# Probabilistic Program Abstractions

**Steven Holtzen** and **Todd Millstein** and **Guy Van den Broeck**
Computer Science Department
University of California, Los Angeles
{sholtzen,todd,guyvdb}@cs.ucla.edu

## Abstract

Abstraction is a fundamental tool for reasoning about complex systems. Program abstraction has been utilized to great effect for analyzing deterministic programs. At the heart of program abstraction is the relationship between a concrete program, which is difficult to analyze, and an abstract program, which is more tractable. Program abstractions, however, are typically not probabilistic. We generalize non-deterministic program abstractions to probabilistic program abstractions by explicitly quantifying the non-deterministic choices. Our framework upgrades key definitions and properties of abstractions to the probabilistic context. We also discuss preliminary ideas for performing inference on probabilistic abstractions and general probabilistic programs.

## 1 INTRODUCTION & MOTIVATION

Program abstractions are a richly studied method from the programming languages community for reasoning about intractably complex programs (Cousot and Cousot, 1977). An abstraction is typically an over-approximation to a program: any execution that is possible in the original program is contained within the abstraction. Over-approximation allows abstractions to be used to prove *program invariants*: any property of all executions in the abstraction is also true of all executions in the original program. To achieve this goal while being more tractable than the concrete program, abstractions work on a simplified domain. The abstraction selectively models particular aspects of the original program while utilizing non-determinism to conservatively model the rest.

Non-deterministic abstractions are useful for verifying properties such as reachability in a concrete program. However, abstractions are decidedly not probabilistic: they are concerned with the possible, not the probable. Therefore, they fail to support more nuanced queries such as probabilistic reachability, or probabilistic program inference. We seek to enhance the program abstraction framework by explicitly quantifying the non-deterministic choices made in the abstraction, turning the program abstraction into a probabilistic model. That is, our probabilistic abstractions are themselves probabilistic programs, which have been the subject of intense study recently (e.g., Goodman et al. (2008); Fierens et al. (2013); Wood et al. (2014); Carpenter et al. (2016)).

The key contribution of this paper is the development of a foundational theory for probabilistic program abstractions. We define probabilistic abstractions as a natural generalization of traditional abstractions, using random variables as the abstraction mechanism instead of non-determinism. We also formalize the relationship between a probabilistic abstraction and a concrete program, again generalizing from the non-deterministic setting. This includes semantics in both the concrete and abstract domain, the connection between these semantics, and the notion of a sound probabilistic over-approximation.

A well-known construction of non-deterministic program abstractions is that of a *predicate abstraction* (Graf and Saïdi, 1997; Ball et al., 2001). It induces an abstraction relative to a given set of Boolean predicates about the program state. We define *probabilistic predicate abstractions*, which are represented by a simple Bernoulli probabilistic program, as an instance of our framework, and a generalization of classical predicate abstraction.

We conclude with a discussion of ideas for performing inference in probabilistic predicate abstractions, building on *model checking* techniques from the programming languages community and *weighted model counting* from the artificial intelligence community. We then discuss how probabilistic abstractions could be used to simplify inference in probabilistic concrete programs.

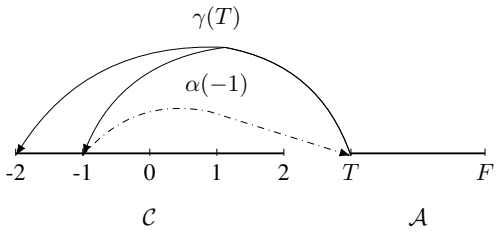


Figure 1: Visualization of a simple predicate domain. The five concrete states over an integer variable $x$ in the range $[-2, 2]$ are abstracted to two states based on the valuation of the predicate $(x < 0)$. We see, for example, that $\alpha(-1) = T$, and $\gamma(T) = \{-2, -1\}$).

# 2 NON-DETERMINISTIC PROGRAM ABSTRACTION

In this section we provide the semantics and properties of an *over-approximate non-deterministic* abstraction and provide an example of a particular class of over-approximations known as predicate abstractions.

## 2.1 SEMANTICS AND PROPERTIES

A concrete program is a syntactic object written $\mathcal{C}$. The semantics of a concrete program, which for simplicity we also denote $\mathcal{C}$, is a function from input states to output states over some concrete domain $\mathcal{D}_\mathcal{C}$. Concrete states are total assignments to all variables in the concrete domain, which we denote $z \in \mathcal{D}_\mathcal{C}$.

In general, the problem of proving that a given program satisfies a desired invariant is undecidable. Advances in theorem proving techniques such as Satisfiability Modulo Theory (SMT) solvers (e.g., De Moura and Bjørner (2008)) render reasoning in many useful theories tractable, yet there exist common program structures that lie outside of supported theories.

The framework of abstract interpretation (Cousot and Cousot, 1977) provides a general technique for relating a concrete program $\mathcal{C}$ to another program $\mathcal{A}$ which we refer to as an *abstraction*. We describe a specialization of the abstract interpretation framework.

**Definition 2.1. Abstract semantics of an abstraction.** The abstract semantics of an abstraction $\mathcal{A}$, which for simplicity we also denote $\mathcal{A}$, is a function from input states to sets of output states over an abstract domain $\mathcal{D}_\mathcal{A}$, written $\mathcal{A} : \mathcal{D}_\mathcal{A} \to 2^{\mathcal{D}_\mathcal{A}}$.

Intuitively, the nondeterminism in the abstract semantics of an abstraction represents uncertainty due to the loss of information in abstracting $\mathcal{C}$ to $\mathcal{A}$. We represent this non-determinism as a set of possible abstract states, denoted $a \in \mathcal{D}_\mathcal{A}$. To relate concrete programs with abstractions we introduce two mappings between concrete and abstract states.

