Probabilistic Program Abstractions

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Steven John Holtzen
Abstraction is a fundamental tool for reasoning about a complex system. Program abstraction has been utilized to great effect for analyzing deterministic programs. At the heart of a program abstraction is a connection between the abstract program, which is simple to analyze, and the concrete program, which may be extremely complex. Program abstractions, however, are typically not probabilistic. In this thesis I generalize a particular class of non-deterministic program abstractions known as sound over-approximations to the probabilistic context. Sound over-approximations are a family of abstract programs which are guaranteed to contain the original program as a subset of their behavior. This thesis shows that when imbued with a probabilistic semantics, sound over-approximations define a family of probabilistic programs which capture key properties of the original program. It then introduces a mechanism for generating sound probabilistic over-approximations as a generalization of a well-known program abstraction technique known as predicate abstraction. Finally, the problem of inference and learning in the context of probabilistic program abstractions are briefly described.
The thesis of Steven John Holtzen is approved.

Adnan Darwiche

Guy Van den Broeck, Committee Co-chair

Todd D. Millstein, Committee Co-chair

University of California, Los Angeles

2017
# Table of Contents

1 Introduction & Background ............................................. 1  
  1.1 Non-Deterministic Program Abstractions .......................... 1  
    1.1.1 Semantics and Properties of Program Abstractions .......... 2  
    1.1.2 Predicate Abstraction ......................................... 4  
  1.2 Probabilistic Programming Languages ............................ 6  
    1.2.1 A Survey of Probabilistic Programming Systems .......... 7  
    1.2.2 Probabilistic Program Semantics ............................. 7  
    1.2.3 A Bernoulli Probabilistic Programming Language .......... 7  

2 Probabilistic Program Abstractions ................................. 9  
  2.1 Probabilistic Semantics ............................................ 9  
    2.1.1 Non-Deterministic Semantics ................................. 10  
    2.1.2 Concretization Invariance ................................... 11  

3 Probabilistic Predicate Abstractions ............................... 13  
  3.1 Generating Probabilistic Predicate Abstractions ............... 13  
    3.1.1 Branch Statements ............................................ 13  
  3.2 Assignment Statements ............................................ 14  
    3.2.1 Predicate Constraints ......................................... 14  

4 Inference & Learning .................................................. 16  
  4.1 Inference .......................................................... 16  
    4.1.1 Model Checking ............................................... 16  
    4.1.2 Weighted Model Counting ..................................... 17  
  4.2 Learning .......................................................... 18  
    4.2.1 Branch Probabilities .......................................... 18  
    4.2.2 Assignment Probabilities ..................................... 18  
    4.2.3 Marginal Matching ............................................. 19  

5 Conclusion & Future Work ............................................. 20
LIST OF FIGURES

1.2 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 $(x < 0)$. For example, $\alpha(-1) = T$, and $\gamma(T) = \{-2, -1\}$. .................................................. 2

1.3 A concrete program and its associated predicate abstraction. .......................... 4

2.1 A probabilistic predicate abstraction over the domain $D_A = \{x<0\}$. The distribution $\Pr[A]$ over $D_C$ is generated by (1) a distribution over abstract states $\Pr_A$ and (2) one of two concretization distributions: $\Pr_1$ or $\Pr_2$. ................................. 11

3.1 A concrete program and its associated probabilistic predicate abstraction. 15
**Notation**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{D}_C$</td>
<td>Concrete domain, elements of which are denoted $z$</td>
</tr>
<tr>
<td>$\mathcal{D}_A$</td>
<td>Abstract domain, elements of which are denoted $a$</td>
</tr>
<tr>
<td>$\mathcal{C}$</td>
<td>Concrete semantics, $\mathcal{C} : \mathcal{D}_C \rightarrow \mathcal{D}_C$</td>
</tr>
<tr>
<td>$\mathcal{A}$</td>
<td>Abstract semantics of abstraction, $\mathcal{A} : \mathcal{D}_A \rightarrow 2^{\mathcal{D}_A}$</td>
</tr>
<tr>
<td>$[\mathcal{A}]$</td>
<td>Concrete semantics of abstraction, $[\mathcal{A}] : \mathcal{D}_C \rightarrow 2^{\mathcal{D}_C}$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Abstraction function, $\alpha : \mathcal{D}_C \rightarrow \mathcal{D}_A$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Concretization function, $\gamma : \mathcal{A} \rightarrow 2^{\mathcal{D}_C}$</td>
</tr>
<tr>
<td>$\text{Pr}_A(a_o \mid a_i)$</td>
<td>Abstract semantics of probabilistic abstraction, probability of transitioning from abstract state $a_i$ to $a_o$</td>
</tr>
<tr>
<td>$\text{Pr}_{[\mathcal{A}]}(z_o \mid z_i)$</td>
<td>Concrete semantics of probabilistic abstraction, probability of transitioning from concrete state $z_i$ to $z_o$ according to the abstraction $[\mathcal{A}]$</td>
</tr>
<tr>
<td>$\text{Pr}_{[\mathcal{C}]}(z_o \mid z_i)$</td>
<td>Concrete semantics of probabilistic concrete program, probability of transitioning from concrete state $z_i$ to $z_o$ according to a probabilistic concrete program</td>
</tr>
<tr>
<td>$\text{Pr}_\gamma(z \mid a)$</td>
<td>Concretization distribution, probability of concretizing an abstract state $a$ to some concrete state $z$</td>
</tr>
<tr>
<td>$\mathcal{A}(a_i)_\downarrow$</td>
<td>Non-deterministic semantics of probabilistic abstraction, $\mathcal{A}(a_i)_\downarrow = { a \mid \text{Pr}_A(a \mid a_i) &gt; 0 }$</td>
</tr>
<tr>
<td>$\gamma(a)_\downarrow$</td>
<td>Non-deterministic semantics of probabilistic concretization distribution, $\gamma(a)<em>\downarrow = { z \mid \text{Pr}</em>\gamma(z \mid a) &gt; 0 }$</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction & Background

Program abstractions are a richly studied method from the programming languages community for reasoning about intractably complex programs (P. Cousot and R. Cousot, 1977). One sort of abstraction is an over-approximation to a program: any execution which 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. Abstractions have many highly desirable properties: they are tunable and so can be iteratively refined to capture more of the original program. However, abstractions are decidedly non-probabilistic: they are concerned with the possible, not the probable.

