# Sound Abstraction and Decomposition of Probabilistic Programs

**Steven Holtzen** [1] **Guy Van den Broeck** [1] **Todd Millstein** [1]

## Abstract

Probabilistic programming languages are a flexible tool for specifying statistical models, but this flexibility comes at the expense of efficient analysis. It is currently difficult to compactly represent the subtle independence properties of a probabilistic program and to exploit independence properties to decompose inference. Classical graphical model abstractions do capture some properties of the underlying distribution, enabling inference algorithms to operate at the level of the graph topology. However, we observe that graph-based abstractions are often too coarse to capture interesting properties of programs. We propose a form of sound abstraction for probabilistic programs wherein the abstractions are themselves simplified programs. We provide a theoretical foundation for these abstractions, as well as an algorithm to generate them. Experimentally, we also illustrate the practical benefits of our framework as a tool to decompose probabilistic program inference.

## 1. Introduction

Graph-based abstractions of probability distributions such as Bayesian and Markov networks have played a central role in the rise of probabilistic machine learning models. Historically, these abstractions have served as both a semantic and computational tool. Semantically, the topology of the graph both asserts strong independence assumptions about the distribution – such as $d$-separation (Pearl, 1988) – and provides a scaffolding for specifying the salient quantities which define the distribution compactly, e.g., in terms of conditional probabilities. Computationally, the graph is used to construct efficient inference algorithms such as join-tree and variable elimination, which abstract away the details of local factors (Koller & Friedman, 2009; Kschischang et al., 2006). Sparsity of the graphical model implies conditional independences that can speed up inference by factorization; for example, low-width tree decompositions of the graph abstraction yield efficient junction trees.

This paper studies the question: when a probability distribution is defined by a probabilistic program, what is a natural form of abstraction? We identify abstractions that *semantically* encode conditional independence assumptions about the distribution, and *computationally* help guide the design of algorithms that decompose inference. Our approach is to automatically generate such an abstraction by utilizing techniques from the program verification community for analyzing *deterministic* code. In essence, by using a generalization of the well-known framework of predicate abstraction (Graf & Saïdi, 1997; Ball et al., 2001), we generate abstract probabilistic programs which capture properties of the original program.

We present a novel notion of *distributional soundness* which ensures that the distribution modeled by the abstract probabilistic program is consistent with the distribution of the concrete probabilistic program. Then, we provide a theory and methodology for constructing distributionally sound abstract programs for a wide class of probabilistic programs, and show that this process can decompose the concrete program by exploiting subtle independence structures such as context-sensitive and conditional independence (Boutilier et al., 1996; Pearl, 1988). Finally, we demonstrate the benefits of abstraction by recovering specialized inference procedures on a variety of archetypal statistical models encoded as probabilistic programs.

## 2. Motivating Example

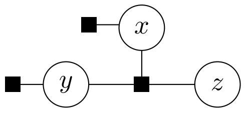
Probabilistic programs can exhibit complex structure. In particular, they admit complex operations such as control-flow logic and numerical manipulation, which entangle random variables in ways that are difficult to reason about. Consider Figure 1a, which shows a simple probabilistic program that combines 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.,

[1]University of California, Los Angeles. Correspondence to: Steven Holtzen <sholtzen@cs.ucla.edu>, Guy Van den Broeck <guyvdb@cs.ucla.edu>, Todd Millstein <todd@cs.ucla.edu>.

*Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

```
x ←discrete_dist();
y ←continuous_dist();
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 1a. Factors ■ encapsulate the two sub-programs and represent dependencies between the three random variables $x$, $y$, and $z$.

```
{x = 0} ←flip(θ_{x=0});
{0 ≤ y < 1} ←flip(θ_{0≤y<1});
{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 $\texttt{flip}(\theta)$ expression is true with probability $\theta$.

*Figure 1.* Abstracting a probabilistic program as a factor graph and a probabilistic predicate abstraction.

2016; Wood et al., 2014). This would require jointly integrating over the random variables $x$ and $y$ (or approximating the integral by sampling).

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; Winn & Minka, 2009; Pfeffer, 2009).Figure 1b 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, we 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.

We propose to instead utilize a *simpler probabilistic program* as our 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. The parameters of this probabilistic program will be chosen so that it is distributionally sound with respect to the original program. In this paper, 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. Our approach assumes that the predicates are provided *a priori*; we discuss potential automated techniques for selecting predicates in Section 5.1.

As an example of the flexibility of probabilistic programs as a language for abstraction, we illustrate their ability to capture a nuanced decomposition which relies on properties of multiplication. We observe that the program in Figure 1a has the following property: after executing Line 3, $z = 0$ if and only if $x = 0$ or $0 \leq y < 1$. Our notion of abstraction is capable of representing this relationship, and our approach can automatically produce such an abstraction.