**Definition 2.2. Abstraction and concretization functions.** An *abstraction function* for $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{A}$ is a function $\alpha : \mathcal{D}_\mathcal{C} \to \mathcal{D}_\mathcal{A}$ that maps each concrete state to its abstract representative. A *concretization function* for $\mathcal{D}_\mathcal{C}$ and $\mathcal{D}_\mathcal{A}$ is a function $\gamma : \mathcal{D}_\mathcal{A} \to 2^{\mathcal{D}_\mathcal{C}}$ that maps each abstract state to a set of concrete states. When applied to sets, $\gamma$ and $\alpha$ respectively concretize or abstract each element of the set.

Abstraction and concretization functions are related.

**Definition 2.3. Compatibility.** An abstraction function $\alpha$ and concretization function $\gamma$ are *compatible* if $z \in \gamma(\alpha(z))$ for all $z \in \mathcal{D}_\mathcal{C}$. As an extension, the two functions are *strongly compatible* if they are compatible and for any $a$ and $z \in \gamma(a)$, we have that $z \notin \gamma(a')$ for any $a' \neq a$.

A *predicate domain* is a well-studied abstract domain induced by a given sequence of predicates $(p_1, \ldots, p_n)$ about the concrete state. The abstract domain $\mathcal{D}_\mathcal{A}$ consists of $n$ Boolean variables $(b_1, \ldots, b_n)$ and so has $2^n$ possible elements, one for each valuation to the $n$ variables. For instance, suppose $\mathcal{D}_\mathcal{C}$ consists of a single integer variable $x$ whose value is in the range $[-2, 2]$. The single predicate $(x < 0)$ induces an abstract domain with two possible states, representing the concrete states where $(x < 0)$ is true and false. See Figure 1 for a visualization. The abstraction function $\alpha$ maps each concrete state $z$ to the abstract state $(p_1(z), \ldots, p_n(z))$, and the concretization function $\gamma$ maps each abstract state $a$ to the set of concrete states consistent with it: $\{z \in \mathcal{D}_\mathcal{C} \mid (p_1(z), \ldots, p_n(z)) = a\}$. The functions $\alpha$ and $\gamma$ are strongly compatible for predicate domains.

Intuitively, an abstraction represents a set of possible concrete programs, which is formalized as follows:

**Definition 2.4. Concrete semantics of an abstraction.** The concrete semantics of an abstraction $\mathcal{A}$, given compatible abstraction and concretization functions $\alpha$ and $\gamma$, is a function $[\![\mathcal{A}]\!] : \mathcal{D}_\mathcal{C} \to 2^{\mathcal{D}_\mathcal{C}}$ defined as follows:

$$[\![\mathcal{A}]\!](z) = \gamma\big(\mathcal{A}(\alpha(z))\big),$$

where $\gamma$ is applied to each element of $\mathcal{A}(\alpha(z))$.

Ultimately we wish to prove properties about a particular concrete program $\mathcal{C}$ by reasoning about some simpler abstract program $\mathcal{A}$. From the above definition of an abstraction's concrete semantics we immediately obtain the following criterion for relating a specific concrete program $\mathcal{C}$ to $\mathcal{A}$:

**Definition 2.5. Sound over-approximation.** Let $\mathcal{A}$ be some abstract program with compatible abstraction and

```
1  if(x<0) {
2    x = 0
3  } else {
4    x = x + 1
5  }
```

Figure 2: A simple concrete program over an integer variable x.

concretization functions $\alpha$ and $\gamma$. The tuple $(\mathcal{A}, \alpha, \gamma)$ is a sound over-approximation of $\mathcal{C}$ if for all $z \in \mathcal{D}_{\mathcal{C}}$, $\mathcal{C}(z) \in [\![\mathcal{A}]\!](z)$.

In other words, $\mathcal{A}$ is sound for $\mathcal{C}$ if the result of any concrete execution of $\mathcal{C}$ is contained within the possible concretizations of the result of $\mathcal{A}$ executed on the abstracted input. Sound over-approximations can be used to verify *safety* properties of programs, which intuitively express the fact that certain "bad" things never happen (e.g., no null dereferences will occur). Every safety property can be formalized as a requirement that some set $\mathcal{B}$ of "bad" states in the concrete program never be reached. To prove that $\mathcal{C}(z) \notin \mathcal{B}$ for each concrete state $z$, it suffices to prove that $\gamma([\![\mathcal{A}]\!](a)) \cap \mathcal{B} = \emptyset$ for each abstract state $a \in \mathcal{D}_{\mathcal{A}}$, where $\mathcal{A}$ is a sound over-approximation of $\mathcal{C}$.

In general, the construction of an abstraction is a careful balance between *precision*, the fidelity of the abstraction to the original concrete program, and *tractability*, how difficult the abstraction is to construct and reason about. For abstract predicate domains, adding more predicates to the domain increases precision but also makes the abstraction more costly to produce and analyze.

The semantics above treats programs $\mathcal{C}$ and $\mathcal{A}$ as black-box input-output functions. Nevertheless, the semantics straightforwardly generalizes to assign meaning to every single line of code in the programs, allowing us to establish a sound over-approximation throughout.

## 2.2 PREDICATE ABSTRACTION

A *predicate abstraction* is a well-studied program abstraction whose abstract domain is a predicate domain (Graf and Saïdi, 1997; Ball et al., 2001) (see the previous section for the definition of a predicate domain). Predicate abstractions are known as *Boolean programs*: the domain $\mathcal{D}_{\mathcal{A}} = \{T, F\}^n$. Safety checking in Boolean programs is decidable: a Boolean program has a finite set of states over a fixed number of Boolean variables, making it decidable to obtain the set of reachable states. Given a concrete program $\mathcal{C}$ and a set of $n$ predicates $(p_1, \ldots, p_n)$ over the concrete domain $\mathcal{D}_{\mathcal{C}}$, the goal of the predicate abstraction process is to construct an abstract Boolean program $\mathcal{A}$ that forms a sound over-approximation of $\mathcal{C}$ and is as precise as possible relative to the given predicates.

