UNIVERSITY OF CALIFORNIA
Los Angeles

Exploiting Program Structure for Scaling Probabilistic Programming

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Steven J. Holtzen

2021

ABSTRACT OF THE DISSERTATION

Exploiting Program Structure for Scaling Probabilistic Programming

by

Steven J. Holtzen
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Todd Millstein, Co-Chair
Professor Guy Van den Broeck, Co-Chair

Probabilistic modeling and reasoning are central tasks in artificial intelligence and machine learning. A probabilistic model is a rough description of the world: the model-builder attempts to capture as much detail about the world's complexities as she can, and when no more detail can be given the rest is left as probabilistic uncertainty. Once constructed, the goal of a model is to perform automated *inference*: compute the probability that some particular fact is true about the world. It is natural for the model-builder to want a flexible expressive language – the world is a complex thing to describe – and over time this has led to a trend of increasingly powerful modeling languages. This trend is taken to its apex by *probabilistic programming languages* (PPLs), which enable modelers to specify probabilistic models using the facilities of a full programming language. However, this expressivity comes at a cost: the computational cost of inference is in direct tension with the flexibility of the modeling language, and so it becomes increasingly difficult to design automated inference algorithms that scale to the kinds of systems that model builders want to create.

This thesis focuses on the central question: *how can we design effective probabilistic programming languages that profitably trade off expressivity and tractability for inference?* The approach taken here is first to identify and exploit important *structure* that a probabilistic

program may possess. The kinds of structure considered here are discrete program structure and symmetry. Programs are heterogeneous objects, so different parts of programs may exhibit different kinds of structure; in the second part of the thesis I show how to *decompose* heterogeneous probabilistic program inference using a notion of program abstraction. These contributions enable new applications of probabilistic programs in domains such as text analysis, verification of probabilistic systems, and classical simulation of quantum algorithms.

The dissertation of Steven J. Holtzen is approved.

Jens Palsberg

Eran Halperin

Guy Van den Broeck, Committee Co-Chair

Todd Millstein, Committee Co-Chair

University of California, Los Angeles

2021

For Anna

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First I would like to thank my advisors Guy Van den Broeck and Todd Millstein. I could not have asked for a better advising duo. Guy is a tireless source of ideas. If you are stuck, Guy will get you unstuck, and will often surprise you when he does. Todd is methodical and patient: he is not satisfied until he understands every bit of what you are trying to tell him. I am lucky to work with both of them.

My lab-mates in the StarAI lab at UCLA – Tal Friedman, YooJung Choi, Yitao Liang, Pasha Khosravi, Zhe Zeng, Honghua Zhang, Kareem Ahmed, and Aishwarya Sivaraman – are a constant source of inspiration and companionship. At some point or another I've cornered every one of them and made them listen to me at the whiteboard. Thanks for your patience and your genuine curiosity. Your presence is one of the things I have most missed most during this past year.

My friends, mentors, and advisors in the Center for Vision, Cognition, Learning, and Autonomy at UCLA – Mark Edmonds, Yixin Zhu, Siyuan Qi, Yibiao Zhao, and Song-Chun Zhu – gave me my initial inspiration and taste of research. They taught me how to dream big and aim high with research.

Jon Aytac and Philip Johnson-Freyd were my mathematics role models throughout my PhD. Whenever I described my problems to them, their first response was always to frame it in the broader context of mathematics. In doing so, they showed me the power and beauty of higher math – categories, Fourier analysis, and so many words I wish I understood – and inspired me to reach towards it. Without them this work would have looked very different.

My mentors and advisors at Sandia National Laboratories – Philip Kegelmeyer, Karim Mahrous, and Chris Harrison – made every Summer I spent visiting Livermore a valuable experience. I would especially like to thank Karim for his initial vision and ambition in making my unusual relationship with Sandia possible.

I was extremely fortunate to have some excellent collaborators during my PhD. In particular, it was a pleasure working closely with Yipeng Huang, Sebastian Junges, and Marcell Vazquez-Chanlatte. Thanks for your curiosity and excitement.

My family was behind me every step of the way on this long and sometimes arduous PhD. journey. When things got tough, it was always a joy to go back home to Arcadia and spend some time in the sun. Thank you for your unending support.

Finally I want to thank Anna, who this thesis is dedicated to. Though we spent some time separated by distance, you were never far away.

Attribution. Chapter 3 is based on Holtzen et al. [2020]. Chapter 4 is based on Holtzen et al. [2019]. Chapter 5 is based on Holtzen et al. [2017] and Holtzen et al. [2018]. On all these references I am the primary author and the other authors were my advisors.

VITA

2014	Intern, Symantec Corporation
2015	Intern, Palantir Technologies
2015	B.S. *cum laude* in Computer Science. UCLA.
2015–2021	Member Technical Staff, Sandia National Laboratories
2015–2017	National Physical Sciences Consortium Fellowship
2017	UCLA Computer Science Outstanding Graduating Master's Student
2017	M.S. in Computer Science. UCLA.
2018	Teaching Assistant, CS267A: Statistical Relational Learning and Probabilistic Programming
2020	Teaching Assistant, CS30: Principles and Practices of Computing
2020–2021	UCLA Dissertation Year Fellowship
2020	ACM SIGPLAN Distinguished Paper Award for *Scaling Exact Inference for Discrete Probabilistic Programs*
2021	UCLA Computer Science Outstanding Graduating PhD. Student

PUBLICATIONS

S. Holtzen, Y. Zhao, T. Gao, J. B. Tenenbaum, and S.-C. Zhu. Inferring human intent from video by sampling hierarchical plans. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1489–1496. IEEE, 2016.

S. Holtzen, T. Millstein, and G. Van den Broeck. Probabilistic program abstractions. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

S. Holtzen, G. Van den Broeck, and T. Millstein. Sound abstraction and decomposition of probabilistic programs. In *International Conference on Machine Learning (ICML)*, 2018.

S. Holtzen, T. Millstein, and G. Van den Broeck. Generating and sampling orbits for lifted probabilistic inference. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

H. Zhang, S. Holtzen, and G. Van den Broeck. On the relationship between probabilistic circuits and determinantal point processes. In *Uncertainty in Artificial Intelligence (UAI)*, 2020.

S. Holtzen, G. Van den Broeck, and T. Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang. (OOPSLA)*, 2020.

Y. Huang, S. Holtzen, T. Millstein, G. Van den Broeck, and M. Martonosi. Logical abstractions for noisy variational quantum algorithm simulation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

S. Holtzen, S. Junges, M. Vazquez-Chanlatte, T. Millstein, S. Seshia, and G. Van den Broeck. Model checking finite-horizon Markov chains with probabilistic inference. In *Conference on Computer-Aided Verification (CAV)*, 2021.

CHAPTER 1

Introduction

The sciences do not try to explain, they hardly even try to interpret, they mainly make models.

John von Neumann

Probabilistic modeling is at the core of many scientific disciplines. A *probabilistic model* consists of two parts. First, one gives a rough description of the world called a *model*. The world is too complex to write down any precise set of rules describing its behavior, so the philosophy taken by a probabilistic model is to simplify the world by permitting the modeler to express uncertainty in the form of probabilities: "it is too hard to state for certain whether or not it will rain tomorrow, so we will settle for a 90% chance of rain." What good is a model if it cannot tell you anything? Given a probabilistic model, the goal is then to perform various forms of *probabilistic inference*: computing the probability that the model will exhibit some behavior.

Traditionally the development of a probabilistic model is tightly coupled with its corresponding inference algorithm. This poses two key challenges. First is the *challenge of accessibility*: there are few people who are capable of both designing an effective model and inference algorithm, both of which require highly specialized domain knowledge. Second is the *challenge of modularity*: by tightly coupling modeling and inference, it makes the model inflexible: in this tightly coupled situation, changing the model requires rewriting the entire delicately designed inference procedure. Taken together, these two challenges limit the applicability of probabilistic modeling to specific scientific disciplines with the technical resources to overcome them.

Probabilistic modeling frameworks resolve the problems of accessibility and modularity. The idea is to give a general-purpose accessible *probabilistic modeling language* in which the user specifies a model. Then, given a model, the system will automatically perform inference in a generic fashion. This strategy separates the two concerns of modeling and inference: an expert in inference can design a generic inference algorithm, and a domain expert can provide the model using the easy-to-use modeling language.

However, probabilistic modeling frameworks are no panacea. Probabilistic inference in many cases is computationally intractable, and hence every probabilistic modeling language must make a central tradeoff between *expressivity* and *tractability*. More flexible

1

models are attractive because they are more accessible to users and allow them to express properties of the world in a richer vocabulary. This increase in flexibility poses challenges for designing effective scalable inference.

The most flexible kinds of modeling languages are *probabilistic programming languages* (PPLs). The key idea of a PPL is to endow a traditional programming language – like Python, C, or Haskell – with a notion of uncertainty, such as the ability to flip a coin. Then, the *meaning of the program*, also known as its *semantics*, is defined as a probability distribution over all possible runs through the program. PPLs are obviously extremely flexible and general – they allow a modeler to express incredibly nuanced descriptions of the world – but this flexibility comes at the cost of effective inference algorithms.

Currently, the lack of scalability of inference is one of the primary factors holding probabilistic programming languages back from wide-spread application, and so scaling inference will be the central topic of this thesis. In particular, since inference is in general hard in the worst case, this thesis will argue that *finding and exploiting program structure using program analysis is essential for scaling inference to large probabilistic programs.*

1.1   Contributions & Structure of the Thesis

At a high level, the main contribution of this thesis is giving *three new strategies for scaling inference by exploiting the structure of probabilistic programs.* Chapter 3 describes `Dice`, a new probabilistic programming language for discrete probabilistic programs that exploits program factorization and modularity in order to scale. Then, Chapter 4 describes a new inference algorithm for exploiting the symmetry of probabilistic programs. Clearly there is a need for mixing and matching inference algorithms, since programs are complex heterogeneous objects with varying structure, so Chapter 5 gives a general-purpose approach for decomposing probabilistic programs based on program abstraction in order to mix and match the appropriate inference strategies.

The remainder of this chapter will go into further detail on the structure and motivation of each subsequent chapter.

1.1.1   Chapter 2: Foundations

Probabilistic programming is naturally a broad synthesis of topics in artificial intelligence and programming languages. This chapter aims to invite members of both audiences to the thesis by providing key background content for both perspectives. In particular, this chapter introduces (1) key definitions from probability; (2) high-level themes such as the trade off between conciseness and tractability of a probabilistic model; (3) the syntax and semantics of the `Dice` probabilistic programming language; (4) foundations of probabilistic program inference.

1.1.2 Chapter 3: `Dice`: Exploiting Factorization in Discrete Probabilistic Programs

One of the most challenging kinds of probabilistic programs for many existing probabilistic programming systems are *discrete programs*: programs that contain discrete random variables. This chapter introduces an inference algorithm for the `Dice` probabilistic programming language that exploits *factorization* and *modularity* – two common and important properties of probabilistic programs – in order to scale. In order to find this structure, `Dice` compiles probabilistic programs according to the following pipeline:

Dice Program	(1)	Dice Front-End	(2)	Dice Middle-End	(3)	Tractable Probabilistic Model	(4)	Query Answer

Figure 1.1: System diagram for performing a query on a `Dice` program.

First, programs are input to the system as source code and they are parsed into an intermediate representation by the `Dice` front-end. Then, in Step (2) programs are transformed into a core *intermediate representation* that supports various optimizations, desugaring, and other code-to-code transformations.

In a normal compiler, Step (3) would translate the intermediate representation into machine runnable code. This is different for `Dice`: `Dice` translates the intermediate representation into a *tractable probabilistic model* (TPM) [Choi et al., 2020]. This step may be expensive – this chapter will show that it is worst-case PSPACE-hard – but once it is successfully completed inference is efficient in the size of the compiled representation. Concretely, this means that Step (4) – going from the TPM to the Query Answer – is an efficient step.

Different kinds of TPMs are capable of recognizing and exploiting different kinds of program structure. Hence, the exact choice of back-end TPM heavily influences the scalability of inference. This chapter proposes *binary decision diagrams* (BDDs) as a back-end TPM, and shows that this choice exploits two kinds of program structure in order to scale inference to large programs: factorization and modularity. Intuitively, factorization is a property of the *size of the interface between two programs*. If two probabilistic programs communicate using a relatively small interface – for instance, one program calls another program that only takes a single argument – then inference should be able to exploit this small interface to solve the two inference problems nearly independently. Modularity, on the other hand, is a property of function calls: if the same function is called multiple times, then a modular analysis would analyze that function call once and re-use that analysis across subsequent calls. Both of these properties are abundant in natural programs, and BDDs naturally exploit both properties to scale inference to large examples.

This chapter will show that, together, these traits allow `Dice`'s inference algorithm to scale to programs that are orders of magnitude larger than existing commonly-used probabilistic programming languages on examples from text analysis, network verifica-
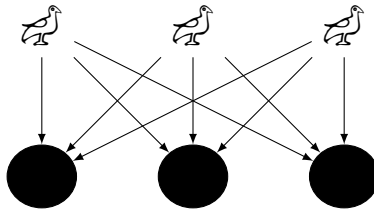
Figure 1.2: Pigeonhole problem illustration with 3 pigeons and 3 holes.

tion, and discrete Bayesian networks. This chapter also proves this inference algorithm correct with respect to a formal language semantics.

1.1.3   Chapter 4: Exploiting Symmetry with Lifted Inference

`Dice` compiles programs to binary decision diagrams, which implicitly exploits factorization and program modularity. However, this is not the only structure that a probabilistic program may possess that can enable fast probabilistic inference.

Symmetry is an orthogonal property to factorization that binary decision diagrams are not natively able to exploit that can enable fast inference. Intuitively, symmetry naturally arises when modeling distributions that are invariant under permutations of the sample space. Symmetries can be abundant in problems with little or no factorization, and exploiting symmetry can exponentially speed up probabilistic inference. In general, the class of inference algorithms that inherently exploit symmetry are referred to as *lifted inference* [Poole, 2003, Kersting, 2012, Niepert and Van den Broeck, 2014].

This is best illustrated with an example. Figure 1.2 gives an illustration of the classic *pigeonhole problem* with 3 pigeons and 3 holes. This situation can be made probabilistic by assuming that each pigeon has an identical strong preference for hiding in a hole without any other pigeons, so these configurations are given a high probability: the question then is *what is the probability that two pigeons end up in the same hole*?

Since all pigeons and holes are identical, the problem has abundant symmetry: the probability that the leftmost and rightmost pigeon end up in the leftmost hole is identical to the probability that they end up in the rightmost hole. Put more formally, the distribution is *invariant* to re-labelings of pigeons and holes. While this problem has symmetry, there is little factorization structure: a `Dice` program encoding of this problem would fail to effectively compile a BDD for even a modest number of pigeons and holes. Hence, there is a need for *more kinds of TPMs* that can be used as `Dice` backends that exploit different kinds of structure: in particular, in this case, we desire one that exploits symmetry.

A key challenge in applying lifted inference is *identifying the symmetries*. Most current approaches to lifted inference require a *relational representation*, for instance as a weighted first-order logic [Getoor and Taskar, 2007]. In other words, they require the problem to be given in a way that makes extracting the symmetries a matter of inspecting the structure
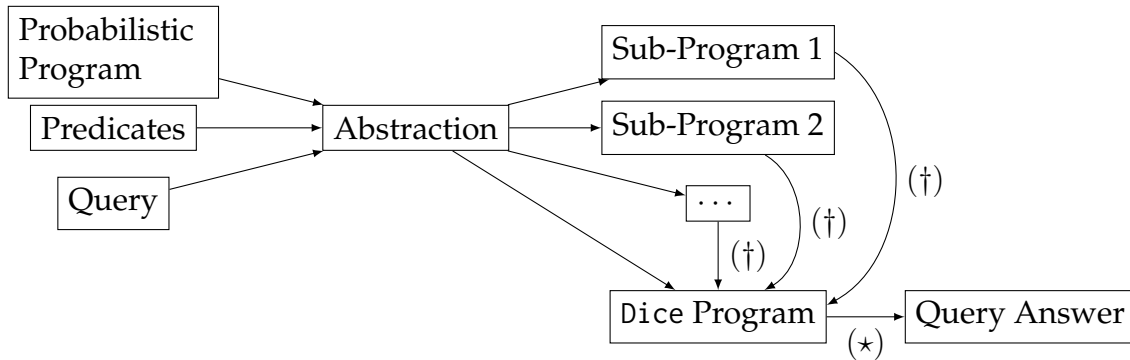
Figure 1.3: Diagram describing decomposition by abstraction.

of a first-order sentence. This does not directly apply to non-first order models such as probabilistic programs or probabilistic graphical models such as factor graphs, so this chapter aims to *expand the domain of lifted inference to encompass more probabilistic models beyond relational representations*. In particular, it (1) gives the first (exact) lifted inference algorithms for factor graphs, and (2) gives a lifted Markov-Chain Monte Carlo algorithm that provably scales in the degree of symmetry of the factor graph. Ultimately, this gives a foundation for new `Dice` backends that exploit symmetry.

1.1.4 Chapter 5: Decomposing Inference & Probabilistic Program Abstractions

So far the thesis has introduced two new strategies for performing inference in probabilistic programs that both work in very different ways and exploit different program structure. Moreover, there is a myriad of approaches to probabilistic inference beyond those introduced here. Hence there is a clear problem of *combining different approaches to probabilistic inference in order to perform inference on heterogeneous programs with different properties* that this chapter addresses.

This chapter gives a method for decomposing probabilistic program inference via *program abstraction*. Program abstractions – and in particular *predicate abstractions* – have a rich and successful history in non-probabilistic program analysis. The key idea is to generate a simplified *abstract program* from the original *concrete program* that captures a few key properties. This abstraction property simplifies the analysis – the new abstract program is by design simpler to analyze than the concrete program. If all goes well, the abstract program will then contain sufficient information to verify the concrete program.

This chapter gives a generalization for non-probabilistic predicate abstraction to probabilistic programs and shows how it decomposes inference. An outline is given in Figure 1.3. First, the programmer provides three pieces of data: a probabilistic program, a set of *predicates* that are Boolean random variables that capture properties about the program, and a query. Then, the abstraction engine automatically generates (1) a set of sub-

programs that are themselves probabilistic programs, and (2) an abstract `Dice` program that captures the relationship among predicates. Note that the abstract `Dice` program is discrete regardless of whether or not the input program is discrete: this is because the `Dice` program is only concerned with predicates.

In order to evaluate the final query, the abstract `Dice` program is *parameterized* by querying the sub-programs. Each (†) arrow in the figure represents a sub-query that queries a small part of the original program: this is the stage where inference is decomposed, since evaluating these (†) queries will ideally only require inspecting smaller portions of the original program. Finally, the final query is answered via a standard `Dice` inference query along the (⋆) arrow, as outlined in Chapter 3.

Formally, as contributions, this chapter (1) introduces a new notion of probabilistic predicate abstractions and shows how to automatically generate them from a probabilistic program; (2) gives a new soundness relation between abstract and concrete program called *distributional soundness*; (3) shows how distributionally sound abstractions decompose probabilistic inference.

1.1.5 Chapter 6: Conclusion

There is no one-size-fits-all solution to probabilistic inference, so this thesis cannot conclude with a solution that solves all problems. Each new approach to inference opens up a few avenues for applying probabilistic programs in new places. Chapter 3 shows how to apply probabilistic programs effectively in discrete domains that were previously out of reach, Chapter 4 applies them to domains with symmetry, and Chapter 5 shows how to mix and match programs with different kinds of structure. But long term, there remains many deep foundational questions, and this chapter highlights a few of them that will require sustained work.

CHAPTER 2

Foundations

Probability is not really about numbers. It is about the structure of reasoning.

Glenn Shafer

Programming languages and artificial intelligence are two very distinct fields. One of the goals of this chapter is to bring readers from both audiences into the thesis, in order to bring to bear techniques from both perspectives to the problem of probabilistic program inference. This section will begin by laying the foundations of important ideas in probability. Then, it will introduce the core topics in programming languages, and conclude with a discussion specific to probabilistic programs.

A brief note on expected background: this thesis assumes a basic familiarity with the notation and standard concepts from set theory, logic, and computational complexity.[1] Each subsequent chapter will contain its own self-contained background section: the goal of this chapter is to lay broad foundations that stretch across chapters.

2.1 Probabilistic Modeling

We begin with the foundation: probability.

Definition 2.1 (Discrete probability distribution). *Let Ω be a (countably infinite) set called the* sample space, *and let $\mathcal{E}$ be the set of all subsets of Ω called the* event space. *A discrete probability distribution on Ω is a map $\Pr : \mathcal{E} \to [0, 1]$ that assigns a probability to each possible event $E \in \mathcal{E}$ that satisfies the following axioms:*

1. Non-negativity: $\Pr(E) \geq 0$ *for all $E \in \mathcal{E}$.*
2. Unit measure: $\Pr(\Omega) = 1$.
3. Additivity: *For any countable sequence of disjoint events of $\{E_i\}$, it holds that $\Pr(\bigcup_i E_i) = \sum_i \Pr(E_i)$.*

---

[1]Jaynes [2003] has an excellent perspective on probability in the sciences. Pearl [1988] contains a computer-science and artificial intelligence perspective on probabilistic reasoning. Sipser [1996] gives an excellent introduction to computational complexity. Gunter [1992] gives an overview of program semantics and logic.

State (Ω)	⚀	⚁	⚂	⚃	⚄	⚅
Probability	1/6	1/6	1/6	1/6	1/6	1/6

Table 2.1: The distribution on dice rolls.

A central focus of this thesis is various representations of probability distributions, and the algorithmic implications of these data structures:

Definition 2.2 (Probabilistic model (informal)). *Given a set Ω, a probabilistic model M on Ω is a representation of a probability distribution $\Pr : \Omega \to [0, 1]$. We denote the size of the model as $|M|$, for some appropriate definition of size.*

This definition is informal because it hinges on what a "representation" is. We will formalize this notion throughout the thesis, but here we will make things more concrete by considering one of the simplest probabilistic models: a table. A *tabular probabilistic model* simply lists the probability of each element in Ω. For a tabular probabilistic model, $|M| = |\Omega|$. For instance, we can specify the distribution of a fair 6-sided dice roll using Table 2.1.

Given a probabilistic model, the next goal is to *query* it to find out useful information about the world. For instance, a simple query is: *what is the probability that a particular dice roll is even*? Given a tabular representation of a probability distribution, we must use additivity to compute the probability of any event that is not a singleton:

$$\Pr(\text{a dice roll is even}) = \Pr(⚁) + \Pr(⚃) + \Pr(⚅). \tag{2.1}$$

Besides the probabilities of events, one is often interested in the probabilities of event A occurring *given* the presence of some other event B, often referred to as *evidence*. For instance, suppose we want to know the probability that dice roll of ⚁ occurs *given* that the dice roll is even. This computation is known as *the conditional probability of A given B*, denoted $\Pr(A \mid B)$, and is computed as:

$$\Pr(A \mid B) \triangleq \frac{\Pr(A \cap B)}{\Pr(B)}. \tag{2.2}$$

The symbol "$\triangleq$" denotes a definition. In the dice roll example, we would have $A = \{⚁\}$, $B = \{⚁, ⚃, ⚅\}$, so $\Pr(A \mid B) = \frac{1}{3}$. Note that computing the probability of an event is a special case of conditional probability when $B = \Omega$.

One of the main goals of this thesis is automating probabilistic inference, so here we ask the core algorithmic question: *how might we go about performing inference?* As shown earlier, computing the probability of an event requires scanning over the entire table and

```
1  let cold = flip(1/1000) in
2  let flu = flip(1/10000) in
3  let coughing = if(cold || flu) then flip(1/2)
4    else false in
5  let highTemperature = if flu then flip 3/4 else
6    flip 1/100 in
7  let _ = observe coughing in
8  (cold, flu)
```

(a) An example probabilistic program for the medical scenario.
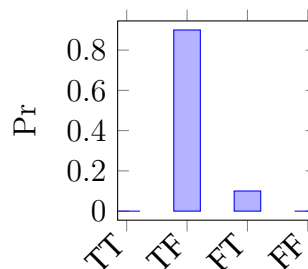
(b) The probability distribution encoded by the program.

Figure 2.1: An example probabilistic program and its distribution.

collecting the total probability of entries that are contained in the event. The run-time of this algorithm is $\mathcal{O}(|M|)$ – linear in the size of the tabular representation – assuming that checking membership in an event can be done in constant time. The problem is that tables are a very inconvenient way to represent probability distributions. Tables are not *concise*: sample spaces, as we will see in later sections, can be prohibitively large – sometimes greater than the number of atoms in the universe – so writing down a table is a hopeless task.

This tradeoff between how concise a model is and how tractable it is for various inference queries is one of the central objects of study in this thesis. In the following sections we will explore other modeling language frameworks that are more concise and expressive than tables, but at the cost of more challenging inference.

2.2 Probabilistic Programming Languages

Consider the following probabilistic scenario where a doctor wishes to model the relationship between symptoms and diseases:

- The average patient has a $1/1000$ chance of having the cold; a $1/10000$ chance of having the flu; a $1/100$ chance of having a high temperature; and does not cough.

- Common colds and the flu both cause coughing with probability $1/2$;

- The flu causes a high temperature with probability $3/4$.

Tables are an inadequate means of specifying the above distribution. They require the doctor to specify the probability of every possible world, which is a very unintuitive task that would not scale to larger systems with more symptoms and diseases. Much more intuitive, from the perspective of the doctor, is to specify the relationship between

symptoms and diseases programmatically: "if the patient has the flu, then half of the time they have a cough".

A *probabilistic programming language* extends a normal programming language with the ability to represent probability distributions. As we will see, this is a very natural way to represent a distribution: one can use the rich facilities of a programming language to describe the nuanced relationship between uncertain events. Concretely, the above scenario can very naturally be modeled as a program in Figure 2.1. This program is written in the Dice probabilistic programming language, which will be formally introduced and elaborated on in subsequent sections and in Chapter 3.

First, the program in Figure 2.1a assigns the identifiers cold and flu to the quantities flip 1/1000 and flip 1/10000 respectively. The syntax flip θ introduces a Boolean random variable that is true with probability θ and false with probability $1 - \theta$: this program defines a probability distribution. In the case of flu and cold, these flips represent the *prior* probability that an average member of the population has either of these two diseases. Then, each of the symptoms – coughing and highTemperature – is assigned to different distributions depending on whether or not the patient has particular diseases. Note that standard programming language constructs, such as assignments and if-statements, can be naturally applied to random variables, enabling the construction of rich distributions.

Given this set of relationships between symptoms and diseases, a doctor would like to know the answer to Bayesian inference query: *what is the probability that a patient with a cough has the flu*? This too can be encoded into the probabilistic program by the addition of an observe construct, which applies Bayesian conditioning to the current program. This is shown on Line 7, where the evidence is introduced via observe coughing, which implicitly rejects all computations that do not satisfy the condition that the patient is coughing. Finally, the program returns the pair (cold, flu), which are the main quantities of interest: the probability distribution on diseases.

Figure 2.1b shows the output of this program. Note that it is not a particular value like a typical program: rather, it is a probability distribution that assigns a probability to every possible value that the program can output. The column "TF" corresponds to the case when cold is true and flu is false: this is the most likely column with a probability of about 0.9. Note that this is markedly different from the prior probability: for a random person, the probability of having a cold is 1/1000, so the presence of a cough increased the probability of a cold by several orders of magnitude.

The above discussion gives an intuitive overview of what probabilistic programs are and how they work, but it is informal. The follow section formalizes the relationship between probabilistic programs and their distributions.

```
1  v ::= T | F
2  aexp ::= x | v
3  e ::= aexp | let x = e in e | flip θ
4        | if aexp then e else e | observe aexp
```

Figure 2.2: A subset of the `Dice` syntax given in Backus-Naur form.

2.2.1 Syntax and Semantics of `Dice`

This section formally defines a fragment of the `Dice` probabilistic programming language. More language features will be introduced in Chapter 3, but this section develops a core subset of the language for simplicity.

The most basic element of a programming language is *syntax*: the formal rules that describe how programs are presented to the system. Syntax is usually defined using a compositional grammar that describes how big programs are made out of smaller programs. The core syntax of `Dice` is given in Figure 2.2, which presents the grammar using a recursive description.

The syntax has three important parts: (1) *values*, denoted v, which are simply either true (T) or false (F); (2) *atomic expressions* aexp which are either values or variable identifiers x; or (3) expressions, denoted e, which are the most important language component. There are two kinds of expressions: probabilistic expressions which create or manipulate distributions (flip θ and observe aexp) and non-probabilistic expressions that are familiar from normal functional programming languages.

2.2.2 Semantics

Once a syntax is defined, the next step is to give a meaning to each syntactic term: this is called *semantics*. Figure 2.3 provides a reference for the semantics for `Dice` expressions. The *semantic function* $[\![\cdot]\!]$ maps syntactic expressions to *unnormalized probability distributions*, which intuitively is a probability distribution that relaxes the unit measure requirement:

Definition 2.3 (Unnormalized distribution). *Let Ω be a sample space. Then a map $\Pr : \Omega \to [0, 1]$ is an* unnormalized probability distribution *on Ω if (1) it satisfies additivity and non-negativity and (2) $\Pr(\Omega) \leq 1$. The quantity $\Pr(\Omega)$ is called the* normalizing constant *and is usually denoted Z.*

The simplest rule is for values. The semantics of values v_1, denoted $[\![v_1]\!]$, is assigned to be equal to the *Dirac delta distribution* $\delta(v_1)$ that assigns probability 1 to v_1 and probability 0 to all other values. The semantics of flip θ produces a probability distribution on the sample space T, F that maps T to θ and F to $1 - \theta$.

11

$$\llbracket \texttt{if } v_g \texttt{ then } \texttt{e}_1 \texttt{ else } \texttt{e}_2 \rrbracket (v) \triangleq \begin{cases} \llbracket \texttt{e}_1 \rrbracket (v) & \text{if } v_g = \texttt{T} \\ \llbracket \texttt{e}_2 \rrbracket (v) & \text{if } v_g = \texttt{F} \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket \texttt{flip } \theta \rrbracket (v) \triangleq \begin{cases} \theta & \text{if } v = \texttt{T} \\ 1 - \theta & \text{if } v = \texttt{F} \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket \texttt{observe } v_1 \rrbracket (v) \triangleq \begin{cases} 1 & \text{if } v_1 = \texttt{T and } v = \texttt{T}, \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket v_1 \rrbracket (v) \triangleq \big(\delta(v_1) \big)(v)$$

$$\llbracket \texttt{let } x = \texttt{e}_1 \texttt{ in } \texttt{e}_2 \rrbracket (v) \triangleq \sum_{v'} \llbracket \texttt{e}_1 \rrbracket (v') \times \llbracket \texttt{e}_2[x \mapsto v'] \rrbracket (v)$$

Figure 2.3: Semantics for $\texttt{Dice}$ 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 most interesting aspect of a probabilistic programming language is that one can neatly combine and manipulate probability distributions using the rules of a programming language. The main workhorse for this capability is composition rules such as the $\texttt{let}$ expression, which has the form $\texttt{let } x = \texttt{e}_1 \texttt{ in } \texttt{e}_2$ and intuitively means "let x take on the value defined by $\texttt{e}_1$ in $\texttt{e}_2$". How do we define this when $\texttt{e}_1$ is a probability distribution?