A separate effort in the artificial intelligence community seeks to utilize program semantics to generate statistical models. These probabilistic programs define a probability distribution through their semantics: for example, by including an operator such as \( \text{flip}(\theta) \), a Boolean variable which is true with some probability \( \theta \). Examples include Stan (Carpenter et al., 2016), Church (Goodman et al., 2008), and Anglican (Wood, Meent, and Mansinghka, 2014). Similar to other statistical models such as Bayesian networks, the goal of a probabilistic program is to be an accurate surrogate for the true generating process for some observed data.

This thesis explores the deep connections between probabilistic programs and program abstractions. As an overview of this thesis, it begins in the following sections with a review of existing techniques for program abstraction and reasoning about probabilistic programs. I describe over-approximate program abstractions and introduce predicate abstractions, a particular kind of over-approximate abstraction. I discuss existing probabilistic programs and their associated semantics. In Chapter 2 a general theory of probabilistic program abstractions is introduced which generalizes non-deterministic over-approximate program abstractions to the probabilistic context. In Chapter 3 I introduce a particular family of probabilistic over-approximations known as probabilistic predicate abstractions. In Chapter 4 I describe how to perform learning and inference in probabilistic predicate abstractions.

1.1 Non-Deterministic Program Abstractions

Program abstractions are a powerful tool for reasoning about complex software. This section describes the semantics and properties of an over-approximate abstraction and provides an example of a particular class of over-approximations known as predicate abstractions.

Over-approximate program abstractions are well-studied within the field in programming language analysis. Over-approximate abstractions have been used for a tremendous variety of
Figure 1.2: 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 ($x < 0$). For example, $\alpha(-1) = T$, and $\gamma(T) = \{-2, -1\}$.

purposes in program analysis, such as the verification of C programs (Ball, Majumdar, et al., 2001), a variety of compiler optimizations (Kennedy and Allen, 2002), and countless other applications (P. Cousot and R. Cousot, 2014; Clarke et al., 2005; McMillan, 2006; Henzinger, Jhala, and Majumdar, 2005). Program abstractions focus on analyzing a deterministic program and establishing whether or not a property holds on all executions of that program. The following subsections define the formal semantics of program abstractions and predicate abstractions.

1.1.1 Semantics and Properties of Program Abstractions

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

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 Theories (SMT) solvers (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 (P. Cousot and R. Cousot, 1977) provides a general technique for relating a concrete program $\mathcal{C}$ to another program $\mathcal{A}$ which is referred to as an abstraction. The following definition describes a specialization of the more general abstract interpretation framework.

Definition 1.1.1. Abstract semantics of an abstraction. The abstract semantics of an abstraction $\mathcal{A}$, which for simplicity is also denoted $\mathcal{A}$, is a function from input states to sets of output states over an abstract domain $\mathcal{D}_A$, written $\mathcal{A} : \mathcal{D}_A \rightarrow 2^{\mathcal{D}_A}$.

The nondeterminism in the abstract semantics of an abstraction represents uncertainty due to the loss of information in abstracting $\mathcal{C}$ to $\mathcal{A}$. This non-determinism is represented as a set of possible abstract states. The following two mappings between concrete and abstract states relate concrete programs with abstractions.

Definition 1.1.2. Abstraction and concretization functions. An abstraction function for $\mathcal{D}_C$ and $\mathcal{D}_A$ is a function $\alpha : \mathcal{D}_C \rightarrow \mathcal{D}_A$ that maps each concrete state to its abstract
A concretization function for \( D_C \) and \( D_A \) is a function \( \gamma : D_A \rightarrow 2^{D_C} \) 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 as follows:

**Definition 1.1.3. Compatibility.** An abstraction function \( \alpha \) and concretization function \( \gamma \) are compatible if \( z \in \gamma(\alpha(z)) \) for all \( z \in D_C \). As an extension, the two functions are strongly compatible if they are compatible and for any \( a \) and \( z \in \gamma(a) \), it is the case that \( z \notin \gamma(a') \) for any \( a' \neq a \).

**Example 1.1.1. Predicate domains.** 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 \( D_A \) 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. 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: \( \{ z \in D_C \mid (p_1(z), \ldots, p_n(z)) = a \} \). The functions \( \alpha \) and \( \gamma \) are strongly compatible for the case of predicate domains.

For example, suppose \( D_C \) 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, respectively representing the concrete states where \( (x < 0) \) is true and false. See Figure 1.2 for a visualization.

Intuitively, an abstraction represents a set of possible concrete programs. The following definition formalizes this notion:

**Definition 1.1.4. Concrete semantics of an abstraction.** The concrete semantics of an abstraction \( A \), given compatible abstraction and concretization functions \( \alpha \) and \( \gamma \), is a function \( [A] : D_C \rightarrow 2^{D_C} \) defined as follows:

\[
[A](z) = \gamma(\alpha(z)).
\]

Ultimately one wishes to prove properties about a particular concrete program \( C \) by reasoning about some tractable \( A \). From the above definition of an abstraction’s concrete semantics one immediately obtains the following criterion for relating a specific concrete program \( C \) to \( A \):

**Definition 1.1.5. Sound over-approximation.** Let \( A \) be some abstract program with compatible abstraction and concretization functions \( \alpha \) and \( \gamma \). The tuple \( (A, \alpha, \gamma) \) is a sound over-approximation of \( C \) if for all \( z \in D_C \), \( C(z) \in [A](z) \).

