part of the BDD construction. In addition, Dice exploits functional abstraction in an orthogonal manner by modularly compiling a BDD for each function once and then reusing this BDD at each call site, thereby amortizing the cost of the BDD construction across all callers.

To illustrate the benefits of functional abstraction, we adapt an example from recent work in probabilistic verification of computer networks via probabilistic programs [Gehr et al. 2018]. Figure 2a shows a “diamond” network that contains four servers, labeled S_i . The network’s behavior is naturally probabilistic, to account for dynamics such as load balancing and congestion. In this case, server S_1 forwards an incoming packet to either S_2 or S_3 , each with probability 50%. In turn, those servers forward packets received from S_1 to S_4 , except that S_3 has a 0.1% chance of dropping such a packet. The diamond function in Figure 2b defines the behavior of this network as a probabilistic program in Dice. The argument boolean s_1 represents the existence of an incoming packet to S_1 from the left, and the function returns a boolean indicating whether a packet was delivered to S_4 .

As mentioned above, Dice compiles functions modularly, so Dice first compiles the diamond function to a BDD, shown in Figure 2c. The variable s_1 represents the unknown input to the function, and the f_i variables represent the flips in the function body, as in our previous example. Next Dice will create the BDD for the “main” expression in lines 8–10 of Figure 2b. During this process, the BDD for the diamond function is reused at each call site using standard BDD composition operations like conjunction (Section 4 describes this in more detail). The final BDD for the program is shown in Figure 2d, where each variable f_i^j represents the i th flip in the j th call to diamond.

The final BDD automatically identifies and exploits functional abstraction. For example, the structure of the BDD makes it clear that the third call to diamond depends only on the output of the second call to diamond, rather than the particular execution path taken to produce that output. As a result, even though there are three sub-networks, and therefore 2^6 possible joint assignments to flips, the BDD only has 8 nodes. More generally, this BDD will grow linearly in the number of composed diamond calls, though the number of possible executions grows exponentially. Hence

functional abstraction both produces smaller BDDs, which leads to faster WMC computation, and reduces BDD compilation time by compiling each function once. We show in Section 5 that these capabilities provide orders of magnitude speedups in inference.

2.3 Bayesian Inference & Observations

Bayesian inference is a general and popular technique for reasoning about the probability of events in the presence of *evidence*. Dice, similar to other PPLs, supports Bayesian reasoning through an observe expression. Specifically, the expression “observe *e*” represents evidence (or an *observation*) that *e* is true; the expression always evaluates to true, but it has the side effect that executions on which *e* is not true are defined to have 0 probability.

Dice supports first-class observations, including inside of functions. An example is shown in Figure 3, which shows another rich class of discrete probabilistic inference problems that come from *text analysis*. For this problem the goal is to decrypt a given piece of ciphertext by inferring the most likely encryption key. We assume that the plaintext was encrypted using a *Caesar cipher*, which simply shifts characters by a fixed but unknown constant, so the encryption key is an integer between 0 and 25 (e.g., with key 2, “abc” becomes “cde”).

The task of decrypting encrypted ciphertext can be cast as a probabilistic inference task by using *frequency analysis* [Katz et al. 1996]. In the English language each letter has a certain probability of being used: for instance, the frequency of letter “E” is 12.02%. In Figure 3, the function `EncryptChar` is a *generative model* for how each letter in the ciphertext was created. The function takes as an argument the encryption key as well as a received ciphertext character *c*. First a plaintext character `randomChar` is chosen according to its empirical distribution (the `ChooseChar` function is not shown but straightforward). Then this character is encrypted with the given key and we observe that the ciphertext is the actual ciphertext character *c* that we received. To make the inference problem more challenging and realistic, we assume that there is a chance that the encryptor mistakenly forgets to encrypt a character, in which case we do not perform the observation. Initially, the key (*k*) is assumed to be uniformly random (line 6). After invoking `EncryptChar` once for each received ciphertext character (lines 7–8), the posterior distribution on the key is returned.

The interaction of probabilistic inference with observations is subtle. Observations have a non-local and “backwards” effect on the probability distribution, which must be carefully preserved when performing inference. In our example, the observation inside of `EncryptChar` affects the posterior distribution of its argument *key*. These non-local effects are the bane of sampling-based inference algorithms: observations can impose complex constraints — such as the need in our example for `ChooseChar` to draw the right character — that make it challenging for sampling algorithms to find sufficiently many valid samples (we highlight this challenge in Section 5).

The WBF compilation strategy outlined in the previous section is inadequate for capturing the semantics of the `EncryptChar` function: this function always returns true, so its compiled BDD would be trivial. Clearly this is incorrect, since the `EncryptChar` function has an additional, implicit effect on the program, by making certain encryption keys more or less likely to be the correct one. To handle observations, we augment our compilation strategy to produce a second logical formula,

```

1 fun EncryptChar(key:int, c:char):Bool {
2   let randomChar = ChooseChar() in
3   let ciphertext = (randomChar+key)%26 in
4   let fail = flip 0.0001 in
5   if fail then true else
6     observe ciphertext == c
7 }
8 let k = UniformInt(0, 25) in
9 let _ = EncryptChar(k, 'H') in
10 ... // encrypt n total characters
11 in k

```

Fig. 3. A frequency analyzer for a noisy Caesar cipher.

```

1   $\tau ::= \text{Bool} \mid \tau_1 \times \tau_2$ 
2   $v ::= T \mid F \mid (v, v)$ 
3   $\text{aexp} ::= x \mid v$ 
4   $e ::= \text{aexp} \mid \text{fst aexp} \mid \text{snd aexp} \mid (\text{aexp}, \text{aexp}) \mid \text{let } x = e \text{ in } e \mid \text{flip } \theta$ 
5       $\mid \text{if aexp then } e \text{ else } e \mid \text{observe aexp} \mid f(\text{aexp})$ 
6   $\text{func} ::= \text{fun } f(x:\tau):\tau \{ e \}$ 
7   $p ::= e \mid \text{func } p$ 

```

Fig. 4. Syntax for the core Dice language. The metavariable f ranges over function names, x over variable names, and θ over real numbers in the range $[0, 1]$.

which we call the *accepting formula* and denote γ . The accepting formula represents all possible assignments to flips that cause all observes in the program to be satisfied. Together the formulas φ and γ capture the meaning of the program: we can compute the posterior distribution on k by computing weighted model counts of the form $\text{WMC}(\varphi \wedge \gamma, w) / \text{WMC}(\gamma, w)$ for each value of k . Note that γ serves two roles: it constrains φ to only those executions that satisfy the observations, and its weighted model count computes the normalizing constant for the final probability distribution.

3 THE DICE LANGUAGE

Dice is a first-order functional language augmented with constructs for probabilistic programming. This section describes the language formally, providing its syntax and compositional semantics.

3.1 Syntax

The core syntax of Dice is given in Figure 4. We enforce an A-normal form via the usage of atomic expressions (aexp) [Flanagan et al. 1993], which simplifies the semantics and compilation rules. A program is a sequence of functions followed by the "main" expression. Each function is non-recursive and can only call functions that precede it. The language supports booleans, tuples, and typical operations over those types. In addition to this core syntax our Dice implementation supports convenient syntactic sugar for logical operations (\wedge , \vee , and \neg), statically bounded loops, bounded-size integers, and arbitrary function arity, as we describe in Section 5.1. We utilize these extensions in our examples freely.

Dice supports two probabilistic expressions. First, the expression $\text{flip } \theta$, where θ is a real number between 0 and 1, denotes the distribution that has the value true with probability θ and false with probability $1 - \theta$. Second, the expression $\text{observe } e$ enables Bayesian reasoning by incorporating *evidence*. Specifically, $\text{observe } e$ represents the *observation* that e has the value true. Semantically, executions on which e does not have the value true are defined to have 0 probability, which has the effect of implicitly increasing the probabilities of other executions. We define the expression $\text{observe } e$ to always evaluate to true.

3.2 Semantics

The semantics of Dice programs is largely standard. We overview the semantics and highlight its key aspects and design choices. We begin with the semantics of Dice expressions, which are naturally represented as a probability distribution on values. Formally, let V be the set of all Dice values. Then, a *discrete probability distribution* on V is a function $\text{Pr} : V \rightarrow [0, 1]$ such that $\sum_{v \in V} \text{Pr}(v) = 1$.

Figure 5 provides the semantics for Dice expressions. First we will discuss the semantics of expressions without function calls and observations, which are deferred to Sections 3.2.1 and 3.2.2

$$\begin{aligned}
\llbracket v_1 \rrbracket (v) &\triangleq (\delta(v_1))(v) & \llbracket \text{fst } (v_1, v_2) \rrbracket (v) &\triangleq (\delta(v_1))(v) & \llbracket \text{snd } (v_1, v_2) \rrbracket (v) &\triangleq (\delta(v_2))(v) \\
\llbracket \text{if } v_g \text{ then } e_1 \text{ else } e_2 \rrbracket (v) &\triangleq \begin{cases} \llbracket e_1 \rrbracket (v) & \text{if } v_g = \text{T} \\ \llbracket e_2 \rrbracket (v) & \text{if } v_g = \text{F} \\ 0 & \text{otherwise} \end{cases} & \llbracket \text{flip } \theta \rrbracket (v) &\triangleq \begin{cases} \theta & \text{if } v = \text{T} \\ 1 - \theta & \text{if } v = \text{F} \\ 0 & \text{otherwise} \end{cases} \\
\llbracket \text{observe } v_1 \rrbracket (v) &\triangleq \begin{cases} 1 & \text{if } v_1 = \text{T and } v = \text{T}, \\ 0 & \text{otherwise} \end{cases} & \llbracket f(v_1) \rrbracket (v) &\triangleq (T(f))(v_1)(v) \\
\llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket (v) &\triangleq \sum_{v'} \llbracket e_1 \rrbracket (v') \times \llbracket e_2[x \mapsto v'] \rrbracket (v)
\end{aligned}$$

Fig. 5. Semantics for Dice expressions. The function $\delta(v)$ is a probability distribution that assigns a probability of 1 to the value v and 0 to all other values. The implicit context T maps function names to their semantics.

respectively. The *semantic function* $\llbracket \cdot \rrbracket$ maps expressions to unnormalized probability distributions (i.e., distributions that do not necessarily sum to 1). The semantics of values and tuple access are straightforward. For example, the semantics of the expression `fst (F, T)` is the probability distribution that assigns probability 1 to F and 0 to all other values. The semantics for conditionals follows from its usual semantics. In well-formed (i.e., closed) programs the conditional guard v_g is always a value, because the language uses A-normal form. Hence, the semantics of `if` selects either the *then*-branch or *else*-branch's semantics depending on the value of v_g . For completeness of the semantics, we define the semantics of `if` to be the always-zero function if the argument is not a Boolean.

The semantics for flips produces the corresponding Bernoulli distribution. For example, the expression `flip 0.8` denotes the distribution that assigns T the probability 0.8, F the probability 0.2, and all other values the probability 0.