A simple re-writing of Equation 2.2 shows us that $\Pr(A \cap B) = \Pr(A \mid B) \times \Pr(B)$: this version of the equation is often referred to as the *chain rule of probability*, since it allows one to break a joint distribution into a product of conditional probabilities. We can use the chain rule to give a semantics to $\texttt{let}$:

$$\llbracket \texttt{let } x = \texttt{e}_1 \texttt{ in } \texttt{e}_2 \rrbracket (v) \triangleq \sum_{v'} \llbracket \texttt{e}_1 \rrbracket (v') \times \llbracket \texttt{e}_2[x \mapsto v'] \rrbracket (v) \qquad (2.3)$$

Breaking this down, the syntax $\texttt{e}_2[x \mapsto v']$ is standard programming language syntax that denotes substituting the variable x with the value v' in $\texttt{e}_2$; this can be thought of as a *conditional probability* on values given x is substituted for v', which justifies the use of the chain rule. The following example shows how to apply the semantics of $\texttt{let}$-expressions to a simple program:

Example 2.1: Semantics of a $\texttt{let}$-statement

Consider the following simple program with a free variable x:

$$\texttt{let } x = \texttt{flip } 0.1 \texttt{ in flip } 0.4 \vee x \qquad \text{(EXLET)}$$

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 $[\![\texttt{let } x = \texttt{flip } 0.1 \texttt{ in flip } 0.4 \vee x]\!]\,(\mathrm{T})$, the following sum is computed:

$$[\![\texttt{flip } 0.1]\!]\,(\mathrm{T}) \times [\![\texttt{flip } 0.4 \vee x[x \mapsto \mathrm{T}]]\!]\,(\mathrm{T}) + [\![\texttt{flip } 0.1]\!]\,(\mathrm{F}) \times [\![\texttt{flip } 0.4 \vee x[x \mapsto \mathrm{F}]]\!]\,(\mathrm{T})$$
$$= 0.1 \times 1.0 + 0.9 * 0.4 = 0.46.$$

A-Normal Form The syntax of Dice requires that programs be written in *A-normal form* [Flanagan et al., 1993]. 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.

2.2.2.1 Observations & Bayesian Conditioning

Observations complicate the goal of associating a probability distribution with each program expression. The semantics of observe in Figure 2.3 follows prior work by assigning probability 0 to a failed observation [Borgström et al., 2011, Kozen, 1979, Claret et al., 2013, Huang and Morrisett, 2016, 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
```
$$\text{(OBSPROG)}$$

Because the observe expression is falsified when both x and y are false, that scenario has probability 0. Hence according to our semantics $[\![\text{OBSPROG}]\!]\,(\mathrm{T}) = 0.6$ and $[\![\text{OBSPROG}]\!]\,(\mathrm{F}) = 0.12$. 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 that is normalized at the end of the program to produce a valid probability distribution. Here we explore the subtle properties of this unnormalized distribution, which will serve a crucial purpose later during our compilation strategy. Let $[\![\texttt{e}]\!]_A$ denote the normalizing constant and $[\![\texttt{e}]\!]_D$ denote the normalized distribution for an expression. These two quantities can be straightforwardly computed from the

13

```
1  let x = flip₁ 0.1 in
2  let y = if x then flip₂ 0.2 else flip₃ 0.3 in
3  let z = if y then flip₄ 0.4 else flip₅ 0.5 in z
```

Figure 2.4: Example `Dice` program.

unnormalized semantics in Figure 2.3:

$$\llbracket e \rrbracket_A \triangleq \sum_v \llbracket e \rrbracket (v), \qquad \llbracket e \rrbracket_D (v) \triangleq \frac{1}{\llbracket e \rrbracket_A} \llbracket e \rrbracket (v). \qquad (2.4)$$

For instance, in the above example $\llbracket \text{OBSPROG} \rrbracket_A = 0.12 + 0.6 = 0.72$, $\llbracket \text{OBSPROG} \rrbracket_D (\text{T}) = 0.6/0.72 \approx 0.83$, and $\llbracket \text{OBSPROG} \rrbracket_D (\text{F}) = 0.12/0.72 \approx 0.17$. 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 [Fierens et al., 2015, Claret et al., 2013]. Later in Section 3.3.1.2 we will describe the approach `Dice` takes for handling observations in more detail once more language features are introduced. Ultimately we will show that this normalizing constant must be carefully constructed in order to give a compositional semantics to programs; the utility of this will become clear when we introduce functions.

2.3 Probabilistic Program Inference

Now that each `Dice` program has a semantics it is time to perform probabilistic inference: computing the probability that the program outputs a particular value. The semantics given in the previous section already give a recipe for how to perform inference: each program is associated with a sum of products over all possible assignments to `flips` in the program: performing inference in this way is called *path-enumeration inference* and is a common strategy for performing exact inference on probabilistic programs in the literature [Sankaranarayanan et al., 2013, Albarghouthi et al., 2017, Geldenhuys et al., 2012, Filieri et al., 2013].

Consider the example `Dice` program in Figure 3.2. The subscript on each `flip` is not part of the syntax but rather used to refer to them uniquely in our discussion. Path-

enumeration on this program would is given by the following sum of products:

$$\underbrace{0.1}_{\text{x=T}} \cdot \underbrace{0.2}_{\text{y=T}} \cdot \underbrace{0.4}_{\text{z=T}} \quad + \quad \underbrace{0.1}_{\text{x=T}} \cdot \underbrace{0.8}_{\text{y=F}} \cdot \underbrace{0.5}_{\text{z=T}} \quad + \quad \underbrace{0.9}_{\text{x=F}} \cdot \underbrace{0.3}_{\text{y=T}} \cdot \underbrace{0.4}_{\text{z=T}} \quad + \quad \underbrace{0.9}_{\text{x=F}} \cdot \underbrace{0.7}_{\text{y=F}} \cdot \underbrace{0.5}_{\text{z=T}} \quad (2.5)$$

How does exhaustive enumeration scale as this program grows in size? For this example the program grows by adding one additional layer to the chain of `flip`s 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.

It will be shown show later (Theorem 3.3) that exact inference in `Dice` is PSPACE-hard in the size of the program. Hence, there is an important difference between tabular representations of distributions and probabilistic programs: it is possible to write a small probabilistic program for which inference is computationally intractable. Hence, probabilistic programs are more concise than tables – you can write a small program that encodes a distribution on a very large sample space – but inference is not in general tractable. This leads to the fundamental tradeoff in probabilistic modeling languages, which will be a central topic in this thesis:

> **The Fundamental Tradeoff Between Tractability and Conciseness**: A key design decision when creating probabilistic modeling languages is the fundamental tradeoff between *tractability* of inference and the *conciseness* of the representation.

This does not mean that inference in `Dice` is in general hopeless: worst-case performance is different from the common case or average case. However, the presence of worst-case programs does shape the design of inference algorithms in fundamental ways that will be explored in later chapters. In particular, this implies that *there is no universal solution to fast inference in probabilistic programs*: inference must, at its core, take advantage of the delicate structure that is unique to the programs that users write in practice.

2.4 Conclusion

This chapter introduced a number of important concepts that will be revisited in subsequent chapters. The key important definitions such as a probability distribution and probabilistic models will be used repeatedly. The tradeoff between tractability and conciseness is the root of the motivation for why probabilistic program inference is a hard and important problem. The semantics of `Dice` and basic notions of probabilistic programs will be especially important in Chapter 3. And finally, the basics of probabilistic inference will motivate our further explorations in Chapter 4.

CHAPTER 3

`Dice`: Exploiting Factorization

One of the most challenging kinds of probabilistic programs for many existing probabilistic programming systems are *discrete programs*: programs that contain discrete random variables. This chapter develops 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 it to perform exact inference on probabilistic programs with hundreds of thousands of random variables.

The 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. In sum, this chapter (1) shows how to compositionally reduce `Dice` inference to WMC, (2) proves this compilation correct with respect to a denotational semantics, (3) empirically demonstrates the performance benefits over prior approaches, and (4) analyzes the types of structure that allow `Dice` to scale to large probabilistic programs.

3.1 Introduction

As we have seen in prior chapters, 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.

This chapter scales inference for an important class of probabilistic programs: those whose probability distributions are *discrete*. Most PPLs today focus on handling continuous random variables. In the continuous setting one usually desires approximate inference techniques, such as forms of sampling [Wingate and Weber, 2013, Kucukelbir et al., 2015, Jordan et al., 1999, Bingham et al., 2019, Dillon et al., 2017, Carpenter et al., 2016, Nori et al., 2014, Chaganty et al., 2013]. However, handling continuous variables
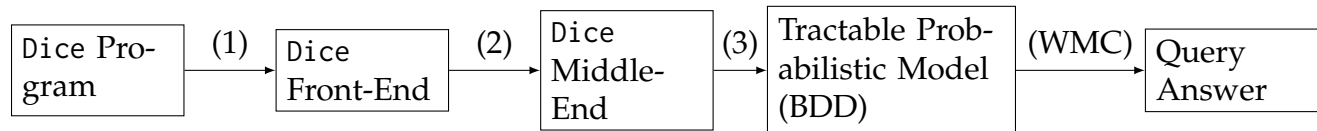
Figure 3.1: System diagram for performing a query on a `Dice` program.

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 [Obermeyer et al., 2019, Gorinova et al., 2020, Zhou et al., 2020].

Discrete programs are not a new challenge, and there are existing PPLs that support exact inference for discrete probabilistic programs [Narayanan et al., 2016, Gehr et al., 2016, Sankaranarayanan et al., 2013, Albarghouthi et al., 2017, Goodman and Stuhlmüller, 2014, Wang et al., 2018, Claret et al., 2013, Pfeffer, 2007a, Bingham et al., 2019, Geldenhuys et al., 2012]. However, there are 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 Chapter 3.5 show `Dice` performing exact inference on a real-world probabilistic program that is 1.9MB large.

An outline of `Dice` is given in Figure 3.1. At its core, `Dice` builds on the *knowledge compilation* approach to probabilistic inference [Chavira and Darwiche, 2008, 2005, Darwiche, 2009, Fierens et al., 2015, Chavira et al., 2006]. This chapter shows how to compile `Dice` programs to *weighted Boolean formulas* (WBF) and then perform exact inference via *weighted model counting* (WMC) on those formulas. `Dice` programs are parsed (1) and translated into an intermediate representation (2). Then, they are compiled to a tractable

representation (3) that supports efficient WMC and hence inference. This tractable representation is a *binary decision diagram* (BDD) that supports efficient weighted model counting, described in more detail later.

Employing knowledge compilation for probabilistic inference in `Dice` requires generalizing 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, novel compilation rules that compositionally associate `Dice` programs with weighted Boolean formulas are developed. 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. Performing WMC on these formulas then performs exact Bayesian inference with arbitrary observations throughout the program. Second, `Dice` compiles functions *modularly*: each function is compiled to a BDD once, and efficient BDD composition operations are exploited to reuse this BDD at each call site. This technique produces the same final BDD that would otherwise be produced, but it allows amortizing the costly BDD construction phase across all callers, which can provide orders-of-magnitude speedups.

In sum, this chapter presents the following technical contributions:

- It describes the `Dice` language and illustrate its utility through three motivating examples (Chapter 3.2).

- It formalizes `Dice`'s semantics (Chapter 3.3) and its compilation to weighted Boolean formulas (Chapter 3.4). It proves 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.

- It empirically compares `Dice`'s performance to that of prior PPLs with exact inference (Chapter 3.5). It describes 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.

- It analyzes some of the benefits of `Dice`'s compilation strategy in Chapter 3.6. First it proves that `Dice` inference is **PSPACE**-hard. Then it characterizes cases where `Dice` scales efficiently, and which types of structure it exploits in the distribution. It illustrates where to find that structure in the program code as well as the compiled BDD form. Finally these results are used to provide a technical comparison with prior exact inference algorithms.

`Dice` is available at `https://github.com/SHoltzen/dice`.

```
1  let x = flip₁ 0.1 in
2  let y = if x then flip₂ 0.2 else
3      flip₃ 0.3 in
4  let z = if y then flip₄ 0.4 else
5      flip₅ 0.5 in z
```

(a) Example `Dice` program.

(b) Compiled BDD with weighted model counts.
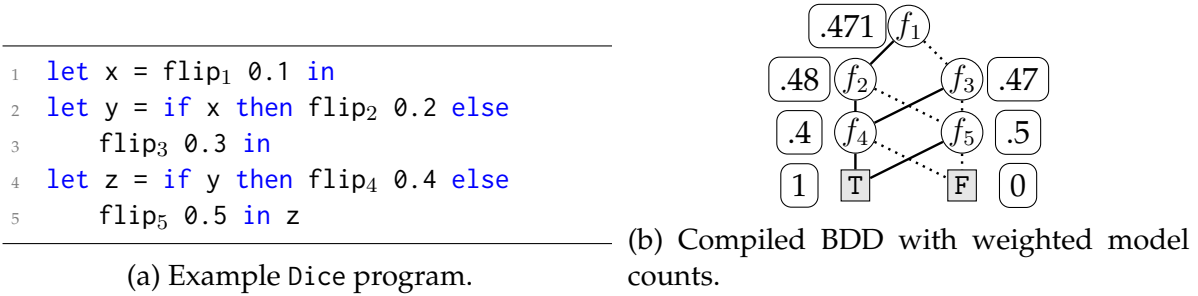
Figure 3.2: Illustration of compiling a `Dice` program that exploits factorization.

3.2 An Overview of Dice

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

3.2.1 Factorizing Inference

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 3.2a, reproduced from Chapter 2. 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 3.2a 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 [Sankaranarayanan et al., 2013, Albarghouthi et al., 2017, Geldenhuys et al., 2012, Filieri et al., 2013]. Concretely this would involve computing the following sum of products:

$$\underbrace{0.1}_{x=T} \cdot \underbrace{0.2}_{y=T} \cdot \underbrace{0.4}_{z=T} \; + \; \underbrace{0.1}_{x=T} \cdot \underbrace{0.8}_{y=F} \cdot \underbrace{0.5}_{z=T} \; + \; \underbrace{0.9}_{x=F} \cdot \underbrace{0.3}_{y=T} \cdot \underbrace{0.4}_{z=T} \; + \; \underbrace{0.9}_{x=F} \cdot \underbrace{0.7}_{y=F} \cdot \underbrace{0.5}_{z=T} \tag{3.1}$$

This thesis focuses 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

19

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 3.11d in Chapter 3.5.

However, the sum in Equation 3.1 has redundant computation, and thus can be factorized as:

$$\underbrace{0.1}_{x=T} \cdot \left(\underbrace{0.2}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.8}_{y=F} \cdot \underbrace{0.5}_{z=T} \right) + \underbrace{0.9}_{x=F} \cdot \left(\underbrace{0.3}_{y=T} \cdot \underbrace{0.4}_{z=T} + \underbrace{0.7}_{y=F} \cdot \underbrace{0.5}_{z=T} \right). \qquad (3.2)$$

Such factorizations are abundant in this example, and in many others. `Dice` exploits these factorizations to scale, and in Chapter 3.5 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 [Chavira and Darwiche, 2008, Darwiche, 2009, Koller and Friedman, 2009a, Boutilier et al., 1996, Pearl, 1988].

3.2.2 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 3.1 (Weighted Boolean Formula). *Let φ be a Boolean formula over variables X, let L be the set of all* literals *(assignments to variables) over X, and $w : L \to \mathbb{R}$ 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 3.2a into a WBF, introduce one Boolean variable f_i for each expression $\text{flip}_i \ \theta$ in the program. The 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, probabilistic inference is performed 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 3.2 (Weighted Model Count). *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,* $\texttt{WMC}(\varphi, w) \triangleq \sum_{\omega \in \texttt{Mods}(\varphi)} w(\omega)$.

What has been achieved? So far, not much! The WMC task is known to be #P-hard for arbitrary Boolean formulas. Indeed, the formula φ_{ex} above is isomorphic to the structure of Equation 3.1, 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 [Chavira and Darwiche, 2008, Darwiche and Marquis, 2002, Bryant, 1986]. Formally, as outlined in Chapter 2, BDDs are a *tractable probabilistic model*.

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 3.2b shows the compiled BDD for the program in Figure 3.2a. 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 3.2b 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 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 3.2.

3.2.3 Leveraging Functional Abstraction

```
1   fun diamond(s₁:Bool):Bool {
2     let route = flip₁ 0.5 in
3     let s₂ = if route then s₁ else F
          in
4     let s₃ = if route then F else s₁
          in
5     let drop = flip₂ 0.0001 in
6     s₂ ∨ (s₃ ∧ ¬drop)
7   }
8   let net1 = diamond(T) in
9   let net2 = diamond(net1) in
10  diamond(net2)
```

(a) Network diagram.

(b) Dice program.

(c) diamond function.
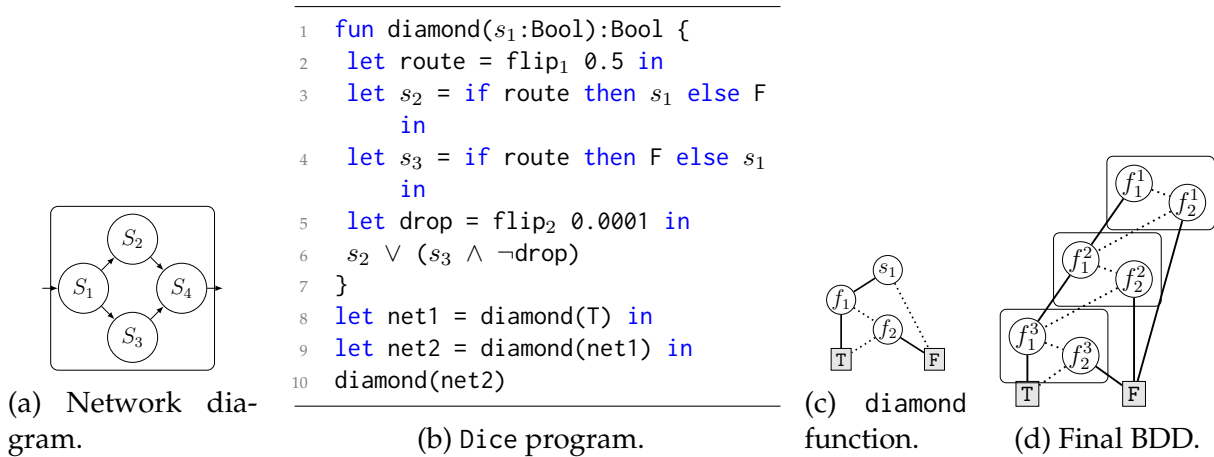
(d) Final BDD.

Figure 3.3: A sub-network, its description as a probabilistic program, a compiled function, and the final BDD.

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, consider an example from recent work in probabilistic verification of computer networks via probabilistic programs [Gehr et al., 2018]. Figure 3.3a 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 3.3b 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 3.3c. 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 3.3b. During this process, the BDD for the diamond function is reused at each call site using standard BDD composition operations like conjunction (Chapter 3.4

22

```
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
```

Figure 3.4: A frequency analyzer for a noisy Caesar cipher.

describes this in more detail). The final BDD for the program is shown in Figure 3.3d, where each variable f_i^j represents the ith flip in the jth 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. Chapter 3.5 shows that these capabilities provide orders of magnitude speedups in inference.

3.2.4 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 3.4, 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 3.4, 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 Chapter 3.5).

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, the compilation strategy is augmented to produce a second logical formula, which is called the *accepting formula* and denoted 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 $\mathtt{WMC}(\varphi \wedge \gamma, w)/\mathtt{WMC}(\gamma, w)$ 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.

3.3 The Dice Language

Chapter 2 gave an introduction to a small subset of the complete Dice language; here

```
1   τ ::= Bool | τ₁ × τ₂
2   v ::= T | F | (v, v)
3   aexp ::= x | v
4   e ::= aexp | fst aexp | snd aexp | (aexp, aexp) | let x = e in e | flip θ
5         | if aexp then e else e | observe aexp | f(aexp)
6   func ::= fun f(x:τ): τ { e }
7   p ::= e | func p
```

Figure 3.5: 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 $[0, 1]$.

extra language constructs like tuples and functions are introduced.

3.3.1 Semantics

Recall from Chapter 2.2.1 that the semantic bracket $[\![\cdot]\!]$ associates each `Dice` expression with an unnormalized distribution, and further denote the set of all `Dice` values as V. Figure 3.6 provides the full semantics for `Dice` expressions. 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.

3.3.1.1 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 $[\![\text{func}]\!] : V \to V \to [0, 1]$ is defined as follows:

$$[\![\text{fun } f(x : \tau) : \tau'\{e\}]\!] (v) \triangleq [\![e[x \mapsto v]]\!] \tag{3.3}$$

To give a semantics for function calls the semantic judgment is extended 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 $[\![e]\!]^T : V \to [0, 1]$, and similarly for the semantics of function definitions above, but we leave T implicit when it is clear from the context. Figure 3.6 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 $[\![p]\!]^T : V \to [0, 1]$. Intuitively, each function is given a semantics in the context of the prior functions, and then the semantics of

$$\llbracket v_1 \rrbracket \, (v) \triangleq \big(\delta(v_1)\big)(v) \qquad \llbracket \texttt{fst}\ (v_1, v_2) \rrbracket \, (v) \triangleq \big(\delta(v_1)\big)(v) \qquad \llbracket \texttt{snd}\ (v_1, v_2) \rrbracket \, (v) \triangleq \big(\delta(v_2)\big)(v)$$

$$\llbracket \texttt{if}\ v_g\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2 \rrbracket \, (v) \triangleq \begin{cases} \llbracket \texttt{e}_1 \rrbracket \, (v) & \text{if } v_g = \texttt{T} \\ \llbracket \texttt{e}_2 \rrbracket \, (v) & \text{if } v_g = \texttt{F} \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket \texttt{flip}\ \theta \rrbracket \, (v) \triangleq \begin{cases} \theta & \text{if } v = \texttt{T} \\ 1 - \theta & \text{if } v = \texttt{F} \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket \texttt{observe}\ v_1 \rrbracket \, (v) \triangleq \begin{cases} 1 & \text{if } v_1 = \texttt{T} \text{ and } v = \texttt{T}, \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket f(v_1) \rrbracket \, (v) \triangleq \Big(\big(T(f)\big)(v_1)\Big)(v)$$

$$\llbracket \texttt{let}\ x = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 \rrbracket \, (v) \triangleq \sum_{v'} \llbracket \texttt{e}_1 \rrbracket \, (v') \times \llbracket \texttt{e}_2[x \mapsto v'] \rrbracket \, (v)$$

Figure 3.6: Full 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.

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 $\eta(\texttt{func})$ denotes the name of the function func:

$$\llbracket \bullet\ \texttt{e} \rrbracket^T \triangleq \llbracket \texttt{e} \rrbracket^T \qquad \llbracket \texttt{func}\ \texttt{p} \rrbracket^T \triangleq \llbracket \texttt{p} \rrbracket^{T \cup \left\{ \eta(\texttt{func}) \mapsto \llbracket \texttt{func} \rrbracket^T \right\}}. \tag{3.4}$$

3.3.1.2   Semantics of Observation

Now that Dice functions have been introduced we are ready to discuss the subtle compositional semantics of observations. 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 }   (3.5)
fun g(x:Bool):Bool { true }                                              (3.6)
```

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,

$$[\![\texttt{let x = flip 0.1 in let obs = f(x) in x}]\!]_D\,(\texttt{T}) = 0.1/0.55$$
$$[\![\texttt{let x = flip 0.1 in let obs = g(x) in x}]\!]_D\,(\texttt{T}) = 0.1$$

The quantity $[\![\cdot]\!]_A$ carries exactly the information needed to distinguish these functions. Specifically, $[\![\texttt{e}]\!]_A$ represents the probability that $\texttt{e}$ has an *accepting* execution, which satisfies all observations, so we call it the *accepting semantics*. In the above example, $[\![\texttt{g(F)}]\!]_A = 1$ but $[\![\texttt{f(F)}]\!]_A = 0.5$: the function call $\texttt{f(F)}$ 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 3.6 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.

3.4 Probabilistic Inference for `Dice`

This section formalizes the approach to probablistic 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 Suciu, 2017]. However, to our knowledge it has not been previously applied to a probablistic programming language with traditional programming language constructs, functions, and first-class observations.

The bulk of this section formalizes a novel algorithm for compiling `Dice` programs to WBF. This compilation is introduced in stages: first on the Boolean sub-language, then with the addition of tuples, and finally with the addition of functions. Along the way a correctness theorem is stated that formally relates WMC queries over a program's compiled WBF to the semantics from the previous section. Finally I illustrate how to use BDDs to represent WBFs, which enables the approach to automatically perform factorized inference.

3.4.1 A Primer on Logical Notation

This part of the thesis makes heavy use of *logical notation* which may be unfamiliar to some readers. Formally, *inference rules* are composed of *premises*, written on top of a bar, and *conclusions* written below the bar. If there are multiple premises, they are often separated

$$\frac{}{\mathtt{T} \rightsquigarrow (\mathrm{T}, \mathrm{T}, \emptyset)} \text{ (C-TRUE)} \qquad \frac{}{\mathtt{F} \rightsquigarrow (\mathrm{F}, \mathrm{T}, \emptyset)} \text{ (C-FALSE)} \qquad \frac{}{x \rightsquigarrow (\mathbf{x}, \mathrm{T}, \emptyset)} \text{ (C-IDENT)}$$

$$\frac{\text{fresh } \mathbf{f}}{\mathtt{flip}\, \theta \rightsquigarrow \left(\mathbf{f}, \mathrm{T}, (\mathbf{f} \mapsto \theta, \mathrm{T}, \bar{\mathbf{f}} \mapsto 1 - \theta)\right)} \text{ (C-FLIP)} \qquad \frac{\mathtt{aexp} \rightsquigarrow (\varphi, \mathrm{T}, \emptyset)}{\mathtt{observe\ aexp} \rightsquigarrow (\mathrm{T}, \varphi, \emptyset)} \text{ (C-OBS)}$$

$$\frac{\mathtt{aexp} \rightsquigarrow (\varphi_g, \mathrm{T}, \emptyset) \qquad \mathtt{e}_T \rightsquigarrow (\varphi_T, \gamma_T, w_T) \qquad \mathtt{e}_E \rightsquigarrow (\varphi_E, \gamma_E, w_E)}{\mathtt{if\ aexp\ then\ e}_T\ \mathtt{else\ e}_E \rightsquigarrow \left(\left((\varphi_g \wedge \varphi_T) \vee ((\overline{\varphi}_g \wedge \varphi_E), \left((\varphi_g \wedge \gamma_T) \vee ((\overline{\varphi}_g \wedge \gamma_E), w_T \cup w_E\right)\right.}$$
$$\text{(C-ITE)}$$

$$\frac{\mathtt{e}_1 \rightsquigarrow (\varphi_1, \gamma_1, w_1) \qquad \mathtt{e}_2 \rightsquigarrow (\varphi_2, \gamma_2, w_2)}{\mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \rightsquigarrow \left(\varphi_2[\mathbf{x} \mapsto \varphi_1], \gamma_1 \wedge \gamma_2[\mathbf{x} \mapsto \varphi_1], w_1 \cup w_2\right)} \text{ (C-LET)}$$

Figure 3.7: Compiling Boolean expressions to WBFs.

by a space. For instance, the well-known rule *modus ponens* which says that "if P implies Q and P is true, then Q must be true" can be written as the following inference rule:

$$\frac{P \Rightarrow Q \qquad P}{Q} \qquad\qquad \text{(MODUS PONENS)}$$

Another commonly-used notational convention for logic is *sequent notation*, which makes use of the *turnstyle operator* "⊢". Equations to the left of the turnstyle are premises and to the right are conclusions, so *modus ponens* can again be written using sequent notation:

$$P \Rightarrow Q, \ \ P \vdash Q. \qquad\qquad (3.7)$$

3.4.2 Compiling Boolean `Dice` Expressions

The formal compilation judgment for Boolean `Dice` expressions has the form $\mathtt{e} \rightsquigarrow (\varphi, \gamma, w)$, where φ and γ are logical formulas and w is a weight function (recall Definition 3.1). This judgment form will be extended later to accommodate other language features. The symbol φ is called the *unnormalized formula*: it represents all possible assignments to variables

and `flips` for which e evaluates to true, ignoring observations. The symbol γ is 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, here are two examples to build intuition on the compilation to WBF and how it is used to perform inference.

Example 3.1: Compiling (EXLET)

Recall the following example program:

$$\texttt{let } x = \texttt{flip } 0.1 \texttt{ in flip } 0.4 \vee x \qquad \text{(EXLET)}$$

The above expression 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, $\gamma = \texttt{T}$ 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 $[\![\text{EXLET}]\!](\texttt{T}) = \texttt{WMC}(\varphi, w) = 0.46$ and $[\![\text{EXLET}]\!](\texttt{F}) = \texttt{WMC}(\overline{\varphi}, w) = 0.54$.

Example 3.2: Compiling (OBSPROG)

Recall the following example program:

```
let x = flip 0.6 in let y = flip 0.3 in let _ = observe x ∨ y in x
```
$$\text{(OBSPROG)}$$

The above program 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 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 $\overline{\varphi} \wedge \gamma$ is true if and only if the program evaluates to F and satisfies all observations. Then, with the appropriate weight function w, Bayesian inference on (OBSPROG) is performed via two weighted model counts: $[\![(\text{OBSPROG})]\!]_D(\texttt{T}) = \texttt{WMC}(\varphi \wedge \gamma, w)/\texttt{WMC}(\gamma, w) \approx 0.83$ and $[\![(\text{OBSPROG})]\!]_D(\texttt{F}) = \texttt{WMC}(\overline{\varphi} \wedge \gamma, w)/\texttt{WMC}(\gamma, w) \approx 0.17$.

The formal compilation rules are shown in Figure 3.7. 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` $0.4 \rightsquigarrow (f, \texttt{T}, w)$ where w maps f to 0.4 and $\bar{f}$ to 0.6, and f is a fresh Boolean variable. Hence $\texttt{WMC}(f \wedge \texttt{T}, w) = 0.4 = [\![\texttt{flip } 0.4]\!](\texttt{T})$ and $\texttt{WMC}(\bar{f}, w) = 0.6 = [\![\texttt{flip } 0.4]\!](\texttt{F})$.

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* $\varphi_1[\mathbf{x} \mapsto \varphi_2]$ replaces all occurrences of $\mathbf{x}$ 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 their accepting formulas are simply conjoined. To illustrate the rule, here is 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):

$$
\cfrac{
\cfrac{\text{fresh } f_1}{\text{flip } 0.1 \rightsquigarrow (f_1, \mathsf{T}, w_1)}
\qquad
\cfrac{
\cfrac{}{x \rightsquigarrow (\mathbf{x}, \mathsf{T}, \emptyset)}
\qquad
\cfrac{\text{fresh } f_2}{\text{flip } 0.4 \rightsquigarrow (f_2, \mathsf{T}, w_2)}
}{\text{flip } 0.4 \vee x \rightsquigarrow (f_2 \vee \mathbf{x}, \mathsf{T}, w_2)}
}{\text{let } x = \text{flip } 0.1 \text{ in flip } 0.4 \vee x \rightsquigarrow (f_2 \vee \mathbf{x}[\mathbf{x} \mapsto f_1], \mathsf{T}, w_1 \cup w_2)} \text{ (ExLETCOMPILATION)}
$$

This compilation matches Example 3.1 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 3.1 (Boolean Expression Correctness). *Let* e *be a Boolean* Dice *expression with free variables* $x_1, \ldots, x_n$ *and suppose* e $\rightsquigarrow (\varphi, \gamma, w)$. *Then for any Boolean values* $v_1, \ldots, v_n$:

- $[\![e[x_i \mapsto v_i]]\!]_A = \mathtt{WMC}(\gamma[\mathbf{x}_i \mapsto v_i], w)$

- *for any Boolean value* v,

$$
[\![e[x_i \mapsto v_i]]\!]_D (v) = \frac{\mathtt{WMC}(((\varphi \Leftrightarrow v) \wedge \gamma)[\mathbf{x}_i \mapsto v_i], w)}{\mathtt{WMC}(\gamma[\mathbf{x}_i \mapsto v_i], w)}.
$$

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

$$
[\![e[x_i \mapsto v_i]]\!] (v) = \mathtt{WMC}(((\varphi \Leftrightarrow v) \wedge \gamma)[\mathbf{x}_i \mapsto v_i], w).
$$

30

3.4.3 Tuples & Typed Compilation

Now the compilation rules are extended 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 generalizing the compilation judgment, which now has the following form:

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

First, 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:

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

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

Figure 3.8 shows the new rules for compiling tuples and also presents updated versions of the rules from Figure 3.7, 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 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 $F_{\textbf{Bool}}(x) \triangleq x$ 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 for defining the compilation for tuple creation as well as fst and snd in Figure 3.8.