In other words, \( A \) is sound for \( C \) if the result of any concrete execution of \( C \) is contained within the possible concretizations of the result of \( A \) 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 \( B \) of “bad” states in
if\( (x < 0) \) {
  x = 0
} else {
  x = x + 1
}

(a) A simple concrete program over an integer variable \( x \).

if(*) {
  assume({x < 3})
  {x < -4} = F
  {x < 3} = T
} else {
  assume(!{x < -4})
  {x < -4} = choose (F, !{x < 3} \lor !{x < -4})
  {x < 3} = choose({x < -4}, !{x < 3})
}

(b) A predicate abstraction of the program in Figure 1.3a induced by the predicates \( x < -4 \) and \( x < 3 \).

Figure 1.3: A concrete program and its associated predicate abstraction.

The concrete program never be reached. To prove that \( C(z) \not\in B \) for each concrete state \( z \), it suffices to prove that \( \gamma(\mathcal{A}(a)) \cap B = \emptyset \) for each abstract state \( a \in \mathcal{D}_A \), where \( \mathcal{A} \) is a sound over-approximation of \( C \).

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

1.1.2 Predicate Abstraction

A predicate abstraction is a well-studied program abstraction whose abstract domain is a predicate domain (Graf and Saïdi, 1997; Ball, Majumdar, et al., 2001). See Ex. 1.1.1 above for the definition of a predicate domain. Predicate abstractions are known as Boolean programs: the domain \( \mathcal{D}_A = \{T, F\}^n \). Checking safety properties of Boolean programs is decidable: they have a finite set of states over a fixed number of Boolean variables. The goal of the predicate abstraction process is to construct an abstract program \( \mathcal{A} \) for a given concrete program \( C \) that satisfies Def. 1.1.5, given a set of \( n \) predicates \( (p_1, \ldots, p_n) \) over the concrete domain \( \mathcal{D}_C \).

The simple program in Figure 1.3a is an example to illustrate the predicate abstraction process. The Boolean program induced by the predicates \( x < -4 \) and \( x < 3 \) for this example is shown in Figure 1.3b. Following the notation of Ball, Majumdar, et al. (2001), the \( * \) operator represents nondeterministic choice, and the Boolean variable associated with a predicate \( p \) is denoted \( \{p\} \). In the following sections the predicate abstraction process for branches and assignment statements are described in turn.

1.1.2.1 Abstracting Branches

Consider a conditional statement of the form

\[
\text{if } (p) \{ \cdots \} \text{ else } \{ \cdots \}
\]

in the concrete program. Let \( p^T \) denote the strongest propositional formula over the predi-
cates $p_1, \ldots, p_n$ that is implied by $p$ and $p^F$ 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 one 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 (Ball, Majumdar, et al., 2001). The predicate abstraction process translates the above conditional as follows in the Boolean program:

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

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

In the program of Figure 1.3a, then $x<0$ is true in the then clause. In Figure 1.3b, the strongest information the 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 1.3a, while the abstraction in Figure 1.3b only knows that $x<-4$ is false.

### 1.1.2.2 Abstracting Assignment Statements

Consider an assignment statement of the form $x = e$ in the concrete program. In the corresponding point of the abstract program the values of all Boolean variables must be updated to reflect the update to the value of $x$. Suppose the variable $\{p_i\}$ is to be updated. Let $p^T_i$ denote the weakest propositional formula over the predicates $p_1, \ldots, p_n$ such that $p^T_i$ holding before the assignment $x = e$ suffices to ensure that $p_i$ will be true after the assignment. Similarly let $p^F_i$ denote the weakest propositional formula over the predicates $p_1, \ldots, p_n$ such that $p^F_i$ 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 program statement with respect to a predicate (Dijkstra, 1976). The predicate abstraction process updates the Boolean variable $\{p_i\}$ as follows in the Boolean program:

$$\{p_i\} = \text{choose}(\{p^T_i\}, \{p^F_i\})$$

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 1.3a. 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 1.3b. More interestingly, consider the assignment statement $x=x+1$
in Figure 1.3a. If \(x<-4\) is true before the assignment, then one can be sure that \(x<3\) is true afterward. If \(x<3\) is false before the assignment, then one 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 \texttt{choose(\{x<4\}, \neg\{x<3\})}.

1.1.2.3 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 1.3b. 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 1.3a. On the other hand, the example 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.

1.2 Probabilistic Programming Languages

Probabilistic programs are typically viewed in the artificial intelligence community as statistical models, such as in Stan (Carpenter et al., 2016), Church (Goodman et al., 2008), and Anglican (Wood, Meent, and Mansinghka, 2014). The overall goals of these systems are largely to (i) ease the development of increasingly sophisticated models; (ii) decouple the modeling and inference problem for a wide class of statistical frameworks; (iii) provide a richer modeling framework than existing techniques.

Parallel to the efforts of the machine learning and artificial intelligence communities, the programming languages community has been developing techniques for reasoning about probabilistic systems, for example to prove correctness properties (Audebaud and Paulin-Mohring, 2009) and guide abstraction refinements (Grigore and Yang, 2016). The programming languages community provides a careful calculus for the definition of a probabilistic programming language based on combining standard programming language semantics with measure transformer semantics (Borgström, Gordon, et al., 2013), which for example allows one to define a formal correspondence between a probabilistic program and a probabilistic graphical model. The following subsections provide a high-level survey of existing probabilistic programming systems, and provide an informal semantics of a simple Bernoulli probabilistic programming language.
1.2.1 A Survey of Probabilistic Programming Systems

Broadly there are two categories of probabilistic programming languages within the machine learning and artificial intelligence community, which are primarily distinguished by the tradeoffs they make between expressivity and tractability.

The first category seeks to design a programming language which may be compiled into a different tractable representation with an equivalent semantics. For example, the language Figaro (Pfeffer, 2009) compiles a specification embedded in the programming language Scala to a graphical model; the languages FACTORIE (McCallum, Schultz, and Singh, 2009) and Infer.NET compile to a factor graph representation. The language STAN (Carpenter et al., 2016) is more general, but restricts the model space to continuously differentiable functions in order to apply various black-box inference algorithms. Probabilistic logic programs such as Problog (Vlasselaer et al., 2015) are compiled to weighted Boolean formulae, which are represented as sentential decision diagrams, a representation in which certain classes of statistical queries are tractable.