Given the three predicates above, we will automatically generate the abstract probabilistic program in Figure 1c, which only models the distribution on the three predicates; such abstractions are a specific kind of *probabilistic predicate abstraction* (Holtzen et al., 2017). We 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 1c 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, we have decomposed the concrete program: 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 for be different for discrete and continuous distributions. This motivating example raises the following questions, which the remainder of the paper will be devoted to answering:

**Formalization** What is a distributionally sound probabilistic program abstraction? (Section 4)

**Existence** For a fixed choice of predicates, can a distributionally sound abstraction always be generated? What is an algorithm for doing so? (Section 5)

**Usefulness** What are the benefits of constructing and querying a distributionally sound abstraction over querying the original program? (Section 6)

## 3. Background

The goal of this section is to provide a concise background in semantics of probabilistic programming languages and program abstractions. We first describe the language of probability theory. Then, this language is used to give the semantics of probabilistic programming languages. Finally, we describe predicate abstractions.

### 3.1. Probability Theory

We will require some standard notions from probability theory such as a measurable space, probability space, and measurable function. Please see the appendix for detailed definitions. We will denote probability spaces as $(\Omega, \Sigma, \mu)$, where $\Omega$ is a sample space, $\Sigma$ is a $\sigma$-algebra on $\Omega$, $(\Omega, \Sigma)$ is a measurable space, and $\mu$ is a probability measure. Of particular importance is the notion of a push-forward probability measure, which will be the foundation of our probabilistic program semantics:

**Definition 1** (Push-forward)**.** Let $(\Omega, \Sigma, \mu)$ be a probability space and $(\Omega', \Sigma')$ be a measurable space. Let $f : \Omega \to \Omega'$ be a measurable function. Then, the *push-forward* of $\mu$ through $f$ is a probability measure $\nu$ on $(\Omega', \Sigma')$ 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.

### 3.2. Semantics of Probabilistic Programs

The probabilistic programs we study 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 2** (Semantics of probabilistic programs)**.** A probabilistic program $\mathcal{P}$ has two semantic components:

1. An initial probability space $(\Omega, \Sigma, \mu)$. The sample space $\Omega$ is the set of joint states of the variables in the program.
2. A measurable function $[\![\mathcal{P}]\!] : \Omega \to \Omega'$. It is implied that there exists some $\sigma$-algebra $\Sigma'$ on $\Omega'$ such that $(\Omega', \Sigma')$ form a measurable space.

We say the probability measure induced by $\mathcal{P}$ is the probability measure which results from pushing $\mu$ through $[\![\mathcal{P}]\!]$.

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 our technique to be applied to measurable sub-programs within such languages (Kozen, 1981). Further, many existing useful probabilistic programming languages do not have loops. We leave the generalization of our work to loopy programs as future work.

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

### 3.3. Predicate Abstraction

Predicate abstraction is a common style of program analysis (Graf & Saïdi, 1997; Ball et al., 2001). At a high level, the goal is to generate an abstract program that is 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 3** (Predicate domain)**.** Let $\Omega$ be a domain, and let $\Psi = \{\psi_1, \psi_2, \cdots, \psi_n\}$ be a collection of predicates on $\Omega$. Then the *predicate domain* $\mathcal{D}_\Psi$ over $\Psi$ is the set of all $2^n$ truth assignments to the predicates in $\Psi$.

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 $\Psi$.

When the collection of predicates $\Psi$ is insufficient to capture the behavior of the concrete program, the abstraction must behave *non-deterministically* in order to remain an over-approximation. This is best illustrated with an example:

**Example 1** (A simple predicate abstraction)**.** Consider the concrete program $\mathcal{C} =$ `x` $\leftarrow$ `x + 1;`, which simply increments a variable `x`. We consider the predicate domain $\Psi = \{x < 0\}$. Our 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 $\star$ symbol: `{x<0}` $\leftarrow$ `{x<0}` $\wedge\ \star$;

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 predicate 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 4** (Weakest precondition)**.** Let $\mathcal{P}$ be a program and $\phi$ be a predicate. Then the *weakest precondition of* $\mathcal{P}$ *with respect to* $\phi$, denoted $\mathbf{WP}(\mathcal{P}, \phi)$, is the most general predicate $\psi$ such that $\psi$ holding before executing $\mathcal{P}$ implies that $\phi$ holds after executing $\mathcal{P}$.

**Probabilistic predicate abstractions** Building on techniques from the program analysis community for constructing predicate abstractions, we recently introduced a notion of probabilistic predicate abstraction and a technique for constructing them (Holtzen et al., 2017). Specifically, each non-deterministic Boolean choice in a non-deterministic predicate abstraction is replaced with a Bernoulli random variable. However, the notion of soundness in that work is a probabilistic analog of the traditional notion of over-approximation, which is not sufficient to relate the results of inference on the abstract and concrete programs. This motivates our stronger notion of *distributional soundness*, which is defined in the next section.