The C-ITE rule shows how to generalize the compilation of conditionals to accommodate tuples. The rule requires conjoining a Boolean expression φ_g (the compiled guard) with a potential tuple of formulas (the compiled then and else branches). To do this, conjunction must be generalized to *broadcasted conjunction*, denoted $\varphi_g \underset{\tau}{\wedge} \dot{\varphi}$, by conjoining φ_g with all the Boolean expressions in the tuple $\dot{\varphi}$. Formally, it is defined inductively as:

- $\varphi_a \underset{\textbf{Bool}}{\wedge} \varphi_b \triangleq \varphi_a \wedge \varphi_b$

- $\varphi_a \underset{\tau_1 \times \tau_2}{\wedge} (\dot{\varphi}_{b1}, \dot{\varphi}_{b2}) \triangleq \left(\varphi_a \underset{\tau_1}{\wedge} \dot{\varphi}_{b1}, \ \varphi_a \underset{\tau_2}{\wedge} \dot{\varphi}_{b2} \right).$

In addition to broadcasted conjunction, C-ITE also requires *point-wise disjunction*, denoted $\dot{\varphi}_1 \underset{\tau}{\vee} \dot{\varphi}_2$. Point-wise disjunction is nearly identically defined inductively as:

31

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \leadsto (F_\tau(x), \mathtt{T}, \emptyset)} \text{ (C-IDENT)} \quad \frac{\Gamma(x_1) = \tau_1 \quad \Gamma(x_2) = \tau_2}{\Gamma \vdash (x_1, x_2) : \tau_1 \times \tau_2 \leadsto ((F_{\tau_1}(x_1), F_{\tau_2}(x_2)), \mathtt{T}, \emptyset)} \text{ (C-TUP)}$$

$$\frac{\Gamma(x) = \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ x : \tau_1 \leadsto (F_{\tau_1}(x_l), \mathtt{T}, \emptyset)} \text{ (C-FST)} \quad \frac{\Gamma(x) = \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ x : \tau_2 \leadsto (F_{\tau_2}(x_r), \mathtt{T}, \emptyset)} \text{ (C-SND)}$$

$$\frac{\Gamma \vdash \mathsf{aexp} : \mathbf{Bool} \leadsto (\varphi_g, \mathtt{T}, \emptyset) \quad \Gamma \vdash \mathsf{e}_T : \tau \leadsto (\dot{\varphi}_T, \gamma_T, w_T) \quad \Gamma \vdash \mathsf{e}_E : \tau \leadsto (\dot{\varphi}_E, \gamma_E, w_E)}{\Gamma \vdash \mathsf{if}\ \mathsf{aexp}\ \mathsf{then}\ \mathsf{e}_T\ \mathsf{else}\ \mathsf{e}_E : \tau \leadsto \left(\left((\varphi_g \underset{\tau}{\wedge} \dot{\varphi}_T) \underset{\tau}{\dot{\vee}} (\overline{\varphi}_g \underset{\tau}{\wedge} \dot{\varphi}_E) \right), \left((\varphi_g \wedge \gamma_T) \vee (\overline{\varphi}_g \wedge \gamma_E), w_T \cup w_E \right) \right)}$$
$$\text{(C-ITE)}$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 \leadsto (\dot{\varphi}_1, \gamma_1, w_1) \quad \Gamma \cup \{x : \tau_1\} \vdash \mathsf{e}_2 : \tau_2 \leadsto (\dot{\varphi}_2, \gamma_2, w_2)}{\Gamma \vdash \mathsf{let}\ x : \tau_1 = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \tau_2 \leadsto \left(\dot{\varphi}_2[\mathbf{x} \overset{\tau}{\mapsto} \dot{\varphi}_1], \gamma_1 \wedge \gamma_2[\mathbf{x} \overset{\tau}{\mapsto} \dot{\varphi}_1], w_1 \cup w_2 \right)} \text{ (C-LET)}$$

Figure 3.8: 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.

- $\varphi_1 \underset{\mathbf{Bool}}{\dot{\vee}} \varphi_2 \triangleq \varphi_1 \vee \varphi_2$ and

- $(\dot{\varphi}_{11}, \dot{\varphi}_{12}) \underset{\tau_1 \times \tau_2}{\dot{\vee}} (\dot{\varphi}_{21}, \dot{\varphi}_{22}) \triangleq (\dot{\varphi}_{11} \underset{\tau_1}{\dot{\vee}} \dot{\varphi}_{21}, \dot{\varphi}_{12} \underset{\tau_2}{\dot{\vee}} \dot{\varphi}_{22})$.

Finally, to generalize the compilation of let expressions, the C-LET rule employs a generalized version of substitution called *typed substitution* $\dot{\varphi}_2[\mathbf{x} \overset{\tau_1}{\mapsto} \dot{\varphi}_1]$ that substitutes the compiled version of e_1 into the compiled version of e_2. Typed substitution inductively as follows:

$$\varphi_2[\mathbf{x} \overset{\mathbf{Bool}}{\longmapsto} \varphi_1] \triangleq \varphi_2[\mathbf{x} \mapsto \varphi_1], \quad \varphi_2[\mathbf{x} \overset{\tau_a \times \tau_b}{\longmapsto} (\dot{\varphi}_a, \dot{\varphi}_b)] \triangleq \varphi_2[\mathbf{x}_l \overset{\tau_a}{\mapsto} \dot{\varphi}_a][\mathbf{x}_r \overset{\tau_b}{\mapsto} \dot{\varphi}_b],$$

$$(\dot{\varphi}_1, \dot{\varphi}_2)[\mathbf{x} \overset{\tau}{\mapsto} \dot{\varphi}] \triangleq (\dot{\varphi}_1[\mathbf{x} \overset{\tau}{\mapsto} \dot{\varphi}], \dot{\varphi}_2[\mathbf{x} \overset{\tau}{\mapsto} \dot{\varphi}]).$$

We can state and prove a natural generalization of our key lemma from the previous subsection, Lemma 3.2. The lemma depends on *pointwise iff*, denoted $\dot{\varphi}_1 \overset{\tau}{\Longleftrightarrow} \dot{\varphi}_2$ and defined inductively as follows: $\varphi_1 \overset{\mathbf{Bool}}{\Longleftrightarrow} \varphi_2 \triangleq \varphi_1 \Leftrightarrow \varphi_2$ and $(\dot{\varphi}_1, \dot{\varphi}_2) \overset{\tau_1 \times \tau_2}{\Longleftrightarrow} (\dot{\varphi}'_1, \dot{\varphi}'_2) \triangleq$ $\left(\dot{\varphi}_1 \overset{\tau_1}{\Longleftrightarrow} \dot{\varphi}'_1 \right) \wedge \left(\dot{\varphi}_2 \overset{\tau_2}{\Longleftrightarrow} \dot{\varphi}'_2 \right)$. Finally the following key correctness lemma can be stated:

$$\frac{\Gamma \cup \{x_1 : \tau_1\}, \Phi \vdash \mathsf{e} : \tau_2 \rightsquigarrow (\dot\varphi, \gamma, w)}{\Gamma, \Phi \vdash \mathsf{fun}\ f(x_1 : \tau_1) : \tau_2\ \{\mathsf{e}\} \rightsquigarrow (\dot\varphi, \gamma, w)}\ \text{(C-FUNC)} \qquad \frac{\Gamma, \Phi \vdash \mathsf{e} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)}{\Gamma, \Phi \vdash \bullet\ \mathsf{e} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)}\ \text{(C-PROG1)}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash \mathsf{fun}\ f(x_1 : \tau_1) : \tau_2\ \{\mathsf{e}\} \rightsquigarrow (\dot\varphi_f, \gamma_f, w_f) \\[4pt] \Gamma \cup \{f \mapsto \tau_1 \to \tau_2\}, \Phi \cup \{f \mapsto (\mathbf{x}_1, \dot\varphi_f, \gamma_f, w_f)\} \vdash \mathsf{p} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)\end{array}}{\Gamma, \Phi \vdash \mathsf{fun}\ f(x_1 : \tau_1) : \tau_2\ \{\mathsf{e}\}\ \mathsf{p} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)}\ \text{(C-PROG2)}$$

$$\frac{\begin{array}{c}\Gamma(f) = \tau_1 \to \tau_2 \qquad \Gamma(x_1) = \tau_1 \\[4pt] \Phi(f) = (\mathbf{x}_{arg}, \dot\varphi, \gamma, w) \qquad (\dot\varphi', \gamma', w') = \mathtt{RefreshFlips}(\mathbf{x}_{arg}, \dot\varphi, \gamma, w)\end{array}}{\Gamma, \Phi \vdash f(x_1) : \tau_2 \rightsquigarrow (\dot\varphi'[\mathbf{x}_{arg} \overset{\tau_1}{\longmapsto} \mathbf{x}_1], \gamma'[\mathbf{x}_{arg} \overset{\tau_1}{\longmapsto} \mathbf{x}_1], w')}\ \text{(C-FUNCCALL)}$$

Figure 3.9: Compiling functions and programs. These assume without loss of generality but for simplicity that function calls are only ever given identifiers as arguments.

Lemma 3.3 (Typed Correctness Without Functions). *Let e be a* `Dice` *expression without function calls, and suppose $\{x_i : \tau_i\} \vdash \mathsf{e} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$. Then for any values $\{v_i : \tau_i\}$ and $v : \tau$, we have that $[\![\mathsf{e}[x_i \mapsto v_i]]\!]\,(v) = \mathtt{WMC}\left(\left((\dot\varphi \overset{\tau}{\Leftrightarrow} v) \wedge \gamma\right)[\mathbf{x}_i \overset{\tau_i}{\longmapsto} v_i], w\right).$*

3.4.4 Functions & Programs

We conclude the development of `Dice` compilation by introducing functions and programs in Figure 3.9. This requires introducing 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 $(\mathbf{x}_{arg}, \dot\varphi, \gamma, w)$ where $\mathbf{x}_{arg}$ 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 $\Gamma, \Phi \vdash \mathsf{func} \rightsquigarrow (\dot\varphi, \gamma, w)$ compiles function definitions. As shown in C-FUNC, the function's body is simply compiled in an appropriate type environment. The judgment $\Gamma, \Phi \vdash \mathsf{p} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$ 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 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 $\Gamma, \Phi \vdash \mathsf{e} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$, and C-FUNCCALL shows the rule for compiling function calls. The rule simply looks up the function's com-

Figure 3.10: A BDD derivation tree for (EXLETCOMPILATION), with environments elided for visual clarity.

piled WBF and substitutes the actual argument for the formal argument. One subtlety is ensuring 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. An auxiliary function $\texttt{RefreshFlips}(x_{arg}, \dot\varphi, \gamma, w)$ is used for this purpose. Now it is possible to state the full correctness theorem for Dice compilation:

Theorem 3.1 (Compilation Correctness). *Let* p *be a* Dice *program and* $\emptyset, \emptyset \vdash$ p $: \tau \leadsto (\dot\varphi, \gamma, w)$. *Then: (1)* $[\![p]\!]_A = \texttt{WMC}(\gamma, w)$, *and (2) for any value* $v : \tau$, $[\![p]\!]_D(v) = \texttt{WMC}((\dot\varphi \stackrel{\tau}{\Leftrightarrow} v) \wedge \gamma, w)/\texttt{WMC}(\gamma, w)$.

All proofs for this chapter can be found in Appendix A.1. As before, division by zero is defined to be zero, and the above theorem is proved as a corollary of the following stronger property:

Theorem 3.2 (Typed Program Correctness). *Let* p *be a* Dice *program* $\emptyset, \emptyset \vdash$ p $: \tau \leadsto (\dot\varphi, \gamma, w)$. *Then for any* $v : \tau$, *we have that* $[\![p]\!](v) = \texttt{WMC}((\dot\varphi \stackrel{\tau}{\Leftrightarrow} v) \wedge \gamma, w)$.

3.4.5 Binary Decision Diagrams as WBF

Weighted model counting on WBFs is still #P-hard, so the compilation above is not necessarily advantageous. Now it is time to 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

Chapter 3.4.2. Each step in this derivation can be translated into a corresponding BDD operation, as illustrated by the *BDD derivation tree* in Figure 3.10. 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 substitution — are all standard operations that are available in BDD packages such as CUDD [Somenzi].

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 chapter is devoted to showing that the BDD can be efficient to construct for useful programs. In Chapter 3.5 we show this experimentally, and Chapter 3.6 characterizes the hardness of Dice inference.

3.5 Dice Implementation & Empirical Evaluation

This section describes the implementation and empirical evaluation of Dice. Dice is implemented in OCaml and uses CUDD as its backend for compiling BDDs [Somenzi]. First I 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 I describe our empirical evaluation of Dice's performance in comparison with prior PPLs on a suite of benchmarks. In Chapter 3.6 I give context to these experiments and discuss why Dice succeeds on many benchmarks where others fail.

3.5.1 Dice Extensions, Ergonomics, and Implementation Details

The actual implementation extends the core Dice syntax from Figure 3.5 in several ways. The constraint on A-normal form is relaxed here, allowing more arbitrary placement of expressions. Syntactic sugar for the usual Boolean operators $\wedge, \vee$ and $\neg$ is supported. Finally, bounded integers and bounded iteration are both supported as well, and are described in more detail next.

3.5.1.1 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:

$$
\texttt{discrete(0.1, 0.4. 0.5)} \rightsquigarrow
\begin{cases}
\texttt{let v0 = flip(0.1) in} \\
\texttt{let v1 = ¬v0 ∧ flip(0.4/(0.4 + 0.5)) in} \\
\texttt{let v2 = ¬v0 ∧ ¬v1 in (v0, (v1, v2))}
\end{cases}
$$

Formally, for a discrete distribution $\texttt{discrete}(\theta_1, \theta_2, \cdots, \theta_n)$, 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.

3.5.1.2 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 [Cusumano-Towner et al., 2018, Gehr et al., 2016, Claret et al., 2013, Pfeffer, 2007a, Goodman and Stuhlmüller, 2014]. 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`:

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

Many useful examples — such as the network reachability example from Chapter 3.2 — can be expressed as bounded iteration.

3.5.1.3 Variable Ordering

The *variable ordering* — the order in which variables are branched on in a BDD — is a critical parameter that determines how compactly a BDD can represent a particular logical formula [Meinel and Theobald, 1998, Bryant, 1986]. Finding the optimal order — the one that minimizes the size of the BDD — is NP-hard, so one must typically resort to heuristics for choosing orderings that work well in practice. `Dice` orders variables according to the syntactic order in which they occur in the program, mirroring the topological variable ordering heuristic from Bayesian networks [Darwiche, 2009]. We anticipate future work in deriving more sophisticated variable ordering heuristics from static program analyses.

3.5.1.4 Multi-rooted BDDs

`Dice` typically needs to represent many BDDs at the same time that share structure. The accepting and unnormalized formulas may share sub-formulas, or tuples may compile to

formulas that share some substructure. *Multi-rooted BDDs* naturally exploit this repeated substructure to compactly represent multiple Boolean formulas in a single data structure. For instance, the following example program that returns a tuple is compiled into the multi-rooted BDD shown after:

$$\text{let } x = \text{flip}_1\ 0.6 \text{ in let } y = x \wedge \text{flip}_2\ 0.4 \text{ in } (x, y)$$

This is compiled to a multi-rooted BDD, with each root shown with initial arrows:
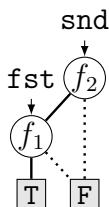
3.5.2 Empirical Performance Evaluation

This section describes the empirical evaluation of an implementation of the `Dice` compilation rules and BDD-based inference algorithm. Chapter 3.2 highlights some program structure that BDD compilation exploits, and Chapter 3.6 explores this structure further, but the question remains: does this structure exist in practice, and can `Dice` effectively exploit it? These questions are investigated from three angles:

Q1: Comparison with Existing PPLs How quickly can `Dice` perform exact inference on benchmark probabilistic programs from the literature? This is evaluated in Chapter 3.5.2.1.

Q2: Exploiting Functions What are the performance benefits of modular compilation for functions? This is evaluated in in Chapter 3.5.2.2 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? Chapter 3.5.2.3 compares `Dice` against `Ace` [Chavira and Darwiche, 2008], a state-of-the-art discrete Bayesian network solver.

The evaluation compares `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) [Gehr et al., 2016, Claret et al., 2013, Dehnert et al., 2017, Narayanan et al., 2016]. Chapter 3.6.3 discusses this class of inference algorithms more thoroughly. From this class PSI is compared against [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 [Wingate and Weber, 2013, Sankaranarayanan et al., 2013, Albarghouthi et al., 2017, Goodman and Stuhlmüller, 2014, Chistikov et al., 2015, Filieri et al., 2013, Geldenhuys et al., 2012]. Both PSI and WEBPPL [Goodman and Stuhlmüller, 2014] have a mode that supports dynamic-programming exact inference, and both are compared against 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 a best-effort attempt was made 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. 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.

3.5.2.1 Baselines

Table 3.1 summarizes the experimental performance results on well-known baselines which includes all of the discrete programs that PSI and R2 were evaluated on [Gehr et al., 2016, Nori et al., 2014, Borgström et al., 2011]. 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.

Two other columns – "# Paths" and "BDD Size" – are included. These give a proxy for how hard each inference problem is. The "# Paths" column gives how many paths would

---

[1] PSI version `2d21f9fe04cf3aac533e08ccc2df18179947baad` was used.

[2] `https://github.com/sharkdp/hyperfine`

Table 3.1: *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

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.

3.5.2.2 Modular Compilation

Now the motivating examples from Chapter 3.2 are returned to in order to see how `Dice` compares with existing methods, and against a version of itself where all function calls are inlined. Figure 3.11 shows how different algorithms scale as the size of the problem grows (note that all plots are in log-log scale).

Encryption   Figure 3.4 introduced the Caesar cipher motivating example, and Figure 3.11a 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 Chapter 3.6.3.

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 3.11a shows how it scales for this par-

(a) Caesar cipher with errors. (b) Diamond net. (c) Ladder net. (d) Figure 3.2.
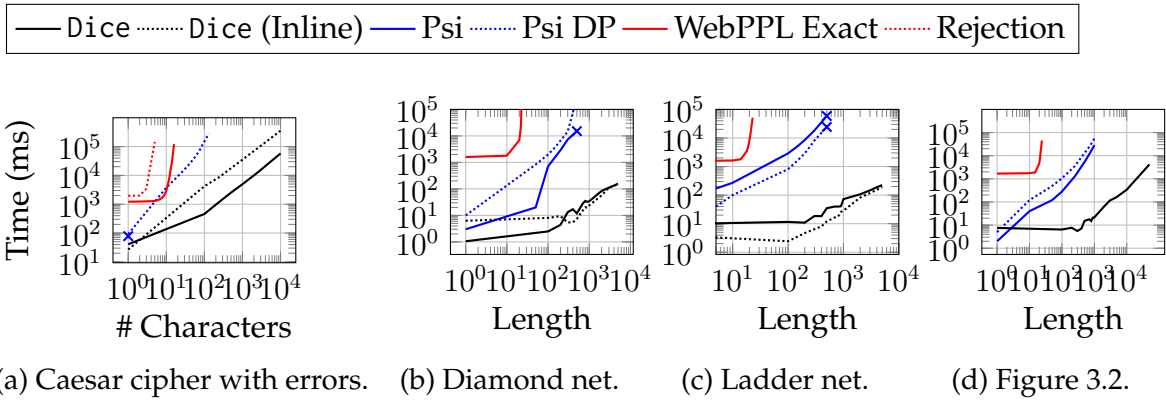
Figure 3.11: Log-log scaling plots illustrating the benefits of separate compilation of functions. An "x"-mark denotes a runtime error was encountered at that point. The time reported for `Dice` inference includes the time required to compile and perform WMC. The standard deviation for the run-times are negligible.

ticular 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 let us examine how separate compilation helps in the network reachability task described in Figure 3.3. Figure 3.11b shows how exact inference scales in the number of diamond subnetworks. There is 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.

It is expected to see overall linear scaling of `Dice` for many network topologies due to conditional independence. To evaluate this, Figure 3.11c shows a version where instead of diamonds a *ladder network* of the following structure is used:
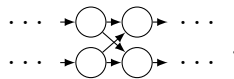
The goal is to determine the probability of a packet reaching the end of a network that consists of a chain of ladder subnetworks where each has a similar probabilistic routing policy to the diamond network. `Dice` continues to scale well, while this example is challenging for the other methods, in part since the number of paths is exponential in the length of the network.
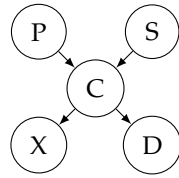
Figure 3.12: The "Cancer" Bayesian network.

3.5.2.3 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 the performance of `Dice` is evaluated on discrete Bayesian networks that are translated into equivalent Psi and `Dice` programs. These benchmarks were selected from the Bayesian Network Repository, an online repository of well-known Bayesian networks.[3] 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 3.2 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 3.12 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 $\Pr(Ⓧ)$.

Table 3.2 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, Table 3.3 compares `Dice` against `Ace`, a state-of-the-art Bayesian network solver [Chavira

---

[3]https://www.bnlearn.com/bnrepository/

Table 3.2: *Single Marginal Inference.* Comparison of inference algorithms (times are milliseconds). A "✗" 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	✗	✗	**25**	509	1.0×10^{36}	1.3×10^3
Insurance	✗	✗	**212**	984	1.2×10^{40}	1.0×10^5
Hepar2	✗	✗	**54**	48	2.9×10^{69}	1.3×10^3
Hailfinder	✗	✗	**618**	2656	2.0×10^{76}	6.5×10^4
Pigs	✗	✗	**72**	5618	7.3×10^{492}	35
Water	✗	✗	**2590**	1.0×10^4	3.2×10^{54}	5.1×10^4
Munin	✗	✗	**1866**	8.1×10^5	2.1×10^{1622}	1.1×10^4

Table 3.3: *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

and Darwiche, 2008]. The task here is to compute *all marginal probabilities*, a strictly harder task than the single-marginal task considered earlier. 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 `Dice` compilation is that a single (potentially expensive) compilation, once completed, can be efficiently reused to perform many marginal probability queries: this is a key benefit of compiling to a tractable probabilistic model. This capability is highlighted in Table 3.3, which 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 3.3 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.

3.6   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. This section provides discussion and analysis that provide context for these results. First Chapter 3.6.1 asks: how hard is exact inference in `Dice`? It shows 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 Chapter 3.5 succeed at all? This question is explored in Chapter 3.6.2 by identifying different forms of program structure that `Dice` exploits in order to scale. Finally, Chapter 3.6.3 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.

3.6.1 Computational Hardness of Exact `Dice` Inference

The experiments in Chapter 3.5 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 the experiments in Chapter 3.5 utilize a polynomial-time reduction from discrete Bayesian networks to `Dice` programs. However, exact inference for `Dice` is PSPACE-hard, and therefore likely harder than discrete Bayesian network inference as PP⊆PSPACE:

Theorem 3.3. *Exact inference in* `Dice` *is* **PSPACE**-*hard.*

Proof Sketch. The PSPACE-hardness of `Dice` inference follows directly from the expressiveness of non-recursive Boolean programs. In particular, there is a polynomial-time reduction from the *quantified Boolean formula* (QBF) problem, which is PSPACE-complete, to such a program. This reduction can also be used to reduce QBF to the problem of determining the probability that a `Dice` program outputs true. In particular, the construction relies on the expressiveness of nested function calls. Each nested function call corresponds to either a universal or existential quantifier, and the innermost call can be such that it evaluates a fully-quantified CNF. □

This result depends on the expressiveness of functions, which Bayesian networks lack.

3.6.2 When Is `Dice` Inference Fast?

`Dice` inference, in the worst case, is extremely hard. Why, then, do the experiments in Chapter 3.5 succeed? Put another way: when is it possible to guarantee that the BDD derivation tree is efficient to construct (i.e., polynomial in the size of the program)? This section explores 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, which here are shown to generalize to `Dice` programs.

3.6.2.1 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 (Chapter 3.5.2.1). Programs naturally have conditional independence, implied by their control flow, function boundaries, etc. In the motivating example in Figure 3.2b, 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 3.11d.

   `Dice` naturally exploits conditional independence. This is formalized 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 $\mathcal{O}(|B_1| \times |B_2|)$, 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 3.1. *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 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 $\mathcal{O}(|B_1| + |B_2|)$ for a variable order that orders the variables in B_1 before z and z before the variables in B_2.*

Proof Sketch. The proof is by construction. For instance, for conjunction, BDDs for B_1 and B_2 are of the form:
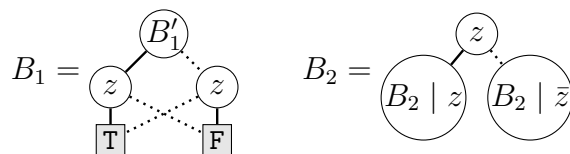
where B_1' is the BDD for B_1 with z separated out and $B_2 \mid z$ is the BDD for B_2 with $z = $ T. The BDD for $B_1 \wedge B_2$ can be constructed in linear time by traversing B_1' and rerouting all high edges coming from z to that end in T to $B_2 \mid z$, and all low edges from z that end in T to $B_2 \mid \bar{z}$. $\qquad \square$

   Proposition 3.1 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. The motivating example in Figure 3.3 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 3.3d grows as $\mathcal{O}(|\text{diamond}| \times c)$, where c is the number of calls to the `diamond` procedure and $|\text{diamond}|$ is the size of the compiled BDD for the procedure.

```
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.

(b) Compiled BDD.

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

(c) Structure without independence.
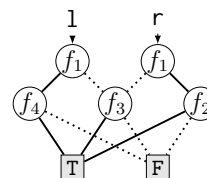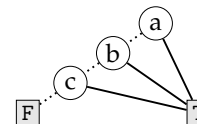
(d) Compiled BDD.

Figure 3.13: `Dice` programs and their compiled BDDs illustrating different degrees of structure.

`Dice` exploits another, more fine-grained form of independence called *context-specific independence*. Historically, context-specific independence has led to significant speedups in graphical model inference [Boutilier et al., 1996]. The benefits are briefly sketched 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 3.13a. 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 3.13b 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 the `Dice` 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. See Boutilier et al. [1996] for more on the performance benefits of exploiting context-specific independence in probabilistic graphical models.

3.6.2.2 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 prop-
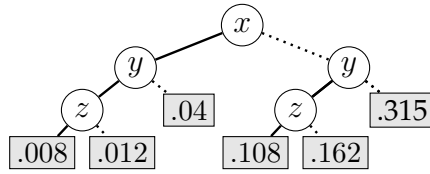
Figure 3.14: An ADD representation of the distribution in Equation 3.1.

erties that can make performance more efficient, including determinism, context-specific independence, and other properties [Boutilier et al., 1996, Gogate and Dechter, 2011, Sang et al., 2005, Chavira and Darwiche, 2008].

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 3.13c gives an example `Dice` function that computes the disjunction of three arguments. Figure 3.13d 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 3.3.

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. This section considers *algebraic compilation targets* as a foil to the `Dice` uses in order to highlight the 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 3.14 shows an ADD for the program in Figure 3.2a if it returned a tuple of x, y, and z. ADDs encode probabilities of total as-

signments of variables: in this example, a probability of 0.008 is given to the assignment $x = y = z = \mathtt{T}$.

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 $\mathtt{flip}$ 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 3.14, which collapses states with the same probability — for example, if $x = y = \mathtt{F}$, then the ADD terminates with a node that does not depend on z's value: $\boxed{.315}$.

Global repetitious probabilities are an orthogonal property to independence. ADDs do not exploit independence in the same way as $\mathtt{Dice}$. 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 $\mathtt{Dice}$'s BDD representation, the size of a compiled ADD is sensitive to the precise parameters chosen for $\mathtt{flips}$ 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 of paths in the probabilistic program. As shown in Table 3.1, 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.

3.7 Conclusion

This chapter presented a new approach to exact inference for discrete probabilistic programs and implement it in the $\mathtt{Dice}$ probabilistic programming language. It (1) showed how to reduce exact inference for $\mathtt{Dice}$ 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 $\mathtt{Dice}$ in key scenarios.

In the future I hope to extend $\mathtt{Dice}$ in several ways. First, I believe that the insights of $\mathtt{Dice}$ can be cleanly integrated into many existing probabilistic programming systems, even those with approximate inference that can handle continuous random variables. I see this as an exciting avenue for extending the reach of approximate inference algo-

Intermezzo 1: `Rubicon`: Model Checking Markov Chains with `Dice`

Probabilistic model checking is a sub-field of automated verification that seeks to verify properties of probabilistic processes. An example problem is as follows. Suppose there are n factories. Each day, the workers at each factory collectively decide whether or not to strike. Furthermore, since no two factories are identical, the probability to begin striking and to stop striking are different for each factory. Assuming that each factory transitions synchronously and in parallel with the others, a standard query is: "what is the probability that all the factories are simultaneously striking within h days?"

There are many mature tools for performing probabilistic model checking such as `Storm` [Dehnert et al., 2017] and `Prism` [Kwiatkowska et al., 2011]. At its core probabilistic model checking has much in common with probabilistic inference, and consequently probabilistic model checkers implicitly rely on similar algorithms to probabilistic inference. Concretely, probabilistic model checkers commonly employ algebraic representations to represent probability distributions symbolically during verification. Hence it is natural to ask: *how well can `Dice` serve as a probabilistic model checker?*

Holtzen et al. [2021] developed `Rubicon` to test this hypothesis, and showed that translating probabilistic model checking problems and queries into `Dice` programs and relying on `Dice`'s inference algorithm is a potentially profitable avenue for speeding up probabilistic model checkers. Importantly, this is because `Dice` uses a fundamentally different approach to inference – BDDs instead of ADDs – and so scales differently on certain classes of problems.

The following figure shows how `Dice` can scale on a the example factory problem in comparison with `Storm` and `Prism` as the number of factories increases:
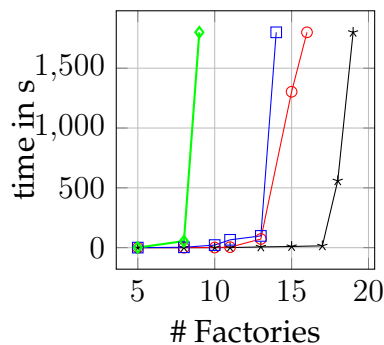
The above figure compares the performance of `Rubicon` (─∗─), `Storm`'s explicit engine (─∘─), `Storm`'s symbolic engine that uses ADDs (─□─) and `Prism` (─◆─). As the number of parallel factories grows, the state space of the problem grows exponentially, so `Dice` can scale to an order of magnitude more states on this example.

rithms, which currently struggle with discreteness. Second, I believe that `Dice` can be extended to handle more powerful data structures and programming constructs, notably forms of unbounded loops and recursion. And finally, I hope to further explore the landscape of weighted model counting approaches.

3.8 Bibliographic Notes

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`.

Applications   Due in part to their flexibility and ease of use, PPLs have been applied in a variety of scientific disciplines, including computer vision [Ritchie et al., 2016], cancer screening [Jacobs et al., 2016], biological modeling [Becker et al., 2017], psychology [Van de Schoot et al., 2017], modeling population dynamics [Hooten et al., 2017, Dhir et al., 2017], and more.

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, Hur et al., 2015, Pfeffer, 2007b, Chaganty et al., 2013, Wood et al., 2014, van de Meent et al., 2015, Mansinghka et al., 2013, Goodman et al., 2008, Saad and Mansinghka, 2016, Mansinghka et al., 2018] and variational approximations [Bingham et al., 2019, Dillon et al., 2017, Wingate and Weber, 2013, Kucukelbir et al., 2015, Minka et al., 2014]. Other approaches use symbolic techniques to perform inference but are similar in spirit, in the sense that they separately enumerate paths through the program [Sankaranarayanan et al., 2013, Albarghouthi et al., 2017, Geldenhuys et al., 2012, Filieri 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, Zeng and Van den Broeck, 2020, Dos Martires et al., 2019] — to perform inference by integrating along paths through a probabilistic program.

Additionally, sampling and variational algorithms are distinguished from this 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, Hoffman and Gelman, 2014, Gram-Hansen et al., 2018, Wingate and Weber, 2013, Kucukelbir

et al., 2015, Minka et al., 2014]. 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 this approach based on weighted model counting; see Chapter 3.6.3 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 Pfeffer [2009], McCallum et al. [2009], Minka et al. [2014], Bornholt et al. [2014]. 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 this approach exploits, as shown in Chapter 3.6.2.1 [Darwiche, 2009].

Probabilistic Logic Programs   Closest to the approach presented in this chapter are techniques for exact inference in probabilistic logic programs De Raedt et al. [2007], Riguzzi and Swift [2011], Fierens et al. [2015], Vlasselaer et al. [2015]. Similar to this 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 [Pfeffer et al., 2018, Mansinghka et al., 2018, Holtzen et al., 2018]. Hence the goal is similar in spirit to this chapter's goal of automated program factorization. These approaches are complementary: `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 [Jhala and Majumdar, 2009, Biere, 2009]. `Dice`'s compilation 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]; see Intermezzo 1 and Holtzen et al. [2021]. 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 probablistic 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 `Dice` but differs markedly in its goals and encoding strategy.

CHAPTER 4

Exploiting Symmetry with Lifted Inference

> *Symmetry is what we see at a glance;*
> *based on the fact that there is no reason for*
> *any difference.*

<div align="right">Blaise Pascal</div>

Chapter 3 described how to exploit factorization and modularity in probabilistic programs, but this is not the only kind of structure that a probabilistic program or probabilistic model can exhibit.

Recall the pigeonhole example described in Chapter 1.1.3. Imagine here that there is just a single pigeon, but there are 5 holes that it wants to hide in. What is the probability that the pigeon is in a particular hole? This is easy to answer at a glance: the probability that it is in any particular hole is 1/5, since each hole is equally likely and there are 5 of them. But what happens if we try to encode this situation as a Dice program? Here is an example of how this might be encoded:

```
1  let isInHole1 = flip 1/5 in
2  let isInHole2 = if !isInHole1 then flip 1/4 else false in
3  let isInHole3 = if !isInHole1 && !isInHole2 then flip 1/3 else false in
4  let isInHole4 = if !isInHole1 && !isInHole2 && !isInHole3 then flip 1/2 else
       false in
5  let isInHole5 = !isInHole1 && !isInHole2 && !isInHole3 && !isInHole4 in
6  (isInHole1, isInHole2, isInHole3, isInHole4, isInHole5)
```

This program returns a distribution on 5 holes, but it has interesting structure. Programs have *execution order*: the pigeon must decide, in order, which hole it wants to be in, which breaks the delicate symmetry on the holes. In this program, each hole is distinct from the others because of their ordering. Moreover, observe that there is no independence between each of the isInHole variables: each depends on all the previous. More

broadly, even if the symmetry is obvious, existing probabilistic program inference algorithms do not

The key to this problem is identifying and exploiting properties of the distribution that make inference tractable. *Lifted inference algorithms* identify symmetry as a property that enables efficient inference and seek to scale with the degree of symmetry of a probability model [Poole, 2003, Kersting, 2012, Niepert and Van den Broeck, 2014]. Many existing exact inference algorithms, such as the BDD compilation strategy employed by `Dice`, are unaware of and cannot directly exploit symmetry for speeding up inference. To exploit this structure, we will need an entirely different inference methodology.

Lifted inference identifies *orbits* of the distribution: sets of points in the probability space that are guaranteed to have the same probability. This enables inference strategies that scale in the number of distinct orbits. Highly symmetric distributions have few orbits relative to the size of their state space, allowing lifted inference algorithms to scale to large probability distributions with scant independence. Thus, lifted inference algorithms identify symmetry as a complement to independence in the search for efficient inference algorithms.

This chapter introduces a new family of exact and approximate lifted inference algorithms. It will also focus on a class of probabilistic models not explored yet in this thesis: *factor graphs* [Koller and Friedman, 2009a]. The reason for focusing initially on factor graphs rather than probabilistic programs is practical.[1] One of the main challenge in designing a lifted inference technique is finding and identifying symmetries. The techniques presented in this chapter will heavily rely on *graph isomorphism tools* for finding symmetries: in the future, I hope this can be used as a foundation for new probabilistic program inference algorithms that exploit symmetry.

4.1 Introduction

A *factor graph* is a bipartite graph that defines a distribution on random variables by a set of functions called *factors*, which intuitively give a weight to assignments to a subset of variables.[2] Formally stated:

Definition 4.1 (Factor graph). *Let* $\mathbf{X}$ *be a set called the* variable set. *We call a vector* $\mathbf{x} = \{0, 1\}^{|\mathbf{X}|}$ *an* assignment. *A function* $f : \{0, 1\}^k \to \mathbb{R}$ *is called a* factor. *Then a* factor graph *is a bipartite graph defined by the tuple* $G = (\mathbf{X}, F, E)$, *where* $\mathbf{X}$ *are variable nodes,* F *are factor nodes associated with factors, and* E *are edges between variables and factor nodes.*

A factor graph is a probabilistic model. It defines a distribution on assignments via

---

[1]There are probabilistic programs that compile programs to factor graphs, and so the methods described in this directly apply to these languages [McCallum et al., 2009, Minka et al., 2014, Bornholt et al., 2014].

[2]For a good reference on factor graphs and other graphical models, see Koller and Friedman [2009a].
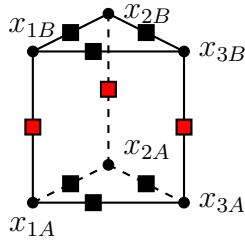
Figure 4.1: The pigeon-hole factor graph example with 3 pigeons and 2 holes.

the following:

$$\Pr(\mathbf{x}) \triangleq \frac{1}{Z} \prod_{f \in F} f(\mathbf{x}). \tag{4.1}$$

The symbol Z is a normalizing constant. Observe that this definition is not quite right: a factor $f_i : \{0,1\}^k \to \mathbb{R}$ is only well-defined on a k-dimensional sub-vector of $\mathbf{x}$. This is where the edges between variables and factors come in: the edges will tell us which variables are required data for evaluating a particular factor. Let $\mathsf{Vars}(f)$ be the set of indices of variables that share an edge with factor f. As convention, we say $f(\mathbf{x})$ is the factor $f : \{0,1\}^k \to \mathbb{R}$ applied to a k-dimensional sub-vector of $\mathbf{x}$ given by:

$$f((x_1, x_2, \cdots, x_n)) \triangleq f((x_{p_1}, x_{p_2}, \cdots, x_{p_k})) \quad \text{where } p_i \in \mathsf{Vars}(f). \tag{4.2}$$

Hence, intuitively, the graph structure tells us how the distribution factorizes into a product of factors.

This definition can seem complicated at first, so it is best to further develop it with an example. Figure 4.1 shows how to encode as a factor graph the 3-pigeons in 2-holes situation similar to the one in Figure 1.2. By convention, factors are denoted with square boxes and variables are denoted with round nodes. Each of the three pigeons is identified by a number, and the two holes are identified by the letters A and B: hence the variable node x_{1B} is 1 if and only if pigeon 1 is in hole B.

There are two kinds of factors in this graph: *hole factors* (shown in red) that connect two holes and give a large weight to a state in which no two pigeons are in the same hole, and *pigeon factors* (in black) that give a large weight to a state in which no two pigeons are in more than a single hole simultaneously. For instance, we might define hole factors f_h

as and pigeon factors f_p as:

$$f_h(x_{iA}, x_{iB}) = \begin{cases} 1 & \text{if } x_{iA} = x_{iB} = 1 \\ 1000 & \text{if } x_{iA} \neq x_{iB} \\ 10 & \text{otherwise.} \end{cases} \tag{4.3}$$

$$f_p(x_{iB}, x_{jB}) = \begin{cases} -\infty & \text{if } x_{iB} = x_{jB} = 1 \\ 100000 & \text{if } x_{iB} \neq x_{jB} \\ 1 & \text{otherwise.} \end{cases} \tag{4.4}$$

We create one hole factor for each pair of holes and one pigeon factor for each pair of pigeons. Each of these factors has special structure: *they are symmetric about their arguments*, meaning that we can always permute the order in which arguments are presented to the factor without changing the probability. This is a key property which will be exploited during lifted inference.

The first question that must be answered is: *how can we find the symmetries implied by factors in a factor graph*? A key observation made in the lifted inference literature is that the symmetries of a probability distribution directly correspond to automorphisms of the colored graph [Bui et al., 2013, Niepert, 2012]. Any permutation of vertices that preserves the graph structure leaves the distribution unchanged.[3] Two assignments (see Definition 4.1) that are reachable from one another via a sequence of permutations are in the same *orbit*; all assignments in the same orbit thus have the same probability.

Figure 4.2 shows the orbits of the 3-pigeon 2-hole scenario up to inversion of true and false assignments. Each orbit is boxed. There are few orbits relative to the number of states, which is the property that lifted inference algorithms exploit.

This chapter presents both exact and approximate inference strategies that scale with the number of orbits of a probability distribution. The exact inference algorithm is as follows. First, generate a single *canonical representative* from each orbit; in Figure 4.2, canonical representatives are shown in bold. Then for each representative, compute the size of its orbit. If both of these steps are efficient, then this inference computation scales efficiently with the number of orbits. This *orbit generation* procedure is at the heart of many existing lifted inference algorithms that construct sufficient statistics of the distribution from a relational representation [Niepert and Van den Broeck, 2014]. An exact lifted inference algorithm is given in Chapter 4.3 that applies this methodology to arbitrary factor graphs by using graph isomorphism tools to generate canonical representatives and compute orbit sizes.

Next, Chapter 4.4 describes an approximate inference algorithm called *orbit-jump MCMC*

---

[3]It is assumed here w.l.o.g. but for simplicity that the factors are individually fully symmetric. Asymmetric factors can either be made symmetric by duplicating variable nodes [Niepert, 2012] or encoded using colored edges [Bui et al., 2013].

100 000
010 000
001 000
000 100
000 010
000 001

000 000

| **110 000** | 011 000 | 101 000 |
| 000 110 | 000 011 | 000 101 |

| **100 100** | 010 010 | 001 001 |

| **100 010** | 100 001 |
| 010 100 | 001 100 |

| **110 100** | 110 010 | 011 001 |
| 100 110 | 010 110 | 001 011 |

| **111 000** | 000 111 |

| **110 001** | 101 010 | 011 100 |
| 001 110 | 010 101 | 100 011 |
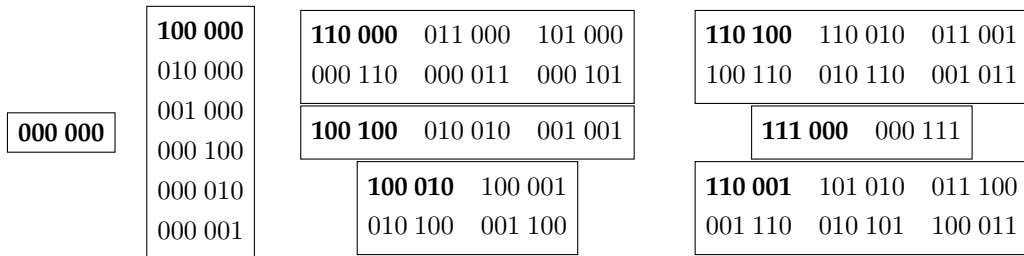
Figure 4.2: Orbits of the assignments to variables in the pigeonhole problem. An assignment is written as a binary string $x_{1A}x_{2A}x_{3A}x_{1B}x_{2B}x_{3B}$. Each orbit is boxed, and each canonical representative is bold. Cases where there are 4 or greater true variable assignments are omitted, as these are symmetric to previously listed cases where the true and false values are flipped.

that provably mixes quickly in the number of distinct orbits of the distribution. This algorithm uses as its proposal the *uniform orbit distribution*: the distribution defined by choosing an orbit of the distribution uniformly at random, and then choosing an element within that orbit uniformly at random. We present a novel application of the *Burnside process* in order to draw samples from the uniform orbit distribution [Jerrum, 1993], and show how to implement the Burnside process on factor graphs by using graph isomorphism tools. Thus, this orbit-jump MCMC provides an alternative to lifted MCMC that trades computation time for provably good sample quality.

Note, however, that purely scaling in the number of orbits is not a panacea. The presented methods are both limited: there are liftable probability models that still have too many orbits for these methods to be effective. The presented methods *only* exploit symmetry, which is in contrast to existing exact lifted inference algorithms that simultaneously exploit symmetry and independence. Therefore, the presented algorithms scale exponentially for certain well-known liftable distributions, such as the friends and smokers Markov logic network [Niepert and Van den Broeck, 2014]. Thus, this work provides a foundation for future work on inference for factor graphs that exploits both symmetry and independence.

4.2 Background

This section gives a brief description of important concepts from group theory and approximate lifted inference that will be used throughout the chapter.

4.2.1 Group Theory

This section briefly reviews some standard terminology and notation from group theory, following Artin [1998]. A *group* $\mathcal{G}$ is a pair $(S, \cdot)$ where S is a set and $\cdot : S \times S \to S$ is a

binary associative function such that there is an identity element and every element in S has an inverse under $(\cdot)$. The *order* of a group is the number of elements of its underlying set, and is denoted $|\mathcal{G}|$. A *permutation group acting on a set* Ω is a set of bijections $g : \Omega \to \Omega$ that forms a group under function composition. For $\mathcal{G}$ acting on Ω, a function $f : \Omega \to \Omega'$ is $\mathcal{G}$-*invariant* if $f(g \cdot x) = f(x)$ for any $g \in \mathcal{G}, x \in \Omega$. Two elements $x, x' \in \Omega$ are in the same *orbit* under $\mathcal{G}$ if there exists $g \in \mathcal{G}$ such that $x = g \cdot x'$. Orbit membership is an equivalence relation, written $x \sim_{\mathcal{G}} x'$. The set of all elements in the same orbit is denoted $\mathrm{Orb}_{\mathcal{G}}(x)$. A *stabilizer* of x is an element $g \in \mathcal{G}$ such that $g \cdot x = x$; the set of all stabilizers of x is a group called the *stabilizer subgroup*, denoted $\mathrm{Stab}_{\mathcal{G}}(x)$. The subscript in the previous notation is elided when clear. A *cycle* $(x_1 \, x_2 \, \cdots \, x_n)$ is a permutation $x_1 \mapsto x_2, x_2 \mapsto x_3, \cdots, x_n \mapsto x_1$. A permutation can be written as a product of disjoint cycles.

4.2.2   Lifted Probabilistic Inference & Graph Automorphism Groups

Lifted inference relies on the ability to identify the symmetries of probability distributions. In existing exact lifted inference methods, the symmetries are evident from the relational structure of the probability model [Poole, 2003, De Salvo Braz et al., 2005, Gogate and Domingos, 2011, Van den Broeck, 2013]. In order to extend the insights of lifted inference to models where the symmetries are less accessible, many lifted approximation algorithms rely on graph isomorphism tools to identify the symmetries of probability distributions [Niepert, 2012, Bui et al., 2013, Mckay and Piperno, 2014].

A *colored graph* is a 3-tuple $\mathtt{G} = (V, E, C)$ where (V, E) are the vertices and edges of an undirected graph and $C : V \to \mathbb{N}$ assigns a non-negative integer, or *color*, to each vertex.

Definition 4.2. *Let* $\mathtt{G} = (V, E, C)$ *and* $\mathtt{G}' = (V, E', C')$ *be colored graphs. Then* $\mathtt{G}$ *and* $\mathtt{G}'$ *are* color-isomorphic *to one another, denoted* $\cong$*, if there exists a bijection* $\phi : V \to V$ *such that (1)* $(v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E'$*; and (2) for all* $v \in V$, $C(v) = C'(v)$.

Factor graphs have a natural encoding as a colored graph:

Definition 4.3 (Induced colored graph). *Let* $\mathcal{F} = (\mathbf{X}, F)$ *be a factor graph with variables* $\mathbf{X}$*, and factors* F*, where* F *are symmetric functions on assignments to variables* $\mathbf{X}$*, written* $\mathbf{x}$*. Then the* colored graph induced by $\mathcal{F}$ *is a tuple* (V, E, C) *where* $V = \mathbf{X} \cup F$*, the set of edges* E *connects variables and factors in* $\mathcal{F}$*, and* C *is a partition such that (1) factor nodes are given the same color iff they are identical factors, and (2) variables are colored with a single color that is distinct from the factor colors.*

A related notion is the color-automorphism group a colored graph:

Definition 4.4. *The* color automorphism group *of a colored graph* $\mathtt{G} = (V, E, C)$*, denoted* $\mathbb{A}(\mathtt{G})$*, is the set of all color isomorphisms onto itself.*

The group $\mathbb{A}(\mathtt{G})$ acts on the vertices of $\mathtt{G}$ by permuting them. The color automorphism group of a colored factor graph is directly related to the symmetries of the underlying distribution:

Theorem 4.1 (Bui et al. [2013], Theorem 2)**.** *Let $\mathcal{F}$ be a factor graph and $\mathtt{G}$ be its induced colored graph. Then, the distribution of $\mathcal{F}$ is $\mathbb{A}(\mathtt{G})$-invariant.*

4.3 Exact Lifted Inference

This section describes the exact lifted inference procedure. First I will discuss the group-theoretic properties of orbit generation that enable efficient exact lifted inference. Then, I describe the algorithm for implementing orbit generation on colored factor graphs. Finally, I present some case studies demonstrating the performance of the algorithm.

4.3.1 $\mathcal{G}$-**Invariance & Tractability**

This section describes the group-theoretic underpinnings of the orbit-generation procedure and describes its relationship with previous work on tractability through exchangeability. We will capture the behavior of a $\mathcal{G}$-invariant probability distribution on a set of *canonical representatives* of each orbit:

Definition 4.5. *Let $\mathcal{G}$ be a group that acts on a set Ω. Then, there exists a set of* canonical *representatives $\Omega/\mathcal{G} \subseteq \Omega$ and surjective canonization function $\sigma : \Omega \to \Omega/\mathcal{G}$ such that for any $x, y \in \Omega$, (1) $\mathrm{Orb}(x) = \mathrm{Orb}(\sigma(x))$; and (2) $\mathrm{Orb}(x) = \mathrm{Orb}(y)$ if and only if $\sigma(x) = \sigma(y)$.*

In statistics, σ is often called a *sufficient statistic* of a partially exchangeable distribution [Niepert and Van den Broeck, 2014, Diaconis and Freedman, 1980]. The motivating example hinted at a general-purpose solution for exact inference that proceeds in two phases. First, one constructs a representative of each orbit; then, one efficiently computes the size of that orbit. We can formalize this using group theory:

Theorem 4.2. *Let Pr be a $\mathcal{G}$-invariant distribution on Ω, and evidence $\mathbf{e} : \Omega \to \mathbf{Bool}$ be a $\mathcal{G}$-invariant function. Then, the complexity of computing the most probable explanation (MPE) is $\mathrm{poly}(|\Omega/\mathcal{G}|)$ if the following can be computed in $\mathrm{poly}(|\Omega/\mathcal{G}|)$:*

1. *Evaluate $\mathrm{Pr}(x)$ for $x \in \Omega$;*
2. *(Canonical generation) Generate a set of canonical representatives $\Omega/\mathcal{G}$,*

Moreover, if $|\mathrm{Orb}(x)|$ can be computed in $\mathrm{poly}(|\Omega/\mathcal{G}|)$, then $\mathrm{Pr}(\mathbf{e})$ can be computed in $\mathrm{poly}(|\Omega/\mathcal{G}|)$.

Proof. To compute the MPE, choose:

$$\arg\max_{\{x \in \Omega/\mathcal{G} \mid \mathbf{e}(x)=\mathtt{T}\}} \mathrm{Pr}(x). \tag{4.5}$$

The $\mathcal{G}$-invariance of e allows us to evaluate e on only x without considering other elements of $\mathrm{Orb}(x)$. To compute $\mathrm{Pr}(\mathbf{e})$, compute

$$\sum_{\{x \in \Omega/\mathcal{G} \ | \ \mathbf{e}(x)=\mathtt{T}\}} |\mathrm{Orb}(x)| \times \mathrm{Pr}(x). \tag{4.6}$$

Both of these can be accomplished in $\mathrm{poly}(|\Omega/\mathcal{G}|)$. $\qquad\square$

Niepert and Van den Broeck [2014] identified a connection between bounded-width *exchangeable decompositions* and tractable (i.e., domain-lifted) exact probabilistic inference using the above approach. Exchangeable decompositions are a particular kind of $\mathcal{G}$-invariance. Let $\mathrm{Pr}(\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n)$ be a distribution on sets of variables $\mathbf{X}_i$. Let S_n be a group of all permutations on a set of size n. Then, this distribution has an exchangeable decomposition along $\{\mathbf{X}_i\}$ if, for any $g \in S_n$:

$$\mathrm{Pr}(\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n) = \mathrm{Pr}(\mathbf{X}_{g \cdot 1}, \mathbf{X}_{g \cdot 2}, \cdots, \mathbf{X}_{g \cdot n})$$

Niepert and Van den Broeck [2014] showed how to perform exact lifted probabilistic inference on any distribution with a fixed-width exchangeable decomposition by directly constructing canonical representatives. However, this construction does not generalize to other kinds of symmetries, and thus cannot be applied to factor graphs which may have arbitrarily complex symmetric structure. In the next section, we show how to apply Theorem 4.2 to factor graphs.

4.3.2 Orbit Generation

The previous section shows that inference can be efficient if one can (1) construct representatives of each orbit class, (2) compute how large each orbit is. This section gives an algorithm for performing these two operations for colored factor graphs. First, the procedure for encoding variable assignments directly into the colored factor graph is described, providing a way to leverage graph isomorphism tools to compute canonical representatives and orbit sizes for assignments to variables in factor graphs. This colored assignment encoding is one of the key technical contributions of this chapter, and forms a foundation for the exact and approximate inference algorithms. Then, I will give a breadth-first search procedure for generating all canonical representatives of a colored factor graph.

4.3.2.1 Encoding Assignments

The objective in this section is to leverage graph isomorphism tools to compute the key quantities necessary for applying the procedure described in Theorem 4.2 to factor graphs. Let $\mathtt{G}$ be the induced colored graph of $\mathcal{F}$. As terminology, an element $\mathbf{x} \in \mathbf{Bool}^{\mathbf{X}}$ is an
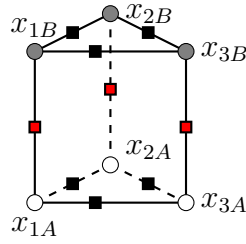
Figure 4.3: A colored graph of the 3-pigeon 2-hole problem that encodes the assignment $\mathbf{x} = 000\,111$. True variable nodes are gray and false variable nodes are white.

assignment to variables $\mathbf{X}$. We will use graph isomorphism tools to construct (1) a canonization function for variable assignments, $\sigma : \mathbf{Bool}^{\mathbf{X}} \to \mathbf{Bool}^{\mathbf{X}}/\mathbb{A}(\mathtt{G})$; and (2) the size of the orbit of $\mathbf{x} \in \mathbf{Bool}^{\mathbf{X}}$ under $\mathbb{A}(\mathtt{G})$. To do this, assignments are encoded directly into the colored factor graph:

Definition 4.6. *Let $\mathcal{F} = (\mathbf{X}, F)$ be a factor graph, let $\mathbf{x} \in \mathbf{Bool}^{\mathbf{X}}$, and let $\mathtt{G} = (V, E, C)$ be the colored graph induced by $\mathcal{F}$. Then the* assignment-encoded colored graph, *denoted $\mathtt{G}(\mathcal{F}, \mathbf{x})$, is the colored graph that colors the variable nodes that are true and false in $\mathbf{x}$ with distinct colors in $\mathtt{G}$.*

An example is shown in Figure 4.3, which shows an encoding of the assignment $000\,111$. The assignment $000\,111$ is isomorphic to the assignment $111\,000$ under the action of $\mathbb{A}(\mathtt{G})$, specifically flipping holes. Then, assignments that are in the same orbit under $\mathbb{A}(\mathtt{G})$ have isomorphic colored graph encodings:

Theorem 4.3. *Let $\mathcal{F} = (\mathbf{X}, F)$ be a factor graph, $\mathtt{G}$ be its colored graph encoding, and $\mathbf{x}, \mathbf{x}' \in \mathbf{Bool}^{\mathbf{X}}$. Then, $\mathbf{x} \sim \mathbf{x}'$ under the action of $\mathbb{A}(\mathtt{G})$ iff $\mathtt{G}(\mathcal{F}, \mathbf{x}) \cong \mathtt{G}(\mathcal{F}, \mathbf{x}')$.*

Proof. Let $\mathtt{G}_1 = (V_1, E_1, C_1) = \mathtt{G}(\mathcal{F}, \mathbf{x})$ and $\mathtt{G}_2 = (V_2, E_2, C_2) = g \cdot \mathtt{G}(\mathcal{F}, \mathbf{x})$. Assume $\mathbf{x} \sim \mathbf{x}'$. Then there exists an element $g \in \mathbb{A}(\mathtt{G})$ such that $g \cdot \mathbf{x} = \mathbf{x}'$. First we show colors are preserved. By construction of the colored assignment encoding, for any variable node $v \in \mathtt{G}_1$, $\mathtt{color}(v, C_1) = \mathtt{color}(g \cdot v, C_2)$. The colors of factor nodes are preserved because $\mathbb{A}(\mathtt{G})$ by definition preserves them. The fact that $g \in \mathbb{A}(\mathtt{G})$ directly implies that vertex neighborhoods are preserved. Then $\mathtt{G}_1 \cong \mathtt{G}_2$.

Assume $\mathtt{G}_1 \cong \mathtt{G}_2$; then there exists $g \in \mathbb{A}(\mathtt{G})$ such that $g \cdot \mathtt{G}_1 = \mathtt{G}_2$. By the construction of the colored encoding, this g also preserves the colors of the variable vertices, so $g \cdot \mathbf{x} = \mathbf{x}'$. $\qquad\square$

Canonization The goal now is to use graph isomorphism tools to construct a canonization function for variable assignments. In particular, it maps all isomorphic assignments to exactly one member of their orbit. This will rely on *colored graph canonization*, a well-studied problem in graph theory for which there exist many implementations [Mckay and Piperno, 2014]:

61

Definition 4.7. *Let* $\mathtt{G} = (V, E, C)$ *be a colored graph. Then a* colored graph canonization *is a canonization function* $\sigma : V \to V/\mathbb{A}(\mathtt{G})$.

A colored graph canonization function applied to Figure 4.3 will select exactly one color-isomorphic vertex configuration as the canonical one, for example putting all pigeons in hole A. Then, the canonization of the assignment-encoded colored graph is a canonization of variable assignments:

Definition 4.8. *Let* $\mathcal{F} = (\mathbf{X}, F)$ *and* $\mathbf{x} = \{(x, v)\}$ *be a variable assignment, where* $x \in \mathbf{X}$ *and* $v \in \mathbf{Bool}$. *Let* $\sigma_{\mathtt{G}(\mathcal{F}, \mathbf{x})}$ *be a canonization of* $\mathtt{G}(\mathcal{F}, \mathbf{x})$. *Then, let* $\sigma' : \mathbf{Bool}^{\mathbf{X}} \to \mathbf{Bool}^{\mathbf{X}}$ *be defined* $\sigma'(\mathbf{x}) = \{(\sigma_{\mathtt{G}(\mathcal{F}, \mathbf{x})}(x), v) \mid (x, v) \in \mathbf{x}\}$. *Then* σ' *is called the* induced variable canonization *of* $\mathbf{Bool}^{\mathbf{X}}$.

Intuitively, an induced variable canonization computes the canonization of the assignment-encoded colored graph, and then applies that canonization function to variables. Then,

Proposition 4.1. *For a factor graph* $\mathcal{F}$ *with colored graph* $\mathtt{G}$, *the induced variable canonization is a canonization function* $\mathbf{Bool}^{\mathbf{X}} \to \mathbf{Bool}^{\mathbf{X}}/\mathbb{A}(\mathtt{G})$.

4.3.2.2 Computing the Size of an Orbit

Theorem 4.2 requires efficiently computing the size of the orbit of an assignment, a task that at first glance seems hard. In fact, the orbit size can be computed by reducing the problem to a graph isomorphism call and an efficient group order computation. This reduction hinges on the following well-known theorem that relates the size of an orbit to the size of a stabilizer:

Theorem 4.4 (Orbit-stabilizer [Artin, 1998]). *Let* $\mathcal{G}$ *act on* Ω. *Then for any* $x \in \Omega$, $|\mathcal{G}| = |\mathrm{Stab}(x)| \times |\mathrm{Orb}(x)|$.

Proof. This well-known theorem has many proofs, and one is included here due to its fundamental importance. Let $\mathrm{Orb}(x) = \{x_1, x_2, \cdots, x_n\}$, and let $P = \{\pi_1, \pi_2, \cdots, \pi_n\}$ be such that $\pi_i \cdot x = x_i$. Then, $|P| = |\mathrm{Orb}(x)|$. We will show that every element of G can be written in exactly one way as a product of an element in P and an element in $\mathrm{Stab}(x)$. This fact directly implies the theorem.

First, we show that each element can be written as a product of $\alpha \in P$ and $\beta \in \mathrm{Stab}(x)$. Let $g \in G$. Then, for some $\pi_i \in P$, we have that $\pi_i \cdot x = \pi \cdot x$, so $\pi_i^{-1} \cdot \pi \in \mathrm{Stab}(x)$. So, we have that:

$$\underbrace{\pi_i}_{\in P} \cdot \underbrace{(\pi_i^{-1} \cdot \pi)}_{\in \mathrm{Stab}(x)} = \pi. \tag{4.7}$$
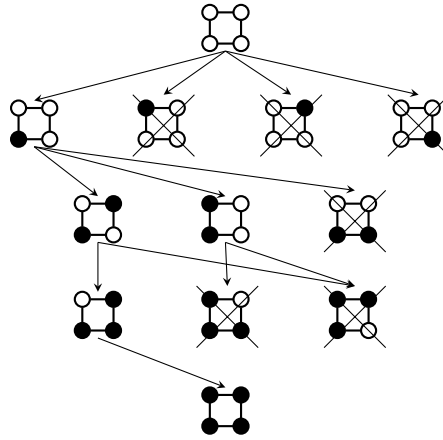
Figure 4.4: Example breadth-first search tree, read top-down. White nodes encode false assignments, and black nodes encode true assignments.

Now, we show that this product is unique. Let $\pi \in G$. Assume there exist $\alpha_1, \alpha_2 \in P$ and $\beta_1, \beta_2 \in \mathrm{Stab}(x)$ such that $\pi = \alpha_1 \cdot \beta_1 = \alpha_2 \cdot \beta_2$. Both β_1 and β_2 stabilize x, so $\alpha_1 \cdot \beta_1 = \alpha_2 \cdot \beta_2$ implies $\alpha_1 = \alpha_2$, which implies $\beta_1 = \beta_2$. $\qquad\square$

Then, to compute orbit size of assignments $\mathbf{x}$, compute (1) the stabilizer group of an assignment $\mathrm{Stab}_{\mathbb{A}(\mathtt{G})}(\mathbf{x})$ and (2) the order of $\mathbb{A}(\mathtt{G})$ and the assignment stabilizer. Computing the order of a group is efficient, and high-performance algorithms are implemented in computational group theory tools such as GAP [GAP, Seress, 2003].

4.3.2.3 Generating All Canonical Representatives

The algorithm for generating canonical representatives is a simple breadth-first search that relies on assignment canonization. This procedure is a kind of *isomorph-free exhaustive generation*, and there exist more sophisticated procedures than the one we present here [McKay, 1998].

Let $\mathbf{x}$ be some variable assignment. Then, an *augmentation* of $\mathbf{x}$ is a copy of $\mathbf{x}$ with one variable that was previously false assigned to true. We denote the set of all augmentations as $\mathcal{A}(\mathbf{x})$. The breadth-first search tree will be defined by a series of augmentations as follows:

Orbit generation breadth-first search

1. Nodes of the search tree are assignments $\mathbf{x}$.
2. The root of the tree is the all false assignment.
3. Each level L of the search tree has exactly L true assignments to variables.
4. Nodes are expanded until level $|\mathbf{X}|$.

---

5. Before expanding a node, check if it is not isomorphic to one that has already been expanded by computing its canonical form.

6. Then, expand a node $\mathbf{x}$ by adding $\mathcal{A}(\mathbf{x})$ to the frontier.

An example of this breadth-first search procedure is visualized in Figure 4.4. The search is performed on a 4-variable factor graph that has one factor on each edge, and all factors are symmetric. The factors are elided in the figure for visual clarity. Each arrow represents an augmentation. Crossed out graphs are pruned due to being isomorphic with a previously expanded node.

Now we bound the number of required graph isomorphism calls for this search procedure:

Theorem 4.5. *For a factor graph* $\mathcal{F} = (\mathbf{X}, F)$ *with* $|\mathbf{Bool}^{\mathbf{X}}/\mathbb{A}(\mathtt{G})|$ *canonical representatives, the above breadth-first search requires at most* $|\mathbf{X}| \times |\mathbf{Bool}^{\mathbf{X}}/\mathbb{A}(\mathtt{G})|$ *calls to a graph isomorphism tool.*

Proof. There are at most $|\mathbf{Bool}^{\mathbf{X}}/\mathbb{A}(\mathtt{G})|$ expansions, and each expansion adds at most $|\mathbf{X}|$ nodes to the frontier. A canonical form must be computed for each node that is added to the frontier. □

Pruning expansions   This expansion process can be further optimized by preemptively reducing the number of nodes that are added to the frontier in Step 6, using the following lemma:

Lemma 4.1 (Expansion Pruning). *Let* $\mathcal{F}$ *be a factor graph,* $\mathbf{x}$ *be a variable assignment, and* $\mathbf{x}_1, \mathbf{x}_2$ *be augmentations of* $\mathbf{x}$ *that update variables* x *and* y *respectively. Then,* $\mathbf{x}_1 \sim \mathbf{x}_2$ *under* $\mathbb{A}(\mathtt{G})$ *if* x *and* y *are in the same variable orbit under* $\mathbb{A}(\mathtt{G}(\mathcal{F}, \mathbf{x}))$.

Proof. Let $\mathtt{G}_1 = \mathtt{G}(\mathcal{F}, \mathbf{x}_1)$ and $\mathtt{G}_2 = \mathtt{G}(\mathcal{F}, \mathbf{x}_2)$. Assume x and y are in the same orbit under $\mathbb{A}(\mathtt{G}(\mathcal{F}, \mathbf{x}))$; then there exists $g \in \mathbb{A}(\mathtt{G}(\mathcal{F}, \mathbf{x}))$ such that $g \cdot x = y$. There is only one vertex color that differs between $\mathtt{G}_1$ and $\mathtt{G}_2$: x and y. Then, $g \cdot \mathtt{G}_1 = \mathtt{G}_2$, so Theorem 4.3 then shows $\mathbf{x}_1 \sim \mathbf{x}_2$. □

Using this lemma we can update Step 6 to only include a single element of each variable orbit of $\mathbf{X}$ under $\mathbb{A}(\mathtt{G}(\mathcal{F}, \mathbf{x}))$.

4.3.3   Exact Lifted Inference Algorithm

This section will combine the theory of the previous two sections to perform exact lifted inference on factor graphs. Algorithm 1 performs exact lifted inference via a breadth-first search over canonical assignments. Variable r holds a set of canonical representatives, q holds the frontier, p accumulates the unnormalized probability of the evidence, and Z accumulates the normalizing constant. A graph isomorphism tool is used to compute σ

(a) Inference for pairwise factor graph.

(b) Inference for 2-hole pigeonhole problem.

Figure 4.5: Evaluation of Algorithm 1. A red circle indicates that `Ace` ran out of memory at that time.

on Line 5. Each time the algorithm finds a new representative, it computes the size of the orbit using the orbit stabilizer theorem on Line 9; `GAP` is used to compute the order of these permutation groups. Lemma 4.1 is used on Line 13 to avoid adding augmentations to the frontier that are known a-priori to be isomorphic to prior ones. This algorithm can be easily modified to produce the MPE by simply returning the canonical representative from r with the highest probability.

Experimental Evaluation To validate the proposed method Algorithm 1 was implemented using the `Sage` math library, which wraps `GAP` and a graph isomorphism tool [The Sage Developers, 2018].[4] The lifted inference procedure is compared against `Ace`, an exact inference tool for discrete Bayesian networks that is unaware of the symmetry of the model [Chavira and Darwiche, 2005]. Figure 4.5 shows experimental results for performing exact lifted inference on two families of factor graphs. The first is a class of pairwise factor graphs that have an identical symmetric potential between all nodes, with one factor (in red) designated as an evidence factor:
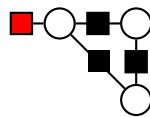
Figure 4.5b evaluates exact lifted inference on the pigeonhole problem from Chapter 5.1 with two holes and increasing number of pigeons. In both experiments, the number of orbits grows linearly, even though there is little independence. Thus, `Ace` scales exponentially, since the treewidth grows quickly, while the lifted method appear to scale subexponentially. This is the first example of performing exact inference on this family of models.

---

[4]The source code for the exact and approximate inference algorithms can be found at `https://github.com/SHoltzen/orbitgen`.

Algorithm 1: `ExactLiftedInference`$(\mathcal{F}, \mathbf{e})$

    Data: A factor graph $\mathcal{F} = (\mathbf{X}, F)$ with color encoding $\mathtt{G}$; $\mathbb{A}(\mathtt{G})$-invariant evidence $\mathbf{e}$

    Result: The probability of evidence $\Pr(\mathbf{e})$

1 $r \leftarrow$ empty set, $p \leftarrow 0$, $Z \leftarrow 0$;

2 $q \leftarrow$ queue containing the all-false assignment;

3 **while** *q is not empty* **do**

4 $\mathbf{x} \leftarrow q.\mathrm{pop}()$;

5 $\mathtt{Canon} \leftarrow \sigma(\mathtt{G}(\mathcal{F}, \mathbf{x}))$;   `// Invoke graph iso. tool`

6   **if** $\mathtt{Canon} \in r$ **then**

7 | continue;

8   **end**

9   Insert $\mathtt{Canon}$ into r;

10 $|\mathrm{Orb}(\mathbf{x})| \leftarrow |\mathbb{A}(\mathtt{G})|/|\mathrm{Stab}_{\mathbb{A}(\mathtt{G})}(\mathbf{x})|$;   `// Invoke GAP`

11   **if** $\mathbf{e}(\mathbf{x}) = \mathtt{T}$ **then**

12 | $p \leftarrow p + |\mathrm{Orb}(\mathbf{x})| \times F(\mathbf{x})$;

13   **end**

14 $Z \leftarrow Z + |\mathrm{Orb}(\mathbf{x})| \times F(\mathbf{x})$;

15   **for** *o from each variable orbit of* $\mathrm{Stab}_{\mathbb{A}(\mathtt{G})}(\mathbf{x})$ **do**

16   **if** *o is a false variable* **then**

17 | $\mathbf{x}' \leftarrow \mathbf{x}$ with o true;

18 | Append $\mathbf{x}'$ to q;

19   **end**

20   **end**

21 **end**

22 **return** p/Z

4.4 Orbit-Jump Markov-Chain Monte Carlo

This section introduces *orbit-jump MCMC*, an MCMC algorithm that mixes quickly when the distribution has few orbits, at the cost of requiring multiple graph isomorphism calls for each transition. The algorithm is summarized in Algorithm 2. Orbit-jump MCMC is an alternative to Lifted MCMC [Niepert, 2012, 2013] that generates provably high-quality samples at the expense of more costly transitions. Lifted MCMC exploits symmetric structure to quickly transition *within* orbits. Lifted MCMC is efficient to implement: it requires only a single call to a graph isomorphism tool. However, lifted MCMC relies on Gibbs sampling to jump *between orbits*, and therefore has no guarantees about its mixing time for distributions with few orbits. Orbit-jump MCMC is a Metropolis-Hastings MCMC
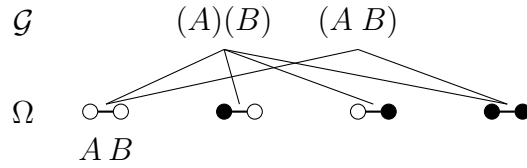
Figure 4.6: Illustration of the Burnside process on a colored graph with two nodes and two colors.

algorithm that uses the following distribution as its proposal:[5]

Definition 4.9. *Let $\mathcal{G}$ act on Ω. Then for $x \in \Omega$, the* uniform orbit distribution *is:*

$$\mathrm{Pr}_{\Omega/\mathcal{G}}(x) \triangleq \frac{1}{|\Omega/\mathcal{G}| \times |\mathrm{Orb}(x)|} \tag{4.8}$$

This is the probability of uniformly choosing an orbit $o \in \Omega/\mathcal{G}$, and then sampling uniformly from $\sigma^{-1}(o)$.

The *orbit-jump MCMC chain* for a $\mathcal{G}$-invariant distribution Pr is defined as follows, initialized to $x \in \Omega$:

A step in orbit-jump MCMC

1. Sample $x' \sim \mathrm{Pr}_{\Omega/\mathcal{G}}$;
2. Accept x' with probability $\min\left(1, \frac{\mathrm{Pr}(x') \times |\mathrm{Orb}(x')|}{\mathrm{Pr}(x) \times |\mathrm{Orb}(x)|}\right)$

This Markov chain has Pr as its stationary distribution. Orbit-jump MCMC has a high probability of proposing transitions *between* orbits, which is an alternative to the within-orbit exploration of lifted MCMC.[6]

Next we will describe how to sample from $\mathrm{Pr}_{\Omega/\mathcal{G}}$ using an MCMC method known as the *Burnside process*. Then, we will discuss the mixing time of this proposal, and prove that it mixes in the number of orbits of the distribution.

4.4.1 Sampling From the Uniform Orbit Distribution

Jerrum [1993] gave an MCMC technique known as the *Burnside process* for drawing samples from $\mathrm{Pr}_{\Omega/\mathcal{G}}$. The Burnside process is a Markov Chain Monte Carlo method defined

---

[5]For a good introduction Metropolis-Hastings in the context of probabilistic models, see Murphy [2012, Chapter 24].

[6]This proposal is independent of the previous state, a scheme that is sometimes called *Metropolized independent sampling* (MIS) [Liu, 1996]. Importance sampling is an alternative to MIS. We use MIS rather than importance sampling in order to make the connection with lifted MCMC more explicit.

as follows, beginning from some $x \in \Omega$:

> **A step in the Burnside process**
>
> 1. Sample $g \sim \text{Stab}(x)$ uniformly;
> 2. Sample $x \sim \text{Fix}(g)$ uniformly, where $\text{Fix}(g) = \{x \in \Omega \mid g \cdot x = x\}$. Elements of $\text{Fix}(g)$ are called *fixers*.

Theorem 4.6 (Jerrum [1993])**.** *The stationary distribution of the Burnside process is equal to* $\text{Pr}_{\Omega/\mathcal{G}}$.

This process can be visualized as a random walk on a bipartite graph. One set of nodes are elements of Ω, and the other set are elements of $\mathcal{G}$. There is an edge between $x \in \Omega$ and $g \in \mathcal{G}$ iff $g \cdot x = x$.

An example of this bipartite graph is shown in Figure 4.6. The set Ω is the set of 2-node colored graphs, and the group $\mathcal{G} = S_2$ permutes the vertices of the graph. The identity element $(A)(B)$ stabilizes all elements of Ω, and so has an edge to every element in x; $(A\ B)$ only stabilizes graphs whose vertices have the same color.

Jerrum [1993] proved that the Burnside process mixes rapidly for several important groups, but it does not always mix quickly [Goldberg and Jerrum, 2002]. In such cases, it is important to draw sufficient samples from the Burnside process in order to guarantee that the orbit-jump proposal is unbiased. Next we will describe how to implement the Burnside process on factor graphs using the machinery from Chapter 4.3.2.1.

4.4.1.1 Burnside Process on Factor Graphs

For $\mathcal{G}$ acting on a set of variables $\mathbf{X}$, the Burnside process requires the ability to (1) draw samples uniformly from the stabilizer subgroup of an assignment to variables, and (2) sample a random fixer for any group element in $\mathcal{G}$. Here we describe how to perform these two computations for a colored factor graph $\mathcal{F} = (\mathbf{X}, F)$.[7] This procedure is summarized in lines 3–7 in Algorithm 2.

4.4.1.2 Stabilizer Sampling

Chapter 4.3.2.1 showed how to compute the stabilizer group of $\mathbf{x} \in \mathbf{Bool^X}$ using graph isomorphism tools. Sampling uniformly from the stabilizer group relies on the *product replacement algorithm*, which is an efficient procedure for uniformly sampling group elements [Pak, 2000]. This step occurs on Line 4 of Algorithm 2.

---

[7]This process is conceptually similar to the procedure for randomly sampling orbits in the Pólya-theory setting described by Goldberg [2001], but this is the first time that this procedure is applied directly to factor graphs

---
Algorithm 2: A step of Orbit-jump MCMC

Data: A factor graph $\mathcal{F} = (\mathbf{X}, F)$, a point $\mathbf{x} \in \mathbf{Bool^X}$, number of Burnside process steps k

1   $\mathbf{x}' \leftarrow \mathbf{x}$;
2   **for** $i \in \{1, 2, \cdots, k\}$ **do**
3      $\mathcal{G}_{\text{Stab}} \leftarrow \mathbb{A}(\mathtt{G}(\mathcal{F}, \mathbf{x}'))$;   `// Invoke graph iso. tool`
4   Sample $s \sim \mathcal{G}_{\text{Stab}}$ using product replacement;
5   **for** *Each variable cycle c of s* **do**
6          $v \sim \text{Bernoulli}(1/2)$;
7   Assign all variables c in $\mathbf{x}'$ to v;
8   **end**
9   **end**
10 Accept $\mathbf{x}'$ with probability $\min\left(1, \frac{F(\mathbf{x}') \times |\text{Orb}(\mathbf{x}')|}{F(\mathbf{x}) \times |\text{Orb}(\mathbf{x})|}\right)$

4.4.1.3   Fixer Sampling

Let $g \in \mathcal{G}$ be a permutation that acts on the vertices of a colored factor graph. Then we uniformly sample an assignment-encoded colored factor graph that is fixed by g in the following way. First, decompose g into a product of disjoint cycles. Then, for each cycle that contains variable nodes, choose a truth assignment uniformly randomly, and then color the vertices in that cycle with that color. This colored graph is fixed by g and is uniformly random by the independence of coloring each cycle and the fact that all colorings fixed by g can be obtained in this manner. This step occurs on lines $5 - 8$ in Algorithm 2.

4.4.2   Mixing Time of Orbit-Jump MCMC

The *total variation distance* between two discrete probability measures μ and ν on Ω, denoted $d_{TV}(\mu, \nu)$, is:

$$d_{TV}(\mu, \nu) = \frac{1}{2} \sum_{x \in \Omega} |\mu(x) - \nu(x)|. \tag{4.9}$$

The *mixing time* of a Markov chain is the minimum number of iterations that the chain must be run starting in any state until the total variation distance between the chain and its stationary distribution is less than some parameter $\varepsilon > 0$. The mixing time of orbit-jump MCMC can be bounded in terms of the number of orbits, which is a property not enjoyed by lifted MCMC:

Theorem 4.7. *Let* $\Pr$ *be a* $\mathcal{G}$*-invariant distribution on* Ω *and let* P *be the transition matrix of*
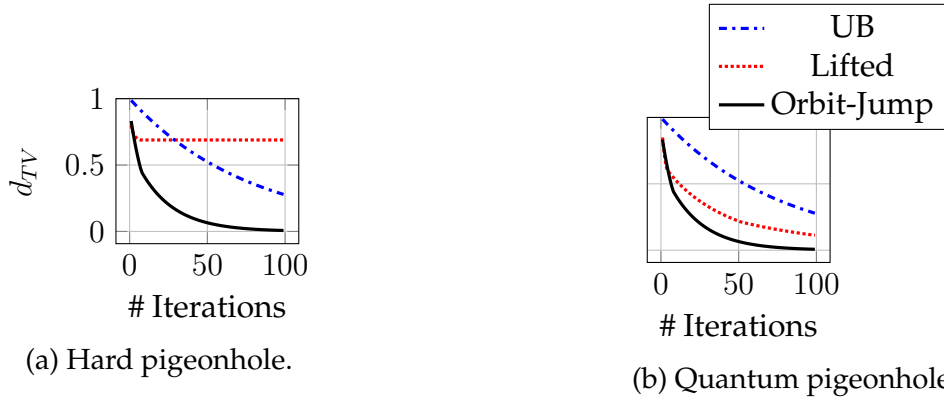
(a) Hard pigeonhole.

(b) Quantum pigeonhole.

Figure 4.7: Total variation distance between Markov chains and their stationary distributions for a pigeonhole problem with 5 pigeons and 2 holes. "Lifted" is lifted MCMC [Niepert, 2012] and "UB" is the upper bound predicted by Theorem 4.7.

orbit-jump MCMC. Then, for any $x \in \Omega$, $d_{TV}(P^t x, \mathrm{Pr}) \leq \left(\frac{|\Omega/\mathcal{G}|-1}{|\Omega/\mathcal{G}|}\right)^t$. *It follows that for any* $\varepsilon > 0$, $d_{TV}(P^t x, \mathrm{Pr}) \leq \varepsilon$ *if* $t \geq \log(\varepsilon^{-1}) \times |\Omega/\mathcal{G}|$.

Proof. See Appendix A.2. □

Note that the bound on this mixing time does not take into account the cost of drawing samples from $\mathrm{Pr}_{\Omega/\mathcal{G}}$, which involves multiple graph isomorphism calls.

4.4.2.1 Pigeonhole case study

In order to empirically evaluate its performance, the orbit-jump MCMC procedure on factor graphs was implemented using Sage. The mixing time of lifted MCMC [Niepert, 2012, 2013] and orbit-jump MCMC are compared in Figure 4.7, which computes the total variation distance of these two MCMC methods from their stationary distribution as a function of the number of iterations on two versions of the pigeonhole problem.[8] The first version in Figure 4.7a is the motivating example with hard constraints from Chapter 5.1. The second version in Figure 4.7b shows a *"quantum" pigeonhole problem*, where the constraint in Equation 4.4 is relaxed so that pigeons are allowed to be placed into multiple holes (i.e., the case when $x_{iB} = x_{jB} = 1$ is given a finite negative weight.).

Lifted MCMC fails to converge in Figure 4.7a because it cannot transition due to the hard constraint from Equation 4.4; this illustrates that lifted MCMC can fail even for distributions with few orbits. In addition to comparing against lifted MCMC, we also compare the theoretical upper bound from Theorem 4.7 against the two mixing times. This

---

[8]In these experiments, for each step of orbit-jump MCMC, we use 7 steps of the Burnside process.

upper bound only depends on the number of orbits, and does not depend on the parameterization of the distribution.[9] Orbit-jump MCMC converges to the true distribution in both cases faster than lifted MCMC, and the upper bound ensures that orbit-jump MCMC cannot get stuck in low-probability orbits. Note however that lifted MCMC transitions are less expensive to compute than orbit-jump MCMC transitions. We hope to explore this practical tradeoff between sample quality and the cost of drawing a sample in future work.

4.5   Conclusion

This chapter provided the first exact and approximate lifted inference algorithms for factor graphs that provably scale in the number of orbits. However, the presented methods are limited: there are tractable highly symmetric distributions that still have too many orbits for these methods to be effective. Existing lifted inference algorithms utilize independence to extract highly symmetric sub-problems, which is an avenue for integrating independence into this current approach. A further limitation of the approach is that it exploits only symmetries on variables; additional forms of symmetries, such as block symmetries, are beyond the scope of the presented algorithms [Madan et al., 2018].

4.6   Bibliographic Notes

Lifted inference   Existing exact lifted inference algorithms apply to relational models [Getoor and Taskar, 2007]. The tractability of exact lifted inference was studied by Niepert and Van den Broeck [2014], but their approach cannot be directly applied to factor graphs. Approximate lifted inference can be applied to factor graphs, but existing approaches do not provably mix quickly in the number of orbits [Niepert, 2012, 2013, Bui et al., 2013, Van den Broeck and Niepert, 2015, Madan et al., 2018, Kersting et al., 2009, Gogate et al., 2012].

Symmetry in constraint satisfaction and logic   Some techniques for satisfiability and constraint satisfaction also exploit symmetry. The goal in that context is to quickly select one of many symmetric candidate solutions, so a key difference is that in the current setting one must exhaustively explore the search space. Sabharwal [2005] augments a SAT-solver with symmetry-aware branching capabilities. Symmetry has also been exploited in integer-linear programming [Margot, 2010, Ostrowski et al., 2007, Margot, 2003].

---

[9]For this example, there are 78 orbits.

CHAPTER 5

Composing Inference Algorithms

> *Abstraction is all relative; one person's abstraction is another person's bread and butter.*

<div align="right">Charles Pinter</div>

Thus far this thesis has introduced two new strategies for performing inference in probabilistic programs that both work in very different ways and exploit different program structure. Chapter 3 showed how to exploit factorization in order to scale inference, but it did so at the cost of supporting other kinds of program features like continuous random variables and unbounded loops. Chapter 4 showed how to exploit symmetry, but at the expense of exploiting factorization. There are many more inference algorithms that make different tradeoffs between tractability and expressivity, so this motivates the following key problem: *how can we mix and match inference algorithms depending on the kind of structure exhibited by a heterogeneous probabilistic program?*

This chapter gives a method for decomposing probabilistic program inference via *program abstraction*. Program abstractions – and in particular *predicate abstractions* – have a rich and successful history in non-probabilistic program analysis [Ball et al., 2001, Cousot and Cousot, 1977]. The key idea is to generate a simplified *abstract program* from the original *concrete program* that captures a few key properties. This abstraction property simplifies the analysis – the new abstract program is by design simpler to analyze than the concrete program. Ideally the abstract program will then contain sufficient information to verify various properties of the concrete program.

This chapter gives a generalization for non-probabilistic predicate abstraction to probabilistic programs and shows how it decomposes inference. An outline is given in Figure 5.1. First, the programmer provides three pieces of data: a probabilistic program, a set of *predicates* that are Boolean random variables that capture properties about the program, and a query. Then, the abstraction engine automatically generates (1) a set of sub-
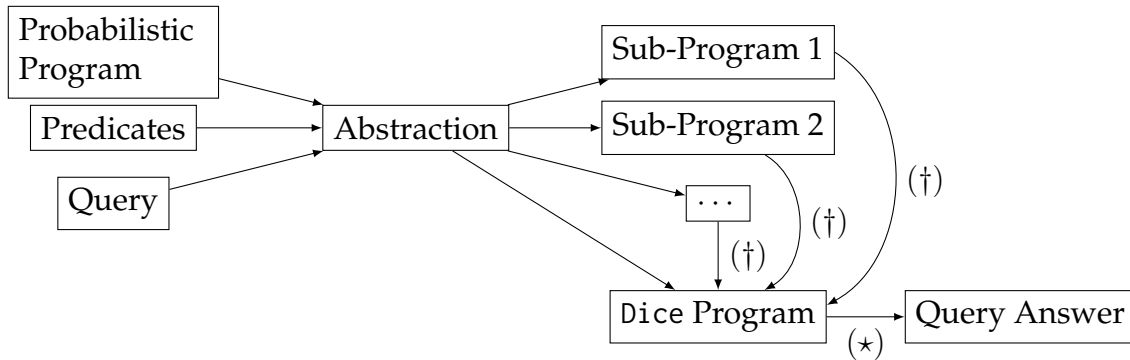

Figure 5.1: Diagram describing decomposition by abstraction.

programs that are themselves probabilistic programs, and (2) an abstract `Dice` program that captures the relationship among predicates.

In order to evaluate the final query, the abstract `Dice` program is *parameterized* by querying the sub-programs. Each (†) arrow in the figure represents a sub-query that queries a small part of the original program: this is the stage where inference is decomposed, since evaluating these (†) queries will ideally only require inspecting smaller portions of the original program. Finally, the final query is answered via a standard `Dice` inference query along the (⋆) arrow, as outlined in Chapter 3.

Formally, as an outline, (1) Chapter 5.1 introduces a new notion of probabilistic predicate abstractions and shows how to automatically generate them from a probabilistic program; (2) Chapter 5.2 gives background; (3) Chapter 5.3 gives a new soundness relation between abstract and concrete program called *distributional soundness*; (4) Chapter 5.4 shows how to construct distributionally sound abstractions; and (5) Chapter 5.5 shows empirically how distributionally sound abstractions decompose probabilistic in order to speed up inference.

5.1 Motivating Example

Probabilistic programs can exhibit complex structure. In particular, they admit complex operations such as control-flow logic and numerical manipulation, which entangle random variables in ways that are difficult to reason about. Consider Figure 5.2a, which shows a simple probabilistic program that combines two random variables via multiplication. We wish to compute the query $\Pr(z = 0)$ on this program. Initially, this seems to be difficult since the variables x and y are entangled via multiplication. In a typical probabilistic programming system such as Stan, Psi, or Anglican, this query would be evaluated by sampling or integration beginning on Line 1 of the program Carpenter et al. [2016], Gehr et al. [2016], Wood et al. [2014]. This would require jointly integrating over the random variables x and y (or approximating the integral by sampling).

73