The most interesting case in the semantics is for `let`, as it shows the path enumeration that is required when sequencing probabilistic expressions. Consider the example:

$$\text{let } x = \text{flip } 0.1 \text{ in flip } 0.4 \vee x \quad (\text{ExLET})$$

To compute the probability that (ExLET) results in some value v , we must consider all possible ways in which that value could result, based on all possible values v' for x . Concretely, to evaluate $\llbracket \text{let } x = \text{flip } 0.1 \text{ in flip } 0.4 \vee x \rrbracket (\text{T})$, the following sum is computed: $\llbracket \text{flip } 0.1 \rrbracket (\text{T}) \times \llbracket \text{flip } 0.4 \vee x[x \mapsto \text{T}] \rrbracket (\text{T}) + \llbracket \text{flip } 0.1 \rrbracket (\text{F}) \times \llbracket \text{flip } 0.4 \vee x[x \mapsto \text{F}] \rrbracket (\text{T}) = 0.1 \times 1.0 + 0.9 \times 0.4 = 0.46$.

3.2.1 Functions and Programs. Dice supports non-recursive functions. We generalize the semantics of expressions to functions in a natural way. Specifically, the semantics of a function f is a *conditional probability distribution*, which is a function from each value v to a probability distribution for $f(v)$. Formally, the semantics of a function $\llbracket \text{func} \rrbracket : V \rightarrow V \rightarrow [0, 1]$ is defined as follows:

$$\llbracket \text{fun } f(x : \tau) : \tau' \{e\} \rrbracket (v) \triangleq \llbracket e[x \mapsto v] \rrbracket \quad (3)$$

We can now give a semantics to function calls. To do so, we extend the semantics judgment to include a *function table* T , which is a finite map from function names to their conditional probability distributions. Formally our semantics judgment for expressions now has the form $\llbracket e \rrbracket^T : V \rightarrow [0, 1]$, and similarly for the semantics of function definitions above, but we leave T implicit when it is

clear from the context. Figure 5 provides the semantics of a function call: the function’s conditional probability distribution is found in T , and the probability distribution associated with the actual argument v is retrieved.

Finally, we define the semantics of programs $\llbracket p \rrbracket^T : V \rightarrow [0, 1]$. Intuitively, each function is given a semantics in the context of the prior functions, and then the semantics of the program is defined as the semantics of the “main” expression. We formalize this semantics inductively via the following two rules, where \bullet denotes the empty sequence and $\eta(\text{func})$ denotes the name of the function func :

$$\llbracket \bullet \ e \rrbracket^T \triangleq \llbracket e \rrbracket^T \quad \llbracket \text{func } p \rrbracket^T \triangleq \llbracket p \rrbracket^{T \cup \{\eta(\text{func}) \mapsto \llbracket \text{func} \rrbracket^T\}}. \quad (4)$$

3.2.2 Observations & Bayesian Conditioning. Observations complicate the goal of associating a probability distribution with each program expression. Our semantics of observe in Figure 5 follows prior work by assigning probability 0 to a failed observation [Borgström et al. 2011; Claret et al. 2013; Huang and Morrisett 2016; Kozen 1979; Nori et al. 2014]. Now consider the following example program:

```
let x = flip 0.6 in let y = flip 0.3 in let _ = observe x ∨ y in x  (ObsProg)
```

Because the observe expression is falsified when both x and y are false, that scenario has probability 0. Hence according to our semantics $\llbracket \text{ObsProg} \rrbracket(T) = 0.6$ and $\llbracket \text{ObsProg} \rrbracket(F) = 0.12$. As a result the meaning of this program is not a valid probability distribution.

The standard approach to handling this issue is to treat the semantics as producing an *unnormalized* distribution, which need not sum to 1 and which is normalized at the very end to produce a valid probability distribution for the entire program. Here we explore the subtle properties of this unnormalized distribution, which will serve a crucial purpose later during our compilation strategy. Let $\llbracket e \rrbracket_A$ denote the normalizing constant and $\llbracket e \rrbracket_D$ denote the normalized distribution for an expression. These two quantities can be straightforwardly computed from the unnormalized semantics in Figure 5:

$$\llbracket e \rrbracket_A \triangleq \sum_v \llbracket e \rrbracket(v), \quad \llbracket e \rrbracket_D(v) \triangleq \frac{1}{\llbracket e \rrbracket_A} \llbracket e \rrbracket(v). \quad (5)$$

For instance, in the above example $\llbracket \text{ObsProg} \rrbracket_A = 0.12 + 0.6 = 0.72$, $\llbracket \text{ObsProg} \rrbracket_D(T) = 0.6/0.72 \approx 0.83$, and $\llbracket \text{ObsProg} \rrbracket_D(F) = 0.12/0.72 \approx 0.17$. In the event that $\llbracket e \rrbracket_A = 0$, the distributional semantics is also defined to be zero.

By construction, $\llbracket \cdot \rrbracket_D$ always yields a probability distribution (or the always-zero function in the event that the accepting semantics is zero), so we call it the *distributional semantics*. This is the quantity that we need in order to answer inference queries. What does $\llbracket \cdot \rrbracket_A$ represent? Typically it is not given a meaning but rather simply considered to be an arbitrary normalizing constant that is only computed for the entire program. And indeed, the normalizing constant is irrelevant for the purposes of performing global inference: the probabilities in the unnormalized semantics can be scaled arbitrarily without changing $\llbracket \cdot \rrbracket_D$. This “normalize at the end” mode of operation is standard for many PPLs that use an unnormalized semantics [Claret et al. 2013; Fierens et al. 2015].

However, when reasoning about partial programs, the distributional semantics alone is not sufficient. For example, consider these two functions:

```
fun f(x:Bool):Bool { let y = x ∨ flip(0.5) in let z = observe y in y }  (6)
```

```
fun g(x:Bool):Bool { true }  (7)
```

Because the observation in f requires y to be true, the two functions have the identical distributional semantics: they both return true with probability 1, regardless of the argument x . However, these

two functions are not equivalent! Specifically, the observation in f has the effect of changing the probability distribution of the argument x when the function is called. Concretely,

$$\llbracket \text{let } x = \text{flip } 0.1 \text{ in let obs} = f(x) \text{ in } x \rrbracket_D (T) = 0.1/0.55$$

$$\llbracket \text{let } x = \text{flip } 0.1 \text{ in let obs} = g(x) \text{ in } x \rrbracket_D (T) = 0.1$$

The quantity $\llbracket \cdot \rrbracket_A$ carries exactly the information needed to distinguish these functions. Specifically, $\llbracket e \rrbracket_A$ represents the probability that e has an *accepting* execution, which satisfies all observations, so we call it the *accepting semantics*. In the above example, $\llbracket g(F) \rrbracket_A = 1$ but $\llbracket f(F) \rrbracket_A = 0.5$: the function call $f(F)$ will succeed only half of the time. This quantity allows us to precisely compute the effect of the observation on any caller.

In summary, the semantics in Figure 5 computes an unnormalized distribution. However, since the normalizing constant is exactly the accepting probability, the semantics has the effect of computing two key quantities on each program fragment, both of which are necessary to characterize its meaning: its normalized probability distribution and its probability of accepting. Later this accepting semantics will be explicitly represented during compilation as the accepting formula.

4 PROBABILISTIC INFERENCE FOR DICE

This section formalizes our approach to probabilistic inference in Dice via reduction to *weighted model counting* (WMC). In this style, a probabilistic model is compiled to a *weighted Boolean formula* (WBF) such that WMC queries on the WBF exactly correspond to inference queries on the original model. This approach has been successfully used to perform exact inference in discrete Bayesian networks as well as probabilistic databases and logic programs [Chavira and Darwiche 2008; Fierens et al. 2015; Van den Broeck and Suciú 2017]. However, to our knowledge it has not been previously applied to a probabilistic programming language with traditional programming language constructs, functions, and first-class observations.

The bulk of this section formalizes our novel algorithm for compiling Dice programs to WBF. We describe this compilation in stages: first on the Boolean sub-language, then with the addition of tuples, and finally with the addition of functions. We also state and prove a correctness theorem, which formally relates WMC queries over a program's compiled WBF to the semantics from the previous section. Finally we illustrate how we use BDDs to represent WBFs, which enables the approach to automatically perform factorized inference.

4.1 Compiling Boolean Dice Expressions

The formal compilation judgment for Boolean Dice expressions has the form $e \rightsquigarrow (\varphi, \gamma, w)$, where φ and γ are logical formulas and w is a weight function (recall Definition 2.1). This judgment form will be extended later to accommodate other language features. We call φ the *unnormalized formula*: it represents all possible assignments to variables and flips for which e evaluates to true, ignoring observations. We call γ the *accepting formula*: it represents all possible assignments to variables and flips that cause all observations in e to succeed. Before showing the formal rules, we present two examples to build intuition on the compilation to WBF and how it is used to perform inference.

Example 4.1. The expression (ExLet) from the previous section compiles to the unnormalized formula $\varphi = f_1 \vee f_2$, where f_1 and f_2 are Boolean variables associated with `flip 0.1` and `flip 0.4` respectively. Since there are no observations, $\gamma = T$ for this example. The weight function w assigns weights to the literals of f_1 and f_2 that correspond with their probabilities in (ExLet). Then we have that $\llbracket \text{ExLet} \rrbracket (T) = \text{WMC}(\varphi, w) = 0.46$ and $\llbracket \text{ExLet} \rrbracket (F) = \text{WMC}(\overline{\varphi}, w) = 0.54$.

Example 4.2. The program (ObsProg) from the previous section compiles to the unnormalized formula $\varphi = f_1$ and the accepting formula $\gamma = f_1 \vee f_2$, where f_1 corresponds with `flip 0.6` and f_2

$$\begin{array}{c}
\frac{}{T \rightsquigarrow (T, T, \emptyset)} \text{ (C-TRUE)} \quad \frac{}{F \rightsquigarrow (F, T, \emptyset)} \text{ (C-FALSE)} \quad \frac{}{x \rightsquigarrow (x, T, \emptyset)} \text{ (C-IDENT)} \\
\\
\frac{\text{fresh } f}{\text{flip } \theta \rightsquigarrow (f, T, (f \mapsto \theta, T, \bar{f} \mapsto 1 - \theta))} \text{ (C-FLIP)} \quad \frac{\text{aexp } \rightsquigarrow (\varphi, T, \emptyset)}{\text{observe aexp } \rightsquigarrow (T, \varphi, \emptyset)} \text{ (C-OBS)} \\
\\
\frac{\text{aexp } \rightsquigarrow (\varphi_g, T, \emptyset) \quad e_T \rightsquigarrow (\varphi_T, \gamma_T, w_T) \quad e_E \rightsquigarrow (\varphi_E, \gamma_E, w_E)}{\text{if aexp then } e_T \text{ else } e_E \rightsquigarrow \left(((\varphi_g \wedge \varphi_T) \vee ((\bar{\varphi}_g \wedge \varphi_E), ((\varphi_g \wedge \gamma_T) \vee ((\bar{\varphi}_g \wedge \gamma_E), w_T \cup w_E) \right)} \text{ (C-ITE)} \\
\\
\frac{e_1 \rightsquigarrow (\varphi_1, \gamma_1, w_1) \quad e_2 \rightsquigarrow (\varphi_2, \gamma_2, w_2)}{\text{let } x = e_1 \text{ in } e_2 \rightsquigarrow (\varphi_2[x \mapsto \varphi_1], \gamma_1 \wedge \gamma_2[x \mapsto \varphi_1], w_1 \cup w_2)} \text{ (C-LET)}
\end{array}$$

Fig. 6. Compiling Boolean expressions to WBFs.

with `flip 0.3`. Hence the formula $\varphi \wedge \gamma$ is true if and only if the program evaluates to T and satisfies all observations, and similarly $\bar{\varphi} \wedge \gamma$ is true if and only if the program evaluates to F and satisfies all observations. Then, with the appropriate weight function w , we can perform Bayesian inference on (**ObsProg**) via two weighted model counts: $\llbracket (\text{ObsProg}) \rrbracket_D(T) = \text{WMC}(\varphi \wedge \gamma, w) / \text{WMC}(\gamma, w) \approx 0.83$ and $\llbracket (\text{ObsProg}) \rrbracket_D(F) = \text{WMC}(\bar{\varphi} \wedge \gamma, w) / \text{WMC}(\gamma, w) \approx 0.17$.