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

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

**Definition 6** (Distributional soundness)**.** Let $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ and $[\![\mathcal{A}]\!] : \mathcal{D}_\Psi \to \mathcal{D}_{\Psi'}$ be measurable functions, where $\mathcal{D}_\Psi$ and $\mathcal{D}_\Psi{}'$ are predicate domains on $\Omega$ and $\Omega'$ respectively. Then $[\![\mathcal{A}]\!]$ is a *distributionally sound abstraction of* $[\![\mathcal{C}]\!]$ if the following diagram commutes for any initial concrete probability space $(\Omega, \Sigma, \mu)$:

$$\begin{tikzcd}
(\Omega, \Sigma, \mu) \arrow[r, "{[\![\mathcal{C}]\!]}"] \arrow[d, "\alpha"] & (\Omega', \Sigma', \mu') \arrow[d, "\alpha'"] \\
(\mathcal{D}_\Psi, \Sigma_{\mathcal{D}_\Psi}, \nu) \arrow[r, "{[\![\mathcal{A}]\!]}"] & (\mathcal{D}_{\Psi'}, \Sigma_{\mathcal{D}_{\Psi'}}, \nu')
\end{tikzcd}$$

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 $\mathcal{D}_{\Psi'}$. Specifically, we describe a class of events for which we can perform inference using exclusively the abstraction:

**Definition 7** (Corresponding events)**.** Let $(\Omega, \Sigma, \mu)$ be a probability space, $(\mathcal{D}_\Psi, \Sigma_{\mathcal{D}_\Psi})$ be a measurable space over predicate domain $\mathcal{D}_\Psi$, and $[\cdot]$ be an abstraction function. Then for any abstract event $e_{\mathcal{D}_\Psi} \in \Sigma_{\mathcal{D}_\Psi}$, there exists a *corresponding concrete event* $e_\Omega = \bigcup \left\{ [a]^{-1} \mid a \in e_{\mathcal{D}_\Psi} \right\}$. We call the pair $(e_{\mathcal{D}_\Psi}, e_\Omega)$ an *event pair*.

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

**Proposition 1** (Distributional soundness implies soundness for inference)**.** *Let* $[\![\mathcal{A}]\!] : \mathcal{D}_\Psi \to \mathcal{D}_{\Psi'}$ *be a distributionally sound abstraction of* $[\![\mathcal{C}]\!] : \Omega \to \Omega$*. Then for any initial probability space* $(\Omega, \Sigma, \mu)$*, and any event pair* $(e_{\mathcal{D}_{\Psi'}}, e_{\Omega'})$*, it is the case that* $\Pr_{\nu'}(e_{\mathcal{D}_{\Psi'}}) = \Pr_{\mu'}(e_{\Omega'})$*, where* $\mu'$ *is the push-forward of* $\mu$ *through* $[\![\mathcal{C}]\!]$ *and* $\nu'$ *is the push-forward of* $\mu$ *through* $[\![\mathcal{A}]\!] \circ [\cdot]$.

As outlined in Section 1, 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 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* $e_{\mathcal{D}_\Psi} \in \Sigma_{\mathcal{D}_\Psi}$ *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. 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. Initially we are provided a program $\mathcal{C}$ and a set of predicates $\Psi$ of interest. We show how to construct 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 6 commutes.

Given $\mathcal{C}$ and $\Psi$ it is not always possible to generate a distributionally sound abstraction from $\mathcal{D}_\Psi$ to $\mathcal{D}_\Psi$, because the predicates in $\Psi$ might not be sufficiently expressive to capture all the required concrete behavior. We resolve this problem by automatically identifying predicates called *completions* (denoted $\Phi$), which are added to $\Psi$, 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 the decomposition that we saw in Section 1.

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 allows us to avoid directly reasoning about distributions.

**Definition 8** (Tight abstraction)**.** Let $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ and, $[\![\mathcal{A}]\!] : \mathcal{D}_\Psi \to \mathcal{D}_{\Psi'}$ be measurable functions, where $\mathcal{D}_\Psi$ and $\mathcal{D}_\Psi{}'$ are predicate domains. Then we say $[\![\mathcal{A}]\!]$ is a *tight abstraction* of $[\![\mathcal{C}]\!]$ if for any $c \in \Omega$, we have that:

$$\big[[\![\mathcal{C}]\!](c)\big]_{\Psi'} = [\![\mathcal{A}]\!]\big([c]_\Psi\big). \tag{1}$$

**Theorem 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 $[\![\mathcal{C}]\!]$.*

See the supplementary materials for a detailed proof. 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 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[2]:

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

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 $\mathcal{D}_\Psi$ for $[\![\mathcal{A}]\!](\{x < 0\})$ that satisfies condition (1) in Definition 8. 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 2 showed that adding completion predicates $\Phi$ to $\Psi$ enables the creation of a tight abstraction from $\mathcal{D}_{\Psi\cup\Phi}$ to $\mathcal{D}_\Psi$. In general we say that $\Phi$ *completes* $\Psi$ with respect to $\Psi'$ and $[\![\mathcal{C}]\!]$ if there exists a tight abstraction $[\![\mathcal{A}]\!] : \mathcal{D}_{\Psi\cup\Phi} \to \mathcal{D}_{\Psi'}$. We call $\Psi \cup \Phi$ the completed set of predicates and $\mathcal{D}_{\Psi\cup\Phi}$ the completed predicate domain. Algorithm 1 automatically completes a set of predicates $\Psi$ with respect to $\Psi'$ and $[\![\mathcal{C}]\!]$ and generates a corresponding tight abstraction and initial probability space.[3] The algorithm relies on the standard notion of the weakest precondition (see Definition 4). We formally state the correctness of Algorithm 1:

**Theorem 2** (Domain completion)**.** *Let $[\![\mathcal{C}]\!] : \Omega \to \Omega'$ be a measurable function and $\Psi$ and $\Psi'$ be sets of predicates, and let $(\Omega, \Sigma, \mu)$ be an initial concrete probability space. Then Algorithm 1 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 $\nu$ is the push-forward of $\mu$ through $[\cdot]_{\Psi\cup\Phi}$.*

**Discussion** We provide some discussion of Algorithm 1. Then, we describe optimizations that can improve the performance of the algorithm in practice. Algorithm 1 proceeds as follows. First, on Line 4 the set of predicates $\Phi$ 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 $c \in [\phi]^{-1}$, $[[\![\mathcal{C}]\!](c)]_{\Psi'} = a'$.

[2]We will represent functions with discrete domains as sets of pairs, where the first element of the pair is the input and the second element is the output of the function.

[3]We describe Algorithm 1 as directly producing a measurable function, but our implementation adapts standard predicate abstraction techniques (Ball et al., 2001; Holtzen et al., 2017) to generate an abstract probabilistic program.

**Algorithm 1** Domain completion

```
1: Input: probability space $(\Omega, \Sigma, \mu)$, measurable function $[\![\mathcal{C}]\!]$, input predicates $\Psi$ and output predicates $\Psi'$.
2: $[\![\mathcal{A}]\!] \leftarrow []$                    // New tight abstract function
3: $\nu \leftarrow []$                      // New probability measure
4: $\Phi = \left\{ \mathbf{WP}([\![\mathcal{C}]\!], a') \mid a' \in \mathcal{D}_{\Psi'} \right\}$
5: for $a \in \mathcal{D}_{\Psi \cup \Phi}$ do
6:     $c' \leftarrow [\![\mathcal{C}]\!](c)$ for any $c \in [a]^{-1}$
7:     Append $(a, [c']_{\Psi'})$ to $[\![A]\!]$
8:     Append $(a, \Pr_\mu(a))$ to $\nu$
9: end for
10: return $([\![\mathcal{A}]\!], (\mathcal{D}_{\Psi \cup \Phi}, \Sigma_{\mathcal{D}_{\Psi \cup \Phi}}, \nu))$
```

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 5. For each $a \in \mathcal{D}_{\Phi \cup \Psi}$, there is some $\phi \in \mathcal{D}_\Phi$ such that $a$ implies $\phi$, which guarantees that we can give a deterministic function $[\![\mathcal{A}]\!]$ for $a$ following the arguments in the previous paragraph.

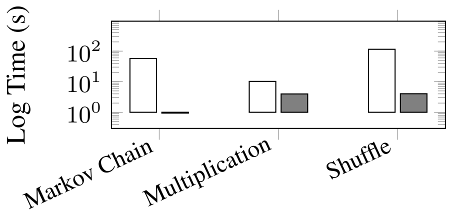
As an example, consider the program $\mathcal{P} = $ `x←x+1` and the predicate $\Psi = \{x < 0\}$. We wish to evaluate Algorithm 1 with input probability space $(\mathcal{D}_\Psi, \Sigma, \mu)$ with initial and final predicate domains over $\Psi$, i.e. $\mathcal{D}_\Psi = \{\{x < 0\}, \neg\{x < 0\}\}$, as in Example 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 $\Psi'$. 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).

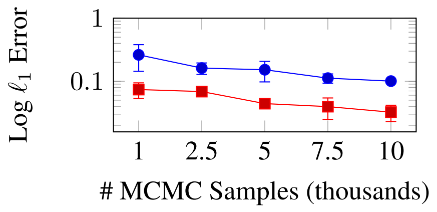
### 5.1. Selecting Predicates

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, we believe that 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-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).



(a) Exact inference results on the Psi system (see Section 6.1). □ 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 $\ell_1$-norm between the true value and the approximated value. See Section 6.2.

*Figure 2.* Experimental results.

## 6. Decomposition via Abstraction

The theory and algorithm presented in the previous sections can be used to simplify inference via a process we call *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 $\Psi$, which must include the necessary predicates for describing $q$ and $e$. Next, we utilize Algorithm 1, 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 1.

