Symbolic Exact Inference for Discrete Probabilistic Programs

Steven Holtzen
UCLA
sholtzen@cs.ucla.edu

Todd Millstein
UCLA
todd@cs.ucla.edu

Guy Van den Broeck
UCLA
guyvdb@cs.ucla.edu

Abstract

The computational burden of probabilistic inference remains a hurdle for applying probabilistic programming languages to practical problems of interest. In this work, we provide a semantic and algorithmic foundation for efficient exact inference on discrete-valued finite-domain imperative probabilistic programs. We leverage and generalize efficient inference procedures for Bayesian networks, which exploit the structure of the network to decompose the inference task, thereby avoiding full path enumeration. To do this, we first compile probabilistic programs to a symbolic representation. Then we adapt techniques from the probabilistic logic programming and artificial intelligence communities in order to perform inference on the symbolic representation. We formalize our approach, prove it sound, and experimentally validate it against existing exact and approximate inference techniques. We show that our inference approach is competitive with inference procedures specialized for Bayesian networks, thereby expanding the class of probabilistic programs that can be practically analyzed.

1 Introduction

When it is computationally feasible, exact probabilistic inference is vastly preferable to approximation techniques. Exact inference methods are deterministic and reliable, so they can be trusted for making high-consequence decisions and do not propagate errors to subsequent analyses. Ideally, one would use exact inference *whenever possible*, only resorting to approximation when exact inference strategies become infeasible. Even when approximating inference, one often performs exact inference in an approximate model. This is the case for a wide range of approximation schemes, including message passing [8, 40], sampling [19, 22], and variational inference [44].

Existing probabilistic programming systems lag behind state-of-the-art techniques for performing exact probabilistic inference in other domains such as graphical models. Fundamentally, inference – both exact and approximate – is theoretically hard [37]. However, exact inference is routinely performed in practice. This is because many interesting inference problems have *structure*: there are underlying repetitions and decompositions that can be exploited to perform inference more efficiently than the worst case. Existing efficient exact inference procedures – notably techniques from the graphical models inference community – systematically find and exploit the underlying structure of the problem in order to mitigate the inherent combinatorial explosion problem of exact probabilistic inference [4, 23, 31].

We seek to close the performance gap between exact inference in discrete graphical models and discrete-valued finite-domain probabilistic programs. The key idea behind existing state-of-the-art inference procedures in discrete graphical models is to compile the graphical model into a representation known as a *weighted Boolean formula* (WBF), which is a symbolic representation of the joint probability distribution over the graphical model's random variables. This symbolic representation exposes key structural elements of the distribution, such as independences between random variables. Then, inference is performed via a weighted sum of the models of the WBF, a process known as *weighted model counting* (WMC). This WMC process exploits the independences present in the WBF and is competitive with state-of-the-art inference techniques in many domains, such as probabilistic logic programming, Bayesian networks, and probabilistic databases [7, 7, 9, 18, 41, 42].

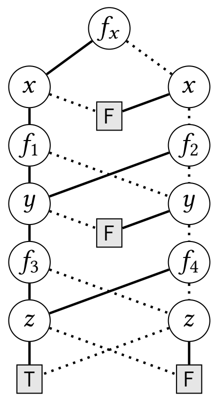
First we give a motivating example that highlights key properties of our approach. Then, we describe our symbolic compilation in more detail; the precise details of our compilation, and its proof of correctness, can be found in the appendix. Then, we illustrate how to use binary decision diagrams to represent the probability distribution of a probabilistic program for efficient inference. Finally, we provide preliminary experimental results illustrating the promise of this approach on several challenging probabilistic programs.

2 Exact Symbolic Inference

In this section we present a motivating example that highlights key elements of our approach. Figure 1a shows a simple probabilistic program that encodes a linear Bayesian network, a structure known as a *Markov chain* [23]. In order to perform inference efficiently on a Markov chain – or any Bayesian network – it is necessary to exploit the *independence structure* of the model. Exploiting independence is one of the key techniques for efficient graphical model inference procedures. Markov chains encoded as probabilistic programs have 2^n paths, where n is the length of the chain. Thus, inference methods that rely on exhaustively exploring the paths in a program – a strategy we refer to as *path-based* inference methods – will require exponential time in the

```
x ~flip_x(0.5);
if(x) { y ~flip_1(0.6) }
else { y ~flip_2(0.4) };
if(y) { z ~flip_3(0.6) }
else { z ~flip_4(0.9) }
```

(a) A simple probabilistic program. The notation $x \sim \texttt{flip}_l(\theta)$ denotes drawing a sample from a Bernoulli(θ) distribution and assigning the outcome to the variable x. The label l is not actually part of the syntax but is used so we can refer to each `flip` uniquely.



(b) A binary decision diagram representing the Boolean formula compiled from the program in Figure 1a; a low edge is denoted a dashed line, and a high edge is denoted with a solid line. The variables f_x, f_1, f_2, f_3, and f_4 correspond to annotations in Figure 1a.

Figure 1. Probabilistic program and its symbolic representation.

length of the Markov chains; see our experiments in Figure 5. Path-based inference is currently a common strategy for performing discrete exact inference in the probabilistic programming community [2, 20, 39].

However, it is well known that Markov chains support linear-time inference in the length of the chain [23]. The reason for this is that the structure of a Markov chain ensures a strong form of *conditional independence*: each node in the chain depends only on the directly preceding node in the chain. In the program of Figure 1a, for example, the probability distribution for z is independent of x *given* y, i.e., if y is fixed to a particular value, then the probability distribution over z can be computed without considering the distribution over x. Therefore inference can be *factorized*: the probability distribution for y can be determined as a function of that for x, and then the probability distribution for z can be determined as a function of that for y. More generally, inference for a chain of length n can be reduced to inference on $n-1$ separate chains, each of length two.

To close this performance gap between Bayesian networks and exact PPL inference, we leverage and generalize state-of-the-art techniques for Bayesian inference, which represent the distribution *symbolically* [6, 18]. In this style, the Bayesian network is compiled to a Boolean function and represented using a binary decision diagram (BDD) or related data structure [14]. The BDD structure directly exploits conditional independences – as well as other forms of independence – by caching and re-using duplicate sub-functions during compilation [1].

In this paper we describe an algorithm for compiling a probabilistic program to a Boolean formula, which can then be represented by a BDD. As an example, Figure 1b shows a BDD representation of the program in Figure 1a. The outcome of each $\texttt{flip}_l(\theta)$ expression in the program is encoded as a Boolean variable labeled f_l. A *model* of the BDD is a truth assignment to all the variables in the BDD that causes the BDD to return T, and each model of the BDD in Figure 1b represents a possible execution of the original program.

The exploitation of the conditional independence structure of the program is clearly visible in the BDD. For example, any feasible execution in which y is true has the same subfunction for z — the subtree rooted at f_3 — regardless of the value of x. The same is true for any feasible execution in which y is false. More generally, the BDD for a Markov chain has size linear in the length of the chain, despite the exponential number of possible execution paths.

To perform inference on this BDD, we first associate a *weight* with each truth assignment to each variable: the variables x, y, and z are given a weight of 1 for both the true and false assignments, and the $\texttt{flip}(\theta)$ variables are given a weight of θ and $1-\theta$ for their true and false assignments respectively. The Boolean formula together with these weights is called a *weighted Boolean formula*.

Finally, we can perform inference on the original probabilistic program relative to a given *inference query* (e.g., "What is the probability that z is false?") via *weighted model counting* (WMC). The weight of a model of the BDD is defined as the product of the weights of each variable assignment in the model, and the WMC of a set of models is the sum of the weights of the models. Then the answer to a given inference query Q is simply the WMC of all models of the BDD that satisfy the query. WMC is a well-studied general-purpose technique for performing probabilistic inference and is currently the state-of-the-art technique for inference in discrete Bayesian networks, probabilistic logic programs, and probabilistic databases [6, 18, 41]. BDDs support linear-time weighted model counting by performing a single bottom-up pass of the diagram [14]: thus, we can compile a single BDD for a probabilistic program, which can be used to exactly answer many inference queries.

3 The DIPPL Language

Here we formally define the syntax and semantics of our discrete finite-domain imperative probabilistic programming language DIPPL language. First we will introduce and discuss the syntax. Then, we will describe the semantics and its

```
s ::=
  | s; s
  | x := e
  | x ~ flip(θ)
  | if e { s } else { s }
  | observe(e)
  | skip
e :: =
  | x
  | T | F
  | e ∨ e
  | e ∧ e
  | ¬ e
```

Figure 2. Syntax of DIPPL.

basic properties. For more details on the semantics, see the appendix.

3.1 Syntax

Figure 2 gives the syntax of our probabilistic programming language DIPPL. Metavariable x ranges over variable names, and metavariable θ ranges over rational numbers in the interval $[0, 1]$. All data is Boolean-valued, and expressions include the usual Boolean operations, though it is straightforward to extend the language to other finite-domain datatypes. In addition to the standard loop-free imperative statements, there are two probablistic statements. The statement $\mathtt{x} \sim \mathtt{flip}(\theta)$ samples a value from the Bernoulli distribution defined by parameter θ (i.e., T with probability θ and F with probability $1 - \theta$). The statement `observe(e)` conditions the current distribution of the program on the event that `e` evaluates to true.

3.2 Semantics

The goal of the semantics of any probabilistic programming language is to define the distribution over which one wishes to perform inference. In this section, we introduce a denotational semantics that directly produces this distribution of interest, and it is defined over program states. A state σ is a finite map from variables to Boolean values, and Σ is the set of all possible states.

We define a denotational semantics for DIPPL, which we call its *transition semantics* and denote $[\![\cdot]\!]_T$. These semantics are given in the appendix. The transition semantics will be the primary semantic object of interest for DIPPL, and will directly produce the distribution over which we wish to perform inference. For some statement `s`, the transition semantics is written $[\![\mathtt{s}]\!]_T(\sigma' \mid \sigma)$, and it computes the (normalized) conditional probability upon executing `s` of transitioning to state σ'. The transition semantics have the following type signature:

$$[\![\mathtt{s}]\!]_T : \Sigma \to \mathtt{Dist}\ \Sigma$$

where $\mathtt{Dist}\ \Sigma$ is the set of all probability distributions over Σ. For example,

$$[\![x \sim \mathtt{flip}(\theta)]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} \theta & \text{if } \sigma' = \sigma[x \mapsto \mathtt{T}] \\ 1 - \theta & \text{if } \sigma' = \sigma[x \mapsto \mathtt{F}] \\ 0 & \text{otherwise} \end{cases}$$

Ultimately, our goal during inference is to compute the probability of some event occurring in the probability distribution defined by the transition semantics of the program.