The formal compilation rules are shown in Figure 6. The above examples show how *closed* programs are compiled, but expressions can also have free variables in them. The rule C-IDENT handles a free variable x simply by introducing a corresponding Boolean variable x . To illustrate the rule C-FLIP, `flip 0.4` $\rightsquigarrow (f, T, w)$ where w maps f to 0.4 and \bar{f} to 0.6, and f is a fresh Boolean variable. Hence $\text{WMC}(f \wedge T, w) = 0.4 = \llbracket \text{flip } 0.4 \rrbracket(T)$ and $\text{WMC}(\bar{f}, w) = 0.6 = \llbracket \text{flip } 0.4 \rrbracket(F)$.

The rule C-OBS handles observes. Since an expression's unnormalized formula ignores observations, the unnormalized formula for `observe aexp` is simply T. The metavariable `aexp` ranges over values and identifiers and hence compiles to an accepting formula of T and an empty weight function (`aexp` stands for *atomic expression*). Finally, the unnormalized formula of `aexp` becomes the accepting formula of `observe aexp`, in order to capture all ways that the observation is satisfied.

The rule C-ITE encodes the usual logical semantics of conditionals. Finally, the C-LET rule shows how to represent expression sequencing. The *logical substitution* $\varphi_1[x \mapsto \varphi_2]$ replaces all occurrences of x in φ_1 with the formula φ_2 . For the accepting formula, the expression `let $x = e_1$ in e_2` only accepts if both expressions accept, so we simply conjoin their accepting formulas. To illustrate the rule, we show the derivation through the rules for our example (**ExLET**), assuming the obvious rule for compiling logical disjunction (which is syntactic sugar for a conditional expression):

$$\frac{\frac{\text{fresh } f_1}{\text{flip } 0.1 \rightsquigarrow (f_1, T, w_1)} \quad \frac{\frac{\text{fresh } f_2}{x \rightsquigarrow (x, T, \emptyset)} \quad \text{flip } 0.4 \rightsquigarrow (f_2, T, w_2)}{\text{flip } 0.4 \vee x \rightsquigarrow (f_2 \vee x, T, w_2)}}{\text{let } x = \text{flip } 0.1 \text{ in flip } 0.4 \vee x \rightsquigarrow (f_2 \vee x[x \mapsto f_1], T, w_1 \cup w_2)} \text{ (ExLETCompilATION)}$$

This compilation matches Example 4.1 above and shows how logical substitution captures expression sequencing. The union of two weight functions, denoted $w_1 \cup w_2$, is simply the union of the two maps w_1 and w_2 ; this is well-defined because no two subexpressions can share flips, so there can be no conflicts.

The statement of correctness for Boolean Dice expressions connects our compilation rules to the formal semantics from the previous section:

LEMMA 4.3 (BOOLEAN EXPRESSION CORRECTNESS). *Let e be a Boolean Dice expression with free variables x_1, \dots, x_n and suppose $e \rightsquigarrow (\varphi, \gamma, w)$. Then for any Boolean values v_1, \dots, v_n :*

- $\llbracket e[x_i \mapsto v_i] \rrbracket_A = \text{WMC}(\gamma[x_i \mapsto v_i], w)$
- for any Boolean value v , $\llbracket e[x_i \mapsto v_i] \rrbracket_D(v) = \frac{\text{WMC}(((\varphi \Leftrightarrow v) \wedge \gamma)[x_i \mapsto v_i], w)}{\text{WMC}(\gamma[x_i \mapsto v_i], w)}$.

As in the earlier definition of the distributional semantics, in the event that a division by zero occurs in the above lemma, the result is defined to be zero. This lemma implies that we can answer inference queries on the original expression via two WMC queries on the compiled WBF. The following key lemma directly implies the one above:

LEMMA 4.4. *Let e be a Boolean Dice expression with free variables x_1, \dots, x_n and suppose $e \rightsquigarrow (\varphi, \gamma, w)$. Then for any Boolean values v_1, \dots, v_n and Boolean value v ,*

$$\llbracket e[x_i \mapsto v_i] \rrbracket(v) = \text{WMC}(((\varphi \Leftrightarrow v) \wedge \gamma)[x_i \mapsto v_i], w).$$

4.2 Tuples & Typed Compilation

Next we extend our compilation rules to support arbitrarily nested tuples. The primary purpose of tuples is to empower Dice functions by enabling multiple arguments and return values. Intuitively, this involves generalizing the compilation target from a single Boolean formula φ to tuples of Boolean formulas. Formally, this extension requires that we generalize the compilation judgment, which now has the following form:

$$\Gamma \vdash e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w).$$

First, our compilation is now *typed*: Γ is the usual type environment for free variables and τ is the type of e . The types are necessary to determine how to properly encode program variables in the compiled logical formulas. Second, compilation produces a collection of Boolean formulas, one per occurrence of the type `Bool` in τ . The new metavariable $\dot{\varphi}$ is defined inductively as either a Boolean formula φ or a pair of the form $(\dot{\varphi}_1, \dot{\varphi}_2)$.

As a concrete example of compiling a program that contains tuples:

$$\{\} \vdash \text{let } x = \text{flip } 0.2 \text{ in } (x, T) : \text{Bool} \times \text{Bool} \rightsquigarrow \left((f_1, T), T, [f_1 \mapsto 0.2, \bar{f}_1 \mapsto 0.8] \right).$$

Here, the resulting compiled formula $\dot{\varphi}$ is a pair of Boolean formulas (f_1, T) .

Figure 7 shows the new rules for compiling tuples and also presents updated versions of the rules from Figure 6, other than the Boolean-specific rules. The extended compilation for tuples is structurally very similar to Boolean compilation, but requires generalizing the Boolean operations in a natural way to accommodate tuples (The full version of the paper summarizes this new notation in the appendix). The new version of C-IDENT uses the *form function* $F_\tau(x)$, which constructs the logical representation of a variable x based on its type τ . It is defined inductively as $F_{\text{Bool}}(x) \triangleq x$ and $F_{\tau_1 \times \tau_2}(x) \triangleq (F_{\tau_1}(x_l), F_{\tau_2}(x_r))$. Note the subscripts x_l and x_r that lexically distinguish the left and right elements. This function also allows us to naturally define the compilation for tuple creation as well as `fst` and `snd` in Figure 7.

The C-ITE rule shows how we generalize the compilation of conditionals to accommodate tuples. The rule requires that we conjoin a Boolean expression φ_g (the compiled guard) with a potential tuple of formulas (the compiled then and else branches). To do this, we define *broadcasted conjunction*, denoted $\varphi_g \wedge_\tau \dot{\varphi}$, as conjoining φ_g with all the Boolean expressions in the tuple $\dot{\varphi}$. Formally, we define it as $\varphi_a \wedge_{\text{Bool}} \varphi_b \triangleq \varphi_a \wedge \varphi_b$ and $\varphi_a \wedge_{\tau_1 \times \tau_2} (\dot{\varphi}_{b1}, \dot{\varphi}_{b2}) \triangleq (\varphi_a \wedge_{\tau_1} \dot{\varphi}_{b1}, \varphi_a \wedge_{\tau_2} \dot{\varphi}_{b2})$. In addition to broadcasted

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow (F_\tau(x), T, \emptyset)} \text{ (C-IDENT)} \quad \frac{\Gamma(x_1) = \tau_1 \quad \Gamma(x_2) = \tau_2}{\Gamma \vdash (x_1, x_2) : \tau_1 \times \tau_2 \rightsquigarrow ((F_{\tau_1}(x_1), F_{\tau_2}(x_2)), T, \emptyset)} \text{ (C-TUP)} \\
\\
\frac{\Gamma(x) = \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } x : \tau_1 \rightsquigarrow (F_{\tau_1}(x), T, \emptyset)} \text{ (C-FST)} \quad \frac{\Gamma(x) = \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } x : \tau_2 \rightsquigarrow (F_{\tau_2}(x), T, \emptyset)} \text{ (C-SND)} \\
\\
\frac{\Gamma \vdash \text{aexp} : \text{Bool} \rightsquigarrow (\varphi_g, T, \emptyset) \quad \Gamma \vdash e_T : \tau \rightsquigarrow (\dot{\varphi}_T, \gamma_T, w_T) \quad \Gamma \vdash e_E : \tau \rightsquigarrow (\dot{\varphi}_E, \gamma_E, w_E)}{\Gamma \vdash \text{if aexp then } e_T \text{ else } e_E : \tau \rightsquigarrow \left(((\varphi_g \wedge \dot{\varphi}_T) \dot{\vee}_\tau ((\overline{\varphi}_g \wedge \dot{\varphi}_E)), ((\varphi_g \wedge \gamma_T) \vee ((\overline{\varphi}_g \wedge \gamma_E), w_T \cup w_E) \right)} \text{ (C-ITE)} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow (\dot{\varphi}_1, \gamma_1, w_1) \quad \Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2 \rightsquigarrow (\dot{\varphi}_2, \gamma_2, w_2)}{\Gamma \vdash \text{let } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2 \rightsquigarrow (\dot{\varphi}_2[x \xrightarrow{\tau} \dot{\varphi}_1], \gamma_1 \wedge \gamma_2[x \xrightarrow{\tau} \dot{\varphi}_1], w_1 \cup w_2)} \text{ (C-LET)}
\end{array}$$

Fig. 7. Typed compilation for tuples. These assume, without loss of generality but for simplicity, that fst, snd, and tuple construction are only ever performed with identifiers as arguments.

conjunction, C-ITE also requires *point-wise disjunction*, denoted $\dot{\varphi}_1 \dot{\vee}_\tau \dot{\varphi}_2$. Point-wise disjunction is defined inductively as $\varphi_1 \dot{\vee}_{\text{Bool}} \varphi_2 \triangleq \varphi_1 \vee \varphi_2$ and $(\dot{\varphi}_{11}, \dot{\varphi}_{12}) \dot{\vee}_{\tau_1 \times \tau_2} (\dot{\varphi}_{21}, \dot{\varphi}_{22}) \triangleq (\dot{\varphi}_{11} \dot{\vee}_{\tau_1} \dot{\varphi}_{21}, \dot{\varphi}_{12} \dot{\vee}_{\tau_2} \dot{\varphi}_{22})$.