```
1  x ←discrete_dist();
2  y ←continuous_dist();
3  z ← x * floor(y);
```

(a) A concrete probabilistic program. Probabilistic sub-program `discrete_dist` returns a discrete random variable. Sub-program `continuous_dist` returns a continuous random variable. `floor` rounds down to the nearest integer.
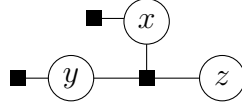
(b) A factor graph which captures the conditional independences in Figure 5.2a. Factors ■ encapsulate the two sub-programs and represent dependencies between the three random variables x, y, and z.

```
1  {x = 0} ←flip(θ_{x=0});
2  {0 ≤ y < 1} ←flip(θ_{0≤y<1});
3  {z = 0} ← {x = 0} ∨ {0 ≤ y < 1};
```

(c) A probabilistic program which captures the distribution only on the predicates $\{x = 0\}$, $\{0 \leq y < 1\}$, and $\{z = 0\}$. A `flip(θ)` expression is true with probability θ.

Figure 5.2: Abstracting a probabilistic program as a factor graph and a probabilistic predicate abstraction.

One option for potentially simplifying inference on this program is to generate a factor graph abstraction on which to perform inference, which is the approach taken by compilation techniques such as Factorie, Infer.Net and Figaro [McCallum et al., 2009, Minka et al., 2014, Pfeffer, 2009]. Figure 5.2b shows such a factor graph abstraction. The parameters of this factor graph are chosen so that it is a *distributionally sound abstraction*: it is possible to instantiate the factors in such a way that the graphical model exactly captures the probabilistic program's intended distribution. However, a key disadvantage is that the graph-based abstraction may be overly coarse, disregarding key structural aspects of the program.

For this example, the abstraction is overly coarse, and thus during inference it yields no useful decompositions. From the perspective of the graph, all three random variables are inextricably linked via an opaque factor. Thus, computing $\Pr(z = 0)$ on the factor graph abstraction would require jointly integrating x and y. Nonetheless, observe that this factor is actually highly structured: in the program, z is linked to x and y via a deterministic multiplication. We wish to exploit this structure.

This chapter proposes to instead utilize a *simpler probabilistic program* as the abstraction, rather than a graph. Specifically, this probabilistic program will only model the distribution on a collection of Boolean predicates – statements about the original program which are true or false. Since it only models Boolean predicates, it will be a discrete prob-

abilistic program and hence we can use `Dice` to perform inference. The parameters of this probabilistic program will be chosen so that it is distributionally sound with respect to the original program. In this chapter, we show how to automatically produce a distributionally sound abstraction for a given program relative to a given set of predicates. While a distributionally sound abstraction always exists, whether that abstraction is informative depends on the choice of predicates. This approach assumes that the predicates are provided *a priori*; automated techniques for selecting predicates are left as future work and discussed in more detail in Chapter 5.4.1.

Let us consider an example of the flexibility of probabilistic programs as a language for abstraction: capturing a nuanced decomposition which relies on properties of multiplication. Observe that the program in Figure 5.2a has the following property: after executing Line 3, $z = 0$ if and only if $x = 0$ or $0 \leq y < 1$. The present notion of abstraction is capable of representing this relationship, and this approach can automatically produce such an abstraction.

Given the three predicates above, the goal is to automatically generate the abstract probabilistic program in Figure 5.2c, which only models the distribution on the three predicates; such abstractions are a specific kind of *probabilistic predicate abstraction* [Holtzen et al., 2017]. This step is part of the "Abstraction" box in Figure 5.1. Denote the Boolean variable that corresponds with a predicate as $\{\cdot\}$. In order for this abstraction to be distributionally sound, it requires the correct parameterization. In this case, we must compute two *sub-queries* on the original probabilistic program:

$$\theta_{x=0} = \Pr(\texttt{discrete\_dist()} = 0)$$
$$\theta_{0 \leq y < 1} = \Pr(0 \leq \texttt{continuous\_dist()} < 1)$$

With these parameters, Figure 5.2c is distributionally sound; computing $\Pr(\{z = 0\})$ on this abstract program will yield the same result as computing $\Pr(z = 0)$ on the original program. Therefore, in the process of parameterizing this abstraction, the concrete program has been decomposed: at no point were we required to jointly integrate x and y. Further, each of the two sub-queries can be answered using the inference method that is most suitable for it, which may be different for discrete and continuous distributions. This motivating example raises the following questions, which the remainder of the chapter will be devoted to answering:

Formalization What is a distributionally sound probabilistic program abstraction? (Chapter 5.3)

Existence For a fixed choice of predicates, can a distributionally sound abstraction always be generated? What is an algorithm for doing so? (Chapter 5.4)

Usefulness What are the benefits of constructing and querying a distributionally sound abstraction over querying the original program? (Chapter 5.5)

5.2 Background

The goal of this section is to provide a concise background in semantics of probabilistic programming languages and program abstractions. The formalism used here for describing probabilistic programs is subtly different than that laid out in the prior Dice chapters, since here we consider a broader class of programs. First this section gives a brief review and expansion of the language of probability theory. Then, this language is used to give the semantics of probabilistic programming languages. Finally, it introduces the necessary background from program analysis: predicate abstractions and weakest preconditions.

5.2.1 Probability Theory

This chapter will require some standard notions from probability theory such as a measurable space, probability space, and measurable function. Probability spaces are denoted (Ω, Σ, μ), where Ω is a sample space, Σ is a σ-algebra on Ω, (Ω, Σ) is a measurable space, and μ is a probability measure. Of particular importance in this chapter is the notion of a push-forward probability measure:

Definition 5.1 (Push-forward). *Let (Ω, Σ, μ) be a probability space and (Ω', Σ') be a measurable space. Let $f : \Omega \to \Omega'$ be a measurable function. Then, the* push-forward *of μ through f is a probability measure ν on (Ω', Σ') such that for any $e \in \Sigma'$, $\nu(e) = \mu(f^{-1}(e))$. As notation, we sometimes treat f as a mapping between probability spaces.*

Some standard notation and concepts from probability theory are necessary during this chapter's formalization of probabilistic programs. First, we define a measurable space:

Definition 5.2 (Measurable space). *Let Ω be a set, called the* sample space. *In the context of programs Ω is sometimes called a domain. A σ-algebra Σ on Ω is a collection of subsets of Ω that is (i) closed under countable unions; (ii) closed under complementation; (iii) contains Ω. We call the pair (Ω, Σ) a* measurable space.

We will rely on the notion of a probability space: a measurable space with a probability measure.

Definition 5.3 (Probability space). *Let (Ω, Σ) be a measurable space and $\mu : \Sigma \to \mathbb{R}$ be a function such that (i) μ is countably additive; (ii) $\mu(\Omega) = 1$. The tuple (Ω, Σ, μ) is called a* probability space, *and μ is called a* probability measure.

Measurable spaces afford a particular class of functions called *measurable functions*. Intuitively, such functions represent a random variable.

Definition 5.4 (Measurable function). *Let (Ω, Σ) and (Ω', Σ') be two measurable spaces. Then a function $f : \Omega \to \Omega'$ is called a* measurable function *if for any $E \in \Sigma'$, we have that $f^{-1}(E) = \{x \in \Omega \mid f(x) \in E\} \in \Sigma$.*

Measurable functions define a transformation between probability spaces known as a *push-forward*:

Definition 5.5 (Push-forward). *Let (Ω, Σ, μ) be a probability space and (Ω', Σ') be a measurable space, and $f : \Omega \to \Omega'$ be a measurable function. Then, the* push-forward *of μ through f is a probability measure ν on (Ω', Σ') such that for any $e \in \Sigma'$, $\nu(e) = \mu(f^{-1}(e))$. As notation, we sometimes treat f as a mapping between probability spaces.*

5.2.2 Semantics of Probabilistic Programs

The probabilistic programs that this chapter studies are defined in two parts: the first part assigns an initial probability distribution to variables, and the second produces a new probability measure that results from the manipulation of these variables through the composition of measurable functions.[1]

Definition 5.6 (Semantics of probabilistic programs). *A probabilistic program p has two semantic components:*

1. *An initial probability space (Ω, Σ, μ). The sample space Ω is the set of joint states of the variables in the program.*
2. *A measurable function $[\![p]\!] : \Omega \to \Omega'$. It is implied that there exists some σ-algebra Σ' on Ω' such that (Ω', Σ') form a measurable space.*

We say the probability measure induced by p is the probability measure which results from pushing μ through $[\![p]\!]$.

This style of semantics does not reason about arbitrary unbounded loops or higher-order functions, as these cannot in general be represented as measurable functions [Aumann, 1961]. However, measurable functions typically form a core component of the underlying semantics of higher-order and loopy programming languages, allowing the abstraction technique described in this chapter to be applied to measurable sub-programs within such languages [Kozen, 1979]. Further, many existing useful probabilistic programming languages do not have loops.

5.2.3 Predicate Abstraction

Predicate abstraction is a common and effective form of program analysis [Graf and Saïdi, 1997, Ball et al., 2001]. At a high level, the goal is to generate an abstract program that is

---

[1]This two-part style of semantics is used by the popular probabilistic programming language Stan [Carpenter et al., 2016].

easier to analyze than the original program, while maintaining a meaningful relationship – known as *soundness* – with the original program. The traditional soundness property for predicate abstraction is *over-approximation*, the property that the abstraction contains the original program's behavior as a subset of its own. This is useful for proving safety properties: for instance, if the abstraction never divides an integer by zero, then neither does the original program.

The way a predicate abstraction accomplishes this feat is by generating an abstract program that only manipulates a selection of *Boolean predicates*. A predicate is a property of the domain of the concrete program. For example, a predicate on the concrete variable x may be $\{x < 4\}$. A collection of predicates forms a predicate domain:

Definition 5.7 (Predicate domain). *Let Ω be a domain, and let $\Psi = \{\psi_1, \psi_2, \cdots, \psi_n\}$ be a collection of predicates on Ω. Then the* predicate domain $\mathcal{D}_A$ *over Ψ is the set of all 2^n truth assignments to the predicates in Ψ.*

As notation, let c be a concrete state. We write $[c]$ to denote the abstract state corresponding with the predicates that hold for c, and $[a]^{-1} = \{c \mid [c] = a\}$ for its inverse. When necessary, we use the subscript $[\cdot]_\Psi$ to denote abstract states with respect to a particular set of predicates Ψ.

When the collection of predicates Ψ is insufficient to capture the behavior of the concrete program, the abstraction must behave *non-deterministically* in order to remain an over-approximation. All of these definitions are best illustrated with an example:

Example 5.1: A simple predicate abstraction

Consider the concrete program $\mathcal{C} = $ x ← x + 1;, which simply increments a variable x. We consider the predicate domain $\Psi = \{x < 0\}$. The goal is to generate an abstract program $\mathcal{A}$ that represents how the predicate $\{x < 0\}$ changes as a result of this assignment to x. Specifically, if x is negative before incrementing, it could remain negative or become non-negative: in this case, we conservatively allow the predicate to take either value. However, if x is non-negative, it is guaranteed to remain non-negative after incrementing. We can write this update using the syntax of a programming language, denoting a non-deterministic Boolean choice with the $\ast$ symbol:

$$\{x<0\} \leftarrow \{x<0\} \wedge \ast;$$

Note that all logical connectives can be naturally extended to a 3-valued over the values $(\text{T}, \text{F}, \ast)$ where $\ast$ represents nondeterminism. For instance, conjunction is naturally extended by defining $\text{T} \wedge \ast \triangleq \ast$, $\text{F} \wedge \ast \triangleq \text{F}$, and so on.

An over-approximate predicate abstraction can be automatically generated for a program relative to a given set of predicates [Ball et al., 2001]. The process of constructing a pred-

icate abstraction relies on the ability to compute a *weakest precondition*, a tool which will be utilized in later technical sections and can be computed automatically for loop-free programs [Dijkstra, 1976]:

Definition 5.8 (Weakest precondition). *Let p be a program and ϕ be a predicate. Then the weakest precondition of p with respect to ϕ, denoted $\mathbf{WP}(p, \phi)$, is the most general predicate ψ such that ψ holding before executing p implies that ϕ holds after executing p.*

5.3 Distributional Soundness

Traditional over-approximate predicate abstractions are insufficient as abstractions for probabilistic programs since they are not *distributionally sound*: they do not preserve the distributions of the given predicates in the original program. In particular, the use of non-determinism is not compatible with distributional soundness; for example, the abstraction shown in Example 5.1 does not preserve $\Pr(x < 0)$ from the original program. This section formally defines what it means for a predicate abstraction $\mathcal{A}$ that manipulates variables from a predicate domain $\mathcal{D}_\mathcal{A}$ to be distributionally sound for a given concrete probabilistic program $\mathcal{C}$.

First we require a way of connecting the concrete and abstract initial probability spaces. There is a straightforward mapping of probability measures on the concrete domain to probability measures on the abstract domain, simply by evaluating the concrete measure for each abstract state's equivalence class.

Definition 5.9 (Probabilistic abstraction function). *Let (Ω, Σ, μ) be a probability space and $(\mathcal{D}_\mathcal{A}, \Sigma_{\mathcal{D}_\mathcal{A}})$ be a measurable space where the sample space is a predicate domain $\mathcal{D}_\mathcal{A}$ over predicates Ψ. Then, a probabilistic abstraction function $\alpha : (\Omega, \Sigma, \mu) \to (\mathcal{D}_\mathcal{A}, \Sigma_{\mathcal{D}_\mathcal{A}}, \nu)$, is defined as the push-forward of μ through $[\cdot]$.*

Now utilizing this definition we give the formal notion of distributional soundness:

Definition 5.10 (Distributional soundness). *Let $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ and $[\![\mathcal{A}]\!] : \mathcal{D}_\mathcal{A} \to \mathcal{D}_{\mathcal{A}'}$ be measurable functions, where $\mathcal{D}_\mathcal{A}$ and $\mathcal{D}_\mathcal{A}'$ are predicate domains on Ω and Ω' respectively. Then $[\![\mathcal{A}]\!]$ is a distributionally sound abstraction of $[\![\mathcal{C}]\!]$ if the following diagram commutes for any initial concrete probability space (Ω, Σ, μ):*

$$
\begin{array}{ccc}
(\Omega, \Sigma, \mu) & \xrightarrow{\ [\![\mathcal{C}]\!]\ } & (\Omega', \Sigma', \mu') \\
\downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha'} \\
(\mathcal{D}_\mathcal{A}, \Sigma_{\mathcal{D}_\mathcal{A}}, \nu) & \xrightarrow{\ [\![\mathcal{A}]\!]\ } & (\mathcal{D}_{\mathcal{A}'}, \Sigma_{\mathcal{D}_{\mathcal{A}'}}, \nu')
\end{array}
$$

Distributional soundness requires that the probability of a predicate being true in the abstraction is *equal* to the probability of the corresponding predicate being true in the concrete program. This in turn implies that inference on the abstraction is sound for queries that can be defined in terms of the predicates in $\mathcal{D}_{\mathcal{A}'}$. Specifically, we describe a class of events for which we can perform inference using exclusively the abstraction:

Definition 5.11 (Corresponding events). *Let (Ω, Σ, μ) be a probability space, $(\mathcal{D}_\mathcal{A}, \Sigma_{\mathcal{D}_\mathcal{A}})$ be a measurable space over predicate domain $\mathcal{D}_\mathcal{A}$, and $[\cdot]$ be an abstraction function. Then for any abstract event $e_{\mathcal{D}_\mathcal{A}} \in \Sigma_{\mathcal{D}_\mathcal{A}}$, there exists a corresponding concrete event $e_\Omega = \bigcup \{[a]^{-1} \mid a \in e_{\mathcal{D}_\mathcal{A}}\}$. We call the pair $(e_{\mathcal{D}_\mathcal{A}}, e_\Omega)$ an* event pair.

Formally, the abstraction can be used to reason about the concrete program by utilizing event pairs:

Proposition 5.1 (Distributional soundness implies soundness for inference). *Let $[\![\mathcal{A}]\!] : \mathcal{D}_\mathcal{A} \to \mathcal{D}_{\mathcal{A}'}$ be a distributionally sound abstraction of $[\![\mathcal{C}]\!] : \Omega \to \Omega$. Then for any initial probability space (Ω, Σ, μ), and any event pair $(e_{\mathcal{D}_{\mathcal{A}'}}, e_{\Omega'})$, it is the case that $\mathrm{Pr}_{\nu'}(e_{\mathcal{D}_{\mathcal{A}'}}) = \mathrm{Pr}_{\mu'}(e_{\Omega'})$, where μ' is the push-forward of μ through $[\![\mathcal{C}]\!]$ and ν' is the push-forward of μ through $[\![\mathcal{A}]\!] \circ [\cdot]$.*

Graph-based abstractions often serve as a semantic tool, by asserting independences that are assumed to hold in the distribution of interest. For probabilistic program abstractions, distributional soundness guarantees that the abstraction is able to exactly capture the concrete program's distribution over some key predicates:

Proposition 5.2 (Independence Assumptions). *Let $\mathcal{C}$ be a concrete probabilistic program and let $[\![\mathcal{A}]\!]$ be a distributionally sound abstraction of $[\![\mathcal{C}]\!]$. Then any conditional independence that holds between abstract events $e_{\mathcal{D}_\mathcal{A}} \in \Sigma_{\mathcal{D}_\mathcal{A}}$ in $\mathcal{A}$ also holds between the corresponding concrete events $e_\Omega \in \Sigma_\Omega$ in $\mathcal{C}$.*

Distributionally sound abstractions are a powerful technique for reasoning about probabilistic programs: they allow one to reason about a simplified program that only manipulates a collection of predicates. The obvious question is: can we always construct such a distributionally sound abstraction for an arbitrary choice of predicates? The following section answers this question affirmatively.

5.4 Constructing Sound Abstractions

The goal of this section is to provide a technique to automatically generate a distributionally sound abstraction for a given concrete program and set of predicates. In particular, it describes how to implement the "Abstraction" box in Figure 5.1. As input the algorithm takes a concrete program $\mathcal{C}$ and a set of predicates Ψ of interest. Then, it constructs a distributionally sound abstraction $\mathcal{A}$, which consists of two parts: (1) a measurable function

$[\![\mathcal{A}]\!]$, and (2) an initial abstract probability space such that the diagram in Definition 5.10 commutes.

Given a user-provided $\mathcal{C}$ and Ψ it is not always possible to generate a distributionally sound abstraction from $\mathcal{D}_A$ to $\mathcal{D}_A$, because the predicates in Ψ might not be sufficiently expressive to capture all the required concrete behavior in order to maintain the original distribution. This is resolved by automatically identifying predicates called *completions* (denoted Φ), which are added to Ψ, yielding a new set of predicates $\Psi \cup \Phi$. Then, we generate a distributionally sound abstraction $\mathcal{A}$ with measurable function $[\![\mathcal{A}]\!] : \mathcal{D}_{\Psi \cup \Phi} \to \mathcal{D}_\Psi$ and initial abstract probability space $(\mathcal{D}_{\Psi \cup \Phi}, \Sigma_{\Psi \cup \Phi}, \nu)$. In the process of constructing the initial probability space, we automatically identify *sub-queries* on the original probabilistic program, which are used to provide the values of the parameters and which are the source of decomposition.

First we give a criterion on abstractions that is sufficient to ensure distributional soundness. Crucially, the criterion is solely a relationship between concrete and abstract states, so it avoids directly reasoning about distributions.

Definition 5.12 (Tight abstraction). *Let* $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ *and,* $[\![\mathcal{A}]\!] : \mathcal{D}_A \to \mathcal{D}_{A'}$ *be measurable functions, where* $\mathcal{D}_A$ *and* $\mathcal{D}_{A'}$ *are predicate domains. Then we say* $[\![\mathcal{A}]\!]$ *is a* tight abstraction of $[\![\mathcal{C}]\!]$ *if for any* $c \in \Omega$, *we have that:*

$$\left[[\![\mathcal{C}]\!] (c) \right]_{\Psi'} = [\![\mathcal{A}]\!] \left([c]_\Psi \right). \tag{5.1}$$

Theorem 5.1 (Tightness implies soundness). *Let* $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ *and* $[\![\mathcal{A}]\!] : \Psi \to \Psi'$ *be measurable functions. Then, if* $[\![\mathcal{A}]\!]$ *is a tight abstraction of* $[\![\mathcal{C}]\!]$, *then* $[\![\mathcal{A}]\!]$ *is a distributionally sound abstraction of* $[\![\mathcal{C}]\!]$.

Proof of Theorem 5.1. Let $\mu : \Sigma \to [0,1]$ be an initial probability measure. The proof will follow by deriving a probability measure on the abstract domain $\nu' : \Sigma_{\mathcal{D}_A} \to [0,1]$ by following both paths in the commutative diagram from Definition 5.10, and showing that the result is the same for both paths.

Following the concrete path, we compute $\mu' : \Sigma' \to [0,1]$, which is the push-forward $\mu'(c') = \mu([\![\mathcal{C}]\!]^{-1}(c'))$. Then, abstracting this measure, we have that

$$\nu' = \alpha'(\mu') = a' \mapsto \mu'([a']^{-1})$$
$$= a' \mapsto \mu([\![\mathcal{C}]\!]^{-1}([a']^{-1})). \tag{5.2}$$

Note that $[a']^{-1}$ is an element of the σ-algebra and therefore the inverse $[\![\mathcal{C}]\!]^{-1}([a']^{-1})$ is well defined.

Next, following the abstract path, we first compute $\nu : \Sigma_{\mathcal{D}_A} \to [0,1]$, which is $\nu =$

$\alpha(\mu) = a \mapsto \mu([a]^{-1})$. Then, we compute ν' using the push-forward of $[\![\mathcal{A}]\!]$:

$$\nu' = a' \mapsto \nu((([\![\mathcal{A}]\!])^{-1}(a')) = a' \mapsto \alpha(\mu)((([\![\mathcal{A}]\!])^{-1}(a'))$$
$$= a' \mapsto \mu([(([\![\mathcal{A}]\!])^{-1}(a')]^{-1}). \qquad (5.3)$$

To prove these ν' measures equivalent, it suffices to show that $[\![\mathcal{C}]\!]^{-1}([a']^{-1}) = [(([\![\mathcal{A}]\!])^{-1}(a')]^{-1}$. This follows from Definition 5.12 by taking the inverse of both sides. $\qquad \square$

   With the guarantee that tight abstractions are sound, we now seek to generate a tight abstraction. Unfortunately, it is not always possible to generate a tight abstraction for an arbitrary choice of predicates. The following example demonstrates this, and also shows how we can find additional predicates called *completions* which, when added to the domain of the abstraction, allow us to generate tight abstractions.

Example 5.2: Completing an abstract domain

Consider the program $[\![\mathcal{C}]\!](x) = x + 1$. A tight abstraction for the predicate domain over the predicate $\{x \text{ is even}\}$ is:

$$[\![\mathcal{A}]\!] = \{(\{x \text{ is even}\}, \neg\{x \text{ is even}\}), (\neg\{x \text{ is even}\}, \{x \text{ is even}\})\}$$

Note here that we are describing the function $[\![\mathcal{A}]\!]$ as a set of pairs, where the first element denotes the domain and the second element denotes its corresponding output. On the other hand, no tight abstraction exists for the predicate domain over the predicate $\Psi = \{x < 0\}$: it is not possible to choose an element of $\mathcal{D}_{\mathcal{A}}$ for $[\![\mathcal{A}]\!](\{x < 0\})$ that satisfies condition (1) in Definition 5.12. However, we observe that if we add the predicate $\{x < -1\}$ to the domain (but *not* to the range) of $[\![\mathcal{A}]\!]$, then we *can* build a tight abstraction of $[\![\mathcal{C}]\!]$:

$$\begin{aligned}[\![\mathcal{A}]\!] = \{&(\{x < -1\} \wedge \{x < 0\}, \{x < 0\}), \\ &(\neg\{x < -1\} \wedge \{x < 0\}, \neg\{x < 0\}), \\ &(\neg\{x < 0\}, \neg\{x < 0\})\}\end{aligned}$$

We call $\{x < -1\}$ a *completion* predicate.

Completing the domain. Example 5.2 showed that adding completion predicates Φ to Ψ enables the creation of a tight abstraction from $\mathcal{D}_{\Psi \cup \Phi}$ to $\mathcal{D}_{\mathcal{A}}$. In general we say that Φ *completes* Ψ with respect to Ψ' and $[\![\mathcal{C}]\!]$ if there exists a tight abstraction $[\![\mathcal{A}]\!] : \mathcal{D}_{\Psi \cup \Phi} \to \mathcal{D}_{\mathcal{A}'}$. We call $\Psi \cup \Phi$ the completed set of predicates and $\mathcal{D}_{\Psi \cup \Phi}$ the completed predicate domain. Algorithm 3 automatically completes a set of predicates Ψ with respect to Ψ' and $[\![\mathcal{C}]\!]$ and generates a corresponding tight abstraction and initial probability space.[2] The algorithm

---

[2]We describe Algorithm 3 as directly producing a measurable function, but the implementation adapts

Algorithm 3: Domain completion

Data: Probability space (Ω, Σ, μ), measurable function $[\![\mathcal{C}]\!]$, input predicates Ψ and output predicates Ψ'

Result: A tight abstraction and a distributionally sound probability space

1 $[\![\mathcal{A}]\!] \leftarrow [\,]$;   `// New tight abstract function`

2 $\nu \leftarrow [\,]$;   `// New probability measure`

3 $\Phi \leftarrow \big\{ \mathbf{WP}([\![\mathcal{C}]\!], a') \mid a' \in \mathcal{D}_{\Psi'} \big\}$;

4 **for** $a \in \mathcal{D}_{\Psi \cup \Phi}$ **do**

5      $c' \leftarrow [\![\mathcal{C}]\!](c)$ for any $c \in [a]^{-1}$;

6   Append $(a, [c']_{\Psi'})$ to $[\![\mathcal{A}]\!]$;

7   Append $(a, \mathrm{Pr}_\mu(a))$ to ν;

8 **end**

9 **return** $([\![\mathcal{A}]\!], (\mathcal{D}_{\Psi \cup \Phi}, \Sigma_{\mathcal{D}_{\Psi \cup \Phi}}, \nu))$;

relies on the standard notion of the weakest precondition (see Definition 5.8). We formally state the correctness of Algorithm 3:

Theorem 5.2 (Domain completion). *Let $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ be a measurable function and Ψ and Ψ' be sets of predicates, and let (Ω, Σ, μ) be an initial concrete probability space. Then Algorithm 3 produces: (1) a tight abstraction $[\![\mathcal{A}]\!] : \mathcal{D}_{\Psi \cup \Phi} \to \mathcal{D}_{\Psi'}$ of $[\![\mathcal{C}]\!]$ over a completed predicate domain $\mathcal{D}_{\Psi \cup \Phi}$; (2) an initial probability space $(\mathcal{D}_{\Psi \cup \Phi}, \Sigma_{\mathcal{D}_{\Psi \cup \Phi}}, \nu)$, where ν is the push-forward of μ through $[\cdot]_{\Psi \cup \Phi}$.*

Proof of Theorem 5.2. We must show that (1) the generated measurable function $[\![\mathcal{A}]\!]$ is a tight abstraction of $[\![\mathcal{C}]\!]$, and (2) that the resulting probability space $(\Omega_{\Psi \cup \Phi}, \Sigma_{\Psi \cup \Phi}, \nu)$ is correctly pushed forward through $[\cdot]_{\Psi \cup \Phi}$. The second point clearly is true, since the loop iterates over each element of $\mathcal{D}_{\Psi \cup \Phi}$ and updates ν accordingly, so we focus on the first point.

It is clear that $[\![\mathcal{A}]\!]$ is a well-defined function, since each element of the domain is assigned to some element of the co-domain in the loop. Then, we must show that the resulting function is tight, i.e. that for any $c \in \Omega$, it is the case that $[[\![\mathcal{C}]\!](c)]_{\Psi'} = [\![\mathcal{A}]\!]([c]_{\Psi \cup \Phi})$.

For each $a \in \mathcal{D}_{\Psi \cup \Phi}$, there is some $a_\phi \in \mathcal{D}_\Phi$ such that a implies a_ϕ. For any concrete state c such that $[c]_{\Psi \cup \Phi}$ implies a_ϕ, by the definition of the weakest precondition, $[[\![\mathcal{C}]\!](c)]_{\Psi'} = a'$ for some $a' \in \mathcal{D}_{\mathcal{A}'}$. Then, we let $[\![\mathcal{A}]\!]([c]_{\Psi \cup \Phi}) = a'$, so by definition $[\![\mathcal{A}]\!]$ is a tight measurable function. $\square$

---

standard predicate abstraction techniques [Ball et al., 2001, Holtzen et al., 2017] to generate an abstract probabilistic program.

Discussion We provide some discussion of Algorithm 3. Then, we describe optimizations that can improve the performance of the algorithm in practice. Algorithm 3 proceeds as follows. First, on Line 3 the set of predicates Φ is generated using the weakest precondition. By construction, there exists a tight measurable function from $\mathcal{D}_\Phi$ to $\mathcal{D}_{\Psi'}$. This fact relies on the definition of the weakest precondition. Formally, for each $\phi \in \mathcal{D}_\Phi$, there exists some $a' \in \mathcal{D}_{\Psi'}$ such that for any $c \in [\phi]^{-1}$, $[\![\mathcal{C}]\!](c)]_{\Psi'} = a'$.

Now, we must construct a tight measurable function on the domain $\mathcal{D}_{\Psi \cup \Phi}$ and compute the appropriate sub-queries, both of which are done in the loop beginning on Line 4. For each $a \in \mathcal{D}_{\Phi \cup \Psi}$, there is some $\phi \in \mathcal{D}_\Phi$ such that a implies ϕ, which guarantees that we can give a deterministic function $[\![\mathcal{A}]\!]$ for a following the arguments in the previous paragraph.


Example 5.3: Running Algorithm 3.

Consider the program $p = x \leftarrow x+1$ and the predicate $\Psi = \{x < 0\}$. We wish to evaluate Algorithm 3 with input probability space $(\mathcal{D}_\Psi, \Sigma, \mu)$ with initial and final predicate domains over Ψ, i.e. $\mathcal{D}_\Psi = \{\{x < 0\}, \neg\{x < 0\}\}$, as in Example 5.2. Then, $\Phi = \{x < -1\}$, and $\mathcal{D}_{\Psi \cup \Phi} = \{\{x < 0\} \wedge \{x < -1\}, \neg\{x < 0\} \wedge \{x < -1\}, \dots\}$. Consider the case $a = \{x < 0\} \wedge \{x < -1\}$. The algorithm will select a $c \in [a]^{-1}$; for example -2. Then, $[\![\mathcal{A}]\!](a)$ will be assigned to $[-2+1]_{\Psi'} = [-1]_{\Psi'} = \{x < 0\}$.

As described, this algorithm produces 2^n completion predicates, where n is the size of Ψ'. However, in practice various logical optimizations are used to reduce the number of completion predicates and sub-queries, such as exploiting logical implication between predicates, pruning unsatisfiable configurations of predicates, and exploiting independence between non-overlapping predicates [Ball et al., 2001, Holtzen et al., 2017].

5.4.1 Selecting Predicates

Thus far we have assumed the collection of predicates from which the abstraction is built is provided *a priori*. In general, the problem of finding a useful set of predicates – i.e., one that fruitfully decomposes the program – is hard. Nonetheless, even simple heuristics may work well for many programs. For example, one approach is to include each Boolean expression in the program as a predicate; this has the useful property of capturing the behavior of `if` and `observe` statements, constructs that many existing probabilistic programming systems struggle with due to their non-differentiability [Carpenter et al., 2016].

More generally, much of the insight from decades of research on constructing non-deterministic predicate abstractions can be applied here, and generalizing these techniques to the setting of probabilistic predicate abstractions is a direction for future research. For instance, a common technique for predicate generation is *counterexample-*
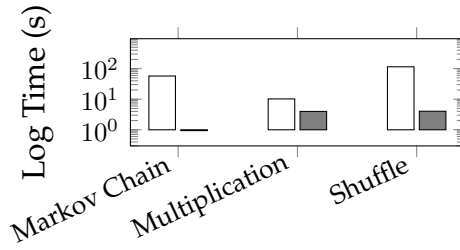
(a) Exact inference results on the Psi system (see Chapter 5.5.1). ☐ represent un-abstracted models, and ▇ represent abstracted models.
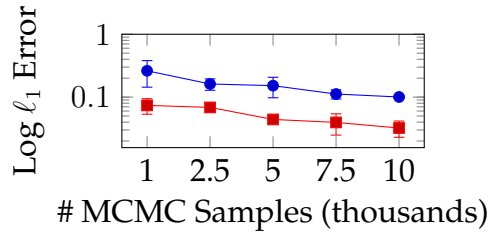
(b) Convergence for approximate inference, lower is better. The red boxes show the log error for the decomposed MCMC sampler; the blue circles show log error for the non-decomposed MCMC sampler. The error bars show the upper and lower quartile, points show the mean over 20 runs, and error is the ℓ_1-norm between the true value and the approximated value. See Chapter 5.5.2.

Figure 5.3: Experimental results.

guided refinement, which iteratively generates new predicates on demand, until the abstraction is rich enough to either prove or disprove a query of interest [Clarke et al., 2003].

5.5 Decomposition via Abstraction

The theory and algorithm presented in the previous sections can be used to simplify inference via a process called *decomposition via abstraction*. The process is as follows. First, we are given a program $\mathcal{C}$ over which we wish to perform some inference query $\Pr(q \mid e)$. Then we choose, or are provided with, a set of predicates Ψ, which must include the necessary predicates for describing q and e. Next, we utilize Algorithm 3, which (1) generates a tight abstract probabilistic program $\mathcal{A}$, and (2) parameterizes the abstraction by performing sub-queries to the original probabilistic program. To answer queries, we perform inference on the abstraction $\mathcal{A}$. This is sound due to Proposition 5.1.

Figure 5.3a shows the computational benefits of decomposition via abstraction on exact and approximate inference tasks, which are elaborated on in the following subsections.

5.5.1 Exact Inference

This section asks the question: does decomposition improve the performance of exact inference? Many existing techniques for exact probabilistic program inference utilize *path-based* decompositions [Gehr et al., 2016, Chistikov et al., 2015, Albarghouthi et al., 2017, Sankaranarayanan et al., 2013]. Specifically, they operate by integrating the probability mass along each path of a probabilistic program. This section shows how the decomposition technique serves to complement path-based decompositions in the following way. For each probabilistic program, we used Psi [Gehr et al., 2016] to compute an inference query on (1) the concrete program, and (2) the abstract program and the sub-queries.[3] The total time for each query task is reported in Figure 5.3a. There are three experiments that each highlight an important property of probabilistic programs that may be exploited via abstraction. In each case, it is shown experimentally that the total time spent parameterizing and performing inference on the abstraction is less than the time spent performing inference on the original concrete program. Interestingly, we will see that different kinds of abstractions enable existing inference algorithms to exploit different kinds of structure, including factorization and symmetry.

Multiplication This experiment uses a complete version of the example described in Chapter 5.1 and illustrates how abstraction via decomposition can automatically perform *context-sensitive decomposition*. Specifically, computing the sub-queries during the abstraction procedure can implicitly decompose a complex probability distribution, even when a factor-graph representation is fully connected. Given the appropriate predicates, Algorithm 3 automatically constructs an abstraction and exploits these independence properties when performing sub-queries.

Markov Chain Decomposition via abstraction can exploit conditional independences that are typically unexploited by existing probabilistic programming inference algorithms. One particular example is a *Markov chain*, a model which has exponentially many paths yet retains linear-time exact inference Koller and Friedman [2009b]:

$$n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k$$
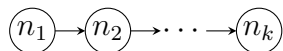
In order to compute $\Pr(n_1 \mid n_k)$, path-based inference techniques must integrate $\mathcal{O}(2^k)$ paths, which quickly becomes infeasible as the Markov chain grows. However, there is a natural choice of predicates for decomposing such programs: simply including the guard of each `if`-statement. By applying an optimized Algorithm 3 recursively on each `if`-statement in turn, we recover a linear-time inference algorithm for Markov-Chain-like probabilistic programs, and more generally a join-tree-like inference algorithm for

---

[3]Psi build `5334524fe` was used for these experiments.

Bayesian-network-like programs. For demonstration, consider performing inference on the following Boolean-valued Markov chain, although strategy generalizes to more complex networks:

```
1  n₁ ←flip(θₙ₁);
2  n₂ ←if n₁ then flip(θₙ₂|ₙ₁) else flip(θₙ₂|n̄₁)
3  ...
4  nₖ ←if nₖ₋₁ then flip(θₙₖ|nₖ₋₁) else flip(θₙₖ|n̄ₖ₋₁);
```

First we generate an abstraction using the predicates $\{n_1\}$ and $\{n_k\}$. The algorithm generates (1) an abstract program which describes the relationship between these two predicates, and (2) sub-queries necessary for computing the parameters in (1). The generated abstract program is:

```
1  {n₁} ←flip(θₙ₁);
2  {nₖ} ←if {n₁} then flip(θₙₖ|ₙ₁) else flip(θₙₖ|n̄₁);
```

Next we must evaluate the sub-queries. The parameter θ_{n_1} is from the original program; it is the prior on the first variable in the chain. The parameters $\theta_{n_k|n_1}$ and $\theta_{n_k|\overline{n_1}}$ are completion predicates, which must both be evaluated on the concrete program. To evaluate these sub-queries, we can utilize abstraction recursively, this time using the predicates $\{n_1\}$, $\{n_k\}$, and $\{n_{k-1}\}$. The intermediate abstract program is:

```
1  {n₁} ←flip(θₙ₁);
2  {nₖ₋₁} ←if {n₁} then flip(θₙₖ₋₁|ₙ₁) else flip(θₙₖ₋₁|n̄₁);
3  {nₖ} ←if {nₖ₋₁} then flip(θₙₖ|nₖ₋₁) else flip(θₙₖ|n̄ₖ₋₁);
```

The sub-query on Line 3 implicitly exploits the conditional independence between n_1 and n_{k-1} given n_k. In this case, $\theta_{n_k|n_{k-1},n_1} = \theta_{n_k|n_{k-1},\overline{n_1}}$, so Line 3 performs only one of these equivalent queries. This is an optimization that Algorithm 1 would not do automatically, as it would naively consider all possible joint assignments to predicates on Line 3, and would thus evaluate both of these equivalent sub-queries. In practice, identifying duplicate sub-queries will be an important optimization. In this case probabilistic program slicing would discover this equivalence [Hur et al., 2014]. The process of querying the concrete program recursively utilizing abstraction may be repeated inductively for each sub-program. Ultimately, n sub-programs will be generated, each with 2 paths, for a total of $2n$ sub-queries.

Note that this is quite similar to the way that Dice exploits factorization. However, Dice does not require the user to select predicates.

Shuffle   Recall from Chapter 4 that many intractable models can be rendered tractable by exploiting the underlying symmetry of random variables. This example illustrates the potential connections between probabilistic program abstraction and lifted inference. Consider the following probabilistic program, which shuffles a small deck of cards:

```
1  deck←[1,2,3,4,5,6];
2  for idx in [0..5) {
3      j←uniformInt(idx, 6);
4      swap(deck[j], deck[i]);
5  }
```

We wish to compute $\Pr(\text{deck[0]} = 1)$, i.e. the probability that the top card of the deck is still 1 after shuffling. There is a key symmetry that reduces the state space of this problem: it is not necessary to model the distribution on all the cards. For answering this query, it is sufficient to treat the cards as either "1" or "not 1", since all cards that are not 1 are exchangeable. Specifically, we can create an abstract program by changing the first line of the original program:

```
1  deck←[{1},¬{1},¬{1},¬{1},¬{1},¬{1}];
```

Before this abstraction, there were $6!$ arrangements of cards; after this abstraction, there are only 6, drastically reducing the cost of inference.

Note that while this abstract program is distributionally sound, it is not a predicate abstraction and thus not generated by Algorithm 3. Specifically, this abstraction is constructed by surgically abstracting portions of the concrete program, rather than by building an abstraction from the ground up with predicates. Automating such abstractions is an interesting direction for future work.

5.5.2 Approximate Inference

Many existing probabilistic programming systems rely on approximate inference methods such as Markov-Chain Monte Carlo or variational approximations to perform inference [Carpenter et al., 2016, Wood et al., 2014, Goodman et al., 2008, Tran et al., 2017]. These techniques typically make assumptions about the underlying program structure in order to perform well: for example, Hamiltonian Monte-Carlo will assume that the underlying distribution is continuous, and variational inference assumes that the distribution can be well-captured by the proposal family. In general, we may utilize decomposition via abstraction to apply approximate inference methods to evaluate the sub-queries for which they are best suited.

Consider the following probabilistic program. We wish to infer the probability that x is less than a constant k given three noisy observations about x (as notation, $\mathcal{N}(\mu, \sigma)$ is a normal distribution with mean μ and variance σ):

```
1  x←𝒩(μ, σ);
2  y₁ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
3  y₂ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
4  y₃ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
```