4 Symbolic Compilation for Inference

Existing approaches to exact inference for imperative PPLs perform *path enumeration*: each execution path is individually analyzed to determine the probability mass along the path, and the probability masses of all paths are summed. As argued earlier, such approaches are inefficient due to the need to enumerate complete paths and the inability to take advantage of key properties of the probability distribution across paths, notably forms of independence.

In this section we present an alternative approach to exact inference for PPLs, which is inspired by state-of-the-art techniques for exact inference in Bayesian networks [6]. We describe how to compile a probabilistic program to a *weighted Boolean formula*, which symbolically represents the program as a relation between input and output states. Inference is then reduced to performing a *weighted model count* (WMC) on this formula, which can be performed efficiently using BDDs and related data structures.

4.1 Weighted Model Counting

Weighted model counting is a well-known general-purpose technique for performing probabilistic inference in the artificial intelligence and probabilistic logic programming communities, and it is currently the state-of-the-art technique for performing inference in certain classes of Bayesian networks and probabilistic logic programs [6, 18, 38, 41]. There exist a variety of general-purpose black-box tools for performing weighted model counting, similar to satisfiability solvers [27, 29, 30].

First, we give basic definitions from propositional logic. A *literal* is either a Boolean variable or its negation. For a formula φ over variables V, a sentence ω is a *model* of φ if it is a conjunction of literals, contains every variable in V, and $\omega \models \varphi$. We denote the set of all models of φ as $\mathtt{Mod}(\varphi)$. Now we are ready to define a weighted Boolean formula:

Definition 4.1 (Weighted Boolean Formula)**.** Let φ be a Boolean formula, L be the set of all literals for variables that occur in φ, and $w : L \to \mathbb{R}^+$ be a function that associates a real-valued positive weight with each literal $l \in L$. The pair (φ, w) is a *weighted Boolean formula* (WBF).

Next, we define the weighted model counting task, which computes a weighted sum over the models of a weighted Boolean formula:

Definition 4.2 (Weighted Model Count)**.** Let (φ, w) be a weighted Boolean formula. Then, the *weighted model count* (WMC) of (φ, w) is defined as:

$$\mathsf{WMC}(\varphi, w) \triangleq \sum_{\omega \in \mathsf{Mod}(\varphi)} \prod_{l \in \omega} w(l) \tag{1}$$

where the set $l \in \omega$ is the set of all literals in the model ω.

The process of symbolic compilation associates a DIPPL program with a weighted Boolean formula and is described next.

4.2 Symbolic Compilation

We formalize symbolic compilation of a DIPPL program to a weighted Boolean formula as a relation denoted $\mathsf{s} \rightsquigarrow (\varphi, w)$. The formal rules for this relation are described in detail in the appendix; here we describe the important properties of this compilation. Intuitively, the formula φ produced by the compilation represents the program s as a relation between initial states and final states, where initial states are represented by unprimed Boolean variables $\{x_i\}$ and final states are represented by primed Boolean variables $\{x_i'\}$. These compiled weighted Boolean formulas will have a probabilistic semantics that allow them to be interpreted as a transition probability for the original statement.

Our goal is to ultimately give a correspondence between the compiled weighted Boolean formula and the original denotational semantics of the statement. First we define the translation of a state σ to a logical formula:

Definition 4.3 (Boolean state)**.** Let $\sigma = \{(x_1, b_1), \dots, (x_n, b_n)\}$. We define the *Boolean state* $\mathbf{F}(\sigma)$ as $l_1 \wedge \ldots \wedge l_n$ where for each i, l_i is x_i if $\sigma(x_i) = \mathsf{T}$ and $\neg x_i$ if $\sigma(x_i) = \mathsf{F}$. For convenience, we also define a version that relabels state variables to their primed versions, $\mathbf{F}'(\sigma) \triangleq \mathbf{F}(\sigma)[x_i \mapsto x_i']$.

Now, we formally describe how every compiled weighted Boolean formula can be interpreted as a conditional probability by computing the appropriate weighted model count:

Definition 4.4 (Compiled semantics)**.** Let (φ, w) be a weighted Boolean formula, and let σ and σ' be states. Then, the *transition semantics* of (φ, w) is defined:

$$[\![(\varphi, w)]\!]_T(\sigma' \mid \sigma) \triangleq \frac{\mathsf{WMC}(\varphi \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w)}{\mathsf{WMC}(\varphi \wedge \mathbf{F}(\sigma), w)} \tag{2}$$

Moreover, the transition semantics of Definition 4.4 allows for more general queries to be phrased as WMC tasks as well. For example, the probability of some event α being true in the output state σ' can be computed by replacing $\mathbf{F}'(\sigma')$ in Equation 2 by a Boolean formula for α.

Finally, we state our correctness theorem, which describes the relation between the semantics of the compiled WBF to the denotational semantics of DIPPL:

Theorem 4.5 (Correctness of Compilation Procedure)**.** *Let* s *be a* DIPPL *program,* V *be the set of all variables in* s*, and* $\mathsf{s} \rightsquigarrow (\varphi, w)$*. Then for all states* σ *and* σ' *over the variables in* V*, we have:*

$$[\![\mathsf{s}]\!]_T(\sigma' \mid \sigma) = [\![(\varphi, w)]\!]_T(\sigma' \mid \sigma) \tag{3}$$

Proof. A complete proof can be found in Appendix D. □

The above theorem allows us to perform inference via weighted model counting on the compiled WBF for a DIPPL program. See the appendix for details on this compilation procedure, and proof of its correctness.

5 Efficient Inference

Inference is theoretically hard [37]. Exploiting the structure of the problem – and in particular, exploiting various forms of independence – are essential for scalable and practical inference procedures [4, 23, 31]. In this section, we will represent a compiled weighted Boolean formula as a binary decision diagram (BDD). We will show how BDDs implicitly exploit the problem structure.

5.1 BDD Representation

BDDs are a popular choice for representing the set of reachable states in the symbolic model checking community [3]. BDDs support a variety of useful properties which make them suitable for this task: they support an array of pairwise composition operations, including conjunction, disjunction, existential quantification and variable relabeling. These composition operators are *efficient*, i.e. performing them requires time polynomial in the sizes of the two BDDs that are being composed.

In addition to supporting efficient compositional operators, BDDs also support a variety of efficient queries, including satisfiability and weighted model counting [14].

5.2 Exploiting Program Structure

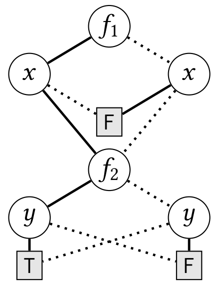
Compilation to BDDs – and related representations – is currently the state-of-the-art approach to inference in certain kinds of discrete Bayesian networks, probabilistic logic programs, and probabilistic databases [6, 18, 41]. The fundamental reason is that BDDs exploit *duplicate sub-functions*: if there is a sub-function that is constructed more than once in the symbolic compilation, that duplicate sub-function is cached and re-used. This sub-function deduplication is critical for efficient inference. In this section, we explore how BDDs exploit specific properties of the program and discuss when a program will have a small BDD.

Independence Exploiting independence is essential for efficient inference and is the backbone of existing state-of-the-art inference algorithms. There are three kinds of independence structure which we seek to exploit. The first is the strongest form:

Definition 5.1 (Independence)**.** Let $\Pr(X, Y)$ be a joint probability distribution over sets of random variables X and Y.

```
1  x ~flip₁(0.6);
2  y ~flip₂(0.7)
```

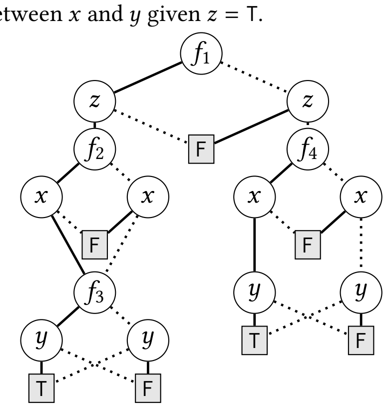
(a) A probabilistic program illustrating independence between variables x and y.



(b) A BDD representing the logical formula compiled from the program in Figure 3a. The variables f_1 and f_2 correspond to the `flip` statements on lines 1 and 2 respectively.

```
z ~flip₁(0.5);
if(z) {
  x ~flip₂(0.6);
  y ~flip₃(0.7)
} else {
  x ~flip₄(0.4);
  y := x
}
```

(c) A probabilistic program illustrating the context-specific independence between x and y given $z = \mathsf{T}$.



(d) A BDD representing the logical formula compiled from the program in Figure 3c. The variables f_1, f_2, f_3, and f_4 correspond to the annotated `flip` statements.

Figure 3. Example DIPPL programs and the BDDs each of them compile to. This compilation assumes that the initial state is the true BDD.

Then, we say that X is *independent* of Y, written $X \perp\!\!\!\perp Y$, if $\Pr(X, Y) = \Pr(X) \times \Pr(Y)$. In this case, we say that this distribution *factorizes* over the variables X and Y.

Figure 3a shows a probabilistic program with two independent random variables x and y. The corresponding BDD generated in Figure 3b exploits the independence between the variables x and y. In particular, we see that node f_2 does not depend on the particular value of x. Thus, the BDD *factorizes* the distribution over x and y. As a consequence, the size of the BDD grows linearly with the number of independent random variables.

Conditional independence The next form of independence we consider is *conditional independence*:

Definition 5.2 (Conditional independence)**.** Let $\Pr(X, Y, Z)$ be a joint probability distribution over sets of random variables X, Y, and Z. Then, we say X is independent of Z *given* Y, written $X \perp\!\!\!\perp Z \mid Y$, if $\Pr(X, Z \mid Y) = \Pr(X \mid Y) \times \Pr(Z \mid Y)$.

Figure 1 gave an example probabilistic program that exhibits conditional independence. In this program, the variables x and z are correlated unless y is fixed to a particular value: thus, x and z are conditionally independent given y. Figure 1b shows how this conditional independence is exploited by the BDD; thus, Markov chains have BDD representations that are linear in size to the length of chain.

Conditional independence is exploited by specialized inference algorithms for Bayesian networks like the join-tree algorithm [23]. However, conditional independence is not exploited by path-based – or enumerative – probabilistic program inference procedures, such as the method utilized by Psi [21].