We use the simple program in Figure 2 as an example to illustrate the predicate abstraction process. The Boolean program induced by the predicates `x<-4` and `x<3` is shown in Figure 3. Following the notation of Ball et al. (2001), the $\star$ operator represents nondeterministic choice, and the Boolean variable associated with predicate $p$ is denoted `{`$p$`}`. We describe the predicate abstraction process for branches and assignments in turn.

```
1  if(*) {
2    assume({x<3})
3    {x<-4}, {x<3} = F, T
4  } else {
5    assume(!{x<-4})
6    {x<-4}, {x<3} =
7       choose(F, !{x<3} ∨ !{x<-4}),
8       choose({x<-4}, !{x<3})
9  }
```

Figure 3: A predicate abstraction of the program in Figure 2 induced by the predicates `x<-4` and `x<3`. Note that predicate updates that are abstractions of the same concrete assignment statement are updated simultaneously.

### 2.2.1 Abstracting Branches

Consider a conditional statement of the form

```
if (p) {···} else {···}
```

in the concrete program. Let $\mathtt{p}^T$ denote the strongest propositional formula over the predicates $p_1, \ldots, p_n$ that is implied by `p` and $\mathtt{p}^F$ denote the strongest propositional formula over the predicates $p_1, \ldots, p_n$ that is implied by `!p`. These formulas represent the most precise information we can know inside the *then* and *else* branches respectively, given the predicates in the abstraction. They can be obtained through queries to an SMT solver, assuming that `p` and the $n$ predicates are all in decidable logical theories; see Ball et al. (2001) for details. The predicate abstraction process translates the above conditional as follows in the Boolean program:

```
if (*) {
  assume({p^T}) ...
} else {
  assume({p^F}) ...
}
```

Here `{`$\mathtt{p}^T$`}` is $\mathtt{p}^T$ but with each predicate $p_i$ replaced by its Boolean counterpart `{`$p_i$`}`, and similarly for `{`$\mathtt{p}^F$`}`. The statement `assume(`$\varphi$`)`, which is standard in the programming languages community, silently ignores executions which do not satisfy $\varphi$. Note that `{`$\mathtt{p}^T$`}` and `{`$\mathtt{p}^F$`}` can simultaneously be true, which allows the execution to nondeterministically take either branch of the conditional.

In the program of Figure 2, we know that `x<0` is true

in the *then* clause. In Figure 3, the strongest information our abstraction can know at that point is that (the Boolean variable corresponding to) `x<3` is true. Similarly, `x<0` is false in the *else* branch in Figure 2, while the abstraction in Figure 3 only knows that `x<-4` is false.

### 2.2.2 Abstracting Assignment Statements

Consider an assignment statement of the form `x = e` in the concrete program. In the corresponding point of the abstract program we must *simultaneously* update the values of all Boolean variables to reflect the update to the value of `x`. Suppose we want to update the variable $\{p_i\}$. Let $p_i^T$ denote the weakest propositional formula over the predicates $p_1, \ldots, p_n$ such that $p_i^T$ holding before the assignment `x = e` suffices to ensure that $p_i$ will be true after the assignment. Similarly let $p_i^F$ denote the weakest propositional formula over the predicates $p_1, \ldots, p_n$ such that $p_i^F$ holding before the assignment `x = e` suffices to ensure that $p_i$ will be false after the assignment. Again an SMT solver can be used to obtain these formulas, leveraging the standard notion of the *weakest precondition* of an assignment statement with respect to a predicate (Dijkstra, 1976). The predicate abstraction process updates the Boolean variable $\{p_i\}$ as follows in the Boolean program:

$\{p_i\}$ = choose($\{p_i^T\}$, $\{p_i^F\}$)

Here choose($\varphi_1$, $\varphi_2$) returns $T$ if $\varphi_1$ is satisfied, otherwise returns $F$ if $\varphi_2$ is satisfied, and otherwise chooses nondeterministically between $T$ and $F$.

Consider the assignment statement `x = 0` in Figure 2. The abstraction process described above will assign `{x<3}` in the Boolean program to `choose(T, F)`, which simplifies to just $T$ as shown in Figure 3. More interestingly, consider the assignment statement `x = x + 1` in Figure 2. If `x<-4` is true before the assignment, then we can be sure that `x<3` is true afterward. If `x<3` is false before the assignment, then we can be sure that `x<3` is false afterward. If neither of these is the case, then the abstraction does not have enough information to know the value of `x<3` after the assignment. Hence in the Boolean program `{x<3}` is assigned to `choose({x<-4}, !{x<3})`.

**Invariants** Multiple predicates that involve the same variable are typically constrained in some way. For example, the predicates `{{x<3}, {x<-4}}` are constrained due to the relationship `{x<-4}`⇒`{x<3}`. This constraint is an invariant which increases the precision of the abstraction with minimal decrease in tractability. We call this constraint $\mathcal{I}$, and we can enforce it simply by inserting an `assume(`$\mathcal{I}$`)` statement after each set of assignments.

### 2.2.3 Proving Program Invariants

A predicate abstraction is a sound over-approximation of the original concrete program. Further, because a Boolean program has a finite set of possible states at each point in the program, it can be exhaustively explored via a form of *model checking*, which conceptually executes the program in all possible ways (Ball and Rajamani, 2000). Model checking produces the set of reachable states at each point in the program, and this information can be used to verify invariants of the original program.