Another class of probabilistic programming languages do not restrict the semantics of the programming language to enforce the ability to compile to a tractable representation. These more general languages universally rely on variants of Monte-Carlo estimation to perform inference. Examples include Church (Goodman et al., 2008), Anglican (Wood, Meent, and Mansinghka, 2014), Venture (Mansinghka, Selsam, and Perov, 2014), Quicksand (Ritchie, 2014), PyMC3 (Salvatier, Wiecki, and Fonnesbeck, 2016), and others.

1.2.2 Probabilistic Program Semantics

Recently there has been an effort within the programming languages community to formalize the semantics of probabilistic programming languages (Borgström, Lago, et al., 2016; Claret et al., 2013; Borgström, Gordon, et al., 2013). In general, these approaches rely on a measure-transformer semantics, which augment a typical small-step or big-step program semantics with probabilities for transitioning between states. In this thesis, focus is placed on the semantics of discrete loop-free probabilistic programs, which require less subtlety in their definitions.

This thesis considers a conditional probability distribution over program executions, specifically \( \Pr_{[C]}(z_o \mid z_i) \) is the probability of transitioning from initial state \( z_i \) to output state \( z_o \) under the probabilistic transition semantics defined by the probabilistic program \([C]\). Thus a probabilistic program defines a conditional probability distribution over output states given input states.

1.2.3 A Bernoulli Probabilistic Programming Language

This thesis defines a simple probabilistic programming language, BERN, which contains only (1) Boolean variables; (2) Boolean assignments; (3) if-statements; and (4) a \texttt{flip}(\theta) operator, which is a Bernoulli random variable with parameter \( \theta \). Many existing probabilistic programming languages include within the semantics of the language an \texttt{observe} statement, which ignores executions that do not satisfy some condition. 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 allows one to elegantly reason about concrete programs with loops using fix-points (Ball, Majumdar, et al., 2001; P. Cousot and R. Cousot, 1977); however, reasoning about the semantics of probabilistic predicate abstractions of programs with loops is deferred to future work.

As an example of a probabilistic program in BERN, one could write a program such as:

```plaintext
Burglary = flip(0.2)
if (Burglary) {
    JohnCalls = flip(0.4)
} else {
    JohnCalls = flip(0.01)
}
observe(JohnCalls)
```

This probabilistic program defines a simple relationship between the event Burglary and the dependent event JohnCalls. This program would allow one to query for Pr(Burglary | JohnCalls).
CHAPTER 2
Probabilistic Program Abstractions

The primary contribution of this thesis is the extension of the non-deterministic program abstractions of the previous chapter to the probabilistic context. First the abstraction semantics of Section 1.1.1 are extended to the probabilistic context, and soundness criteria for probabilistic program abstractions are defined. Next, the predicate abstraction process from Section 1.1.2 is extended to the probabilistic context by placing distributions on the non-deterministic choices.

2.1 Probabilistic Semantics

Section 1.1.1 identifies both the abstract and concrete semantics of a program abstraction. The non-deterministic semantics are generalized to probabilistic semantics by describing families of probability distributions which have a probabilistic analog of sound over-approximation. Syntactically, abstractions are probabilistic programs, so the abstract semantics of a probabilistic abstraction is simply the semantics of that program.

Definition 2.1.1. Abstract semantics. Let \( a_i, a_o \in D_\mathcal{A} \). The abstract semantics of a probabilistic abstraction \( \mathcal{A} \), denoted \( \Pr_{\mathcal{A}}(a_o | a_i) \), is a conditional probability distribution over abstract domain \( D_\mathcal{A} \), which describes the probability of transitioning from an initial state \( a_i \) to an output state \( a_o \) under the abstraction \( \mathcal{A} \).

Next, the concretization function \( \gamma \) is generalized to the probabilistic context:

Definition 2.1.2. Concretization distribution. Let \( z \in D_C \) and \( a \in D_\mathcal{A} \). A concretization distribution is a conditional probability distribution \( \Pr_{\gamma}(z | a) \) that describes the probability of concretizing an abstract state \( a \) to some concrete state \( z \).

Only with membership in the set \( \gamma \) was relevant in the non-deterministic setting. Here, \( \gamma \) is generalized to the probabilistic context by placing a distribution over possible concretizations.\(^1\) Concretization distributions and abstraction functions are related as follows:

Definition 2.1.3. Compatibility. An abstraction function \( \alpha \) and concretization distribution \( \Pr_{\gamma} \) are compatible when, for all \( z \in D_C \), \( \Pr_{\gamma}(z | \alpha(z)) > 0 \). Furthermore, these functions are strongly compatible if they are compatible and for any \( a \) and \( z \) such that \( \Pr_{\gamma}(z | a) > 0 \), it is the case that \( \Pr_{\gamma}(z | a') = 0 \) for all \( a' \neq a \).

Now the concrete semantics of a probabilistic abstraction may be defined:

\(^1\)For continuous concrete domains, concretization distributions directly generalize to concretization densities.
Definition 2.1.4. Concrete semantics. Let $z_i, z_o \in D_C$ be some input and output concrete states. The concrete semantics of an abstraction $A$, 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 $z_o$:

$$\Pr[A](z_o \mid z_i) = \sum_{a_o \in D_A} \Pr_\gamma(z_o \mid a_o) \Pr_A(a_o \mid \alpha(z_i)).$$

In the case when $\alpha$ and $\Pr_\gamma$ are strongly compatible, the above definition may be refined:

Proposition 2.1.1. Let $z_o, z_i \in D_C$. 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:

$$\Pr[A](z_o \mid z_i) = \Pr_\gamma(z_o \mid a_o) \Pr_A(a_o \mid \alpha(z_i)).$$

As an example, it was shown previously that predicate domains allow for strongly compatible concretization and abstraction functions. Figure 2.1 shows a probabilistic extension to a non-deterministic predicate abstraction over a concrete program which takes no inputs. The probability of any concrete state (e.g. $x=-1$) is determined by the probability of that concrete state’s sole corresponding abstract state and concretization distribution (e.g. $\Pr_A(\{x<0\})$ and $\Pr_\gamma(x = -1 \mid \{x < 0\})$).