Context-specific independence The final form of independence we will discuss is context-specific independence. Context-specific independence is a weakening of conditional independence that occurs when two sets of random variables are independent only when a third set of variables all take on a particular *value* [4]:

Definition 5.3 (Context-specific independence)**.** Consider a joint probability distribution $\Pr(X, Y, Z)$ over sets of random variables X, Y, and Z, and let c be an assignment to variables in Z. Then, we say X is contextually independent of Y *given* $Z = c$, written $X \perp\!\!\!\perp Y \mid Z = c$, if $\Pr(X, Y \mid Z = c) = \Pr(X \mid Z = c) \times \Pr(Y \mid Z = c)$.

An example program that exhibits context-specific independence is show in Figure 3c. The variables x and y are correlated if $z = \mathsf{F}$ or if z is unknown, but they are independent if $z = \mathsf{T}$. Thus, x is independent of y given $z = \mathsf{T}$.

The equivalent BDD generated in Figure 3d exploits the conditional independence of x and y given $z = \mathsf{T}$ by first branching on the value of z, and then representing the configurations of x and y as two sub-functions. Note here that the variable order of the BDD is relevant. The BDD generated in Figure 3d exploits the context-specific independence of x and y given $z = \mathsf{T}$ by representing x and y in a factorized manner when $z = \mathsf{T}$. Note how the sub-function when $z = \mathsf{T}$ is isomorphic to Figure 3b.

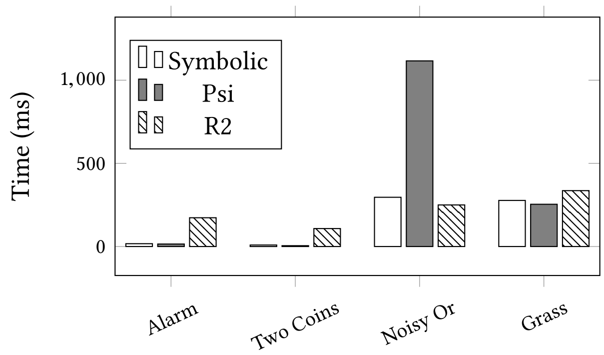


Figure 4. Performance comparison for discrete inference between Psi [21], R2 [28], and symbolic inference. The y-axis gives the time in milliseconds to perform inference.

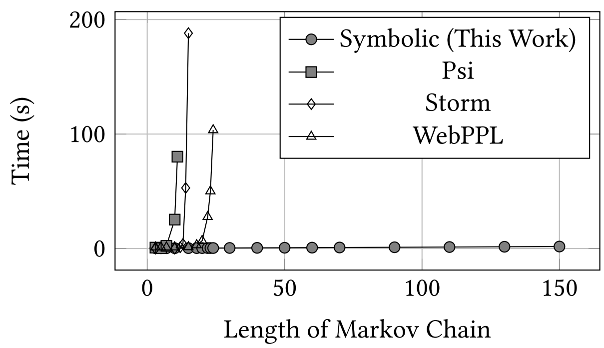


Figure 5. Experimental comparison between techniques for performing exact inference on a Markov chain.

In general, exploiting context-specific independence is challenging and is not directly supported in typical Bayesian network inference algorithms such as the join-tree algorithm. Context-specific independence is often present when there is some amount of *determinism*, and exploiting context-specific independence was one of the original motivations for the development of WMC for Bayesian networks [6, 38]. Probabilistic programs are very often partially deterministic; thus, we believe exploiting context-specific independence is essential for practical efficient inference in this domain. To our knowledge, no existing imperative or functional PPL inference system currently exploits context-specific independence.

6 Implementation & Experiments

In this section we experimentally validate the effectiveness of our symbolic compilation procedure for performing inference on DIPPL programs. We directly implemented the compilation procedure described in Section 4 in Scala. We used the JavaBDD library in order create and manipulate binary decision diagrams [45].

6.1 Experiments

Our goal is to validate that it is a viable technique for performing inference in practice and performs favorably in comparison with existing exact (and approximate) inference techniques.

First, we discuss a collection of simple baseline inference tasks to demonstrate that our symbolic compilation is competitive with Psi [21], R2 [28], and the Storm probabilistic model checker [16]. Then, we elaborate on the motivating example from Section 2 and clearly demonstrate how our symbolic approach can exploit conditional independence to scale to large Markov models. Next, we show how our technique can achieve performance that is competitive with specialized Bayesian network inference techniques. Finally, we demonstrate how our symbolic compilation can exploit context-specific independence to perform inference on a synthetic grids dataset. All experiments were conducted on a 2.3GHz Intel i5 processor with 16GB of RAM.

6.1.1 Baselines

In Figure 4 we compared our technique against Psi [21] and R2 [28] on the collection of all discrete probabilistic programs that they were both evaluated on. Psi[1] is an exact inference compilation technique, so its performance can be directly compared against our performance. R2 is an approximate inference engine and cannot produce exact inference results. The timings reported for R2 are the time it took R2 to produce an approximation that is within 3% of the exact answer[2].

The code for each of the models – Alarm, Two Coins, Noisy Or, and Grass – was extracted from the source code found in the R2 and Psi source code repositories and then translated to DIPPL. These baseline experiments show that our symbolic technique is competitive with existing methods on well-known example models. However, these examples are too small to demonstrate the benefits of symbolic inference: each example is less than 25 lines. In subsequent sections, we will demonstrate the power of symbolic inference by exploiting independence structure in much larger discrete models.

6.1.2 Markov Chain

Section 2 discussed Markov chains and demonstrated that a compact BDD can be compiled that exploits the conditional independence of the network. In particular, a Markov chain of length n can be compiled to a linear-sized BDD in n.

Figure 5 shows how two exact probabilistic programming inference tools compare against our symbolic inference technique for inference on Markov chains. WebPPL [46] and Psi [21] rely on enumerative concrete exact inference, which is

[1] We used Psi version `52b31ba`.
[2] Our performance figures for R2 are excerpted from Gehr et al. [21]. We were not able to run R2 to perform our own experiments due to inability to access the required version of Visual Studio.

Model	Us (s)	BN Time (s) [6]	Size of BDD
Alarm [6]	1.872	0.21	52k
Halfinder	12.652	1.37	157k
Hepar2	7.834	0.28 [11]	139k
pathfinder	62.034	14.94	392k

Table 1. Experimental results for Bayesian networks encoded as probabilistic programs. We report the time it took to perform exact inference in seconds for our method compared against the Bayesian network inference algorithm from Chavira and Darwiche [6], labeled as "BN Time". In addition, we report the final size of our compiled BDD.

exponential in the length of the chain. To compare against Storm, we compiled these models directly into discrete-time Markov chains. As the length of the Markov chain grows, the size of the encoded discrete-time Markov chain grows exponentially. Symbolic inference exploits the conditional independence of each variable in the chain, and is thus linear time in the length of the chain.

6.1.3 Bayesian Network Encodings

In this section we demonstrate the power of our symbolic representation by performing exact inference on Bayesian networks encoded as probabilistic programs. We compared the performance of our symbolic compilation procedure against an exact inference procedure for Bayesian networks [6]. Each of these Bayesian networks is from Chavira and Darwiche [6][3]. Table 1 shows the experimental results: our symbolic approach is competitive with specialized Bayesian network inference.

The goal of these experiments is to benchmark the extent to which one sacrifices efficient inference for a more flexible modeling framework; Ace is at an inherent advantage in this comparison for two main reasons. First, our inference algorithm is compositional, while Ace considers the whole Bayesian network at once. This gives Ace an advantage on this benchmark. Ace compiles Bayesian networks to *d-DNNFs*, which is a family of circuits that are not efficiently composable, but are faster to compile than BDDs [14]. Our technique compiles to BDDs, which are slower to compile than d-DNNFs, but support a compositional line-by-line compilation procedure. Second, Bayesian networks are in some sense a worst-case probabilistic program, since they have no interesting program structure beyond the graph structure that Ace already exploits.

These Bayesian networks are not necessarily Boolean valued: they may contain multi-valued nodes. For instance, the Alarm network has three values that the StrokeVolume variable may take. We encode these multi-valued nodes as

[3]The networks can also be found at http://www.bnlearn.com/bnrepository

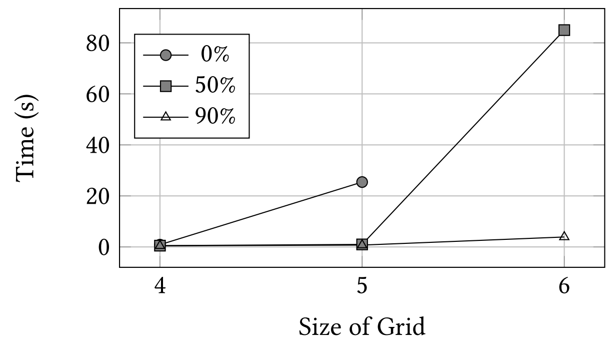
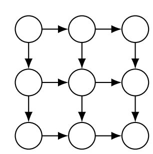


Figure 6. Experiment evaluating the effects of determinism on compiling an encoding of a grid Bayesian network. The $n\%$ result means that there is $n\%$ determinism present. Time was cut off at a max of 300 seconds.

Boolean program variables using a one-hot encoding in a style similar to Sang et al. [38]. The generated DIPPL files are quite large: the pathfinder program has over ten thousand lines of code. Furthermore, neither ProbLog [15, 17] nor Psi could perform inference within 300 seconds on the alarm example, the smallest of the above examples, thus demonstrating the power of our encoding over probabilistic logic programs and enumerative inference on this example.

6.1.4 Grids

This experiment showcases how our method exploits context-specific independence to perform inference more efficiently in the presence of determinism. Grids were originally introduced by Sang et al. [38] to demonstrate the effectiveness of exploiting determinism during Bayesian network inference. A 3-grid is a Boolean-valued Bayesian network arranged in a three by three grid:



For these experiments we encoded grid Bayesian networks into probabilistic programs. Grids are typically hard inference challenges even for specialized Bayesian network inference algorithms. However, in the presence of determinism, the grid inference task can become vastly easier. A grid is $n\%$-deterministic if $n\%$ of the `flips` in the program are replaced with assignments to constants. Figure 6 shows how our symbolic compilation exploits the context-specific independence induced by the determinism of the program in order to perform inference more efficiently.

7 Related Work

First we discuss two closely related individual works on exact inference for PPLs; then we discuss larger categories of related work.