Consider the Boolean program in Figure 3. All executions of this program end in a state where the Boolean variable `{x<-4}` has the value $F$. This implies that `x` always ends in a value greater than or equal to `-4` in the original program in Figure 2. On the other hand, our predicate abstraction is not precise enough to verify that `x` always ends in a nonnegative value, though that is true of the original program. A different choice of predicates would enable such reasoning in the abstraction.

**Selecting predicates** The selection of predicates is clearly a critical component of an effective predicate abstraction. In this work we focus on the definition and construction of probabilistic predicate abstractions given a fixed set of predicates, leaving automated selection of predicates for future work. The programming languages community has developed several approaches to the problem of predicate selection. A common approach is to use a form of *counterexample-driven refinement*, which iteratively adds predicates until the abstraction is precise enough to prove or disprove the desired property of the concrete program (e.g., Ball and Rajamani (2002)). Extending these techniques to the probabilistic context is a challenging and exciting research problem.

## 3 PROBABILISTIC PROGRAM ABSTRACTION

The primary contribution of this paper is the extension of the non-deterministic program abstractions of the previous section to the probabilistic context. We begin by defining a simple probabilistic programming language. Syntactically, our probabilistic predicate abstractions will simply be probabilistic programs in this language. Next, we generalize the abstraction semantics of Section 2.1 to the probabilistic context, and define soundness criteria for probabilistic program abstractions. Finally, we generalize the predicate abstraction process from Section 2.2 to the probabilistic context by placing distributions on the non-deterministic choices.

### 3.1 PROBABILISTIC PROGRAMMING

We define a simple probabilistic programming language, BERN, which contains only (1) Boolean variables; (2) Boolean operators; (3) Boolean assignments; (4) `if` statements; (5) a `flip`($\theta$) operator, which is a Bernoulli random variable with parameter $\theta$; and (6) an `observe`($\varphi$) statement, which ignores executions that do not satisfy some condition $\varphi$. Note that `observe` statements can also be captured by a conditional probability query on the distribution.

An extension to BERN is to introduce a `goto` construct, which would allow it to reason about underlying concrete programs with arbitrary control flow. The predicate abstraction framework makes reasoning about loopy concrete programs tractable (Ball et al., 2001); however, we defer generalizing the semantics of loopy probabilistic predicate abstractions to future work. As an example of a BERN program, one can construct a program that encodes a Bayesian network ⓐ → ⓑ:

```
a = flip(θ1)
if(a) { b = flip(θ2)}
else  { b = flip(θ3)}
observe(b)
```

This probabilistic program defines the conditional probability of each event by utilizing the control-flow features of BERN. For example, $\Pr(b \mid \neg a) = \theta_3$. The `observe` statement conditions the Bayesian network on some evidence: thus, queries about $a$ in this program correspond to $\Pr(a \mid b)$.

Probabilistic programming has proven a natural tool for the construction of generative statistical models. As such, infrastructure for computing queries on probabilistic programs has begun to develop in the AI and programming languages communities (Carpenter et al., 2016; Goodman et al., 2008; Wood et al., 2014; Fierens et al., 2013).

### 3.2 PROBABILISTIC SEMANTICS

Section 2.1 identifies both the abstract and concrete semantics of a program abstraction. We generalize these non-deterministic semantics to probabilistic semantics by producing families of compatible probability distributions described by constraints on their support.

Since syntactically abstractions will be probabilistic programs, the abstract semantics of a probabilistic abstraction are simply the semantics of that program, broadly defined.

**Definition 3.1. Abstract semantics.** Let $a_i, a_o \in \mathcal{D}_\mathcal{A}$. The abstract semantics of a probabilistic abstraction $\mathcal{A}$, denoted $\Pr_\mathcal{A}(a_o \mid a_i)$, is a *conditional probability distribution* over abstract domain $\mathcal{D}_\mathcal{A}$, which describes the probability of transitioning from an initial set of states $a_i$ to an output state $a_o$ under the abstraction $\mathcal{A}$.

To define the concrete semantics of a probabilistic abstraction, we first need to generalize the concretization function $\gamma$ to the probabilistic context.

**Definition 3.2. Concretization distribution.** Let $z \in \mathcal{D}_\mathcal{C}$ and $a \in \mathcal{D}_\mathcal{A}$. A concretization distribution is a *conditional probability distribution* $\Pr_\gamma(z \mid a)$ that describes the probability of concretizing an abstract state $a$ to some concrete state $z$.

In the non-deterministic setting, we were concerned only with membership in the set $\gamma$. Here, we generalized $\gamma$ to the probabilistic context by placing a distribution over possible concretizations.[1] Concretization distributions and abstraction functions are related as follows:

**Definition 3.3. Compatibility.** An abstraction function $\alpha$ and concretization distribution $\Pr_\gamma$ are *compatible* when, for all $z \in \mathcal{D}_\mathcal{C}$, $\Pr_\gamma(z \mid \alpha(z)) > 0$. Furthermore, these functions are *strongly compatible* if they are compatible and for any $a$ and $z$ such that $\Pr_\gamma(z \mid a) > 0$, we have that $\Pr_\gamma(z \mid a') = 0$ for all $a' \neq a$.

We are now in a position to define the concrete semantics of a probabilistic abstraction.