Figure 2a shows the computational benefits of decomposition via abstraction on exact and approximate inference tasks, which are elaborated on in the following sub-sections.

### 6.1. Exact Inference

We ask 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. We show how our decomposition technique serves to complement path-based decompositions in the following way. For each probabilistic program, we used Psi[4] (Gehr et al., 2016) to compute an inference query on (1) the concrete program, and (2) the abstract program and the sub-queries. We report the total time for each query task in Figure 2a. The supplementary material gives the source code of each probabilistic program. We report three experiments, which each highlight an important property of probabilistic programs that may be exploited via abstraction. In each case, we show 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.

**Multiplication** This experiment uses a complete version of the example described in Section 2 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 1 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 & Friedman, 2009):

$$\begin{tikzcd} \boxed{n_1} \arrow[r] & \boxed{n_2} \arrow[r] & \cdots \arrow[r] & \boxed{n_k} \end{tikzcd}$$

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

```
1  n₁ ← flip($\theta_{n_1}$);
2  n₂ ← if n₁ then flip($\theta_{n_2|n_1}$) else flip($\theta_{n_2|\overline{n_1}}$)
3  ...
4  n_k ← if n_{k−1} then flip($\theta_{n_k|n_{k-1}}$) else
5      flip($\theta_{n_k|\overline{n_{k-1}}}$);
```

First we generate an abstraction using the predicates $\{n_1\}$ and $\{n_k\}$. Our 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  {$n_1$} ← flip($\theta_{n_1}$);
2  {$n_k$} ← if {$n_1$} then flip($\theta_{n_k|n_1}$) else
3      flip($\theta_{n_k|\overline{n_1}}$);
```

Next we must evaluate the sub-queries. The parameter $\theta_{n_1}$ 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  {$n_1$} ← flip($\theta_{n_1}$);
2  {$n_{k-1}$} ← if {$n_1$} then flip($\theta_{n_{k-1}|n_1}$) else
       flip($\theta_{n_{k-1}|\overline{n_1}}$);
3  {$n_k$} ← if {$n_{k-1}$} then flip($\theta_{n_k|n_{k-1}}$) else
       flip($\theta_{n_k|\overline{n_{k-1}}}$);
```

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.

**Shuffle** Many intractable models can be rendered tractable by exploiting the underlying symmetry of random variables; this is often called *lifted inference* (Kersting, 2012; Niepert & Van den Broeck, 2014). This example illustrates the potential connections between probabilistic program abstraction and lifted inference. Consider the following probabilistic program, which shuffles a small deck

[4] We utilized build `5334524fe`.

of cards:

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

We wish to compute $\Pr(\texttt{deck[0]} = 1)$, 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 our 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:

```
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 1. 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.

### 6.2. 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 $\mu$ and variance $\sigma$):

```
x←𝒩(μ, σ);
y₁ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
y₂ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
y₃ ←if(x<k){𝒩(μ_y,σ_y)} else {𝒩(μ'_y,σ'_y)};
observe(y₁<c ∧ y₂<c ∧ y₃ ≥c);
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 2b, which shows the performance of MCMC on the un-abstracted model using WebPPL with a fixed number of samples (Goodman & Stuhlmüller, 2014).

The red performance line in Figure 2b shows the convergence of an abstracted model generated by Algorithm 1 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 program, the final query on the abstract program may be performed using enumeration, which can handle discontinuities and low-probability evidence. See the appendix for the full text of these programs.

## 7. 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; our technique 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 & Monerau, 2012; Monniaux, 2000; 2001).

## 8. Conclusion

This work addresses the question: what is a useful abstraction for a probabilistic program? We showed that such a useful abstraction must be distributionally sound, and described the theory and practice for constructing such abstractions. Then, we empirically validated this approach on approximate and exact inference tasks. For future work, we plan to explore loopy programs as well as automated predicate discovery techniques.

## Acknowledgments

This work is partially supported by NSF grants #CCF-1527923, #IIS-1657613, #IIS-1633857 and DARPA XAI grant #N66001-17-2-4032. The authors would like to thank Tal Friedman and Jon Aytac for helpful discussions.

## References

Albarghouthi, A., D'Antoni, L., Drews, S., and Nori, A. Fairsquare: Probabilistic verification for program fairness. In *OOPSLA*, volume 1, pp. 80:1–80:30, October 2017.

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

Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K. Automatic predicate abstraction of c programs. In *Proc. of PLDI*, pp. 203–213, 2001.

Boutilier, C., Friedman, N., Goldszmidt, M., and Koller, D. Context-specific independence in bayesian networks. In *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*, UAI'96, pp. 115–123, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-412-X.

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M. A., Li, P., and Riddell, A. Stan: A probabilistic programming language. *J. Statistical Software*, VV(Ii), 2016.

Chaganty, A., Nori, A. V., and Rajamani, S. K. Efficiently Sampling Probabilistic Programs via Program Analysis. *Proc. of AISTATS*, 31:153–160, 2013. ISSN 15337928.