Claret et al. [10] compiles imperative probabilistic progams to algebraic decision diagrams (ADDs) via a form of dataflow analysis [10]. This approach is fundamentally different from our approach, as the ADD cannot represent the distribution in a factorized way. An ADD must contain the probability of each model of the Boolean formula as a leaf node. Thus, it cannot exploit the independence structure required to compactly represent joint probability distributions with independence structure efficiently.

Also closely related is the work of Pfeffer et al. [34], which seeks to decompose the probabilistic program inference task at specific program points where the distribution is known to factorize due to conditional independence. This line of work only considers conditional independence — not context-specific independence — and requires hand-annotated program constructs in order to expose and exploit the independences.

Path-based Program Inference Many techniques for performing inference in current probabilistic programming languages are enumerative or *path-based*: they perform inference by integrating or approximating the probability mass along each path of the probabilistic program [2, 21, 39, 46]. The complexity of inference for path-based inference algorithms scales with the number of paths through the program. The main weakness with these inference strategies is that they cannot exploit common structure across paths – such as independence – and thus scale poorly on examples with many paths.

Probabilistic Logic Programs Most prior work on exact inference for probabilistic programs was developed for probabilistic logic programs [15, 17, 35, 36, 43]. Similar to our work, these techniques compile a probablistic logic program into a weighted Boolean formula and utilize state-of-the-art WMC solvers to compile the WBF into a representation that supports efficient WMC evaluation, such as a binary decision diagram (BDD) [5], sentential decision diagram (SDD) [13], or d-DNNF circuit [14]. Currently, WMC-based inference remains the state-of-the-art inference strategy for probabilistic logic programs. These techniques are not directly applicable to imperative probabilistic programs such as DIPPL due to the presence of sequencing, arbitrary observation, and other imperative programming constructs.

Probabilistic Model Checkers Probabilistic model checkers such as Storm [16] and Prism [24] can be used to perform Bayesian inference on probabilistic systems. These methods work by compiling programs to a representation such as a discrete-time Markov chain or Markov decision process, for which there exist well-known inference strategies. These representations allow probabilistic model checkers to reason about loops and non-termination. In comparison with this work, probabilistic model checkers suffer from a state-space explosion similar to path-based inference methods due to the fact that they devote a node to each possible configuration of variables in the program.

Compilation-based PPLs There exists a large number of PPLs that perform inference by converting the program into a probabilistic graphical model [25, 26, 32, 33], assuming a fixed set of random variables. There are two primary shortcomings of these techniques in relation to ours. The first is that these techniques cannot exploit the context-specific independence present in the program structure, since the topology of the graph – either a Bayesian network or factor graph – does not make this information explicit. Second, these techniques restrict the space of programs to those that can be compiled. Thus they require constraints on the space of programs, such as requiring a statically-determined number of variables, or requiring that loops can be statically unrolled. Currently, we have similar constraints in that our compilation technique cannot handle unbounded loops, that we hope to address in future work.

8 Conclusion & Future Work

In conclusion, we developed a semantics and symbolic compilation procedure for exact inference in a discrete imperative probabilistic programming language called DIPPL. In doing so, we have drawn connections among the probabilistic logic programming, symbolic model checking, and artificial intelligence communities. We theoretically proved our symbolic compilation procedure correct and experimentally validated it against existing probabilistic systems. Finally, we showed that our method is competitive with state-of-the-art Bayesian network inference tasks, showing that our compilation procedures scales to large complex probability models.

We anticipate much future work in this direction. First, we plan to extend our symbolic compilation procedure to handle richer classes of programs. For instance, we would like to support almost-surely terminating loops and procedures, as well as enrich the class of datatypes supported by the language. Second, we would like to quantify precisely the complexity of inference for discrete probabilistic programs. The graphical models community has metrics such as *tree-width* that provide precise notions of the complexity of inference; we believe such notions may exist for probabilistic programs as well [12, 23]. Finally, we anticipate that techniques from the symbolic model checking community – such as Bebop [3] – may be applicable here, and applying these techniques is also promising future work.

Acknowledgments

This work is partially supported by National Science Foundation grants IIS-1657613, IIS-1633857, and CCF-1837129; DARPA XAI grant N66001-17-2-4032, NEC Research, a gift from Intel, and a gift from Facebook Research. The authors would like to thank Joe Qian for assistance with the development of the language semantics and its properties.

References

[1] Sheldon B. Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* C-27 (1978), 509–516.

[2] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 80 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133904

[3] Thomas Ball and Sriram K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN Model Checking and Software Verification*. 113–130.

[4] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. 1996. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 115–123.

[5] R. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE TC* C-35 (1986), 677–691.

[6] Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *J. Artificial Intelligence* 172, 6-7 (April 2008), 772–799. https://doi.org/10.1016/j.artint.2007.11.002

[7] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. 2006. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning* 42, 1-2 (2006), 4–20.

[8] Arthur Choi and Adnan Darwiche. 2011. Relax, Compensate and then Recover. In *New Frontiers in Artificial Intelligence*, Takashi Onada, Daisuke Bekki, and Eric McCready (Eds.). Lecture Notes in Computer Science, Vol. 6797. Springer Berlin / Heidelberg, 167–180.

[9] YooJung Choi, Adnan Darwiche, and Guy Van den Broeck. 2017. Optimal Feature Selection for Decision Robustness in Bayesian Networks. In *IJCAI 2017 Workshop on Logical Foundations for Uncertainty and Machine Learning*. http://starai.cs.ucla.edu/papers/ChoiLFU17.pdf

[10] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (2013), 92. https://doi.org/10.1145/2491411.2491423

[11] Giso H Dal, Alfons W Laarman, and Peter JF Lucas. 2018. Parallel Probabilistic Inference by Weighted Model Counting. In *International Conference on Probabilistic Graphical Models*. 97–108.

[12] A. Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

[13] Adnan Darwiche. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. 819.

[14] A. Darwiche and P. Marquis. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.

[15] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.. In *Proceedings of IJCAI*, Vol. 7. 2462–2467.

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

[17] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2013. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *J. Theory and Practice of Logic Programming* 15(3) (2013), 358 – 401.

[18] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. 2011. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of UAI*. 211–220.

[19] Tal Friedman and Guy Van den Broeck. 2018. Approximate Knowledge Compilation by Online Collapsed Importance Sampling. In *Advances in Neural Information Processing Systems 31 (NIPS)*.

[20] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

[21] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. *Proc. of ESOP/ETAPS* 9779 (2016), 62–83.

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

[23] D. Koller and N. Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[24] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 585–591.

[25] a McCallum, K Schultz, and S Singh. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. *Proc. of NIPS* 22 (2009), 1249–1257.

[26] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. (2014). Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[27] C. Muise, S. McIlraith, J.C. Beck, and E. Hsu. 2010. Fast d-DNNF Compilation with sharpSAT. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*.

[28] Aditya V Nori, Sriram K Rajamani, and Selva Samuel. 2014. R2 : An Efficient MCMC Sampler for Probabilistic Programs. *Aaai* (2014), 2476–2482.

[29] Umut Oztok and Adnan Darwiche. 2014. On Compiling CNF into Decision-DNNF. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP)*. 42–57.

[30] U. Oztok and A. Darwiche. 2015. A Top-Down Compiler for Sentential Decision Diagrams. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*.

[31] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

[32] A. Pfeffer. 2001. IBAL: A probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence*, Vol. 17. 733–740.

[33] Avi Pfeffer. 2009. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report* 137 (2009).

[34] Avi Pfeffer, Brian Ruttenberg, William Kretschmer, and Alison OConnor. 2018. Structured Factored Inference for Probabilistic Programming. In *International Conference on Artificial Intelligence and Statistics*. 1224–1232.

[35] Joris Renkens, Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. 2014. Explanation-based approximate weighted model counting for probabilistic logics. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence, AAAI*. http://starai.cs.ucla.edu/papers/RenkensAAAI14.pdf

[36] Fabrizio Riguzzi and Terrance Swift. 2011. The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty. *Theory and Practice of Logic Programming* 11, 4–5 (2011), 433–449.

[37] Dan Roth. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82, 1 (1996), 273–302.

[38] Tian Sang, Paul Beame, and Henry A Kautz. 2005. Performing Bayesian inference by weighted model counting. In *AAAI*, Vol. 5. 475–481.

[39] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. *SIGPLAN Not.* 48, 6 (June 2013), 447–458. https://doi.org/10.1145/2499370.2462179

[40] David Sontag, Amir Globerson, and Tommi Jaakkola. 2011. Introduction to dual composition for inference. In *Optimization for Machine Learning*. MIT Press.

[41] Guy Van den Broeck and Dan Suciu. 2017. *Query Processing on Probabilistic Data: A Survey*. Now Publishers. https://doi.org/10.1561/

1900000052

[42] Jonas Vlasselaer, Wannes Meert, Guy Van den Broeck, and Luc De Raedt. 2016. Exploiting Local and Repeated Structure in Dynamic Bayesian Networks. *Artificial Intelligence* 232 (March 2016), 43 – 53. https://doi.org/10.1016/j.artint.2015.12.001

[43] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. 2015. Anytime inference in probabilistic logic programs with Tp-compilation. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*.

[44] Martin J Wainwright, Michael I Jordan, et al. 2008. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning* 1, 1–2 (2008), 1–305.

[45] John Whaley. 2007. JavaBDD. http://javabdd.sourceforge.net. (2007).