Finally, to generalize the compilation of let expressions, in the C-LET rule we employ *typed substitution* $\dot{\varphi}_2[x \xrightarrow{\tau} \dot{\varphi}_1]$ to substitute the compiled version of e_1 into the compiled version of e_2 . We define typed substitution inductively as follows:

$$\begin{aligned}
\varphi_2[x \xrightarrow{\text{Bool}} \varphi_1] &\triangleq \varphi_2[x \mapsto \varphi_1], \quad \varphi_2[x \xrightarrow{\tau_a \times \tau_b} (\dot{\varphi}_a, \dot{\varphi}_b)] \triangleq \varphi_2[x_l \xrightarrow{\tau_a} \dot{\varphi}_a][x_r \xrightarrow{\tau_b} \dot{\varphi}_b], \\
(\dot{\varphi}_1, \dot{\varphi}_2)[x \xrightarrow{\tau} \dot{\varphi}] &\triangleq (\dot{\varphi}_1[x \xrightarrow{\tau} \dot{\varphi}], \dot{\varphi}_2[x \xrightarrow{\tau} \dot{\varphi}]).
\end{aligned}$$

We can state and prove a natural generalization of our key lemma from the previous subsection, Lemma 4.4. The lemma depends on *pointwise iff*, denoted $\dot{\varphi}_1 \xleftrightarrow{\tau} \dot{\varphi}_2$ and defined inductively as follows: $\varphi_1 \xleftrightarrow{\text{Bool}} \varphi_2 \triangleq \varphi_1 \Leftrightarrow \varphi_2$ and $(\dot{\varphi}_1, \dot{\varphi}_2) \xleftrightarrow{\tau_1 \times \tau_2} (\dot{\varphi}'_1, \dot{\varphi}'_2) \triangleq (\dot{\varphi}_1 \xleftrightarrow{\tau_1} \dot{\varphi}'_1) \wedge (\dot{\varphi}_2 \xleftrightarrow{\tau_2} \dot{\varphi}'_2)$. Then we can state the following lemma:

LEMMA 4.5 (TYPED CORRECTNESS WITHOUT FUNCTIONS). *Let e be a Dice expression without function calls, and suppose $\{x_i : \tau_i\} \vdash e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$. Then for any values $\{v_i : \tau_i\}$ and $v : \tau$, we have that $\llbracket e[x_i \mapsto v_i] \rrbracket(v) = \text{WMC}\left(\left((\dot{\varphi} \xleftrightarrow{\tau} v) \wedge \gamma\right)[x_i \xrightarrow{\tau_i} v_i], w\right)$.*

4.3 Functions & Programs

We conclude our development of Dice compilation by introducing functions and programs in Figure 8. We introduce a new piece of context Φ into our judgment, which maps function names to their compiled function bodies. Function names are mapped to a 4-tuple $(x_{arg}, \dot{\varphi}, \gamma, w)$ where x_{arg} is the logical variable for the function's formal argument and the other items are respectively the function body's compiled unnormalized formula, accepting formula, and weight function.

The judgment $\Gamma, \Phi \vdash \text{func} \rightsquigarrow (\dot{\varphi}, \gamma, w)$ compiles function definitions. As shown in C-FUNC, we simply compile the function's body in an appropriate type environment. The judgment $\Gamma, \Phi \vdash p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$ compiles programs by compiling each function in order, followed by the “main” expression. The rules C-PROG1 and C-PROG2 perform this compilation. After each function is

$$\begin{array}{c}
\frac{\Gamma \cup \{x_1 : \tau_1\}, \Phi \vdash e : \tau_2 \rightsquigarrow (\dot{\varphi}, \gamma, w)}{\Gamma, \Phi \vdash \text{fun } f(x_1 : \tau_1) : \tau_2 \{e\} \rightsquigarrow (\dot{\varphi}, \gamma, w)} \text{ (C-FUNC)} \quad \frac{\Gamma, \Phi \vdash e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)}{\Gamma, \Phi \vdash \bullet e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)} \text{ (C-PROG1)} \\
\\
\frac{\Gamma, \Phi \vdash \text{fun } f(x_1 : \tau_1) : \tau_2 \{e\} \rightsquigarrow (\dot{\varphi}_f, \gamma_f, w_f) \quad \Gamma \cup \{f \mapsto \tau_1 \rightarrow \tau_2\}, \Phi \cup \{f \mapsto (x_1, \dot{\varphi}_f, \gamma_f, w_f)\} \vdash p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)}{\Gamma, \Phi \vdash \text{fun } f(x_1 : \tau_1) : \tau_2 \{e\} p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)} \text{ (C-PROG2)} \\
\\
\frac{\Gamma(f) = \tau_1 \rightarrow \tau_2 \quad \Gamma(x_1) = \tau_1 \quad \Phi(f) = (x_{arg}, \dot{\varphi}, \gamma, w) \quad (\dot{\varphi}', \gamma', w') = \text{RefreshFlips}(x_{arg}, \dot{\varphi}, \gamma, w)}{\Gamma, \Phi \vdash f(x_1) : \tau_2 \rightsquigarrow (\dot{\varphi}'[x_{arg} \xrightarrow{\tau_1} x_1], \gamma'[x_{arg} \xrightarrow{\tau_1} x_1], w')} \text{ (C-FUNCALL)}
\end{array}$$

Fig. 8. Compiling functions and programs. These assume without loss of generality but for simplicity that function calls are only ever given identifiers as arguments.

compiled, its compiled WBF is added to Φ and its name and type are added to Γ , for use in subsequent compilation.

The final judgment form for expressions is $\Gamma, \Phi \vdash e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$, and C-FUNCALL shows the rule for compiling function calls. The rule simply looks up the function's compiled WBF and substitutes the actual argument for the formal argument. One subtlety is that we must ensure that the flips in each call to a function are independent of one another. Our compilation approach makes it straightforward to do so: simply replace all of the variables in $\dot{\varphi}$ and γ , aside from the formal argument x_{arg} , with fresh variables. We use an auxiliary function $\text{RefreshFlips}(x_{arg}, \dot{\varphi}, \gamma, w)$ for this purpose. We now state the full correctness theorem for Dice compilation:

THEOREM 4.6 (COMPILATION CORRECTNESS). *Let p be a Dice program and $\emptyset, \emptyset \vdash p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$. Then: (1) $\llbracket p \rrbracket_A = \text{WMC}(\gamma, w)$, and (2) for any value $v : \tau$, $\llbracket p \rrbracket_D(v) = \text{WMC}((\dot{\varphi} \xrightarrow{\tau} v) \wedge \gamma, w) / \text{WMC}(\gamma, w)$.*

As before, division by zero is defined to be zero, and we prove this theorem as a corollary of the following stronger property:

THEOREM 4.7 (TYPED PROGRAM CORRECTNESS). *Let p be a Dice program $\emptyset, \emptyset \vdash p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$. Then for any $v : \tau$, we have that $\llbracket p \rrbracket(v) = \text{WMC}((\dot{\varphi} \xrightarrow{\tau} v) \wedge \gamma, w)$.*

4.4 Binary Decision Diagrams as WBF

Weighted model counting on WBFs is still #P-hard, so our compilation above is not necessarily advantageous. Now we reap the benefits of this translation by representing WBF with binary decision diagrams (BDDs), a data structure that facilitates efficient inference by exploiting the program structure to minimize the size of the WBF. A BDD is a popular data structure for representing Boolean formulas, and there is a rich literature of using BDDs to represent the state space of non-probabilistic programs during model checking [Clarke et al. 1999; Jhala and Majumdar 2009].

The compilation rules in the previous subsections were deliberately designed to facilitate BDD compilation. Consider the example compilation (EXLETCOMPILATION) from Section 4.1. Each step in this derivation can be translated into a corresponding BDD operation, as illustrated by the *BDD derivation tree* in Figure 9. The final BDD is compiled compositionally, at each step exploiting program structure to produce a minimal, canonical representation (for the given variable ordering). The operations necessary for constructing this derivation tree — BDD conjunction, disjunction, and

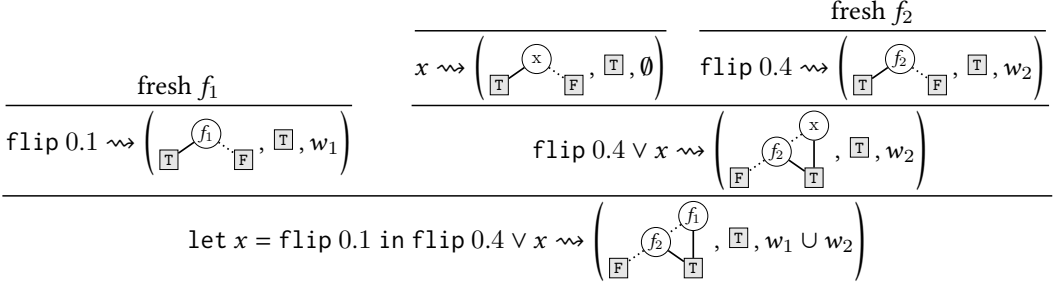


Fig. 9. A BDD derivation tree for (ExLETCompILATION).

substitution — are all standard operations that are available in BDD packages such as CUDD [Somenzi [n.d.]].

The cost of Dice inference is dominated by the cost of constructing the corresponding BDD derivation tree: that step is computationally hard in general, while WMC on the final BDD is linear time in the size of the BDD. However, BDDs can exploit program structure in order to allow compilation to scale efficiently on many examples. The remainder of this paper is devoted to showing that the BDD can be efficient to construct for useful programs. In Section 5 we show this experimentally, and Section 6 characterizes the hardness of Dice inference.

5 DICE IMPLEMENTATION & EMPIRICAL EVALUATION

Now we describe our implementation and empirical evaluation of Dice. Dice is implemented in OCaml and uses CUDD as its backend for compiling BDDs [Somenzi [n.d.]]. First we describe extensions to the core Dice syntax that make programming more ergonomic and enable us to more easily implement some of the benchmark programs. Then we describe our empirical evaluation of Dice’s performance in comparison with prior PPLs on a suite of benchmarks. In Section 6 we give context to these experiments and discuss why Dice succeeds on many benchmarks where others fail.

5.1 Dice Extensions & Ergonomics

Our implementation extends the core Dice syntax from Figure 4 in several ways. We relax the constraint on A-normal form, allowing more arbitrary placement of expressions. We also include syntactic sugar for the usual Boolean operators \wedge , \vee and \neg . Finally, we include support for bounded integers and bounded iteration, both of which are described in more detail next. Further details of our implementation can be found in the full version of this paper.