**Definition 3.4. Concrete semantics.** Let $z_i, z_o \in \mathcal{D}_\mathcal{C}$ be some input and output concrete states. The concrete semantics of an abstraction $\mathcal{A}$ given a compatible abstraction function $\alpha$ and concretization distribution $\Pr_\gamma$ is a *conditional probability distribution* describing the probability of transitioning from $z_i$ to $z_o$:

$$\Pr_{[\![\mathcal{A}]\!]}(z_o \mid z_i) = \sum_{a_o \in \mathcal{D}_\mathcal{A}} \Pr_\gamma(z_o \mid a_o) \Pr_\mathcal{A}(a_o \mid \alpha(z_i)).$$

In the case when $\alpha$ and $\Pr_\gamma$ are strongly compatible, we can refine the above definition:

**Proposition 3.1.** Let $z_o, z_i \in \mathcal{D}_\mathcal{C}$. For strongly compatible $\alpha$ and $\Pr_\gamma$, there exists a single $a_o$ for which $\Pr_\gamma(z_o \mid a_o) > 0$. Thus the sum may be collapsed:

$$\Pr_{[\![\mathcal{A}]\!]}(z_o \mid z_i) = \Pr_\gamma(z_o \mid a_o) \Pr_\mathcal{A}(a_o \mid \alpha(z_i)).$$

As an example, we saw previously that predicate domains allow for strongly compatible concretization and abstraction functions. We see in Figure 4 a probabilistic extension to non-deterministic predicate abstraction.

Under the probabilistic semantics, we can define a probabilistic analog of the over-approximation property of $\mathcal{A}$ as a constraint on the support of $\Pr_{[\![\mathcal{A}]\!]}$.

[1] For continuous concrete domains, concretization distributions directly generalize to concretization densities.

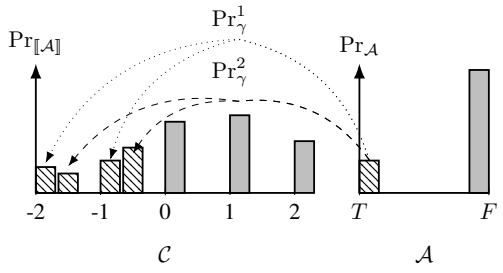


Figure 4: Probabilistic predicate abstraction over domain $\mathcal{D}_\mathcal{A} = \{\{\texttt{x<0}\}\}$. Distribution $\Pr_{[\![\mathcal{A}]\!]}$ over $\mathcal{D}_\mathcal{C}$ is generated by (1) a distribution over abstract states $\Pr_\mathcal{A}$ and (2) one of two concretization distributions: $\Pr^1_\gamma$ or $\Pr^2_\gamma$.

**Definition 3.5. Sound probabilistic over-approximation.** Let $\mathcal{A}$ be a probabilistic program abstraction with compatible abstraction function $\alpha$ and concretization distribution $\Pr_\gamma$. Then the tuple ($\mathcal{A}$, $\alpha$, $\Pr_\gamma$) is a sound probabilistic over-approximation of concrete program $\mathcal{C}$ if for all $z \in \mathcal{D}_\mathcal{C}$, $\Pr_{[\![\mathcal{A}]\!]}(\mathcal{C}(z) \mid z) > 0$.

#### 3.2.1 Non-Deterministic Semantics

A sound probabilistic over-approximation is a generalization of a sound non-deterministic over-approximation in the sense that it provides a distribution over feasible states. Thus a sound probabilistic over-approximation has a corresponding sound non-deterministic over-approximation, which we make precise in the following definitions:

**Definition 3.6. Non-deterministic semantics.** Let $\mathcal{A}$ be a probabilistic program abstraction with compatible concretization distribution $\Pr_\gamma$ and abstraction function $\alpha$. Then there is a corresonding non-deterministic concretization function $\gamma(a)_\downarrow = \{z \mid \Pr_\gamma(z \mid a) > 0)\}$ and abstract non-deterministic program $\mathcal{A}(a)_\downarrow = \{a' \mid \Pr_\mathcal{A}(a' \mid a) > 0)\}$.

We observe that $\gamma(a)_\downarrow$ is compatible with $\alpha$ if $\Pr_\gamma$ is compatible with $\alpha$. Further, soundness of a probabilistic abstraction implies soundness of its corresponding non-deterministic abstraction, and vice versa:

**Theorem 3.1. Non-deterministic sound over-approximation.** For any probabilistic program abstraction $\mathcal{A}$ with compatible concretization distribution $\Pr_\gamma$ and abstraction function $\alpha$, the tuple $(\mathcal{A}, \alpha, \Pr_\gamma)$ is a sound probabilistic over-approximation to concrete program $\mathcal{C}$ if and only if the tuple $(\mathcal{A}(\cdot)_\downarrow, \alpha, \gamma(\cdot)_\downarrow)$ is a sound non-deterministic over-approximation to $\mathcal{C}$.

#### 3.2.2 Concretization Invariance

The concrete semantics $\Pr_{[\![\mathcal{A}]\!]}$ are necessary for reasoning about the concrete domain. However, directly analyzing $\Pr_{[\![\mathcal{A}]\!]}$ is made difficult by the necessity of selecting some compatible concretization distribution $\Pr_\gamma$. Significantly, in the case when a concrete query can be precisely represented using a set of abstract states, $\mathcal{A}$ alone provides sufficient structure to compute a probability in $\Pr_{[\![\mathcal{A}]\!]}$ independent of the choice of $\Pr_\gamma$:

**Theorem 3.2. Concretization distribution invariance.** Let $\mathcal{A}$ be a probabilistic program abstraction with strongly compatible concretization distribution $\Pr_\gamma$ and abstraction function $\alpha$. For any $z_i \in \mathcal{D}_\mathcal{C}$ and $a_o \in \mathcal{D}_\mathcal{A}$,

$$\Pr_{[\![\mathcal{A}]\!]}(a_o|z_i) \stackrel{\text{def}}{=} \sum_{z_o \in \gamma(a_o)_\downarrow} \Pr_{[\![\mathcal{A}]\!]}(z_o|z_i) = \Pr_\mathcal{A}(a_o|\alpha(z_i)).$$

In other words, the probability of an abstracted event occurring in the concrete semantics is equivalent to the probability of that event in the abstract semantics, regardless of the concretization distribution.

We see a visualization of this theorem in Figure 4. Regardless of whether $\Pr^1_\gamma$ or $\Pr^2_\gamma$ are chosen,

$$\begin{aligned}\Pr_{[\![\mathcal{A}]\!]}\left(\gamma(\alpha(x=-1))_\downarrow\right) &= \Pr_{[\![\mathcal{A}]\!]}(\{-1,-2\}) \\ &= \Pr_\mathcal{A}(\{x<0\}).\end{aligned}$$

As a consequence, queries performed on the abstraction $\mathcal{A}$ represent queries performed on the set of all possible strongly-compatible concretization distributions. Thus, even though in the probabilistic setting we must reason about a distribution over concrete states, we can still lift our analyses to the abstract domain, similar to the benefits of non-deterministic abstraction in Section 2.2.3.

### 3.3 PROBABILISTIC PREDICATE ABSTRACTIONS

Thus far we have seen a semantics for a probabilistic program abstraction, but we do not yet have a way to generate one for a particular program. In this section, we seek to generalize predicate abstraction to the probabilistic domain, and show that in general a probabilistic predicate abstraction is a family of Boolean probabilistic programs with Bernoulli `flip` parameters.

#### 3.3.1 Branch Statements

We saw in Section 2.2.1 that a predicate abstraction of an `if` statement is of the form

```
if(*) {assume(α) ... } else {assume(β) ...}
```

where $\alpha$ and $\beta$ represent the most precise information we can know about the state of predicates at the *then* and *else* branches of the program. The behavior of the abstraction is non-deterministic in the case when both $\alpha$ and $\beta$ hold. A probabilistic predicate abstraction of this statement should explicitly quantify the probability of choosing a particular path when either path is possible in the abstraction.

To do so, we first rewrite the predicate abstraction's `if` statement equivalently as follows:

`if(`$\neg\beta \vee (\alpha \wedge *)$`) { ... } else { ... }`

As in the original formulation, this version ensures that the *then* clause will not be taken if $\alpha$ is false and the *else* clause will not be taken if $\beta$ is false.[2] The non-deterministic choice $*$ then determines which path to take when both predicates are true.

A probabilistic predicate abstraction must represent a distribution over paths when $\alpha$ and $\beta$ both hold. Under the semantics of BERN, we may do so simply by replacing the non-deterministic choice with a `flip`:

`if(`$\neg\beta \vee (\alpha \wedge$`flip(`$\theta$`))) { ... } else { ... }`

Thus a probabilistic version of the predicate abstraction in Figure 3 would have an `if` statement with guard `{x<-4}`$\vee$`({x<3}`$\wedge$`flip(`$\theta$`))`, where $\theta$ represents the conditional probability that the branch is taken given $-4 \leq x < 3$. As long as $0 < \theta < 1$, all concrete executions are contained within the support of this probabilistic program abstraction, implying that it is a sound probabilistic over-approximation.

### 3.3.2 Assignment Statements

Section 2.2.2 showed that a concrete assignment is abstracted to a set of predicate assignments of the form $\gamma$ `= choose(`$\alpha$`,` $\beta$`)`, where $\gamma$ is a predicate and $\alpha$ and $\beta$ encode the most precise update we can make to $\gamma$. The abstraction behaves non-deterministically: it may assign $\gamma$ to either `true` or `false` when neither $\alpha$ nor $\beta$ holds. Thus, the probabilistic generalization of an assignment statement needs to represent the conditional probability of $\gamma$ given $\neg\alpha \wedge \neg\beta$.

First, we re-write the `choose` statement, introducing a non-deterministic $*$ operator similar to the previous section. We may write an equivalent update to $\gamma$:

$\gamma$ `=` $\alpha \vee (\neg\beta \wedge *)$

As above, in BERN we then replace $*$ with a Bernoulli random variable:

$\gamma$ `=` $\alpha \vee (\neg\beta \wedge$ `flip(`$\theta$`))`

[2]Note that by construction $\alpha$ and $\beta$ cannot both be false.

For example, under this strategy the assignment statement `x=x+1` from Figure 3 would be abstracted to the following BERN program statements, given predicates `{x<-3}` and `{x<4}`:

```
{x<-4}, {x<3} =
    ({x<-4} ∧ {x<3} ∧ flip(θ1)),
    ({x<-4} ∨ ({x<3} ∧ flip(θ2)))
```

### 3.4 INVARIANTS

In the non-deterministic case, enforcing invariants among predicates is a lightweight procedure of inserting `assume` statements in order to increase the precision of the abstraction. Analogously, in the probabilistic case, we wish to represent distributions over predicates while disallowing inconsistent predicate states. In this section we explore the consequences of enforcing invariants on the abstraction.

An initial approach to enforcing invariants is to straightforwardly generalize the non-deterministic procedure by inserting `observe(`$\mathcal{I}$`)` statements between each assignment, where $\mathcal{I}$ is the invariant which must hold over the predicates. For example, for the concrete program `x=x+10` with the predicates `{x<-4}` and `{x<3}`, we generate the following abstraction:

```
{x<-4}, {x<3} =
     ({x<-4} ∧ {x<3} ∧ flip(θ1)),
     ({x<-4} ∧ {x<3} ∧ flip(θ2))
observe({x<-4}⇒{x<3})
```

A key downside is that the parameters no longer have a *local semantics*: conditioning correlates the otherwise independent flips. This complicates the probability computation, which now involves a partition function.

Therefore we present an alternative abstraction construction procedure which preserves the local semantics of the parameters of the abstraction while enforcing invariants over predicates. Consider again the concrete program `x=x+10`. We generate an abstraction using the same predicates as before. However, instead of simply inserting `observe` statements, we utilize control flow in order to effectively condition on the previously assigned value:

```
{x<3} = {x<3} ∧ {x<-4} ∧ flip(θ1)
if({x<3}) {
  {x<-4} = {x<-4} ∧ flip(θ2)
} else {
  {x<-4} = F
}
```

This abstraction, which we call *structurally dependent*, updates each predicate sequentially, considering all previous decisions. Each concrete statement is abstracted to several abstract statements which utilize control flow to disallow invalid states. The state `{x<-4}`$\wedge$`!{x<3}`

is guaranteed to have 0 probability without the use of `observe` statements. Further, the parameters have a local interpretation as a conditional probability: it is not necessary to compute a partition function to compute the probability of a particular predicate configuration.

Fundamentally, these two methods of constructing the abstraction represent different factorizations of the distribution. In the non-deterministic context with invariant enforcement, these two abstractions are equivalent.

# 4 DISCUSSION

This paper focuses on the definition and key properties of probabilistic program abstractions. In this section we discuss natural next steps for the work. Traditional non-deterministic program abstractions are typically used to produce the set of reachable program states, in order to verify invariants. The analogous operation on a probabilistic program abstraction is inference. First we discuss possible approaches to inference for probabilistic predicate abstractions, by leveraging both model checking and weighted model counting. Second, we discuss how the ability to perform inference on a probabilistic abstraction could be a key enabler for a new approach to performing inference on more general probabilistic programs. The main idea is to reduce inference on a probabilistic program to the task of choosing particular `flip` probabilities for a corresponding probabilistic abstraction.

## 4.1 INFERENCE FOR PROBABILISTIC PREDICATE ABSTRACTIONS

We believe that existing techniques from the programming languages literature which are designed for working with non-deterministic Boolean programs can be extended to perform inference on BERN programs. We can then use weighted model counting to evaluate queries. We note that abstractions allow one to query the marginal probability of an event at any point in the program, not merely upon program termination.

**Probabilistic Model Checking** The problem of computing the set of reachable states in a Boolean program is known as the *model checking problem* and has been extensively studied by the programming languages community. Commonly one represents the set of reachable states at any point in the program as some Boolean knowledge base $\Delta$. In many existing tools, $\Delta$ is represented using a binary decision diagram (Ball and Rajamani, 2000). Inference in BERN is thus an extension to the traditional model checking paradigm in which we introduce weighted variables for the state of each `flip`. During model checking, we treat each `flip` as an unconstrained Boolean variable.

```
1  a = unif [0, 10)
2  if (a < 5) { b = unif [0, 10) }
3    else { b = unif [0, 20) }
4  if (b < 5) { c = unif [0, 10) }
5    else { c = unif [0, 20) }
```

(a) Probabilistic program for Bayesian network ⓐ → ⓑ → ⓒ.

```
{a<5} = flip(1/2)
if({a<5}) { {b<5} = flip(1/2) }
  else { {b<5} = flip(1/4) }
if({b<5}) { {c<5} = flip(1/2) }
  else { {c<5} = flip(1/4) }
```

(b) Probabilistic abstraction with `{a<5}`, `{b<5}`, and `{c<5}`.

Figure 5: A concrete probabilistic program and a probabilistic abstraction for computing $\Pr_{[\![\mathcal{A}]\!]}(c < 5)$.

For example, consider the probabilistic predicate abstraction statement `{x<4} = {x<4} ∧ flip(`$\theta$`)`. We assume $\Delta$ = `{x<4}` prior to execution of statement. Following this statement, $\Delta'$ = `({x<4}∧flip(`$\theta$`)) ∨ (!{x<4} ∧ !flip(`$\theta$`))`. See Ball and Rajamani (2000) for more details.

**Weighted Model Counting** Whereas model checking is usually concerned with determining whether $\mathcal{A}$ can reach a particular state, in probabilistic program inference we are concerned with the weighted sum of reachable states, where the weights are induced by the parameters of the `flip`s in each model. The programming languages community has two primary methodologies for computing the set of reachable states in a Boolean program: (1) knowledge compilation to binary decision diagrams (Ball and Rajamani, 2000), and (2) satisfiability methods (Donaldson et al., 2011). Both of these approaches can be generalized to perform weighted model counting for inference in BERN.

The knowledge compilation approach to model checking is already capable of performing weighted model counting due to the nature of the queries efficiently supported by a binary decision diagram (Darwiche and Marquis, 2001), and is used for inference in discrete probabilistic programs (Fierens et al., 2013) and Bayesian networks (Chavira and Darwiche, 2008). The satisfiability approach to model checking can be extended to perform weighted model counting. This problem is #P-hard (Valiant, 1979), but a number of recent approximation methods have been explored (Chakraborty et al., 2013; Belle et al., 2015b; Zhao et al., 2016); see Gomes et al. (2009) for a survey of the subject.

### 4.2 INFERENCE FOR GENERAL PROBABILISTIC PROGRAMS

Consider the probabilistic program in Figure 5a and suppose we want to evaluate $\Pr_{[\![\mathcal{C}]\!]}(c < 5)$. We will sketch an approach to doing so using probabilistic predicate abstractions.

Figure 5b shows a probabilistic predicate abstraction for our original probabilistic program, induced by the predicates `{a<5}`, `{b<5}`, and `{c<5}`. Initially each `flip` has its own parameter to represent its probability. In the figure, we show particular values for each parameter, which were computed by performing queries on fragments of the original concrete program. For example, the concrete assignment `a = unif[0, 10)` is abstracted to `{a<5} = flip(1/2)` by computing $\Pr(a < 5)$ on this single statement of the concrete program. The other parameters can be learned similarly. The key point is that each of these queries is much easier to evaluate in the original program than the actual query of interest, as they are over smaller fragments of the program.

Now we show that the abstraction captures enough detail to answer our query precisely. Computing the weighted model count using the approach described in the previous subsection, we see that:

$$\Pr_{\mathcal{A}}(\texttt{\{c<5\}}) = \overbrace{0.5 \cdot 0.5 \cdot 0.5}^{\texttt{\{a<5\},\{b<5\},\{c<5\}}} + \overbrace{0.5 \cdot 0.5 \cdot 0.25}^{\texttt{\{a<5\},!\{b<5\},\{c<5\}}} + \underbrace{0.5 \cdot 0.25 \cdot 0.5}_{\texttt{!\{a<5\},\{b<5\},\{c<5\}}} + \underbrace{0.5 \cdot 0.75 \cdot 0.25}_{\texttt{!\{a<5\},!\{b<5\},\{c<5\}}} = \frac{11}{32}.$$

The result is in fact the answer to the original query.

In this way, the inference problem on $\mathcal{C}$ is decomposed into two, potentially much simpler, problems: (i) fixing the parameters of an abstraction, and (ii) weighted model counting on the abstraction. There remains considerable theoretical work to formally connect the semantics of the probabilistic abstraction with a probabilistic concrete program, as well as practical work to realize the benefits of the approach on desired applications.

## 5 RELATED WORK

**Probabilistic reasoning and static analysis.** Several recent works leverage a probabilistic model to guide refinements of a program abstraction (Grigore and Yang, 2016; Zhang et al., 2017). However, the abstractions themselves are not probabilistic. Gehr et al. (2016) use static analysis of a probabilistic program to decompose the problem of inference along paths, which are then dispatched to specialized integration tools depending on the constraints of each path; this work analyzes the original concrete program and does not rely on abstractions.

**Probabilistic abstract interpretation.** Probabilistic abstract interpretation is used to reason about programs with probabilistic semantics, for example to place upper bounds on the probability of a particular path (Monniaux, 2000) or construct Monte-Carlo methods (Monniaux, 2001); this line of work does not explore the connections between abstractions and probabilistic programs, nor does it model concrete program marginals. However, our work does not reason about unbounded loops. The framework of Cousot and Monerau (2012) is a highly general framework for reasoning about programs using probabilistic abstract interpretation; however, they do not consider the abstraction itself to be a statistical model.

**Probabilistic programming systems.** Many systems have been developed within the AI and programming languages communities that tackle the problem of probabilistic program inference, but few utilize abstractions. Systems such as Church (Goodman et al., 2008), Anglican (Wood et al., 2014), Stan (Carpenter et al., 2016), BLOG (Milch et al., 2005), and others directly analyze the concrete program. Weighted model counting and knowledge compilation have been used to perform probabilistic program inference (Fierens et al., 2013); they also do not leverage program abstractions. Several probabilistic inference approaches capture distributions in continuous domains by using Boolean predicates, either as an approximation (Michels et al., 2016) or as an exact representation (Belle et al., 2015a). Finally, program abstraction with the purpose of inference is an instance of approximate lifted inference (Kersting, 2012): the abstract domain groups together sets of concrete states, with the aim of reasoning at the higher level.

## 6 CONCLUSION

Probabilistic program abstractions are currently unexplored territory for aiding in the analysis of programs, despite the popularity of probabilistic programming. We provided a formal framework, derived useful properties, and described probabilistic predicate abstractions techniques. Much theoretical and practical work remains to be done in exploring alternative characterizations, showing relationships between concrete programs and their abstractions, and building practical probabilistic abstraction tools. We hope our framework provides the foundational theory to enable these advances in the future.

#### Acknowledgements

This work is partially supported by NSF grants #CCF-1527923, #IIS-1657613, and #IIS-1633857, and by DARPA grant #N66001-17-2-4032. S.H. is supported by a National Physical Sciences Consortium Fellowship.

## References


T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Model Checking and Software Verification*, pages 113–130, 2000.

T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of POPL*, pages 1–3, 2002.

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

V. Belle, A. Passerini, and G. Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proc. of IJCAI*, pages 2770–2776, 2015a.

V. Belle, G. Van den Broeck, and A. Passerini. Hashing-based approximate probabilistic inference in hybrid domains. In *Proc. of UAI*, pages 141–150, 2015b.

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

S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Proc. of CP*, pages 200–216, 2013.

M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *J. Artificial Intelligence*, 172(6-7):772–799, Apr. 2008.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL*, pages 238–252, 1977.

P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *Proc. of ESOP*, pages 169–193, 2012.

A. Darwiche and P. Marquis. A Knowledge Compilation Map. *Proc. of IJCAI*, 17:175–182, 2001.

L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS/ETAPS*, pages 337–340, Berlin, Heidelberg, 2008.

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

A. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 356–371. Springer, 2011.

D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *J. Theory and Practice of Logic Programming*, 15(3): 358 – 401, 2013.

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

C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.

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

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

R. Grigore and H. Yang. Abstraction Refinement Guided by a Learnt Probabilistic Model. *Proc. of POPL*, pages 485–498, 2016.

K. Kersting. Lifted probabilistic inference. In *Proc. of ECAI*, pages 33–38, 2012.

S. Michels, A. Hommersom, and P. J. F. Lucas. Approximate probabilistic inference with bounded error for hybrid probabilistic logic programming. In *Proc. of IJCAI*, pages 3616–3622, 2016.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *Proc. of IJCAI*, pages 1352–1359, 2005.

D. Monniaux. Abstract interpretation of probabilistic semantics. In *International Symposium on Static Analysis*, pages 322–339, 2000.

D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. *SIGPLAN Not.*, 36 (3):93–101, Jan. 2001.

L. G. Valiant. The complexity of computing the permanent. *J. Theoretical Computer Science*, 8:189–201, 1979.

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

X. Zhang, X. Si, and M. Naik. Combining the logical and the probabilistic in program analysis. In *Proc. ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 27–34, 2017.

S. Zhao, S. Chaturapruek, A. Sabharwal, and S. Ermon. Closing the gap between short and long xors for model counting. In *Proc. of AAAI*, pages 3322–3329, 2016.