[46] David Wingate and Theophane Weber. 2013. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299* (2013).

$$[\![\texttt{skip}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma \\ 0 & \text{otherwise} \end{cases}$$

$$[\![x \sim \texttt{flip}(\theta)]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} \theta & \text{if } \sigma' = \sigma[x \mapsto \mathsf{T}] \\ 1-\theta & \text{if } \sigma' = \sigma[x \mapsto \mathsf{F}] \\ 0 & \text{otherwise} \end{cases}$$

$$[\![x := \mathsf{e}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma[x \mapsto [\![\mathsf{e}]\!](\sigma)] \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{observe}(\mathsf{e})]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} 1 & \text{if } \sigma' = \sigma \text{ and } [\![\mathsf{e}]\!](\sigma) = \mathsf{T} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\mathsf{s}_1; \mathsf{s}_2]\!]_T(\sigma' \mid \sigma) \triangleq \frac{\sum_{\tau \in \Sigma} [\![\mathsf{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathsf{s}_2]\!]_T(\sigma' \mid \tau) \times [\![\mathsf{s}_2]\!]_A(\tau)}{\sum_{\tau \in \Sigma} [\![\mathsf{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathsf{s}_2]\!]_A(\tau)}$$

$$[\![\texttt{if e } \{\mathsf{s}_1\} \texttt{ else } \{\mathsf{s}_2\}]\!]_T(\sigma' \mid \sigma) \triangleq \begin{cases} [\![\mathsf{s}_1]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathsf{e}]\!](\sigma) = \mathsf{T} \\ [\![\mathsf{s}_2]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathsf{e}]\!](\sigma) = \mathsf{F} \end{cases}$$

(a) Transition semantics of DIPPL. $[\![\mathsf{s}]\!]_T(\sigma' \mid \sigma)$ gives the conditional probability upon executing s of transitioning to state σ' *given* that the start state is σ and no observe statements in s are violated. If every execution path violates an observation, $[\![\mathsf{s}]\!]_T(\sigma' \mid \sigma) = \bot$.

$$[\![\texttt{skip}(\mathsf{e})]\!]_A(\sigma) \triangleq 1$$

$$[\![x \sim \texttt{flip}(\theta)]\!]_A(\sigma) \triangleq 1$$

$$[\![x := \mathsf{e}]\!]_A(\sigma) \triangleq 1$$

$$[\![\texttt{observe}(\mathsf{e})]\!]_A(\sigma) \triangleq \begin{cases} 1 & \text{if } [\![\mathsf{e}]\!](\sigma) = \mathsf{T} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\mathsf{s}_1; \mathsf{s}_2]\!]_A(\sigma) \triangleq [\![\mathsf{s}_1]\!]_A(\sigma) \times \sum_{\tau \in \Sigma} \left([\![\mathsf{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathsf{s}_2]\!]_A(\tau)\right)$$

$$[\![\texttt{if e } \{\mathsf{s}_1\} \texttt{ else } \{\mathsf{s}_2\}]\!]_A(\sigma) \triangleq \begin{cases} [\![\mathsf{s}_1]\!]_A(\sigma) & \text{if } [\![\mathsf{e}]\!](\sigma) = \mathsf{T} \\ [\![\mathsf{s}_2]\!]_A(\sigma) & \text{otherwise} \end{cases}$$

(b) Accepting semantics of DIPPL. $[\![\mathsf{s}]\!]_A(\sigma)$ gives the probability that no observations are violated by executing s beginning in state σ.

Figure 7. Semantics of DIPPL.

A Semantics of DIPPL

The goal of the semantics of any probabilistic programming language is to define the distribution over which one wishes to perform inference. In this section, we introduce a denotational semantics that directly produces this distribution of interest, and it is defined over program states. A state σ is a finite map from variables to Boolean values, and Σ is the set of all possible states. We will be interested in probability distributions on Σ, defined formally as follows:

Definition A.1 (Discrete probability distribution)**.** Let Ω be a set called the *sample space.* Then, a *discrete probability distribution* on Ω is a function $\Pr : 2^{\omega} \to [0, 1]$ such that (1) $\Pr(\Omega) = 1$; (2) for any $\omega \in \Omega$, $\Pr(\omega) \geq 0$; (3) for any countable set of disjoint elements $\{A_i\}$ where $A_i \subseteq 2^{\Omega}$, we have that $\Pr(\bigcup_i \{A_i\}) = \sum_i \Pr(A_i)$.

We denote the set of all possible discrete probability distributions with Σ as the sample space as $\texttt{Dist}\ \Sigma$. We add a special element to $\texttt{Dist}\ \Sigma$, denoted $\bot$, which is the function that assigns a probability of zero to all states (this will be necessary to represent situations where an `observe`d expression is false).

We define a denotational semantics for DIPPL, which we call its *transition semantics* and denote $[\![\cdot]\!]_T$. These semantics are summarized in Figure 7a. The transition semantics will be the primary semantic object of interest for DIPPL, and will directly produce the distribution over which we wish to perform inference. For some statement `s`, the transition semantics is written $[\![\texttt{s}]\!]_T(\sigma' \mid \sigma)$, and it computes the conditional probability upon executing `s` of transitioning to state σ' *given* that the start state is σ and no `observe` statements are violated. The transition semantics have the following type signature:

$$[\![\texttt{s}]\!]_T : \Sigma \to \texttt{Dist}\ \Sigma$$

The transition semantics of DIPPL is shown in Figure 7a. The semantics of `skip`, assignment, and conditionals are straightforward. The semantics of sampling from a Bernoulli distribution is analogous to that for assignment, except that there are two possible output states depending on the value that was sampled. An `observe` statement has no effect if the associated expression is true in σ; otherwise the semantics has the effect of mapping σ to the special $\bot$ distribution.

The Role of Observe in Sequencing The transition semantics of DIPPL require that each statement be interpreted as a conditional probability. Ideally, we would like this conditional probability to be sufficient to describe the semantics of compositions. Perhaps surprisingly, the conditional probability distribution of transitioning from one state to another alone is insufficient for capturing the behavior of compositions in the presence of observations. We will illustrate this principle with an example. Consider the following two DIPPL statements:

$$bar_1 = \left\{ \begin{array}{l} \texttt{if}(x)\ \{\ y \sim \texttt{flip(1/4)}\ \} \\ \texttt{else}\ \{\ y \sim \texttt{flip(1/2)}\ \} \end{array} \right\},$$

$$bar_2 = \left\{ \begin{array}{l} y \sim \texttt{flip(1/2)}; \\ \texttt{observe}(x \lor y); \\ \texttt{if}(y)\ \{\ y \sim \texttt{flip(1/2)}\ \} \\ \texttt{else}\ \{\ y\ \texttt{:=}\ \texttt{F}\ \} \end{array} \right\}.$$

Both statements represent exactly the same *conditional* probability distribution from input to output states:

$$\begin{aligned} [\![bar_1]\!]_T(\sigma' \mid \sigma) &= [\![bar_2]\!]_T(\sigma' \mid \sigma) \\ &= \begin{cases} 1/2 & \text{if } x[\sigma] = x[\sigma'] = \texttt{F}, \\ 1/4 & \text{if } x[\sigma] = x[\sigma'] = \texttt{T} \text{ and } y[\sigma'] = \texttt{T}, \\ 3/4 & \text{if } x[\sigma] = x[\sigma'] = \texttt{T} \text{ and } y[\sigma'] = \texttt{F}, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

This is easy to see for bar_1, which encodes these probabilities directly. For bar_2, intuitively, when $y = \texttt{T}$ in the output, both `flip` statements must return `T`, which happens with probability $1/4$. When $x = \texttt{F}$ in the input, bar_2 uses an `observe` statement to disallow executions where the first `flip` returned `F`. Given this observation, the *then* branch is always taken, so output $y = \texttt{T}$ has probability $1/2$.

Because the purpose of probabilistic programming is often to represent a conditional probability distribution, one is easily fooled into believing that these programs are equivalent. This is not the case: bar_1 and bar_2 behave differently when sequenced with other statements. For example, consider the sequences $(foo; bar_1)$ and $(foo; bar_2)$ where

$$foo = \left\{\ x \sim \texttt{flip(1/3)}\ \right\}.$$

Let σ'_{ex} be an output state where $x = \texttt{F}, y = \texttt{T}$, and let σ_{ex} be an arbitrary state. The first sequence's transition semantics behave naturally for this output state:

$$[\![foo; bar_1]\!]_T(\sigma'_{ex} \mid \sigma_{ex}) = 2/3 \cdot 1/2 = 1/3 \tag{4}$$

However, $(foo; bar_2)$ represents a different distribution: the `observe` statement in bar_2 will disallow half of the execution paths where foo set $x = \texttt{T}$. After the `observe` statement is executed in bar_2, $\Pr(x = \texttt{T}) = \frac{1}{2}$: the observation has increased the probability of x being true in foo, which was $1/3$. Thus, it is clear foo and bar_2 cannot be reasoned about solely as conditional probability distributions: `observe` statements in bar_2 affect the conditional probability of foo. Thus, the semantics of sequencing requires information beyond solely the conditional probability of each of the sub-statements, as we discuss next.

$$\frac{}{\texttt{skip} \rightsquigarrow (\gamma(V), \delta(V))}$$

$$\frac{\text{fresh} f}{\begin{array}{c} x \sim \texttt{flip}(\theta) \rightsquigarrow \\ \big((x' \Leftrightarrow f) \wedge \gamma(V \setminus \{x\}), \delta(V)[f \mapsto \theta, \neg f \mapsto 1 - \theta]\big) \end{array}}$$

$$\frac{}{x := \texttt{e} \rightsquigarrow ((x' \Leftrightarrow \texttt{e}) \wedge \gamma(V \setminus \{x\}), \delta(V))}$$

$$\frac{\begin{array}{c} s_1 \rightsquigarrow (\varphi_1, w_1) \qquad s_2 \rightsquigarrow (\varphi_2, w_2) \\ \varphi_2' = \varphi_2[x_i \mapsto x_i', x_i' \mapsto x_i''] \end{array}}{\texttt{s}_1; \texttt{s}_2 \rightsquigarrow ((\exists x_i'.\varphi_1 \wedge \varphi_2')[x_i'' \mapsto x_i'], w_1 \uplus w_2)}$$

$$\frac{}{\texttt{observe(e)} \rightsquigarrow (\texttt{e} \wedge \gamma(V), \delta(V))}$$

$$\frac{s_1 \rightsquigarrow (\varphi_1, w_1) \qquad s_2 \rightsquigarrow (\varphi_2, w_2)}{\texttt{if e } \{\texttt{s}_1\} \texttt{ else } \{\texttt{s}_2\} \rightsquigarrow ((\texttt{e} \wedge \varphi_1) \vee (\neg\texttt{e} \wedge \varphi_2), w_1 \uplus w_2)}$$

Figure 8. Symbolic compilation rules.

Sequencing Semantics The most interesting case in the semantics is sequencing. We compute the transition semantics of sequencing $[\![\texttt{s}_1; \texttt{s}_2]\!]_T(\sigma' \mid \sigma)$ using the rules of probability. To do this we require the ability to compute the probability that a particular statement will not violate an observation when beginning in state σ. Thus we introduce a helper relation that we call the *accepting semantics* (denoted $[\![\texttt{s}]\!]_A$), which provides the probability that a given statement will be *accepted* (i.e., that no `observe`s will fail) when executed from a given initial state:

$$[\![\texttt{s}]\!]_A : \Sigma \to [0, 1]$$

The accepting semantics is defined in Figure 7b. The first three rules in the figure are trivial. An `observe` statement accepts with probability 1 if the associated expression is true in the given state, and otherwise with probability 0. A sequence of two statements accepts if both statements accept, so the rule simply calculates that probability by summing over all possible intermediate states. Last, the accepting probability of an `if` statement in state σ is simply the accepting probability of whichever branch will be taken from that state.

Now we can use the accepting semantics to give the transition semantics for sequencing. First, we can compute the probability of both transitioning from some initial state σ to some final state σ' *and* the fact that no observations are violated in $\texttt{s}_1$ or $\texttt{s}_2$:

$$\alpha = \sum_{\tau \in \Sigma} [\![\texttt{s}_1]\!]_T(\tau \mid \sigma) \times [\![\texttt{s}_2]\!]_T(\sigma' \mid \tau) \times [\![\texttt{s}_1]\!]_A(\sigma) \times [\![\texttt{s}_2]\!]_A(\tau).$$

In order to obtain the distribution of transitioning between states σ and σ' *given* that no observations are violated, we must re-normalize this distribution by the probability that no observations are violated:

$$\beta = [\![\texttt{s}_1]\!]_A(\sigma) \times \sum_{\tau \in \Sigma} [\![\texttt{s}_1]\!]_T(\tau \mid \sigma) \times [\![\texttt{s}_2]\!](\tau).$$

Thus, our conditional probability is $[\![\texttt{s}_1; \texttt{s}_2]\!]_T(\sigma' \mid \sigma) = \frac{\alpha}{\beta}$. For completeness, we define the $0/0$ case to be equal to 0.

B Symbolic Compilation

In this section we formally define our symbolic compilation of a DIPPL program to a weighted Boolean formula, denoted $\texttt{s} \rightsquigarrow (\varphi, w)$. Intuitively, the formula φ produced by the compilation represents the program $\texttt{s}$ as a relation between initial states and final states, where initial states are represented by unprimed Boolean variables $\{x_i\}$ and final states are represented by primed Boolean variables $\{x_i'\}$. This is similar to a standard encoding for model checking Boolean program, except we include auxiliary variables in the encoding which are neither initial nor final state variables [3]. These compiled weighted Boolean formula will have a probabilistic semantics which allow them to be interpreted as either an accepting or transition probability for the original statement.

Our goal is to ultimately give a correspondence between the compiled weighted Boolean formula and the original denotational semantics of the statement. First we define the translation of a state σ to a logical formula:

Definition B.1 (Boolean state)**.** Let $\sigma = \{(x_1, b_1), \ldots, (x_n, b_n)\}$. We define the *Boolean state* $\mathbf{F}(\sigma)$ as $l_1 \wedge \ldots \wedge l_n$ where for each i, l_i is x_i if $\sigma(x_i) = \mathtt{T}$ and $\neg x_i$ if $\sigma(x_i) = \mathtt{F}$. For convenience, we also define a version that relabels state variables to their primed versions, $\mathbf{F}'(\sigma) \triangleq \mathbf{F}(\sigma)[x_i \mapsto x_i']$.

Now, we formally describe how every compiled weighted Boolean formula can be interpreted as a conditional probability by computing the appropriate weighted model count:

Definition B.2 (Transition and accepting semantics)**.** Let (φ, w) be a weighted Boolean formula, and let σ and σ' be states. Then, the *transition semantics* of (φ, w) is defined:

$$[\![(\varphi, w)]\!]_T(\sigma' \mid \sigma) \triangleq \frac{\mathtt{WMC}(\varphi \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w)}{\mathtt{WMC}(\varphi \wedge \mathbf{F}(\sigma), w)} \tag{5}$$

In addition the *accepting semantics* of (φ, w) is defined:

$$[\![(\varphi, w)]\!]_A(\sigma) \triangleq \mathtt{WMC}(\varphi \wedge \mathbf{F}(\sigma), w).$$

Moreover, the transition semantics of Definition 4.4 allow for more general queries to be phrased as WMC tasks as well. For example, the probability of some event α being true in the output state σ' can be computed by replacing $\mathbf{F}'(\sigma')$ in Equation 2 by a Boolean formula for α.

Finally, we state our correctness theorem, which describes the relation between the accepting and transition semantics of the compiled WBF to the denotational semantics of DIPPL:

Theorem B.3 (Correctness of Compilation Procedure)**.** *Let* $\mathtt{s}$ *be a* DIPPL *program,* V *be the set of all variables in* $\mathtt{s}$*, and* $\mathtt{s} \rightsquigarrow (\varphi, w)$*. Then for all states* σ *and* σ' *over the variables in* V*, we have:*

$$[\![\mathtt{s}]\!]_T(\sigma' \mid \sigma) = [\![(\varphi, w)]\!]_T(\sigma' \mid \sigma) \tag{6}$$

and

$$[\![\mathtt{s}]\!]_A(\sigma) = [\![(\varphi, w)]\!]_A(\sigma). \tag{7}$$

Proof. A complete proof can be found in Appendix D. □

Theorem 4.5 allows us to perform inference via weighted model counting on the compiled WBF for a DIPPL program. Next we give a description of the symbolic compilation rules that satisfy this theorem.

C Symbolic Compilation Rules

In this section we describe the symbolic compilation which satisfy Theorem 4.5 for each DIPPL statement. The rules for symbolic compilation are defined in Figure 8. They rely on several conventions. We denote by V the set of all variables in the entire program being compiled. If $V = \{x_1, \ldots, x_n\}$ then we use $\gamma(V)$ to denote the formula $(x_1 \Leftrightarrow x_1') \wedge \ldots \wedge (x_n \Leftrightarrow x_n')$, and we use $\delta(V)$ to denote the weight function that maps each literal over $\{x_1, x_1', \ldots, x_n, x_n'\}$ to 1.

The WBF for `skip` requires that the input and output states are equal and provides a weight of 1 to each literal. The WBF for an assignment $x := \mathtt{e}$ requires that x' be logically equivalent to `e` and all other variables' values are unchanged. Note that `e` is already a Boolean formula by the syntax of DIPPL so expressions simply compile to themselves. The WBF for drawing a sample from a Bernoulli distribution, $x \sim \mathtt{flip}(\theta)$, is similar to that for an assignment, except that we introduce a (globally) fresh variable f to represent the sample and weight its true and false literals respectively with the probability of drawing the corresponding value.

The WBE for an `observe` statement requires the corresponding expression to be true and that the state remains unchanged. The WBE for an `if` statement compiles the two branches to formulas and then uses the standard logical semantics of conditionals. The weight function $w_1 \uplus w_2$ is a *shadowing union* of the two functions, favoring w_2. However, by construction whenever two weight functions created by the rules have the same literal in their domain, the corresponding weights are equal. Finally, the WBE for a sequence composes the WBEs for the two sub-statements via a combination of variable renaming and existential quantification.

In the following section, we delineate the advantages of utilizing WMC for inference, and describe how WMC exploits program structure in order to perform inference efficiently.

D Proof of Theorem 4.5

D.1 Properties of WMC

We begin with some important lemmas about weighted model counting:

Lemma D.1 (Independent Conjunction). *Let α and β be Boolean sentences which share no variables. 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:

$$
\begin{aligned}
\mathtt{WMC}(\alpha \wedge \beta, w) &= \sum_{\omega \in \mathsf{Mod}(\alpha \wedge \beta)} \prod_{l \in \omega} w(l) \\
&= \sum_{\omega_\alpha \in \mathsf{Mod}(\alpha)} \sum_{\omega_\beta \in \mathsf{Mod}(\beta)} \prod_{a \in \omega_\alpha} w(a) \times \prod_{b \in \omega_\beta} w(b) \\
&= \left[\sum_{\omega_\alpha \in \mathsf{Mod}(\alpha)} \prod_{a \in \omega_\alpha} w(a) \right] \times \left[\sum_{\omega_\beta \in \mathsf{Mod}(\beta)} \prod_{b \in \omega_\beta} w(b) \right] \\
&= \mathtt{WMC}(\alpha, w) \times \mathtt{WMC}(\beta, w).
\end{aligned}
$$

□

Lemma D.2. *Let α be a Boolean sentence and x be a conjunction of literals. For any weight function w,* $\mathtt{WMC}(\alpha, w) = \mathtt{WMC}(\alpha \mid x, w) \times \mathtt{WMC}(x, w)$. [4]

Proof. Follows from Lemma D.1 and the fact that $\alpha \mid x$ and x share no variables by definition:

$$
\begin{aligned}
\mathtt{WMC}(\alpha \mid x) \times \mathtt{WMC}(x, w) &= \mathtt{WMC}((\alpha \mid x) \wedge x, w) && \text{By Lemma D.1} \\
&= \mathtt{WMC}(\alpha, w).
\end{aligned}
$$

□

Lemma D.3. *Let α be a sentence, x be a conjunction of literals, and w be some weight function. If for all $l \in x$ we have that $w(l) = 1$, then* $\mathtt{WMC}(\alpha \mid x, w) = \mathtt{WMC}(\alpha \wedge x, w)$.

Proof.

$$
\begin{aligned}
\mathtt{WMC}(\alpha \wedge x) &= \mathtt{WMC}((\alpha \wedge x) \mid x, w) \times \underbrace{\mathtt{WMC}(x, w)}_{=1} \\
&= \mathtt{WMC}(\alpha \mid x, w).
\end{aligned}
$$

□

Lemma D.4 (Mutually Exclusive Disjunction). *Let α and β be Boolean be mutually exclusive Boolean sentences (i.e., $\alpha \Leftrightarrow \neg\beta$). Then, for any weight function w,* $\mathtt{WMC}(\alpha \vee \beta, w) = \mathtt{WMC}(\alpha, w) + \mathtt{WMC}(\beta, w)$.

Proof. The proof relies on the fact that, if two sentences α and β are mutually exclusive, then any model ω of $\alpha \vee \beta$ either entails α or entails β. We denote the set of models which entail α as Ω_α, and the set of models which entail β as Ω_β. Then,

$$
\begin{aligned}
\mathtt{WMC}(\alpha \vee \beta, w) &= \sum_{\omega_\alpha \in \Omega_\alpha} \sum_{\omega_\beta \in \Omega_\beta} \prod_{l \in \omega_\alpha} w(l) \prod_{l \in \omega_\beta} w(l) \\
&= \mathtt{WMC}(\alpha, w) + \mathtt{WMC}(\beta, w).
\end{aligned}
$$

□

The following notion of functional dependency will be necessary for reasoning about the compilation of the composition:

Definition D.5 (Functionally dependent WBF). Let (α, w) be a WBF, and let $\mathbf{X}$ and $\mathbf{Y}$ be two variable sets which partition the variables in α. Then we say that $\mathbf{X}$ is *functionally dependent on* $\mathbf{Y}$ *for* α if for any total assignment to variables in $\mathbf{Y}$, labeled y, there is at most one total assignment to variables in $\mathbf{X}$, labeled x, such that $x \wedge y \models \alpha$.