Under the probabilistic semantics, a probabilistic analog of the over-approximation property of $A$ may be defined as a constraint on $\Pr[A]$:

Definition 2.1.5. Sound probabilistic over-approximation. Let $A$ be a probabilistic program abstraction with compatible abstraction function $\alpha$ and concretization distribution $\Pr_\gamma$. Then the tuple $(A, \alpha, \Pr_\gamma)$ is a sound probabilistic over-approximation of concrete program $C$ if for all $z \in D_C$, $\Pr[A](C(z) \mid z) > 0$.

2.1.1 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 there exists a direct connection between a sound probabilistic over-approximation and a corresponding sound non-deterministic over-approximation by considering states with a non-zero probability, which are explored in the following definitions:

Definition 2.1.6. Non-deterministic semantics. Let $A$ be a probabilistic program abstraction with compatible concretization distribution $\Pr_\gamma$ and abstraction function $\alpha$. Then there is a corresponding non-deterministic concretization function $\gamma(a)_\downarrow = \{z \mid \Pr_\gamma(z \mid a) > 0\}$, and abstract non-deterministic program $A(a)_\downarrow = \{a' \mid \Pr_A(a' \mid a) > 0\}$.

It is clear from the above definition that $\gamma(a)_\downarrow$ is compatible with $\alpha$ if $\Pr_\gamma$ is compatible with $\alpha$. The criteria for a sound probabilistic over-approximation are the criteria for a sound over-approximation in the non-deterministic semantics:
Figure 2.1: A probabilistic predicate abstraction over the domain $\mathcal{D}_A = \{x < 0\}$. The distribution $\Pr_{[A]}$ over $\mathcal{D}_C$ is generated by (1) a distribution over abstract states $\Pr_{A}$ and (2) one of two concretization distributions: $\Pr^1_{\gamma}$ or $\Pr^2_{\gamma}$.

**Theorem 2.1.1. 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 $\mathcal{C}$.

**2.1.2 Concretization Invariance**

The concrete semantics $\Pr_{[A]}$ are necessary for reasoning about the concrete domain. However, directly analyzing $\Pr_{[A]}$ 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 $\Pr_{[A]}$ independent of the choice of $\Pr_{\gamma}$:

**Theorem 2.1.2. Concretization distribution invariance.** Let $\mathcal{A}$ be a probabilistic program abstraction with strongly compatible concretization distribution $\Pr_{\gamma}$ and abstraction function $\alpha$. For any $z_i \in \mathcal{D}_C$ and $a_o \in \mathcal{D}_A$,

$$\sum_{z_o \in \gamma(a_o)\downarrow} \Pr_{[A]}(z_o|z_i) = \Pr_{A}(a_o|\alpha(z_i)).$$

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.
Proof.

\[
\sum_{z_o \in \gamma(a_o)} \Pr_{\mathcal{A}[\bar{\gamma}]}(z_o | z_i) \\
= \sum_{z_o \in \gamma(a_o)} \sum_{a'_o \in \mathcal{D}_A} \Pr_\gamma(z_o | a'_o) \Pr_\mathcal{A}(a'_o | \alpha(z_i)) \\
= \sum_{a'_o \in \mathcal{D}_A} \Pr_\mathcal{A}(a'_o | \alpha(z_i)) \sum_{z_o \in \gamma(a_o)} \Pr_\gamma(z_o | a'_o) \\
\text{1 if } a'_o = a_o, \text{ 0 otherwise} \\
= \Pr_\mathcal{A}(a_o | \alpha(z_i)).
\]

Figure 2.1 shows a visualization of this theorem. Regardless of whether or not \(\Pr_{\gamma_1}^1\) or \(\Pr_{\gamma_2}^2\) are chosen,

\[
\Pr_{\mathcal{A}[\bar{\gamma}]}(\gamma(\alpha(x = -1))) = \Pr_{\mathcal{A}[\bar{\gamma}]}((-1, -2)) = \Pr_\mathcal{A}(\{x < 0\}).
\]

As a consequence, queries performed on the abstraction \(\mathcal{A}\) represent queries performed on the many possible strongly-compatible concretization densities. Thus, even though in the probabilistic setting one must reason about a distribution over concrete states, one can still lift analyses on the concrete domain to the abstract domain, similar to the benefits of non-deterministic abstraction in Section 1.1.2.3.
CHAPTER 3

Probabilistic Predicate Abstractions

Thus far a semantics of probabilistic program abstraction has been developed. This chapter generalizes predicate abstraction to the probabilistic context in order to algorithmically generated a probabilistic program abstraction for a desired concrete program.

3.1 Generating Probabilistic Predicate Abstractions

Chapter 1 showed that one can generate an abstraction by separately abstracting a concrete program’s branch and assignment statements. This procedure can be generalized to generate probabilistic predicate abstractions.

3.1.1 Branch Statements

Section 1.1.2.1 showed that a predicate abstraction of an if-statement is of the form
\[ \text{if}(*)\{\text{assume}(\alpha)\} \text{else}\{\text{assume}(\beta)\}, \]
where \( \alpha \) and \( \beta \) represent the most precise information one 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.