Chistikov, D., Dimitrova, R., and Majumdar, R. Approximate counting in smt and value estimation for probabilistic programs. In *Proc. of TACAS*, pp. 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.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003. ISSN 0004-5411. doi: 10.1145/876638.876643.

Cousot, P. and Monerau, M. Probabilistic abstract interpretation. In *Proc. of ESOP*, pp. 169–193, 2012. ISBN 9783642288685. doi: 10.1007/978-3-642-28869-20_9.

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

Gehr, T., Misailovic, S., and Vechev, M. Psi: Exact symbolic inference for probabilistic programs. *Proc. of ESOP/ETAPS*, 9779:62–83, 2016. ISSN 16113349.

Goodman, N. D. and Stuhlmüller, A. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2018-5-22.

Goodman, N. D., Mansinghka, V. K., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. Church: A language for generative models. In *Proc. of UAI*, pp. 220–229, 2008.

Graf, S. and Saïdi, H. Construction of abstract state graphs with PVS. In *Proc. of CAV*, volume 1254, pp. 72–83. Springer-Verlag, June 1997.

Holtzen, S., Millstein, T., and Van den Broeck, G. Probabilistic program abstractions. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*, August 2017.

Hur, C.-K., Nori, A. V., Rajamani, S. K., and Samuel, S. Slicing probabilistic programs. *Proc. of PLDI*, pp. 133–144, 2014. doi: 10.1145/2594291.2594303.

Kersting, K. Lifted probabilistic inference. In *Proceedings of the 20th European Conference on Artificial Intelligence*, ECAI'12, pp. 33–38, Amsterdam, The Netherlands, The Netherlands, 2012. IOS Press. ISBN 978-1-61499-097-0. doi: 10.3233/978-1-61499-098-7-33.

Koller, D. and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009. ISBN 0262013193, 9780262013192.

Kozen, D. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328 – 350, 1981. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(81)90036-2.

Kschischang, F. R., Frey, B. J., and Loeliger, H. A. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theor.*, 47(2):498–519, September 2006. ISSN 0018-9448. doi: 10.1109/18.910572.

McCallum, A., Schultz, K., and Singh, S. Factorie: Probabilistic programming via imperatively defined factor graphs. *Proc. of NIPS*, 22:1249–1257, 2009. ISSN 03643417.

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

Monniaux, D. Abstract interpretation of probabilistic semantics. In *International Symposium on Static Analysis*, pp. 322–339, 2000. ISBN 3-540-67668-6.

Monniaux, D. An abstract monte-carlo method for the analysis of probabilistic programs. *SIGPLAN Not.*, 36(3):93–101, January 2001. ISSN 0362-1340. doi: 10.1145/373243.360211.

Niepert, M. and Van den Broeck, G. Tractability through exchangeability: A new perspective on efficient probabilistic inference. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence, AAAI Conference on Artificial Intelligence*, July 2014.

Nori, A., Hur, C.-K., Rajamani, S., and Samuel, S. R2: An efficient mcmc sampler for probabilistic programs. AAAI, July 2014.

Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-73-7.

Pfeffer, A. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, pp. 1–9, 2009. ISSN 10450823.

Sankaranarayanan, S., Chakarov, A., and Gulwani, S. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. *SIGPLAN Not.*, 48(6):447–458, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462179.

Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

Winn, J. and Minka, T. Probabilistic programming with infer.net. September 2009.

Wood, F., van de Meent, J. W., and Mansinghka, V. A new approach to probabilistic programming inference. In *Proc. of AISTATS*, pp. 1024–1032, 2014.

## A. Background: Probability Theory

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

**Definition 9** (Measurable space)**.** Let $\Omega$ be a set, called the *sample space*. In the context of programs $\Omega$ is sometimes called a domain. A $\sigma$-algebra $\Sigma$ on $\Omega$ is a collection of subsets of $\Omega$ that is (i) closed under countable unions; (ii) closed under complementation; (iii) contains $\Omega$. We call the pair $(\Omega, \Sigma)$ a *measurable space*.

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

**Definition 10** (Probability space)**.** Let $(\Omega, \Sigma)$ be a measurable space and $\mu : \Sigma \to \mathbb{R}$ be a function such that (i) $\mu$ is countably additive; (ii) $\mu(\Omega) = 1$. The tuple $(\Omega, \Sigma, \mu)$ is called a *probability space*, and $\mu$ is called a *probability measure*.

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

**Definition 11** (Measurable function)**.** Let $(\Omega, \Sigma)$ and $(\Omega', \Sigma')$ 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 12** (Push-forward)**.** Let $(\Omega, \Sigma, \mu)$ be a probability space and $(\Omega', \Sigma')$ be a measurable space, and $f : \Omega \to \Omega'$ be a measurable function. Then, the *push-forward* of $\mu$ through $f$ is a probability measure $\nu$ on $(\Omega', \Sigma')$ 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.