```
5   observe($y_1$<c ∧ $y_2$<c ∧ $y_3$ ≥c);
6   return x<k;
```

Approximate inference techniques such as Markov-Chain Monte-Carlo (MCMC) or direct sampling struggle with this example: the distribution is multi-modal, non-differentiable, and the a-priori probability of the observations being satisfied is low. This is evidenced by the blue circle performance line in Figure 5.3b, which shows the performance of MCMC on the un-abstracted model using WebPPL [Goodman and Stuhlmüller, 2014] with a fixed number of samples.

The red performance line in Figure 5.3b shows the convergence of an abstracted model generated by Algorithm 3 with respect to the predicates $\{x < k\}, \{y_i < c\}$. This abstraction allows us to perform a hybrid inference procedure. Each sub-query (i.e., computing $\Pr(x < k)$) is differentiable and uni-modal, and can be easily evaluated using MCMC; in this experiment, we evaluated each sub-query using a portion of a fixed total budget of samples. Because the abstraction itself is a discrete `Dice` program, the final query on the abstract program may be performed using enumeration, which can handle discontinuities and low-probability evidence.

5.6   Related Work

Graph compilation. There exists a family of inference tools that compile probabilistic programs to structured probabilistic models [Pfeffer, 2009, McCallum et al., 2009, Minka et al., 2014]. Often, these tools are too coarse; the techniques presented here can exploit more decompositions than a graph captures by exploiting nuanced program structure.

Program analysis. Some approximate inference tools integrate static information from the program: for instance, Chaganty et al. [2013] and Nori et al. [2014] utilize symbolic execution or weakest precondition computations to draw samples more efficiently from a probabilistic program. However, they do not exploit statistical decompositions such as conditional independence, and they perform their analyses over the entire program, rather than performing sub-queries. Probabilistic abstract interpretation has been studied in prior work, but in all cases the soundness relationship is weaker than distributional soundness [Cousot and Monerau, 2012, Monniaux, 2000, 2001].

5.7   Conclusion

This chapter addresses the question: what is a useful abstraction for a probabilistic program? It showed that such a useful abstraction must be distributionally sound, and described the theory and practice for constructing such abstractions. Then, it empirically validated this approach on approximate and exact inference tasks.

CHAPTER 6

Conclusion

This chapter will conclude the thesis with discussion of the contributions, future work and open problems, and finally end with a broad future outlook.

6.1 Contributions & Outlook

There is no one-size-fits-all solution to probabilistic program inference. Each new approach to inference reveals avenues for applying probabilistic programs in new places or designing languages with richer and more expressive features. Long term, there remains many deep foundational questions, and this chapter highlights a few of them that will require sustained work.

6.1.1 Challenges and Opportunities in Probabilistic Programming Language Design

Chapter 3 showed how to apply probabilistic programs effectively in discrete domains that were previously out of reach. It showed that, by using a strategy of compiling programs to tractable representations, it is possible to scale to large language models, verify properties of large computer networks, compete with state-of-the-art Bayesian network solvers, and compete with probabilistic model checkers [Holtzen et al., 2021]. However, `Dice` is in its infancy: there are many substantive improvements that are necessary to make it a standard tool in a programmer's toolbox. I divide these improvements and future work goals into 3 broad categories: *usability*, *expressivity*, and *applications*. I see these goals as challenges for *all* current probabilistic programming languages, but I highlight specific aspects of them as they relate to `Dice`.

6.1.1.1 Usability

Currently writing probabilistic programs – in any probabilistic programming languages – is a very delicate task, and I would argue is still quite difficult for the average user. The exact way in which the program is written can have drastic impacts on performance of inference, and the inference algorithm itself can often be inscrutable from the perspective of the user. Hence, there is a need for *usable probabilistic program inference*: probabilistic programming systems that assist the user in designing programs for which inference

scales.

A very similar usability challenge exists in traditional software design, and I advocate that we should draw inspiration from techniques from this area. In particular, the areas of *compiler design* and *software debugging* are two compelling areas that have many insights for probabilistic programming language designers [Aho et al., 1986].

Probabilistic programming languages are in need dire need of *compiler optimizations*: techniques for helping users write fast programs without needing deep knowledge of the internal workings of the inference algorithm. For instance, `Dice`'s inference algorithm is currently quite naive and eager in how it builds large BDDs: it will happily compile both branches of an `if`-statement even if the guard makes one of those branches impossible. A smart optimization here is *branch elimination*: if you can prove that a branch of an `if`-statement is never exercised, then that BDD should never be compiled. This flavor of *probabilistic program optimization* – and many others – will be critical for designing scalable turn-key inference that does not depend on exactly how the user writes the program.

Aside from the scalability issue, probabilistic programs also suffer from a *correctness issue*: currently there is almost no language support for programmers to track down and isolate errors in probabilistic programs. Ideas like *probabilistic program debugging* will be critical long-term for designing usable systems that work in practice [Nandi et al., 2017].

6.1.1.2 Expressivity

One of the key arguments of this thesis is that expressivity is tightly coupled with scalability: having a language with many features is not particularly compelling if inference for all but the most simple programs is hopelessly intractable. I advocate for an approach to language design that is in harmony with scalability of inference: when adding a feature, be cognizant of how it impacts the underlying inference algorithm, and ideally characterize how it affects inference using complexity-theoretic arguments.

Claim 1

Probabilistic modeling languages should be designed in concert with their inference algorithms.

What does this mean in the context of `Dice`? It asks the question: *how many language features can we add while still remaining compatible with* `Dice`'s *BDD compilation strategy*? How far can we push BDDs? For instance, handling continuous random variables is an extremely important concept for representing distributions, but they seem incompatible with BDDs. However, consider the special case of a beta-prior on a Bernoulli variable: it is well-known that this is a special case of *Bayesian conjugacy*, and there exist closed-form solutions for the posterior of such instances. Can `Dice` exploit this to handle some limited forms of continuity while maintaining its BDD backend?

Moreover, as we observed in Chapter 4, there are plenty of *discrete* distributions that Dice cannot handle, but for which there exist efficient *specialized* inference strategies. *These are opportunities*: how can we extend Dice to handle these kinds of distributions? For instance, there are many specialized inference algorithms for handling *distributions on permutations*, but Dice would struggle with these kinds of distributions since there is little independence [Huang et al., 2009]. Can Dice be extended with these ideas to handle these new kinds of distributions?

6.1.1.3 Applications

There is a nearly infinite space of probabilistic programming language designs that trade off all possible combinations of language features and inference algorithm possibilities. How can we choose a particular point in this vast design space? I advocate that *language design be driven by applications*:

Claim 2

Applications should drive the development of probabilistic modeling language enhancements and features.

Dice was initially motivated by applications in language modeling, network verification, and probabilistic graphical models. After it was initially developed, Dice found new application in probabilistic model checking [Holtzen et al., 2021]. Each of these applications motivated specific features and design decisions in Dice: for instance none of these problems required continuous random variables or unbounded loops.

On the horizon, I see future applications of Dice in surprising areas like classical simulation of quantum algorithms [Huang et al., 2021], and in less surprising but still challenging areas such as linguistics and bioinformatics. To reach this goal, we will need to add new features to Dice – for instance, forms of loops or the ability to represent complex amplitudes instead of probabilities. Each of these features should be carefully considered in the context of the kinds of inference algorithms that they would allow us to employ.

6.1.2 Symmetry and Lifted Inference

Chapter 4 gave a new foundation for exploiting symmetry in probabilistic graphical models, but there is still much work to do before this foundation can be directly applied inside of probabilistic program inference algorithms like Dice. I see two key challenges in bridging this gap: (1) designing a tractable back-end that exploits symmetry and (2) integrating this back-end with Dice, a property called *compositionality*.

6.1.2.1 New Tractable Back-ends

One of the key insights behind knowledge compilation – the philosophy that drives `Dice` inference – is the relationship between *fast inference* and *tractable representations*: the key idea that, if inference is fast, we can often capture this fast computation as a compact circuit that has certain properties [Darwiche and Marquis, 2002]. This motivates the following claim:

> **Claim 3**
>
> If it is possible to perform fast inference, then we should be able to identify a tractable probabilistic model that isolates the computation.

Given the foundation laid in Chapter 4, a future goal driven by the above claim is *identifying a tractable representation that captures the symmetry exploited by lifted inference*. Van den Broeck [2013] studied this very question in the context of first-order sentences, but the challenge remains open for other representations like propositional factor graph and probabilistic programs.

Symmetry is just the beginning. There are countless other situations in which tractable probabilistic reasoning is possible: for instance, determinantal point processes (DPPs) are a well-known TPM that has been studied in the context of subset selection for machine learning [Kulesza and Taskar, 2012]. Zhang et al. [2020] and Zhang et al. [2021] study the problem of constructing a probabilistic circuit that captures the computation of a DPP: one day this foundation may well yield a backend TPM that exploits the structure implied by a DPP.

6.1.2.2 Compositionality of TPMs

Probabilistic programs are compositional by design: big programs are made up of smaller programs. This compositionality must be reflected in the inference algorithm if there is to be any hope of scaling to large probabilistic programs. Hence, there is a dire need of *compositional inference algorithms* that allow inference results for smaller sub-programs to be combined to give inference results about the entire program. For instance, in `Dice`, each sub-program is associated with a BDD, which is itself a compositional object that can be combined with other BDDs to give an inference result for the whole program.

As more TPMs are developed, the question of composing them with other TPMs becomes increasingly pressing. This field is in its infancy and there are many important questions about when it is possible to combine two different kinds of TPMs that have yet to be properly posed. However, I will argue that probabilistic programming languages provide the most compelling motivation for this study: compositionality is the essence of programming, so compositionality of TPMs will be the essence of probabilistic program inference.

6.1.3 Abstraction and Distributional Soundness

The previous section posed several challenges in designing compositional tractable probabilistic models motivated by the goal of combining inference results for sub-programs. One avenue for this process of program decomposition was given in Chapter 5: breaking the program up structurally into sub-components given by the program's behavior on a set of predicates.

There are a number of important avenues for extending this work so that it becomes an integral part of a probabilistic programming work-flow. The primary challenge is *automating the abstraction process*. Currently the method in Chapter 5 requires the user to provide a set of predicates, but this is quite unwieldy: methods from verification like *counter-example guided abstraction refinement* [Clarke et al., 2000, McIver et al., 2005] could give avenues for automating predicate abstraction construction.

6.2 Discussion

The idea of a probabilistic program is widely regarded to have originated in Kozen [1979]. In this context, the goal was to verify and give a formal semantics to randomized algorithms. However, since then, the scope of objectives for probabilistic programs has vastly widened to include not only verifying randomized algorithms, but also data analysis and modeling probabilistic agents and systems. I would like to conclude this thesis with a call to action that touches on broader ideas. In particular I would like to close with some high-level calls to the artificial intelligence and programming languages communities, bringing some attention to the shared insights that will be necessary from each field in order to achieve progress. To the PL community:

- Probabilistic program semantics are important, but they are not the only problem. There are many interesting classes of languages that do not have particularly interesting semantics but have very interesting inference algorithms.

- Think beyond sampling for probabilistic program inference: compositional exact inference has a very "PL" flavor and there are many opportunities still for applying programming language ideas in this direction.

- Formalization of probabilistic program inference is in its infancy: there is much work to do here still.

To the AI community:

- Think about compositionality: how can we combine different kinds of modeling families in natural ways? Different kinds of inference algorithms?

- Think about formalization and verification: how can we prove inference algorithms correct? Programs should have semantics.

Ultimately I argue that ideas from both of these communities are necessary for designing strategies for scaling inference by exploiting program structure.

APPENDIX A

Proofs

A.1 Chapter 3

A.1.1 Important Lemmas

Lemma A.1 (Independent Conjunction). *Let α and β be Boolean sentences which share no variables; we call such sentences* independent. *Then, for any weight function w, $\mathtt{WMC}(\alpha \wedge \beta, w) = \mathtt{WMC}(\alpha, w) \times \mathtt{WMC}(\beta, w)$.*

Proof. The proof relies on the fact that, if two sentences α and β share no variables, then any model ω of $\alpha \wedge \beta$ can be split into two components, ω_α and ω_β, such that $\omega = \omega_\alpha \wedge \omega_\beta$, $\omega_\alpha \Rightarrow \alpha$, and $\omega_\beta \Rightarrow \beta$, and ω_α and ω_β share no variables. Then: $\mathtt{WMC}(\alpha \wedge \beta, w) = \sum_{\omega \in \mathtt{Mods}(\alpha \wedge \beta)} \prod_{l \in \omega} w(l) = \left[\sum_{\omega_\alpha \in \mathtt{Mods}(\alpha)} \prod_{a \in \omega_\alpha} w(a) \right] \times \left[\sum_{\omega_\beta \in \mathtt{Mods}(\beta)} \prod_{b \in \omega_\beta} w(b) \right] = \mathtt{WMC}(\alpha, w) \times \mathtt{WMC}(\beta, w)$. $\square$

Proposition A.1 (Inclusion-Exclusion). *For any two formulas φ_1 and φ_2 and weight function w, $\mathtt{WMC}(\varphi_1 \vee \varphi_2, w) = \mathtt{WMC}(\varphi_1, w) + \mathtt{WMC}(\varphi_2, w) - \mathtt{WMC}(\varphi_1 \wedge \varphi_2, w)$. Note the important* mutual exclusion *case when $\varphi_1 \wedge \varphi_2 = \mathtt{F}$.*

A.1.2 Correctness of Expression Compilation

Lemma A.2 (Value Correctness). *For any values v and v' of type τ, $[\![v]\!]\,(v') = \mathtt{WMC}(v \overset{\tau}{\Leftrightarrow} v', \emptyset)$.*

Proof. By induction on τ:

- $\tau = \mathbf{Bool}$. Then case analysis:

- $[\![\mathtt{T}]\!](\mathtt{T}) = 1 = \mathtt{WMC}(\mathtt{T} \Leftrightarrow \mathtt{T}, \emptyset)$
- $[\![\mathtt{T}]\!](\mathtt{F}) = 0 = \mathtt{WMC}(\mathtt{T} \Leftrightarrow \mathtt{F}, \emptyset)$
- $[\![\mathtt{F}]\!](\mathtt{F}) = 1 = \mathtt{WMC}(\mathtt{F} \Leftrightarrow \mathtt{F}, \emptyset)$
- $[\![\mathtt{F}]\!](\mathtt{T}) = 0 = \mathtt{WMC}(\mathtt{F} \Leftrightarrow \mathtt{T}, \emptyset)$

- Inductive step: $\tau = \tau_1 \times \tau_2$. Then,

$$
\begin{aligned}
[\![(v_1, v_2)]\!]((v_1', v_2')) &= [\![v_1]\!](v_1') \times [\![v_2]\!](v_2') \\
&= \mathtt{WMC}(v_1 \overset{\tau_1}{\Longleftrightarrow} v_1', \emptyset) \times \mathtt{WMC}(v_2 \overset{\tau_1}{\Longleftrightarrow} v_2', \emptyset) \qquad \text{Induction Hyp.} \\
&= \mathtt{WMC}(v_1 \overset{\tau_1}{\Longleftrightarrow} v_1' \wedge v_2 \overset{\tau_1}{\Longleftrightarrow} v_2', \emptyset) \qquad \text{Independent Conj.} \\
&= \mathtt{WMC}((v_1, v_2) \overset{\tau_1 \times \tau_2}{\Longleftrightarrow} (v_1', v_2'), \emptyset).
\end{aligned}
$$

$\square$

Lemma A.3 (Typed Substitution). *For any values $v, v_x : \tau$, it holds that $(v \overset{\tau}{\Leftrightarrow} v_x) = (F_\tau(x) \overset{\tau}{\Leftrightarrow} v)[\mathbf{x} \overset{\tau}{\mapsto} v_x]$.*

Proof. By induction on τ:

- $\tau = \mathbf{Bool}$. Then, $(v \Leftrightarrow v_x) = (v \Leftrightarrow \mathbf{x})[\mathbf{x} \mapsto v_x] = (v \Leftrightarrow F_{\mathbf{Bool}}(x))[\mathbf{x} \mapsto v_x]$.
- $\tau = \tau_1 \times \tau_2$. Then, let $v = (v^l, v^r)$ and $v_x = (v_x^l, v_x^r)$. Then,

$$
\begin{aligned}
(v^l, v^r) \overset{\tau_1 \times \tau_2}{\Longleftrightarrow} (v_x^l, v_x^r) &= (v^l \overset{\tau_1}{\Leftrightarrow} v_x^l) \wedge (v^r \overset{\tau_2}{\Leftrightarrow} v_x^r) \\
&= (v^l \overset{\tau_1}{\Leftrightarrow} F_{\tau_1}(x_l))[\mathbf{x}_l \overset{\tau_1}{\mapsto} v_x^l] \wedge (v^r \overset{\tau_2}{\Leftrightarrow} F_{\tau_2}(x_r))[\mathbf{x}_r \overset{\tau_1}{\mapsto} v_x^r] \quad \text{Ind. Hyp.} \\
&= (v^l \overset{\tau_1}{\Leftrightarrow} F_{\tau_1}(x_l) \wedge (v^r \overset{\tau_2}{\Leftrightarrow} F_{\tau_2}(x_r)))[\mathbf{x}_l \overset{\tau_1}{\mapsto} v_x^l][\mathbf{x}_r \overset{\tau_1}{\mapsto} v_x^r] \\
&= ((v^l, v^r) \overset{\tau_1 \times \tau_2}{\Longleftrightarrow} F_{\tau_1 \times \tau_2}(x))[\mathbf{x} \overset{\tau_1 \times \tau_2}{\longmapsto} (v_x^l, v_x^r)].
\end{aligned}
$$

$\square$

Lemma A.4 (Typed Correctness Without Procedures). *Let* e *be a* Dice *expression without procedure calls. Let $\{x_i : \tau_i\} \vdash$ e $: \tau \rightsquigarrow (\dot\varphi, \gamma, w)$. Then for any values $\{v_i : \tau_i\}$ and $v : \tau$, we have that $[\![\mathtt{e}[x_i \mapsto v_i]]\!](v) = \mathtt{WMC}\left(((v \overset{\tau}{\Leftrightarrow} \varphi) \wedge \gamma)[\mathbf{x}_i \overset{\tau_i}{\mapsto} v_i], w\right)$.*

Proof. The proof is by structural induction on the syntax of Boolean Dice programs. First, we prove that the theorem holds for the non-inductive terms:

- e $=$ T and e $=$ F follow directly from Lemma A.2.

- $e = \text{flip } \theta$. Then, $\Gamma \vdash \text{flip } \theta : \textbf{Bool} \rightsquigarrow (\mathbf{f}, \texttt{T}, w)$ for a fresh $\mathbf{f}$. Then, $\texttt{WMC}(\mathbf{f} \wedge \texttt{T}, w) = \theta = [\![\text{flip } \theta]\!] (\texttt{T})$ and $\texttt{WMC}(\overline{f}, w) = 1 - \theta = [\![\text{flip } \theta]\!] (\texttt{F})$.

- $e = x$. Then, $\Gamma \vdash x : \tau \rightsquigarrow (\dot\varphi, \texttt{T}, \emptyset)$, and let $v_x : \tau$ be the value substituted for x.

$$\left[\!\!\left[x[x \xmapsto{\tau} v_x] \right]\!\!\right] (v) = [\![v_x]\!] (v)$$
$$= \texttt{WMC}((v_x \overset{\tau}{\Longleftrightarrow} v) \wedge \texttt{T}, \emptyset) \qquad\qquad \text{Lemma A.2}$$
$$= \texttt{WMC}\left(((F_\tau(x) \overset{\tau}{\Longleftrightarrow} v) \wedge \texttt{T})[\mathbf{x} \xmapsto{\tau} v_x], \emptyset \right) \qquad \text{Lemma A.3}$$

- $e = \text{fst } x$. Assume $\Gamma(x) = \tau_1 \times \tau_2$. Then, $\Gamma \vdash \text{fst } x : \tau_1 \rightsquigarrow (F_{\tau_1}(x_l), \texttt{T}, \emptyset)$. Let $v_x = (v_x^l, v_x^r) : \tau_1 \times \tau_2$ be the value substituted for x. Then,

$$\left[\!\!\left[\text{fst } x[x \xmapsto{\tau \times \tau'} v_x] \right]\!\!\right] (v) = [\![v_x^l]\!] (v)$$
$$= \texttt{WMC}((v_x^l \overset{\tau}{\Longleftrightarrow} v) \wedge \texttt{T}, \emptyset) \qquad\qquad \text{Lemma A.2}$$
$$= \texttt{WMC}\left(((F_\tau(x_l) \overset{\tau}{\Longleftrightarrow} v) \wedge \texttt{T})[\mathbf{x} \xmapsto{\tau \times \tau'} v_x], \emptyset \right) \qquad \text{Lemma A.3}$$

An analogous argument holds for $\text{snd } x$.

- $e = (x_1, x_2)$. Then, $\Gamma \vdash (x_1, x_2) : \tau_1 \times \tau_2 \rightsquigarrow ((F_{\tau_1}(x_1), F_{\tau_2}(x_2)), \texttt{T}, \emptyset)$. Let $v_1 : \tau_1$ and $v_2 : \tau_2$ be the value substituted for x_1 and x_2 respectively, and let $v = (v^l, v^r)$. Then,

$$\left[\!\!\left[(x_1, x_2)[x_1 \xmapsto{\tau_1} v_1, x_2 \xmapsto{\tau_2} v_2] \right]\!\!\right] ((v^l, v^r))$$
$$= [\![(v_1, v_2)]\!] (v^l, v^r)$$
$$= \texttt{WMC}\left((v_1 \overset{\tau_1}{\Longleftrightarrow} v^l) \wedge (v_2 \overset{\tau_2}{\Longleftrightarrow} v^r) \wedge \texttt{T}, \emptyset \right) \qquad\qquad \text{Lemma A.2}$$
$$= \texttt{WMC}\left(\left(F_{\tau_1}(x_1) \overset{\tau_1}{\Longleftrightarrow} v^l\right) \wedge \left(F_{\tau_2}(x_2) \overset{\tau_2}{\Longleftrightarrow} v^r\right) \wedge \texttt{T}[x_1 \xmapsto{\tau_1} v_1, x_2 \xmapsto{\tau_2} v_2], \emptyset \right) \quad \text{Lemma A.3}$$

Now for the inductive terms:

- $e = \text{let } e_1 \text{ in } e_2$. Assume $\Gamma \vdash e_1 : \tau_1 \rightsquigarrow (\dot\varphi_1, \gamma_1, w_1)$ and $\Gamma \cup \{x : \tau_1\} \vdash e_2 : \tau_2 \rightsquigarrow (\dot\varphi_2, \gamma_2, w_2)$. For notational simplicity, assume that the substitution $[x_i \xmapsto{\tau_i} v_i]$ has been applied to $\dot\varphi_1, \gamma_1, \dot\varphi_2, \gamma_2$, and that all weighted model counts are performed with the

weight $w_1 \cup w_2$. Then,

$$\llbracket(\texttt{let } x = \texttt{e}_1 \texttt{ in } \texttt{e}_2)[x_i \mapsto v_i]\rrbracket(\texttt{T})$$

$$= \sum_v \llbracket\texttt{e}_1[x_i \mapsto v_i]\rrbracket(v) \times \llbracket\texttt{e}_2[x_i \mapsto v_i, x \mapsto v]\rrbracket(\texttt{T})$$

$$= \sum_{v_x \in \tau_1} \texttt{WMC}\big((\dot\varphi_1 \overset{\tau_1}{\Longleftrightarrow} v_x) \wedge \gamma_1\big) \times \texttt{WMC}\big(((\dot\varphi_2 \overset{\tau_2}{\Longleftrightarrow} v) \wedge \gamma_2)[\mathbf{x} \overset{\tau_1}{\mapsto} v_x]\big) \qquad \text{Ind. Hyp.}$$

$$= \sum_{v_x \in \tau_1} \texttt{WMC}\Big((\dot\varphi_1 \overset{\tau_1}{\Longleftrightarrow} v_x) \wedge \gamma_1 \wedge \big((\dot\varphi_2 \overset{\tau_2}{\Longleftrightarrow} v) \wedge \gamma_2\big)[\mathbf{x} \overset{\tau_1}{\mapsto} v_x]\Big) \qquad \text{Indep. Conj.}$$

$$= \texttt{WMC}\Big(\bigvee_{v_x \in \tau_1} (\dot\varphi_1 \overset{\tau_1}{\Longleftrightarrow} v_x) \wedge \gamma_1 \wedge \big((\dot\varphi_2 \overset{\tau_2}{\Longleftrightarrow} v) \wedge \gamma_2\big)[\mathbf{x} \overset{\tau_1}{\mapsto} v_x]\Big) \qquad \text{Mut. Excl.}$$

$$= \texttt{WMC}\Big(((\dot\varphi_2 \overset{\tau_2}{\Longleftrightarrow} v_2) \wedge \gamma_1 \wedge \gamma_2)[\mathbf{x} \overset{\tau_1}{\mapsto} \dot\varphi_1]\Big)$$

- $\texttt{e} = \texttt{observe } g$. Assume $\Gamma \vdash g : \textbf{Bool} \rightsquigarrow (\varphi, \texttt{T}, w)$. This case relies on interpreting the semantics of $\llbracket\texttt{observe } g[x_i \mapsto v_i]\rrbracket(v)$ as $\llbracket g[x_i \mapsto v_i]\rrbracket(\texttt{T}) \times \llbracket\texttt{T}\rrbracket(v)$. Then,

$$\llbracket\texttt{observe } g[x_i \mapsto v_i]\rrbracket(v) = \llbracket g[x_i \mapsto v_i]\rrbracket(\texttt{T}) \times \llbracket\texttt{T}\rrbracket(v)$$

$$= \texttt{WMC}(\varphi \wedge \texttt{T}, w) \times \texttt{WMC}(v \wedge \texttt{T}). \qquad \text{Ind. Hyp.}$$

$$= \texttt{WMC}(\varphi \wedge v, w). \qquad \text{Indep. Conj.}$$

- $\texttt{e} = \texttt{if } g \texttt{ then } \texttt{e}_T \texttt{ else } \texttt{e}_E$. Assume $\Gamma \vdash g : \textbf{Bool} \rightsquigarrow (\varphi_g, \texttt{T}, w_g)$, $\Gamma \vdash \texttt{e}_T : \tau \rightsquigarrow (\dot\varphi_T, \gamma_T, w_T)$, $\Gamma \vdash \texttt{e}_E : \tau \rightsquigarrow (\dot\varphi_E, \gamma_E, w_E)$. Again assume for notational simplicity that all weighted model counts are performed with the weight function $w_g \cup w_2 \cup w_g$ and that the substitutions $[x_i \overset{\tau_i}{\mapsto} v_i]$ have been performed on the compiled formulas. Then,

$$\llbracket\texttt{if } g \texttt{ then } \texttt{e}_T \texttt{ else } \texttt{e}_E\rrbracket(v)$$

$$= \llbracket g\rrbracket(\texttt{T}) \times \llbracket\texttt{e}_T\rrbracket(v) + \llbracket g\rrbracket(\texttt{F}) \times \llbracket\texttt{e}_E\rrbracket(v)$$

$$= \texttt{WMC}(\varphi_g \wedge \texttt{T}) \times \texttt{WMC}((\dot\varphi_T \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_T) + \texttt{WMC}(\overline\varphi_g \wedge \texttt{T}) \times \texttt{WMC}((\dot\varphi_E \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_E) \qquad \text{Ind. Hyp.}$$

$$= \texttt{WMC}(\varphi_g \wedge (\dot\varphi_T \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_T) + \texttt{WMC}(\overline\varphi_g \wedge (\dot\varphi_E \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_E) \qquad \text{Indep. Conj.}$$

$$= \texttt{WMC}((\varphi_g \wedge (\dot\varphi_T \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_T) \vee (\overline\varphi_g \wedge (\dot\varphi_E \overset{\tau}{\Longleftrightarrow} v) \wedge \gamma_E)) \qquad \text{Mut. Excl.}$$

$$= \texttt{WMC}\Big(\Big((\varphi_g \underset{\tau}{\wedge} \dot\varphi_T) \underset{\tau}{\dot\vee} (\overline\varphi_g \underset{\tau}{\wedge} \dot\varphi_E)\Big) \overset{\tau}{\Longleftrightarrow} v \wedge \Big((\varphi_g \wedge \gamma_T) \vee (\overline\varphi_g \wedge \gamma_E)\Big)\Big)$$

$\square$

A.1.3 Theorem 3.2

First we extend Lemma 3.3 to show that Boolean function call compilation is correct. First we need some preliminaries. The semantics and compilation of an expression can only be compared if the function context they are compiled in is *compatible*:

Definition A.1 (Table Compatibility). *Let Φ be a compiled function table, T be a function table, and Γ be a type environment. Then we say T and Φ are* compatible *if for any function identifier x, where $\Gamma(x) = \tau_1 \to \tau_2$ and $\Phi(x) = (\mathbf{x}, \dot{\varphi}, \gamma, w)$, it holds for any argument value $v^x : \tau_1$ and value $v : \tau_2$, $T(x)(v^x)(v) = \mathtt{WMC}\big(((\dot{\varphi} \stackrel{\tau_2}{\iff} v) \wedge \gamma)[\mathbf{x} \stackrel{\tau_1}{\mapsto} v^x], w\big)$.*

Then, we can extend Lemma 3.3 to assume compatible tables:

Theorem A.1 (Boolean Correctness with Procedure Calls). *Let e be a* Dice *expression with function calls, T and Φ be compatible tables, let $\{x_i : \tau_i\}, \Phi \vdash e : \tau \rightsquigarrow (\varphi, \gamma, w)$. Then, for any values $\{v_i : \tau_i\}$ and $v : \tau$, we have that $[\![e[x_i \mapsto v_i]]\!](v) = \mathtt{WMC}\Big(((\varphi \stackrel{\tau}{\iff} v) \wedge \gamma)[\mathbf{x}_i \stackrel{\tau_i}{\mapsto} v_i]\Big)$.*

Proof. The proof is identical to the proof of Lemma 3.3 except for the addition of the function call syntax, which we prove here.

Assume $e = x_1(x_2)$ and assume $\Phi(x_1) = (\mathbf{x}_{arg}, \dot{\varphi}, \gamma, w)$. Assume $(\dot{\varphi}', \gamma', w) = \mathtt{RefreshFlips}(\dot{\varphi}, \gamma, w)$ Then, $x_1(x_2) \rightsquigarrow (\dot{\varphi}[\mathbf{x}_{arg} \mapsto \mathbf{x}_2], \gamma[\mathbf{x}_{arg} \mapsto \mathbf{x}_2], w)$. Then the result follows directly from table compatibility:

$$
\begin{aligned}
[\![x(v^x)]\!] (\mathtt{T}) &= T(x)(v^x)(\mathtt{T}) \\