Consider the predicate abstraction shown in Figure 3.1. The concrete program is of the form
\[ \text{if}(x<0)\{\ldots\}\text{else}\{\ldots\}. \]
The corresponding predicate abstraction with the predicates \{x<3\} and \{x<-4\} is of the form
\[ \text{if}(*)\{\text{assume}((x<3)\ldots)\} \text{else}\{\text{assume}(!\{x<-4\})\ldots\}. \]
Intuitively, this means that if the then branch is taken, then \( x<3 \) since \( x<0 \); if the else branch is taken, then \( x\geq -4 \) since \( x\geq 0 \).

An abstract if-statement with a concrete guard \( \gamma \) in turn may be rewritten in an alternative and equivalent way. Note that \( \gamma \Rightarrow \alpha \) and \( \neg\gamma \Rightarrow \beta \). Thus one can remove the assume statements by generating a new guard condition in terms of \( \alpha \) and \( \beta \), with a non-deterministic \( * \) which determines which path to take when \( \alpha \land \beta \) holds:
\[ \text{if}(\neg\beta \lor (\alpha \land *)) \{ \ldots \} \text{else} \{ \ldots \} \]
Extending the running example, the equivalent guard may be written as
\[ \text{if}((x<-4)\lor((x<3)\land *))\].
The abstraction is not precise enough to represent the behavior of the concrete program when \( x=2 \); either path must be permissible in this case for the abstraction to remain an over-approximation of the concrete program.

Thus, a probabilistic predicate abstraction must represent a distribution over paths when \( \alpha \land \beta \). Under the semantics of BERN, one may write a probabilistic program which encodes
such a distribution:

\[
\text{if}(\neg \beta \lor (\alpha \land \text{flip}(\theta))) \{ \ldots \} \text{ else } \{ \ldots \}
\]

Thus a probabilistic predicate abstraction of the running example is

\[
\text{if}(\{x < -4\} \lor (\{x < 3\} \land \text{flip}(\theta)))
\]

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 have a non-zero probability in the probabilistic program abstraction, implying it is a sound probabilistic over-approximation.

## 3.2 Assignment Statements

Section 1.1.2.2 showed that a concrete assignment is abstracted to a set of predicate assignments of the form \(\gamma = \text{choose}(\alpha, \beta)\), where \(\gamma\) is a predicate and \(\alpha\) and \(\beta\) encode the most precise update one can make to \(\gamma\). The abstraction behaves non-deterministically: it may assign \(\gamma\) to either \text{true} or \text{false} when \(\neg \alpha \land \neg \beta\) holds. Thus, the probabilistic generalization of an assignment statement needs to represent the conditional probability of \(\gamma\) given \(\neg \alpha \land \neg \beta\).

First, the \text{choose} statement must be rewritten by introducing a non-deterministic * operator similar to the previous section. Now an equivalent update to \(\gamma\) may be written:

\[
\gamma = \alpha \lor (\neg \beta \land *)
\]

A simple way to represent a probability distribution over the set of non-deterministic outcomes for this statement in BERN is to replace the * with a Bernoulli random variable:

\[
\gamma = \alpha \lor (\neg \beta \land \text{flip}(\theta))
\]

For example, under this replacement strategy the abstraction of the concrete statement \(x=x+1\) with predicates \(\{x<-3\}\) and \(\{x<4\}\) would be abstracted to the BERN program:

\[
\begin{align*}
\{x<-4\} &= \{x<-4\} \land \{x<3\} \land \text{flip}(\theta_1) \\
\{x<3\} &= \{x<-4\} \lor (\{x<3\} \land \text{flip}(\theta_2))
\end{align*}
\]

This strategy is referred to as \textit{independent flip substitution} because it assumes that each \text{flip} is an independent event.

### 3.2.1 Predicate Constraints

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 \(\{x<-3\} \Rightarrow \{x<4\}\). This constraint is an invariant which increases the precision of the abstraction if it is enforced upon execution of each set of predicate assignments. This constraint is denoted \(I\), and is enforced by inserting an \text{observe}(I) statement after each assignment; see Figure 3.1b for an illustration.

### 3.2.1.1 Structural Dependence

This section presents an alternative to independent flip substitution that obviates the need for predicate constraints.
if (x < 0) {
    x = 0
} else {
    x = x + 1
}

(a) A simple concrete program over an integer variable x.

observe (!{x < -3} ∨ {x < 4})
if ({x < -3} ∨ ({x < 4} ∧ flip(θ₀))) {
    {x < -3} = False
    observe (!{x < -3} ∨ {x < 4})
    {x < 4} = True
    observe (!{x < -3} ∨ {x < 4})
} else {
    {x < -3} = {x < -3} ∧ {x < 4} ∧ flip(θ₁)
    observe (!{x < -3} ∨ {x < 4})
    {x < 4} = {x < -3} ∨ ({x < 4} ∧ flip(θ₂))
    observe (!{x < -3} ∨ {x < 4})
}

(b) A probabilistic predicate abstraction of Figure 3.1a with predicates {x<3}, {x<4}.

Consider again the concrete program x = x - 5. As an example, consider an abstraction using the same predicates as before. However, instead of simply substituting each * for a flip, a different abstraction is constructed by conditioning on the previously assigned value:

{x<3} = {x<3} ∨ {x<7} ∨ flip(θ₁)
if ({x<3}) {
    {x<7} = true
} else {
    {x<7} = {x<7} ∨ flip(θ₂)
}

A key point is that, in independent flip substitution, all predicate updates are made simultaneously. Using structural dependence, each predicate is updated sequentially, considering all previous decisions. In this new abstraction, the state {x<3} ∧ !{x<7} is guaranteed to have 0 probability via the structure of the abstraction, thus guaranteeing the invariant $I$ is never violated. Thus, one may sample each flip event independently without risk of generating a 0-probability state which must be ignored, rendering these events independent. In general, one may always construct an abstraction which structurally disallows invalid states.
CHAPTER 4
Inference & Learning

A probabilistic predicate abstraction encodes a probability distribution over program executions $\Pr_A$. This chapter briefly explores the problem of inference and learning in probabilistic predicate abstractions.