## B. Proofs of Theorems

*Proof of Theorem 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 $\nu' : \Sigma_{\mathcal{D}_\Psi} \to [0, 1]$ by following both paths in the commutative diagram from Definition 6, 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

$$\begin{aligned}\nu' = \alpha'(\mu') &= a' \mapsto \mu'([a']^{-1}) \\ &= a' \mapsto \mu([\![\mathcal{C}]\!]^{-1}([a']^{-1})). \qquad (2)\end{aligned}$$

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

Next, following the abstract path, we first compute $\nu : \Sigma_{\mathcal{D}_\Psi} \to [0, 1]$, which is $\nu = \alpha(\mu) = a \mapsto \mu([a]^{-1})$. Then, we compute $\nu'$ using the push-forward of $[\![\mathcal{A}]\!]$:

$$\begin{aligned}\nu' = a' \mapsto \nu(([\![\mathcal{A}]\!])^{-1}(a')) &= a' \mapsto \alpha(\mu)(([\![\mathcal{A}]\!])^{-1}(a')) \\ &= a' \mapsto \mu([([\![\mathcal{A}]\!])^{-1}(a')]^{-1}). \qquad (3)\end{aligned}$$

To prove these $\nu'$ measures equivalent, it suffices to show that $[\![\mathcal{C}]\!]^{-1}([a']^{-1}) = [([\![\mathcal{A}]\!])^{-1}(a')]^{-1}$. This follows from Definition 8 by taking the inverse of both sides. □

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

## C. Source Code for Experiments

For each experiment, the time reported in Figure 2a for the "un-abstracted model" is the time taken to evaluate the program given in the "Concrete program" sub-section, and the time given to evaluate the "abstracted model" is the net-total of running all the programs given in the "Sub-queries" section plus the "Abstract program" section.

### C.1. Markov Chain

#### C.1.1. Concrete program

```
% def main(){
    n1 := flip(0.5);
    n2 := 0;
    if n1 {
      n2 = flip(0.1);
    } else {
      n2 = flip(0.95);
    }

    n3 := 0;
    if n2 {
      n3 = flip(0.8);
    } else {
      n3 = flip(0.2);
    }

    n4 := 0;
    if n3 {
      n4 = flip(0.1);
    } else {
      n4 = flip(0.4);
    }

    n5 := 0;
    if n4 {
      n5 = flip(0.2);
    } else {
      n5 = flip(0.9);
    }

  n6 := 0;
  if n5 {
     n6 = flip(0.3);
  } else {
     n6 = flip(0.7);
  }

  n7 := 0;
  if n6 {
     n7 = flip(0.4);
  } else {
     n7 = flip(0.2);
  }

  n8 := 0;
  if n7 {
     n8 = flip(0.2);
  } else {
     n8 = flip(0.9);
  }

  n9 := 0;
  if n8 {
     n9 = flip(0.01);
  } else {
     n9 = flip(0.001);
  }

  observe(n9);
  return n1;
}
```

### C.1.2. Abstract program

```
% def main(){
  n1 := flip(0.5);
  n9 := 0;
  if n1 {
     // result of decomp2a
     n9 = flip(4561249/625000000);
  } else {
     // result of decomp2b
     n9 = flip(9189971/1250000000);
  }
  observe(n9);
  return n1;
}
```

### C.1.3. Sub-queries

*Listing 1.* decomp2a

```
% // Pr(n9 | n1 = t)
def main(){
   // result of decomp3a
   n8 := flip(437361/625000);
   n9 := 0;
   if n8 {
      n9 = flip(0.01);
   } else {
      n9 = flip(0.001);
   }
   return n9;
}
```

*Listing 2.* decomp2b

```
% // Pr(n9 | n1 = f)
def main(){
   // result of decomp3b
   n8 := flip(882219/1250000);
   n9 := 0;
   if n8 {
      n9 = flip(0.01);
   } else {
      n9 = flip(0.001);
   }
   return n9;
}
```

The remaining decompositions are similar to these first two, and omitted.

## C.2. Multiplication

### C.2.1. Concrete program

```
% // a complex mixture
def cont_mix() {
    a := uniform(0,10);
    b := uniform(0,5);
    c := uniform(0,5);
    if c < 0 {
       c = c * 4;
    }
    if a < c {
       b = b + a;
    }
    return a + b + c;
}

def markov() {
    n1 := flip(0.5);
    n2 := 0;
    if n1 {
      n2 = flip(0.1);
    } else {
      n2 = flip(0.95);
    }

    n3 := 0;
    if n2 {
      n3 = flip(0.8);
    } else {
      n3 = flip(0.2);
    }

    n4 := 0;
    if n3 {
      n4 = flip(0.1);
    } else {
      n4 = flip(0.4);
    }

    n5 := 0;
    if n4 {
      n5 = flip(0.2);
    } else {
      n5 = flip(0.9);
    }

    n6 := 0;
    if n5 {
      n6 = flip(0.3);
    } else {
      n6 = flip(0.7);
    }

   n7 := 0;
   if n6 {
      n7 = flip(0.4);
   } else {
      n7 = flip(0.2);
   }
   return n7;
}

def main() {
    r := markov();
    mul := 0;
    if r {
       mul = 0;
    } else {
      mul = 1;
    }

    z := floor(cont_mix()) * mul;
    return z < 1
}
```