&= \mathtt{WMC}\big(((\dot{\varphi} \stackrel{\tau_2}{\iff} v) \wedge \gamma)[\mathbf{x} \stackrel{\tau_1}{\mapsto} v^x], w\big) && \text{Table Compatibility} \\
&= \mathtt{WMC}\big(((\dot{\varphi}' \stackrel{\tau_2}{\iff} v) \wedge \gamma')[\mathbf{x} \stackrel{\tau_1}{\mapsto} v^x], w\big) && \text{Defn. of } \mathtt{RefreshFlips}
\end{aligned}
$$

$\square$

Now we are ready for the main theorem:

Theorem A.2 (Typed Program Correctness). *Let p be a* Dice *program $\Gamma \vdash p : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$. Then for any $v : \tau$, we have that $[\![p]\!](v) = \mathtt{WMC}((\dot{\varphi} \stackrel{\tau}{\iff} v) \wedge \gamma, w)$.*

Proof. • *Base case*: $p = e$. Assume $\Gamma, \Phi \bullet e : \tau \rightsquigarrow (\dot{\varphi}, \gamma, w)$. Then, $[\![\bullet e]\!](v) = [\![e]\!](v) = \mathtt{WMC}((\dot{\varphi} \stackrel{\tau}{\iff} v) \wedge \gamma, w)$, by Theorem A.1.

• *Inductive step*: The program is of the form $p_1 = \mathtt{fun}\ x_1(x_2)\ \{e\}\ p_2$.

Assume that $\Gamma, \Phi \vdash \mathtt{fun}\ x_1(x_2)\ \{e\} : \tau_1 \to \tau_2 \rightsquigarrow (\dot{\varphi}_f, \gamma_f, w_f)$. Let $T' = T \cup \{x_1 \mapsto [\![func]\!]\}$ and $\Phi' = \Phi \cup \{x_1 \mapsto (\mathbf{x}_2, \dot{\varphi}_f, \gamma_f, w_f)\}$. Then, Theorem A.1 guarantees that T' and Φ' are

compatible tables. Let $\Gamma \cup \{x_1 \mapsto \tau_1 \rightarrow \tau_2\}, \Phi' \vdash \mathsf{p_2} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$. Then,

$$\llbracket \mathsf{fun}\ x_1(x_2)\ \{\mathsf{e}\}\ \mathsf{p_2} \rrbracket^T (v) = \llbracket \mathsf{p_2} \rrbracket^{T'} (v)$$
$$= \mathsf{WMC}\Big(\big(\dot\varphi \overset{\mathcal{T}}{\Leftrightarrow} v\big) \wedge \gamma, w\Big) \qquad\qquad \text{By Ind. Hyp.}$$

$\square$

Finally we prove Theorem 3.1, restated here for convenience:

Theorem A.3 (Compilation Correctness)**.** *Let* p *be a* Dice *program and* $\emptyset, \emptyset \vdash \mathsf{p} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$. *Then:*

- $\llbracket \mathsf{p} \rrbracket_A = \mathsf{WMC}(\gamma, w)$

- *for any value* $v : \tau$, $\llbracket \mathsf{p} \rrbracket_D (v) = \mathsf{WMC}((\dot\varphi \overset{\mathcal{T}}{\Leftrightarrow} v) \wedge \gamma, w)/\mathsf{WMC}(\gamma, w)$.

Proof. Let $\{\}, \{\} \vdash \mathsf{p} : \tau \rightsquigarrow (\dot\varphi, \gamma, w)$. Then,

$$\llbracket \mathsf{p} \rrbracket_A = \sum_v \mathsf{WMC}((\dot\varphi \overset{\mathcal{T}}{\Leftrightarrow} v) \wedge \gamma, w) \qquad\qquad \text{Theorem 3.2}$$

$$= \mathsf{WMC}\left(\bigvee_v ((\dot\varphi \overset{\mathcal{T}}{\Leftrightarrow} v) \wedge \gamma), w\right) \qquad\qquad \text{Mut. Excl.}$$

$$= \mathsf{WMC}(\gamma, w).$$

Then, $\llbracket \mathsf{p} \rrbracket_D (v) = \llbracket \mathsf{p} \rrbracket (v)/\sum_{v'} \llbracket \mathsf{p} \rrbracket (v') = \mathsf{WMC}((\dot\varphi \overset{\mathcal{T}}{\Leftrightarrow} v) \wedge \gamma, w)/\mathsf{WMC}(\gamma, w)$ by Theorem 3.2 and the above argument. $\square$

A.2   Chapter 4

Theorem 4.7. The proof will proceed as follows. First, we will split up $\Pr$ into two distributions: a between-orbit distribution, which describes the probability of transitioning between two orbits, and a within-orbit distribution, which is uniform. We will bound the total variation distance for these two quantities, and combine these results to get a bound on the total variation distance on the original distribution using the following lemma:

Lemma A.5. *Let* $\mu(x, y)$ *and* $\nu(x, y)$ *be two distributions on* $X \times Y$. *Let* $\mu_x(x) = \sum_y \mu(x, y)$, *defined similarly for* ν. *If for all* $(x, y) \in X \times Y$ *it holds that* $\Pr_\mu(y \mid x) = \Pr_\nu(y \mid x)$, *then* $d_{TV}(\mu, \nu) = d_{TV}(\mu_x, \nu_x)$.

Proof.

$$
\begin{aligned}
d_{TV}(\mu, \nu) =& \frac{1}{2} \sum_{x,y} |\mathrm{Pr}_\mu(x,y) - \mathrm{Pr}_\nu(x,y)| & \\
=& \frac{1}{2} \sum_{x,y} |\mathrm{Pr}_\mu(y \mid x)\mathrm{Pr}_{\mu_x}(x) - \mathrm{Pr}_\nu(y \mid x)\mathrm{Pr}_{\nu_x}(x)| & \text{Chain rule} \\
=& \frac{1}{2} \sum_{x,y} \mathrm{Pr}_\mu(y \mid x) \times |\mathrm{Pr}_{\mu_x}(x) - \mathrm{Pr}_{\nu_x}(x)| & \text{Since } 0 \leq \mathrm{Pr}_\mu(y \mid x) = \mathrm{Pr}_\nu(y \mid x) \leq 1 \\
=& \frac{1}{2} \sum_x \left(|\mathrm{Pr}_{\mu_x}(x) - \mathrm{Pr}_{\nu_x}(x)| \times \underbrace{\sum_y \mathrm{Pr}_\mu(y \mid x)}_{=1} \right) & \\
=& d_{TV}(\mu_x, \nu_x). &
\end{aligned}
$$

$\square$

Now we begin the main proof. Let $\mathrm{Pr}(x)$ be a $\mathcal{G}$-invariant distribution on a set Ω, and let $P_x^t(y)$ be the probability of transitioning from a state x to a state y after t steps under the orbit-jump proposal. We can write $\mathrm{Pr}(x)$ as a product of a *between-orbit* (Pr_B) and *within-orbit* (Pr_W) distribution, where Pr_B is a distribution on $\Omega/\mathcal{G}$ and Pr_W is a distribution on Ω:

$$
\mathrm{Pr}(x) = \underbrace{\mathrm{Pr}(x) \times |\mathrm{Orb}(x)|}_{\mathrm{Pr}_B(\sigma(x))} \times \underbrace{\frac{1}{|\mathrm{Orb}(x)|}}_{\mathrm{Pr}_W(x|\sigma(x))} \tag{A.1}
$$

I.e., for some $o \in \Omega/\mathcal{G}$, for some $x \in \sigma^{-1}(o)$, $\mathrm{Pr}_B(o) = \mathrm{Pr}(x) \times |\mathrm{Orb}(x)|$. Similarly, the distribution P_x^t can be divided into a between-orbit and within-orbit component. We define a new Markov chain B *between orbits* that has the following transition rule from some initial state $\sigma(x) \in \Omega/\mathcal{G}$:

1. Sample $x' \sim \mathrm{Pr}_{\Omega/\mathcal{G}}$
2. Accept $\sigma(x')$ with probability $\frac{\mathrm{Pr}(x') \times |\mathrm{Orb}(x')|}{\mathrm{Pr}(x) \times |\mathrm{Orb}(x)|}$.

Then, for some $y \in \Omega$ and $\hat{y} = \sigma(y)$,

$$
P_x^t(y) = B_{\sigma(x)}^t(\hat{y}) \times \mathrm{Pr}_W(y \mid \hat{y}), \tag{A.2}
$$

where we used the important fact that the orbit-jump proposal that defines P_x^t is uniform within orbits. Now we can rewrite the total variation distance that we wish to upper-

bound:

$$d_{TV}\left(P_x^t(y), \Pr(y)\right) = d_{TV}\left(B_{\sigma(x)}^t(\hat{y}) \times \Pr_W(y \mid \hat{y}), \Pr_B(\hat{y}) \times \Pr_W(y \mid \hat{y})\right) \tag{A.3}$$

Now using Lemma A.5 we can simplify the bound on the total variation distance to be the total variation distance of the between-orbit distributions:

$$d_{TV}(P_x^t(y), \Pr(y)) = d_{TV}(B_{\sigma(x)}^t(\hat{y}), \Pr_B(\hat{y})). \tag{A.4}$$

Now, our goal is to upper-bound $d_{TV}(B_{\sigma(x)}^t, \Pr_B)$. To do this we will use a standard *coupling argument*. A *coupling* is a way to run two copies of a Markov chain P at the same time with the following properties:

1. Both copies in isolation evolve according to P;
2. If both copies are in the same state, they remain in the same state.

Two coupled chains can be used to acquire upper-bounds on the total variation distance of a Markov chain by upper-bounding the probability that a Markov chain starting from two initial distributions – one in its stationary distribution and the other in an arbitrary location – will *coalesce* into the same state:

Lemma A.6 ([Levin and Peres, 2017] Theorem 5.4). *Let P be a transition matrix on state-space Ω with stationary distribution π. Let $\{(X_t, Y_t)\}$ be coupled chains that evolve according to P of length t, starting from an initial state $x \in \Omega$ and $y \sim \pi$. Then,*

$$d_{TV}(P_x^t, \pi) \leq \Pr(X_t \neq Y_t). \tag{A.5}$$

Now we define the coupled chains $\{(X_t, Y_t)\}$. Let $X_0 \in \Omega/\mathcal{G}$ be an arbitrarily chosen initial element, and let $Y_0 \sim \Pr_B$ be an element chosen according to $\Pr_B$. At each time step t, choose a state $o \in \Omega/\mathcal{G}$ uniformly at random. Then, *both* chains attempt to transition to o, using the standard metropolis correction criteria to decide whether or not to accept o. In order to guarantee coalescence, if both chains are in the same state, then we define them to accept or reject a new state together. Intuitively, these two chains simulate the Markov chain B starting from different initial states, where they both share a common source of randomness. Then by Lemma A.6,

$$d_{TV}(B_{X_0}^t, \Pr_B) \leq \Pr(X_t \neq Y_t). \tag{A.6}$$

This probability can be upper bounded as follows. There exists a (possibly non-

unique) maximum probability state $M \in \Omega/\mathcal{G}$:

$$M = \sigma \left(\arg \max_x \, \Pr(x) \times |\mathrm{Orb}(x)| \right).$$

If both Markov chains uniformly choose M to transition to, then by the Metropolis rule they will both accept and thus coalesce. Since the proposal is uniform, $\Pr(X_t \neq Y_t)$ is upper-bounded by the probability of not transitioning to M after t steps, so:

$$\Pr(X_t \neq Y_t) \leq \left(\frac{|\Omega/\mathcal{G}| - 1}{|\Omega/\mathcal{G}|} \right)^t, \tag{A.7}$$

which gives the first bound in the theorem. This quantity can be upper bounded by a parameter $\varepsilon > 0$ representing the chosen error tolerance. Solving for t:

$$t \geq \log(\varepsilon) \times \left[\log \left(\frac{|\Omega/\mathcal{G}| - 1}{|\Omega/\mathcal{G}|} \right) \right]^{-1}$$

Using the identity:

$$\log \left(\frac{x - 1}{x} \right) = - \left(\frac{1}{x} + \frac{1}{2x^2} + \cdots \right),$$

we then have that:

$$t \geq \log(\varepsilon^{-1}) \times |\Omega/\mathcal{G}| \geq \log(\varepsilon^{-1}) \times \left(\frac{1}{|\Omega/\mathcal{G}|} + \frac{1}{2|\Omega/\mathcal{G}|^2} + \cdots \right)^{-1}, \tag{A.8}$$

which gives the second bound and concludes the proof.

$\square$

Bibliography

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*. Addison Wesley, 1986.

Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: Probabilistic verification of program fairness. *Proc. ACM Program. Lang.*, 1(OOPSLA):80:1–80:30, October 2017. ISSN 2475-1421. doi: 10.1145/3133904. URL `http://doi.acm.org/10.1145/3133904`.

Michael Artin. *Algebra*. Birkhäuser, 1998.

Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 1961-12:–, 1961.

R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

Lindsay A Becker, Brenda Huang, Gregor Bieri, Rosanna Ma, David A Knowles, Paymaan Jafar-Nejad, James Messing, Hong Joo Kim, Armand Soriano, Georg Auburger, et al. Therapeutic reduction of ataxin-2 extends lifespan and reduces pathology in tdp-43 mice. *Nature*, 544(7650):367, 2017.

Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. pages 2770–2776, 2015.

Armin Biere. Bounded model checking. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 14. IOS Press, 2009.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research (JMLR)*, 20(1):973–978, 2019.

Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. In *European Symposium on Programming (ESOP)*, pages 77–96. Springer, 2011.

James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. Uncertain<t>: A first-order type for uncertain data. In *ACM SIGPLAN Notices*, volume 49, pages 51–66. ACM, 2014. doi: 10.1145/2654822.2541958.

Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 115–123. Morgan Kaufmann Publishers Inc., 1996.

R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE TC*, C-35: 677–691, 1986. doi: 10.1109/TC.1986.1676819.

Hung Hai Bui, Tuyen N. Huynh, and Sebastian Riedel. Automorphism groups of graphical models and lifted variational inference. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 132–141, 2013. URL `http://dl.acm.org/citation.cfm?id=3023638.3023652`.

Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *J. Statistical Software*, VV(Ii), 2016. doi: 10.18637/jss.v076.i01.

Arun Chaganty, Aditya V Nori, and Sriram K Rajamani. Efficiently Sampling Probabilistic Programs via Program Analysis. *Conference on Artificial Intelligence and Statistics (AISTATS)*, 31:153–160, 2013. ISSN 15337928.

Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1306–1312, 2005.

Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *J. Artificial Intelligence*, 172(6-7):772–799, April 2008. ISSN 0004-3702. doi: 10.1016/j.artint.2007.11.002.

Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal on Approximate Reasoning (IJAR)*, 42 (1):4–20, 2006.

Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in smt and value estimation for probabilistic programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 320–334, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-662-46680-3. doi: 10.1007/978-3-662-46681-0_26.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. oct 2020. URL `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf`.

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. *Foundations of Software Engineering (FSE)*, page 92, 2013. doi: 10.1145/2491411.2491423. URL `http://dl.acm.org/citation.cfm?doid=2491411.2491423`.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer-Aided Verification (CAV)*, pages 154–169. Springer, 2000.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003. ISSN 0004-5411. doi: 10.1145/876638.876643.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977. doi: 10.1145/512950.512973.

Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In *European Symposium on Programming (ESOP)*, pages 169–193, 2012. ISBN 9783642288685. doi: 10.1007/978-3-642-28869-2_9.

Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin Vechev, and Vikash K Mansinghka. Incremental inference for probabilistic programs. In *ACM SIGPLAN Notices*, volume 53, pages 571–585. ACM, 2018. doi: 10.1145/3296979.3192399.

A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009. doi: 10.1017/CBO9780511811357.

Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. page 819, 2011.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. volume 7, pages 2462–2467, 2007.

Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. pages 1319–1325, 2005.

Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

Neil Dhir, Frank Wood, Matthijs Vákár, Andrew Markham, Matthew Wijers, Paul Trethowan, Byron Du Preez, Andrew Loveridge, and David MacDonald. Interpreting lion behaviour with nonparametric probabilistic programs. *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

Persi Diaconis and David Freedman. De Finetti's generalizations of exchangeability. In *Studies in Inductive Logic and Probability*, pages 2–233. University of California Press, 1980.

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. Exact and approximate weighted model integration with probability density functions using knowledge compilation. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 33, pages 7825–7833, 2019.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *J. Theory and Practice of Logic Programming*, 15(3):358 – 401, 2015. doi: 10.1017/S1471068814000076.

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

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247. ACM, 1993. doi: 10.1145/155090.155113. URL `https://doi.org/10.1145/155090.155113`.

GAP. *GAP – Groups, Algorithms, and Programming, Version 4.10.0*. The GAP Group, 2018.

Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 9779:62–83, 2016. ISSN 16113349. doi: 10.1007/978-3-319-41528-4_4.

Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: probabilistic inference for networks. In *ACM SIGPLAN Notices*, volume 53, pages 586–602. ACM, 2018. doi: 10.1145/3296979.3192400.

Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012. doi: 10.1145/2338965.2336773.

Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. The MIT Press, 2007. ISBN 0262072882.

V. Gogate and R. Dechter. Samplesearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2):694–729, 2011.

Vibhav Gogate and Pedro Domingos. Probabilistic theorem proving. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 256–265, 2011.

Vibhav Gogate, Abhay Kumar Jha, and Deepak Venugopal. Advances in lifted importance sampling. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

L.A. Goldberg. Computation in permutation groups: Counting and randomly sampling orbits. *Surveys in Combinatorics*, pages 109–143, 2001.

Leslie Ann Goldberg and Mark Jerrum. The Burnside process converges slowly. *Combinatorics, Probability and Computing*, 11(1):21–34, 2002. doi: 10.1017/S096354830100493X.

Noah D Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

Maria I Gorinova, Dave Moore, and Matthew D Hoffman. Automatic reparameterisation of probabilistic programs. *International Conference on Machine Learning (ICML)*, 2020.

Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer-Aided Verification (CAV)*, volume 1254, pages 72–83. Springer-Verlag, June 1997.

Bradley Gram-Hansen, Yuan Zhou, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. Hamiltonian monte carlo for probabilistic programs with discontinuities. *arXiv preprint arXiv:1804.03523*, 2018.

Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research (JMLR)*, 15(1):1593–1623, 2014.

Steven Holtzen, Yibiao Zhao, Tao Gao, Joshua B Tenenbaum, and Song-Chun Zhu. Inferring human intent from video by sampling hierarchical plans. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1489–1496. IEEE, 2016.

Steven Holtzen, Todd Millstein, and Guy Van den Broeck. Probabilistic program abstractions. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Sound abstraction and decomposition of probabilistic programs. In *International Conference on Machine Learning (ICML)*, 2018.

Steven Holtzen, Todd Millstein, and Guy Van den Broeck. Generating and sampling orbits for lifted probabilistic inference. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence (UAI)*, jul 2019. URL `http://starai.cs.ucla.edu/papers/HoltzenUAI19.pdf`.

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, pages 140:1–140:31, 2020. doi: 10.1145/3428208.

Steven Holtzen, Sebastian Junges, Marcell Vazquz-Chanlatte, Todd Millstein, Sanjit Seshia, and Guy Van den Broeck. Model checking finite-horizon markov chains with probabilistic inference. In *International Conference on Computer Aided Verifiation (CAV)*, 2021.

Mevin B Hooten, Devin S Johnson, Brett T McClintock, and Juan M Morales. *Animal movement: statistical models for telemetry data*. CRC press, 2017.

Daniel Huang and Greg Morrisett. An application of computable distributions to the semantics of probabilistic programming languages. In *European Symposium on Programming (ESOP)*, pages 337–363, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_14. URL `https://doi.org/10.1007/978-3-662-49498-1_14`.

Jonathan Huang, Carlos Guestrin, and Leonidas Guibas. Fourier theoretic probabilistic inference over permutations. *Journal of Machine Learning Research (JMLR)*, 10(5), 2009.

Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. Logical abstractions for noisy variational quantum algorithm simulation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Samuel. Slicing probabilistic programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 133–144, 2014. doi: 10.1145/2594291.2594303.

Chung-Kil Hur, Aditya V. Nori, Sriram K. Rajamani, and Selva Sammuel. A Provably Correct Sampler for Probabilistic Programs. *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, (FSTTCS):1–14, 2015. ISSN 18688969. doi: 10.4230/LIPIcs.FSTTCS.2015.475.

Ian J Jacobs, Usha Menon, Andy Ryan, Aleksandra Gentry-Maharaj, Matthew Burnell, Jatinderpal K Kalsi, Nazar N Amso, Sophia Apostolidou, Elizabeth Benjamin, Derek Cruickshank, et al. Ovarian cancer screening and mortality in the uk collaborative trial of ovarian cancer screening (ukctocs): a randomised controlled trial. *The Lancet*, 387 (10022):945–956, 2016.

Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

Mark Jerrum. Uniform sampling modulo a group of symmetries using markov chain simulation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 37–47, 1993.

Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, 2009. ISSN 03600300. doi: 10.1145/1592434.1592438.

M.I. Jordan, Z. Ghahramani, T.S. Jaakkola, and L.K. Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999. doi: 10.1023/A: 1007665907178.

Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

Kristian Kersting. Lifted probabilistic inference. In *European Conference on Artificial Intelligence (ECAI)*, pages 33–38, 2012. ISBN 978-1-61499-097-0.

Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting belief propagation. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 277–284, 2009.

D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009a.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009b. ISBN 0262013193, 9780262013192.

Kevin B Korb and Ann E Nicholson. *Bayesian artificial intelligence*. CRC press, 2010. doi: 10.1201/b10391.

Dexter Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science (FOCS)*, SFCS '79, pages 101–114, Washington, DC, USA, 1979. IEEE Computer Society. doi: 10.1109/SFCS.1979.38.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. Automatic variational inference in stan. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 568–576, 2015.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic differentiation variational inference. *The Journal of Machine Learning Research*, 18(1):430–474, 2017.

A. Kulesza and B. Taskar. Determinantal point processes for machine learning. *Found. Trends Mach. Learn.*, 5:123–286, 2012.

Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer-Aided Verification (CAV)*, pages 585–591, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_47.

Johan Henri Petrus Kwisthout. *The computational complexity of probabilistic networks*. Utrecht University, 2009.

David A Levin and Yuval Peres. *Markov chains and mixing times*. American Mathematical Society, 2017.

Michael L Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998. doi: 10.1613/jair.505.

Jun S Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119, 1996.

Gagan Madan, Ankit Anand, Mausam, and Parag Singla. Block-value symmetries in probabilistic graphical models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 886–895, 2018.

Vikash Mansinghka, Tejas D Kulkarni, Yura N Perov, and Josh Tenenbaum. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 1520–1528, 2013.

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Conference on Programming Language Design and Implementation (PLDI)*, PLDI 2018, pages 603–616, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192409. URL http://doi.acm.org/10.1145/3192366.3192409.

François Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98:3–21, 2003.

François Margot. Symmetry in integer linear programming. In *50 Years of Integer Programming*, 2010.

A McCallum, K Schultz, and S Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. *Conference on Neural Information Processing Systems (NeurIPS)*, 22:1249–1257, 2009. ISSN 03643417.

Annabelle McIver, Carroll Morgan, and Charles Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

Brendan D McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26(2): 306 – 324, 1998. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1997.0898. URL http://www.sciencedirect.com/science/article/pii/S0196677497908981.

Brendan D. Mckay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer Verlag, 1998. doi: 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. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

David Monniaux. Abstract interpretation of probabilistic semantics. In *International Symposium on Static Analysis*, pages 322–339, 2000. ISBN 3-540-67668-6.

David Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. *ACM SIGPLAN Notices*, 36(3):93–101, January 2001. ISSN 0362-1340. doi: 10.1145/373243.360211.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 0262018020, 9780262018029.

Chandrakana Nandi, Dan Grossman, Adrian Sampson, Todd Mytkowicz, and Kathryn S McKinley. Debugging probabilistic programs. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 18–26. ACM, 2017.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming*, pages 62–79. Springer, 2016. doi: 10.1007/978-3-319-29604-3_5. URL http://dx.doi.org/10.1007/978-3-319-29604-3_5.

Mathias Niepert. Markov chains on orbits of permutation groups. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 624–633, 2012.

Mathias Niepert. Symmetry-aware marginal density estimation. *AAAI Conference on Artificial Intelligence (AAAI)*, 2013.

Mathias Niepert and Guy Van den Broeck. Tractability through exchangeability: A new perspective on efficient probabilistic inference. In *AAAI*, 2014.

Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 2476–2482, 2014.

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. Tensor variable elimination for plated factor graphs. *International Conference on Machine Learning (ICML)*, pages 4871–4880, 2019.

James Ostrowski, Jeff T. Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. In *IPCO*, 2007.

Igor Pak. What do we know about the product replacement algorithm? In *Groups and Computation III*, pages 301–347, 2000.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Avi Pfeffer. The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. *Introduction to statistical relational learning*, (1993):399, 2007a.

Avi Pfeffer. A general importance sampling algorithm for probabilistic programs. 2007b. URL http://nrs.harvard.edu/urn-3:HUL.InstRepos:25235125.

Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

Avi Pfeffer, Brian Ruttenberg, William Kretschmer, and Alison OConnor. Structured factored inference for probabilistic programming. In *Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1224–1232, 2018.

David Poole. First-order probabilistic inference. 2003.

Fabrizio Riguzzi and Terrance Swift. The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty. *Theory and Practice of Logic Programming*, 11(4–5):433–449, 2011. doi: 10.1017/S147106841100010X.

Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah D. Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances in Neural Information Processing Systems (NIPS 2016)*, 2016.

Feras Saad and Vikash Mansinghka. A probabilistic programming approach to probabilistic data analysis. In *Advances in Neural Information Processing Systems (NIPS)*. 2016.

Ashish Sabharwal. Symchaff: A structure-aware satisfiability solver. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 5, pages 467–474, 2005.

Tian Sang, Paul Beame, and Henry A Kautz. Performing bayesian inference by weighted model counting. In *AAAI Conference on Artificial Intelligence (AAAI)*, volume 5, pages 475–481, 2005.

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices*, 48(6):447–458, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462179.

Àkos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003.

Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

Fabio Somenzi. CUDD: BDD package, University of Colorado, Boulder.

The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.5.0)*, 2018. https://www.sagemath.org.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Jan-Willem van de Meent, Hongseok Yang, Vikash Mansinghka, and Frank Wood. Particle gibbs with ancestor sampling for probabilistic programs. In *Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015.

Rens Van de Schoot, Sonja D Winter, Oisín Ryan, Mariëlle Zondervan-Zwijnenburg, and Sarah Depaoli. A systematic review of bayesian articles in psychology: The last 25 years. *Psychological Methods*, 22(2):217, 2017.

Guy Van den Broeck. *Lifted inference and learning in statistical relational models*. PhD thesis, 2013.

Guy Van den Broeck and Mathias Niepert. Lifted probabilistic inference for asymmetric graphical models. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

Guy Van den Broeck and Dan Suciu. *Query Processing on Probabilistic Data: A Survey*. Foundations and Trends in Databases. Now Publishers, August 2017. doi: 10.1561/ 1900000052. URL `http://web.cs.ucla.edu/~guyvdb/papers/VdBFTDB17.pdf`.

Marcell Vazquez-Chanlatte and Sanjit A Seshia. Maximum causal entropy specification inference from demonstrations. In *International Conference on Computer Aided Verification*. Springer, 2020.

Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. July 2015. doi: 10.1016/j.ijar.2016.06.009.

Di Wang, Jan Hoffmann, and Thomas Reps. Pmaf: An algebraic framework for static analysis of probabilistic programs. *SIGPLAN Not.*, 53(4):513–528, June 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192408. URL `https://doi.org/10.1145/3296979.3192408`.

David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1024–1032, 2014.

Zhe Zeng and Guy Van den Broeck. Efficient search-based weighted model integration. In *Uncertainty in Artificial Intelligence*, pages 175–185. PMLR, 2020.

Honghua Zhang, Steven Holtzen, and Guy Van den Broeck. On the relationship between probabilistic circuits and determinantal point processes. In *Proceedings of the 36th Conference on Uncertainty in Aritifical Intelligence (UAI)*, aug 2020. URL `http://starai.cs.ucla.edu/papers/ZhangUAI20.pdf`.

Honghua Zhang, Brendan Juba, and Guy Van den Broeck. Probabilistic generating circuits. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, jul 2021.

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