[4] The notation "$\alpha \mid x$" means condition α on x.

Lemma D.6 (Functionally Dependent Existential Quantification). *Let (α, w) be a WBF with variable partition* $\mathbf{X}$ *and* $\mathbf{Y}$ *such that* $\mathbf{X}$ *is functionally dependent on* $\mathbf{Y}$ *for* α. *Furthermore, assume that for any conjunction of literals* x *formed from* $\mathbf{X}$, $\texttt{WMC}(x) = 1$. *Then,* $\texttt{WMC}(\alpha) = \texttt{WMC}(\exists\{x_i \in \mathbf{X}\}.\alpha)$.

Proof. The proof follows from Lemma D.3 and Lemma D.4. First, let $\mathbf{X} = x$ be a single variable, and assume all weighted model counts are performed with the weight function w. Then,

$$\begin{aligned}
\texttt{WMC}(\exists x.\alpha) =& \texttt{WMC}((\alpha \mid x) \vee (\alpha \mid \neg x)) \\
=& \texttt{WMC}(\alpha \mid x) + \texttt{WMC}(\alpha \mid \neg x) && \text{By mutual exclusion} \\
=& \underbrace{\frac{1}{\texttt{WMC}(x)}}_{=1} \texttt{WMC}(\alpha \wedge x) + \underbrace{\frac{1}{\texttt{WMC}(\neg x)}}_{=1} \texttt{WMC}(\alpha \wedge \neg x) \\
=& \texttt{WMC}(\alpha \wedge x) + \texttt{WMC}(\alpha \wedge \neg x) \\
=& \texttt{WMC}((\alpha \wedge x) \vee (\alpha \wedge \neg x)) \\
=& \texttt{WMC}(\alpha) && \text{By mutual exclusion}
\end{aligned}$$

This technique easily generalizes to when $\mathbf{X}$ is a set of variables instead of a single variable. ◻

D.2 Main Proof

Let σ and σ' be an input and output state, let V be the set of variables in the entire program. The proof will proceed by induction on terms. We prove the following inductive base cases for terms which are not defined inductively.

D.2.1 Base Cases

Skip First, we show that the accepting semantics correspond. For any σ, we have that $[\![\texttt{skip}]\!]_A(\sigma) = \texttt{WMC}(\gamma(V) \wedge \mathbf{F}(\sigma)) = 1$, since there is only a single satisfying assignment, which has weight 1. Now, we show that the transition semantics correspond:

- Assume $\sigma' = \sigma$. Then,

$$[\![\texttt{s}]\!]_T(\sigma' \mid \sigma) = \frac{\texttt{WMC}(\gamma(V) \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), \delta(V))}{\texttt{WMC}(\gamma(V) \wedge \mathbf{F}(\sigma))} = 1$$

  since we have a single model in both numerator and denominator, both having weight 1.
- Assume $\sigma \neq \sigma'$. Then:

$$[\![\texttt{s}]\!]_T(\sigma' \mid \sigma) = \frac{\texttt{WMC}(\gamma(V) \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), \delta(V))}{\texttt{WMC}(\gamma(V) \wedge \mathbf{F}(\sigma))} = 0$$

  since the numerator counts models of an unsatisfiable sentence.

Sample Let φ and w be defined as in the symbolic compilation rules. First we show that the accepting semantics correspond.

$$\begin{aligned}
[\![x \sim \texttt{flip}(\theta)]\!]_A(\sigma) =& \texttt{WMC}((x' \Leftrightarrow f) \wedge \gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma), w) \\
=& \underbrace{\texttt{WMC}(x' \Leftrightarrow f \mid \gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma), w)}_{=\theta+(1-\theta)=1} \times \underbrace{\texttt{WMC}(\gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma), w)}_{=1,\text{ by def. of }\delta} && \text{By Lemma D.3} \\
=& 1
\end{aligned}$$

Now we show that the transition semantics correspond:

$$\begin{aligned}
[\![x \sim \texttt{flip}(\theta)]\!]_T(\sigma' \mid \sigma) =& \texttt{WMC}(\varphi \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w) \times \underbrace{\frac{1}{\texttt{WMC}(\varphi \wedge \mathbf{F}(\sigma))}}_{=1} \\
=& \texttt{WMC}(\underbrace{\varphi \mid \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma')}_{\alpha}, w) \times \underbrace{\texttt{WMC}(\mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w)}_{=1}
\end{aligned}$$

We can observe the following about α:

- If $\sigma = \sigma'[x \mapsto \texttt{T}]$, then $\texttt{WMC}(\varphi \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w) = \theta$.
- If $\sigma = \sigma'[x \mapsto \texttt{F}]$, then $\texttt{WMC}(\varphi \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'), w) = 1 - \theta$.
- If $\sigma \neq \sigma'[x \mapsto \texttt{F}]$ and $\sigma \neq \sigma'[x \mapsto \texttt{T}]$, $\alpha = \texttt{F}$, so the weighted model count is 0.

Assignment First we show that the accepting semantics correspond:

$$\begin{aligned}[\![x := \mathtt{e}]\!]_A(\sigma) =& \mathtt{WMC}((x' \Leftrightarrow [\![\mathtt{e}]\!]_{\mathcal{S}}) \wedge \gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma), w)\\ =& 1,\end{aligned}$$

since there is exactly a single model and its weight is 1. Now, we show that the transition semantics correspond:

$$[\![x := \mathtt{e}]\!]_T(\sigma' \mid \sigma) = \mathtt{WMC}(\underbrace{(x' \Leftrightarrow [\![\mathtt{e}]\!]_{\mathcal{S}}) \wedge \gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}(\sigma')}_{\alpha}, w) \times \underbrace{\frac{1}{\mathtt{WMC}((x' \Leftrightarrow [\![\mathtt{e}]\!]_{\mathcal{S}}) \wedge \gamma(V \setminus \{x\}) \wedge \mathbf{F}(\sigma))}}_{=1}$$

- Assume $\sigma = \sigma'[x \mapsto [\![\mathtt{e}]\!](\sigma)]$. Then, α has a single model, and the weight of that model is 1, so $\mathtt{WMC}(\alpha, w) = 1$.
- Assume $\sigma \neq \sigma'[x \mapsto [\![\mathtt{e}]\!](\sigma)]$. Then, α is unsatisfiable, so $\mathtt{WMC}(\alpha, w) = 0$.

Observe First we prove the transition semantics correspond:

$$\begin{aligned}[\![\mathtt{observe(e)}]\!]_A(\sigma) =& \mathtt{WMC}([\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \gamma(V) \wedge \mathbf{F}(\sigma), w)\\ =& \begin{cases} 1 & \text{if } \mathbf{F}(\sigma) \models [\![\mathtt{s}]\!]_{\mathcal{S}} \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

Now, we can prove that the transition semantics correspond:

$$[\![\mathtt{observe(e)}]\!]_T(\sigma' \mid \sigma) = \mathtt{WMC}(\underbrace{[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \gamma(V) \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma')}_{\alpha}, w) \times \frac{1}{\mathtt{WMC}(\underbrace{[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \gamma(V) \wedge \mathbf{F}(\sigma)}_{\beta}, w)}$$

We treat a fraction $\frac{0}{0}$ as 0. Then, we can apply case analysis:

- Assume $\sigma = \sigma'$ and $[\![\mathtt{e}]\!](\sigma) = \mathtt{T}$. Then, both α and β have a single model with weight 1, so $[\![\mathtt{observe(e)}]\!]_T(\sigma' \mid \sigma) = 1$.
- Assume $\sigma \neq \sigma'$ or $[\![\mathtt{e}]\!](\sigma) \neq \mathtt{T}$. Then, either $\mathbf{F}(\sigma) \wedge [\![\mathtt{e}]\!]_{\mathcal{S}} \models \mathtt{F}$ or $\gamma(V) \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma') \models \mathtt{F}$; in either case, $[\![\mathtt{observe(e)}]\!]_T(\sigma' \mid \sigma) = 0$.

D.2.2 Inductive Step

Now, we utilize the inductive hypothesis to prove the theorem for the inductively-defined terms. Formally, let $\mathtt{s}$ be a DIPPL term, let $\{\mathtt{s}_i\}$ be sub-terms of $\mathtt{s}$. Then, our inductive hypothesis states that for each sub-term $\mathtt{s}_i$ of $\mathtt{s}$, where $\mathtt{s}_i \rightsquigarrow (\varphi, w)$, we have that for any two states σ, σ', $[\![\mathtt{s}_i]\!]_T(\sigma' \mid \sigma) = [\![(\varphi, w)]\!]_T(\sigma') \mid \sigma)$ and $[\![\mathtt{s}_i]\!]_A(\sigma) = [\![(\varphi, w)]\!]_A(\mathbf{F}(\sigma))$. Then, we must show that the theorem holds for $\mathtt{s}$ using this hypothesis.

Remark 1. *For the inductively defined compilation semantics, the weight function $w = w_1 \uplus w_2$ is a unique and well-defined weight function, since the only source of weighted variables is from a `flip` term, which only assigns a weight to fresh variables; thus, there can never be a disagreement between the two weight functions w_1 and w_2 about the weight of a particular variable.*

Composition Let $\varphi, w, \varphi_1, \varphi_2, \varphi_2', w_1$, and w_2 be defined as in the symbolic compilation rules. By the inductive hypothesis, we have that the theorem holds for (φ_1, w_1) and (φ_2, w_2). We observe that the weighted model counts of φ_2 are invariant under relabelings. I.e., for any states $\sigma, \sigma', \sigma''$:

$$\begin{aligned}\mathtt{WMC}(\varphi_2 \wedge \mathbf{F}(\sigma)) =& \mathtt{WMC}(\varphi_2' \wedge \mathbf{F}'(\sigma))\\ \mathtt{WMC}(\varphi_2 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}(\sigma')) =& \mathtt{WMC}(\varphi_2' \wedge \mathbf{F}'(\sigma) \wedge \mathbf{F}''(\sigma'))\end{aligned}$$

where $\mathbf{F}''(\cdot)$ generates double-primed state variables. Now we show that the WBF compilation has the correct accepting semantics, where each weighted model count implicitly utilizes the weight function w:

$$
\begin{aligned}
[\![\mathsf{s}_1; \mathsf{s}_2]\!]_A(\sigma) =& [\![\mathsf{s}_1]\!]_A(\sigma) \times \sum_{\tau \in \Sigma} ([\![\mathsf{s}_1]\!]_T(\tau \mid \sigma) \times [\![\mathsf{s}_2]\!]_A(\tau)) \\
=& Z \times \sum_{\tau} \frac{\mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\tau), w)}{Z} \times \mathsf{WMC}(\varphi_2' \wedge \mathbf{F}'(\tau)) \quad \text{where } Z = \mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma)) \\
=& \sum_{\tau} \mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\tau)) \times \mathsf{WMC}(\varphi_2' \wedge \mathbf{F}'(\tau)) \\
=& \sum_{\tau} \mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma) \mid \mathbf{F}'(\tau)) \times \mathsf{WMC}(\varphi_2' \mid \mathbf{F}'(\tau)) \times \underbrace{[\mathsf{WMC}(\mathbf{F}'(\tau))]^2}_{=1} && \text{(By Lemma D.2)} \\
=& \sum_{\tau} \mathsf{WMC}\left(\left[\varphi_1 \wedge \mathbf{F}(\sigma) \mid \mathbf{F}'(\tau)\right] \wedge \left[\varphi_2' \mid \mathbf{F}'(\tau)\right]\right) && \text{(By Lemma D.1)} \\
=& \sum_{\tau} \mathsf{WMC}\left(\left[\varphi_1 \wedge \mathbf{F}(\sigma) \wedge \varphi_2' \mid \mathbf{F}'(\tau)\right]\right) \\
=& \sum_{\tau} \mathsf{WMC}\left(\varphi_1 \wedge \mathbf{F}(\sigma) \wedge \varphi_2' \wedge \mathbf{F}'(\tau)\right) \times \underbrace{\frac{1}{\mathsf{WMC}(\mathbf{F}'(\tau))}}_{=1} && \text{(By Lemma D.2)} \\
=& \sum_{\tau} \mathsf{WMC}(\varphi_1 \wedge \varphi_2' \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\tau)) \\
=& \mathsf{WMC}\left(\bigvee_{\tau} \varphi_1 \wedge \varphi_2' \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\tau)\right) && \text{(By Lemma D.4)} \\
=& \mathsf{WMC}\left(\varphi_1 \wedge \varphi_2' \wedge \mathbf{F}(\sigma) \wedge \left[\bigvee_{\tau} \mathbf{F}'(\tau)\right]\right) \\
=& \mathsf{WMC}\left(\varphi_1 \wedge \varphi_2' \wedge \mathbf{F}(\sigma)\right) \\
=& \mathsf{WMC}\left(\exists \{x_i'\}.\varphi_1 \wedge \varphi_2' \wedge \mathbf{F}(\sigma)\right) && \text{(By Lemma D.6)} \\
=& \mathsf{WMC}((\exists \{x_i'\}.\varphi_2' \wedge \mathbf{F}(\sigma)[x'' \mapsto x']).
\end{aligned}
$$

Now, we can prove the transition semantics correspond for composition, where all model counts are implicitly utilizing the weight function w:

$$
\begin{aligned}
[\![\mathtt{s}_1;\mathtt{s}_2]\!]_T(\sigma' \mid \sigma) &= \frac{\sum_{\tau\in\Sigma}[\![\mathtt{s}_1]\!]_T(\tau\mid\sigma)\times[\![\mathtt{s}_2]\!]_T(\sigma'\mid\tau)\times[\![\mathtt{s}_2]\!]_A(\tau)}{\sum_{\tau\in\Sigma}[\![\mathtt{s}_1]\!]_T(\tau\mid\sigma)\times[\![\mathtt{s}_2]\!]_A(\tau)} \\
&= \frac{\sum_{\tau\in\Sigma}[\![\mathtt{s}_1]\!]_T(\tau\mid\sigma)\times[\![\mathtt{s}_2]\!]_T(\sigma'\mid\tau)\times[\![\mathtt{s}_2]\!]_A(\tau)}{\underbrace{\frac{1}{[\![\mathtt{s}_1]\!]_A(\sigma)}\times[\![\mathtt{s}_1]\!]_A(\sigma)\times\sum_{\tau\in\Sigma}[\![\mathtt{s}_1]\!]_T(\tau\mid\sigma)\times[\![\mathtt{s}_2]\!]_A(\tau)}_{=[\![\mathtt{s}_1;\mathtt{s}_2]\!]_A(\sigma)}} \\
&= \frac{\sum_{\tau\in\Sigma}\frac{\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}'(\tau))}{\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma))}\times\mathtt{WMC}(\varphi_2'\wedge\mathbf{F}''(\sigma')\wedge\mathbf{F}'(\tau))}{\frac{1}{\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma))}\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))} && \text{(By inductive hyp.)} \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\sum_{\tau\in\Sigma}\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}'(\tau))\times\mathtt{WMC}(\varphi_2'\wedge\mathbf{F}''(\sigma')\wedge\mathbf{F}'(\tau)) \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\sum_{\tau\in\Sigma}\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma)\mid\mathbf{F}'(\tau))\times\mathtt{WMC}(\varphi_2'\wedge\mathbf{F}''(\sigma')\mid\mathbf{F}'(\tau))\times\underbrace{[\mathtt{WMC}(\mathbf{F}'(\tau))]^2}_{=1} && \text{(By Lemma D.3)} \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\sum_{\tau}\mathtt{WMC}\left(\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\mid\mathbf{F}'(\tau)\right) && \text{(By Lemma D.2)} \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\sum_{\tau}\mathtt{WMC}\left(\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\wedge\mathbf{F}'(\tau)\right)\times\frac{1}{\mathtt{WMC}(\mathbf{F}'(\tau))} && \text{(By Lemma D.3)} \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\mathtt{WMC}\left(\bigvee_{\tau}\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\wedge\mathbf{F}'(\tau)\right) && \text{(By Lemma D.4)} \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\mathtt{WMC}\left(\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\wedge\left[\bigvee_{\tau}\mathbf{F}'(\tau)\right]\right) \\
&= \frac{1}{\mathtt{WMC}(\varphi\wedge\mathbf{F}(\sigma))}\times\mathtt{WMC}\left(\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\right) \\
&= \frac{\mathtt{WMC}\left(\exists\{x_i'\}.\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma)\wedge\mathbf{F}''(\sigma')\right)}{\mathtt{WMC}(\exists\{x_i'\}.\varphi_1\wedge\varphi_2'\wedge\mathbf{F}(\sigma))} && \text{(By Lemma D.6)}
\end{aligned}
$$

`if`*-statements* Let $\varphi_1, \varphi_2, w, \varphi$ be defined as in the compilation rules. First, we prove that the accepting semantics correspond:

$$
\begin{aligned}
[\![\mathtt{if(e)}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}]\!]_A(\sigma) &= \begin{cases}[\![\mathtt{s}_1]\!]_A(\sigma) & \text{if } [\![\mathtt{e}]\!](\sigma)=\mathsf{T}\\ [\![\mathtt{s}_2]\!]_A(\sigma) & \text{if } [\![\mathtt{e}]\!](\sigma)=\mathsf{F}\end{cases} \\
&= \begin{cases}\mathtt{WMC}(\varphi_1\wedge\mathbf{F}(\sigma)) & \text{if } [\![\mathtt{e}]\!]_{\mathcal{S}}\wedge\mathbf{F}(\sigma)\models\mathsf{T}\\ \mathtt{WMC}(\varphi_2\wedge\mathbf{F}(\sigma)) & \text{otherwise}\end{cases} && \text{By Inductive Hyp.} \\
&= \mathtt{WMC}(([\![\mathtt{e}]\!]_{\mathcal{S}}\wedge\varphi_1\wedge\mathbf{F}(\sigma))\vee(\neg[\![\mathtt{e}]\!]_{\mathcal{S}}\wedge\varphi_2\wedge\mathbf{F}(\sigma))) && (\dagger) \\
&= \mathtt{WMC}\left(\left[([\![\mathtt{e}]\!]_{\mathcal{S}}\wedge\varphi_1)\vee(\neg[\![\mathtt{e}]\!]_{\mathcal{S}}\wedge\varphi_2)\right]\wedge\mathbf{F}(\sigma)\right)
\end{aligned}
$$

Where (†) follows from Lemma D.4 and the mutual exclusivity of $[\![\mathtt{e}]\!]_{\mathcal{S}}$ and $\neg[\![\mathtt{e}]\!]_{\mathcal{S}}$. Now we can prove the transition semantics correspond:

$$
\begin{aligned}
[\![\mathtt{if(e)}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}]\!]_T(\sigma' \mid \sigma) &= \begin{cases} [\![\mathtt{s}_1]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{T} \\ [\![\mathtt{s}_2]\!]_T(\sigma' \mid \sigma) & \text{if } [\![\mathtt{e}]\!](\sigma) = \mathsf{F} \end{cases} \\
&= \begin{cases} \frac{\mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'))}{\mathsf{WMC}(\varphi_1 \wedge \mathbf{F}(\sigma))} & \text{if } [\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \mathbf{F}(\sigma) \models \mathsf{T} \\ \frac{\mathsf{WMC}(\varphi_2 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma'))}{\mathsf{WMC}(\varphi_2 \wedge \mathbf{F}(\sigma))} & \text{otherwise} \end{cases} && \text{By Inductive Hyp.} \\
&= \frac{\mathsf{WMC}\left(\left[[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_1 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma')\right] \vee \left[\neg[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_2 \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma')\right]\right)}{\mathsf{WMC}\left(\left[([\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_1) \vee (\neg[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_2)\right] \wedge \mathbf{F}(\sigma)\right)} \\
&= \frac{\mathsf{WMC}\left(\left[([\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_1) \vee (\neg[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_2)\right] \wedge \mathbf{F}(\sigma) \wedge \mathbf{F}'(\sigma')\right)}{\mathsf{WMC}\left(\left[([\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_1) \vee (\neg[\![\mathtt{e}]\!]_{\mathcal{S}} \wedge \varphi_2)\right] \wedge \mathbf{F}(\sigma)\right)}
\end{aligned}
$$

This concludes the proof.