### C.2.2. Abstract program

```
% def main() {
    // result of decomp1
    cont_lt1 := flip(1/1800);
    // result of decomp2
    disc_lt1 := flip(70437/250000);
    z := cont_lt1 || disc_lt1;
    return z;
}
```

### C.2.3. Sub-queries

*Listing 3.* decomp1

```
% def cont_mix() {
    a := uniform(0,10);
    b := uniform(0,5);
    c := uniform(0,5);
    if c < 0 {
      c = c * 4;
    }
    if a < c {
      b = b + a;
    }
    return a + b + c;
}
def main() {
    return cont_mix() < 1;
}
```

*Listing 4.* decomp2

```
% def markov() {
    n1 := flip(0.5);
    n2 := 0;
    if n1 {
      n2 = flip(0.1);
    } else {
      n2 = flip(0.95);
    }

    n3 := 0;
    if n2 {
      n3 = flip(0.8);
    } else {
      n3 = flip(0.2);
    }

    n4 := 0;
    if n3 {
      n4 = flip(0.1);
    } else {
      n4 = flip(0.4);
    }

    n5 := 0;
    if n4 {
      n5 = flip(0.2);
    } else {
      n5 = flip(0.9);
    }

    n6 := 0;
    if n5 {
      n6 = flip(0.3);
    } else {
      n6 = flip(0.7);
    }

   n7 := 0;
   if n6 {
      n7 = flip(0.4);
   } else {
      n7 = flip(0.2);
    }
    return n7;
}

def main() {
    return markov()
}
```

### C.3. Shuffle

#### C.3.1. Concrete program

```
% def main() {
    deck := [1,2,3,4,5,6];
    for idx in [0..5) {
        j := uniformInt(idx, 6);
        tmp := deck[idx];
        deck[idx] = deck[j];
        deck[j] = tmp;
    }
    return deck[0] == 1;
}
```

#### C.3.2. Abstract program

```
% def main() {
    // encode '1' as 1, 'not 1' as 0
    deck := [1,0,0,0,0,0];
    for idx in [0..5) {
        j := uniformInt(idx, 6);
        tmp := deck[idx];
        deck[idx] = deck[j];
        deck[j] = tmp;
    }
    return deck[0] == 1;
}
```

#### C.3.3. Sub-queries

This program required no sub-queries.

## D. Approximate Inference

The approximate inference experiments were implemented using WebPPL (Goodman & Stuhlmüller, 2014).

First, we give the initial un-abstracted model:

```
% var mix = function(x) {
  if(x<0) {
    return gaussian(0, 5);
  } else {
    return gaussian(-10, 5);
  }
}

var model = function() {
  var x = gaussian(0, 10);
  var y = mix(x);
  var y2 = mix(x);
  var y3 = mix(x);
  var y4 = mix(x);

  condition((y < -5) && (y2 < -5)
      && (y3 >= -5) && (y4 >= -5));
  return x < 0;
}

var dist = Infer(
  {method: 'MCMC', samples: 7500, lag: 0,
      burn: 10}, model);
```

```
var intoProb = function(o) {
  // converts an inference result into a
      probability of being true
  ...
}

var gauss1 = function() {
  return gaussian(0, 5) < -5
}

var gauss2 = function() {
  return gaussian(-10, 5) < -5
}

var probGaus1 = function() {
  var dist1 = Infer(
  {method: 'MCMC', samples: num_samples,
      lag: 0, burn: 10}, gauss1);
  return intoProb(dist1)
}

var probGaus2 = function() {
  var dist2 = Infer(
    {method: 'MCMC', samples: num_samples
      , lag: 0, burn: 10},
    gauss2);
  return intoProb(dist2)
}

var dist3 = function() {
  return gaussian(0, 10) < 0;
}

var distGuard = Infer(
  {method: 'MCMC', samples: num_samples,
      lag: 0, burn: 10},
  dist3);

var conditionedGeometric = function() {
  var x = bernoulli(intoProb(distGuard))
      ;
  var y1 = x ? bernoulli(probGaus1()) :
      bernoulli(probGaus2());
  var y2 = x ? bernoulli(probGaus1()) :
      bernoulli(probGaus2());
  var y3 = x ? bernoulli(probGaus1()) :
      bernoulli(probGaus2());
  var y4 = x ? bernoulli(probGaus1()) :
      bernoulli(probGaus2());

  condition(y1 && y2 && !y3 && !y4);
  return x
}
```