This chapter (1) shows that inference is a generalization of model checking, a technique from the programming languages community for determining the set of reachable states in a Boolean program, and (2) explores challenges in fitting a probabilistic predicate abstraction to observed executions of a probabilistic concrete program.

4.1 Inference

Inference in a probabilistic predicate abstraction is the problem of computing $\Pr_A(a_0 \mid a_i)$ for a particular choice of flip parameters $\theta$. One option is to treat this problem as a classic probabilistic program inference problem and rely on an existing probabilistic program inference algorithm, such as the one found in Stan (Carpenter et al., 2016), Church (Goodman et al., 2008), or Anglican (Wood, Meent, and Mansinghka, 2014). However, the probabilistic programs which arise out of probabilistic predicate abstractions are guaranteed to be exclusively over Boolean variables; one can make many optimizations with this knowledge that these more general probabilistic programming languages can not.

Existing techniques from the programming languages literature which are designed for working with Boolean programs may be extended to perform inference on BERN by using weighted model counting to evaluate queries. Note that a probabilistic program abstraction allows one to query the marginal probability of an event at any point in the program, not merely upon program termination.

4.1.1 Model Checking

The problem of computing the set of reachable states program is known as the model checking problem, and has been extensively studied in the context of Boolean programs (Ball and Rajamani, 2000). Commonly one represents the set of reachable states at any point in the program as some Boolean knowledge-base $\Delta$. In many existing model checking tools, $\Delta$ is represented as a binary decision digram. Inference in BERN is thus a simple extension to the traditional model checking paradigm in which weighted variables are introduced for each flip, with the remaining variables being unweighted.

As in the previous sections, model checking a program decomposes into a problem of
model checking if-statements and model checking assignment statements. First assignment statements are presented.

As an example for how to update a set of reachable states $\Delta$ after an assignment, consider the probabilistic predicate abstraction statement $\{x<4\} = \{x<4\} \land \text{flip}(\theta)$. Assume $\Delta = \{x<4\}$ prior to execution of statement. Following this statement, $\Delta' = (\{x<4\} \land \text{flip}(\theta)) \lor (!\{x<4\} \land \text{flip}(\theta))$. In general, for an assignment statement of the form $\gamma = \alpha$, a new $\Delta'$ is computed from $\Delta$ according to the following update using a fresh variable $\gamma'$:

$$\Delta' = [\gamma' \mapsto \gamma](\exists \gamma.(\Delta \land (\gamma' \iff \alpha)))$$

This update (1) generates a fresh variable $\gamma'$; (2) assigns $\gamma'$ equal to the updated value for $\gamma$; (3) quantifies out $\gamma$; (4) relabels $\gamma'$ to $\gamma$.

In order to model check if-statements, two new $\Delta$ states are generated, one for the then branch and the other for the else branch. Joining together two separate control flows is the disjunction of the set of possible states from each incoming control flow path (Ball and Rajamani, 2000).

### 4.1.2 Weighted Model Counting

Whereas model checking is usually concerned with determining whether or not $\mathcal{A}$ can or cannot reach a particular state, in probabilistic program inference one is concerned with the weighted sum of reachable states, where the weights are induced by the parameters of the flips in each model.

It was shown by Valiant (1979) that the problem of counting the number of satisfying solutions to a Boolean system is $\#P$-hard. The problem of weighted model counting is a simple generalization of this problem, and is thus by reduction also $\#P$-hard. The problem of model counting and weighted model counting remains a topic of interest due to its wide applicability in statistics, machine learning, operations research, and many other fields of study (Gomes, Sabharwal, and Selman, 2008).

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 efficient weighted model counting (Darwiche and Marquis, 2001; Chavira and Darwiche, 2008). Knowledge compilation and weighted model counting have been applied to the problem of probabilistic program inference in prior work (Fierens et al., 2015). The satisfiability approach to model checking can be extended to perform weighted model counting. A number of recent approximation methods have been explored (Chakraborty, Meel, and Vardi, 2013; Zhao et al., 2016); see Gomes, Sabharwal, and Selman (2008) for a survey of the subject.

As an example of weighted model counting for probabilistic program inference, the example from the previous section is continued. In order to compute $\Pr_{\mathcal{A}}(\{x < 4\}_o \mid \{x < 4\}_i)$,
compute a weighted model count of $\Delta' \land \{x < 4\}$, which in this case is equal to $\theta$.

4.2 Learning

Next the problem of learning the parameters of a probabilistic program abstraction is explored. Here, the concrete program $C$ is augmented with a distribution on its input, yielding the concrete program distribution $\Pr[C](z_o)$. The goal is for the learned probabilistic abstraction $A$ to approximate this distribution.

In the case when the invariants $I$ are not enforced through observe statements, the flip parameters are independent and correspond to conditional probabilities. Hence, simple counting derives maximum-likelihood estimates for each parameter of the abstraction, similar to Bayesian network parameter learning.

4.2.1 Branch Probabilities

Consider a concrete branch statement of the form $\text{if}(\gamma)$, where $\gamma$ is some concrete expression. This statement is abstracted to an if-statement of the form $\text{if}(\neg \beta \lor (\alpha \land \text{flip}(\theta)))$, where $\alpha$ and $\beta$ are abstract expressions. The parameter $\theta$ is learned by sampling executions of the concrete program. The maximum likelihood parameter for $\theta$ is the expected number of times that the branch is taken in the concrete program when $\alpha \land \beta$.

4.2.2 Assignment Probabilities

Consider an assignment $v = \varphi$, for some concrete expression $\varphi$. Its probabilistic predicate abstraction is a set of $i$ simultaneous updates to predicates of the form $\gamma_i = \alpha_i \lor (\neg \beta_i \land \text{flip}(\theta_i))$. If each of these updates is independent, then one can learn the parameters for each $\theta$ independently using a technique identical to that in the previous section. In the case when these events are constrained by an observe statement, the parameters learned by counting are not the maximum likelihood parameters, and need to be estimated using optimization techniques.