5.1.1 Bounded Integers. Dice supports probability distributions over integers with the discrete keyword; for instance, the expression `discrete(0.1, 0.4, 0.5)` defines a discrete distribution over $\{0, 1, 2\}$ where 0 has probability 0.1, 1 has probability 0.4, and 2 has probability 0.5. There are a number of possible strategies for encoding integers into a WBF. The simplest — and the one we implemented — is a *one-hot encoding*. Specifically, a distribution over n integers is represented as tuple of n Boolean variables, each representing one integer value, and flips are used to ensure that each variable is true with the specified probability. For example, here is the encoding of our

example distribution above:

$$\text{discrete}(0.1, 0.4, 0.5) \rightsquigarrow \begin{cases} \text{let } v0 = \text{flip}(0.1) \text{ in} \\ \text{let } v1 = \neg v0 \wedge \text{flip}(0.4/(0.4 + 0.5)) \text{ in} \\ \text{let } v2 = \neg v0 \wedge \neg v1 \text{ in } (v0, (v1, v2)) \end{cases}$$

Formally, for a discrete distribution $\text{discrete}(\theta_1, \theta_2, \dots, \theta_n)$, the encoded value v_i is true only if (1) $\bigwedge_{k < i} \neg v_k$ holds and (2) a coin flipped with probability $\theta_i / \sum_{j \geq i} \theta_j$ is true. Dice also supports the standard modular arithmetic operations like (+) and (\times) on integers.

5.1.2 Statically Bounded Iteration. Iteration and loops are challenging program constructs to support in PPLs. Dice, like many other PPLs, supports *bounded iteration*: loops that always terminate after a finite number of iterations [Claret et al. 2013; Cusumano-Towner et al. 2018; Gehr et al. 2016; Goodman and Stuhlmüller 2014; Pfeffer 2007b]. It does so via the syntax `iterate(f, init, k)`, where `f` is a function name, `init` is an initialization expression, and `k` is an integer indicating the number of times to call `f`:

$$\text{iterate}(f, \text{init}, k) \rightsquigarrow \underbrace{f(f(\dots f(\text{init})))}_{k \text{ times}}$$

Many useful examples — such as the network reachability example from Section 2 — can be expressed as bounded iteration.

5.2 Empirical Performance Evaluation

We have faithfully implemented the compilation strategy and use of BDDs as described in Section 4. Section 2 highlights some program structure that BDD compilation exploits, and Section 6 explores this structure further, but the question remains: does this structure exist in practice, and can Dice effectively exploit it? We investigate these questions from three angles:

- Q1: Comparison with Existing PPLs** How quickly can Dice perform exact inference on benchmark probabilistic programs from the literature? We evaluate this question in Section 5.2.1.
- Q2: Exploiting Functions** What are the performance benefits of modular compilation for functions? We evaluate this question in Section 5.2.2 by comparing Dice’s performance with and without inlining function calls.
- Q3: Comparison with Bayesian Network Solvers** Discrete Bayesian networks are a special case of Dice programs and are a good source of challenging and realistic inference problems. A natural question here is: how does Dice compare against state-of-the-art Bayesian network solvers that are specialized for this class of programs? In Section 5.2.3 we compare Dice against Ace [Chavira and Darwiche 2008], a state-of-the-art discrete Bayesian network solver.

In our evaluation we compare Dice against state-of-the-art PPLs that employ two different classes of exact inference algorithms:

Algebraic Methods The first class are *algebraic* inference methods that represent the probability distribution as a symbolic expression or algebraic decision diagram (ADD) [Claret et al. 2013; Dehnert et al. 2017; Gehr et al. 2016; Narayanan et al. 2016]. We discuss this class of inference algorithms more thoroughly in Section 6.3. In this class, we compare experimentally against PSI [Gehr et al. 2016].¹

Enumerative Methods The second class of inference methods work by exhaustively *enumerating* all paths through the probabilistic program, possibly using dynamic programming to reduce the search space [Albarghouthi et al. 2017; Chistikov et al. 2015; Filieri et al. 2013; Geldenhuys

¹We used PSI version 2d21f9fe04cf3aac533e08ccc2df18179947baad

Table 1. *Baselines*. Comparison of inference algorithms (times are milliseconds). The total time for Dice is reported under the “Dice” column, and the total size of the final compiled BDD is reported in the “BDD Size” column.

Benchmark	Psi (ms)	DP (ms)	Dice (ms)	# Paths	BDD Size
Grass	167	57	14	95	15
Burglar Alarm	98	10	13	250	11
Coin Bias	94	23	13	4	13
Noisy Or	81	152	13	1640	35
Evidence1	48	32	13	9	5
Evidence2	59	28	13	9	6
Murder Mystery	193	75	10	16	6

et al. 2012; Goodman and Stuhlmüller 2014; Sankaranarayanan et al. 2013; Wingate and Weber 2013]. Both PSI and WEBPPL [Goodman and Stuhlmüller 2014] have a mode that supports dynamic-programming exact inference, and we compare against them experimentally.

Comparing the performance of probabilistic program inference is challenging because performance is closely tied to the intricacies of how the program is structured: semantically equivalent programs may have vastly differing performance. Throughout our experiments we made a best-effort attempt at representing the programs in a way that was maximally performant in each language. The tables in this section report the mean value over at least 5 runs for each experiment; the appendix contains expanded tables that include standard deviations for each result. All experiments were single-threaded and performed on the same server with a 2.66GHz CPU and 512GB of RAM. The timings were recorded using *hyperfine*,² a utility that performs statistical timing analysis of Unix shell commands.

5.2.1 Baselines. Table 1 summarizes the results of our performance experiments on well-known baselines, which includes all of the discrete programs that PSI and R2 were evaluated on [Borgström et al. 2011; Gehr et al. 2016; Nori et al. 2014].³ Each row is a different benchmark. The “Psi”, “DP”, and “Dice” columns give the amount of time (in milliseconds) for respectively (1) Psi’s default inference algorithm [Gehr et al. 2016], (2) Psi’s dynamic programming inference algorithm that is specialized for finite discrete programs, and (3) the total time for Dice to compile a BDD and perform weighted model counting. These examples are small and thus relatively easy for exact inference, but they serve as an important sanity check. Generally these examples are too trivial to differentiate the performance of Dice and Psi.

We include two other columns, “# Paths” and “BDD Size”, that give a proxy for how hard each inference problem is. The “# Paths” column gives how many paths would be explored by a path enumeration algorithm. The “BDD Size” gives the final compiled BDD generated by Dice, which in conjunction with the “# Paths” column gives a metric for how much structure Dice is exploiting.

5.2.2 Modular Compilation. We return to the motivating examples from Section 2 to see how Dice compares with existing methods, and against a version of itself where all function calls are inlined. Figure 10 shows how different algorithms scale as the size of the problem grows (note that all plots are in log-log scale). Figure 10d was introduced and discussed in Section 2.

²<https://github.com/sharkdp/hyperfine>

³One discrete benchmark, “Digit Recognition”, was omitted from this table because the version of Psi that we tested with does not support this program.

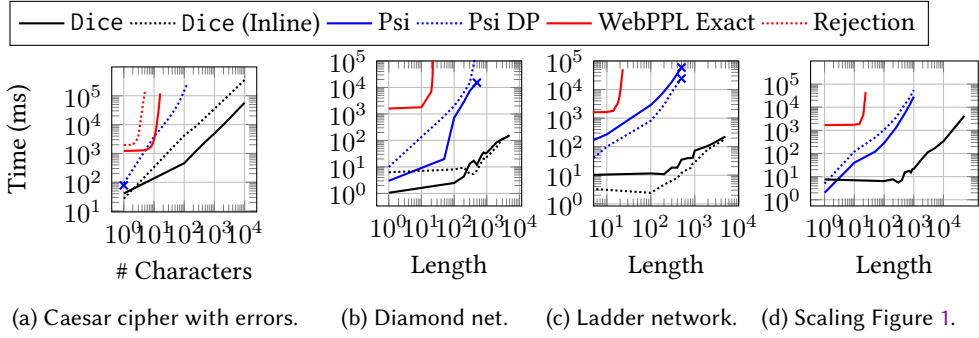


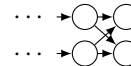
Fig. 10. Log-log scaling plots illustrating the benefits of separate compilation of functions. An “x”-mark denotes a runtime error was encountered at that point. The time reported for Dice inference includes the time required to compile and perform WMC. The standard deviation for the run-times are negligible.

Encryption. Figure 3 introduced the Caesar cipher motivating example, and Figure 10a shows how exact inference on this example scales as the number of characters being encrypted increases. Dice is about an order of magnitude faster than the case when function calls are inlined, and multiple orders of magnitude faster than WebPPL and Psi. In particular, Psi’s default algebraic inference fails to handle the encryption of even a single character; we explore why in Section 6.3.

Approximate inference approaches generally struggle with these kinds of programs, due to the low probability of finding samples that satisfy the observations. To illustrate this, we also report the time it took for rejection sampling to draw 10 accepted samples. WebPPL supports rejection sampling, and Figure 10a shows how it scales for this particular example program. This figure shows that rejection sampling scales exponentially in this case, and thus is not a feasible route around the state-space explosion problem.

Network Reachability. Next we examine how separate compilation helps in the network reachability task described in Figure 2. Figure 10b shows how exact inference scales in the number of diamond subnetworks. We see a modest benefit over inlining: compiling the diamond function multiple times is not very expensive since it is so small. Note that modular function compilation is not strictly beneficial: for this example, the inlined version is faster than the modular version after about 10^2 iterations. Also note that both versions of Dice are multiple orders of magnitude faster than Psi and WebPPL due to the exponential number of paths.

We expect to see overall linear scaling of Dice for many network topologies due to conditional independence. To evaluate this, Figure 10c shows a version where instead of diamonds we use a

ladder network of the following structure: . The goal is to determine the probability of a packet reaching the end of a network that consists of a chain of ladder subnetworks where each has a similar probabilistic routing policy to the diamond network. Dice continues to scale well, while this example is challenging for the other methods, in part since the number of paths is exponential in the length of the network.

5.2.3 Discrete Bayesian Networks. There is currently a lack of challenging discrete probabilistic program benchmarks in the literature. To more rigorously establish the relative performance of Dice and existing algorithms, here we evaluate the performance of Dice on discrete Bayesian networks that we translated into equivalent Psi and Dice programs. These benchmarks were selected from

Table 2. *Single Marginal Inference*. Comparison of inference algorithms (times are milliseconds). A “X” denotes a timeout at 2 hours of running. The total time for Dice is reported under the “Dice” column, and the total size of the final compiled BDD is reported in the “BDD Size” column.

Benchmark	Psi (ms)	DP (ms)	Dice (ms)	# Parameters	# Paths	BDD Size
Cancer	772	46	13	10	1.1×10^3	28
Survey	2477	152	13	21	1.3×10^4	73
Alarm	X	X	25	509	1.0×10^{36}	1.3×10^3
Insurance	X	X	212	984	1.2×10^{40}	1.0×10^5
Hepar2	X	X	54	48	2.9×10^{69}	1.3×10^3
Hailfinder	X	X	618	2656	2.0×10^{76}	6.5×10^4
Pigs	X	X	72	5618	7.3×10^{492}	35
Water	X	X	2590	1.0×10^4	3.2×10^{54}	5.1×10^4
Munin	X	X	1866	8.1×10^5	2.1×10^{1622}	1.1×10^4

the Bayesian Network Repository, an online repository of well-known Bayesian networks.⁴ These programs are (1) *realistic*: each has been used to answer scientific research questions in various domains such as medical diagnosis, weather modeling, and insurance modeling; and (2) *challenging*: many of these examples have on the order of thousands or tens of thousands of random variables.

First, we will compare the performance of Dice and Psi on this task; then we compare Dice against a specialized Bayesian network tool. We will show that Dice significantly outperforms Psi on all of these examples and is competitive with the specialized Bayesian network solver.

Comparison with Psi. Table 2 compares Psi against Dice on the task of computing a *single marginal* of a leaf node of a Bayesian network, a standard Bayesian network query. As an example of this task, Figure 11 shows the “Cancer” Bayesian network [Korb and Nicholson 2010], a simple 5-node network for modeling the probability that a patient has cancer (the © node) given a collection of symptoms (⊗ and ⊕) and causes (Ⓟ and Ⓢ). The single-marginal task for this example is to compute the marginal probability of the leaf node $\Pr(\otimes)$.

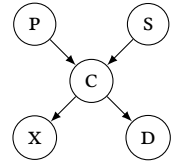


Fig. 11. The “Cancer” Bayesian network.

Table 2 compares the performance of Dice and Psi on the single-marginal inference task for a variety of Bayesian networks. The size of the network — a proxy for the difficulty of the inference task — is given by the number of parameters (the “# Parameters” column in the table). Psi fails to complete the inference within the allotted two hours on any of the medium or larger sized Bayesian networks.

Comparison with a Bayesian Network Solver. As a final test of the Dice’s performance, in Table 3 we compare against Ace, a state-of-the-art Bayesian network solver [Chavira and Darwiche 2008]. The task here is to compute *all marginal probabilities*, a strictly harder task than the single-marginal task considered earlier. We note that Psi fails to complete even a single marginal inference task on any of these examples within 2 hours, so it is omitted from this table.

Part of what makes the all-marginals inference task challenging is that it requires the computation of many queries: one for each node in the Bayesian network. One of the benefits of our compilation is that a single (potentially expensive) compilation, once completed, can be efficiently reused to perform many marginal probability queries. We highlight this capability in Table 3, which

⁴<https://www.bnlearn.com/bnrepository/>

Table 3. *All marginals*. A comparison between Dice and Ace on the all-marginal discrete Bayesian network inference task.

Benchmark	Dice (ms)	Ace (ms)	BDD Size
Alarm	159	422	4.3×10^5
Hailfinder	1280	522	2.1×10^5
Insurance	222	492	2.3×10^5
Hepar2	163	495	5.4×10^5
Pigs	11243	985	2.6×10^5
Water	3320	605	6.8×10^4
Munin	4021194	3500	2.2×10^7

shows the cost of compiling the full joint distribution of the example discrete Bayesian networks. These compilations take on the order of several seconds; however, once compiled, computing each marginal probability — or any other query with a small BDD, such as disjoining together several variables — takes milliseconds. For comparison, Psi cannot compute a single marginal on any of these examples within two hours.

Ace, similar to Dice, reduces the Bayesian network probabilistic inference task to weighted model counting (with a very different encoding scheme). This gives Ace an inherent advantage over Dice on this task: Ace does not support arbitrary program constructs — such as conditional branching, procedures, and observe statements — and hence can specialize directly for Bayesian networks, a limited subclass of Dice programs.

Despite these inherent advantages, Table 3 shows that Dice is competitive with Ace on a number of challenging Bayesian network inference tasks. Ace significantly outperforms Dice only on the very largest network, “Munin”. These results suggest that even though Dice is a general-purpose PPL, it is still a competitive exact inference algorithm for medium-sized Bayesian networks.

6 DISCUSSION & ANALYSIS

The previous section demonstrates empirically that Dice can perform exact inference orders of magnitude faster than existing inference algorithms on a range of benchmarks. In this section we provide discussion and analysis that provide context for these results. First we ask in Section 6.1: how hard is exact inference in Dice? We show that inference is PSPACE-hard, which means that it is likely harder than inference on discrete Bayesian networks. This begs the question: why do the experiments in Section 5 succeed at all? We explore this question in Section 6.2 by identifying different forms of program structure that Dice exploits in order to scale. Finally, Section 6.3 considers algebraic representations as an alternative compilation target for probabilistic programs and discusses the forms of structure that they are and are not capable of exploiting.

6.1 Computational Hardness of Exact Dice Inference

The experiments in Section 5 raise a natural question: how hard is the exact inference challenge for Dice programs? The complexity of exact inference has been well-studied in the context of discrete Bayesian networks. In particular, the decision problem of determining whether or not the probability of an event in a Bayesian network exceeds a certain threshold is PP-complete [Kwisthout 2009; Littman et al. 1998]. The canonical PP-complete problem is MAJSAT, the problem of deciding whether or not the majority of truth assignments satisfy a logical formula. It is clear that exact Dice is PP-hard: indeed, some of our experiments in Section 5 utilize a polynomial-time reduction

from discrete Bayesian networks to Dice programs. However, in fact exact inference for Dice is PSPACE-hard, and therefore likely harder than discrete Bayesian network inference as $PP \subseteq PSPACE$:

THEOREM 6.1. *Exact inference in Dice is PSPACE-hard.*

A proof sketch is in the full version of the paper. This result depends on the expressiveness of functions, which Bayesian networks lack. We leave for future work the investigation of tighter complexity bounds for Dice inference.

6.2 When Is Dice Inference Fast?

Dice inference, in the worst case, is extremely hard. Why, then, do the experiments in Section 5 succeed? Put another way: when can we guarantee that the BDD derivation tree is efficient to construct (i.e., polynomial in the size of the program)? In this section we explore two sources of tractability in Dice inference, both of which are structural properties that a programmer can consciously exploit while designing Dice programs. The first source of structure is *independence*, which implies the existence of factorizations. The second is a more subtle property called *local structure* that implies that, even in some cases without independence, it can still be efficient to construct the BDD derivation tree [Boutilier et al. 1996; Chavira and Darwiche 2005]. These forms of structure were first introduced in the context of graphical models for capturing conditional probability tables with various forms of structure. We show that these insights can be generalized to Dice programs.

6.2.1 Independence. The independence property implies that two program parts communicate only over a limited interface. It is the key reason why Dice performs so well in many of the benchmarks (Section 5.2.1). Programs naturally have conditional independence, implied by their control flow, function boundaries, etc. In the motivating example in Figure 1b, variable z does not depend on x given an assignment to y . This is commonly called *conditional independence* of x and z given y , and it partially explains why Dice scales to thousands of conditionally independent layers in Figure 10d.

Dice naturally exploits conditional independence. We can formalize this by giving bounds on the cost of composing BDDs that are conditionally independent. In general, the operation $B_1 \wedge B_2$ on two BDDs B_1 and B_2 has time and space complexity $O(|B_1| \times |B_2|)$, and similarly for $B_1 \vee B_2$ [Meinel and Theobald 1998]. This implies a worst-case exponential blowup as BDDs are composed. However, Dice can exploit conditional independence — among other properties — to avoid this exponential blowup in practice:

PROPOSITION 6.2. *Let B_1 and B_2 be BDDs that share no variables other than some variable z , and let $|B|$ be the size of the BDD B . Then we say B_1 and B_2 are conditionally independent given z , and computing $B_1 \wedge B_2$ and $B_1 \vee B_2$ has time and space complexity $O(|B_1| + |B_2|)$ for a variable order that orders the variables in B_1 before z and z before the variables in B_2 .*

Proposition 6.2 implies that compositional rules that utilize conjunction and disjunction to compose Dice programs — like C-LET — can be efficient in the presence of conditional independence. One useful source of conditional independence is function calls: they are conditionally independent from all other expressions given their arguments and return value. The motivating example in Figure 2 illustrates an example of this form of conditional independence. Each call to the diamond procedure is independent of all prior calls given only the immediately previous call. It follows that the size of the BDD for the example in Figure 2d grows as $O(|\text{diamond}| \times c)$, where c is the number of calls to the diamond procedure and $|\text{diamond}|$ is the size of the compiled BDD for the procedure.

Dice exploits another, more fine-grained form of independence called *context-specific independence*. Historically, context-specific independence has led to significant speedups in graphical

```

1  let z = flip1 0.5 in
2  let x = if z then flip2 0.6 else flip3 0.7 in
3  let y = if z then flip4 0.7 else x in (x, y)

```

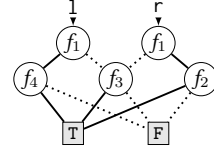
(a) Context-specific independence.

```

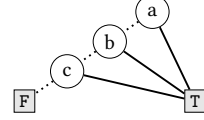
1  fun foo(a:Bool, b:Bool, c:Bool):Bool {
2    a ∨ b ∨ c
3  }

```

(c) Structure without independence.



(b) Compiled BDD.



(d) Compiled BDD.

Fig. 12. Dice programs and their compiled BDDs illustrating different degrees of structure.

model inference [Boutilier et al. 1996]. We briefly sketch its benefits here. Two BDDs B_1 and B_2 are *contextually independent given* $z = v$, for some variable z and value v , if $B_1[z \mapsto v]$ and $B_2[z \mapsto v]$ share no variables [Boutilier et al. 1996]. As for conditional independence, composing contextually independent BDDs can often be efficient.

An example program that exhibits context-specific independence is shown in Figure 12a. The variables x and y are correlated if $z = F$ or if z is unknown, but they are independent if $z = T$. Thus, x is independent of y given $z = T$. Figure 12b shows how our compilation strategy exploits this independence. Since the program evaluates to a tuple, it is compiled to a tuple of two BDDs. However, in our implementation these BDDs share nodes wherever possible, so they can be equivalently viewed as a single, *multi-rooted BDD*. The left and right element of the tuple are represented by the l and r roots respectively. The program’s context-specific independence implies that there will be no shared sub-BDD between l and r if f_1 is true. We refer to Boutilier et al. [1996] for more on the performance benefits of exploiting context-specific independence in probabilistic graphical models.

6.2.2 Local Structure. Finally, it is possible for the BDD compilation process to be efficient even in the absence of independence if the program has structure that is amenable to efficient BDD compilation. Chavira and Darwiche [2005] showed that exploiting local structure led to significant speedups in Bayesian network inference, and this performance was one of the primary motivations for developing Ace. Local structure is a broad category of structural properties that can make performance more efficient, including determinism, context-specific independence, and other properties [Boutilier et al. 1996; Chavira and Darwiche 2008; Gogate and Dechter 2011; Sang et al. 2005].

At its core, local structure is a property that makes compiling a BDD more efficient than naively using a conditional probability table to represent a probability distribution. Figure 12c gives an example Dice function that computes the disjunction of three arguments. Figure 12d shows the compiled BDD for this function. It is compact and hence exploiting the program structure. Note that, if the number of variables disjoined together were to increase, the size of the BDD — and the cost of compiling it — would increase only linearly with the number of variables. This stands in stark contrast to an approach to inference that is agnostic to local structure (such as simple variable elimination), which would not identify that this or-function is a compact way of representing the distribution.

Dice implicitly exploits local structure during inference. For instance, the Bayesian network “Hepar2” has many examples of determinism, sparse probability tables, and context-specific independence; Dice exploits these properties to be competitive with the performance of Ace on this example and others in Table 3.

6.3 Algebraic Representations

Previous sections have shown that BDDs naturally capture and exploit factorization and procedure reuse. While these are common and useful program properties, they are not the only possible ones, and different compilation targets will naturally exploit others. In this section we consider *algebraic compilation targets* as a foil to our approach, to highlight their relative strengths and weaknesses.

In contrast to our WMC approach that explicitly separates the logical representation from probabilities, algebraic approaches integrate probabilities directly into the compilation target. A common algebraic target are *algebraic decision diagrams* (ADDs) [Bahar et al. 1997], which are similar to binary decision diagrams except that they have numeric values as leaves. This makes them a natural choice for compactly encoding probability distributions in the probabilistic programming and probabilistic model checking communities, with different encoding strategies from Dice [Claret et al. 2013; Dehnert et al. 2017; Kwiatkowska et al. 2011]. As an example, Figure 13 shows an ADD for the program in Figure 1a if it returned a tuple of x , y , and z . ADDs encode probabilities of total assignments of variables: in this example, a probability of 0.008 is given to the assignment $x = y = z = T$.

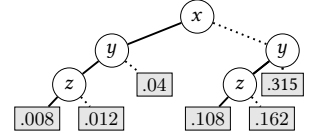


Fig. 13. An ADD representation of the distribution in Equation 1.

ADDs have several similarities with BDDs. First, they support composition operations and so can offer a compositional compilation target [Claret et al. 2013], albeit very different from the one described by our compilation rules. Second, they support efficient inference once the ADD is constructed. Despite these similarities, ADDs have strikingly different scaling properties from BDDs because they exploit different underlying structure of the program. The key difference is that BDDs are agnostic to the `flip` parameters: they naturally exploit logical program structure such as independence and local structure in order to scale without needing to know what any probabilities are. As the previous subsections have argued, BDDs excel at this task. In contrast, ADDs naturally exploit *global repetitious probabilities*: repeated probabilities of possible worlds in the entire distribution. This is shown in Figure 13, which collapses states with the same probability — for example, if $x = y = F$, then the ADD terminates with a node that does not depend on z ’s value: `.315`.

Global repetitious probabilities are an orthogonal property to independence. ADDs do not exploit independence in the same way as Dice. ADDs must explicitly represent the probability of each total instantiation of the variables of interest, corresponding to each possible value of the returned tuple. In our example, this means that the ADD cannot exploit the conditional independence of z and x given y , and instead needs to enumerate their joint probabilities.

Hence, unlike Dice’s BDD representation, the size of a compiled ADD is sensitive to the precise parameters chosen for `flip`s in the program. If these parameters are chosen such that the probability of each total assignment is distinct, and we are interested in a tuple of all the random variables, then the number of leaves in the ADD will equal the number of paths in the probabilistic program. As shown in Table 1, this can be prohibitively large for many examples; the BDD size is typically many orders of magnitude smaller than the number of paths on these real-world programs.

7 RELATED WORK

There is a large literature on probabilistic programming languages and inference algorithms. At a high level, Dice is distinguished from existing PPLs by being the first to use weighted model counting to perform exact inference for a PPL that includes traditional programming language constructs, functions, and first-class observations. In this section we survey the existing literature on probabilistic program inference and provide context for how each relates to Dice.

Path-based inference algorithms. The most common class of probabilistic program inference algorithms today are *operational*, meaning that they work by executing the probabilistic program on concrete values. Common examples include sampling algorithms [Carpenter et al. 2016; Chaganty et al. 2013; Goodman et al. 2008; Hur et al. 2015; Mansinghka et al. 2013, 2018; Pfeffer 2007a; Saad and Mansinghka 2016; van de Meent et al. 2015; Wood et al. 2014] and variational approximations [Bingham et al. 2019; Dillon et al. 2017; Kucukelbir et al. 2015; Minka et al. 2014; Wingate and Weber 2013]. Other approaches use symbolic techniques to perform inference but are similar in spirit, in the sense that they separately enumerate paths through the program [Albarghouthi et al. 2017; Filieri et al. 2013; Geldenhuys et al. 2012; Sankaranarayanan et al. 2013]. These approaches do not factorize the program: they consider entire execution paths as a whole. Chistikov et al. [2015] proposes performing *weighted model integration* — a generalization of weighted model counting to the continuous domain [Belle et al. 2015; Dos Martires et al. 2019; Zeng and Van den Broeck 2020] — to perform inference by integrating along paths through a probabilistic program.

Additionally, sampling and variational algorithms are distinguished from our approach by being approximate rather than exact inference algorithms. In general, these techniques can be applied to both discrete and continuous distributions, though they often rely on program continuity or differentiation to be effective [Carpenter et al. 2016; Gram-Hansen et al. 2018; Hoffman and Gelman 2014; Kucukelbir et al. 2015; Minka et al. 2014; Wingate and Weber 2013]. In contrast to all of these approaches, Dice performs factorized, exact inference on non-smooth, non-differentiable, discrete programs.

Algebraic inference algorithms. A number of PPL inference algorithms work by translating the probabilistic program into an algebraic expression that encodes its probability distribution, and then using symbolic algebra tools in order to manipulate that expression and perform probabilistic inference. Examples include Psi [Gehr et al. 2016], Hakaru [Narayanan et al. 2016], and approaches that employ algebraic decision diagrams [Claret et al. 2013; Dehnert et al. 2017]. Algebraic representations exploit fundamentally different program structure from our approach based on weighted model counting; see Section 6.3 for a discussion.

Graphical model compilation. There exists a large number of PPLs that perform inference by converting the program into a probabilistic graphical model [Bornholt et al. 2014; McCallum et al. 2009; Minka et al. 2014; Pfeffer 2009]. These compilation strategies are limited by the semantics of graphical models: key program structure — such as functions, conditional branching, *etc.* — is usually lost during compilation and so cannot be exploited during inference. Further, graphical models can express conditional independence via the graphical structure, but typical inference algorithms such as variable elimination cannot exploit more subtle, context-specific forms of independence that our approach exploits, as shown in Section 6.2.1 [Darwiche 2009].

Probabilistic Logic Programs. Closest to our approach are techniques for exact inference in probabilistic logic programs [De Raedt et al. 2007; Fierens et al. 2015; Riguzzi and Swift 2011; Vlasselaer et al. 2015]. Similar to our work, these techniques reduce probabilistic inference to weighted model counting and employ representations that support efficient WMC, such as BDDs [Bryant 1986]

or sentential decision diagrams [Darwiche 2011]. Unlike that work, Dice supports traditional programming language constructs, including functions, and it supports first-class observations rather than only observations at the very end of the program. We show how to exploit functional abstraction for modular compilation, and first-class observations require us to explicitly account for an *accepting* probability in both the semantics and the compilation strategy.

Programmer-Guided Inference Decomposition. Several PPLs provide a sublanguage that allows the programmer to provide information that can be used to decompose program inference into multiple separate parts [Holtzen et al. 2018; Mansinghka et al. 2018; Pfeffer et al. 2018]. Hence the goal is similar in spirit to our goal of automated program factorization. These approaches are complementary to ours: Dice automatically finds and exploits program factorizations and local structure, while these approaches can perform sophisticated decompositions through explicit programmer guidance.

Static Analysis & Model Checking. Forms of symbolic model checking often represent the reachable state space of a program as a BDD [Biere 2009; Jhala and Majumdar 2009]. Our work can be thought of as enriching this representation with probabilities: we track the possible assignments to each flip and the accepting formula in order to do exact Bayesian inference via WMC. Static analysis techniques have also been generalized to analyze probabilistic programs. For example, probabilistic abstract interpretation [Cousot and Monerau 2012] provides a general framework for static analysis of probabilistic programs. However, these techniques seek to acquire lower or upper bounds on probabilities, while we target exact inference. Probabilistic model checking (PMC) is a mature generalization of traditional model checking with multiple high-quality implementations [Dehnert et al. 2017; Kwiatkowska et al. 2011]. The goal of PMC is typically to verify that a system meets a given probabilistic temporal logic formula. They can also be used to perform probabilistic inference, but they have not used weighted model counting for inference and instead typically rely on ADDs, which gives them different scaling properties than Dice as we discussed earlier. Vazquez-Chanlatte and Seshia [2020] recently described an approach to learn Boolean task specifications on Markov decision processes. This work shares some core technical machinery with our approach but differs markedly in its goals and encoding strategy.

8 CONCLUSION

We presented a new approach to exact inference for discrete probabilistic programs and implement it in the Dice probabilistic programming language. We (1) showed how to reduce exact inference for Dice to weighted model counting, (2) proved this translation correct, (3) demonstrated the performance of this inference strategy over existing methods, and (4) characterized the efficiency of compiling Dice in key scenarios.

In the future we hope to extend Dice in several ways. First, we believe that the insights of Dice can be cleanly integrated into many existing probabilistic programming systems, even those with approximate inference that can handle continuous random variables. We see this as an exciting avenue for extending the reach of approximate inference algorithms, which currently struggle with discreteness. Second, we believe that Dice can be extended to handle more powerful data structures and programming constructs, notably forms of unbounded loops and recursion. And finally, we hope to further explore the landscape of weighted model counting approaches.

ACKNOWLEDGMENTS

This work is partially supported by NSF grants #IIS-1943641, #IIS-1956441, #CCF-1837129, DARPA grant #N66001-17-2-4032, a Sloan Fellowship, and gifts by Intel and Facebook research. The authors would like to thank Jon Aytac and Philip Johnson-Freyd for feedback on paper drafts.

Table 4. Comparison of inference algorithms on standard baselines (times are milliseconds). The reported time is the mean plus or minus a single standard deviation over 5 runs.

Benchmark	Psi (ms)	DP (ms)	Dice (ms)
Grass	167±2	58±2	14.0±1.0
Burglar Alarm	98±14	30±2	13.0±0.1
Coin Bias	94±19	23±13	13.0±1.5
Noisy Or	81±38	152±10	13.0±2.0
Evidence1	70±34	43±23	12.9±1.3
Evidence2	67±40	46±23	13.2±2.3
Murder Mystery	193±33	75±10	13.6±1.6

Table 5. Comparison of inference algorithms on the single-marginal inference task (times are milliseconds). The reported time is the mean plus or minus a single standard deviation over 5 runs. A single standard deviation and the mean are reported.

Benchmark	Psi (ms)	DP (ms)	Dice (ms)
Cancer [Korb and Nicholson 2010]	772±60	46±2	13±3
Survey [Scutari and Denis 2014]	2477±569	152±58	13±1
Alarm [Beinlich et al. 1989]	✗	✗	25±3
Insurance [Binder et al. 1997]	✗	✗	212±12
Water [Jensen et al. 1989]	✗	✗	2590±21
Hailfinder [Abramson et al. 1996]	✗	✗	618±8
Hepar2 [Onisko 2003]	✗	✗	48±6
Pigs	✗	✗	72±2
Munin [Andreassen et al. 1989]	✗	✗	1866±27

Table 6. *All marginals*. A comparison between Dice and Ace on the all-marginal discrete Bayesian network inference task. A single standard deviation and the mean are reported.

Benchmark	Dice (ms)	Ace (ms)
Alarm	159±12	422±32
Hailfinder	1280±16	522±37
Insurance	222±1	492±34
Hepar2	163±3	495±17
Pigs	11 243±79	985±76
Water	3320±118	605±10
Munin	4 021 194±2 123 290	3500±575

A SUPPLEMENTAL EXPERIMENTAL RESULTS

This section extends the experimental results by showing the mean and standard deviation over at least 5 runs for all of the tables in the main body of the paper. Table 4 extends Table 1, Table 5 extends Table 2, and Table 6 extends Table 3.

REFERENCES

- Bruce Abramson, John Brown, Ward Edwards, Allan Murphy, and Robert L Winkler. 1996. Hailfinder: A Bayesian system for forecasting severe weather. *International Journal of Forecasting* 12, 1 (1996), 57–71. [https://doi.org/10.1016/0169-2070\(95\)00664-8](https://doi.org/10.1016/0169-2070(95)00664-8)
- Aws Albarghouthi, Loris D’Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 80 (Oct. 2017), 30 pages. <https://doi.org/10.1145/3133904>
- Steen Andreassen, Finn V Jensen, Stig Kjær Andersen, B Falck, U Kjærulff, M Woldbye, AR Sørensen, A Rosenfalck, and F Jensen. 1989. MUNIN: an expert EMG Assistant. In *Computer-aided electromyography and expert systems*. Pergamon Press, 255–277. [https://doi.org/10.1016/0924-980x\(95\)00252-g](https://doi.org/10.1016/0924-980x(95)00252-g)
- R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic decision diagrams and their applications. *Formal methods in system design* 10, 2-3 (1997), 171–206.
- Ingo A Beinlich, Henri Jacques Suermondt, R Martin Chavez, and Gregory F Cooper. 1989. The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks. In *AIIME 89*. Springer, 247–256. https://doi.org/10.1007/978-3-642-93437-7_28
- Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *Proc. of IJCAI*. 2770–2776.
- Armin Biere. 2009. Bounded Model Checking. In *Handbook of Satisfiability*, Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, Chapter 14.
- John Binder, Daphne Koller, Stuart Russell, and Keiji Kanazawa. 1997. Adaptive probabilistic networks with hidden variables. *Machine Learning* 29, 2-3 (1997), 213–244. <https://doi.org/10.1023/A:1007421730016>
- 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. *The Journal of Machine Learning Research* 20, 1 (2019), 973–978.
- Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure transformer semantics for Bayesian machine learning. In *European symposium on programming*. Springer, 77–96.
- James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. 2014. Uncertain<T>: A first-order type for uncertain data. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 51–66. <https://doi.org/10.1145/2654822.2541958>
- Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. 1996. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 115–123.
- R. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE TC C-35* (1986), 677–691. <https://doi.org/10.1109/TC.1986.1676819>
- Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2016. Stan: A probabilistic programming language. *Journal of Statistical Software* (2016).
- Arun Chaganty, Aditya Nori, and Sriram Rajamani. 2013. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*. 153–160.
- Mark Chavira and Adnan Darwiche. 2005. Compiling Bayesian networks with local structure. In *IJCAI*. 1306–1312.
- Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *J. Artificial Intelligence* 172, 6-7 (April 2008), 772–799. <https://doi.org/10.1016/j.artint.2007.11.002>
- Mark Chavira, Adnan Darwiche, and Manfred Jaeger. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning* 42, 1 (2006), 4–20.
- Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. 2015. Approximate Counting in SMT and Value Estimation for Probabilistic Programs. In *Proc. of TACAS*. Springer-Verlag New York, Inc., New York, NY, USA, 320–334. https://doi.org/10.1007/978-3-662-46681-0_26
- Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (2013), 92. <https://doi.org/10.1145/2491411.2491423>
- Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.
- Patrick Cousot and Michael Monerau. 2012. Probabilistic abstract interpretation. In *Proc. of ESOP*. 169–193. https://doi.org/10.1007/978-3-642-28869-2_9
- Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K Mansinghka. 2018. Incremental inference for probabilistic programs. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 571–585. <https://doi.org/10.1145/3296979.3192399>
- Adnan Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511811357>
- Adnan Darwiche. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. 819.

- A. Darwiche and P. Marquis. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.
- Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of IJCAI*, Vol. 7. 2462–2467.
- Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*. Springer, 592–600.
- Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. TensorFlow Distributions. *arXiv preprint arXiv:1711.10604* (2017).
- Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. 2019. Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7825–7833.
- 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. *J. Theory and Practice of Logic Programming* 15(3) (2015), 358 – 401. <https://doi.org/10.1017/S1471068414000076>
- Antonio Filieri, Corina S Păsăreanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 622–631.
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, USA, June 23–25, 1993, Robert Cartwright (Ed.). ACM, 237–247. <https://doi.org/10.1145/155090.155113>
- Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: probabilistic inference for networks. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 586–602. <https://doi.org/10.1145/3296979.3192400>
- Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.
- Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 166–176. <https://doi.org/10.1145/2338965.2336773>
- V. Gogate and R. Dechter. 2011. SampleSearch: Importance sampling in presence of determinism. *Artificial Intelligence* 175, 2 (2011), 694–729.
- Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI)*.
- Noah D Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages.
- Maria I Gorinova, Dave Moore, and Matthew D Hoffman. 2020. Automatic Reparameterisation of Probabilistic Programs. *International Conference on Machine Learning (ICML)* (2020).
- Bradley Gram-Hansen, Yuan Zhou, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. 2018. Hamiltonian Monte Carlo for Probabilistic Programs with Discontinuities. *arXiv preprint arXiv:1804.03523* (2018).
- Matthew D Hoffman and Andrew Gelman. 2014. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15, 1 (2014), 1593–1623.
- Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2018. Sound abstraction and decomposition of probabilistic programs. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*.
- Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *arXiv:arXiv:2005.09089*
- Daniel Huang and Greg Morrisett. 2016. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag New York, Inc., New York, NY, USA, 337–363. https://doi.org/10.1007/978-3-662-49498-1_14
- Chung-kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Sammuel. 2015. A Provably Correct Sampler for Probabilistic Programs. *FSTTCS FSTTCS* (2015), 1–14. <https://doi.org/10.4230/LIPIcs.FSTTCS.2015.475>
- FV Jensen, U Kjærulff, KG Olesen, and J Pedersen. 1989. *An expert system for control of waste water treatment—a pilot project*. Technical Report. Technical report, Judex Datasystemer A/S, Aalborg, 1989. In Danish.
- Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *Comput. Surveys* 41, 4 (2009), 1–54. <https://doi.org/10.1145/1592434.1592438>
- M.I. Jordan, Z. Ghahramani, T.S. Jaakkola, and L.K. Saul. 1999. An introduction to variational methods for graphical models. *Machine learning* 37, 2 (1999), 183–233. <https://doi.org/10.1023/A:1007665907178>
- Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. 1996. *Handbook of applied cryptography*. CRC press.
- D. Koller and N. Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.
- Kevin B Korb and Ann E Nicholson. 2010. *Bayesian artificial intelligence*. CRC press. <https://doi.org/10.1201/b10391>

- Dexter Kozen. 1979. Semantics of Probabilistic Programs. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (SFCS '79)*. IEEE Computer Society, Washington, DC, USA, 101–114. <https://doi.org/10.1109/SFCS.1979.38>
- Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. 2015. Automatic variational inference in Stan. In *Advances in neural information processing systems*. 568–576.
- Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. 2017. Automatic differentiation variational inference. *The Journal of Machine Learning Research* 18, 1 (2017), 430–474.
- Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (Snowbird, UT) (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47
- Johan Henri Petrus Kwiouthout. 2009. *The computational complexity of probabilistic networks*. Utrecht University.
- Michael L Littman, Judy Goldsmith, and Martin Mundhenk. 1998. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research* 9 (1998), 1–36. <https://doi.org/10.1613/jair.505>
- Vikash Mansinghka, Tejas D Kulkarni, Yura N Perov, and Josh Tenenbaum. 2013. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems*. 1520–1528.
- 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 (Philadelphia, PA, USA) (PLDI 2018)*. ACM, New York, NY, USA, 603–616. <https://doi.org/10.1145/3192366.3192409>
- A McCallum, K Schultz, and S Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. *Proc. of NIPS* 22 (2009), 1249–1257.
- Christoph Meinel and Thorsten Theobald. 1998. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer Verlag. <https://doi.org/10.1007/978-3-642-58940-9>
- T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. <http://research.microsoft.com/infernet>.
- Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer, 62–79. https://doi.org/10.1007/978-3-319-29604-3_5
- Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI*. 2476–2482.
- Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. 2019. Tensor variable elimination for plated factor graphs. (2019), 4871–4880.
- Agnieszka Onisko. 2003. Probabilistic causal models in medicine: Application to diagnosis of liver disorders. In *Ph. D. dissertation, Inst. Biocybern. Biomed. Eng., Polish Academy Sci., Warsaw, Poland*.
- Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.
- Avi Pfeffer. 2007a. A general importance sampling algorithm for probabilistic programs. (2007). <http://nrs.harvard.edu/urn-3:HUL.InstRepos:25235125>
- Avi Pfeffer. 2007b. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. *Introduction to statistical relational learning* 1993 (2007), 399.
- Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137 (2009).
- Avi Pfeffer, Brian Rutenber, William Kretschmer, and Alison OConnor. 2018. Structured Factored Inference for Probabilistic Programming. In *International Conference on Artificial Intelligence and Statistics*. 1224–1232.
- Fabrizio Riguzzi and Terrance Swift. 2011. The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty. *Theory and Practice of Logic Programming* 11, 4–5 (2011), 433–449. <https://doi.org/10.1017/S147106841100010X>
- Feras Saad and Vikash Mansinghka. 2016. A Probabilistic Programming Approach To Probabilistic Data Analysis. In *Advances in Neural Information Processing Systems (NIPS)*.
- Tian Sang, Paul Beame, and Henry A Kautz. 2005. Performing Bayesian inference by weighted model counting. In *AAAI*, Vol. 5. 475–481.
- Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. *SIGPLAN Not.* 48, 6 (June 2013), 447–458. <https://doi.org/10.1145/2499370.2462179>
- Marco Scutari and Jean-Baptiste Denis. 2014. *Bayesian networks: with examples in R*. CRC press. <https://doi.org/10.1111/biom.12369>
- Fabio Somenzi. [n.d.]. CUDD: BDD package, University of Colorado, Boulder.

- Jan-Willem van de Meent, Hongseok Yang, Vikash Mansinghka, and Frank Wood. 2015. Particle Gibbs with Ancestor Sampling for Probabilistic Programs. In *AISTATS*.
- Guy Van den Broeck and Dan Suciu. 2017. *Query Processing on Probabilistic Data: A Survey*. Now Publishers. <https://doi.org/10.1561/19000000052>
- Marcell Vazquez-Chanlatte and Sanjit A Seshia. 2020. Maximum Causal Entropy Specification Inference from Demonstrations. In *International Conference on Computer Aided Verification*. Springer.
- Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. 2015. Anytime inference in probabilistic logic programs with Tp-compilation. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*. <https://doi.org/10.1016/j.ijar.2016.06.009>
- Di Wang, Jan Hoffmann, and Thomas Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. *SIGPLAN Not.* 53, 4 (June 2018), 513–528. <https://doi.org/10.1145/3296979.3192408>
- David Wingate and Theophane Weber. 2013. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299* (2013).
- Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*. 1024–1032.
- Zhe Zeng and Guy Van den Broeck. 2020. Efficient search-based weighted model integration. In *Uncertainty in Artificial Intelligence*. PMLR, 175–185.
- 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. *International Conference on Machine Learning* (2020).