In general, the distribution over events in the concrete program and learned abstraction will not match exactly, especially when also enforcing invariants $I$. Consider the concrete program $x = x - 5$ and its abstraction generated using the predicates $\{x<3\}$ and $\{x<7\}$:

$$
\begin{align*}
\{x<3\} &= \{x<3\} \lor \{x<7\} \lor \text{flip}(\theta_1) \\
\{x<7\} &= \{x<3\} \lor \{x<7\} \lor \text{flip}(\theta_2)
\end{align*}
$$

If one assumes that, prior to the assignment, $x \sim \text{Unif}[0, 15]$, then by counting one would learn $\theta_1 = \Pr(x \in [0,7] \mid x \geq 7) = 1/8$ and $\theta_2 = \Pr(x \in [0,11] \mid x \geq 7) = 5/8$. These parameters induce the following joint distribution over the predicates after assignment:
<table>
<thead>
<tr>
<th>( {x &lt; 3} )</th>
<th>( {x &lt; 7} )</th>
<th>( \Pr(\cdot \mid x \geq 7) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>((1 - \theta_1)(1 - \theta_2) = \frac{21}{64})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>((1 - \theta_1)\theta_2 = \frac{35}{64})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(\theta_1(1 - \theta_2) = \frac{3}{64})</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(\theta_1\theta_2 = \frac{5}{64})</td>
</tr>
</tbody>
</table>

Observe that the third row of this table violates the constraint \( \mathcal{I} = \{x < 3\} \Rightarrow \{x < 7\} \). Thus, the joint distribution over predicates in the concrete program is different from the distribution which is learned by counting.

### 4.2.3 Marginal Matching

Thus far it has been shown that (i) concretization invariance allows one to reason about concrete states in \( \Pr_{[A]} \) by only looking at the abstract semantics \( \Pr_A \) and (ii) one can generate an abstraction whose independent parameters permit efficient maximum-likelihood learning. This section shows how to evaluate \( \Pr_{[C]} \) by performing a query on a learned probabilistic predicate abstraction \( \Pr_A \). First, an example is presented in which this connection is made clear.

For a given concrete query \( \Pr_{[C]}(\varphi) \), \( \varphi \) is included as the sole predicate in a trivial abstraction in which every branch guard is abstracted to a single \( \text{flip} \) and every assignment statement is abstracted to an empty statement. On the final line of the abstraction, \( \varphi \) is assigned \( \varphi = \text{flip}(\theta) \). It is clear that all flips are independent and the query \( \Pr_A(\varphi) = \theta \). According to the learning algorithm presented in the previous sections, the estimation of \( \theta \) and subsequent inference on \( \Pr_A \) is exactly performing direct sampling on \( C \).

In this example, the abstraction captures the necessary marginal probabilities in order to compute the desired query, since the only marginal exactly corresponds with the query. This notion may be generalized to any abstraction with independent flips (i.e., not subject to constraints).

**Proposition 4.2.1. Independent marginal matching.** Let \((\mathcal{A}, \alpha, \Pr_{[\gamma]})\) be a probabilistic predicate abstraction with (i) strongly compatible \( \Pr_{[\gamma]} \) and \( \alpha \) and (ii) independent parameters learned from a probabilistic concrete program \( \Pr_{[C]} \). Then for any \( a_o \in \mathcal{D}_A \),

\[
\sum_{z_o \in \gamma(a_o)} \Pr_{[\gamma]}(z_o) = \Pr_A(a_o). \tag{4.1}
\]

Consequently, if \( \mathcal{A} \) is a probabilistic predicate abstraction, then it can be used to compute queries about \( C \) involving the predicates from which \( \mathcal{A} \) is constructed.

As an intuition for why this proposition holds, consider the two conditions. Condition (i) is necessary for computing \( \Pr_{[A]} \) solely from queries performed on \( \Pr_A \); see concretization invariance. Condition (ii) is necessary in order to guarantee that each parameter of the abstraction’s maximum likelihood estimates are learned by sampling \( C \). The key is that each of these parameter estimates allows one to compute locally accurate queries: the probability of some event \( \varphi \) is accurately captured at each line of the abstraction.
CHAPTER 5

Conclusion & Future Work

The artificial intelligence and programming languages community have been working towards unifying their approaches to analyzing and reasoning about probabilistic programs. This thesis presents an initial attempt towards unifying the methodologies of these two fields, and provides a common theoretical and notational framework to bridge this gap. Probabilistic predicate abstractions – and probabilistic program abstractions in general – are currently unexplored territory for aiding in the analysis of probabilistic programs. There remain a number of ways of extending this work with future applications and improvements.

Learning the parameters of probabilistic predicate abstractions are limited by strict dependency assumptions, and currently can not handle concrete programs with loops. Non-deterministic predicate abstractions naturally generalize to concrete program with loops via an elegant fixpoint semantics (Ball, Majumdar, et al., 2001). Probabilistic predicate abstractions could be generalized to this setting as well to statically handle loops without unrolling a finite number of times, which is a capability no existing static probabilistic program analysis system possesses.

The parameter learning method presented here is brief and high-level. It involves directly sampling of the concrete program and does not utilize the structure represented by the abstraction or alternative methods for parameter inference. Much future work remains to be done on more efficient mechanisms for estimating the parameters of the abstraction.

Ultimately probabilistic predicate abstractions provide promising future avenues for designing new inference and learning procedures for complex deterministic and non-deterministic concrete programs. Abstractions provide a mechanism for decomposing a single large query on a concrete program into many smaller queries targeted at estimating the parameters of the abstraction. Ultimately, abstractions are necessary for reasoning about complex systems: by generalizing predicate abstractions to the probabilistic context, a new tool may be developed for reasoning about probabilistic systems.


Ritchie, Daniel (2014). “Quicksand: A Lightweight Implementation of Probabilistic Programming for Procedural Modeling and Design”. In:


Vlasselaer, Jonas et al. (2015). “Compiling Probabilistic Logic Programs into Sentential Decision Diagrams”. In:
